Phalcon PHP Framework Documentation

Release 1.0.0

Phalcon Team

October 24, 2013

Contents

Welcome to Phalcon framework, a new approach on PHP frameworks. Our mission is to give you an advanced tool for developing web sites and applications without worrying about performance.

CHAPTER 1

What is Phalcon?

Phalcon is an open source, full stack framework for PHP 5 written as a C-extension, optimized for high performance. You don't need learn or use the C language, since the functionality is exposed as PHP classes ready for you to use. Phalcon also is loosely coupled, allowing you to use its objects as glue components based on the needs of your application.

Phalcon is not only about performance, our goal is to make it robust, rich in features and easy to use!

Table of Contents

2.1 Our motivation

There are many PHP frameworks nowadays, but none of them is like Phalcon (Really, trust us on this one).

Almost all programmers prefer to use a framework. This is primarily because it provides a lot of functionality that is already tested and ready to use, therefore keeping code DRY (Don't Repeat Yourself). However, the framework itself demands a lot of file inclusions and hundreds of lines of code to be interpreted and executed on each request from the actual application. This operation slows the application down and subsequently impacts the end user experience.

2.1.1 The Question

Why can't we have a framework with all of its advantages but with none or very few disadvantages?

This is why Phalcon was born!

During the last few months, we have extensively researched PHP's behavior, investigating areas for significant optimizations (big or small). Through understanding of the Zend Engine, we managed to remove unnecessary validations, compacted code, performed optimizations and generated low-level solutions so as to achieve maximum performance from Phalcon.

2.1.2 Why?

- The use of frameworks has become mandatory in professional development with PHP
- Frameworks offer a structured philosophy to easily maintain projects writing less code and making work more fun

2.1.3 Inner workings of PHP?

- PHP has dynamic and weak variable types. Every time a binary operation is made (ex. 2 + "2"), PHP checks the operand types to perform potential conversions
- PHP is interpreted and not compiled. The major disadvantage is performance loss
- Every time a script is requested it must be first interpreted.

- If a bytecode cache (like APC) isn't used, syntax checking is performed every time for every file in the request

2.1.4 How traditional PHP frameworks work?

- Many files with classes and functions are read on every request made. Disk reading is expensive in terms of performance, especially when the file structure includes deep folders
- Modern frameworks use lazy loading (autoload) to increase performance (for load and execute only the code needed)
- Continuous loading or interpreting is expensive and impacts performance
- The framework code does not change very often, therefore an application needs to load and interpret it every time a request is made

2.1.5 How does a PHP C-extension work?

- C extensions are loaded together with PHP one time on the web server's daemon start process
- Classes and functions provided by the extension are ready to use for any application
- The code isn't interpreted because is already compiled to a specific platform and processor

2.1.6 How does Phalcon work?

- Components are loosely coupled. With Phalcon, nothing is imposed on you: you're free to use the full framework, or just some parts of it as a glue components.
- Low-level optimizations provides the lowest overhead for MVC-based applications
- Interact with databases with maximum performance by using a C-language ORM for PHP
- Phalcon directly accesses internal PHP structures optimizing execution in that way as well

2.1.7 Conclusion

Phalcon is an effort to build the fastest framework for PHP. You now have an even easier and robust way to develop applications without be worrying about performance. Enjoy!

2.2 Framework Benchmarks

In the past, performance was not considered one of the top priorities when developing web applications. Reasonable hardware was able to compensate for that. However when Google decided to take site speed into account in the search rankings, performance became one of the top priorities alongside functionality. This is yet another way in which improving web performance will have a positive impact on a website.

The benchmarks below, show how efficient Phalcon is when compared with other traditional PHP frameworks. These benchmarks are updated as stable versions are released from any of the frameworks mentioned or Phalcon itself.

We encourage programmers to clone the test suite that we are using for our benchmarks. If you have any additional optimizations or comments please write us. Check out source at Github

2.2.1 What was the test environment?

APC intermediate code cache was enabled for all frameworks. Any Apache mod-rewrite feature was disabled when possible to avoid potentially additional overheads.

The testing hardware environment is as follows:

- Operating System: Mac OS X Lion 10.7.4
- Web Server: Apache httpd 2.2.22
- PHP: 5.3.15
- CPU: 2.04 Ghz Intel Core i5
- Main Memory: 4GB 1333 MHz DDR3
- Hard Drive: 500GB SATA Disk

PHP version and info:

APC settings:

2.2.2 List of Benchmarks

Hello World Benchmark

How the benchmarks were performed?

We created a "Hello World" benchmark seeking to identify the smallest load overhead of each framework. Many people don't like this kind of benchmark because real-world applications require more complex features or structures. However, these tests identify the minimum time spent by each framework to perform a simple task. Such a task represents the mimimum requirement for every framework to process a single request.

More specifically, the benchmark only measures the time it takes for a framework to start, run an action and free up resources at the end of the request. Any PHP application based on an MVC architecture will require this time. Due to the simplicity of the benchmark, we ensure that the time needed for a more complex request will be higher.

A controller and a view have been created for each framework. The controller "say" and action "hello". The action only sends data to the view which displays it ("Hello!"). Using the "ab" benchmark tool we sent 2000 requests using 10 concurrent connections to each framework.

What measurements were recorded?

These were the measurements we record to identify the overall performance of each framework:

- Requests per second
- Time across all concurrent requests
- Number of included PHP files on a single request (measured using function get_included_files.
- Memory Usage per request (measured using function memory_get_usage.

Pariticipant Frameworks

- Yii (YII_DEBUG=false) (yii-1.1.13)
- Symfony (2.0.11)

PHP Version 5.3.15

System	Darwin Andress-MacBook-Pro.local 11.4.0 Darwin Kernel Version 11.4.0: Mon Apr 9 19:32:15 PDT 2012; root:xnu-1699.26.8~1/RELEASE_X86_64 x86_64
Build Date	Aug 15 2012 16:31:00
Configure Command	'./configure' '--prefix=/opt/local' '--mandir=/opt/local/share/man' '--infodir=/opt/local/share/info' '--program-suffix=53' '--includedir=/opt/local/include/php53' '--libdir=/opt/local/lib/php53' '--with-config-file-path=/opt/local/etc/php53' '--with-config-file-scan-dir=/opt/local/var/db/php53' '--disable-all' '--enable-bcmath' '--enable-ctype' '--enable-dom' '--enable-fileinfo' '--enable-filter' '--enable-hash' '--enable-json' '--enable-libxml' '--enable-pdo' '--enable-phar' '--enable-session' '--enable-simplexml' '--enable-tokenizer' '--enable-xml' '--enable-xmlreader' '--enable-xmlwriter' '--with-bz2=/opt/local' '--with-mhash=/opt/local' '--with-pcre-regex=/opt/local' '--with-libxml-dir=/opt/local' '--with-zlib=/opt/local' '--without-pear' '--disable-cgi' '--disable-cli' '--disable-fpm' '--with-apxs2=/opt/local/apache2/bin/apxs'
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/opt/local/etc/php53
Loaded Configuration File	/opt/local/etc/php53/php.ini
Scan this dir for additional .ini files	/opt/local/var/db/php53
Additional .ini files parsed	/opt/local/var/db/php53/APC.ini, /opt/local/var/db/php53/http.ini, /opt/local/var/db/php53/mysql.ini, /opt/local/var/db/php53/pear.ini
PHP API	20090626
PHP Extension	20090626
Zend Extension	220090626
Zend Extension Build	API220090626,NTS
PHP Extension Build	API20090626,NTS
Debug Build	no
Thread Safety	disabled
Zend Memory Manager	enabled
Zend Multibyte	disabled

apc

APC Support	enabled
Version	3.1.11
APC Debugging	Disabled
MMAP Support	Enabled
MMAP File Mask	*no value*
Locking type	spin Locks
Serialization Support	php
Revision	$Revision: 325875 $
Build Date	Jul 20 2012 11:18:52

Directive	Local Value	Master Value
apc.cache_by_default	On	On
apc.canonicalize	On	On
apc.coredump_unmap	Off	Off
apc.enable_cli	Off	Off
apc.enabled	On	On
apc.file_md5	Off	Off
apc.file_update_protection	2	2
apc.filters	*no value*	*no value*
apc.gc_ttl	3600	3600
apc.include_once_override	Off	Off
apc.lazy_classes	Off	Off
apc.lazy_functions	Off	Off
apc.max_file_size	1M	1M
apc.mmap_file_mask	*no value*	*no value*
apc.num_files_hint	1000	1000
apc.preload_path	*no value*	*no value*
apc.report_autofilter	Off	Off
apc.rfc1867	Off	Off
apc.rfc1867_freq	0	0
apc.rfc1867_name	APC_UPLOAD_PROGRESS	APC_UPLOAD_PROGRESS
apc.rfc1867_prefix	upload_	upload_
apc.rfc1867_ttl	3600	3600
apc.serializer	default	default
apc.shm_segments	1	1
apc.shm_size	32M	32M
apc.slam_defense	On	On
apc.stat	Off	Off
apc.stat_ctime	Off	Off
apc.ttl	0	0
apc.use_request_time	On	On

- Zend Framework (1.11.11)
- Kohana (3.2.0)
- FuelPHP (1.2.1)
- CakePHP (2.1.3)
- Laravel 3.2.5
- CodeIgniter (2.1.0)
- Nette (2.0.4)

Results

Yii (YII_DEBUG=false) Version yii-1.1.13

```
# ab -n 2000 -c 10 http://localhost/bench/helloworld/yii/index.php?r=say/hello
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/helloworld/yii/index.php?r=say/hello
Document Length:        61 bytes

Concurrency Level:      10
Time taken for tests:   2.081 seconds
Complete requests:      2000
Failed requests:        0
Write errors:           0
Total transferred:      508000 bytes
HTML transferred:       122000 bytes
Requests per second:    961.28 [#/sec] (mean)
Time per request:       10.403 [ms] (mean)
Time per request:       1.040 [ms] (mean, across all concurrent requests)
Transfer rate:          238.44 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   10   4.3      9      42
Processing:     0    0   1.0      0      24
Waiting:        0    0   0.8      0      17
Total:          3   10   4.3      9      42

Percentage of the requests served within a certain time (ms)
  50%      9
  66%     11
  75%     13
  80%     14
  90%     15
  95%     17
  98%     21
```

```
  99%     26
 100%     42 (longest request)
```

Symfony Version 2.1.6

```
# ab -n 2000 -c 10 http://localhost/bench/Symfony/web/app.php/say/hello/
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/Symfony/web/app.php/say/hello/
Document Length:        16 bytes

Concurrency Level:      5
Time taken for tests:   1.848 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      249000 bytes
HTML transferred:       16000 bytes
Requests per second:    541.01 [#/sec] (mean)
Time per request:       9.242 [ms] (mean)
Time per request:       1.848 [ms] (mean, across all concurrent requests)
Transfer rate:          131.55 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    9   4.8      8      61
Processing:     0    0   0.6      0      15
Waiting:        0    0   0.6      0      15
Total:          4    9   4.8      8      61

Percentage of the requests served within a certain time (ms)
  50%      8
  66%      9
  75%     11
  80%     12
  90%     15
  95%     18
  98%     22
  99%     30
 100%     61 (longest request)
```

CodeIgniter 2.1.0

```
# ab -n 2000 -c 10 http://localhost/bench/codeigniter/index.php/say/hello
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
```

```
Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/helloworld/codeigniter/index.php/say/hello
Document Length:        16 bytes

Concurrency Level:      10
Time taken for tests:   1.888 seconds
Complete requests:      2000
Failed requests:        0
Write errors:           0
Total transferred:      418000 bytes
HTML transferred:       32000 bytes
Requests per second:    1059.05 [#/sec] (mean)
Time per request:       9.442 [ms] (mean)
Time per request:       0.944 [ms] (mean, across all concurrent requests)
Transfer rate:          216.15 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    9   4.1      9      33
Processing:     0    0   0.8      0      19
Waiting:        0    0   0.7      0      16
Total:          3    9   4.2      9      33

Percentage of the requests served within a certain time (ms)
  50%      9
  66%     10
  75%     11
  80%     12
  90%     14
  95%     16
  98%     21
  99%     24
 100%     33 (longest request)
```

Kohana 3.2.0

```
# ab -n 2000 -c 10 http://localhost/bench/helloworld/kohana/index.php/say/hello
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/helloworld/kohana/index.php/say/hello
Document Length:        15 bytes

Concurrency Level:      10
```

```
Time taken for tests:   2.324 seconds
Complete requests:      2000
Failed requests:        0
Write errors:           0
Total transferred:      446446 bytes
HTML transferred:       30030 bytes
Requests per second:    860.59 [#/sec] (mean)
Time per request:       11.620 [ms] (mean)
Time per request:       1.162 [ms] (mean, across all concurrent requests)
Transfer rate:          187.60 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   11   5.1     10      64
Processing:     0    0   1.9      0      39
Waiting:        0    0   1.4      0      35
Total:          3   11   5.3     11      64

Percentage of the requests served within a certain time (ms)
  50%     11
  66%     13
  75%     15
  80%     15
  90%     17
  95%     18
  98%     24
  99%     31
 100%     64 (longest request)
```

Fuel 1.2.1

```
# ab -n 2000 -c 10 http://localhost/bench/helloworld/fuel/public/say/hello
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/helloworld/fuel/public/say/hello
Document Length:        16 bytes

Concurrency Level:      10
Time taken for tests:   2.742 seconds
Complete requests:      2000
Failed requests:        0
Write errors:           0
Total transferred:      418000 bytes
HTML transferred:       32000 bytes
Requests per second:    729.42 [#/sec] (mean)
Time per request:       13.709 [ms] (mean)
Time per request:       1.371 [ms] (mean, across all concurrent requests)
Transfer rate:          148.88 [Kbytes/sec] received
```

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   13   6.0     12      79
Processing:     0    0   1.3      0      22
Waiting:        0    0   0.8      0      21
Total:          4   14   6.1     13      80

Percentage of the requests served within a certain time (ms)
  50%     13
  66%     15
  75%     17
  80%     17
  90%     19
  95%     24
  98%     30
  99%     38
 100%     80 (longest request)
```

Cake 2.1.3

```
# ab -n 10 -c 5 http://localhost/bench/cake/say/hello
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient).....done


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/cake/say/hello
Document Length:        16 bytes

Concurrency Level:      5
Time taken for tests:   30.051 seconds
Complete requests:      10
Failed requests:        0
Write errors:           0
Total transferred:      1680 bytes
HTML transferred:       160 bytes
Requests per second:    0.33 [#/sec] (mean)
Time per request:       15025.635 [ms] (mean)
Time per request:       3005.127 [ms] (mean, across all concurrent requests)
Transfer rate:          0.05 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    2   3.6      0      11
Processing: 15009 15020   9.8  15019   15040
Waiting:        9   21   7.9     25      33
Total:      15009 15022   8.9  15021   15040

Percentage of the requests served within a certain time (ms)
  50%  15021
  66%  15024
  75%  15024
```

```
  80%  15032
  90%  15040
  95%  15040
  98%  15040
  99%  15040
 100%  15040 (longest request)
```

Zend Framework 1.11.11

```
# ab -n 2000 -c 10 http://localhost/bench/helloworld/zendfw/public/index.php
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/helloworld/zendfw/public/index.php
Document Length:        16 bytes

Concurrency Level:      10
Time taken for tests:   5.641 seconds
Complete requests:      2000
Failed requests:        0
Write errors:           0
Total transferred:      418000 bytes
HTML transferred:       32000 bytes
Requests per second:    354.55 [#/sec] (mean)
Time per request:       28.205 [ms] (mean)
Time per request:       2.820 [ms] (mean, across all concurrent requests)
Transfer rate:          72.36 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   27   9.6     25      89
Processing:     0    1   3.0      0      70
Waiting:        0    0   2.9      0      70
Total:          9   28   9.6     26      90

Percentage of the requests served within a certain time (ms)
  50%     26
  66%     28
  75%     32
  80%     34
  90%     41
  95%     46
  98%     55
  99%     62
 100%     90 (longest request)
```

Laravel 3.2.5

```
# ab -n 2000 -c 10 http://localhost/bench/helloworld/laravel/public/say/hello

This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/helloworld/laravel/public/say/hello
Document Length:        15 bytes

Concurrency Level:      10
Time taken for tests:   4.090 seconds
Complete requests:      2000
Failed requests:        0
Write errors:           0
Total transferred:      1665162 bytes
HTML transferred:       30045 bytes
Requests per second:    489.03 [#/sec] (mean)
Time per request:       20.449 [ms] (mean)
Time per request:       2.045 [ms] (mean, across all concurrent requests)
Transfer rate:          397.61 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   20   7.6     19      92
Processing:     0    0   2.5      0      53
Waiting:        0    0   2.5      0      53
Total:          6   20   7.6     19      93

Percentage of the requests served within a certain time (ms)
  50%     19
  66%     21
  75%     23
  80%     24
  90%     29
  95%     34
  98%     42
  99%     48
 100%     93 (longest request)
```

Nette 2.0.4

```
# ab -n 2000 -c 10 http://localhost/bench/helloworld/nette/www/index.php

This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
```

```
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/helloworld/nette/www/index.php
Document Length:        24963 bytes

Concurrency Level:      10
Time taken for tests:   7.750 seconds
Complete requests:      2000
Failed requests:        200
   (Connect: 0, Receive: 0, Length: 200, Exceptions: 0)
Write errors:           0
Total transferred:      50370200 bytes
HTML transferred:       49926200 bytes
Requests per second:    258.07 [#/sec] (mean)
Time per request:       38.749 [ms] (mean)
Time per request:       3.875 [ms] (mean, across all concurrent requests)
Transfer rate:          6347.24 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   38  13.1     34     115
Processing:     0    1   4.7      0      99
Waiting:        0    0   4.5      0      98
Total:         15   39  13.2     34     116

Percentage of the requests served within a certain time (ms)
  50%     34
  66%     38
  75%     46
  80%     50
  90%     58
  95%     64
  98%     75
  99%     82
 100%    116 (longest request)
```

Phalcon Version 0.8.0

```
# ab -n 2000 -c 10 http://localhost/bench/helloworld/phalcon/index.php?_url=/say/hello
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/helloworld/phalcon/index.php?_url=/say/hello
Document Length:        16 bytes

Concurrency Level:      10
Time taken for tests:   0.789 seconds
Complete requests:      2000
Failed requests:        0
```

```
Write errors:           0
Total transferred:      418000 bytes
HTML transferred:       32000 bytes
Requests per second:    2535.82 [#/sec] (mean)
Time per request:       3.943 [ms] (mean)
Time per request:       0.394 [ms] (mean, across all concurrent requests)
Transfer rate:          517.56 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    4   1.7      3      23
Processing:     0    0   0.2      0       6
Waiting:        0    0   0.2      0       6
Total:          2    4   1.7      3      23

Percentage of the requests served within a certain time (ms)
  50%      3
  66%      4
  75%      4
  80%      4
  90%      5
  95%      6
  98%      8
  99%     14
 100%     23 (longest request)
```

Graphs The first graph shows how many requests per second each framework was able to accept. The second shows the average time across all concurrent requests.

Conclusion

The compiled nature of Phalcon offers extraordinary performance that outperforms all other frameworks measured in these benchmarks.

Micro Benchmark

How the benchmarks were performed?

We created a "Hello World" benchmark seeking to identify the smallest load overhead of each framework. Similar to the benchmark made with Frameworks.

Using a route for the HTTP method 'GET' we pass a parameter to a handler returning a "Hello $name" response.

What measurements were recorded?

These were the measurements we record to identify the overall performance of each framework:

- Requests per second
- Time across all concurrent requests
- Number of included PHP files on a single request (measured using function get_included_files.
- Memory Usage per request (measured using function memory_get_usage.

Pariticipant Frameworks

- Slim
- Silex

Results

Slim Framework

```
# ab -n 1000 -c 5 http://localhost/bench/micro/slim/say/hello/Sonny
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/micro/slim/say/hello/Sonny
Document Length:        13 bytes

Concurrency Level:      5
Time taken for tests:   0.882 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      206000 bytes
HTML transferred:       13000 bytes
Requests per second:    1134.21 [#/sec] (mean)
Time per request:       4.408 [ms] (mean)
Time per request:       0.882 [ms] (mean, across all concurrent requests)
Transfer rate:          228.17 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    4   2.4      4      33
Processing:     0    0   0.5      0      11
Waiting:        0    0   0.5      0      11
Total:          2    4   2.4      4      33

Percentage of the requests served within a certain time (ms)
  50%      4
  66%      4
  75%      5
  80%      5
  90%      6
  95%      8
  98%     12
  99%     14
 100%     33 (longest request)
```

Silex

```
# ab -n 1000 -c 5 http://localhost/bench/micro/silex/say/hello/Sonny
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80

Document Path:          /bench/micro/silex/say/hello/Sonny
Document Length:        12 bytes

Concurrency Level:      5
Time taken for tests:   2.228 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      225000 bytes
HTML transferred:       12000 bytes
Requests per second:    448.75 [#/sec] (mean)
Time per request:       11.142 [ms] (mean)
Time per request:       2.228 [ms] (mean, across all concurrent requests)
Transfer rate:          98.60 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   11   5.1     10      44
Processing:     0    0   1.1      0      26
Waiting:        0    0   1.1      0      26
Total:          5   11   5.1     10      45

Percentage of the requests served within a certain time (ms)
  50%     10
  66%     12
  75%     13
  80%     14
  90%     17
  95%     20
  98%     25
  99%     29
 100%     45 (longest request)
```

Phalcon 0.5.0

```
# ab -n 1000 -c 5 http://localhost/bench/micro/phalcon/say/hello/Sonny
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)


Server Software:        Apache/2.2.22
Server Hostname:        localhost
Server Port:            80
```

```
Document Path:          /bench/micro/phalcon/say/hello/Sonny
Document Length:        12 bytes

Concurrency Level:      5
Time taken for tests:   0.397 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      205000 bytes
HTML transferred:       12000 bytes
Requests per second:    2516.74 [#/sec] (mean)
Time per request:       1.987 [ms] (mean)
Time per request:       0.397 [ms] (mean, across all concurrent requests)
Transfer rate:          503.84 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    2   0.9      2      11
Processing:     0    0   0.2      0       5
Waiting:        0    0   0.2      0       4
Total:          1    2   0.9      2      11

Percentage of the requests served within a certain time (ms)
  50%      2
  66%      2
  75%      2
  80%      2
  90%      3
  95%      4
  98%      5
  99%      5
 100%     11 (longest request)
```

Graphs The first graph shows how many requests per second each framework was able to accept. The second shows the average time across all concurrent requests.

Conclusion

The compiled nature of Phalcon offers extraordinary performance that outperforms all other frameworks measured in these benchmarks.

2.2.3 ChangeLog

New in version 1.0: Update Mar-20-2012: Benchmarks redone changing the apc.stat setting to Off. More InfoChanged in version 1.1: Update May-13-2012: Benchmarks redone PHP plain templating engine instead of Twig for Symfony. Configuration settings for Yii were also changed as recommended.Changed in version 1.2: Update May-20-2012: Fuel framework was added to benchmarks.Changed in version 1.3: Update Jun-4-2012: Cake framework was added to benchmarks. It is not however present in the graphics, since it takes 30 seconds to run only 10 of 1000.Changed in version 1.4: Update Ago-27-2012: PHP updated to 5.3.15, APC updated to 3.1.11, Yii updated to 1.1.12, Phalcon updated to 0.5.0, Added Laravel, OS updated to Mac OS X Lion. Hardware upgraded.

2.3 Installation

PHP extensions require a slightly different installation method to a traditional php-based library or framework. You can either download a binary package for the system of your choice or build it from the sources.

During the last few months, we have extensively researched PHP's behavior, investigating areas for significant optimizations (big or small). Through understanding of the Zend Engine, we managed to remove unecessary validations, compacted code, performed optimizations and generated low-level solutions so as to achieve maximum performance from Phalcon.

> Phalcon compiles from PHP 5.3.1, but because of old PHP bugs causing memory leaks, we highly recommend you use at least PHP 5.3.11 or greater.
>
> PHP versions below 5.3.9 have several security flaws and these aren't recommended for production web sites. Learn more

2.3.1 Windows

To use phalcon on Windows you can download a DLL library. Edit your php.ini file and then append at the end:

```
extension=php_phalcon.dll
```

Restart your webserver.

The following screencast is a step-by-step guide to install Phalcon on Windows:

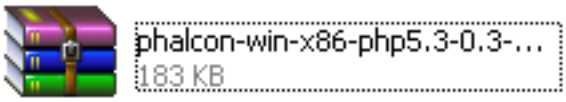
Related Guides

Installation on XAMPP

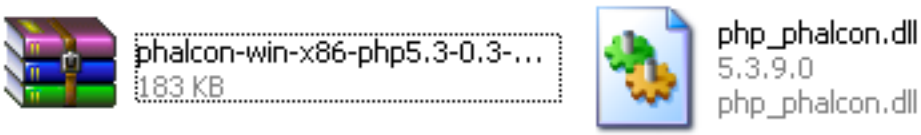
XAMPP is an easy to install Apache distribution containing MySQL, PHP and Perl. Once you download XAMPP, all you have to do is extract it and start using it. Below are detailed instructions on how to install Phalcon on XAMPP for Windows. Using the latest XAMPP version is highly recommended.

Download the right version of Phalcon XAMPP is always releasing 32 bit versions of Apache and PHP. You will need to download the x86 version of Phalcon for Windows from the download section.

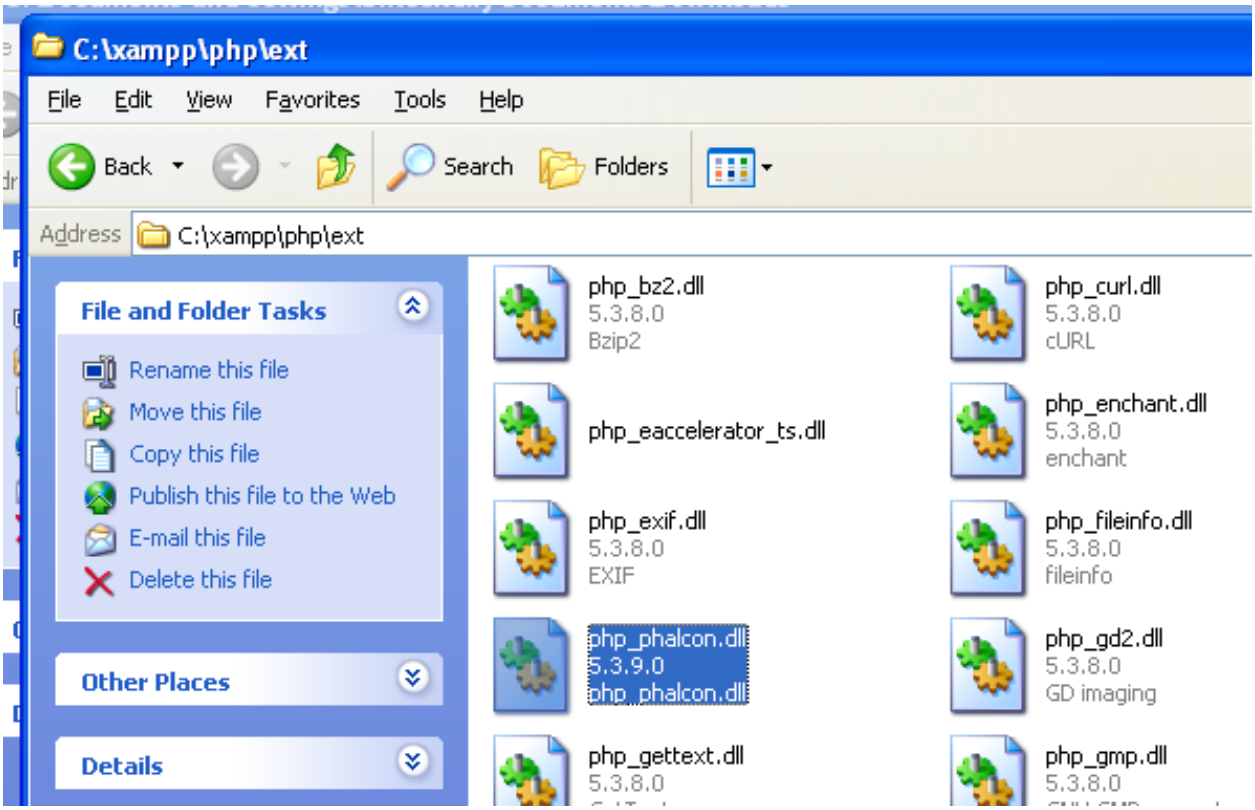
After downloading the Phalcon library you will have a zip file like the one shown below:



Extract the library from the archive to get the Phalcon DLL:



Copy the file php_phalcon.dll to the PHP extensions. If you have installed XAMPP in the c:\xampp folder, the extension needs to be in c:\xampp\php\ext



Edit the php.ini file, it is located at C:\xampp\php\php.ini. It can be edited with Notepad or a similar program. We recommend Notepad++ to avoid issues with line endings. Append at the end of the file: extension=php_phalcon.dll and save it.

Restart the Apache Web Server from the XAMPP Control Center. This will load the new PHP configuration.

Open your browser to navigate to http://localhost. The XAMPP welcome page will appear. Click on the link phpinfo().

phpinfo() will output a significant amount of information on screen about the current state of PHP. Scroll down to check if the phalcon extension has been loaded correctly.

If you can see the phalcon version in the phpinfo() output, congrats!, You are now flying with Phalcon.

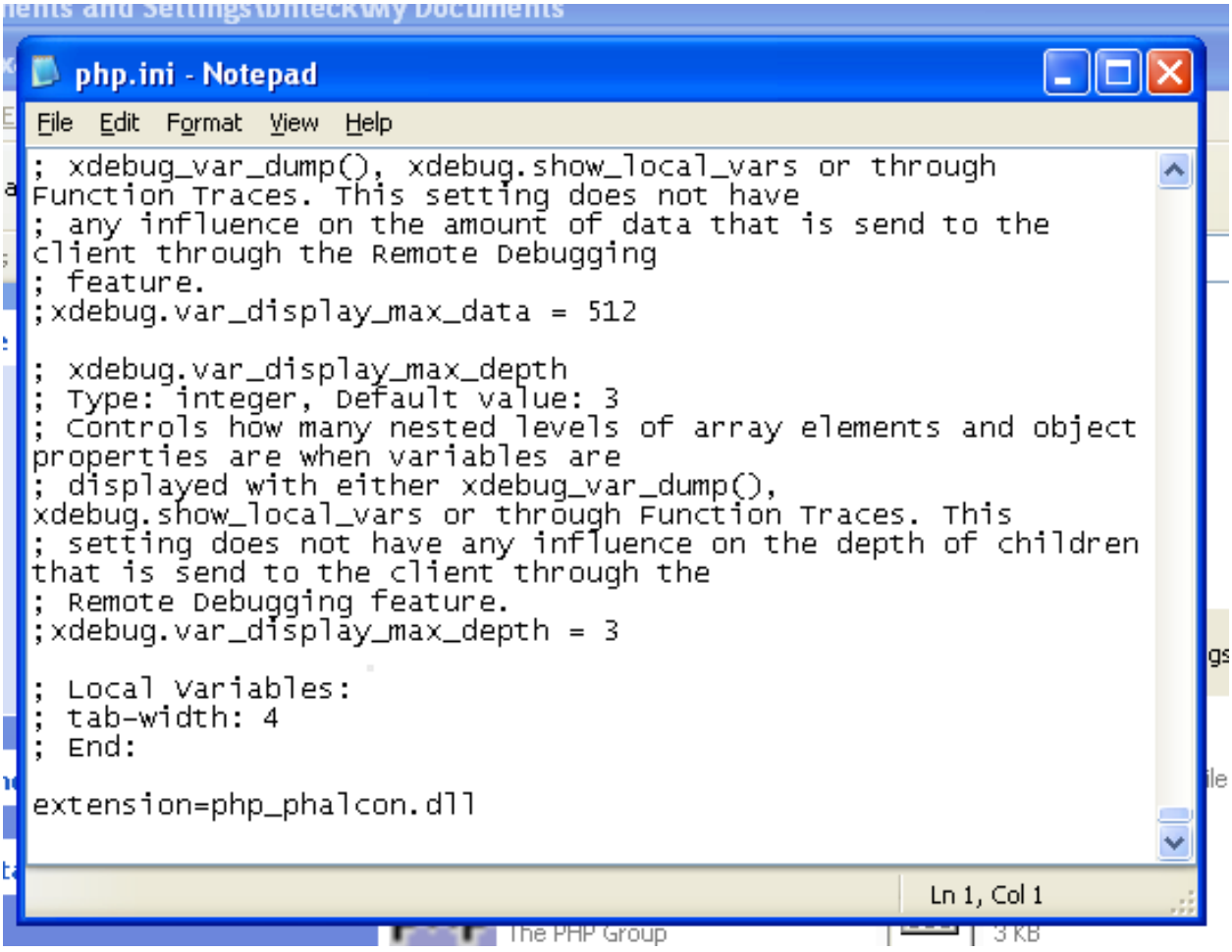
Screencast The following screencast is a step by step guide to install Phalcon on Windows:

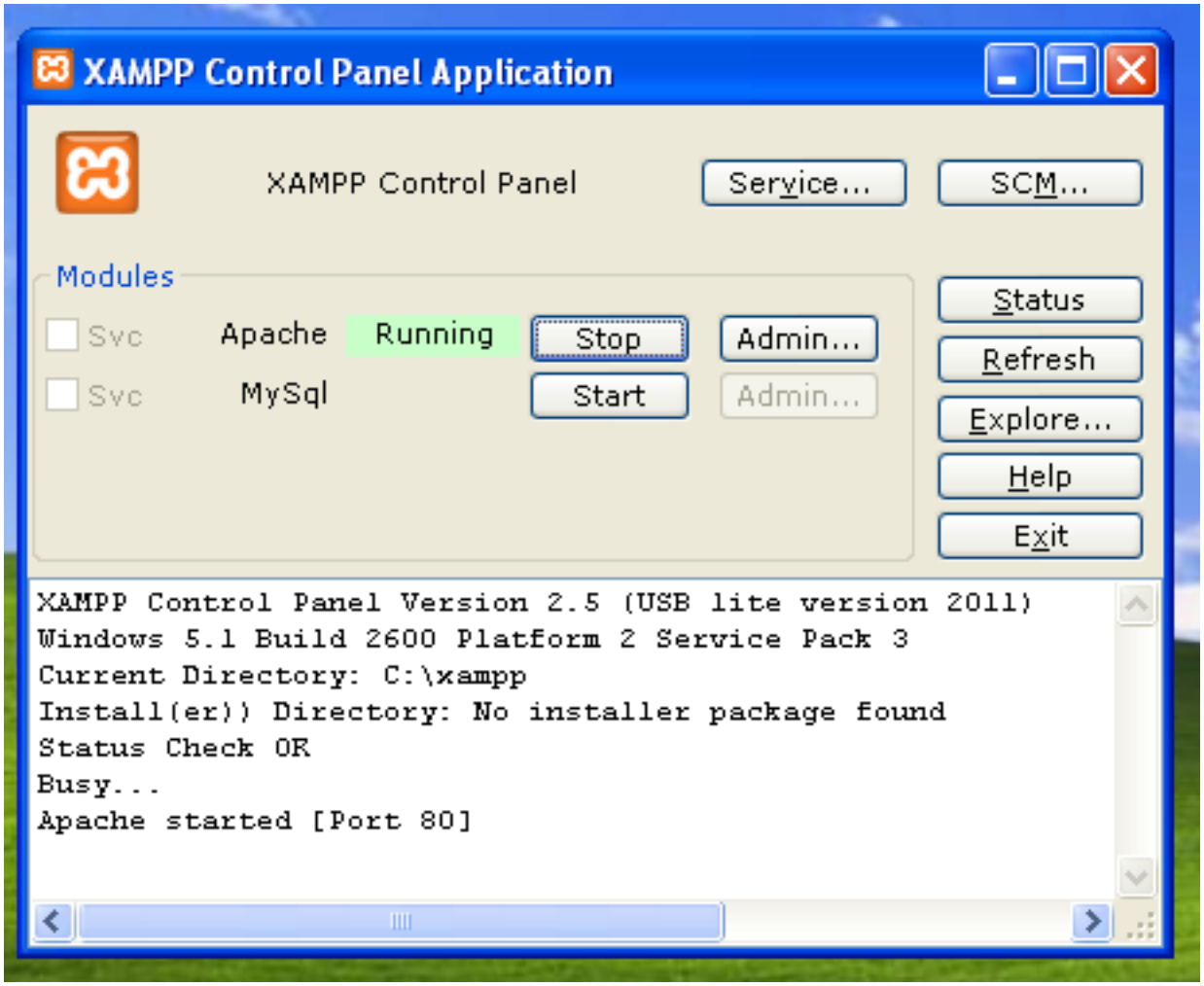
Related Guides

- *General Installation*
- *Detailed Installation on WAMP for Windows*

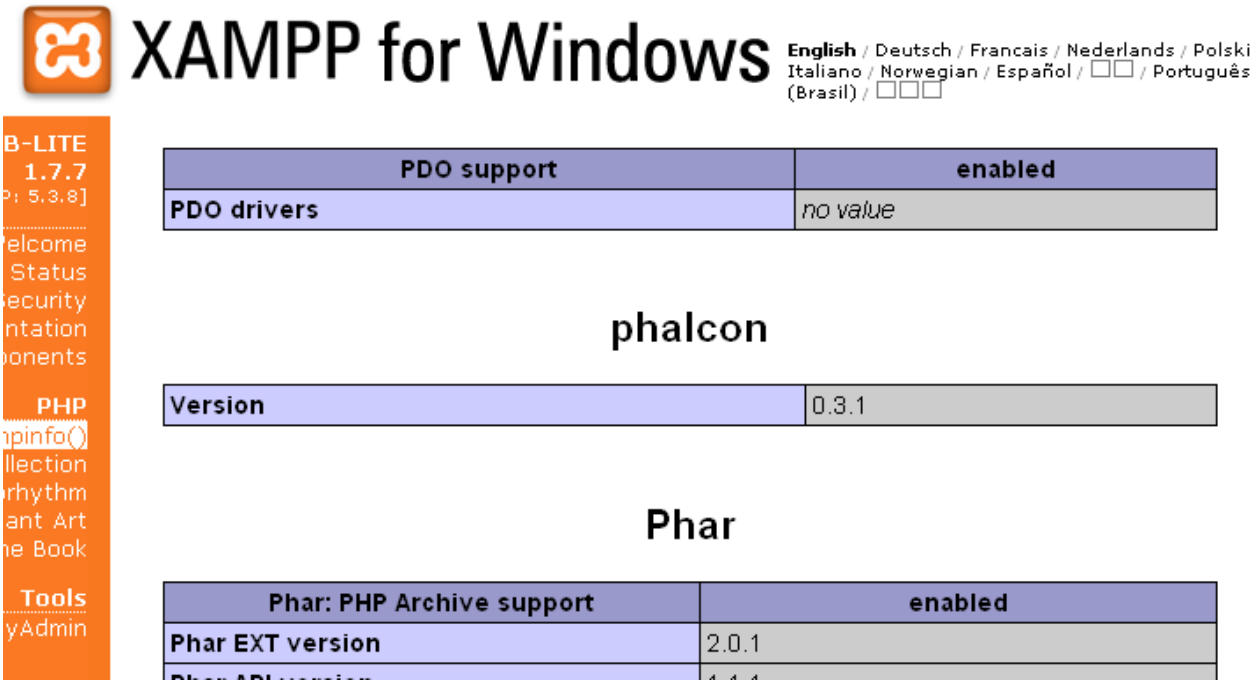
Installation on WAMP

WampServer is a Windows web development environment. It allows you to create web applications with Apache2, PHP and a MySQL database. Below are detailed instructions on how to install Phalcon on WampServer for Windows.


php.ini - Notepad
File Edit Format View Help
; xdebug_var_dump(), xdebug.show_local_vars or through
Function Traces. This setting does not have
; any influence on the amount of data that is send to the
client through the Remote Debugging
; feature.
;xdebug.var_display_max_data = 512
; xdebug.var_display_max_depth
; Type: integer, Default value: 3
; Controls how many nested levels of array elements and object
properties are when variables are
; displayed with either xdebug_var_dump(),
xdebug.show_local_vars or through Function Traces. This
; setting does not have any influence on the depth of children
that is send to the client through the
; Remote Debugging feature.
;xdebug.var_display_max_depth = 3
; Local Variables:
; tab-width: 4
; End:
extension=php_phalcon.dll
Ln 1, Col 1
The PHP Group
3 KB


XAMPP Control Panel Application
XAMPP Control Panel
Service...
SCM...
Modules
Svc
Apache
Running
Stop
Admin...
Svc
MySql
Start
Admin...
Status
Refresh
Explore...
Help
Exit
XAMPP Control Panel Version 2.5 (USB lite version 2011)
Windows 5.1 Build 2600 Platform 2 Service Pack 3
Current Directory: C:\xampp
Install(er)) Directory: No installer package found
Status Check OK
Busy...
Apache started [Port 80]

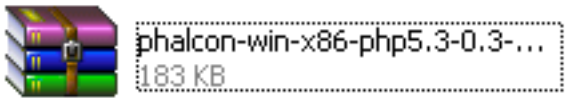

Firefox
XAMPP 1.7.7
localhost/xampp/index.php
XAMPP for W
XAMPP-USB-LITE
1.7.7
[PHP: 5.3.8]
Welcome
Status
Security
Documentation
Components
PHP
phpinfo()
CD Collection
Welcome to XAMPP for W
Congratulations:
You have successfully installed X
Now you can start using Apache an
For OpenSSL support please use th
Good luck, Kay Vogelgesang + Kai


XAMPP for Windows
English / Deutsch / Francais / Nederlands / Polski
Italiano / Norwegian / Español / □□ / Português
(Brasil) / □□□
PDO support
enabled
PDO drivers
no value
phalcon
Version
0.3.1
Phar
Phar: PHP Archive support
enabled
Phar EXT version
2.0.1

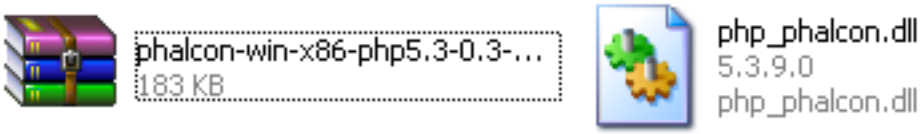
Using the latest WampServer version is highly recommended.

Download the right version of Phalcon WAMP has both 32 and 64 bit versions. From the download section, you can choose the Phalcon for Windows accordingly to your desired architecture.

After download the Phalcon library you will have a zip file like the one shown below:



Extract the library from the archive to get the Phalcon DLL:



Copy the file php_phalcon.dll to the PHP extensions. If WAMP is installed in the c:\wamp folder, the extension needs to be in C:\wamp\bin\php\php5.3.10\ext

Edit the php.ini file, it is located at C:\wamp\bin\php\php5.3.10\php.ini. It can be edited with Notepad or a similar program. We recommend Notepad++ to avoid issues with line endings. Append at the end of the file: extension=php_phalcon.dll and save it.

Also edit another php.ini file, which is located at C:\wamp\bin\apache\Apache2.2.21\bin\php.ini. Append at the end of the file: extension=php_phalcon.dll and save it.

Restart the Apache Web Server. Do a single click on the WampServer icon at system tray. Choose "Restart All Services" from the pop-up menu. Check out that tray icon will become green again.

Open your browser to navigate to http://localhost. The WAMP welcome page will appear. Look at the section "extensions loaded" to check if phalcon was loaded.

Congrats!, You are now flying with Phalcon.

Related Guides

- *General Installation*
- *Detailed Installation on XAMPP for Windows*

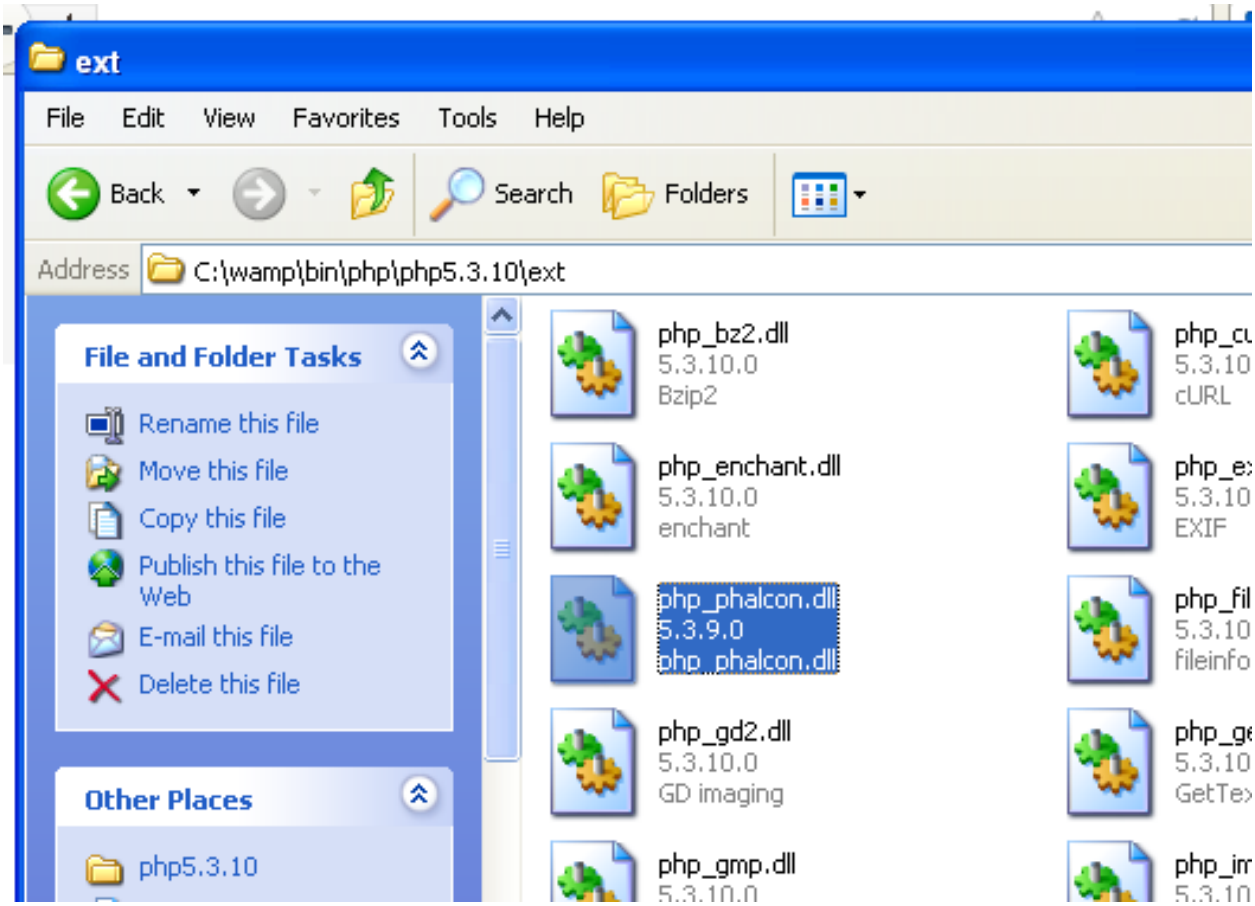
2.3.2 Linux/Solaris/Mac

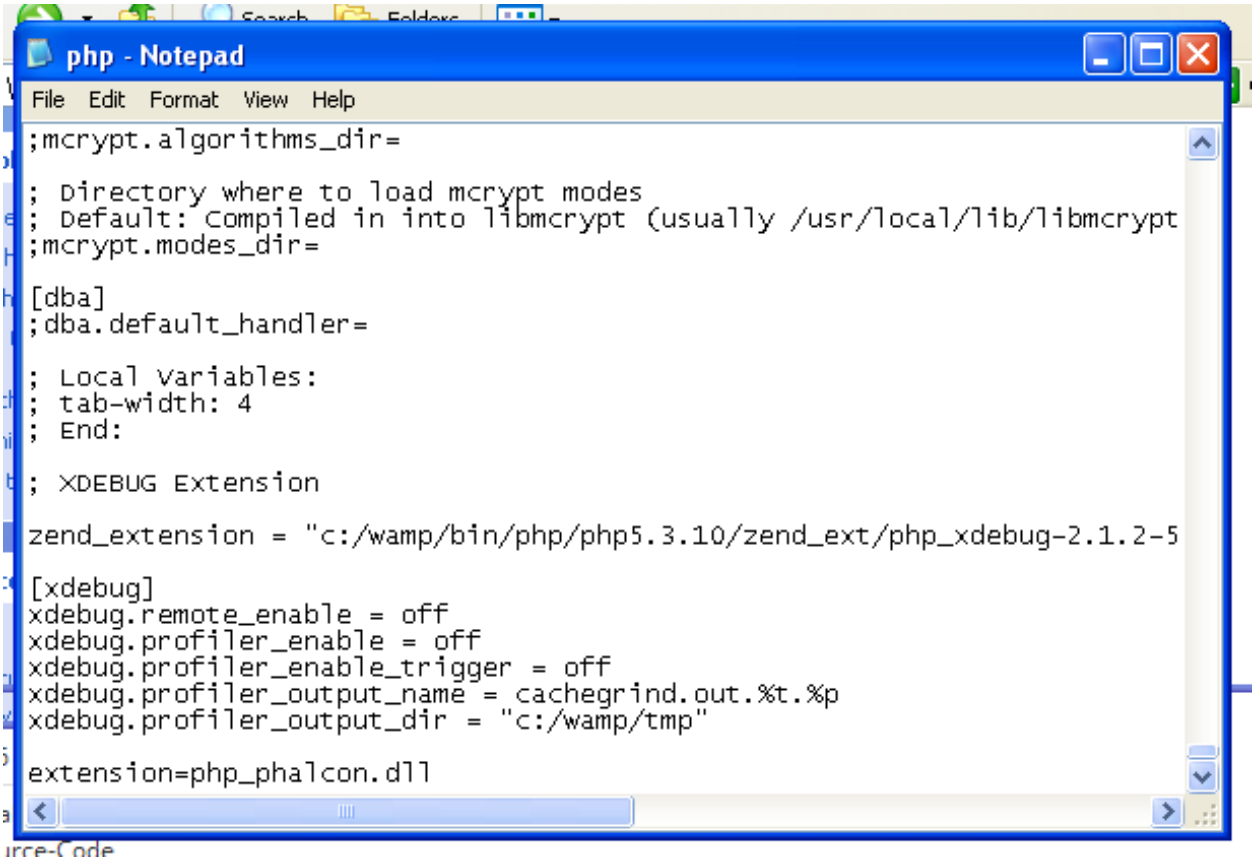
On a Linux/Solaris/Mac system you can easily compile and install the extension from the source code:

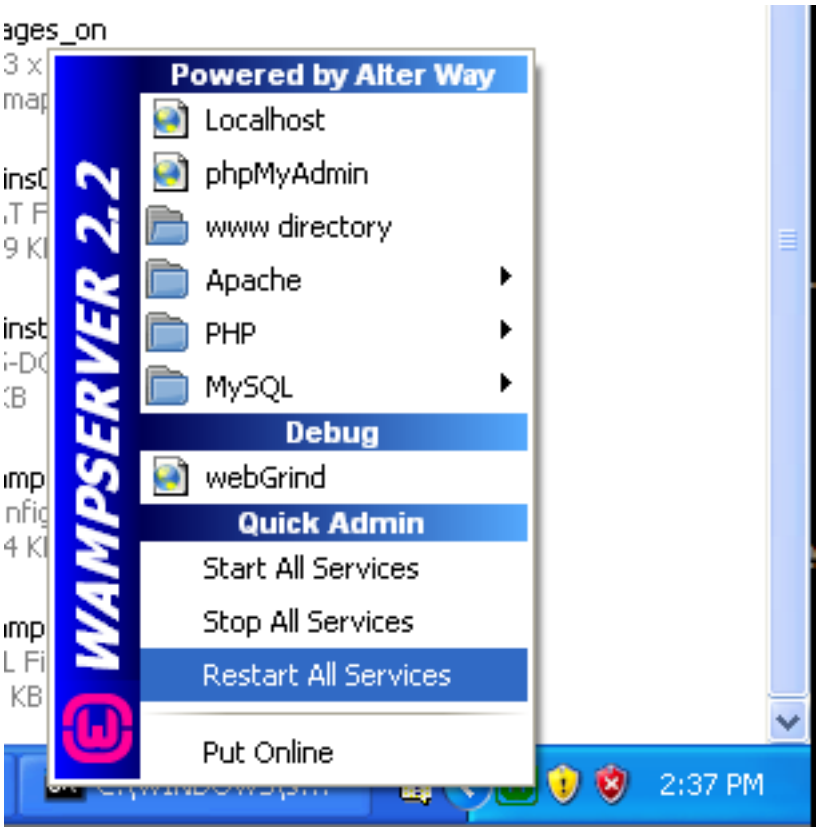
Requirements

Prerequisite packages are:

- PHP 5.3.x/5.4.x development resources


ext
File Edit View Favorites Tools Help
Back
Search
Folders
Address
C:\wamp\bin\php\php5.3.10\ext
File and Folder Tasks
Rename this file
Move this file
Copy this file
Publish this file to the Web
E-mail this file
Delete this file
Other Places
php5.3.10
php_bz2.dll
5.3.10.0
Bzip2
php_enchant.dll
5.3.10.0
enchant
php_phalcon.dll
5.3.9.0
php_phalcon.dll
php_gd2.dll
5.3.10.0
GD imaging
php_gmp.dll


php - Notepad
File Edit Format View Help
;mcrypt.algorithms_dir=
; Directory where to load mcrypt modes
; Default: Compiled in into libmcrypt (usually /usr/local/lib/libmcrypt
;mcrypt.modes_dir=
[dba]
;dba.default_handler=
; Local Variables:
; tab-width: 4
; End:
; XDEBUG Extension
zend_extension = "c:/wamp/bin/php/php5.3.10/zend_ext/php_xdebug-2.1.2-5
[xdebug]
xdebug.remote_enable = off
xdebug.profiler_enable = off
xdebug.profiler_enable_trigger = off
xdebug.profiler_output_name = cachegrind.out.%t.%p
xdebug.profiler_output_dir = "c:/wamp/tmp"
extension=php_phalcon.dll


Powered by Alter Way
WAMPSERVER 2.2
Localhost
phpMyAdmin
www directory
Apache
PHP
MySQL
Debug
webGrind
Quick Admin
Start All Services
Stop All Services
Restart All Services
Put Online
2:37 PM

- GCC compiler (Linux/Solaris) or Xcode (Mac)
- Git (if not already installed in your system - unless you download the package from GitHub and upload it on your server via FTP/SFTP)

Specific packages for common platforms:

```
#Ubuntu
sudo apt-get install git-core gcc autoconf
sudo apt-get install php5-dev php5-mysql

#Suse
sudo yast -i gcc make autoconf2.13
sudo yast -i php5-devel php5-mysql

#CentOS/RedHat
sudo yum install gcc make
sudo yum install php-devel

#Solaris
pkg install gcc-45
pkg install php-53 apache-php53
```

Compilation

Creating the extension:

```
git clone git://github.com/phalcon/cphalcon.git
cd cphalcon/build
sudo ./install
```

Add extension to your php.ini

```
extension=phalcon.so
```

Restart the webserver.

Phalcon automatically detects your architecture, however, you can force the compilation for a specific architecture:

```
sudo ./install 32bits
sudo ./install 64bits
sudo ./install safe
```

2.3.3 FreeBSD

A port is available for FreeBSD. Just only need these simple line commands to install it:

```
pkg_add -r phalcon
```

or

```
export CFLAGS="-O2 -fno-delete-null-pointer-checks"
cd /usr/ports/www/phalcon && make install clean
```

2.3.4 Installation Notes

Installation notes for Web Servers:

Apache Installation Notes

Apache is a popular and well known web server available on many platforms.

Configuring Apache for Phalcon

The following are potential configurations you can use to setup Apache with Phalcon. These notes are primarily focused on the configuration of the mod-rewrite module allowing to use friendly urls and the *router component*. Commonly an application has the following structure:

```
test/
  app/
    controllers/
    models/
    views/
  public/
    css/
    img/
    js/
    index.php
```

Directory under the main Document Root This being the most common case, the application is installed in any directory under the document root. In this case, we use two .htaccess files, the first one to hide the application code forwarding all requests to the application's document root (public/).

```
# test/.htaccess

<IfModule mod_rewrite.c>
    RewriteEngine on
    RewriteRule  ^$ public/    [L]
    RewriteRule  (.*) public/$1 [L]
</IfModule>
```

Now a second .htaccess file is located in the public/ directory, this re-writes all the URIs to the public/index.php file:

```
# test/public/.htaccess

<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php?_url=/$1 [QSA,L]
</IfModule>
```

If you do not want to use .htaccess files you can move these configurations to the apache's main configuration file:

```
<IfModule mod_rewrite.c>

    <Directory "/var/www/test">
        RewriteEngine on
        RewriteRule  ^$ public/    [L]
        RewriteRule  (.*) public/$1 [L]
    </Directory>

    <Directory "/var/www/test/public">
        RewriteEngine On
        RewriteCond %{REQUEST_FILENAME} !-d
        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteRule ^(.*)$ index.php?_url=/$1 [QSA,L]
    </Directory>

</IfModule>
```

Virtual Hosts And this second configuration allows you to install a Phalcon application in a virtual host:

```
<VirtualHost *:80>

    ServerAdmin admin@example.host
    DocumentRoot "/var/vhosts/test/public"
    DirectoryIndex index.php
    ServerName example.host
    ServerAlias www.example.host

    <Directory "/var/vhosts/test/public">
        Options All
        AllowOverride All
        Allow from all
    </Directory>

</VirtualHost>
```

Nginx Installation Notes

Nginx is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server. Unlike traditional servers, Nginx doesn't rely on threads to handle requests. Instead it uses a much more scalable event-driven (asynchronous) architecture. This architecture uses small, but more importantly, predictable amounts of memory under load.

The PHP-FPM (FastCGI Process Manager) is usually used to allow Nginx to process PHP files. Nowadays, PHP-FPM is bundled with any Unix PHP distribution. Phalcon + Nginx + PHP-FPM provides a powerful set of tools that offer maximum performance for your PHP applications.

Configuring Nginx for Phalcon

The following are potential configurations you can use to setup nginx with Phalcon:

Basic configuration

```
server {
    listen   8080;
    server_name localhost.dev;

    root /var/www/phalcon/public;
    index index.php index.html index.htm;

    location / {
        if (-f $request_filename) {
            break;
        }

        if (!-e $request_filename) {
            rewrite ^(.+)$ /index.php?_url=$1 last;
            break;
        }
    }

    location ~ \.php$ {
            try_files $uri =404;
            fastcgi_split_path_info ^(.+\.php)(/.+)$;
            fastcgi_pass 127.0.0.1:9000;
            fastcgi_index index.php;
            fastcgi_param  SCRIPT_FILENAME  $document_root$fastcgi_script_name;
            include fastcgi_params;
    }
}
```

Dedicated Instance

```
server {
    listen       80;
    server_name  localhost;

    charset      utf-8;

    #access_log  /var/log/nginx/host.access.log  main;

    location / {
```

```
        root   /srv/www/htdocs/phalcon-website/public;
        index  index.php index.html index.htm;

        # if file exists return it right away
        if (-f $request_filename) {
            break;
        }

        # otherwise rewrite it
        if (!-e $request_filename) {
            rewrite ^(.+)$ /index.php?_url=$1 last;
            break;
        }
    }

    location ~ \.php {
        # try_files    $uri =404;

        fastcgi_index  /index.php;
        fastcgi_pass   127.0.0.1:9000;

        include fastcgi_params;
        fastcgi_split_path_info       ^(.+\.php)(/.+)$;
        fastcgi_param PATH_INFO       $fastcgi_path_info;
        fastcgi_param PATH_TRANSLATED $document_root$fastcgi_path_info;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }

    location ~* ^/(css|img|js|flv|swf|download)/(.+)$ {
        root $root_path;
    }
}
```

Configuration by Host And this second configuration allow you to have different configurations by host:

```
server {
    listen      80;
    server_name localhost;
    set         $root_path '/var/www/$host/public';
    root        $root_path;

    access_log  /var/log/nginx/$host-access.log;
    error_log   /var/log/nginx/$host-error.log error;

    index index.php index.html index.htm;

    try_files $uri $uri/ @rewrite;

    location @rewrite {
        rewrite ^/(.*)$ /index.php?_url=$1;
    }

    location ~ \.php {
        # try_files    $uri =404;

        fastcgi_index  /index.php;
        fastcgi_pass   127.0.0.1:9000;
```

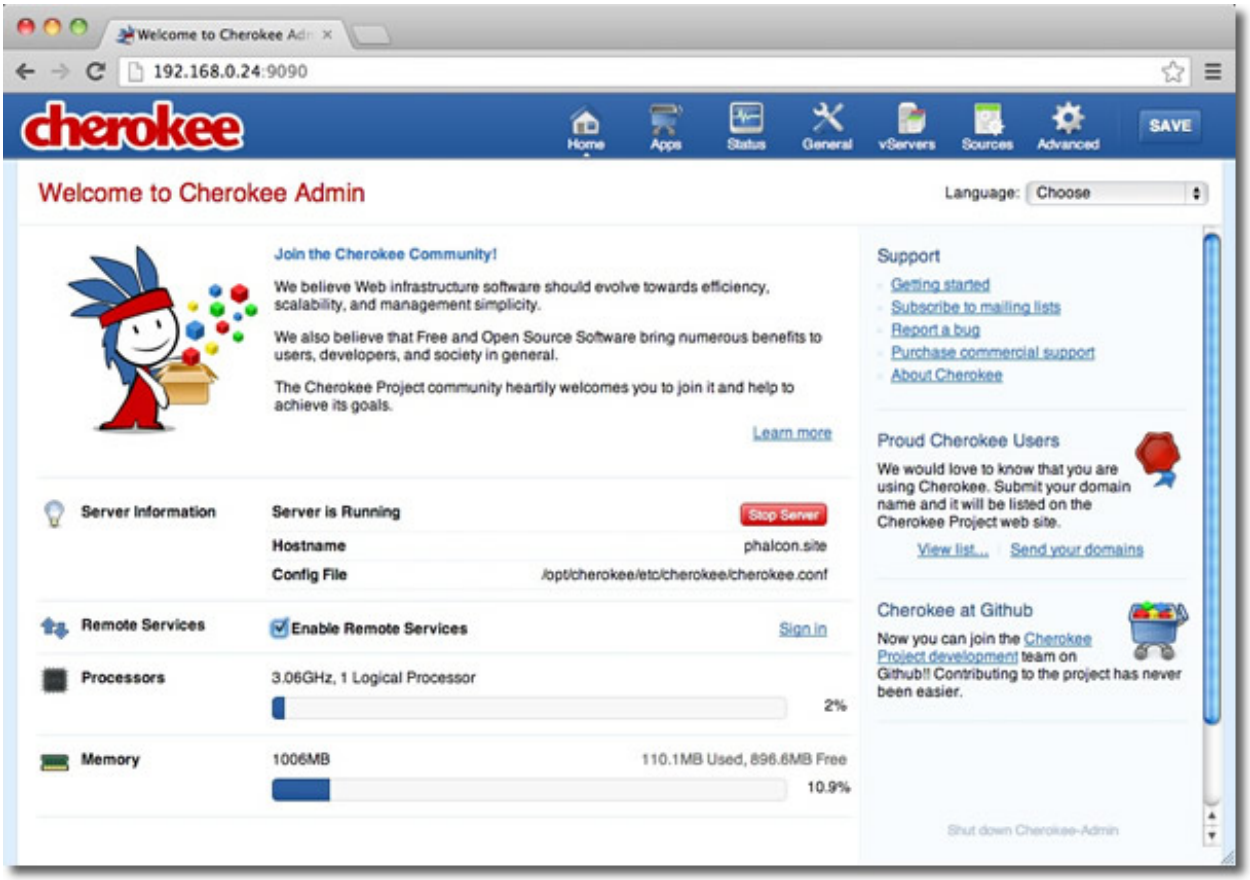
```
        include fastcgi_params;
        fastcgi_split_path_info       ^(.+\.php)(/.+)$;
        fastcgi_param PATH_INFO       $fastcgi_path_info;
        fastcgi_param PATH_TRANSLATED $document_root$fastcgi_path_info;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }

    location ~* ^/(css|img|js|flv|swf|download)/(.+)$ {
        root $root_path;
    }

    location ~ /\.ht {
        deny all;
    }
}
```

Cherokee Installation Notes

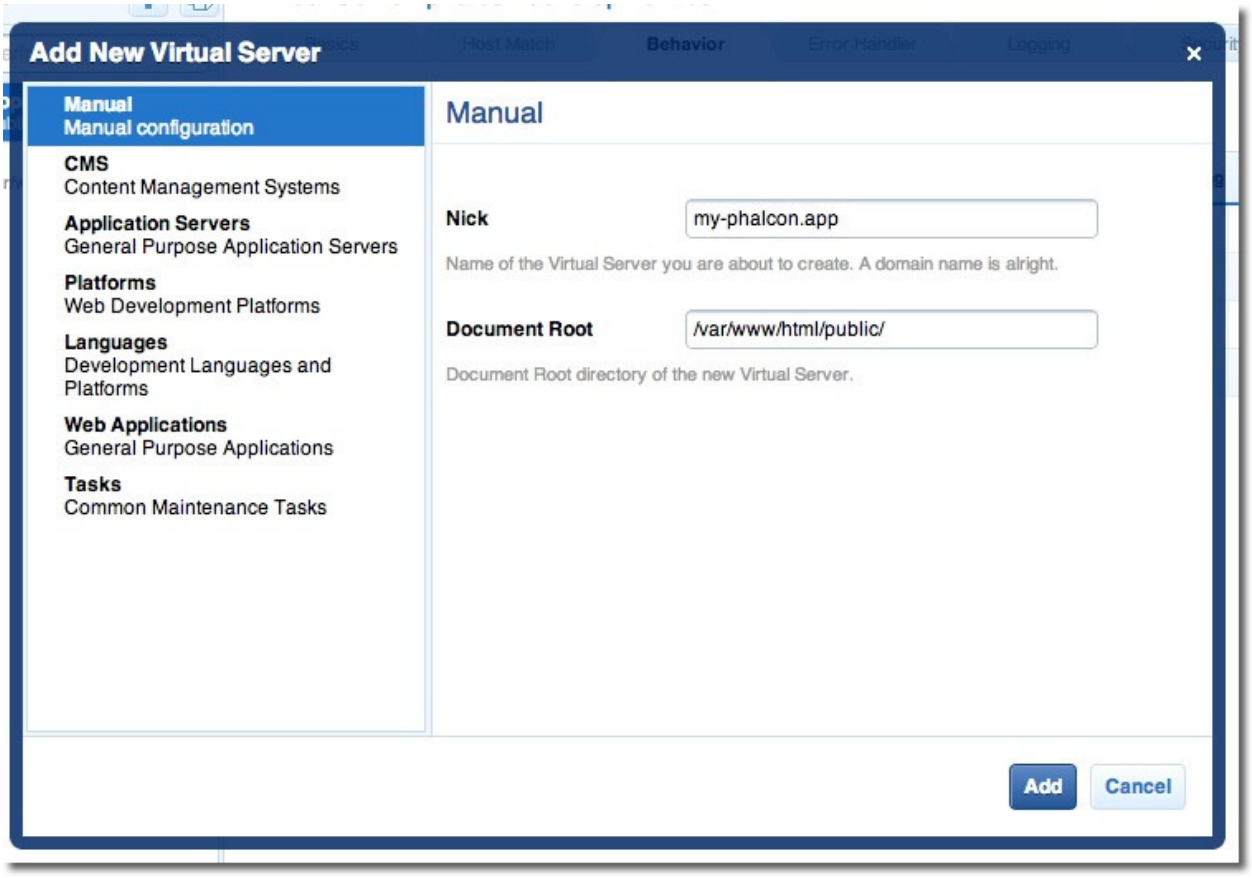
Cherokee is a high-performance web server. It is very fast, flexible and easy to configure.

Configuring Cherokee for Phalcon

Cherokee provides a friendly graphical interface to configure almost every setting available in the web server. Start the cherokee administrator by executing with root /path-to-cherokee/sbin/cherokee-admin



Create a new virtual host by clicking on 'vServers', then add a new virtual server:

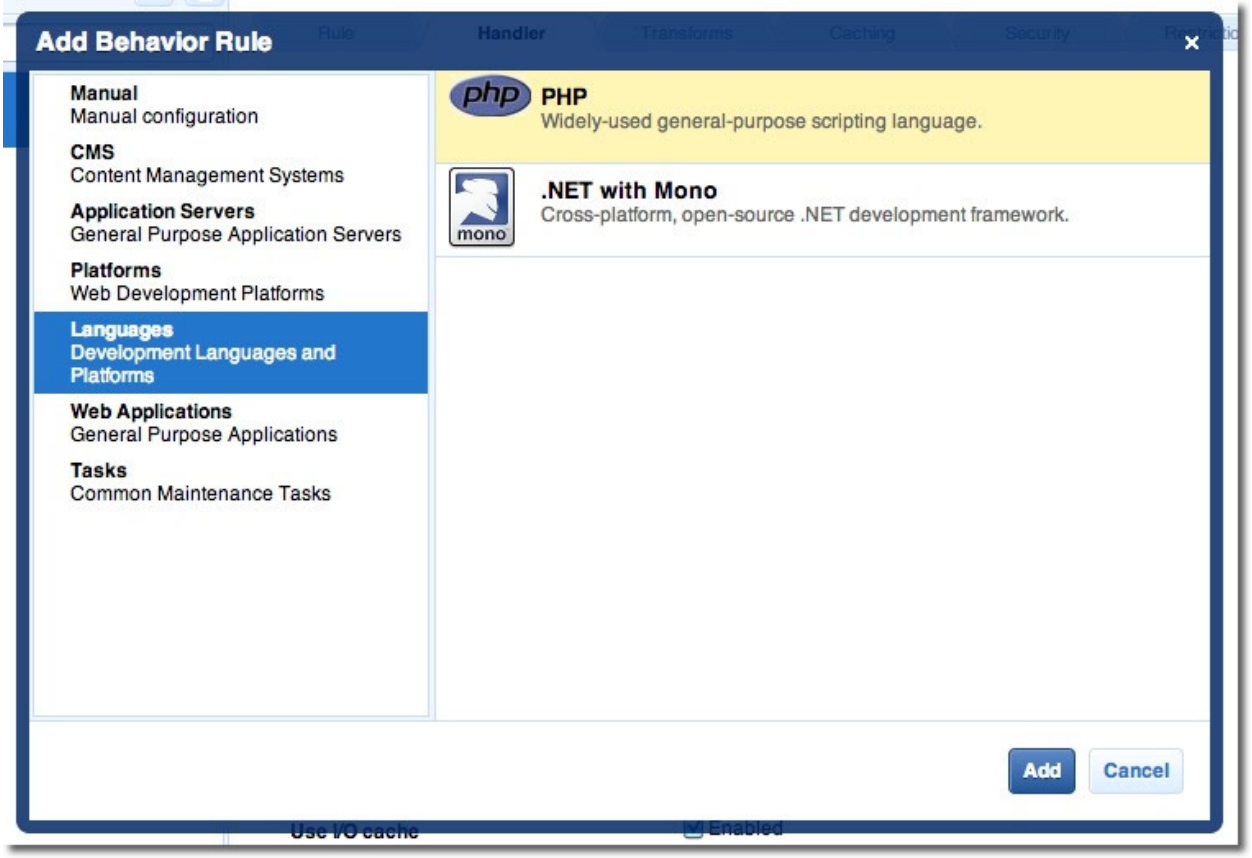


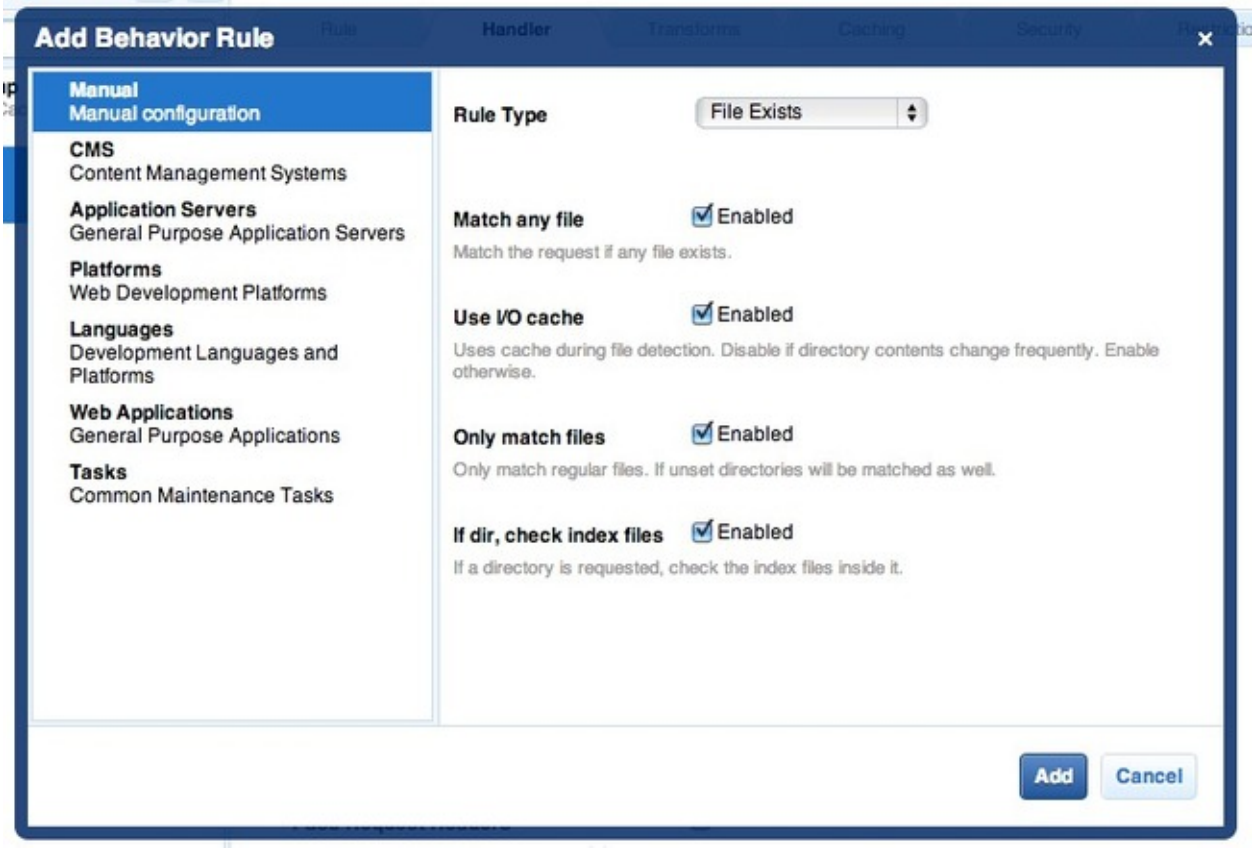
The recently added virtual server must appear at the left bar of the screen. In the 'Behaviors' tab you will see a set of default behaviors for this virtual server. Click the 'Rule Management' button. Remove those labeled as 'Directory /cherokee_themes' and 'Directory /icons':

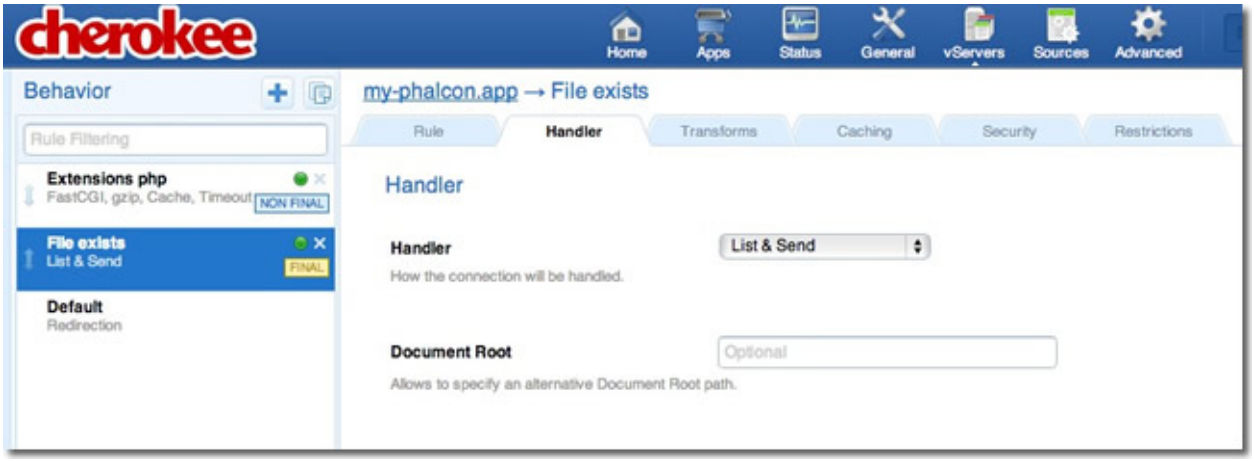
Add the 'PHP Language' behavior using the wizard. This behavior allow you to run PHP applications:

Normally this behavior does not require additional settings. Add another behavior, this time in the 'Manual Configuration' section. In 'Rule Type' choose 'File Exists', then make sure the option 'Match any file' is enabled:

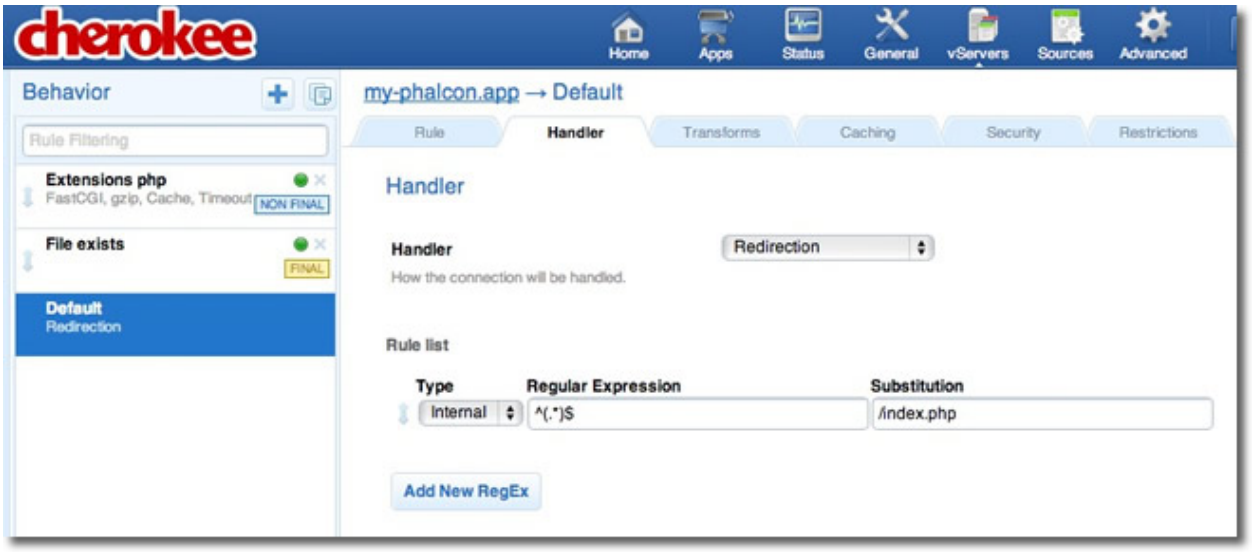
In the 'Handler' tab choose 'List & Send' as handler:


Add Behavior Rule
Manual
Manual configuration
CMS
Content Management Systems
Application Servers
General Purpose Application Servers
Platforms
Web Development Platforms
Languages
Development Languages and Platforms
Web Applications
General Purpose Applications
Tasks
Common Maintenance Tasks
PHP
Widely-used general-purpose scripting language.
.NET with Mono
Cross-platform, open-source .NET development framework.
Add
Cancel


Add Behavior Rule
Manual
Manual configuration
CMS
Content Management Systems
Application Servers
General Purpose Application Servers
Platforms
Web Development Platforms
Languages
Development Languages and Platforms
Web Applications
General Purpose Applications
Tasks
Common Maintenance Tasks
Rule Type
File Exists
Match any file
Enabled
Match the request if any file exists.
Use I/O cache
Enabled
Uses cache during file detection. Disable if directory contents change frequently. Enable otherwise.
Only match files
Enabled
Only match regular files. If unset directories will be matched as well.
If dir, check index files
Enabled
If a directory is requested, check the index files inside it.
Add
Cancel



Edit the 'Default' behavior in order to enable the URL-rewrite engine. Change the handler to 'Redirection', then add the following regular expression to the engine ^(.*)$:



Finally, make sure the behaviors have the following order:

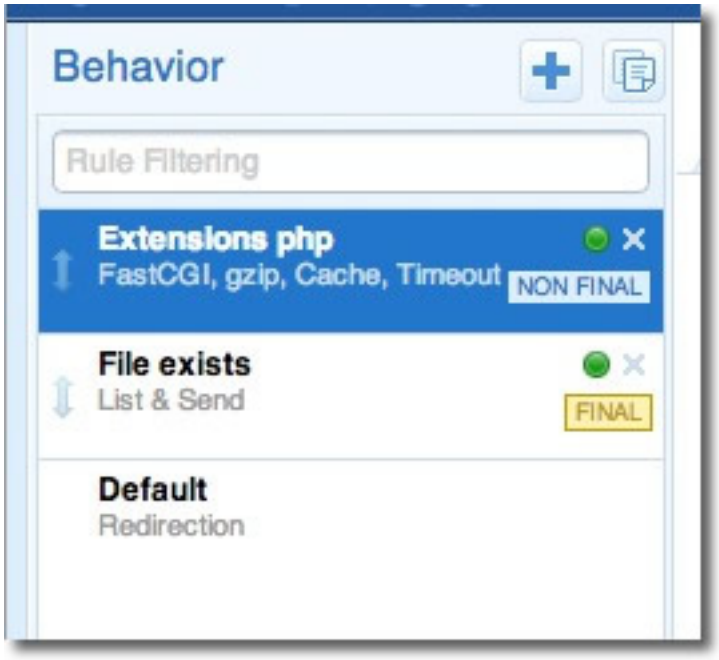
Execute the application in a browser:

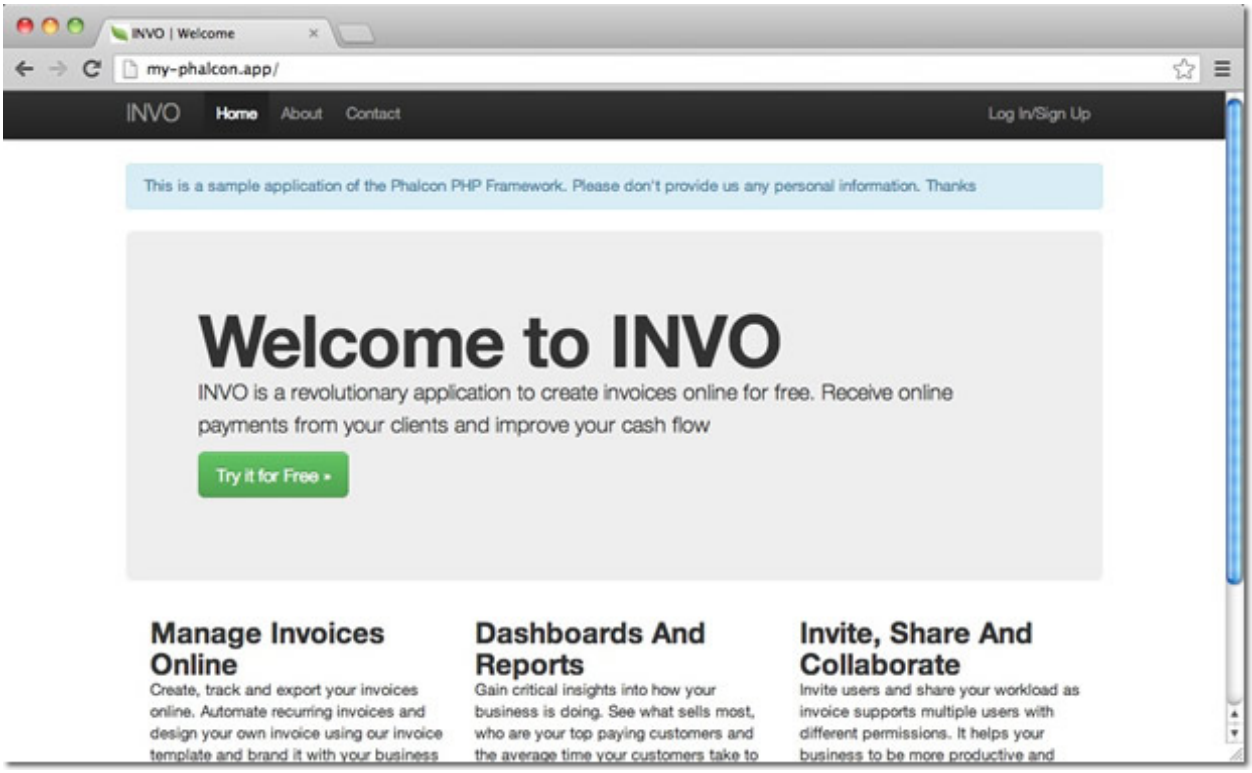
2.4 Tutorial 1: Let's learn by example

Throughout this first tutorial, we'll walk you through the creation of an application with a simple registration form from the ground up. We will also explain the basic aspects of the framework's behavior. If you are interested in automatic code generation tools for Phalcon, you can check our *developer tools*.

2.4.1 Checking your installation

We'll assume you have Phalcon installed already. Check your phpinfo() output for a section referencing "Phalcon" or execute the code snippet below:


Behavior
Rule Filtering
Extensions php
FastCGI, gzip, Cache, Timeout
NON FINAL
File exists
List & Send
FINAL
Default
Redirection


INVO | Welcome
my-phalcon.app/
INVO
Home
About
Contact
Log In/Sign Up
This is a sample application of the Phalcon PHP Framework. Please don't provide us any personal information. Thanks
Welcome to INVO
INVO is a revolutionary application to create invoices online for free. Receive online payments from your clients and improve your cash flow
Try it for Free »
Manage Invoices Online
Create, track and export your invoices online. Automate recurring invoices and design your own invoice using our invoice template and brand it with your business
Dashboards And Reports
Gain critical insights into how your business is doing. See what sells most, who are your top paying customers and the average time your customers take to
Invite, Share And Collaborate
Invite users and share your workload as invoice supports multiple users with different permissions. It helps your business to be more productive and

```
<?php print_r(get_loaded_extensions()); ?>
```

The Phalcon extension should appear as part of the output:

```
Array
(
    [0] => Core
    [1] => libxml
    [2] => filter
    [3] => SPL
    [4] => standard
    [5] => phalcon
    [6] => pdo_mysql
)
```

2.4.2 Creating a project

The best way to use this guide is to follow each step in turn. You can get the complete code here.

File structure

Phalcon does not impose a particular file structure for application development. Due to the fact that it is loosely coupled, you can implement Phalcon powered applications with a file structure you are most comfortable using.

For the purposes of this tutorial and as a starting point, we suggest the following structure:

```
tutorial/
  app/
    controllers/
    models/
    views/
  public/
    css/
    img/
    js/
```

Note that you don't need any "library" directory related to Phalcon. The framework is available in memory, ready for you to use.

Beautiful URLs

We'll use pretty (friendly) urls for this tutorial. Friendly URLs are better for SEO as well as they are easy for users to remember. Phalcon supports rewrite modules provided by the most popular web servers. Making your application's URLs friendly is not a requirement and you can just as easy develop without them.

In this example we'll use the rewrite module for Apache. Let's create a couple of rewrite rules in the /.htaccess file:

```
#/.htaccess
<IfModule mod_rewrite.c>
    RewriteEngine on
    RewriteRule  ^$ public/    [L]
    RewriteRule  (.*) public/$1 [L]
</IfModule>
```

All requests to the project will be rewritten to the public/ directory making it the document root. This step ensures that the internal project folders remain hidden from public viewing and thus posing security threats.

The second set of rules will check if the requested file exists, and if it does it doesn't have to be rewritten by the web server module:

```
#/public/.htaccess
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php?_url=/$1 [QSA,L]
</IfModule>
```

Bootstrap

The first file you need to create is the bootstrap file. This file is very important; since it serves as the base of your application, giving you control of all aspects of it. In this file you can implement initialization of components as well as application behavior.

The public/index.php file should look like:

```
<?php

try {

    //Register an autoloader
    $loader = new \Phalcon\Loader();
    $loader->registerDirs(array(
        '../app/controllers/',
        '../app/models/'
    ))->register();

    //Create a DI
    $di = new Phalcon\DI\FactoryDefault();

    //Setting up the view component
    $di->set('view', function(){
        $view = new \Phalcon\Mvc\View();
        $view->setViewsDir('../app/views/');
        return $view;
    });

    //Handle the request
    $application = new \Phalcon\Mvc\Application();
    $application->setDI($di);
    echo $application->handle()->getContent();

} catch(\Phalcon\Exception $e) {
     echo "PhalconException: ", $e->getMessage();
}
```

Autoloaders

The first part that we find in the boostrap is registering an autoloader. This will be used to load classes as controllers and models in the application. For example we may register one or more directories of controllers increasing the flexibility of the application. In our example we have used the component Phalcon\Loader.

With it, we can load classes using various strategies but for this example we have chosen to locate classes based on predefined directories:

```
<?php

$loader = new \Phalcon\Loader();
$loader->registerDirs(
    array(
        '../app/controllers/',
        '../app/models/'
    )
)->register();
```

Dependency Management

A very important concept that must be understood when working with Phalcon is its *dependency injection container*. It may sound complex but is actually very simple and practical.

A service container is a bag where we globally store the services that our application will use to work. Each time the framework requires a component, will ask the container using a name service agreed. Since Phalcon is a highly decoupled framework, Phalcon\DI acts as glue facilitating the integration of the different components achieving their work together in a transparent manner.

```
<?php

//Create a DI
$di = new Phalcon\DI\FactoryDefault();
```

Phalcon\DI\FactoryDefault is a variant of Phalcon\DI. To make things easier, it has registered most of the components that come with Phalcon. Thus we should not register them one by one. Later there will be no problem in replacing a factory service.

In the next part, we register the "view" service indicating the directory where the framework will find the views files. As the views do not correspond to classes, they cannot be charged with an autoloader.

Services can be registered in several ways, but for our tutorial we'll use lambda functions:

```
<?php

//Setting up the view component
$di->set('view', function(){
    $view = new \Phalcon\Mvc\View();
    $view->setViewsDir('../app/views/');
    return $view;
});
```

In the last part of this file, we find *Phalcon\Mvc\Application*. Its purpose is to initialize the request environment, route the incoming request, and then dispatch any discovered actions; it aggregates any responses and returns them when the process is complete.

```
<?php

$application = new \Phalcon\Mvc\Application();
$application->setDI($di);
echo $application->handle()->getContent();
```

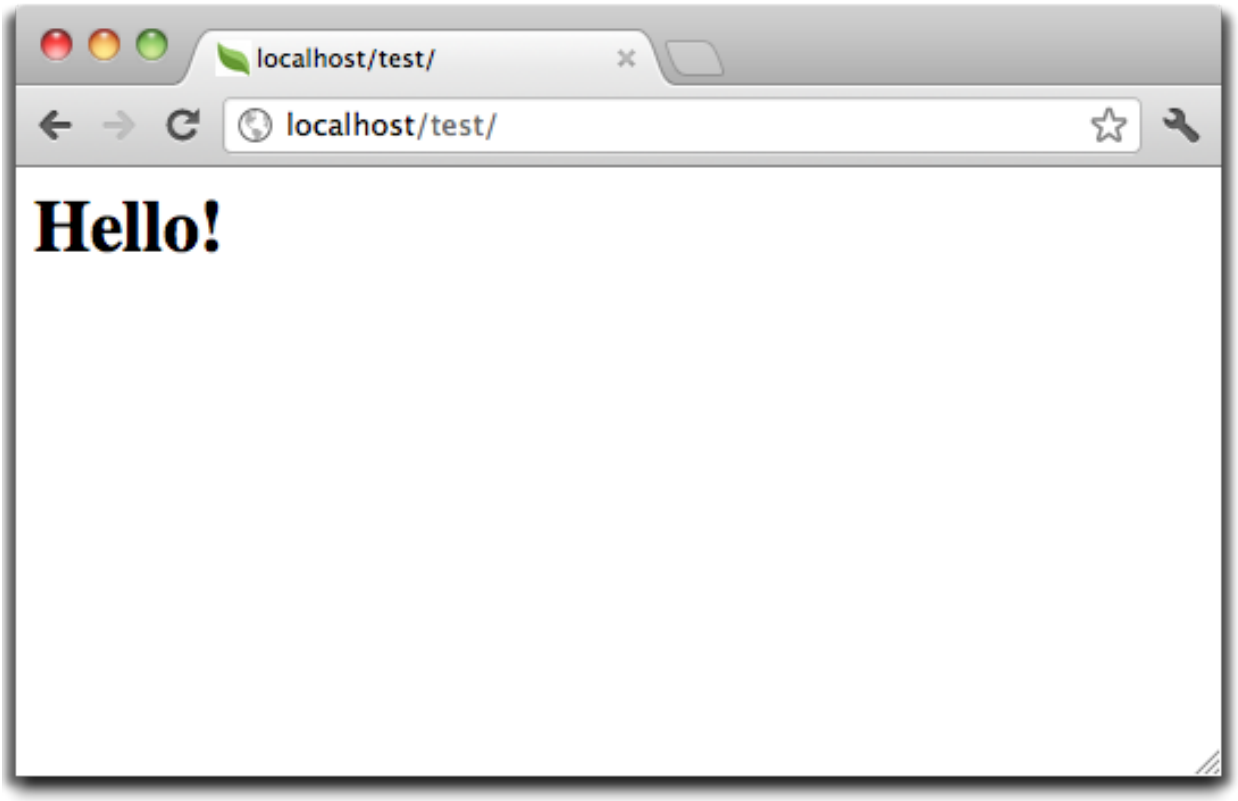
As you can see, the bootstrap file is very short and we do not need to include any additional files. We have set ourselves a flexible MVC application in less than 30 lines of code.

Creating a Controller

By default Phalcon will look for a controller named "Index". It is the starting point when no controller or action has been passed in the request. The index controller (app/controllers/IndexController.php) looks like:

```
<?php

class IndexController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {
        echo "<h1>Hello!</h1>";
    }

}
```

The controller classes must have the suffix "Controller" and controller actions must have the suffix "Action". If you access the application from your browser, you should see something like this:



Congratulations, you're flying with Phalcon!

Sending output to a view

Sending output on the screen from the controller is at times necessary but not desirable as most purists in the MVC community will attest. Everything must be passed to the view that is responsible for outputting data on screen. Phalcon will look for a view with the same name as the last executed action inside a directory named as the last executed controller. In our case (app/views/index/index.phtml):

```
<?php echo "<h1>Hello!</h1>";
```

Our controller (app/controllers/IndexController.php) now has an empty action definition:

```
<?php

class IndexController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

}
```

The browser output should remain the same. The *Phalcon\Mvc\View* static component is automatically created when the action execution has ended. Learn more about *views usage here* .

Designing a sign up form

Now we will change the index.phtml view file, to add a link to a new controller named "signup". The goal is to allow users to sign up in our application.

```
<?php

echo "<h1>Hello!</h1>";

echo Phalcon\Tag::linkTo("signup", "Sign Up Here!");
```

The generated HTML code displays an "A" html tag linking to a new controller:

```
<h1>Hello!</h1> <a href="/test/signup">Sign Up Here!</a>
```

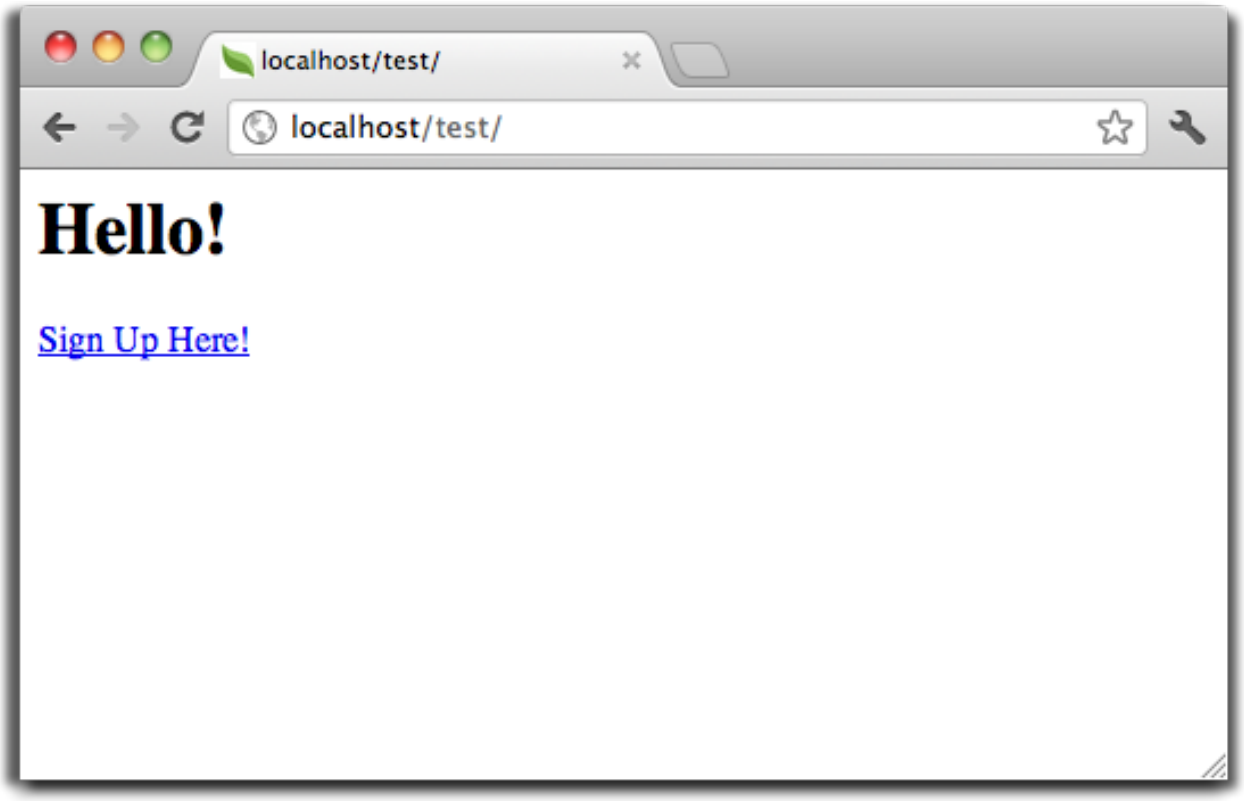
To generate the tag we use the class *Phalcon\Tag*. This is a utility class that allows us to build HTML tags with framework conventions in mind. A more detailed article regarding HTML generation can be *found here*

Here is the controller Signup (app/controllers/SignupController.php):

```
<?php

class SignupController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

}
```

The empty index action gives the clean pass to a view with the form definition:

```
<?php use Phalcon\Tag; ?>

<h2>Sign using this form</h2>

<?php echo Tag::form("signup/register"); ?>
```




```
<p>
    <label for="name">Name</label>
    <?php echo Tag::textField("name") ?>
</p>

<p>
    <label for="name">E-Mail</label>
    <?php echo Tag::textField("email") ?>
</p>

<p>
    <?php echo Tag::submitButton("Register") ?>
</p>

</form>
```

Viewing the form in your browser will show something like this:

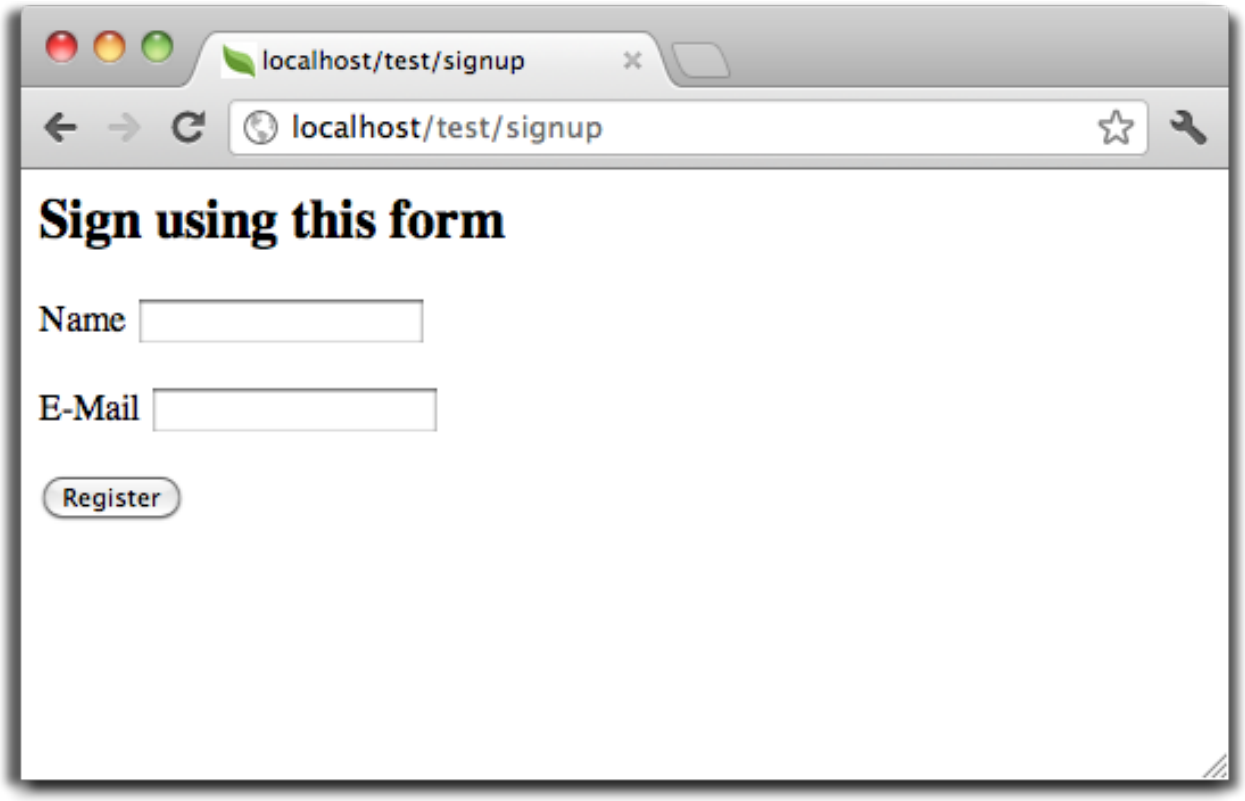
Phalcon\Tag also provides useful methods to build form elements.

The Phalcon\Tag::form method receives only one parameter for instance, a relative uri to a controller/action in the application.

By clicking the "Send" button, you will notice an exception thrown from the framework, indicating that we are missing the "register" action in the controller "signup". Our public/index.php file throws this exception:

> PhalconException: Action "register" was not found on controller "signup"

Implementing that method will remove the exception:




```
<?php

class SignupController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function registerAction()
    {

    }

}
```

If you click the "Send" button again, you will see a blank page. The name and email input provided by the user should be stored in a database. According to MVC guidelines, database interactions must be done through models so as to ensure clean object-oriented code.

Creating a Model

Phalcon brings the first ORM for PHP entirely written in C-language. Instead of increasing the complexity of development, it simplifies it.

Before creating our first model, we need a database table to map it to. A simple table to store registered users can be defined like this:

```
CREATE TABLE `users` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(70) NOT NULL,
  `email` varchar(70) NOT NULL,
  PRIMARY KEY (`id`)
);
```

A model should be located in the app/models directory. The model mapping to "users" table:

```
<?php

class Users extends \Phalcon\Mvc\Model
{

}
```

Setting a Database Connection

In order to be able to use a database connection and subsequently access data through our models, we need to specify it in our bootstrap process. A database connection is just another service that our application has that can be use for several components:

```
<?php

try {

    //Register an autoloader
    $loader = new \Phalcon\Loader();
    $loader->registerDirs(array(
        '../app/controllers/',
        '../app/models/'
    ))->register();

    //Create a DI
    $di = new Phalcon\DI\FactoryDefault();

    //Set the database service
    $di->set('db', function(){
        return new \Phalcon\Db\Adapter\Pdo\Mysql(array(
            "host" => "localhost",
            "username" => "root",
            "password" => "secret",
            "dbname" => "test_db"
        ));
    });

    //Setting up the view component
    $di->set('view', function(){
        $view = new \Phalcon\Mvc\View();
        $view->setViewsDir('../app/views/');
        return $view;
    });

    //Handle the request
    $application = new \Phalcon\Mvc\Application();
    $application->setDI($di);
    echo $application->handle()->getContent();
```

```
} catch(\Phalcon\Exception $e) {
     echo "PhalconException: ", $e->getMessage();
}
```

With the correct database parameters, our models are ready to work and interact with the rest of the application.

Storing data using models

Receiving data from the form and storing them in the table is the next step.

```
<?php

class SignupController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function registerAction()
    {

        $user = new Users();

        //Store and check for errors
        if ($user->save($_POST, array('name', 'email')) == true) {
            echo "Thanks for register!";
        } else {
            echo "Sorry, the following problems were generated: ";
            foreach ($user->getMessages() as $message) {
                echo $message->getMessage(), "<br/>";
            }
        }
    }

}
```

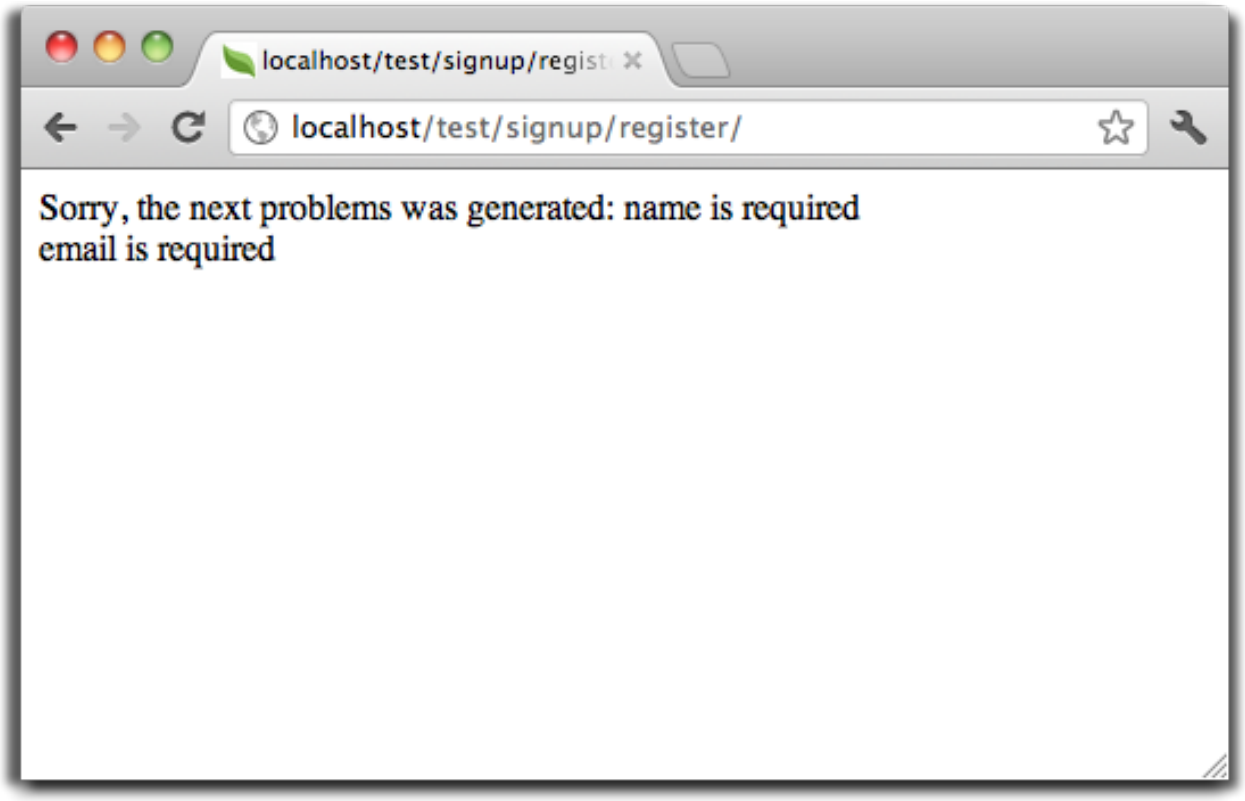
We then instantiate the Users class, which corresponds to a User record. The class public properties map to the fields of the record in the users table. Setting the relevant values in the new record and calling save() will store the data in the database for that record. The save() method returns a boolean value which informs us on whether the storing of the data was successful or not.

The ORM automatically escapes the input preventing SQL injections so we only need to pass the request to the method save().

Additional validation happens automatically on fields that are not null (required). If we don't type any of the required files our screen will look like this:

2.4.3 Conclusion

This is a very simple tutorial and as you can see, it's easy to start building an application using Phalcon. The fact that Phalcon is an extension on your web server has not interfered with the ease of development or features available. We invite you to continue reading the manual so that you can discover additional features offered by Phalcon!



2.4.4 Sample Applications

The following Phalcon powered applications are also available, providing more complete examples:

- INVO application: Invoice generation application. Allows for management of products, companies, product types. etc.
- PHP Alternative website: Multilingual and advanced routing application
- Album O'Rama: A showcase of music albums, handling big sets of data with *PHQL* and using *Volt* as template engine
- Phosphorum: A simple and clean forum

2.5 Tutorial 2: Explaining INVO

In this second tutorial, we'll explain a more complete application in order to deepen the development with Phalcon. INVO is one of the applications we have created as samples. INVO is a small website that allows their users to generate invoices, and do other tasks as manage their customers and products. You can clone its code from Github.

Also, INVO was made with Twitter Bootstrap as client-side framework. Although the application does not generate invoices still it serves as an example to understand how the framework works.

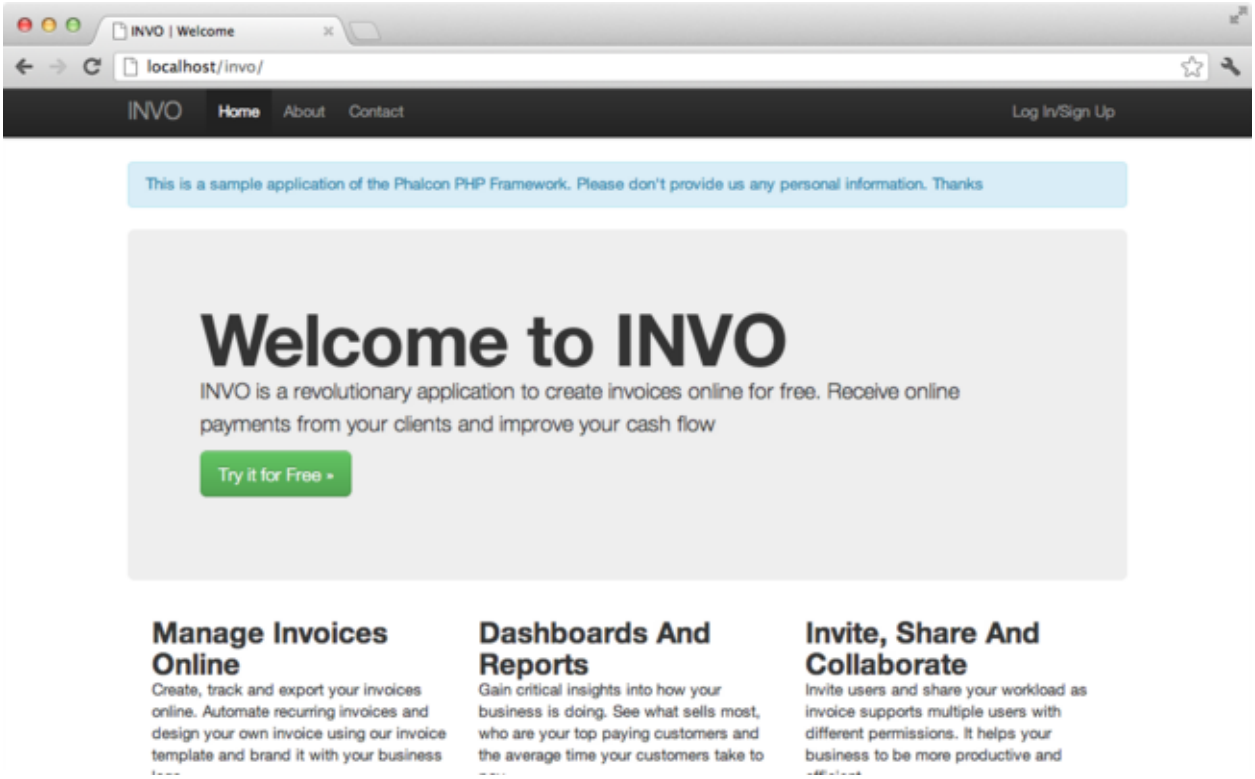
2.5.1 Project Structure

Once you clone the project in your document root you'll see the following structure:

```
invo/
    app/
        app/config/
        app/controllers/
        app/library/
        app/models/
        app/plugins/
        app/views/
    public/
        public/bootstrap/
        public/css/
        public/js/
    schemas/
```

As you know, Phalcon does not impose a particular file structure for application development. This project provides a simple MVC structure and a public document root.

Once you open the application in your browser http://localhost/invo you'll something like this:



The application is divided in two parts, a frontend, that is a public part where visitors can receive information about INVO and request contact information. The second part is the backend, an administrative area where a registered user can manage his/her products and customers.

2.5.2 Routing

INVO uses the standard route that is built-in with the Router component. These routes matches the following pattern: /:controller/:action/:params. This means that the first part of a URI is the controller, the second the action and the rest are the parameters.

The following route /session/register executes the controller SessionController and its action registerAction.

2.5.3 Configuration

INVO has a configuration file that sets general parameters in the application. This file is read in the first lines of the bootstrap file (public/index.php):

```
<?php

//Read the configuration
$config = new Phalcon\Config\Adapter\Ini(__DIR__ . '/../app/config/config.ini');
```

Phalcon\Config allows us to manipulate the file in an object-oriented way. The configuration file contains the following settings:

```
[database]
host     = localhost
username = root
password = secret
name     = invo

[application]
controllersDir = /../app/controllers/
modelsDir      = /../app/models/
viewsDir       = /../app/views/
pluginsDir     = /../app/plugins/
libraryDir     = /../app/library/
baseUri        = /invo/

;[metadata]
;adapter = "Apc"
;suffix = my-suffix
;lifetime = 3600
```

Phalcon hasn't any pre-defined convention settings. Sections help us to organize the options as appropriate. In this file there are three sections to be used later.

2.5.4 Autoloaders

A second part that appears in the boostrap file (public/index.php) is the autoloader. The autoloader registers a set of directories where the application will look for the classes that it eventually will need.

```
<?php

$loader = new \Phalcon\Loader();

$loader->registerDirs(
    array(
        __DIR__ . $config->application->controllersDir,
        __DIR__ . $config->application->pluginsDir,
        __DIR__ . $config->application->libraryDir,
        __DIR__ . $config->application->modelsDir,
    )
)->register();
```

Note that what has been done is registing the directories that were defined in the configuration file. The only directory that is not registered is the viewsDir, because it contains no classes but html + php files.

2.5.5 Handling the Request

Let's go much further, at the end of the file, the request is finally handled by Phalcon\Mvc\Application, this class initializes and executes all the necessary to make the application run:

```
<?php

$application = new \Phalcon\Mvc\Application();
$application->setDI($di);
echo $application->handle()->getContent();
```

2.5.6 Dependency Injection

Look at the second line of the code block above, the variable $application is receiving another variable $di. What is the purpose of that variable? Phalcon is a highly decoupled framework, so we need a component that acts as glue to make everything work together. That component is Phalcon\DI. It is a service container that also performs dependency injection, instantiating all components, as they are needed by the application.

There are many ways of registering services in the container. In INVO most services have been registered using anonymous functions. Thanks to this, the objects are instantiated in a lazy way, reducing the resources needed by the application.

For instance, in the following excerpt is registered the session service, the anonymous function will only be called when the application requires access to the session data:

```
<?php

//Start the session the first time when some component request the session service
$di->set('session', function() {
    $session = new Phalcon\Session\Adapter\Files();
    $session->start();
    return $session;
});
```

Here we have the freedom to change the adapter, perform additional initialization and much more. Note that the service was registered using the name "session". This is a convention that will allow the framework to identify the active service in the services container.

A request can use many services, register each service one to one can be a cumbersome task. For that reason, the framework provides a variant of Phalcon\DI called Phalcon\DI\FactoryDefault whose task is to register all services providing a full-stack framework.

```
<?php

// The FactoryDefault Dependency Injector automatically registers the
// right services providing a full stack framework
$di = new \Phalcon\DI\FactoryDefault();
```

It registers the majority of services with components provided by the framework as standard. If we need to override the definition of some service we could just set it again as we did above with "session". This is the reason for the existence of the variable $di.

2.5.7 Log into the Application

Log in will allow us to work on backend controllers. The separation between backend's controllers and the frontend ones is only logical. All controllers are located in the same directory.

To enter into the system, we must have a valid username and password. Users are stored in the table "users" in the database "invo".

Before we can start session, we need to configure the connection to the database in the application. A service called "db" is set up in the service container with that information. As with the autoloader, this time we are also taking parameters from the configuration file to configure a service:

```
<?php

// Database connection is created based on the parameters defined in the configuration file
$di->set('db', function() use ($config) {
    return new \Phalcon\Db\Adapter\Pdo\Mysql(array(
        "host" => $config->database->host,
        "username" => $config->database->username,
        "password" => $config->database->password,
        "dbname" => $config->database->name
    ));
});
```

Here we return an instance of the MySQL connection adapter. If needed, you could do extra actions such as adding a logger, a profiler or change the adapter, setting up it as you want.

Back then, the following simple form (app/views/session/index.phtml) requests the logon information. We've removed some HTML code to make the example more concise:

```
<?php echo Tag::form('session/start') ?>

    <label for="email">Username/Email</label>
    <?php echo Tag::textField(array("email", "size" => "30")) ?>

    <label for="password">Password</label>
    <?php echo Tag::passwordField(array("password", "size" => "30")) ?>

    <?php echo Tag::submitButton(array('Login')) ?>

</form>
```

The SessionController::startAction (app/controllers/SessionController.phtml) has the task of validate the data entered checking for a valid user in the database:

```
<?php

class SessionController extends ControllerBase
{

    // ...

    private function _registerSession($user)
    {
        $this->session->set('auth', array(
            'id' => $user->id,
            'name' => $user->name
        ));
    }

    public function startAction()
    {
        if ($this->request->isPost()) {

            //Taking the variables sent by POST
```

```
            $email = $this->request->getPost('email', 'email');
            $password = $this->request->getPost('password');

            $password = sha1($password);

            //Find for the user in the database
            $user = Users::findFirst(array(
                "email = :email: AND password = :password: AND active = 'Y'",
                "bind" => array('email' => $email, 'password' => $password)
            ));
            if ($user != false) {

                $this->_registerSession($user);

                $this->flash->success('Welcome '.$user->name);

                //Forward to the 'invoices' controller if the user is valid
                return $this->dispatcher->forward(array(
                    'controller' => 'invoices',
                    'action' => 'index'
                ));
            }

            $this->flash->error('Wrong email/password');
        }

        //Forward to the login form again
        return $this->dispatcher->forward(array(
            'controller' => 'session',
            'action' => 'index'
        ));

    }

}
```

For simplicity, we have used "sha1" to store the password hashes in the database, however, this algorithm is not recommended in real applications, use " *bcrypt*" instead.

Note that multiple public attributes are accessed in the controller like: $this->flash, $this->request or $this->session. These are services defined in services container from earlier. When they're accessed the first time, are injected as part of the controller.

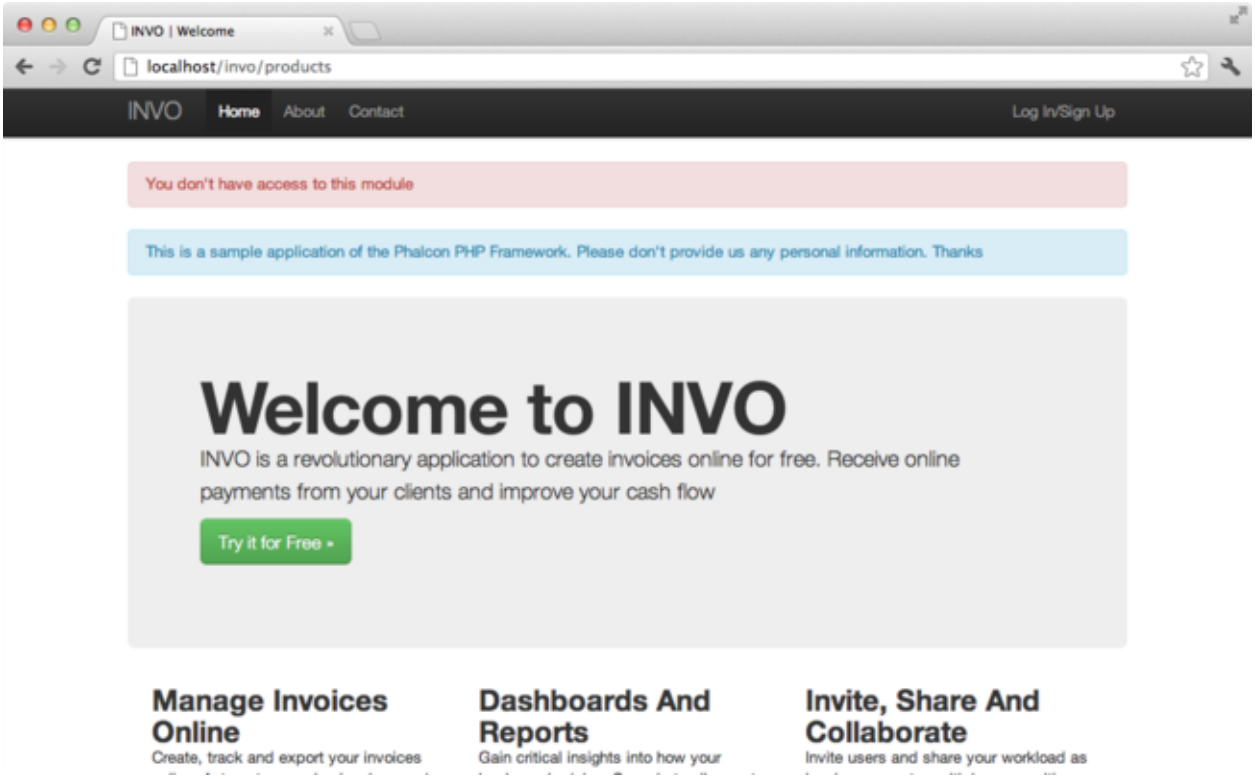
These services are shared, which means that we are always accessing the same instance regardless of the place where we invoke them.

For instance, here we invoke the "session" service and then we stored the user identity in the "auth" variable:

```
<?php

$this->session->set('auth', array(
    'id' => $user->id,
    'name' => $user->name
));
```

2.5.8 Securing the Backend

The backend is a private area where only registered users have access. Therefore, it is necessary to check that only registered users have access to these controllers. If you aren't logged in the application and you try to access, for example, the products controller (that is private) you will see a screen like this:



Every time someone attempts to access any controller/action, the application verifies that the current role (in session) has access to it, otherwise it displays a message like the above and forwards the flow to the home page.

Now let's find out how the application accomplishes this. The first thing to know is that there is a component called *Dispatcher*. It is informed about the route found by the *Routing* component. Then, it is responsible for loading the appropriate controller and execute the corresponding action method.

Normally, the framework creates the Dispatcher automatically. In our case, we want to perform a verification before executing the required action, checking if the user has access to it or not. To achieve this, we have replaced the component by creating a function in the bootstrap:

```
<?php

$di->set('dispatcher', function() use ($di) {
    $dispatcher = new Phalcon\Mvc\Dispatcher();
    return $dispatcher;
});
```

We now have total control over the Dispatcher used in the application. Many components in the framework launch events that allow us to modify the internal flow of operation. As the dependency Injector component acts as glue for components, a new component called *EventsManager* aids us to intercept the events produced by a component routing the events to listeners.

Events Management

A *EventsManager* allows us to attach listeners to a particular type of event. The type that interest us now is "dispatch", the following code filters all events produced by the Dispatcher:

```
<?php

$di->set('dispatcher', function() use ($di) {

    //Obtain the standard eventsManager from the DI
    $eventsManager = $di->getShared('eventsManager');

    //Instantiate the Security plugin
    $security = new Security($di);

    //Listen for events produced in the dispatcher using the Security plugin
    $eventsManager->attach('dispatch', $security);

    $dispatcher = new Phalcon\Mvc\Dispatcher();

    //Bind the EventsManager to the Dispatcher
    $dispatcher->setEventsManager($eventsManager);

    return $dispatcher;
});
```

The Security plugin is a class located at (app/plugins/Security.php). This class implements the method "beforeExecuteRoute". This is the same name as one of the events produced in the Dispatcher:

```
<?php

use \Phalcon\Events\Event;
use \Phalcon\Mvc\Dispatcher;

class Security extends Phalcon\Mvc\User\Plugin
{

    // ...

    public function beforeExecuteRoute(Event $event, Dispatcher $dispatcher)
    {
        // ...
    }

}
```

The hooks events always receive a first parameter that contains contextual information of the event produced and a second one that is the object that produced the event itself. It is not mandatory that plugins extend the class Phalcon\Mvc\User\Plugin, but by doing it they gain easier access to the services in the application.

Now, we're verifying the role in the current session, check to see if he/she has access using the ACL list. If he/she does not have access we redirect him/her to the home screen as explained before:

```
<?php

use \Phalcon\Events\Event;
use \Phalcon\Mvc\Dispatcher;

class Security extends Phalcon\Mvc\User\Plugin
```

```
{

    // ...

    public function beforeExecuteRoute(Event $event, Dispatcher $dispatcher)
    {

        //Check whether the "auth" variable exists in session to define the active role
        $auth = $this->session->get('auth');
        if (!$auth) {
            $role = 'Guests';
        } else {
            $role = 'Users';
        }

        //Take the active controller/action from the dispatcher
        $controller = $dispatcher->getControllerName();
        $action = $dispatcher->getActionName();

        //Obtain the ACL list
        $acl = $this->_getAcl();

        //Check if the Role have access to the controller (resource)
        $allowed = $acl->isAllowed($role, $controller, $action);
        if ($allowed != Phalcon\Acl::ALLOW) {

            //If he doesn't have access forward him to the index controller
            $this->flash->error("You don't have access to this module");
            $dispatcher->forward(
                array(
                    'controller' => 'index',
                    'action' => 'index'
                )
            );

            //Returning "false" we tell to the dispatcher to stop the current operation
            return false;
        }

    }

}
```

Providing an ACL list

In the previous example we have obtained the ACL using the method $this->_getAcl(). This method is also implemented in the Plugin. Now we are going to explain step-by-step how we built the access control list:

```
<?php

//Create the ACL
$acl = new Phalcon\Acl\Adapter\Memory();

//The default action is DENY access
$acl->setDefaultAction(Phalcon\Acl::DENY);

//Register two roles, Users is registered users
```

```
//and guests are users without a defined identity
$roles = array(
    'users' => new Phalcon\Acl\Role('Users'),
    'guests' => new Phalcon\Acl\Role('Guests')
);
foreach ($roles as $role) {
    $acl->addRole($role);
}
```

Now we define the resources for each area respectively. Controller names are resources and their actions are accesses for the resources:

```
<?php

//Private area resources (backend)
$privateResources = array(
    'companies' => array('index', 'search', 'new', 'edit', 'save', 'create', 'delete'),
    'products' => array('index', 'search', 'new', 'edit', 'save', 'create', 'delete'),
    'producttypes' => array('index', 'search', 'new', 'edit', 'save', 'create', 'delete'),
    'invoices' => array('index', 'profile')
);
foreach ($privateResources as $resource => $actions) {
    $acl->addResource(new Phalcon\Acl\Resource($resource), $actions);
}

//Public area resources (frontend)
$publicResources = array(
    'index' => array('index'),
    'about' => array('index'),
    'session' => array('index', 'register', 'start', 'end'),
    'contact' => array('index', 'send')
);
foreach ($publicResources as $resource => $actions) {
    $acl->addResource(new Phalcon\Acl\Resource($resource), $actions);
}
```

The ACL now have knowledge of the existing controllers and their related actions. Role "Users" has access to all the resources of both frontend and backend. The role "Guests" only has access to the public area:

```
<?php

//Grant access to public areas to both users and guests
foreach ($roles as $role) {
    foreach ($publicResources as $resource => $actions) {
        $acl->allow($role->getName(), $resource, '*');
    }
}

//Grant access to private area only to role Users
foreach ($privateResources as $resource => $actions) {
    foreach ($actions as $action) {
        $acl->allow('Users', $resource, $action);
    }
}
```

Hooray!, the ACL is now complete.

2.5.9 User Components

All the UI elements and visual style of the application has been achieved mostly through Twitter Bootstrap. Some elements, such as the navigation bar changes according to the state of the application. For example, in the upper right corner, the link "Log in / Sign Up" changes to "Log out" if a user is logged into the application.

This part of the application is implemented in the component "Elements" (app/library/Elements.php).

```
<?php

class Elements extends Phalcon\Mvc\User\Component
{

    public function getMenu()
    {
        //...
    }

    public function getTabs()
    {
        //...
    }

}
```

This class extends the Phalcon\Mvc\User\Component, it is not imposed to extend a component with this class, but if it helps to more quickly access the application services. Now, we register this class in the services container:

```
<?php

//Register an user component
$di->set('elements', function(){
    return new Elements();
});
```

As controllers, plugins or components within a view, this component also has access to the services registered in the container and by just accessing an attribute with the same name as a previously registered service:

```
<div class="navbar navbar-fixed-top">
    <div class="navbar-inner">
        <div class="container">
            <a class="btn btn-navbar" data-toggle="collapse" data-target=".nav-collapse">
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </a>
            <a class="brand" href="#">INVO</a>
            <?php echo $this->elements->getMenu() ?>
        </div>
    </div>
</div>

<div class="container">
    <?php echo $this->getContent() ?>
    <hr>
    <footer>
        <p>&copy; Company 2012</p>
    </footer>
</div>
```

The important part is:

```
<?php echo $this->elements->getMenu() ?>
```

2.5.10 Working with the CRUD

Most options that manipulate data (companies, products and types of products), were developed using a basic and common CRUD (Create, Read, Update and Delete). Each CRUD contains the following files:

```
invo/
    app/
        app/controllers/
            ProductsController.php
        app/models/
            Products.php
        app/views/
            products/
                edit.phtml
                index.phtml
                new.phtml
                search.phtml
```

Each controller has the following actions:

```
<?php

class ProductsController extends ControllerBase
{

    /**
     * The start action, it shows the "search" view
     */
    public function indexAction()
    {
        //...
    }

    /**
     * Execute the "search" based on the criteria sent from the "index"
     * Returning a paginator for the results
     */
    public function searchAction()
    {
        //...
    }

    /**
     * Shows the view to create a "new" product
     */
    public function newAction()
    {
        //...
    }

    /**
     * Shows the view to "edit" an existing product
     */
    public function editAction()
```

```
    {
        //...
    }

    /**
     * Creates a product based on the data entered in the "new" action
     */
    public function createAction()
    {
        //...
    }

    /**
     * Updates a product based on the data entered in the "edit" action
     */
    public function saveAction()
    {
        //...
    }

    /**
     * Deletes an existing product
     */
    public function deleteAction($id)
    {
        //...
    }

}
```

The Search Form

Every CRUD starts with a search form. This form shows each field that has the table (products), allowing the user creating a search criteria from any field. The "products" table has a relationship to the table "products_types". In this case, we previously queried the records in this table in order to facilitate the search by that field:

```
<?php

/**
 * The start action, it shows the "search" view
 */
public function indexAction()
{
    $this->persistent->searchParams = null;
    $this->view->setVar("productTypes", ProductTypes::find());
}
```

All the "product types" are queried and passed to the view as a local variable "productTypes". Then, in the view (app/views/index.phtml) we show a "select" tag filled with those results:

```
<?php

<div>
    <label for="product_types_id">Product Type</label>
    <?php echo Tag::select(array(
        "product_types_id",
        $productTypes,
```

```
        "using" => array("id", "name"),
        "useDummy" => true
    )) ?>
</div>
```

Note that $productTypes contains the data necessary to fill the SELECT tag using Phalcon\Tag::select. Once the form is submitted, the action "search" is executed in the controller performing the search based on the data entered by the user.

Performing a Search

The action "search" has a dual behavior. When accessed via POST, it performs a search based on the data sent from the form. But when accessed via GET it moves the current page in the paginator. To differentiate one from another HTTP method, we check it using the *Request* component:

```
<?php

/**
 * Execute the "search" based on the criteria sent from the "index"
 * Returning a paginator for the results
 */
public function searchAction()
{

    if ($this->request->isPost()) {
        //create the query conditions
    } else {
        //paginate using the existing conditions
    }

    //...

}
```

With the help of *Phalcon\Mvc\Model\Criteria*, we can create the search conditions intelligently based on the data types and values sent from the form:

```
<?php

$query = Criteria::fromInput($this->di, "Products", $_POST);
```

This method verifies which values are different from "" (empty string) and null and takes them into account to create the query:

- If the field data type is text or similar (char, varchar, text, etc.) It uses an SQL "like" operator to filter the results.
- If the data type is not text or similar, it'll use the operator "=".

Additionally, "Criteria" ignores all the $_POST variables that do not match any field in the table. Values are automatically escaped using "bound parameters".

Now, we store the produced parameters in the controller's session bag:

```
<?php

$this->persistent->searchParams = $query->getParams();
```

A session bag, is a special attribute in a controller that persists between requests. When accessed, this attribute injects a *Phalcon\Session\Bag* service that is independent in each controller.

Then, based on the built params we perform the query:

```
<?php

$products = Products::find($parameters);
if (count($products) == 0) {
    $this->flash->notice("The search did not found any products");
    return $this->forward("products/index");
}
```

If the search doesn't return any product, we forward the user to the index action again. Let's pretend the search returned results, then we create a paginator to navigate easily through them:

```
<?php

$paginator = new Phalcon\Paginator\Adapter\Model(array(
    "data" => $products,    //Data to paginate
    "limit" => 5,           //Rows per page
    "page" => $numberPage   //Active page
));

//Get active page in the paginator
$page = $paginator->getPaginate();
```

Finally we pass the returned page to view:

```
<?php

$this->view->setVar("page", $page);
```

In the view (app/views/products/search.phtml), we traverse the results corresponding to the current page:

```
<?php foreach ($page->items as $product) { ?>
    <tr>
        <td><?= $product->id ?></td>
        <td><?= $product->getProductTypes()->name ?></td>
        <td><?= $product->name ?></td>
        <td><?= $product->price ?></td>
        <td><?= $product->active ?></td>
        <td><?= Tag::linkTo("products/edit/" . $product->id, 'Edit') ?></td>
        <td><?= Tag::linkTo("products/delete/" . $product->id, 'Delete') ?></td>
    </tr>
<?php } ?>
```

Creating and Updating Records

Now let's see how the CRUD creates and updates records. From the "new" and "edit" views the data entered by the user are sent to the actions "create" and "save" that perform actions of "creating" and "updating" products respectively.

In the creation case, we recover the data submitted and assign them to a new "products" instance:

```
<?php

/**
 * Creates a product based on the data entered in the "new" action
 */
public function createAction()
{
```

```
    $products = new Products();
    $products->id = $request->getPost("id", "int");
    $products->product_types_id = $request->getPost("product_types_id", "int");
    $products->name = $request->getPost("name", "striptags");
    $products->price = $request->getPost("price", "double");
    $products->active = $request->getPost("active");

    //...

}
```

Data is filtered before being assigned to the object. This filtering is optional, the ORM escapes the input data and performs additional casting according to the column types.

When saving we'll know whether the data conforms to the business rules and validations implemented in the model Products:

```
<?php

/**
 * Creates a product based on the data entered in the "new" action
 */
public function createAction()
{

    //...

    if (!$products->create()) {

        //The store failed, the following messages were produced
        foreach ($products->getMessages() as $message) {
            $this->flash->error((string) $message);
        }
        return $this->forward("products/new");

    } else {
        $this->flash->success("Product was created successfully");
        return $this->forward("products/index");
    }

}
```

Now, in the case of product updating, first we must present to the user the data that is currently in the edited record:

```
<?php

/**
 * Shows the view to "edit" an existing product
 */
public function editAction($id)
{

    //...

    $product = Products::findFirst(array(
        'id = ?0',
        'bind' => array($id)
    ));
```

```
    Tag::displayTo("id", $product->id);
    Tag::displayTo("product_types_id", $product->product_types_id);
    Tag::displayTo("name", $product->name);
    Tag::displayTo("price", $product->price);
    Tag::displayTo("active", $product->active);

}
```

The displayTo helper sets a default value in the form on the attribute with the same name. Thanks to this, the user can change any value and then sent it back to the database through to the "save" action:

```
<?php

/**
 * Updates a product based on the data entered in the "edit" action
 */
public function saveAction()
{

    //...

    //Find the product to update
    $product = Products::findFirst(array(
        'id = ?0',
        'bind' => array($this->request->getPost("id"))
    ));
    if (!$product) {
        $this->flash->error("products does not exist ".$id);
        return $this->forward("products/index");
    }

    //... assign the values to the object and store it

}
```

2.5.11 Changing the Title Dynamically

When you browse between one option and another will see that the title changes dynamically indicating where we are currently working. This is achieved in each controller initializer:

```
<?php

class ProductsController extends ControllerBase
{

    public function initialize()
    {
        //Set the document title
        Tag::setTitle('Manage your product types');
        parent::initialize();
    }

    //...

}
```

Note, that the method parent::initialize() is also called, it adds more data to the title:

```
<?php

class ControllerBase extends Phalcon\Mvc\Controller
{

    protected function initialize()
    {
        //Prepend the application name to the title
        Phalcon\Tag::prependTitle('INVO | ');
    }

    //...
}
```

Finally, the title is printed in the main view (app/views/index.phtml):

```
<?php use Phalcon\Tag as Tag ?>
<!DOCTYPE html>
<html>
    <head>
        <?php echo Tag::getTitle() ?>
    </head>
    <!-- ... -->
</html>
```

2.5.12 Conclusion

This tutorial covers many more aspects of building applications with Phalcon, hope you have served to learn more and get more out of the framework.

2.6 Tutorial 3: Creating a Simple REST API

In this tutorial, we will explain how to create a simple application that provides a RESTful API using the different HTTP methods:

- GET to retrieve and search data
- POST to add data
- PUT to update data
- DELETE to delete data

2.6.1 Defining the API

The API consists of the following methods:

Method	URL	Action
GET	/api/robots	Retrieves all robots
GET	/api/robots/search/Astro	Searches for robots with 'Astro' in their name
GET	/api/robots/2	Retrieves robots based on primary key
POST	/api/robots	Adds a new robot
PUT	/api/robots/2	Updates robots based on primary key
DELETE	/api/robots/2	Deletes robots based on primary key

2.6.2 Creating the Application

As the application is so simple, we will not implement any full MVC environment to develop it. In this case, we will use a *micro application* to meet our goal.

The following file structure is more than enough:

```
my-rest-api/
    models/
        Robots.php
    index.php
    .htaccess
```

First, we need an .htaccess file that contains all the rules to rewrite the URIs to the index.php file, that is our application:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php?_url=/$1 [QSA,L]
</IfModule>
```

Then, in the index.php file we create the following:

```
<?php

$app = new \Phalcon\Mvc\Micro();

//define the routes here

$app->handle();
```

Now we will create the routes as we defined above:

```
<?php

$app = new Phalcon\Mvc\Micro();

//Retrieves all robots
$app->get('/api/robots', function() {

});

//Searches for robots with $name in their name
$app->get('/api/robots/search/{name}', function($name) {

});

//Retrieves robots based on primary key
$app->get('/api/robots/{id:[0-9]+}', function($id) {

});

//Adds a new robot
$app->post('/api/robots', function() {

});

//Updates robots based on primary key
$app->put('/api/robots/{id:[0-9]+}', function() {
```

```
});

//Deletes robots based on primary key
$app->delete('/api/robots/{id:[0-9]+}', function() {

});

$app->handle();
```

Each route is defined with a method with the same name as the HTTP method, as first parameter we pass a route pattern, followed by a handler. In this case, the handler is an anonymous function. The following route: '/api/robots/{id:[0-9]+}', by example, explicitly sets that the "id" parameter must have a numeric format.

When a defined route matches the requested URI then the application executes the corresponding handler.

2.6.3 Creating a Model

Our API provides information about 'robots', these data are stored in a database. The following model allows us to access that table in an object-oriented way. We have implemented some business rules using built-in validators and simple validations. Doing this will give us the peace of mind that saved data meet the requirements of our application:

```
<?php

use \Phalcon\Mvc\Model\Message;
use \Phalcon\Mvc\Model\Validator\InclusionIn;
use \Phalcon\Mvc\Model\Validator\Uniqueness;

class Robots extends \Phalcon\Mvc\Model
{

    public function validation()
    {
        //Type must be: droid, mechanical or virtual
        $this->validate(new InclusionIn(
            array(
                "field"  => "type",
                "domain" => array("droid", "mechanical", "virtual")
            )
        ));

        //Robot name must be unique
        $this->validate(new Uniqueness(
            array(
                "field"   => "name",
                "message" => "The robot name must be unique"
            )
        ));

        //Year cannot be less than zero
        if ($this->year < 0) {
            $this->appendMessage(new Message("The year cannot be less than zero"));
        }

        //Check if any messages have been produced
        if ($this->validationHasFailed() == true) {
            return false;
        }
```

```
    }

}
```

Now, we must set up a connection to be used by this model:

```
<?php

$di = new \Phalcon\DI\FactoryDefault();

//Set up the database service
$di->set('db', function(){
    return new \Phalcon\Db\Adapter\Pdo\Mysql(array(
        "host" => "localhost",
        "username" => "asimov",
        "password" => "zeroth",
        "dbname" => "robotics"
    ));
});

$app = new \Phalcon\Mvc\Micro();

//Bind the DI to the application
$app->setDI($di);
```

2.6.4 Retrieving Data

The first "handler" that we will implement is which by method GET returns all available robots. Let's use PHQL to perform this simple query returning the results as JSON:

```
<?php

//Retrieves all robots
$app->get('/api/robots', function() use ($app) {

    $phql = "SELECT * FROM Robots ORDER BY name";
    $robots = $app->modelsManager->executeQuery($phql);

    $data = array();
    foreach($robots as $robot){
        $data[] = array(
            'id' => $robot->id,
            'name' => $robot->name,
        );
    }

    echo json_encode($data);
});
```

PHQL, allow us to write queries using a high-level, object-oriented SQL dialect that internally translates to the right SQL statements depending on the database system we are using. The clause "use" in the anonymous function allows us to pass some variables from the global to local scope easily.

The searching by name handler would look like:

```
<?php

//Searches for robots with $name in their name
```

```
$app->get('/api/robots/search/{name}', function($name) use ($app) {

    $phql = "SELECT * FROM Robots WHERE name LIKE :name: ORDER BY name";
    $robots = $app->modelsManager->executeQuery($phql, array(
        'name' => '%'.$name.'%'
    ));

    $data = array();
    foreach($robots as $robot){
        $data[] = array(
            'id' => $robot->id,
            'name' => $robot->name,
        );
    }

    echo json_encode($data);

});
```

Searching by the field "id" it's quite similar, in this case, we're also notifying if the robot was found or not:

```
<?php

//Retrieves robots based on primary key
$app->get('/api/robots/{id:[0-9]+}', function($id) use ($app) {

    $phql = "SELECT * FROM Robots WHERE id = :id:";
    $robot = $app->modelsManager->executeQuery($phql, array(
        'id' => $id
    ))->getFirst();

    if ($robot==false) {
        $response = array('status' => 'NOT-FOUND');
    } else {
        $response = array(
            'status' => 'FOUND',
            'data' => array(
                'id' => $robot->id,
                'name' => $robot->name
            )
        );
    }

    echo json_encode($response);
});
```

2.6.5 Inserting Data

Taking the data as a JSON string inserted in the body of the request, we also use PHQL for insertion:

```
<?php

//Adds a new robot
$app->post('/api/robots', function() use ($app) {

    $robot = json_decode($app->request->getRawBody());
```

```
    $phql = "INSERT INTO Robots (name, type, year) VALUES (:name:, :type:, :year:)";

    $status = $app->modelsManager->executeQuery($phql, array(
        'name' => $robot->name,
        'type' => $robot->type,
        'year' => $robot->year
    ));

    //Check if the insertion was successfull
    if($status->success()==true){

        $robot->id = $status->getModel()->id;

        $response = array('status' => 'OK', 'data' => $robot);

    } else {

        //Change the HTTP status
        $this->response->setStatusCode(500, "Internal Error")->sendHeaders();

        //Send errors to the client
        $errors = array();
        foreach ($status->getMessages() as $message) {
            $errors[] = $message->getMessage();
        }

        $response = array('status' => 'ERROR', 'messages' => $errors);

    }

    echo json_encode($response);

});
```

2.6.6 Updating Data

The data update is similar to insertion. The "id" passed as parameter indicates what robot must be updated:

```
<?php

//Updates robots based on primary key
$app->put('/api/robots/{id:[0-9]+}', function($id) use($app) {

    $robot = json_decode($app->request->getRawBody());

    $phql = "UPDATE Robots SET name = :name:, type = :type:, year = :year: WHERE id = :id:";
    $status = $app->modelsManager->executeQuery($phql, array(
        'id' => $id,
        'name' => $robot->name,
        'type' => $robot->type,
        'year' => $robot->year
    ));

    //Check if the insertion was successfull
    if($status->success()==true){

        $response = array('status' => 'OK');
```

```
    } else {

        //Change the HTTP status
        $this->response->setStatusCode(500, "Internal Error")->sendHeaders();

        $errors = array();
        foreach ($status->getMessages() as $message) {
            $errors[] = $message->getMessage();
        }

        $response = array('status' => 'ERROR', 'messages' => $errors);

    }

    echo json_encode($response);

});
```

2.6.7 Deleting Data

The data delete is similar to update. The "id" passed as parameter indicates what robot must be deleted:

```
<?php

//Deletes robots based on primary key
$app->delete('/api/robots/{id:[0-9]+}', function($id) use ($app) {

    $phql = "DELETE FROM Robots WHERE id = :id:";
    $status = $app->modelsManager->executeQuery($phql, array(
        'id' => $id
    ));
    if($status->success()==true){

        $response = array('status' => 'OK');

    } else {

        //Change the HTTP status
        $this->response->setStatusCode(500, "Internal Error")->sendHeaders();

        $errors = array();
        foreach ($status->getMessages() as $message) {
            $errors[] = $message->getMessage();
        }

        $response = array('status' => 'ERROR', 'messages' => $errors);

    }

    echo json_encode($response);

});
```

2.6.8 Testing our Application

Using curl we'll test every route in our application verifying its proper operation:

Obtain all the robots:

```
curl -i -X GET http://localhost/my-rest-api/api/robots

HTTP/1.1 200 OK
Date: Wed, 12 Sep 2012 07:05:13 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 117
Content-Type: text/html; charset=UTF-8

[{"id":"1","name":"Robotina"},{"id":"2","name":"Astro Boy"},{"id":"3","name":"Terminator"}]
```

Search a robot by its name:

```
curl -i -X GET http://localhost/my-rest-api/api/robots/search/Astro

HTTP/1.1 200 OK
Date: Wed, 12 Sep 2012 07:09:23 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 31
Content-Type: text/html; charset=UTF-8

[{"id":"2","name":"Astro Boy"}]
```

Obtain a robot by its id:

```
curl -i -X GET http://localhost/my-rest-api/api/robots/3

HTTP/1.1 200 OK
Date: Wed, 12 Sep 2012 07:12:18 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 56
Content-Type: text/html; charset=UTF-8

{"status":"FOUND","data":{"id":"3","name":"Terminator"}}
```

Insert a new robot:

```
curl -i -X POST -d '{"name":"C-3PO","type":"droid","year":1977}'
    http://localhost/my-rest-api/api/robots

HTTP/1.1 200 OK
Date: Wed, 12 Sep 2012 07:15:09 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 75
Content-Type: text/html; charset=UTF-8

{"status":"OK","data":{"name":"C-3PO","type":"droid","year":1977,"id":"4"}}
```

Try to insert a new robot with the name of an existing robot:

```
curl -i -X POST -d '{"name":"C-3PO","type":"droid","year":1977}'
    http://localhost/my-rest-api/api/robots

HTTP/1.1 500 Internal Error
Date: Wed, 12 Sep 2012 07:18:28 GMT
```

```
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 63
Content-Type: text/html; charset=UTF-8

{"status":"ERROR","messages":["The robot name must be unique"]}
```

Or update a robot with an unknown type:

```
curl -i -X PUT -d '{"name":"ASIMO","type":"humanoid","year":2000}'
    http://localhost/my-rest-api/api/robots/4

HTTP/1.1 500 Internal Error
Date: Wed, 12 Sep 2012 08:48:01 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 104
Content-Type: text/html; charset=UTF-8

{"status":"ERROR","messages":["Value of field 'type' must be part of
    list: droid, mechanical, virtual"]}
```

Finally, delete a robot:

```
curl -i -X DELETE http://localhost/my-rest-api/api/robots/4

HTTP/1.1 200 OK
Date: Wed, 12 Sep 2012 08:49:29 GMT
Server: Apache/2.2.22 (Unix) DAV/2
Content-Length: 15
Content-Type: text/html; charset=UTF-8

{"status":"OK"}
```

2.6.9 Conclusion

As we have seen, develop a RESTful API with Phalcon is easy. Later in the documentation we'll explain in detail how to use micro applications and the *PHQL* language.

2.7 Using Dependency Injection

The following example is a bit lengthy, but explains why using a service container, service location and dependency injection. First, let's pretend we are developing a component called SomeComponent. This performs a task that is not important now. Our component has some dependency that is a connection to a database.

In this first example, the connection is created inside the component. This approach is impractical; practically we cannot change the connection parameters or the type of database system because the component only works as created.

```
<?php

class SomeComponent
{

    /**
     * The instantiation of the connection is hardcoded inside
     * the component so is difficult replacing it externally
     * or change its behavior
```

```
     */
    public function someDbTask()
    {
        $connection = new Connection(array(
            "host" => "localhost",
            "username" => "root",
            "password" => "secret",
            "dbname" => "invo"
        ));

        // ...
    }

}

$some = new SomeComponent();
$some->someDbTask();
```

To solve this, we create a setter that injects the dependency externally before using it. For now, this seems to be a good solution:

```
<?php

class SomeComponent
{

    protected $_connection;

    /**
     * Sets the connection externally
     */
    public function setConnection($connection)
    {
        $this->_connection = $connection;
    }

    public function someDbTask()
    {
        $connection = $this->_connection;

        // ...
    }

}

$some = new SomeComponent();

//Create the connection
$connection = new Connection(array(
    "host" => "localhost",
    "username" => "root",
    "password" => "secret",
    "dbname" => "invo"
));

//Inject the connection in the component
$some->setConnection($connection);
```

```
$some->someDbTask();
```

Now consider that we use this component in different parts of the application and then we will need to create the connection several times before passing it to the component. Using some kind of global registry where we obtain the connection instance and not have to create it again and again could solve this:

```
<?php

class Registry
{

    /**
     * Returns the connection
     */
    public static function getConnection()
    {
        return new Connection(array(
             "host" => "localhost",
             "username" => "root",
             "password" => "secret",
             "dbname" => "invo"
        ));
    }

}

class SomeComponent
{

    protected $_connection;

    /**
     * Sets the connection externally
     */
    public function setConnection($connection){
        $this->_connection = $connection;
    }

    public function someDbTask()
    {
        $connection = $this->_connection;

        // ...
    }

}

$some = new SomeComponent();

//Pass the connection defined in the registry
$some->setConnection(Registry::getConnection());

$some->someDbTask();
```

Now, let's imagine that we must implement two methods in the component, the first always need to create a new connection and the second always need to use a shared connection:

```php
<?php

class Registry
{

    protected static $_connection;

    /**
     * Creates a connection
     */
    protected static function _createConnection()
    {
        return new Connection(array(
            "host" => "localhost",
            "username" => "root",
            "password" => "secret",
            "dbname" => "invo"
        ));
    }

    /**
     * Creates a connection only once and returns it
     */
    public static function getSharedConnection()
    {
        if (self::$_connection===null){
            $connection = self::_createConnection();
            self::$_connection = $connection;
        }
        return self::$_connection;
    }

    /**
     * Always returns a new connection
     */
    public static function getNewConnection()
    {
        return self::_createConnection();
    }

}

class SomeComponent
{

    protected $_connection;

    /**
     * Sets the connection externally
     */
    public function setConnection($connection){
        $this->_connection = $connection;
    }

    /**
     * This method always needs the shared connection
     */
    public function someDbTask()
```

```
    {
        $connection = $this->_connection;

        // ...
    }

    /**
     * This method always needs a new connection
     */
    public function someOtherDbTask($connection)
    {

    }

}

$some = new SomeComponent();

//This injects the shared connection
$some->setConnection(Registry::getSharedConnection());

$some->someDbTask();

//Here, we always pass a new connection as parameter
$some->someOtherDbTask(Registry::getConnection());
```

So far we have seen how dependency injection solved our problems. Passing dependencies as arguments instead of creating them internally in the code makes our application more maintainable and decoupled. However, to long-term, this form of dependency injection have some disadvantages.

For instance, if the component has many dependencies, we will need to create multiple setter arguments to pass the dependencies or create a constructor that pass them with many arguments, additionally creating dependencies before using the component, every time, makes our code not maintainable as we would like:

```
<?php

//Create the dependencies or retrieve them from the registry
$connection = new Connection();
$session = new Session();
$fileSystem = new FileSystem();
$filter = new Filter();
$selector = new Selector();

//Pass them as constructor parameters
$some = new SomeComponent($connection, $session, $fileSystem, $filter, $selector);

// ... or using setters

$some->setConnection($connection);
$some->setSession($session);
$some->setFileSystem($fileSystem);
$some->setFilter($filter);
$some->setSelector($selector);
```

Think we had to create this object in many parts of our application. If you ever do not require any of the dependencies, we need to go everywhere to remove the parameter in the constructor or the setter where we injected the code. To solve this, we return again to a global registry to create the component. However, it adds a new layer of abstraction before creating the object:

```php
<?php

class SomeComponent
{

    // ...

    /**
     * Define a factory method to create SomeComponent instances injecting its dependencies
     */
    public static function factory()
    {

        $connection = new Connection();
        $session = new Session();
        $fileSystem = new FileSystem();
        $filter = new Filter();
        $selector = new Selector();

        return new self($connection, $session, $fileSystem, $filter, $selector);
    }

}
```

One moment, we returned to the beginning, we are building again the dependencies inside the component! We can move on and find out a way to solve this problem every time. But it seems that time and again we fall back into bad practices.

A practical and elegant way to solve these problems are using a container for dependencies. The containers act as the global registry that we saw earlier. Using the container for dependencies as a bridge to obtain the dependencies allows us to reduce the complexity of our component:

```php
<?php

class SomeComponent
{

    protected $_di;

    public function __construct($di)
    {
        $this->_di = $di;
    }

    public function someDbTask()
    {

        // Get the connection service
        // Always returns a new connection
        $connection = $this->_di->get('db');

    }

    public function someOtherDbTask()
    {

        // Get a shared connection service,
        // this will return the same connection everytime
        $connection = $this->_di->getShared('db');
```

```
        //This method also requires a input filtering service
        $filter = $this->_db->get('filter');

    }

}

$di = new Phalcon\DI();

//Register a "db" service in the container
$di->set('db', function() {
    return new Connection(array(
        "host" => "localhost",
        "username" => "root",
        "password" => "secret",
        "dbname" => "invo"
    ));
});

//Register a "filter" service in the container
$di->set('filter', function() {
    return new Filter();
});

//Register a "session" service in the container
$di->set('session', function() {
    return new Session();
});

//Pass the service container as unique parameter
$some = new SomeComponent($di);

$some->someTask();
```

The component now simply access the service it requires when it needs it, if it does not require a service that is not even initialized saving resources. The component is now highly decoupled. For example, we can replace the manner in which connections are created, their behavior or any other aspect of them and that would not affect the component.

2.7.1 Our approach

Phalcon\DI is a component implementing Dependency Injection and Location of services and it's itself a container for them.

Since Phalcon is highly decoupled, Phalcon\DI is essential to integrate the different components of the framework. The developer can also use this component to inject dependencies and manage global instances of the different classes used in the application.

Basically, this component implements the Inversion of Control pattern. Applying this, the objects do not receive their dependencies using setters or constructors, but requesting a service dependency injector. This reduces the overall complexity since there is only one way to get the required dependencies within a component.

Additionally, this pattern increases testability in the code, thus making it less prone to errors.

2.7.2 Registering services in the Container

The framework itself or the developer can register services. When a component A requires component B (or an instance of its class) to operate, it can request component B from the container, rather than creating a new instance component B.

This way of working gives us many advantages:

- We can replace a component by one created by ourselves or a third party one easily.
- We have full control of the object initialization, allowing us to set these objects, as you need before delivering them to components.
- We can get global instances of components in a structured and unified way

Services can be registered using several types of definitions:

```
<?php

//Create the Dependency Injector Container
$di = new Phalcon\DI();

//By its class name
$di->set("request", 'Phalcon\Http\Request');

//Using an anonymous function, the instance will lazy loaded
$di->set("request", function() {
    return new Phalcon\Http\Request();
});

//Registering directly an instance
$di->set("request", new Phalcon\Http\Request());

//Using an array definition
$di->set("request", array(
    "className" => 'Phalcon\Http\Request'
));
```

The array syntax is also allowed to register services:

```
<?php

//Create the Dependency Injector Container
$di = new Phalcon\DI();

//By its class name
$di["request"] = 'Phalcon\Http\Request';

//Using an anonymous function, the instance will lazy loaded
$di["request"] = function() {
    return new Phalcon\Http\Request();
};

//Registering directly an instance
$di["request"] = new Phalcon\Http\Request();

//Using an array definition
$di["request"] = array(
    "className" => 'Phalcon\Http\Request'
);
```

In the examples given above, when the framework needs to access the request data, it will ask for the service identified as 'request' in the container. The container in turn will return an instance of the required service. A developer might eventually replace a component when he/she needs.

Each of the methods (demonstrated in the example given above) used to set/register a service has advantages and disadvantages. It is up to the developer and the particular requirements that will designate which one is used.

Setting a service by a string is simple, but lacks flexibility. Setting services using an array offers a lot more flexibility, but makes the code more complicated. The lambda function is a good balance between the two, but could lead to more maintenance than one would expect.

Phalcon\DI offers lazy loading for every service it stores. Unless the developer chooses to instantiate an object directly and store it in the container, any object stored in it (via array, string, etc.) will be lazy loaded i.e. instantiated only when requested.

Simple Registration

As seen before, there are several ways to register services. These are we call simple:

String

This type expects the name of a valid class, returning an object of the specified class, if the class is not loaded is loaded using an auto-loader. This type of definition does not allow to define arguments for the class constructor or parameters:

```
<?php

// return new Phalcon\Http\Request();
$di->set('request', 'Phalcon\Http\Request');
```

Object

This type expects an object because the object does not need to be resolved because it is already an object, one could say that there is not really dependency injection here, but it is useful if you want to force that the returned dependency will always the same object/value:

```
<?php

// return new Phalcon\Http\Request();
$di->set('request', new Phalcon\Http\Request());
```

Closures/Anonymous functions

This method offers greater freedom to build the dependency as desired, however, it is difficult to change some of the parameters externally without having to completely change the definition of dependency:

```
<?php

$di->set("db", function() {
    return new \Phalcon\Db\Adapter\Pdo\Mysql(array(
          "host" => "localhost",
          "username" => "root",
          "password" => "secret",
          "dbname" => "blog"
```

```
    ));
});
```

Some of the limitations can be overcome by passing additional variables to the closure's environment:

```
<?php

//Using the $config variable in the current scope
$di->set("db", function() use ($config) {
    return new \Phalcon\Db\Adapter\Pdo\Mysql(array(
         "host" => $config->host,
         "username" => $config->username,
         "password" => $config->password,
         "dbname" => $config->name
    ));
});
```

Complex Registration

If it is required to change the definition of a service without instantiating/resolving the service, then, we need to define the services using the array syntax. Define a service using an array definition can be a little more verbose:

```
<?php

//Register a service 'logger' with a class name and its parameters
$di->set('logger', array(
    'className' => 'Phalcon\Logger\Adapter\File',
    'arguments' => array(
        array(
            'type' => 'parameter',
            'value' => '../apps/logs/error.log'
        )
    )
));

//Using an anonymous function
$di->set('logger', function() {
    return new \Phalcon\Logger\Adapter\File('../apps/logs/error.log');
});
```

Both service registrations above produce the same result. The array definition however, allows for alteration of the service parameters if needed:

```
<?php

//Change the service class name
$di->getService('logger')->setClassName('MyCustomLogger');

//Change the first parameter without instantiate the logger
$di->getService('logger')->setParameter(0, array(
    'type' => 'parameter',
    'value' => '../apps/logs/error.log'
));
```

In addition by using the array syntax you can use three types of dependency injection:

Constructor Injection

This injection type passes the dependencies/arguments to the class constructor. Let's pretend we have the following component:

```
<?php

namespace SomeApp;

use Phalcon\Http\Response;

class SomeComponent
{

    protected $_response;

    protected $_someFlag;

    public function __construct(Response $response, $someFlag)
    {
        $this->_response = $response;
        $this->_someFlag = $someFlag;
    }

}
```

The service can be registered this way:

```
<?php

$di->set('response', array(
    'className' => 'Phalcon\Http\Response'
));

$di->set('someComponent', array(
    'className' => 'SomeApp\SomeComponent',
    'arguments' => array(
        array('type' => 'service', 'name' => 'response'),
        array('type' => 'parameter', 'value' => true)
    )
));
```

The service "response" (Phalcon\Http\Response) is resolved to be passed as the first argument of the constructor, while the second is a boolean value (true) that is passed as it is.

Setter Injection

Classes may have setters to inject optional dependencies, our previous class can be changed to accept the dependencies with setters:

```
<?php

namespace SomeApp;

use Phalcon\Http\Response;

class SomeComponent
```

```
{

    protected $_response;

    protected $_someFlag;

    public function setResponse(Response $response)
    {
        $this->_response = $response;
    }

    public function setFlag($someFlag)
    {
        $this->_someFlag = $someFlag;
    }

}
```

A service with setter injection can be registered as follows:

```
<?php

$di->set('response', array(
    'className' => 'Phalcon\Http\Response'
));

$di->set('someComponent', array(
    'className' => 'SomeApp\SomeComponent',
    'calls' => array(
        array(
            'method' => 'setResponse',
            'arguments' => array(
                array('type' => 'service', 'name' => 'response'),
            )
        ),
        array(
            'method' => 'setFlag',
            'arguments' => array(
                array('type' => 'parameter', 'value' => true)
            )
        )
    )
));
```

Properties Injection

A less common strategy is to inject dependencies or parameters directly in public attributes of the class:

```
<?php

namespace SomeApp;

use Phalcon\Http\Response;

class SomeComponent
{
```

```
    public $response;

    public $someFlag;

}
```

A service with properties injection can be registered as follows:

```
<?php

$di->set('response', array(
    'className' => 'Phalcon\Http\Response'
));

$di->set('someComponent', array(
    'className' => 'SomeApp\SomeComponent',
    'properties' => array(
        array(
            'name' => 'response',
            'value' => array('type' => 'service', 'name' => 'response')
        ),
        array(
            'name' => 'someFlag',
            'value' => array('type' => 'parameter', 'value' => true)
        )
    )
));
```

Supported parameter types include the following:

Type	Description	Example
parameter	Represents a literal value to be passed as parameter	array('type' => 'parameter', 'value' => 1234)
service	Represents another service in the services container	array('type' => 'service', 'name' => 'request')
instance	Represents a object that must be built dynamically	array('type' => 'service', 'className' => 'DateTime', 'arguments' => array('now'))

Resolving a service whose definition is complex may be slightly slower than previously seen simple definitions. However, these provide a more robust approach to define and inject services.

Mixing different types of definitions is allowed, everyone can decide what is the most appropriate way to register the services according to the application needs.

2.7.3 Resolving Services

Obtaining a service from the container is a matter of simply calling the "get" method. A new instance of the service will be returned:

```
<?php $request = $di->get("request");
```

Or by calling through the magic method:

```
<?php

$request = $di->getRequest();
```

Or using the array-access syntax:

```
<?php

$request = $di['request'];
```

Arguments can be passed to the constructor by adding an array parameter to the method "get":

```
<?php

// new MyComponent("some-parameter", "other")
$component = $di->get("MyComponent", array("some-parameter", "other"));
```

2.7.4 Shared services

Services can be registered as "shared" services this means that they always will act as singletons. Once the service is resolved for the first time the same instance it's returned every time a consumer retrieve the service from the container:

```
<?php

//Register the session service as "always shared"
$di->setShared('session', function() {
    $session = new Phalcon\Session\Adapter\Files();
    $session->start();
    return $session;
});

$session = $di->get('session'); // Locates the service for the first time
$session = $di->getSession(); // Returns the first instantiated object
```

An alternative way to register services is pass "true" as third parameter of "set":

```
<?php

//Register the session service as "always shared"
$di->set('session', function() {
    //...
}, true);
```

If a service isn't registered as shared and you want to be sure that a shared instance will be accessed every time the service is obtained from the DI, you can use the 'getShared' method:

```
<?php

$request = $di->getShared("request");
```

2.7.5 Manipulating services individually

Once a service is registered in services container, you can retrieve it to manipulate it individually:

```
<?php

//Register the session service as "always shared"
$di->set('request', 'Phalcon\Http\Request');

//Get the service
$requestService = $di->getService('request');
```

```
//Change its definition
$requestService->setDefinition(function() {
    return new Phalcon\Http\Request();
});

//Change it to shared
$request->setShared(true);

//Resolve the service (return a Phalcon\Http\Request instance)
$request = $requestService->resolve();
```

2.7.6 Instantiating classes via the Services Container

When you request a service to the services container, if it can't find out a service with the same name it'll try to load a class with the same name. With this behavior we can replace any class by another simply by registering a service with its name:

```
<?php

//Register a controller as a service
$di->set('IndexController', function() {
    $component = new Component();
    return $component;
}, true);

//Register a controller as a service
$di->set('MyOtherComponent', function() {
    //Actually returns another component
    $component = new AnotherComponent();
    return $component;
});

//Create a instance via the services container
$myComponent = $di->get('MyOtherComponent');
```

You can take advantage of this, always instantiating your classes via the services container (even if they aren't registered as services). The DI will fallback to a valid autoloader to finally load the class. By doing this, you can easily replace any class in the future by implementing a definition for it.

2.7.7 Automatic Injecting of the DI itself

If a class or component requires the DI itself to locate services, the DI can automatically inject itself to the instances creates by it, to do this, you need to implement the *Phalcon\DI\InjectionAwareInterface* in your classes:

```
<?php

class MyClass implements \Phalcon\DI\InjectionAwareInterface
{

    protected $_di;

    public function setDi($di)
    {
        $this->_di = $di;
    }
```

```
    public function getDi()
    {
        return $this->_di;
    }

}
```

Then once the service is resolved, the $di will be passed to setDi automatically:

```
<?php

//Register the service
$di->set('myClass', 'MyClass');

//Resolve the service (also $myClass->setDi($di) is automatically called)
$myClass = $di->get('myClass');
```

2.7.8 Avoiding service resolution

Some services are used in each of the requests made to the application, eliminate the process of resolving the service could add some small improvement in performance.

```
<?php

//Resolve the object externally instead of using a definition for it:
$router = new MyRouter();

//Pass the resolved object to the service registration
$di->set('router', $router);
```

2.7.9 Organizing services in files

You can better organize your application by moving the service registration to individual files instead of doing everything in the application's bootstrap:

```
<?php

$di->set('router', function() {
    return include ("../app/config/routes.php");
});
```

Then in the file ("../app/config/routes.php") return the object resolved:

```
<?php

$router = new MyRouter();

$router->post('/login');

return $router;
```

2.7.10 Accessing the DI in a static way

If needed you can access the latest DI created in a static function in the following way:

```php
<?php

class SomeComponent
{

    public static function someMethod()
    {
        //Get the session service
        $session = Phalcon\DI::getDefault()->getSession();
    }

}
```

2.7.11 Factory Default DI

Although the decoupled character of Phalcon offers us great freedom and flexibility, maybe we just simply want to use it as a full-stack framework. To achieve this, the framework provides a variant of Phalcon\DI called Phalcon\DI\FactoryDefault. This class automatically registers the appropriate services bundled with the framework to act as full-stack.

```php
<?php $di = new Phalcon\DI\FactoryDefault();
```

2.7.12 Service Name Conventions

Although you can register services with the names you want. Phalcon has a seriers of service naming conventions that allow it to get the right services when you need it requires them.

Service Name	Description	Default	Shared
dispatcher	Controllers Dispatching Service	*Phalcon\Mvc\Dispatcher*	Yes
router	Routing Service	*Phalcon\Mvc\Router*	Yes
url	URL Generator Service	*Phalcon\Mvc\Url*	Yes
request	HTTP Request Environment Service	*Phalcon\Http\Request*	Yes
response	HTTP Response Environment Service	*Phalcon\Http\Response*	Yes
filter	Input Filtering Service	*Phalcon\Filter*	Yes
flash	Flash Messaging Service	*Phalcon\Flash\Direct*	Yes
flashSession	Flash Session Messaging Service	*Phalcon\Flash\Session*	Yes
session	Session Service	*Phalcon\Session\Adapter\Files*	Yes
eventsManager	Events Management Service	*Phalcon\Events\Manager*	Yes
db	Low-Level Database Connection Service	*Phalcon\Db*	Yes
security	Security helpers	*Phalcon\Security*	Yes
escaper	Contextual Escaping	*Phalcon\Escaper*	Yes
annotations	Annotations Parser	*Phalcon\Annotations\Adapter\Memory*	Yes
modelsManager	Models Management Service	*Phalcon\Mvc\Model\Manager*	Yes
modelsMetadata	Models Meta-Data Service	*Phalcon\Mvc\Model\MetaData\Memory*	Yes
transactionManager	Models Transaction Manager Service	*Phalcon\Mvc\Model\Transaction\Manager*	Yes
modelsCache	Cache backend for models cache	None	•
viewsCache	Cache backend for views fragments	None	•

2.7.13 Implementing your own DI

The *Phalcon\DiInterface* interface must be implemented to create your own DI replacing the one provided by Phalcon or extend the current one.

2.8 The MVC Architecture

Phalcon offers the object-oriented classes, necessary to implement the Model, View, Controller architecture (often referred to as MVC) in your application. This design pattern is widely used by other web frameworks and desktop applications.

MVC benefits include:

- Isolation of business logic from the user interface and the database layer
- Making it clear where different types of code belong for easier maintenance

If you decide to use MVC, every request to your application resources will be managed by the MVC architecture. Phalcon classes are written in C language, offering a high performance approach of this pattern in a PHP based application.

2.8.1 Models

A model represents the information (data) of the application and the rules to manipulate that data. Models are primarily used for managing the rules of interaction with a corresponding database table. In most cases, each table in your database will correspond to one model in your application. The bulk of your application's business logic will be concentrated in the models. *Learn more*

2.8.2 Views

Views represent the user interface of your application. Views are often HTML files with embedded PHP code that perform tasks related solely to the presentation of the data. Views handle the job of providing data to the web browser or other tool that is used to make requests from your application. *Learn more*

2.8.3 Controllers

The controllers provide the "flow" between models and views. Controllers are responsible for processing the incoming requests from the web browser, interrogating the models for data, and passing that data on to the views for presentation. *Learn more*

2.9 Using Controllers

The controllers provide a number of methods that are called actions. Actions are methods on a controller that handle requests. By default all public methods on a controller map to actions and are accessible by a URL. Actions are responsible for interpreting the request and creating the response. Usually responses are in the form of a rendered view, but there are other ways to create responses as well.

For instance, when you access a URL like this: http://localhost/blog/posts/show/2012/the-post-title Phalcon by default will decompose each part like this:

Phalcon Directory	blog
Controller	posts
Action	show
Parameter	2012
Parameter	the-post-title

In this case, the PostsController will handle this request. There is no a special location to put controllers in an application, they could be loaded using *autoloaders*, so you're free to organize your controllers as you need.

Controllers must have the suffix "Controller" while actions the suffix "Action". A sample of a controller is as follows:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function showAction($year, $postTitle)
    {

    }
```

```
}
```

Additional URI parameters are defined as action parameters, so that they can be easily accessed using local variables. A controller can optionally extend *Phalcon\Mvc\Controller*. By doing this, the controller can have easy access to the application services.

Parameters without a default value are handled as required. Setting optional values for parameters is done as usual in PHP:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function showAction($year=2012, $postTitle='some default title')
    {

    }

}
```

Parameters are assigned in the same order as they were passed in the route. You can get an arbitrary parameter from its name in the following way:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function showAction()
    {
        $year = $this->dispatcher->getParam('year');
        $postTitle = $this->dispatcher->getParam('postTitle');
    }

}
```

2.9.1 Dispatch Loop

The dispatch loop will be executed within the Dispatcher until there are no actions left to be executed. In the previous example only one action was executed. Now we'll see how "forward" can provide a more complex flow of operation in the dispatch loop, by forwarding execution to a different controller/action.

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{
```

```
    public function indexAction()
    {

    }

    public function showAction($year, $postTitle)
    {
        $this->flash->error("You don't have permission to access this area");

        // Forward flow to another action
        $this->dispatcher->forward(array(
            "controller" => "users",
            "action" => "signin"
        ));
    }

}
```

If users don't have permissions to access a certain action then will be forwarded to the Users controller, signin action.

```
<?php

class UsersController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function signinAction()
    {

    }

}
```

There is no limit on the "forwards" you can have in your application, so long as they do not result in circular references, at which point your application will halt. If there are no other actions to be dispatched by the dispatch loop, the dispatcher will automatically invoke the view layer of the MVC that is managed by *Phalcon\Mvc\View*.

2.9.2 Initializing Controllers

Phalcon\Mvc\Controller offers the initialize method, which is executed first, before any action is executed on a controller. The use of the "__construct" method is not recommended.

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public $settings;

    public function initialize()
    {
        $this->settings = array(
            "mySetting" => "value"
```

```
        );
    }

    public function saveAction()
    {
        if ($this->settings["mySetting"] == "value") {
            //...
        }
    }

}
```

2.9.3 Injecting Services

If a controller extends *Phalcon\Mvc\Controller* then it has easy access to the service container in application. For example, if we have registered a service like this:

```
<?php

$di = new Phalcon\DI();

$di->set('storage', function() {
    return new Storage('/some/directory');
}, true);
```

Then, we can access to that service in several ways:

```
<?php

class FilesController extends \Phalcon\Mvc\Controller
{

    public function saveAction()
    {

        //Injecting the service by just accessing the property with the same name
        $this->storage->save('/some/file');

        //Accessing the service from the DI
        $this->di->get('storage')->save('/some/file');

        //Another way to access the service using the magic getter
        $this->di->getStorage()->save('/some/file');

        //Another way to access the service using the magic getter
        $this->getDi()->getStorage()->save('/some/file');

        //Using the array-syntax
        $this->di['storage']->save('/some/file');
    }

}
```

If you're using Phalcon as a full-stack framework, you can read the services provided *by default* in the framework.

2.9.4 Request and Response

Assuming that the framework provides a set of pre-registered services. We explain how to interact with the HTTP environment. The "request" service contains an instance of *Phalcon\Http\Request* and the "response" contains a *Phalcon\Http\Response* representing what is going to be sent back to the client.

```
<?php

class PostsController extends Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function saveAction()
    {
        // Check if request has made with POST
        if ($this->request->isPost() == true) {
            // Access POST data
            $customerName = $this->request->getPost("name");
            $customerBorn = $this->request->getPost("born");
        }
    }

}
```

The response object is not usually used directly, but is built up before the execution of the action, sometimes - like in an afterDispatch event - it can be useful to access the response directly:

```
<?php

class PostsController extends Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function notFoundAction()
    {
        // Send a HTTP 404 response header
        $this->response->setStatusCode(404, "Not Found");
    }

}
```

Learn more about the HTTP environment in their dedicated articles *request* and *response*.

2.9.5 Session Data

Sessions help us maintain persistent data between requests. You could access a *Phalcon\Session\Bag* from any controller to encapsulate data that need to be persistent.

```
<?php

class UserController extends Phalcon\Mvc\Controller
{

    public function indexAction()
    {
        $this->persistent->name = "Michael";
    }

    public function welcomeAction()
    {
        echo "Welcome, ", $this->persistent->name;
    }

}
```

2.9.6 Using Services as Controllers

Services may act as controllers, controllers classes are always requested from the services container. Accordingly, any other class registered with its name can easily replace a controller:

```
<?php

//Register a controller as a service
$di->set('IndexController', function() {
    $component = new Component();
    return $component;
});
```

2.9.7 Creating a Base Controller

Some application features like access control lists, translation, cache, and template engines are often common to many controllers. In cases like these the creation of a "base controller" is encouraged to ensure your code stays DRY. A base controller is simply a class that extends the *Phalcon\Mvc\Controller* and encapsulates the common functionality that all controllers must have. In turn, your controllers extend the "base controller" and have access to the common functionality.

This class could be located anywhere, but for organizational conventions we recommend it to be in the controllers folder, e.g. apps/controllers/ControllerBase.php. We may require this file directly in the bootstrap file or cause to be loaded using any autoloader:

```
<?php

require "../app/controllers/ControllerBase.php";
```

The implementation of common components (actions, methods, properties etc.) resides in this file:

```
<?php

class ControllerBase extends \Phalcon\Mvc\Controller
{

  /**
   * This action is available for multiple controllers
   */
```

```
  public function someAction()
  {

  }

}
```

Any other controller now inherits from ControllerBase, automatically gaining access to the common components (discussed above):

```
<?php

class UsersController extends ControllerBase
{

}
```

2.9.8 Events in Controllers

Controllers automatically act as listeners for *dispatcher* events, implementing methods with those event names allow you to implement hook points before/after the actions are executed:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function beforeExecuteRoute($dispatcher)
    {
        // This is executed before every found action

        if ($dispatcher->getActionName() == 'save') {

            $this->flash->error("You don't have permission to save posts");

            $this->dispatcher->forward(array(
                'controller' => 'home',
                'action' => 'index'
            ));

            return false;
        }
    }

    public function afterExecuteRoute($dispatcher)
    {
        // Executed after every found action
    }

}
```

2.10 Working with Models

A model represents the information (data) of the application and the rules to manipulate that data. Models are primarily used for managing the rules of interaction with a corresponding database table. In most cases, each table in your

database will correspond to one model in your application. The bulk of your application's business logic will be concentrated in the models.

Phalcon\Mvc\Model is the base for all models in a Phalcon application. It provides database independence, basic CRUD functionality, advanced finding capabilities, and the ability to relate models to one another, among other services. *Phalcon\Mvc\Model* avoids the need of having to use SQL statements because it translates methods dynamically to the respective database engine operations.

> Models are intended to work on a database high layer of abstraction. If you need to work with databases at a lower level check out the *Phalcon\Db* component documentation.

2.10.1 Creating Models

A model is a class that extends from *Phalcon\Mvc\Model*. It must be placed in the models directory. A model file must contain a single class; its class name should be in camel case notation:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

}
```

The above example shows the implementation of the "Robots" model. Note that the class Robots inherits from *Phalcon\Mvc\Model*. This component provides a great deal of functionality to models that inherit it, including basic database CRUD (Create, Read, Update, Destroy) operations, data validation, as well as sophisticated search support and the ability to relate multiple models with each other.

> If you're using PHP 5.4 is recommended declare each column that makes part of the model in order to save memory and reduce the memory allocation.

By default model "Robots" will refer to the table "robots". If you want to manually specify another name for the mapping table, you can use the getSource() method:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function getSource()
    {
        return "the_robots";
    }

}
```

The model Robots now maps to "the_robots" table. The initialize() method aids in setting up the model with a custom behavior i.e. a different table. The initialize() method is only called once during the request.

2.10.2 Models in Namespaces

Namespaces can be used to avoid class name collision. In this case it is necessary to indicate the name of the related table using getSource:

```
<?php

namespace Store\Toys;
```

```
class Robots extends \Phalcon\Mvc\Model
{

    public function getSource()
    {
        return "robots";
    }

}
```

2.10.3 Understanding Records To Objects

Every instance of a model represents a row in the table. You can easily access record data by reading object properties. For example, for a table "robots" with the records:

```
mysql> select * from robots;
+----+------------+------------+------+
| id | name       | type       | year |
+----+------------+------------+------+
|  1 | Robotina   | mechanical | 1972 |
|  2 | Astro Boy  | mechanical | 1952 |
|  3 | Terminator | cyborg     | 2029 |
+----+------------+------------+------+
3 rows in set (0.00 sec)
```

You could find a certain record by its primary key and then print its name:

```
<?php

// Find record with id = 3
$robot = Robots::findFirst(3);

// Prints "Terminator"
echo $robot->name;
```

Once the record is in memory, you can make modifications to its data and then save changes:

```
<?php

$robot = Robots::findFirst(3);
$robot->name = "RoboCop";
$robot->save();
```

As you can see, there is no need to use raw SQL statements. *Phalcon\Mvc\Model* provides high database abstraction for web applications.

2.10.4 Finding Records

Phalcon\Mvc\Model also offers several methods for querying records. The following examples will show you how to query one or more records from a model:

```
<?php

// How many robots are there?
$robots = Robots::find();
echo "There are ", count($robots), "\n";
```

```
// How many mechanical robots are there?
$robots = Robots::find("type = 'mechanical'");
echo "There are ", count($robots), "\n";

// Get and print virtual robots ordered by name
$robots = Robots::find(array(
    "type = 'virtual'",
    "order" => "name"
));
foreach ($robots as $robot) {
    echo $robot->name, "\n";
}

// Get first 100 virtual robots ordered by name
$robots = Robots::find(array(
    "type = 'virtual'",
    "order" => "name",
    "limit" => 100
));
foreach ($robots as $robot) {
   echo $robot->name, "\n";
}
```

You could also use the findFirst() method to get only the first record matching the given criteria:

```
<?php

// What's the first robot in robots table?
$robot = Robots::findFirst();
echo "The robot name is ", $robot->name, "\n";

// What's the first mechanical robot in robots table?
$robot = Robots::findFirst("type = 'mechanical'");
echo "The first mechanical robot name is ", $robot->name, "\n";

// Get first virtual robot ordered by name
$robot = Robots::findFirst(array("type = 'virtual'", "order" => "name"));
echo "The first virtual robot name is ", $robot->name, "\n";
```

Both find() and findFirst() methods accept an associative array specifying the search criteria:

```
<?php

$robot = Robots::findFirst(
    array(
        "type = 'virtual'",
        "order" => "name DESC",
        "limit" => 30
    )
);

$robots = Robots::find(
    array(
        "conditions" => "type = ?1",
        "bind"       => array(1 => "virtual")
    )
);
```

The available query options are:

Parameter	Description	Example
conditions	Search conditions for the find operation. Is used to extract only those records that fulfill a specified criterion. By default Phalcon\Mvc\Model assumes the first parameter are the conditions.	"conditions" => "name LIKE 'steve%'"
bind	Bind is used together with options, by replacing placeholders and escaping values thus increasing security	"bind" => array("status" => "A", "type" => "some-time")
bindTypes	When binding parameters, you can use this parameter to define additional casting to the bound parameters increasing even more the security	"bindTypes" => array(Column::BIND_TYPE_STR, Column::BIND_TYPE_INT)
order	Is used to sort the resultset. Use one or more fields separated by commas.	"order" => "name DESC, status"
limit	Limit the results of the query to results to certain range	"limit" => 10
group	Allows to collect data across multiple records and group the results by one or more columns	"group" => "name, status"
for_update	With this option, *Phalcon\Mvc\Model* reads the latest available data, setting exclusive locks on each row it reads	"for_update" => true
shared_lock	With this option, *Phalcon\Mvc\Model* reads the latest available data, setting shared locks on each row it reads	"shared_lock" => true
cache	Cache the resultset, reducing the continuous access to the relational system	"cache" => array("lifetime" => 3600, "key" => "my-find-key")
hydration	Sets the hydration strategy to represent each returned record in the result	"hydration" => Resultset::HYDRATE_OBJECTS

If you prefer, there is also available a way to create queries in an object-oriented way, instead of using an array of parameters:

```
<?php

$robots = Robots::query()
    ->where("type = :type:")
    ->andWhere("year < 2000")
    ->bind(array("type" => "mechanical"))
    ->order("name")
    ->execute();
```

The static method query() returns a *Phalcon\Mvc\Model\Criteria* object that is friendly with IDE autocompleters.

All the queries are internally handled as *PHQL* queries. PHQL is a high-level, object-oriented and SQL-like language. This language provide you more features to perform queries like joining other models, define groupings, add agreggations etc.

Model Resultsets

While findFirst() returns directly an instance of the called class (when there is data to be returned), the find() method returns a *Phalcon\Mvc\Model\Resultset\Simple*. This is an object that encapsulates all the functionality a resultset has like traversing, seeking specific records, counting, etc.

These objects are more powerful than standard arrays. One of the greatest features of the *Phalcon\Mvc\Model\Resultset* is that at any time there is only one record in memory. This greatly helps in memory management especially when working with large amounts of data.

```
<?php
```

```
// Get all robots
$robots = Robots::find();

// Traversing with a foreach
foreach ($robots as $robot) {
    echo $robot->name, "\n";
}

// Traversing with a while
$robots->rewind();
while ($robots->valid()) {
    $robot = $robots->current();
    echo $robot->name, "\n";
    $robots->next();
}

// Count the resultset
echo count($robots);

// Alternative way to count the resultset
echo $robots->count();

// Move the internal cursor to the third robot
$robots->seek(2);
$robot = $robots->current()

// Access a robot by its position in the resultset
$robot = $robots[5];

// Check if there is a record in certain position
if (isset($robots[3]) {
   $robot = $robots[3];
}

// Get the first record in the resultset
$robot = robots->getFirst();

// Get the last record
$robot = robots->getLast();
```

Phalcon's resultsets emulate scrollable cursors, you can get any row just by accessing its position, or seeking the internal pointer to a specific position. Note that some database systems don't support scrollable cursors, this forces to re-execute the query in order to rewind the cursor to the beginning and obtain the record at the requested position. Similarly, if a resultset is traversed several times, the query must be executed the same number of times.

Storing large query results in memory could consume many resources, because of this, resultsets are obtained from the database in chunks of 32 rows reducing the need for re-execute the request in several cases also saving memory.

Note that resultsets can be serialized and stored in a cache backend. *Phalcon\Cache* can help with that task. However, serializing data causes *Phalcon\Mvc\Model* to retrieve all the data from the database in an array, thus consuming more memory while this process takes place.

```
<?php

// Query all records from model parts
$parts = Parts::find();

// Store the resultset into a file
file_put_contents("cache.txt", serialize($parts));
```

```
// Get parts from file
$parts = unserialize(file_get_contents("cache.txt"));

// Traverse the parts
foreach ($parts as $part) {
   echo $part->id;
}
```

Binding Parameters

Bound parameters are also supported in *Phalcon\Mvc\Model*. Although there is a minimal performance impact by using bound parameters, you are encouraged to use this methodology so as to eliminate the possibility of your code being subject to SQL injection attacks. Both string and integer placeholders are supported. Binding parameters can simply be achieved as follows:

```
<?php

// Query robots binding parameters with string placeholders
$conditions = "name = :name: AND type = :type:";

//Parameters whose keys are the same as placeholders
$parameters = array(
    "name" => "Robotina",
    "type" => "maid"
);

//Perform the query
$robots = Robots::find(array(
    $conditions,
    "bind" => $parameters
));

// Query robots binding parameters with integer placeholders
$conditions = "name = ?1 AND type = ?2";
$parameters = array(1 => "Robotina", 2 => "maid");
$robots     = Robots::find(array(
    $conditions,
    "bind" => $parameters
));

// Query robots binding parameters with both string and integer placeholders
$conditions = "name = :name: AND type = ?1";

//Parameters whose keys are the same as placeholders
$parameters = array(
    "name" => "Robotina",
    1 => "maid"
);

//Perform the query
$robots = Robots::find(array(
    $conditions,
    "bind" => $parameters
));
```

When using numeric placeholders, you will need to define them as integers i.e. 1 or 2. In this case "1" or "2" are considered strings and not numbers, so the placeholder could not be successfully replaced.

Strings are automatically escaped using PDO. This function takes into account the connection charset, so its recommended to define the correct charset in the connection parameters or in the database configuration, as a wrong charset will produce undesired effects when storing or retrieving data.

Additionally you can set the parameter "bindTypes", this allows defining how the parameters should be bound according to its data type:

```
<?php

use \Phalcon\Db\Column;

//Bind parameters
$parameters = array(
    "name" => "Robotina",
    "year" => 2008
);

//Casting Types
$types = array(
    "name" => Column::BIND_PARAM_STR,
    "year" => Column::BIND_PARAM_INT
);

// Query robots binding parameters with string placeholders
$conditions = "name = :name: AND year = :year:";
$robots = Robots::find(array(
    $conditions,
    "bind" => $parameters,
    "bindTypes" => $types
));
```

Since the default bind-type is \Phalcon\Db\Column::BIND_TYPE_STR, there is no need to specify the "bindTypes" parameter if all of the columns are of that type.

Bound parameters are available for all query methods such as find() and findFirst() but also the calculation methods like count(), sum(), average() etc.

2.10.5 Relationships between Models

There are four types of relationships: one-on-one, one-to-many, many-to-one and many-to-many. The relationship may be unidirectional or bidirectional, and each can be simple (a one to one model) or more complex (a combination of models). The model manager manages foreign key constraints for these relationships, the definition of these helps referential integrity as well as easy and fast access of related records to a model. Through the implementation of relations, it is easy to access data in related models from each record in a uniform way.

Unidirectional relationships

Unidirectional relations are those that are generated in relation to one another but not vice versa.

Bidirectional relations

The bidirectional relations build relationships in both models and each model defines the inverse relationship of the other.

Defining relationships

In Phalcon, relationships must be defined in the initialize() method of a model. The methods belongsTo(), hasOne() or hasMany() define the relationship between one or more fields from the current model to fields in another model. Each of these methods requires 3 parameters: local fields, referenced model, referenced fields.

Method	Description
hasMany	Defines a 1-n relationship
hasOne	Defines a 1-1 relationship
belongsTo	Defines a n-1 relationship

The following schema shows 3 tables whose relations will serve us as an example regarding relationships:

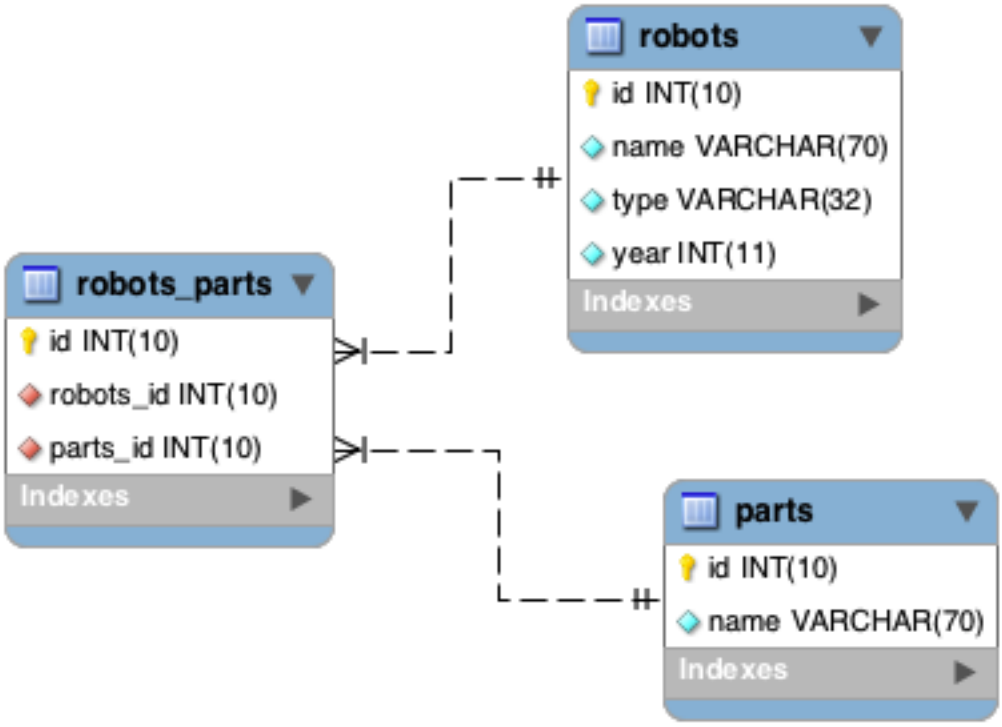
```
CREATE TABLE `robots` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `name` varchar(70) NOT NULL,
    `type` varchar(32) NOT NULL,
    `year` int(11) NOT NULL,
    PRIMARY KEY (`id`)
);

CREATE TABLE `robots_parts` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `robots_id` int(10) NOT NULL,
    `parts_id` int(10) NOT NULL,
    `created_at` DATE NOT NULL,
    PRIMARY KEY (`id`),
    KEY `robots_id` (`robots_id`),
    KEY `parts_id` (`parts_id`)
);

CREATE TABLE `parts` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `name` varchar(70) NOT NULL,
    PRIMARY KEY (`id`)
);
```

- The model “Robots” has many “RobotsParts”.
- The model “Parts” has many “RobotsParts”.
- The model “RobotsParts” belongs to both “Robots” and “Parts” models as a many-to-one relation.

Check the EER diagram to understand better the relations:

The models with their relations could be implemented as follows:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{
    public $id;

    public $name;

    public function initialize()
    {
        $this->hasMany("id", "RobotsParts", "robots_id");
    }

}
```


robots
id INT(10)
name VARCHAR(70)
type VARCHAR(32)
year INT(11)
Indexes
robots_parts
id INT(10)
robots_id INT(10)
parts_id INT(10)
Indexes
parts
id INT(10)
name VARCHAR(70)
Indexes

```php
<?php

class Parts extends \Phalcon\Mvc\Model
{

    public $id;

    public $name;

    public function initialize()
    {
        $this->hasMany("id", "RobotsParts", "parts_id");
    }

}

<?php

class RobotsParts extends \Phalcon\Mvc\Model
{

    public $id;

    public $robots_id;

    public $parts_id;

    public function initialize()
    {
        $this->belongsTo("robots_id", "Robots", "id");
        $this->belongsTo("parts_id", "Parts", "id");
    }

}
```

The first parameter indicates the field of the local model used in the relationship; the second indicates the name of the referenced model and the third the field name in the referenced model. You could also use arrays to define multiple fields in the relationship.

Taking advantage of relationships

When explicitly defining the relationships between models, it is easy to find related records for a particular record.

```php
<?php

$robot = Robots::findFirst(2);
foreach ($robot->robotsParts as $robotPart) {
    echo $robotPart->parts->name, "\n";
}
```

Phalcon uses the magic methods __set/__get/__call to store or retrieve related data using relationships.

By accesing an attribute with the same name as the relationship will retrieve all its related record(s).

```php
<?php

$robot = Robots::findFirst();
$robotsParts = $robot->robotsParts; // all the related records in RobotsParts
```

Also, you can use a magic getter:

```
<?php

$robot = Robots::findFirst();
$robotsParts = $robot->getRobotsParts(); // all the related records in RobotsParts
$robotsParts = $robot->getRobotsParts(array('limit' => 5)); // passing parameters
```

If the called method has a "get" prefix *Phalcon\Mvc\Model* will return a findFirst()/find() result. The following example compares retrieving related results with using magic methods and without:

```
<?php

$robot = Robots::findFirst(2);

// Robots model has a 1-n (hasMany)
// relationship to RobotsParts then
$robotsParts = $robot->robotsParts;

// Only parts that match conditions
$robotsParts = $robot->getRobotsParts("created_at = '2012-03-15'");

// Or using bound parameters
$robotsParts = $robot->getRobotsParts(array(
    "created_at = :date:",
    "bind" => array("date" => "2012-03-15")
));

$robotPart = RobotsParts::findFirst(1);

// RobotsParts model has a n-1 (belongsTo)
// relationship to RobotsParts then
$robot = $robotPart->robots;
```

Getting related records manually:

```
<?php

$robot = Robots::findFirst(2);

// Robots model has a 1-n (hasMany)
// relationship to RobotsParts, then
$robotsParts = RobotsParts::find("robots_id = '" . $robot->id . "'");

// Only parts that match conditions
$robotsParts = RobotsParts::find(
    "robots_id = '" . $robot->id . "' AND created_at = '2012-03-15'"
);

$robotPart = RobotsParts::findFirst(1);

// RobotsParts model has a n-1 (belongsTo)
// relationship to RobotsParts then
$robot = Robots::findFirst("id = '" . $robotPart->robots_id . "'");
```

The prefix "get" is used to find()/findFirst() related records. Depending on the type of relation it will use 'find' or 'findFirst':

Type	Description | Implicit Method
Belongs-To	Returns a model instance of the related record directly | findFirst
Has-One	Returns a model instance of the related record directly | findFirst
Has-Many	Returns a collection of model instances of the referenced model | find

You can also use "count" prefix to return an integer denoting the count of the related records:

```
<?php

$robot = Robots::findFirst(2);
echo "The robot has ", $robot->countRobotsParts(), " parts\n";
```

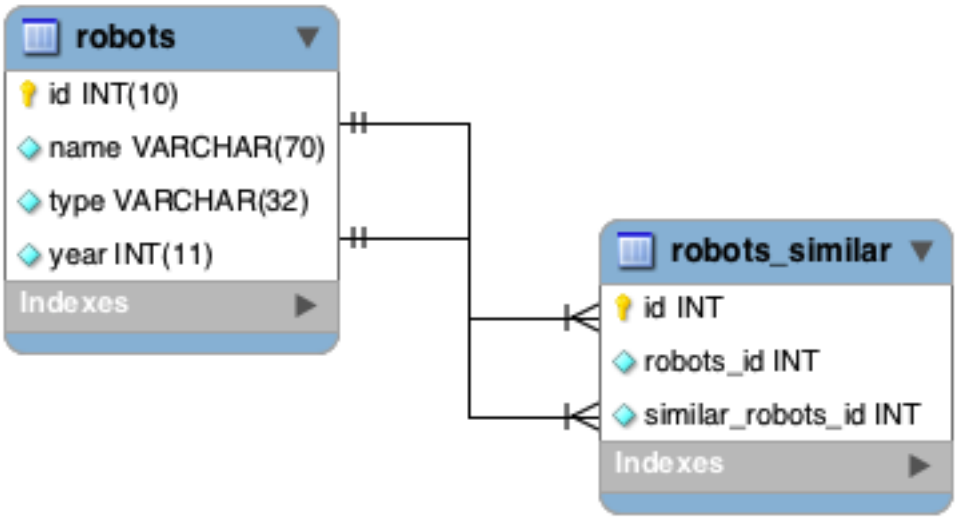
Aliasing Relationships

To explain better how aliases work, let's check the following example:

The table "robots_similar" has the function to define what robots are similar to others:

```
mysql> desc robots_similar;
+------------------+------------------+------+-----+---------+----------------+
| Field            | Type             | Null | Key | Default | Extra          |
+------------------+------------------+------+-----+---------+----------------+
| id               | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| robots_id        | int(10) unsigned | NO   | MUL | NULL    |                |
| similar_robots_id | int(10) unsigned | NO   |     | NULL    |                |
+------------------+------------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

Both "robots_id" and "similar_robots_id" have a relation to the model Robots:



A model that maps this table and its relationships is the following:

```
<?php

class RobotsSimilar extends Phalcon\Mvc\Model
{

    public function initialize()
```

```
    {
        $this->belongsTo('robots_id', 'Robots', 'id');
        $this->belongsTo('similar_robots_id', 'Robots', 'id');
    }

}
```

Since both relations point to the same model (Robots), obtain the records related to the relationship could not be clear:

```
<?php

$robotsSimilar = RobotsSimilar::findFirst();

//Returns the related record based on the column (robots_id)
//Also as is a belongsTo it's only returning one record
//but the name 'getRobots' seems to imply that return more than one
$robot = $robotsSimilar->getRobots();

//but, how to get the related record based on the column (similar_robots_id)
//if both relationships have the same name?
```

The aliases allow us to rename both releationships to solve these problems:

```
<?php

class RobotsSimilar extends Phalcon\Mvc\Model
{

    public function initialize()
    {
        $this->belongsTo('robots_id', 'Robots', 'id', array(
            'alias' => 'Robot'
        ));
        $this->belongsTo('similar_robots_id', 'Robots', 'id', array(
            'alias' => 'SimilarRobot'
        ));
    }

}
```

With the aliasing we can get the related records easily:

```
<?php

$robotsSimilar = RobotsSimilar::findFirst();

//Returns the related record based on the column (robots_id)
$robot = $robotsSimilar->getRobot();
$robot = $robotsSimilar->robot;

//Returns the related record based on the column (similar_robots_id)
$similarRobot = $robotsSimilar->getSimilarRobot();
$similarRobot = $robotsSimilar->similarRobot;
```

Magic Getters vs. Explicit methods

Most IDEs and editors with auto-completion capabilities can not infer the correct types when using magic getters, instead of use the magic getters you can optionally define those methods explicitly with the corresponding docblocks

helping the IDE to produce a better auto-completion:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public $id;

    public $name;

    public function initialize()
    {
        $this->hasMany("id", "RobotsParts", "robots_id");
    }

    /**
     * Return the related "robots parts"
     *
     * @return \RobotsParts[]
     */
    public function getRobotsParts($parameters=null)
    {
        return $this->getRelated('RobotsParts', $parameters);
    }

}
```

Virtual Foreign Keys

By default, relationships do not act like database foreign keys, that is, if you try to insert/update a value without having a valid value in the referenced model, Phalcon will not produce a validation message. You can modify this behavior by adding a fourth parameter when defining a relationship.

The RobotsPart model can be changed to demonstrate this feature:

```
<?php

class RobotsParts extends \Phalcon\Mvc\Model
{

    public $id;

    public $robots_id;

    public $parts_id;

    public function initialize()
    {
        $this->belongsTo("robots_id", "Robots", "id", array(
            "foreignKey" => true
        ));

        $this->belongsTo("parts_id", "Parts", "id", array(
            "foreignKey" => array(
                "message" => "The part_id does not exist on the Parts model"
            )
        ));
```

```
    }

}
```

If you alter a belongsTo() relationship to act as foreign key, it will validate that the values inserted/updated on those fields have a valid value on the referenced model. Similarly, if a hasMany()/hasOne() is altered it will validate that the records cannot be deleted if that record is used on a referenced model.

```
<?php

class Parts extends \Phalcon\Mvc\Model
{

    public function initialize()
    {
        $this->hasMany("id", "RobotsParts", "parts_id", array(
            "foreignKey" => array(
                "message" => "The part cannot be deleted because other robots are using it"
            )
        ));
    }

}
```

2.10.6 Generating Calculations

Calculations are helpers for commonly used functions of database systems such as COUNT, SUM, MAX, MIN or AVG. *Phalcon\Mvc\Model* allows to use these functions directly from the exposed methods.

Count examples:

```
<?php

// How many employees are?
$rowcount = Employees::count();

// How many different areas are assigned to employees?
$rowcount = Employees::count(array("distinct" => "area"));

// How many employees are in the Testing area?
$rowcount = Employees::count("area = 'Testing'");

//Count employees grouping results by their area
$group = Employees::count(array("group" => "area"));
foreach ($group as $row) {
   echo "There are ", $group->rowcount, " in ", $group->area;
}

// Count employees grouping by their area and ordering the result by count
$group = Employees::count(
    array(
        "group" => "area",
        "order" => "rowcount"
    )
);
```

Sum examples:

```php
<?php

// How much are the salaries of all employees?
$total = Employees::sum(array("column" => "salary"));

// How much are the salaries of all employees in the Sales area?
$total = Employees::sum(array(
    "column"     => "salary",
    "conditions" => "area = 'Sales'"
));

// Generate a grouping of the salaries of each area
$group = Employees::sum(array(
    "column" => "salary",
    "group"  => "area"
));
foreach ($group as $row) {
   echo "The sum of salaries of the ", $group->area, " is ", $group->sumatory;
}

// Generate a grouping of the salaries of each area ordering
// salaries from higher to lower
$group = Employees::sum(array(
    "column" => "salary",
    "group"  => "area",
    "order"  => "sumatory DESC"
));
```

Average examples:

```php
<?php

// What is the average salary for all employees?
$average = Employees::average(array("column" => "salary"));

// What is the average salary for the Sales's area employees?
$average = Employees::average(array(
    "column" => "salary",
    "conditions" => "area = 'Sales'"
));
```

Max/Min examples:

```php
<?php

// What is the oldest age of all employees?
$age = Employees::maximum(array("column" => "age"));

// What is the oldest of employees from the Sales area?
$age = Employees::maximum(array(
    "column" => "age",
    "conditions" => "area = 'Sales'"
));

// What is the lowest salary of all employees?
$salary = Employees::minimum(array("column" => "salary"));
```

2.10.7 Hydration Modes

As mentioned above, resultsets are collections of complete objects, this means that every returned result is an object representing a row in the database. These objects can be modified and saved again to persistence:

```
<?php

//Manipulating a resultset of complete objects
foreach (Robots::find() as $robot) {
    $robot->year = 2000;
    $robot->save();
}
```

Sometimes records are obtained only to be presented to a user in read-only mode, in these cases it may be useful to change the way in which records are represented to facilitate their handling. The strategy used to represent objects returned in a resultset is called 'hydration mode':

```
<?php

use Phalcon\Mvc\Model\Resultset;

$robots = Robots::find();

//Return every robot as an array
$robots->setHydrateMode(Resultset::HYDRATE_ARRAYS);

foreach ($robots as $robot) {
    echo $robot['year'], PHP_EOL;
}

//Return every robot as an stdClass
$robots->setHydrateMode(Resultset::HYDRATE_OBJECTS);

foreach ($robots as $robot) {
    echo $robot->year, PHP_EOL;
}

//Return every robot as a Robots instance
$robots->setHydrateMode(Resultset::HYDRATE_RECORDS);

foreach ($robots as $robot) {
    echo $robot->year, PHP_EOL;
}
```

The hydration mode can be passed as a parameter of 'find':

```
<?php

use Phalcon\Mvc\Model\Resultset;

$robots = Robots::find(array(
    'hydration' => Resultset::HYDRATE_ARRAYS
));

foreach ($robots as $robot) {
    echo $robot['year'], PHP_EOL;
}
```

2.10.8 Creating Updating/Records

The method Phalcon\Mvc\Model::save() allows you to create/update records according to whether they already exist in the table associated with a model. The save method is called internally by the create and update methods of *Phalcon\Mvc\Model*. For this to work as expected it is necessary to have properly defined a primary key in the entity to determine whether a record should be updated or created.

Also the method executes associated validators, virtual foreign keys and events that are defined in the model:

```
<?php

$robot       = new Robots();
$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;
if ($robot->save() == false) {
    echo "Umh, We can't store robots right now: \n";
    foreach ($robot->getMessages() as $message) {
        echo $message, "\n";
    }
} else {
    echo "Great, a new robot was saved successfully!";
}
```

An array could be passed to "save" to avoid assign every column manually. Phalcon\Mvc\Model will check if there are setters implemented for the columns passed in the array giving priority to them instead of assign directly the values of the attributes:

```
<?php

$robot = new Robots();
$robot->save(array(
    "type" => "mechanical",
    "name" => "Astro Boy",
    "year" => 1952
));
```

Values assigned directly or via the array of attributes are escaped/sanitized according to the related attribute data type. So you can pass an insecure array without worrying about possible SQL injections:

```
<?php

$robot = new Robots();
$robot->save($_POST);
```

> Without precautions mass assignment could allow attackers to set any database column's value. Only use this feature if you want that a user can insert/update every column in the model, even if those fields are not in the submitted form.

You can set an additional parameter to save to set a whitelist of fields that only must taken into account when doing the mass assignment:

```
<?php

$robot = new Robots();
$robot->save($_POST, array('name', 'type'));
```

Create/Update with Confidence

When an application has a lot of competition, we could be expecting create a record but it is actually updated. This could happen if we use Phalcon\Mvc\Model::save() to persist the records in the database. f we want to be absolutely sure that a record is created or updated, we can change the save() call with create() or update():

```
<?php

$robot       = new Robots();
$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;

//This record only must be created
if ($robot->create() == false) {
    echo "Umh, We can't store robots right now: \n";
    foreach ($robot->getMessages() as $message) {
        echo $message, "\n";
    }
} else {
    echo "Great, a new robot was created successfully!";
}
```

These methods "create" and "update" also accept an array of values as parameter.

Auto-generated identity columns

Some models may have identity columns. These columns usually are the primary key of the mapped table. *Phalcon\Mvc\Model* can recognize the identity column omitting it in the generated SQL INSERT, so the database system can generate an auto-generated value for it. Always after creating a record, the identity field will be registered with the value generated in the database system for it:

```
<?php

$robot->save();

echo "The generated id is: ", $robot->id;
```

Phalcon\Mvc\Model is able to recognize the identity column. Depending on the database system, those columns may be serial columns like in PostgreSQL or auto_increment columns in the case of MySQL.

PostgreSQL uses sequences to generate auto-numeric values, by default, Phalcon tries to obtain the generated value from the sequence "table_field_seq", for example: robots_id_seq, if that sequence has a different name, the method "getSequenceName" needs to be implemented:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function getSequenceName()
    {
        return "robots_sequence_name";
    }

}
```

Storing related records

Magic properties can be used to store a records and its related properties:

```
<?php

// Create a robot
$artist = new Artists();
$artist->name = 'Shinichi Osawa';
$artist->country = 'Japan';

// Create an album
$album = new Albums();
$album->name = 'The One';
$album->artist = $artist; //Assign the artist
$album->year = 2008;

//Save both records
$album->save();
```

Saving a record and its related records in a has-many relation:

```
<?php

// Get an existing artist
$artist = Artists::findFirst('name = "Shinichi Osawa"');

// Create an album
$album = new Albums();
$album->name = 'The One';
$album->artist = $artist;

$songs = array();

// Create a first song
$songs[0] = new Songs();
$songs[0]->name = 'Star Guitar';
$songs[0]->duration = '5:54';

// Create a second song
$songs[1] = new Songs();
$songs[1]->name = 'Last Days';
$songs[1]->duration = '4:29';

// Assign the songs array
$album->songs = $songs;

// Save the album + its songs
$album->save();
```

Saving the album and the artist at the same time uses a transaction so if anything goes wrong with saving the related records, the parent will not be saved either. Messages are passed back to the user for information regarding any errors

Validation Messages

Phalcon\Mvc\Model has a messaging subsystem that provides a flexible way to output or store the validation messages generated during the insert/update processes.

Each message consists of an instance of the class *Phalcon\Mvc\Model\Message*. The set of messages generated can be retrieved with the method getMessages(). Each message provides extended information like the field name that generated the message or the message type:

```
<?php

if ($robot->save() == false) {
    foreach ($robot->getMessages() as $message) {
        echo "Message: ", $message->getMessage();
        echo "Field: ", $message->getField();
        echo "Type: ", $message->getType();
    }
}
```

Phalcon\Mvc\Model can generate the following types of validation messages:

Type	Description
PresenceOf	Generated when a field with a non-null attribute on the database is trying to insert/update a null value
ConstraintViolation	Generated when a field part of a virtual foreign key is trying to insert/update a value that doesn't exist in the referenced model
InvalidValue	Generated when a validator failed because of an invalid value
InvalidCreateAttempt	Produced when a record is attempted to be created but it already exists
InvalidUpdateAttempt	Produced when a record is attempted to be updated but it doesn't exist

The method getMessages() can be overriden in a model to replace/translate the default messages generated automatically by the ORM:

```
<?php

class Robots extends Phalcon\Mvc\Model
{
    public function getMessages()
    {
        $messages = array();
        foreach (parent::getMessages() as $message) {
            switch ($message->getType()) {
                case 'InvalidCreateAttempt':
                    $messages[] = 'The record cannot be created because it already exists';
                    break;
                case 'InvalidUpdateAttempt':
                    $messages[] = 'The record cannot be updated because it already exists';
                    break;
                case 'PresenceOf':
                    $messages[] = 'The field ' . $message->getField() . ' is mandatory';
                    break;
            }
        }
        return $messages;
    }
}
```

Events and Events Manager

Models allow you to implement events that will be thrown when performing an insert/update/delete. They help define business rules for a certain model. The following are the events supported by *Phalcon\Mvc\Model* and their order of

execution:

Operation	Name	Can stop operation?	Explanation
Inserting/Updating	beforeValidation	YES	Is executed before the fields are validated for not nulls/empty strings or foreign keys
Inserting	beforeValidationOnCreate	YES	Is executed before the fields are validated for not nulls/empty strings or foreign keys when an insertion operation is being made
Updating	beforeValidationOnUpdate	YES	Is executed before the fields are validated for not nulls/empty strings or foreign keys when an updating operation is being made
Inserting/Updating	onValidationFails	YES (already stopped)	Is executed after an integrity validator fails
Inserting	afterValidationOnCreate	YES	Is executed after the fields are validated for not nulls/empty strings or foreign keys when an insertion operation is being made
Updating	afterValidationOnUpdate	YES	Is executed after the fields are validated for not nulls/empty strings or foreign keys when an updating operation is being made
Inserting/Updating	afterValidation	YES	Is executed after the fields are validated for not nulls/empty strings or foreign keys
Inserting/Updating	beforeSave	YES	Runs before the required operation over the database system
Updating	beforeUpdate	YES	Runs before the required operation over the database system only when an updating operation is being made
Inserting	beforeCreate	YES	Runs before the required operation over the database system only when an inserting operation is being made
Updating	afterUpdate	NO	Runs after the required operation over the database system only when an updating operation is being made
Inserting	afterCreate	NO	Runs after the required operation over the database system only when an inserting operation is being made
Inserting/Updating	afterSave	NO	Runs after the required operation over the database system

Implementing Events in the Model's class

The easier way to make a model react to events is implement a method with the same name of the event in the model's class:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function beforeValidationOnCreate()
    {
        echo "This is executed before creating a Robot!";
    }

}
```

Events can be useful to assign values before performing an operation, for example:

```
<?php

class Products extends \Phalcon\Mvc\Model
{
```

```
    public function beforeCreate()
    {
        //Set the creation date
        $this->created_at = date('Y-m-d H:i:s');
    }

    public function beforeUpdate()
    {
        //Set the modification date
        $this->modified_in = date('Y-m-d H:i:s');
    }

}
```

Using a custom Events Manager

Additionally, this component is integrated with *Phalcon\Events\Manager*, this means we can create listeners that run when an event is triggered.

```
<?php

class Robots extends Phalcon\Mvc\Model
{

    public function initialize()
    {

        $eventsManager = new \Phalcon\Events\Manager();

        //Attach an anonymous function as a listener for "model" events
        $eventsManager->attach('model', function($event, $robot) {
            if ($event->getType() == 'beforeSave') {
                if ($robot->name == 'Scooby Doo') {
                    echo "Scooby Doo isn't a robot!";
                    return false;
                }
            }
            return true;
        });
    }

}

$robot = new Robots();
$robot->name = 'Scooby Doo';
$robot->year = 1969;
$robot->save();
```

In the example given above the EventsManager only acts as a bridge between an object and a listener (the anonymous function). If we want all objects created in our application use the same EventsManager, then we need to assign it to the Models Manager:

```
<?php

//Registering the modelsManager service
$di->setShared('modelsManager', function() {
```

```
    $eventsManager = new \Phalcon\Events\Manager();

    //Attach an anonymous function as a listener for "model" events
    $eventsManager->attach('model', function($event, $model){

        //Catch events produced by the Robots model
        if (get_class($model) == 'Robots') {

            if ($event->getType() == 'beforeSave') {
                if ($modle->name == 'Scooby Doo') {
                    echo "Scooby Doo isn't a robot!";
                    return false;
                }
            }

        }
        return true;
    });

    //Setting a default EventsManager
    $modelsManager = new Phalcon\Mvc\Models\Manager();
    $modelsManager->setEventsManager($eventsManager);
    return $modelsManager;
});
```

If a listener returns false that will stop the operation that is executing currently.

Implementing a Business Rule

When an insert, update or delete is executed, the model verifies if there are any methods with the names of the events listed in the table above.

We recommend that validation methods are declared protected to prevent that business logic implementation from being exposed publicly.

The following example implements an event that validates the year cannot be smaller than 0 on update or insert:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function beforeSave()
    {
        if ($this->year < 0) {
            echo "Year cannot be smaller than zero!";
            return false;
        }
    }

}
```

Some events return false as an indication to stop the current operation. If an event doesn't return anything, *Phalcon\Mvc\Model* will assume a true value.

Validating Data Integrity

Phalcon\Mvc\Model provides several events to validate data and implement business rules. The special "validation" event allows us to call built-in validators over the record. Phalcon exposes a few built-in validators that can be used at this stage of validation.

The following example shows how to use it:

```
<?php

use Phalcon\Mvc\Model\Validator\InclusionIn,
    Phalcon\Mvc\Model\Validator\Uniqueness;

class Robots extends \Phalcon\Mvc\Model
{

    public function validation()
    {

        $this->validate(new InclusionIn(
            array(
                "field"  => "type",
                "domain" => array("Mechanical", "Virtual")
            )
        ));

        $this->validate(new Uniqueness(
            array(
                "field"   => "name",
                "message" => "The robot name must be unique"
            )
        ));

        return $this->validationHasFailed() != true;
    }

}
```

The above example performs a validation using the built-in validator "InclusionIn". It checks the value of the field "type" in a domain list. If the value is not included in the method then the validator will fail and return false. The following built-in validators are available:

Name	Explanation	Example
PresenceOf	Validates that a field's value isn't null or empty string. This validator is automatically added based on the attributes marked as not null on the mapped table	*Example*
Email	Validates that field contains a valid email format	*Example*
ExclusionIn	Validates that a value is not within a list of possible values	*Example*
InclusionIn	Validates that a value is within a list of possible values	*Example*
Numericality	Validates that a field has a numeric format	*Example*
Regex	Validates that the value of a field matches a regular expression	*Example*
Uniqueness	Validates that a field or a combination of a set of fields are not present more than once in the existing records of the related table	*Example*
StringLength	Validates the length of a string	*Example*
Url	Validates a URL format	*Example*

In addition to the built-in validatiors, you can create your own validators:

```
<?php

use Phalcon\Mvc\Model\Validator,
    Phalcon\Mvc\Model\ValidatorInterface;

class MaxMinValidator extends Validator implements ValidatorInterface
{

    public function validate($model)
    {
        $field = $this->getOption('field');

        $min = $this->getOption('min');
        $max = $this->getOption('max');

        $value = $model->$field;

        if ($min <= $value && $value <= $max) {
            $this->appendMessage("The field doesn't have the right range of values", $field, "MaxMin
            return false;
        }
        return true;
    }
```

```
}
```

Adding the validator to a model:

```
<?php

class Customers extends \Phalcon\Mvc\Model
{

    public function validation()
    {
        $this->validate(new MaxMinValidator(
            array(
                "field"  => "price",
                "min" => 10,
                "max" => 100
            )
        ));
        if ($this->validationHasFailed() == true) {
            return false;
        }
    }

}
```

The idea of creating validators is make them reusable between several models. A validator can also be as simple as:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function validation()
    {
        if ($this->type == "Old") {
            $message = new Phalcon\Mvc\Model\Message(
                "Sorry, old robots are not allowed anymore",
                "type",
                "MyType"
            );
            $this->appendMessage($message);
            return false;
        }
        return true;
    }

}
```

Avoiding SQL injections

Every value assigned to a model attribute is escaped depending of its data type. A developer doesn't need to escape manually each value before storing it on the database. Phalcon uses internally the bound parameters capability provided by PDO to automatically escape every value to be stored in the database.

```
mysql> desc products;
+-----------------+-----------------+------+-----+---------+----------------+
| Field           | Type            | Null | Key | Default | Extra          |
```

```
+------------------+------------------+------+-----+---------+----------------+
| id               | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| product_types_id | int(10) unsigned | NO   | MUL | NULL    |                |
| name             | varchar(70)      | NO   |     | NULL    |                |
| price            | decimal(16,2)    | NO   |     | NULL    |                |
| active           | char(1)          | YES  |     | NULL    |                |
+------------------+------------------+------+-----+---------+----------------+
5 rows in set (0.00 sec)
```

If we use just PDO to store a record in a secure way, we need to write the following code:

```
<?php

$productTypesId = 1;
$name = 'Artichoke';
$price = 10.5;
$active = 'Y';

$sql = 'INSERT INTO products VALUES (null, :productTypesId, :name, :price, :active)';
$sth = $dbh->prepare($sql);

$sth->bindParam(':productTypesId', $productTypesId, PDO::PARAM_INT);
$sth->bindParam(':name', $name, PDO::PARAM_STR, 70);
$sth->bindParam(':price', doubleval($price));
$sth->bindParam(':active', $active, PDO::PARAM_STR, 1);

$sth->execute();
```

The good news is that Phalcon do this for you automatically:

```
<?php

$product = new Products();
$product->product_types_id = 1;
$product->name = 'Artichoke';
$product->price = 10.5;
$product->active = 'Y';
$product->create();
```

2.10.9 Skipping Columns

To tell Phalcon\Mvc\Model that always omits some fields in the creation and/or update of records in order to delegate the database system the assignation of the values by a trigger or a default:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function initialize()
    {
        //Skips fields/columns on both INSERT/UPDATE operations
        $this->skipAttributes(array('year', 'price'));

        //Skips only when inserting
        $this->skipAttributesOnCreate(array('created_at'));

        //Skips only when updating
```

```
        $this->skipAttributesOnUpdate(array('modified_in'));
    }

}
```

This will ignore globally these fields on each INSERT/UPDATE operation on the whole application. Forcing a default value can be done in the following way:

```
<?php

$robot = new Robots();
$robot->name = 'Bender';
$robot->year = 1999;
$robot->created_at = new \Phalcon\Db\RawValue('default');
$robot->create();
```

A callback also can be used to create a conditional assigment of automatic default values:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{
    public function beforeCreate()
    {
        if ($this->price > 10000) {
            $robot->type = new \Phalcon\Db\RawValue('default');
        }
    }
}
```

> Never use a \Phalcon\Db\RawValue to assign external data (such as user input) or variable data. The value of these fields is ignored when binding parameters to the query. So it could be used to attack the application injecting SQL.

Dynamic Update

SQL UPDATE statements are by default created with every column defined in the model (full all-field SQL update). You can change specific models to make dynamic updates, in this case, just the fields that had changed are used to create the final SQL statement.

In some cases this could improve the performance by reducing the traffic between the application and the database server, this specially helps when the table has blob/text fields:

```
<?php

class Robots extends Phalcon\Mvc\Model
{
    public function initalize()
    {
        $this->useDynamicUpdate(true);
    }
}
```

2.10.10 Deleting Records

The method Phalcon\Mvc\Model::delete() allows to delete a record. You can use it as follows:

```
<?php

$robot = Robots::findFirst(11);
if ($robot != false) {
    if ($robot->delete() == false) {
        echo "Sorry, we can't delete the robot right now: \n";
        foreach ($robot->getMessages() as $message) {
            echo $message, "\n";
        }
    } else {
        echo "The robot was deleted successfully!";
    }
}
```

You can also delete many records by traversing a resultset with a foreach:

```
<?php

foreach (Robots::find("type='mechanical'") as $robot) {
    if ($robot->delete() == false) {
        echo "Sorry, we can't delete the robot right now: \n";
        foreach ($robot->getMessages() as $message) {
            echo $message, "\n";
        }
    } else {
        echo "The robot was deleted successfully!";
    }
}
```

The following events are available to define custom business rules that can be executed when a delete operation is performed:

Operation	Name	Can stop operation?	Explanation
Deleting	beforeDelete	YES	Runs before the delete operation is made
Deleting	afterDelete	NO	Runs after the delete operation was made

With the above events can also define business rules in the models:

```
<?php

class Robots extends Phalcon\Mvc\Model
{

    public function beforeDelete()
    {
        if ($this->status == 'A') {
            echo "The robot is active, it can be deleted";
            return false;
        }
        return true;
    }

}
```

2.10.11 Validation Failed Events

Another type of events are available when the data validation process finds any inconsistency:

Operation	Name	Explanation
Insert or Update	notSave	Triggered when the INSERT or UPDATE operation fails for any reason
Insert, Delete or Update	onValidationFails	Triggered when any data manipulation operation fails

2.10.12 Behaviors

Behaviors are shared conducts that several models may adopt in order to re-use code, the ORM provides an API to implement behaviors in your models. Also, you can use the events and callbacks as seen before as an alternative to implement Behaviors with more freedom.

A behavior must be added in the model initializer, a model can have zero or more behaviors:

```
<?php

use Phalcon\Mvc\Model\Behavior\Timestampable;

class Users extends \Phalcon\Mvc\Model
{
    public $id;

    public $name;

    public $created_at;

    public function initialize()
    {
        $this->addBehavior(new Timestampable(
            array(
                'beforeCreate' => array(
                    'field' => 'created_at',
                    'format' => 'Y-m-d'
                )
            )
        ));
    }

}
```

The following built-in behaviors are provided by the framework:

Name	Description
Timestampable	Allows to automatically update a model's attribute saving the datetime when a record is created or updated
SoftDelete	Instead of permanently delete a record it marks the record as deleted changing the value of a flag column

Timestampable

This behavior receives an array of options, the first level key must be an event name indicating when the column must be assigned:

```
<?php

public function initialize()
{
    $this->addBehavior(new Timestampable(
```

```
        array(
            'beforeCreate' => array(
                'field' => 'created_at',
                'format' => 'Y-m-d'
            )
        )
    ));
}
```

Each event can have its own options, 'field' is the name of the column that must be updated, if 'format' is a string it will be used as format of the PHP's function date, format can also be an anonymous function providing you the free to generate any kind timestamp:

```
<?php

public function initialize()
{
    $this->addBehavior(new Timestampable(
        array(
            'beforeCreate' => array(
                'field' => 'created_at',
                'format' => function() {
                    $datetime = new Datetime(new DateTimeZone('Europe/Stockholm'));
                    return $datetime->format('Y-m-d H:i:sP');
                }
            )
        )
    ));
}
```

If the option 'format' is omitted a timestamp using the PHP's function time, will be used.

SoftDelete

This behavior can be used in the following way:

```
<?php

use Phalcon\Mvc\Model\Behavior\SoftDelete;

class Users extends \Phalcon\Mvc\Model
{

    const DELETED = 'D';

    const NOT_DELETED = 'N';

    public $id;

    public $name;

    public $status;

    public function initialize()
    {
        $this->addBehavior(new SoftDelete(
            array(
                'field' => 'status',
```

```
                'value' => Users::DELETED
            )
        ));
    }

}
```

This behavior accepts two options: 'field' and 'value', 'field' determines what field must be updated and 'value' the value to be deleted. Let's pretend the table 'users' has the following data:

```
mysql> select * from users;
+----+---------+--------+
| id | name    | status |
+----+---------+--------+
|  1 | Lana    | N      |
|  2 | Brandon | N      |
+----+---------+--------+
2 rows in set (0.00 sec)
```

If we delete any of the two records the status will be updated instead of delete the record:

```
<?php

Users::findFirst(2)->delete();
```

The operation will result in the following data in the table:

```
mysql> select * from users;
+----+---------+--------+
| id | name    | status |
+----+---------+--------+
|  1 | Lana    | N      |
|  2 | Brandon | D      |
+----+---------+--------+
2 rows in set (0.01 sec)
```

Note that you need to specify the deleted condition in your queries to effectively ignore them as deleted records, this behavior doesn't support that.

Creating your own behaviors

The ORM provides an API to create your own behaviors. A behavior must be a class implementing the *Phalcon\Mvc\Model\BehaviorInterface* Also, Phalon\Mvc\Model\Behavior provides most of the methods needed to ease the implementation of behaviors.

The following behavior is an example, it implements the Blamable behavior which helps identify the user that is performed operations over a model:

```
<?php

use Phalcon\Mvc\ModelInterface,
    Phalcon\Mvc\Model\BehaviorInterface;

class Blameable extends Behavior implements BehaviorInterface
{

    public function notify($eventType, $model)
    {
```

```
        switch ($eventType) {

            case 'afterCreate':
            case 'afterDelete':
            case 'afterUpdate':


                $userName = // ... get the current user from session

                //Store in a log the username - event type and primary key
                file_put_contents('logs/blamable-log.txt', $userName.' '.$eventType.' '.$model->id);

                break;

            default:
                /* ignore the rest of events */
        }
    }

}
```

The former is a very simple behavior, but it illustrates how to create a behavior, now let's add this behavior to a model:

```
<?php

class Profiles extends \Phalcon\Mvc\Model
{

    public function initialize()
    {
        $this->addBehavior(new Blamable());
    }

}
```

A behavior is also capable of intercept missing methods on your models:

```
<?php

use Phalcon\Mvc\Model\Behavior,
    Phalcon\Mvc\Model\BehaviorInterface;

class Sluggable extends Behavior implements BehaviorInterface
{

    public function missingMethod($model, $method, $arguments=array())
    {
        // if the method is 'getSlug' convert the title
        if ($method == 'getSlug') {
            return Phalcon\Tag::friendlyTitle($model->title);
        }
    }

}
```

Call that method on a model that implements Sluggable returns a SEO friendly title:

```
<?php
```

```
$title = $post->getSlug();
```

Using Traits as behaviors

Starting from PHP 5.4 you can use Traits to re-use code in your classes, this is another way to implement custom behaviors. The following trait implements a simple version of the Timestampable behavior:

```
<?php

trait MyTimestampable
{

    public function beforeCreate()
    {
        $this->created_at = date('r');
    }

    public function beforeUpdate()
    {
        $this->updated_at = date('r');
    }

}
```

Then you can use it in your model as follows:

```
<?php

class Products extends \Phalcon\Mvc\Model
{
    use MyTimestampable;
}
```

2.10.13 Transactions

When a process performs multiple database operations, it is often that each step is completed successfully so that data integrity can be maintained. Transactions offer the ability to ensure that all database operations have been executed successfully before the data are committed to the database.

Transactions in Phalcon allow you to commit all operations if they have been executed successfully or rollback all operations if something went wrong.

Manual Transactions

If an application only uses one connection and the transactions aren't very complex, a transaction can be created by just moving the current connection to transaction mode, doing a rollback or commit if the operation is successfully or not:

```
<?php

class RobotsController extends Phalcon\Mvc\Controller
{
    public function saveAction()
    {
        $this->db->begin();
```

```
        $robot = new Robots();

        $robot->name = "WALL·E";
        $robot->created_at = date("Y-m-d");
        if ($robot->save() == false) {
            $this->db->rollback();
            return;
        }

        $robotPart = new RobotParts();
        $robotPart->robots_id = $robot->id;
        $robotPart->type = "head";
        if ($robotPart->save() == false) {
            $this->db->rollback();
            return;
        }

        $this->db->commit();
    }
}
```

Implicit Transactions

Existing relationships can be used to store records and their related instances, this kind of operation implicitly creates a transaction to ensure that data are correctly stored:

```
<?php

$robotPart = new RobotParts();
$robotPart->type = "head";

$robot = new Robots();
$robot->name = "WALL·E";
$robot->created_at = date("Y-m-d");
$robot->robotPart = $robotPart;

$robot->save(); //Creates an implicit transaction to store both records
```

Isolated Transactions

Isolated transactions are executed in a new connection ensuring that all the generated SQL, virtual foreign key checking and business rules are isolated from the main connection. This kind of transaction requires a transaction manager that globally manages each transaction created ensuring that it's correctly rollbacked/commited before ending the request:

```
<?php

use Phalcon\Mvc\Model\Transaction\Manager as TxManager,
    Phalcon\Mvc\Model\Transaction\Failed as TxFailed;

try {

    //Create a transaction manager
    $manager = new TxManager();

    // Request a transaction
```

```
    $transaction = $manager->get();

    $robot = new Robots();
    $robot->setTransaction($transaction);
    $robot->name = "WALL·E";
    $robot->created_at = date("Y-m-d");
    if ($robot->save() == false) {
        $transaction->rollback("Cannot save robot");
    }

    $robotPart = new RobotParts();
    $robotPart->setTransaction($transaction);
    $robotPart->robots_id = $robot->id;
    $robotPart->type = "head";
    if ($robotPart->save() == false) {
        $transaction->rollback("Cannot save robot part");
    }

    //Everything goes fine, let's commit the transaction
    $transaction->commit();

} catch(TxFailed $e) {
    echo "Failed, reason: ", $e->getMessage();
}
```

Transactions can be used to delete many records in a consistent way:

```
<?php

use Phalcon\Mvc\Model\Transaction\Manager as TxManager,
    Phalcon\Mvc\Model\Transaction\Failed as TxFailed;

try {

    //Create a transaction manager
    $manager = new TxManager();

    //Request a transaction
    $transaction = $manager->get();

    //Get the robots will be deleted
    foreach (Robots::find("type = 'mechanical'") as $robot) {
        $robot->setTransaction($transaction);
        if ($robot->delete() == false) {
            //Something goes wrong, we should to rollback the transaction
            foreach ($robot->getMessages() as $message) {
                $transaction->rollback($message->getMessage());
            }
        }
    }

    //Everything goes fine, let's commit the transaction
    $transaction->commit();

    echo "Robots were deleted successfully!";

} catch(TxFailed $e) {
    echo "Failed, reason: ", $e->getMessage();
```

```
}
```

Transactions are reused no matter where the transaction object is retrieved. A new transaction is generated only when a commit() or rollback() is performed. You can use the service container to create an overall transaction manager for the entire application:

```
<?php

$di->setShared('transactions', function(){
    return new \Phalcon\Mvc\Model\Transaction\Manager();
});
```

Then access it from a controller or view:

```
<?php

class ProductsController extends \Phalcon\Mvc\Controller
{

    public function saveAction()
    {

        //Obtain the TransactionsManager from the services container
        $manager = $this->di->getTransactions();

        //Or
        $manager = $this->transactions;

        //Request a transaction
        $transaction = $manager->get();

        //...
    }

}
```

While a transaction is active, the transaction manager will always return the same transaction across the application.

2.10.14 Independent Column Mapping

The ORM supports an independent column map, which allows the developer to use different column names in the model to the ones in the table. Phalcon will recognize the new column names and will rename them accordingly to match the respective columns in the database. This is a great feature when one needs to rename fields in the database without having to worry about all the queries in the code. A change in the column map in the model will take care of the rest. For example:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function columnMap()
    {
        //Keys are the real names in the table and
        //the values their names in the application
        return array(
            'id' => 'code',
```

```
            'the_name' => 'theName',
            'the_type' => 'theType',
            'the_year' => 'theYear'
        );
    }

}
```

Then you can use the new names naturally in your code:

```
<?php

//Find a robot by its name
$robot = Robots::findFirst("theName = 'Voltron'");
echo $robot->theName, "\n";

//Get robots ordered by type
$robot = Robots::find(array('order' => 'theType DESC'));
foreach ($robots as $robot) {
    echo 'Code: ', $robot->code, "\n";
}

//Create a robot
$robot = new Robots();
$robot->code = '10101';
$robot->theName = 'Bender';
$robot->theType = 'Industrial';
$robot->theYear = 2999;
$robot->save();
```

Take into consideration the following the next when renaming your columns:

- References to attributes in relationships/validators must use the new names
- Refer the real column names will result in an exception by the ORM

The independent column map allow you to:

- Write applications using your own conventions
- Eliminate vendor prefixes/suffixes in your code
- Change column names without change your application code

2.10.15 Operations over Resultsets

If a resultset is composed of complete objects, the resultset is in the ability to perform operations on the records obtained in a simple manner:

Updating related records

Instead of doing this:

```
<?php

foreach ($robots->getParts() as $part) {
    $part->stock = 100;
    $part->updated_at = time();
```

```
    if ($part->update() == false) {
        foreach ($part->getMessages() as $message) {
            echo $message;
        }
        break;
    }
}
```

you can do this:

```
<?php

$robots->getParts()->update(array(
    'stock' => 100,
    'updated_at' => time()
));
```

'update' also accepts an anonymous function to filter what records must be updated:

```
<?php

$data = array(
    'stock' => 100,
    'updated_at' => time()
);

//Update all the parts except these whose type is basic
$robots->getParts()->update($data, function($part) {
    if ($part->type == Part::TYPE_BASIC) {
        return false;
    }
    return true;
}
```

Deleting related records

Instead of doing this:

```
<?php

foreach ($robots->getParts() as $part) {
    if ($part->delete() == false) {
        foreach ($part->getMessages() as $message) {
            echo $message;
        }
        break;
    }
}
```

you can do this:

```
<?php

$robots->getParts()->delete();
```

'delete' also accepts an anonymous function to filter what records must be deleted:

```php
<?php

//Delete only whose stock is greater or equal than zero
$robots->getParts()->delete(function($part) {
    if ($part->stock < 0) {
        return false;
    }
    return true;
});
```

2.10.16 Record Snapshots

Specific models could be set to maintain a record snapshot when they're queried. You can use this feature to implement auditing or just to know what fields are changed according to the data queried from the persistence:

```php
<?php

class Robots extends Phalcon\Mvc\Model
{
    public function initalize()
    {
        $this->keepSnapshots(true);
    }
}
```

When activating this feature the application consumes a bit more of memory to keep track of the original values obtained from the persistence. In models that have this feature activated you can check what fields changed:

```php
<?php

//Get a record from the database
$robot = Robots::findFirst();

//Change a column
$robot->name = 'Other name';

var_dump($robot->getChangedFields()); // ['name']
var_dump($robot->hasChanged('name')); // true
var_dump($robot->hasChanged('type')); // false
```

2.10.17 Models Meta-Data

To speed up development *Phalcon\Mvc\Model* helps you to query fields and constraints from tables related to models. To achieve this, *Phalcon\Mvc\Model\MetaData* is available to manage and cache table meta-data.

Sometimes it is necessary to get those attributes when working with models. You can get a meta-data instance as follows:

```php
<?php

$robot = new Robots();

// Get Phalcon\Mvc\Model\Metadata instance
$metaData = $robot->getModelsMetaData();

// Get robots fields names
```

```
$attributes = $metaData->getAttributes($robot);
print_r($attributes);

// Get robots fields data types
$dataTypes = $metaData->getDataTypes($robot);
print_r($dataTypes);
```

Caching Meta-Data

Once the application is in a production stage, it is not necessary to query the meta-data of the table from the database system each time you use the table. This could be done caching the meta-data using any of the following adapters:

Adapter	Description	API
Memory	This adapter is the default. The meta-data is cached only during the request. When the request is completed, the meta-data are released as part of the normal memory of the request. This adapter is perfect when the application is in development so as to refresh the meta-data in each request containing the new and/or modified fields.	*Phalcon\Mvc\Model\MetaData\Memory*
Session	This adapter stores meta-data in the $_SESSION superglobal. This adapter is recommended only when the application is actually using a small number of models. The meta-data are refreshed every time a new session starts. This also requires the use of session_start() to start the session before using any models.	*Phalcon\Mvc\Model\MetaData\Session*
Apc	The Apc adapter uses the Alternative PHP Cache (APC) to store the table meta-data. You can specify the lifetime of the meta-data with options. This is the most recommended way to store meta-data when the application is in production stage.	*Phalcon\Mvc\Model\MetaData\Apc*
Files	This adapter uses plain files to store meta-data. By using this adapter the disk-reading is increased but the database access is reduced	*Phalcon\Mvc\Model\MetaData\Files*

As other ORM's dependencies, the metadata manager is requested from the services container:

```
<?php

$di['modelsMetadata'] = function() {

    // Create a meta-data manager with APC
    $metaData = new \Phalcon\Mvc\Model\MetaData\Apc(array(
        "lifetime" => 86400,
        "prefix"   => "my-prefix"
    ));

    return $metaData;
};
```

Meta-Data Strategies

As mentioned above the default strategy to obtain the model's meta-data is database introspection. In this strategy, the information schema is used to know the fields in a table, its primary key, nullable fields, data types, etc.

You can change the default meta-data introspection in the following way:

```
<?php

$di['modelsMetadata'] = function() {

    // Instantiate a meta-data adapter
```

```
    $metaData = new \Phalcon\Mvc\Model\MetaData\Apc(array(
        "lifetime" => 86400,
        "prefix"   => "my-prefix"
    ));

    //Set a custom meta-data introspection strategy
    $metaData->setStrategy(new MyInstrospectionStrategy());

    return $metaData;
};
```

Database Introspection Strategy

This strategy doesn't require any customization and is implicitly used by all the meta-data adapters.

Annotations Strategy

This strategy makes use of *annotations* to describe the columns in a model:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    /**
     * @Primary
     * @Identity
     * @Column(type="integer", nullable=false)
     */
    public $id;

    /**
     * @Column(type="string", length=70, nullable=false)
     */
    public $name;

    /**
     * @Column(type="string", length=32, nullable=false)
     */
    public $type;

    /**
     * @Column(type="integer", nullable=false)
     */
    public $year;

}
```

Annotations must be placed in properties that are mapped to columns in the mapped source. Properties without the @Column annotation are handled as simple class attributes.

The following annotations are supported:

Name	Description
Primary	Mark the field as part of the table's primary key
Identity	The field is an auto_increment/serial column
Column	This marks an attribute as a mapped column

The annotation @Column supports the following parameters:

Name	Description
type	The column's type (string, integer, decimal, boolean)
length	The column's length if any
nullable	Set whether the column accepts null values or not

The annotations strategy could be set up this way:

```
<?php

$di['modelsMetadata'] = function() {

    // Instantiate a meta-data adapter
    $metaData = new \Phalcon\Mvc\Model\MetaData\Apc(array(
        "lifetime" => 86400,
        "prefix"   => "my-prefix"
    ));

    //Set a custom meta-data database introspection
    $metaData->setStrategy(new \Phalcon\Mvc\Model\MetaData\Strategy\Annotations());

    return $metaData;
};
```

Manual Meta-Data

Phalcon can obtain the metadata for each model automatically without the developer must set them manually using any of the introspection strategies presented above.

The developer also has the option of define the metadata manually. This strategy overrides any strategy set in the meta-data manager. New columns added/modified/removed to/from the mapped table must be added/modified/removed also for everything to work properly.

The following example shows how to define the meta-data manually:

```
<?php

use Phalcon\Mvc\Model\MetaData,
    Phalcon\Db\Column;

class Robots extends \Phalcon\Mvc\Model
{

    public function metaData()
    {
        return array(

            //Every column in the mapped table
            MetaData::MODELS_ATTRIBUTES => array(
                'id', 'name', 'type', 'year'
            ),

            //Every column part of the primary key
            MetaData::MODELS_PRIMARY_KEY => array(
                'id'
            ),

            //Every column that isn't part of the primary key
```

```
            MetaData::MODELS_NON_PRIMARY_KEY => array(
                'name', 'type', 'year'
            ),

            //Every column that doesn't allows null values
            MetaData::MODELS_NOT_NULL => array(
                'id', 'name', 'type', 'year'
            ),

            //Every column and their data types
            MetaData::MODELS_DATA_TYPES => array(
                'id' => Column::TYPE_INTEGER,
                'name' => Column::TYPE_VARCHAR,
                'type' => Column::TYPE_VARCHAR,
                'year' => Column::TYPE_INTEGER
            ),

            //The columns that have numeric data types
            MetaData::MODELS_DATA_TYPES_NUMERIC => array(
                'id' => true,
                'year' => true,
            ),

            //The identity column, use boolean false if the model doesn't have
            //an identity column
            MetaData::MODELS_IDENTITY_COLUMN => 'id',

            //How every column must be bound/casted
            MetaData::MODELS_DATA_TYPES_BIND => array(
                'id' => Column::BIND_PARAM_INT,
                'name' => Column::BIND_PARAM_STR,
                'type' => Column::BIND_PARAM_STR,
                'year' => Column::BIND_PARAM_INT,
            ),

            //Fields that must be ignored from INSERT SQL statements
            MetaData::MODELS_AUTOMATIC_DEFAULT_INSERT => array(
                'year' => true
            ),

            //Fields that must be ignored from UPDATE SQL statements
            MetaData::MODELS_AUTOMATIC_DEFAULT_UPDATE => array(
                'year' => true
            )

        );
    }

}
```

2.10.18 Pointing to a different schema

If a model is mapped to a table that is in a different schemas/databases than the default. You can use the getSchema method to define that:

```
<?php
```

```
class Robots extends \Phalcon\Mvc\Model
{

    public function getSchema()
    {
        return "toys";
    }

}
```

2.10.19 Setting multiple databases

In Phalcon, all models can belong to the same database connection or have an individual one. Actually, when *Phalcon\Mvc\Model* needs to connect to the database it requests the "db" service in the application's services container. You can overwrite this service setting it in the initialize method:

```
<?php

//This service returns a MySQL database
$di->set('dbMysql', function() {
     return new \Phalcon\Db\Adapter\Pdo\Mysql(array(
        "host" => "localhost",
        "username" => "root",
        "password" => "secret",
        "dbname" => "invo"
    ));
});

//This service returns a PostgreSQL database
$di->set('dbPostgres', function() {
     return new \Phalcon\Db\Adapter\Pdo\PostgreSQL(array(
        "host" => "localhost",
        "username" => "postgres",
        "password" => "",
        "dbname" => "invo"
    ));
});
```

Then, in the Initialize method, we define the connection service for the model:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function initialize()
    {
        $this->setConnectionService('dbPostgres');
    }

}
```

But Phalcon offers you more flexibility, you can define the connection that must be used to 'read' and for 'write'. This is specially useful to balance the load to your databases implementing a master-slave architecture:

```
<?php
```

```
class Robots extends \Phalcon\Mvc\Model
{

    public function initialize()
    {
        $this->setReadConnectionService('dbSlave');
        $this->setWriteConnectionService('dbMaster');
    }

}
```

The ORM also provides Horizontal Sharding facilities, by allowing you to implement any 'shard' selection according to the query conditions:

```
<?php

class Robots extends Phalcon\Mvc\Model
{
    public function selectReadConnection($intermediate, $bindParams, $bindTypes)
    {
        //Check if there is a 'where' clause in the select
        if (isset($intermediate['where'])) {

            $conditions = $intermediate['where'];

            //Choose the possible shard according to the conditions
            if ($conditions['left']['name'] == 'id') {
                $id = $conditions['right']['value'];
                if ($id > 0 && $id < 10000) {
                    return $this->getDI()->get('dbShard1');
                }
                if ($id > 10000) {
                    return $this->getDI()->get('dbShard2');
                }
            }
        }

        //Use a default shard
        return $this->getDI()->get('dbShard0');
    }

}
```

The method 'selectReadConnection' is called to choose the right connection, this method intercepts any new query executed:

```
<?php

$robot = Robots::findFirst('id = 101');
```

2.10.20 Logging Low-Level SQL Statements

When using high-level abstraction components such as *Phalcon\Mvc\Model* to access a database, it is difficult to understand which statements are finally sent to the database system. *Phalcon\Mvc\Model* is supported internally by *Phalcon\Db*. *Phalcon\Logger* interacts with *Phalcon\Db*, providing logging capabilities on the database abstraction layer, thus allowing us to log SQL statements as they happen.

```
<?php

$di->set('db', function() {

    $eventsManager = new \Phalcon\Events\Manager();

    $logger = new \Phalcon\Logger\Adapter\File("app/logs/debug.log");

    //Listen all the database events
    $eventsManager->attach('db', function($event, $connection) use ($logger) {
        if ($event->getType() == 'beforeQuery') {
            $logger->log($connection->getSQLStatement(), \Phalcon\Logger::INFO);
        }
    });

    $connection = new \Phalcon\Db\Adapter\Pdo\Mysql(array(
        "host" => "localhost",
        "username" => "root",
        "password" => "secret",
        "dbname" => "invo"
    ));

    //Assign the eventsManager to the db adapter instance
    $connection->setEventsManager($eventsManager);

    return $connection;
});
```

As models access the default database connection, all SQL statements that are sent to the database system will be logged in the file:

```
<?php

$robot = new Robots();
$robot->name = "Robby the Robot";
$robot->created_at = "1956-07-21"
if ($robot->save() == false) {
    echo "Cannot save robot";
}
```

As above, the file *app/logs/db.log* will contain something like this:

```
[Mon, 30 Apr 12 13:47:18 -0500][DEBUG][Resource Id #77] INSERT INTO robots
(name, created_at) VALUES ('Robby the Robot', '1956-07-21')
```

2.10.21 Profiling SQL Statements

Thanks to *Phalcon\Db*, the underlying component of *Phalcon\Mvc\Model*, it's possible to profile the SQL statements generated by the ORM in order to analyze the performance of database operations. With this you can diagnose performance problems and to discover bottlenecks.

```
<?php

$di->set('profiler', function(){
    return new \Phalcon\Db\Profiler();
}, true);
```

```
$di->set('db', function() use ($di) {

    $eventsManager = new \Phalcon\Events\Manager();

    //Get a shared instance of the DbProfiler
    $profiler = $di->getProfiler();

    //Listen all the database events
    $eventsManager->attach('db', function($event, $connection) use ($profiler) {
        if ($event->getType() == 'beforeQuery') {
            $profiler->startProfile($connection->getSQLStatement());
        }
        if ($event->getType() == 'afterQuery') {
            $profiler->stopProfile();
        }
    });

    $connection = new \Phalcon\Db\Adapter\Pdo\Mysql(array(
        "host" => "localhost",
        "username" => "root",
        "password" => "secret",
        "dbname" => "invo"
    ));

    //Assign the eventsManager to the db adapter instance
    $connection->setEventsManager($eventsManager);

    return $connection;
});
```

Profiling some queries:

```
<?php

// Send some SQL statements to the database
Robots::find();
Robots::find(array("order" => "name");
Robots::find(array("limit" => 30);

//Get the generated profiles from the profiler
$profiles = $di->get('profiler')->getProfiles();

foreach ($profiles as $profile) {
   echo "SQL Statement: ", $profile->getSQLStatement(), "\n";
   echo "Start Time: ", $profile->getInitialTime(), "\n";
   echo "Final Time: ", $profile->getFinalTime(), "\n";
   echo "Total Elapsed Time: ", $profile->getTotalElapsedSeconds(), "\n";
}
```

Each generated profile contains the duration in miliseconds that each instruction takes to complete as well as the generated SQL statement.

2.10.22 Injecting services into Models

You may be required to access the application services within a model, the following example explains how to do that:

```php
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function notSave()
    {
        //Obtain the flash service from the DI container
        $flash = $this->getDI()->getFlash();

        //Show validation messages
        foreach ($this->getMesages() as $message) {
            $flash->error((string) $message);
        }
    }

}
```

The "notSave" event is triggered every time that a "create" or "update" action fails. So we're flashing the validation messages obtaining the "flash" service from the DI container. By doing this, we don't have to print messages after each save.

2.10.23 Disabling/Enabling Features

In the ORM we have implemented a mechanism that allow you to enable/disable specific features or options globally on the fly. According to how you use the ORM you can disable that you aren't using. These options can also be temporarily disabled if required:

```php
<?php

\Phalcon\Mvc\Model::setup(array(
    'events' => false,
    'columnRenaming' => false
));
```

The available options are:

Option	Description
events	Enables/Disables callbacks, hooks and event notifications from all the models
columnRenaming	Enables/Disables the column renaming
notNullValidations	The ORM automatically validate the not null columns present in the mapped table
virtualForeignKeys	Enables/Disables the virtual foreign keys

2.10.24 Stand-Alone component

Using *Phalcon\Mvc\Model* in a stand-alone mode can be demonstrated below:

```php
<?php

$di = new Phalcon\DI();

//Setup a connection
$di->set('db', new \Phalcon\Db\Adapter\Pdo\Sqlite(array(
    "dbname" => "sample.db"
)));
```

```
//Set a models manager
$di->set('modelsManager', new \Phalcon\Mvc\Model\Manager());

//Use the memory meta-data adapter or other
$di->set('modelsMetadata', new \Phalcon\Mvc\Model\Metadata\Memory());

class Robots extends Phalcon\Mvc\Model
{

}

echo Robots::count();
```

2.11 Phalcon Query Language (PHQL)

Phalcon Query Language, PhalconQL or simply PHQL is a high-level, object-oriented SQL dialect that allows to write queries using a standardized SQL-like language. PHQL is implemented as a parser (written in C) that translates syntax in that of the target RDBMS.

To achieve the highest performance possible, Phalcon provides a parser that uses the same technology as SQLite. This technology provides a small in-memory parser with a very low memory footprint that is also thread-safe.

The parser first checks the syntax of the pass PHQL statement, then builds an intermediate representation of the statement and finally it converts it to the respective SQL dialect of the target RDBMS.

In PHQL, we've implemented a set of features to make your access to databases more secure:

- Bound parameters are part of the PHQL language helping you to secure your code
- PHQL only allows one SQL statement to be executed per call preventing injections
- PHQL ignores all SQL comments which are often used in SQL injections
- PHQL only allows data manipulation statements, avoiding altering or dropping tables/databases by mistake or externally without authorization
- PHQL implements a high-level abstraction allowing you to handle models as tables and class attributes as fields

2.11.1 Usage Example

To better explain how PHQL works consider the following example. We have two models "Cars" and "Brands":

```
<?php

class Cars extends Phalcon\Mvc\Model
{
    public $id;

    public $name;

    public $brand_id;

    public $price;

    public $year;

    public $style;
```

```
    /**
     * This model is mapped to the table sample_cars
     */
    public function getSource()
    {
        return 'sample_cars';
    }

    /**
     * A car only has a Brand, but a Brand have many Cars
     */
    public function initialize()
    {
        $this->belongsTo('brand_id', 'Brands', 'id');
    }
}
```

And every Car has a Brand, so a Brand has many Cars:

```
<?php

class Brands extends Phalcon\Mvc\Model
{

    public $id;

    public $name;

    /**
     * The model Brands is mapped to the "sample_brands" table
     */
    public function getSource()
    {
        return 'sample_brands';
    }

    /**
     * A Brand can have many Cars
     */
    public function initialize()
    {
        $this->hasMany('id', 'Cars', 'brand_id');
    }
}
```

2.11.2 Creating PHQL Queries

PHQL queries can be created just instantiating the class *Phalcon\Mvc\Model\Query*:

```
<?php

// Instantiate the Query
$query = new Phalcon\Mvc\Model\Query("SELECT * FROM Cars");

// Pass the DI container
$query->setDI($di);
```

```
// Execute the query returning a result if any
$cars = $query->execute();
```

From a controller or a view, it's easy create/execute them using a injected *models manager*:

```
<?php

//Executing a simple query
$query = $this->modelsManager->createQuery("SELECT * FROM Cars");
$cars = $query->execute();

//With bound parameters
$query = $this->modelsManager->createQuery("SELECT * FROM Cars WHERE name = :name:");
$cars = $query->execute(array(
    'name' => 'Audi'
));
```

Or simply execute it:

```
<?php

//Executing a simple query
$cars = $this->modelsManager->executeQuery("SELECT * FROM Cars");

//Executing with bound parameters
$cars = $this->modelsManager->executeQuery("SELECT * FROM Cars WHERE name = :name:", array(
    'name' => 'Audi'
));
```

2.11.3 Selecting Records

As the familiar SQL, PHQL allows querying of records using the SELECT statement we know, except that instead of specifying tables, we use the models classes:

```
<?php

$query = $manager->createQuery("SELECT * FROM Cars ORDER BY Cars.name");
$query = $manager->createQuery("SELECT Cars.name FROM Cars ORDER BY Cars.name");
```

Classes in namespaces are also allowed:

```
<?php

$phql = "SELECT * FROM Formula\Cars ORDER BY Formula\Cars.name";
$query = $manager->createQuery($phql);

$phql = "SELECT Formula\Cars.name FROM Formula\Cars ORDER BY Formula\Cars.name";
$query = $manager->createQuery($phql);

$phql = "SELECT c.name FROM Formula\Cars c ORDER BY c.name";
$query = $manager->createQuery($phql);
```

Most of the SQL standard is supported by PHQL even nonstandard directives as LIMIT:

```
<?php

$phql   = "SELECT c.name FROM Cars AS c "
```

```
    . "WHERE c.brand_id = 21 ORDER BY c.name LIMIT 100";
$query = $manager->createQuery($phql);
```

Result Types

Depending on the type of columns we query, the result type will vary. If you retrieve a single whole object, then the object returned is a *Phalcon\Mvc\Model\Resultset\Simple*. This kind of resultset is a set of complete model objects:

```
<?php

$phql = "SELECT c.* FROM Cars AS c ORDER BY c.name";
$cars = $manager->executeQuery($phql);
foreach ($cars as $car) {
    echo "Name: ", $car->name, "\n";
}
```

This is exactly the same as:

```
<?php

$cars = Cars::find(array("order" => "name"));
foreach ($cars as $car) {
    echo "Name: ", $car->name, "\n";
}
```

Complete objects can be modified and re-saved in the database because they represent a complete record of the associated table. There are other types of queries that do not return complete objects, for example:

```
<?php

$phql = "SELECT c.id, c.name FROM Cars AS c ORDER BY c.name";
$cars = $manager->executeQuery($phql);
foreach ($cars as $car) {
    echo "Name: ", $car->name, "\n";
}
```

We are only requesting some fields in the table therefore those cannot be considered an entire object. In this case, also returns a resulset type *Phalcon\Mvc\Model\Resultset\Simple*. However, each element is a standard object that only contain the two columns that were requested.

These values that don't represent complete objects we call them scalars. PHQL allows you to query all types of scalars: fields, functions, literals, expressions, etc..:

```
<?php

$phql = "SELECT CONCAT(c.id, ' ', c.name) AS id_name FROM Cars AS c ORDER BY c.name";
$cars = $manager->executeQuery($phql);
foreach ($cars as $car) {
    echo $car->id_name, "\n";
}
```

As we can query complete objects or scalars, also we can query both at once:

```
<?php

$phql   = "SELECT c.price*0.16 AS taxes, c.* FROM Cars AS c ORDER BY c.name";
$result = $manager->executeQuery($phql);
```

The result in this case is an object *Phalcon\Mvc\Model\Resultset\Complex*. This allows access to both complete objects and scalars at once:

```
<?php

foreach ($result as $row) {
    echo "Name: ", $row->cars->name, "\n";
    echo "Price: ", $row->cars->price, "\n";
    echo "Taxes: ", $row->taxes, "\n";
}
```

Scalars are mapped as properties of each "row", while complete objects are mapped as properties with the name of its related model.

Joins

It's easy to request records from multiple models using PHQL. Most kinds of Joins are supported. As we defined relationships in the models. PHQL adds these conditions automatically:

```
<?php

$phql  = "SELECT Cars.name AS car_name, Brands.name AS brand_name FROM Cars JOIN Brands";
$rows = $manager->executeQuery($phql);
foreach ($rows as $row) {
    echo $row->car_name, "\n";
    echo $row->brand_name, "\n";
}
```

By default, a INNER JOIN is assumed. You can specify the type of JOIN in the query:

```
<?php

$phql = "SELECT Cars.*, Brands.* FROM Cars INNER JOIN Brands";
$rows = $manager->executeQuery($phql);

$phql = "SELECT Cars.*, Brands.* FROM Cars LEFT JOIN Brands";
$rows = $manager->executeQuery($phql);

$phql = "SELECT Cars.*, Brands.* FROM Cars LEFT OUTER JOIN Brands";
$rows = $manager->executeQuery($phql);

$phql = "SELECT Cars.*, Brands.* FROM Cars CROSS JOIN Brands";
$rows = $manager->executeQuery($phql);
```

Also is possibly, manually set the conditions of the JOIN:

```
<?php

$phql = "SELECT Cars.*, Brands.* FROM Cars INNER JOIN Brands ON Brands.id = Cars.brands_id";
$rows = $manager->executeQuery($phql);
```

Also, the joins can be created using multiple tables in the FROM clause:

```
<?php

$phql = "SELECT Cars.*, Brands.* FROM Cars, Brands WHERE Brands.id = Cars.brands_id";
$rows = $manager->executeQuery($phql);
foreach ($rows as $row) {
    echo "Car: ", $row->cars->name, "\n";
```

```
    echo "Brand: ", $row->brands->name, "\n";
}
```

If an alias is used to rename the models in the query, those will be used to name the attributes in the every row of the result:

```
<?php

$phql = "SELECT c.*, b.* FROM Cars c, Brands b WHERE b.id = c.brands_id";
$rows = $manager->executeQuery($phql);
foreach ($rows as $row) {
    echo "Car: ", $row->c->name, "\n";
    echo "Brand: ", $row->b->name, "\n";
}
```

Aggregations

The following examples show how to use aggregations in PHQL:

```
<?php

// How much are the prices of all the cars?
$phql = "SELECT SUM(price) AS summatory FROM Cars";
$row  = $manager->executeQuery($phql)->getFirst();
echo $row['summatory'];

// How many cars are by each brand?
$phql = "SELECT Cars.brand_id, COUNT(*) FROM Cars GROUP BY Cars.brand_id";
$rows = $manager->executeQuery($phql);
foreach ($rows as $row) {
    echo $row->brand_id, ' ', $row["1"], "\n";
}

// How many cars are by each brand?
$phql = "SELECT Brands.name, COUNT(*) FROM Cars JOIN Brands GROUP BY 1";
$rows = $manager->executeQuery($phql);
foreach ($rows as $row) {
    echo $row->name, ' ', $row["1"], "\n";
}

$phql = "SELECT MAX(price) AS maximum, MIN(price) AS minimum FROM Cars";
$rows = $manager->executeQuery($phql);
foreach ($rows as $row) {
    echo $row["maximum"], ' ', $row["minimum"], "\n";
}

// Count distinct used brands
$phql = "SELECT COUNT(DISTINCT brand_id) AS brandId FROM Cars";
$rows = $manager->executeQuery($phql);
foreach ($rows as $row) {
    echo $row->brandId, "\n";
}
```

Conditions

Conditions allow us to filter the set of records we want to query. The WHERE clause allows to do that:

```php
<?php

// Simple conditions
$phql = "SELECT * FROM Cars WHERE Cars.name = 'Lamborghini Espada'";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.price > 10000";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE TRIM(Cars.name) = 'Audi R8'";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.name LIKE 'Ferrari%'";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.name NOT LIKE 'Ferrari%'";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.price IS NULL";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.id IN (120, 121, 122)";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.id NOT IN (430, 431)";
$cars = $manager->executeQuery($phql);

$phql = "SELECT * FROM Cars WHERE Cars.id BETWEEN 1 AND 100";
$cars = $manager->executeQuery($phql);
```

Also, as part of PHQL, prepared parameters automatically escape the input data, introducing more security:

```php
<?php

$phql = "SELECT * FROM Cars WHERE Cars.name = :name:";
$cars = $manager->executeQuery($phql, array("name" => 'Lamborghini Espada'));

$phql = "SELECT * FROM Cars WHERE Cars.name = ?0";
$cars = $manager->executeQuery($phql, array(0 => 'Lamborghini Espada'));
```

2.11.4 Inserting Data

With PHQL is possible insert data using the familiar INSERT statement:

```php
<?php

// Inserting without columns
$phql = "INSERT INTO Cars VALUES (NULL, 'Lamborghini Espada', "
      . "7, 10000.00, 1969, 'Grand Tourer')";
$manager->executeQuery($phql);

// Specifyng columns to insert
$phql = "INSERT INTO Cars (name, brand_id, year, style) "
      . "VALUES ('Lamborghini Espada', 7, 1969, 'Grand Tourer')";
$manager->executeQuery($phql);

// Inserting using placeholders
```

```
$phql = "INSERT INTO Cars (name, brand_id, year, style) "
      . "VALUES (:name:, :brand_id:, :year:, :style)";
$manager->executeQuery($sql,
    array(
        'name'     => 'Lamborghini Espada',
        'brand_id' => 7,
        'year'     => 1969,
        'style'    => 'Grand Tourer',
    )
);
```

Phalcon not just only transform the PHQL statements into SQL. All events and business rules defined in the model are executed as if we created individual objects manually. Let's add a business rule on the model cars. A car cannot cost less than $ 10,000:

```
<?php

use Phalcon\Mvc\Model\Message;

class Cars extends Phalcon\Mvc\Model
{

    public function beforeCreate()
    {
        if ($this->price < 10000)
        {
            $this->appendMessage(new Message("A car cannot cost less than $ 10,000"));
            return false;
        }
    }

}
```

If we made the following INSERT in the models Cars, the operation will not be successful because the price does not meet the business rule that we implemented:

```
<?php

$phql   = "INSERT INTO Cars VALUES (NULL, 'Nissan Versa', 7, 9999.00, 2012, 'Sedan')";
$result = $manager->executeQuery($phql);
if ($result->success() == false)
{
    foreach ($result->getMessages() as $message)
    {
        echo $message->getMessage();
    }
}
```

2.11.5 Updating Data

Updating rows is very similar than inserting rows. As you may know, the instruction to update records is UPDATE. When a record is updated the events related to the update operation will be executed for each row.

```
<?php

// Updating a single column
$phql = "UPDATE Cars SET price = 15000.00 WHERE id = 101";
```

```
$manager->executeQuery($phql);

// Updating multiples columns
$phql = "UPDATE Cars SET price = 15000.00, type = 'Sedan' WHERE id = 101";
$manager->executeQuery($phql);

// Updating multiples rows
$phql = "UPDATE Cars SET price = 7000.00, type = 'Sedan' WHERE brands_id > 5";
$manager->executeQuery($phql);

// Using placeholders
$phql = "UPDATE Cars SET price = ?0, type = ?1 WHERE brands_id > ?2";
$manager->executeQuery($phql, array(
    0 => 7000.00,
    1 => 'Sedan',
    2 => 5
));
```

An UPDATE statement performs the update in two phases:

- First, if the UPDATE has a WHERE clause it retrieves all the objects that match these criteria,
- Second, based on the queried objects it updates/changes the requested attributes storing them to the relational database

This way of operation allows that events, virtual foreign keys and validations take part of the updating process. In summary, the following code:

```
<?php

$phql = "UPDATE Cars SET price = 15000.00 WHERE id > 101";
$success = $manager->executeQuery($phql);
```

is somewhat equivalent to:

```
<?php

$messages = null;

$process = function() use (&$messages) {
    foreach (Cars::find("id > 101") as $car) {
        $car->price = 15000;
        if ($car->save() == false) {
            $messages = $car->getMessages();
            return false;
        }
    }
    return true;
}

$success = $process();
```

2.11.6 Deleting Data

When a record is deleted the events related to the delete operation will be executed for each row:

```
<?php
```

```
// Deleting a single row
$phql = "DELETE FROM Cars WHERE id = 101";
$manager->executeQuery($phql);

// Deleting multiple rows
$phql = "DELETE FROM Cars WHERE id > 100";
$manager->executeQuery($phql);

// Using placeholders
$phql = "DELETE FROM Cars WHERE id BETWEEN :initial: AND :final:";
$manager->executeQuery(
    $phql,
    array(
        'initial' => 1,
        'final' => 100
    )
);
```

DELETE operations are also executed in two phases like UPDATEs.

2.11.7 Creating queries using the Query Builder

A builder is available to create PHQL queries without the need to write PHQL statements, this is also IDE friendly:

```
<?php

$robots = $this->modelsManager->createBuilder()
    ->from('Robots')
    ->join('RobotsParts')
    ->limit(20)
    ->order('Robots.name')
    ->getQuery()
    ->execute();
```

That is the same as:

```
<?php

$phql = "SELECT Robots.*
    FROM Robots JOIN RobotsParts p
    ORDER BY Robots.name LIMIT 20";
$result = $manager->executeQuery($phql);
```

More examples of the builder:

```
<?php

$builder->from('Robots')
// 'SELECT Robots.* FROM Robots'

// 'SELECT Robots.*, RobotsParts.* FROM Robots, RobotsParts'
$builder->from(array('Robots', 'RobotsParts'))

// 'SELECT * FROM Robots'
$phql = $builder->columns('*')
                ->from('Robots')

// 'SELECT id FROM Robots'
```

```
$builder->columns('id')
        ->from('Robots')

// 'SELECT id, name FROM Robots'
$builder->columns(array('id', 'name'))
        ->from('Robots')

// 'SELECT Robots.* FROM Robots WHERE Robots.name = "Voltron"'
$builder->from('Robots')
        ->where('Robots.name = "Voltron"')

// 'SELECT Robots.* FROM Robots WHERE Robots.id = 100'
$builder->from('Robots')
        ->where(100)

// 'SELECT Robots.* FROM Robots WHERE Robots.type = "virtual" AND Robots.id > 50'
$builder->from('Robots')
        ->where('type = "virtual"')
        ->andWhere('id > 50')

// 'SELECT Robots.* FROM Robots WHERE Robots.type = "virtual" OR Robots.id > 50'
$builder->from('Robots')
        ->where('type = "virtual"')
        ->orWhere('id > 50')

// 'SELECT Robots.* FROM Robots GROUP BY Robots.name'
$builder->from('Robots')
        ->groupBy('Robots.name')

// 'SELECT Robots.* FROM Robots GROUP BY Robots.name, Robots.id'
$builder->from('Robots')
        ->groupBy(array('Robots.name', 'Robots.id'))

// 'SELECT Robots.name, SUM(Robots.price) FROM Robots GROUP BY Robots.name'
$builder->columns(array('Robots.name', 'SUM(Robots.price)'))
    ->from('Robots')
    ->groupBy('Robots.name')

// 'SELECT Robots.name, SUM(Robots.price) FROM Robots
// GROUP BY Robots.name HAVING SUM(Robots.price) > 1000'
$builder->columns(array('Robots.name', 'SUM(Robots.price)'))
    ->from('Robots')
    ->groupBy('Robots.name')
    ->having('SUM(Robots.price) > 1000')

// 'SELECT Robots.* FROM Robots JOIN RobotsParts');
$builder->from('Robots')
    ->join('RobotsParts')

// 'SELECT Robots.* FROM Robots JOIN RobotsParts AS p');
$builder->from('Robots')
    ->join('RobotsParts', null, 'p')

// 'SELECT Robots.* FROM Robots JOIN RobotsParts ON Robots.id = RobotsParts.robots_id AS p');
$builder->from('Robots')
    ->join('RobotsParts', 'Robots.id = RobotsParts.robots_id', 'p')

// 'SELECT Robots.* FROM Robots
```

```
// JOIN RobotsParts ON Robots.id = RobotsParts.robots_id AS p
// JOIN Parts ON Parts.id = RobotsParts.parts_id AS t'
$builder->from('Robots')
    ->join('RobotsParts', 'Robots.id = RobotsParts.robots_id', 'p')
    ->join('Parts', 'Parts.id = RobotsParts.parts_id', 't')

// 'SELECT r.* FROM Robots AS r'
$builder->addFrom('Robots', 'r')

// 'SELECT Robots.*, p.* FROM Robots, Parts AS p'
$builder->from('Robots')
    ->addFrom('Parts', 'p')

// 'SELECT r.*, p.* FROM Robots AS r, Parts AS p'
$builder->from(array('r' => 'Robots'))
        ->addFrom('Parts', 'p')

// 'SELECT r.*, p.* FROM Robots AS r, Parts AS p');
$builder->from(array('r' => 'Robots', 'p' => 'Parts'))

// 'SELECT Robots.* FROM Robots LIMIT 10'
$builder->from('Robots')
    ->limit(10)

// 'SELECT Robots.* FROM Robots LIMIT 10 OFFSET 5'
$builder->from('Robots')
        ->limit(10, 5)
```

2.11.8 Escaping Reserved Words

PHQL has a few reserved words, if you want to use any of them as attributes or models names, you need to escape those words using the cross-database escaping delimiters '[' and ']':

```
<?php

$phql = "SELECT * FROM [Update]";
$result = $manager->executeQuery($phql);

$phql = "SELECT id, [Like] FROM Posts";
$result = $manager->executeQuery($phql);
```

The delimiters are dynamically translated to valid delimiters depending on the database system where the application is currently running on.

2.11.9 PHQL Lifecycle

Being a high-level language, PHQL gives developers the ability to personalize and customize different aspects in order to suit their needs. The following is the life cycle of each PHQL statement executed:

- The PHQL is parsed and converted into an Intermediate Representation (IR) which is independent of the SQL implemented by database system
- The IR is converted to valid SQL according to the database system associated to the model

2.11.10 Using Raw SQL

A database system could offer specific SQL extensions that aren't supported by PHQL, in this case, a raw SQL can be appropiate:

```
<?php

use Phalcon\Mvc\Model\Resultset\Simple as Resultset;

class Robots extends Phalcon\Mvc\Model
{
    public static function findByCreateInterval()
    {
        // A raw SQL statement
        $sql = "SELECT * FROM robots WHERE id > 0";

        // Base model
        $robot = new Robots();

        // Execute the query
        return new Resultset(null, $robot, $robot->getReadConnection()->query($sql));
    }
}
```

If Raw SQL queries are common in your application a generic method could be added to your model:

```
<?php

use Phalcon\Mvc\Model\Resultset\Simple as Resultset;

class Robots extends Phalcon\Mvc\Model
{
    public static function findByRawSql($conditions, $params=null)
    {
        // A raw SQL statement
        $sql = "SELECT * FROM robots WHERE $conditions";

        // Base model
        $robot = new Robots();

        // Execute the query
        return new Resultset(null, $robot, $robot->getReadConnection()->query($sql, $params));
    }
}
```

The above findByRawSql could be used as follows:

```
<?php

$robots = Robots::findByRawSql('id > ?', array(10));
```

2.11.11 Troubleshooting

Some things to keep in mind when using PHQL:

- Classes are case-sensitive, if a class is not defined as it was defined this could lead to an unexpected behavior.
- The correct charset must be defined in the connection to bind parameters with success.

- Aliased classes aren't replaced by full namespaced classes since this only occurs in PHP code and not inside strings

2.12 Caching in the ORM

Every application is different, we could have models whose data change frequently and others that rarely change. Accessing database systems is often one of the most common bottlenecks in terms of performance. This is due to the complex connection/communication processes that PHP must do in each request to obtain data from the database. Therefore, if we want to achieve good performance we need to add some layers of caching where the application requires it.

This chapter explains the possible points where it is possible to implement caching to improve performance. The framework gives you the tools to implement the cache where you demand of it according to the architecture of your application.

2.12.1 Caching Resultsets

A well established technique to avoid the continuous access to the database is to cache resultsets that don't change frequently using a system with faster access (usually memory).

When *Phalcon\Mvc\Model* requires a service to cache resultsets, it will request it to the Dependency Injector Container with the convention name "modelsCache".

As Phalcon provides a component to *cache* any kind of data, we'll explain how to integrate it with Models. First, you must register it as a service in the services container:

```
<?php

//Set the models cache service
$di->set('modelsCache', function() {

        //Cache data for one day by default
        $frontCache = new \Phalcon\Cache\Frontend\Data(array(
                "lifetime" => 86400
        ));

        //Memcached connection settings
        $cache = new \Phalcon\Cache\Backend\Memcache($frontCache, array(
                "host" => "localhost",
                "port" => "11211"
        ));

        return $cache;
});
```

You have complete control in creating and customizing the cache before being used by registering the service as an anonymous function. Once the cache setup is properly defined you could cache resultsets as follows:

```
<?php

// Get products without caching
$products = Products::find();

// Just cache the resultset. The cache will expire in 1 hour (3600 seconds)
$products = Products::find(array(
        "cache" => array("key" => "my-cache")
```

```
));

// Cache the resultset for only for 5 minutes
$products = Products::find(array(
        "cache" => array("key" => "my-cache", "lifetime" => 300)
));

// Using a custom cache
$products = Products::find(array("cache" => $myCache));
```

Caching could be also applied to resultsets generated using relationships:

```
<?php

// Query some post
$post = Post::findFirst();

// Get comments related to a post, also cache it
$comments = $post->getComments(array(
        "cache" => array("key" => "my-key")
));

// Get comments related to a post, setting lifetime
$comments = $post->getComments(array(
        "cache" => array("key" => "my-key", "lifetime" => 3600)
));
```

When a cached resultset needs to be invalidated, you can simply delete it from the cache using the previously specified key.

Note that not all resultsets must be cached. Results that change very frequently should not be cached since they are invalidated very quickly and caching in that case impacts performance. Additionally, large datasets that do not change frequently could be cached, but that is a decision that the developer has to make based on the available caching mechanism and whether the performance impact to simply retrieve that data in the first place is acceptable.

2.12.2 Overriding find/findFirst

As seen above, these methods are available in models that inherit *Phalcon\Mvc\Model*:

```
<?php

class Robots extends Phalcon\Mvc\Model
{

        public static function find($parameters=null)
        {
                return parent::find($parameters);
        }

        public static function findFirst($parameters=null)
        {
                return parent::findFirst($parameters);
        }

}
```

By doing this, you're intercepting all the calls to these methods, this way, you can add a cache layer or run the query if there is no cache. For example, a very basic cache implementation, uses a static property to avoid that a record would

be queried several times in a same request:

```
<?php

class Robots extends Phalcon\Mvc\Model
{

        protected static $_cache = array();

        /**
         * Implement a method that returns a string key based
         * on the query parameters
         */
        protected static function _createKey($parameters)
        {
                $uniqueKey = array();
                foreach ($parameters as $key => $value) {
                        if (is_scalar($value)) {
                                $uniqueKey[] = $key . ':' . $value;
                        } else {
                                if (is_array($value)) {
                                        $uniqueKey[] = $key . ':[' . self::_createKey($value) .']';
                                }
                        }
                }
                return join(',', $uniqueKey);
        }

        public static function find($parameters=null)
        {

                //Create an unique key based on the parameters
                $key = self::_createKey($parameters);

                if (!isset(self::$_cache[$key])) {
                        //Store the result in the memory cache
                        self::$_cache[$key] = parent::find($parameters);
                }

                //Return the result in the cache
                return self::$_cache[$key];
        }

        public static function findFirst($parameters=null)
        {
                // ...
        }

}
```

Access the database is several times slower than calculate a cache key, you're free in implement the key generation strategy you find better for your needs. Note that a good key avoids collisions as much as possible, this means that different keys returns unrelated records to the find parameters.

In the above example, we used a cache in memory, it is useful as a first level cache. Once we have the memory cache, we can implement a second level cache layer like APC/XCache or a NoSQL database:

```
<?php
```

```
public static function find($parameters=null)
{

        //Create an unique key based on the parameters
        $key = self::_createKey($parameters);

        if (!isset(self::$_cache[$key])) {

                //We're using APC as second cache
                if (apc_exists($key)) {

                        $data = apc_fetch($key);

                        //Store the result in the memory cache
                        self::$_cache[$key] = $data;

                        return $data;
                }

                //There are no memory or apc cache
                $data = parent::find($parameters);

                //Store the result in the memory cache
                self::$_cache[$key] = $data;

                //Store the result in APC
                apc_store($key, $data);

                return $data;
        }

        //Return the result in the cache
        return self::$_cache[$key];
}
```

This gives you full control on how the the caches must be implemented for each model, if this strategy is common to several models you can create a base class for all of them:

```
<?php

class CacheableModel extends Phalcon\Mvc\Model
{

        protected static function _createKey($parameters)
        {
                // .. create a cache key based on the parameters
        }

        public static function find($parameters=null)
        {
                //.. custom caching strategy
        }

        public static function findFirst($parameters=null)
        {
                //.. custom caching strategy
        }
}
```

Then use this class as base class for each 'Cacheable' model:

```
<?php

class Robots extends CacheableModel
{

}
```

2.12.3 Forcing Cache

Earlier we saw how Phalcon\Mvc\Model has a built-in integration with the caching component provided by the framework. To make a record/resultset cacheable we pass the key 'cache' in the array of parameters:

```
<?php

// Cache the resultset for only for 5 minutes
$products = Products::find(array(
        "cache" => array("key" => "my-cache", "lifetime" => 300)
));
```

This gives us the freedom to cache specific queries, however if we want to cache globally every query performed over the model, we can override the find/findFirst method to force every query to be cached:

```
<?php

class Robots extends Phalcon\Mvc\Model
{

        protected static function _createKey($parameters)
        {
                // .. create a cache key based on the parameters
        }

        public static function find($parameters=null)
        {

                //Convert the parameters to an array
                if (!is_array($parameters)) {
                        $parameters = array($parameters);
                }

                //Check if a cache key wasn't passed
                //and create the cache parameters
                if (!isset($parameters['cache'])) {
                        $parameters['cache'] = array(
                                "key" => self::_createKey($parameters),
                                "lifetime" => 300
                        );
                }

                return parent::find($parameters);
        }

        public static function findFirst($parameters=null)
        {
                //...
        }
```

```
}
```

2.12.4 Caching PHQL Queries

All queries in the ORM, no matter how high level syntax we used to create them are handled internally using PHQL. This language gives you much more freedom to create all kinds of queries. Of course these queries can be cached:

```
<?php

$phql = "SELECT * FROM Cars WHERE name = :name:";

$query = $this->modelsManager->executeQuery($phql);

$query->setCache(array(
        "key" => "cars-by-name",
        "lifetime" => 300
));

$cars = $query->execute(array(
        'name' => 'Audi'
));
```

2.12.5 Reusable Related Records

Some models may have relationships to other models. This allows us to easily check the records that relate to instances in memory:

```
<?php

//Get some invoice
$invoice = Invoices::findFirst();

//Get the customer related to the invoice
$customer = $invoice->customer;

//Print his/her name
echo $customer->name, "\n";
```

This example is very simple, a customer is queried and can be used as required, for example, to show its name. This also applies if we retrieve a set of invoices to show customers that correspond to these invoices:

```
<?php

//Get a set of invoices
// SELECT * FROM invoices
foreach (Invoices::find() as $invoice) {

        //Get the customer related to the invoice
        // SELECT * FROM customers WHERE id = ?
        $customer = $invoice->customer;

        //Print his/her name
        echo $customer->name, "\n";
}
```

A customer may have one or more bills, this means that the customer may be unnecessarily more than once. To avoid this, we could mark the relationship as reusable, this way, we tell the ORM to automatically reuse the records instead of re-querying them again and again:

```
<?php

class Invoices extends \Phalcon\Mvc\Model
{

        public function initialize()
        {
                $this->belongsTo("customers_id", "Customer", "id", array(
                        'reusable' => true
                ));
        }

}
```

This cache works in memory only, this means that cached data are released when the request is terminated. You can add a more sophisticated cache for this scenario overriding the models manager:

```
<?php

class CustomModelsManager extends \Phalcon\Mvc\Model\Manager
{

        /**
         * Returns a reusable object from the cache
         *
         * @param string $modelName
         * @param string $key
         * @return object
         */
        public function getReusableRecords($modelName, $key){

                //If the model is Products use the APC cache
                if ($modelName == 'Products'){
                        return apc_fetch($key);
                }

                //For the rest, use the memory cache
                return parent::getReusableRecords($modelName, $key);
        }

        /**
         * Stores a reusable record in the cache
         *
         * @param string $modelName
         * @param string $key
         * @param mixed $records
         */
        public function setReusableRecords($modelName, $key, $records){

                //If the model is Products use the APC cache
                if ($modelName == 'Products'){
                        apc_store($key, $records);
                        return;
                }
```

```
                //For the rest, use the memory cache
                parent::setReusableRecords($modelName, $key, $records);
        }
}
```

Do not forget to register the custom models manager in the DI:

```
<?php

$di->setShared('modelsManager', function() {
        return new CustomModelsManager();
});
```

2.12.6 Caching Related Records

When a related record is queried, the ORM internally builds the appropiate condition and gets the required records using find/findFirst in the target model according to the following table:

Type	Description | Implicit Method
Belongs-To	Returns a model instance of the related record directly | findFirst
Has-One	Returns a model instance of the related record directly | findFirst
Has-Many	Returns a collection of model instances of the referenced model | find

This means that when you get a related record you could intercept how these data are obtained by implementing the corresponding method:

```
<?php

//Get some invoice
$invoice = Invoices::findFirst();

//Get the customer related to the invoice
$customer = $invoice->customer; // Invoices::findFirst('...');

//Same as above
$customer = $invoice->getCustomer(); // Invoices::findFirst('...');
```

Accordingly, we could replace the findFirst method in the model Invoices and implement the cache we consider most appropriate:

```
<?php

class Invoices extends Phalcon\Mvc\Model
{

        public static function findFirst($parameters=null)
        {
                //.. custom caching strategy
        }
}
```

2.12.7 Caching Related Records Recursively

In this scenario, we assume that everytime we query a result we also retrieve their associated records. If we store the records found together with their related entities perhaps we could reduce a bit the overhead required to obtain all entities:

```php
<?php

class Invoices extends Phalcon\Mvc\Model
{

	protected static function _createKey($parameters)
	{
		// .. create a cache key based on the parameters
	}

	protected static function _getCache($key)
	{
		// returns data from a cache
	}

	protected static function _setCache($key)
	{
		// stores data in the cache
	}

	public static function find($parameters=null)
	{
		//Create a unique key
		$key = self::_createKey($parameters);

		//Check if there are data in the cache
		$results = self::_getCache($key);

		// Valid data is an object
		if (is_object($results)) {
			return $results;
		}

		$results = array();

		$invoices = parent::find($parameters);
		foreach ($invoices as $invoice) {

			//Query the related customer
			$customer = $invoice->customer;

			//Assign it to the record
			$invoice->customer = $customer;

			$results[] = $invoice;
		}

		//Store the invoices in the cache + their customers
		self::_setCache($key, $results);

		return $results;
	}

	public function initialize()
	{
		// add relations and initialize other stuff
	}
}
```

Getting the invoices from the cache already obtains the customer data in just one hit, reducing the overall overhead of the operation. Note that this process can also be performed with PHQL following an alternative solution:

```
<?php

class Invoices extends \Phalcon\Mvc\Model
{

        public function initialize()
        {
                // add relations and initialize other stuff
        }

        protected static function _createKey($conditions, $params)
        {
                // .. create a cache key based on the parameters
        }

        public function getInvoicesCustomers($conditions, $params=null)
        {
                $phql = "SELECT Invoices.*, Customers.*
                FROM Invoices JOIN Customers WHERE " . $conditions;

                $query = $this->getModelsManager()->executeQuery($phql);

                $query->setCache(array(
                        "key" => self::_createKey($conditions, $params),
                        "lifetime" => 300
                ));

                return $query->execute($params);
        }

}
```

2.12.8 Caching based on Conditions

In this scenario, the cache is implemented conditionally according to current conditions received. According to the range where the primary key is located we choose a different cache backend:

Type	Cache Backend
1 - 10000	mongo1
10000 - 20000	mongo2
> 20000	mongo3

The easiest way is adding an static method to the model that chooses the right cache to be used:

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

        public static function queryCache($initial, $final)
        {
                if ($initial >= 1 && $final < 10000) {
                        return self::find(array(
                                'id >= ' . $initial . ' AND id <= '.$final,
                                'cache' => array('service' => 'mongo1')
```

```
                ));
        }
        if ($initial >= 10000 && $final <= 20000) {
                return self::find(array(
                        'id >= ' . $initial . ' AND id <= '.$final,
                        'cache' => array('service' => 'mongo2')
                ));
        }
        if ($initial > 20000) {
                return self::find(array(
                        'id >= ' . $initial,
                        'cache' => array('service' => 'mongo3')
                ));
        }
    }

}
```

This approach solves the problem, however, if we want to add other parameters such orders or conditions we would have to create a more complicated method. Additionally, this method does not work if the data is obtained using related records or a find/findFirst:

```
<?php

$robots = Robots::find('id < 1000');
$robots = Robots::find('id > 100 AND type = "A"');
$robots = Robots::find('(id > 100 AND type = "A") AND id < 2000');

$robots = Robots::find(array(
        '(id > ?0 AND type = "A") AND id < ?1',
        'bind' => array(100, 2000),
        'order' => 'type'
));
```

To achieve this we need to intercept the intermediate representation (IR) generated by the PHQL parser and thus customize the cache everything possible:

The first is create a custom builder, so we can generate a totally customized query:

```
<?php

class CustomQueryBuilder extends Phalcon\Mvc\Model\Query\Builder
{

    public function getQuery()
    {
        $query = new CustomQuery($this->getPhql());
        $query->setDI($this->getDI());
        return $query;
    }

}
```

Instead of directly returning a Phalcon\Mvc\Model\Query, our custom builder returns a CustomQuery instance, this class looks like:

```
<?php

class CustomQuery extends Phalcon\Mvc\Model\Query
```

```
{

        /**
         * The execute method is overrided
         */
        public function execute($params=null, $types=null)
        {
                //Parse the intermediate representation for the SELECT
                $ir = $this->parse();

                //Check if the query has conditions
                if (isset($ir['where'])) {

                        //The fields in the conditions can have any order
                        //We need to recursively check the conditions tree
                        //to find the info we're looking for
                        $visitor = new CustomNodeVisitor();

                        //Recursively visits the nodes
                        $visitor->visit($ir['where']);

                        $initial = $visitor->getInitial();
                        $final = $visitor->getFinal();

                        //Select the cache according to the range
                        //...

                        //Check if the cache has data
                        //...
                }

                //Execute the query
                $result = $this->_executeSelect($ir, $params, $types);

                //cache the result
                //...

                return $result;
        }

}
```

Implementing a helper (CustomNodeVisitor) that recursively checks the conditions looking for fields that tell us the possible range to be used in the cache:

```
<?php

class CustomNodeVisitor
{

        protected $_initial = 0;

        protected $_final = 25000;

        public function visit($node)
        {
                switch ($node['type']) {
```

```
                        case 'binary-op':

                                $left = $this->visit($node['left']);
                                $right = $this->visit($node['right']);
                                if (!$left || !$right) {
                                        return false;
                                }

                                if ($left=='id') {
                                        if ($node['op'] == '>') {
                                                $this->_initial = $right;
                                        }
                                        if ($node['op'] == '=') {
                                                $this->_initial = $right;
                                        }
                                        if ($node['op'] == '>=')        {
                                                $this->_initial = $right;
                                        }
                                        if ($node['op'] == '<') {
                                                $this->_final = $right;
                                        }
                                        if ($node['op'] == '<=')        {
                                                $this->_final = $right;
                                        }
                                }
                                break;

                        case 'qualified':
                                if ($node['name'] == 'id') {
                                        return 'id';
                                }
                                break;

                        case 'literal':
                                return $node['value'];

                        default:
                                return false;
                }
        }

        public function getInitial()
        {
                return $this->_initial;
        }

        public function getFinal()
        {
                return $this->_final;
        }
}
```

Finally, we could replace the find method in the Robots model to use the custom classes we've created:

```
<?php

class Robots extends Phalcon\Mvc\Model
{
        public static function find($parameters=null)
```

```
        {

                if (!is_array($parameters)) {
                        $parameters = array($parameters);
                }

                $builder = new CustomQueryBuilder($parameters);
                $builder->from(get_called_class())

                if (isset($parameters['bind'])) {
                        return $builder->getQuery()->execute($parameters['bind']);
                } else {
                        return $builder->getQuery()->execute();
                }

        }
}
```

2.13 ODM (Object-Document Mapper)

In addition to its ability to *map tables* in relational databases, Phalcon can map documents from NoSQL databases. The ODM offers a CRUD functionality, events, validations among other services.

Due to the absence of SQL queries and planners, NoSQL databases can see real improvements in performance using the Phalcon approach. Additionally, there are no SQL building reducing the possibility of SQL injections.

The following NoSQL databases are supported:

Name	Description
MongoDB	MongoDB is a scalable, high-performance, open source NoSQL database.

2.13.1 Creating Models

A model is a class that extends from *Phalcon\Mvc\Collection*. It must be placed in the models directory. A model file must contain a single class; its class name should be in camel case notation:

```
<?php

class Robots extends \Phalcon\Mvc\Collection
{

}
```

> If you're using PHP 5.4 is recommended declare each column that makes part of the model in order to save memory and reduce the memory allocation.

By default model "Robots" will refer to the collection "robots". If you want to manually specify another name for the mapping collection, you can use the getSource() method:

```
<?php

class Robots extends \Phalcon\Mvc\Collection
{
        public function getSource()
        {
                return "the_robots";
```

```
        }
}
```

2.13.2 Understanding Documents To Objects

Every instance of a model represents a document in the collection. You can easily access collection data by reading object properties. For example, for a collection "robots" with the documents:

```
$ mongo test
MongoDB shell version: 1.8.2
connecting to: test
> db.robots.find()
{ "_id" : ObjectId("508735512d42b8c3d15ec4e1"), "name" : "Astro Boy", "year" : 1952,
        "type" : "mechanical" }
{ "_id" : ObjectId("5087358f2d42b8c3d15ec4e2"), "name" : "Bender", "year" : 1999,
        "type" : "mechanical" }
{ "_id" : ObjectId("508735d32d42b8c3d15ec4e3"), "name" : "Wall-E", "year" : 2008 }
>
```

2.13.3 Models in Namespaces

Namespaces can be used to avoid class name collision. In this case it is necessary to indicate the name of the related collection using getSource:

```
<?php

namespace Store\Toys;

class Robots extends \Phalcon\Mvc\Collection
{

        public function getSource()
        {
                return "robots";
        }

}
```

You could find a certain document by its id and then print its name:

```
<?php

// Find record with _id = "5087358f2d42b8c3d15ec4e2"
$robot = Robots::findById("5087358f2d42b8c3d15ec4e2");

// Prints "Bender"
echo $robot->name;
```

Once the record is in memory, you can make modifications to its data and then save changes:

```
<?php

$robot = Robots::findFirst(array(
        array('name' => 'Astroy Boy')
));
```

```
$robot->name = "Voltron";
$robot->save();
```

2.13.4 Setting a Connection

Connections are retrieved from the services container. By default, Phalcon tries to find the connection in a service called "mongo":

```
<?php

// Simple database connection to localhost
$di->set('mongo', function() {
        $mongo = new Mongo();
        return $mongo->selectDb("store");
}, true);

// Connecting to a domain socket, falling back to localhost connection
$di->set('mongo', function() {
        $mongo = new Mongo("mongodb:///tmp/mongodb-27017.sock,localhost:27017");
        return $mongo->selectDb("store");
}, true);
```

2.13.5 Finding Documents

As *Phalcon\Mvc\Collection* relies on the Mongo PHP extension you have the same facilities to query documents and convert them transparently to model instances:

```
<?php

// How many robots are there?
$robots = Robots::find();
echo "There are ", count($robots), "\n";

// How many mechanical robots are there?
$robots = Robots::find(array(
        array("type" => "mechanical")
));
echo "There are ", count($robots), "\n";

// Get and print mechanical robots ordered by name upward
$robots = Robots::find(array(
        array("type" => "mechanical"),
        "sort" => array("name" => 1)
));

foreach ($robots as $robot) {
        echo $robot->name, "\n";
}

// Get first 100 mechanical robots ordered by name
$robots = Robots::find(array(
        array("type" => "mechanical"),
        "sort" => array("name" => 1),
        "limit" => 100
));
```

```
foreach ($robots as $robot) {
   echo $robot->name, "\n";
}
```

You could also use the findFirst() method to get only the first record matching the given criteria:

```
<?php

// What's the first robot in robots collection?
$robot = Robots::findFirst();
echo "The robot name is ", $robot->name, "\n";

// What's the first mechanical robot in robots collection?
$robot = Robots::findFirst(array(
        array("type" => "mechanical")
));
echo "The first mechanical robot name is ", $robot->name, "\n";
```

Both find() and findFirst() methods accept an associative array specifying the search criteria:

```
<?php

// First robot where type = "mechanical" and year = "1999"
$robot = Robots::findFirst(array(
        "type" => "mechanical",
        "year" => "1999"
));

// All virtual robots ordered by name downward
$robots = Robots::find(array(
        "conditions" => array("type" => "virtual"),
        "sort"       => array("name" => -1)
));
```

The available query options are:

Parameter	Description	Example
conditions	Search conditions for the find operation. Is used to extract only those records that fulfill a specified criterion. By default Phalcon_model assumes the first parameter are the conditions.	"conditions" => array('$gt' => 1990)
sort	Is used to sort the resultset. Use one or more fields as each element in the array, 1 means ordering upwards, -1 downward	"order" => array("name" => -1, "statys" => 1)
limit	Limit the results of the query to results to certain range	"limit" => 10
skip	Skips a number of results	"skip" => 50

If you have experience with SQL databases, you may want to check the SQL to Mongo Mapping Chart.

2.13.6 Aggregations

A model can return calculations using aggregation framework provided by Mongo. The aggregated values are calculate without having to use MapReduce. With this option is easy perform tasks such as totaling or averaging field values:

2.13.7 Creating Updating/Records

The method Phalcon\Mvc\Collection::save() allows you to create/update documents according to whether they already exist in the collection associated with a model. The 'save' method is called internally by the create and update methods of *Phalcon\Mvc\Collection*.

Also the method executes associated validators and events that are defined in the model:

```
<?php

$robot       = new Robots();
$robot->type = "mechanical";
$robot->name = "Astro Boy";
$robot->year = 1952;
if ($robot->save() == false) {
        echo "Umh, We can't store robots right now: \n";
        foreach ($robot->getMessages() as $message) {
                echo $message, "\n";
        }
} else {
        echo "Great, a new robot was saved successfully!";
}
```

The "_id" property is automatically updated with the MongoId object created by the driver:

```
<?php

$robot->save();
echo "The generated id is: ", $robot->getId();
```

Validation Messages

Phalcon\Mvc\Collection has a messaging subsystem that provides a flexible way to output or store the validation messages generated during the insert/update processes.

Each message consists of an instance of the class *Phalcon\Mvc\Model\Message*. The set of messages generated can be retrieved with the method getMessages(). Each message provides extended information like the field name that generated the message or the message type:

```
<?php

if ($robot->save() == false) {
        foreach ($robot->getMessages() as $message) {
                echo "Message: ", $message->getMessage();
                echo "Field: ", $message->getField();
                echo "Type: ", $message->getType();
        }
}
```

Validation Events and Events Manager

Models allow you to implement events that will be thrown when performing an insert or update. They help define business rules for a certain model. The following are the events supported by *Phalcon\Mvc\Collection* and their order of execution:

Operation	Name	Can stop operation?	Explanation
Inserting/Updating	beforeValidation	YES	Is executed before the validation process and the final insert/update to the database
Inserting	beforeValidationOnCreate	YES	Is executed before the validation process only when an insertion operation is being made
Updating	beforeValidationOnUpdate	YES	Is executed before the fields are validated for not nulls or foreign keys when an updating operation is being made
Inserting/Updating	onValidationFails	YES (already stopped)	Is executed before the validation process only when an insertion operation is being made
Inserting	afterValidationOnCreate	YES	Is executed after the validation process when an insertion operation is being made
Updating	afterValidationOnUpdate	YES	Is executed after the validation process when an updating operation is being made
Inserting/Updating	afterValidation	YES	Is executed after the validation process
Inserting/Updating	beforeSave	YES	Runs before the required operation over the database system
Updating	beforeUpdate	YES	Runs before the required operation over the database system only when an updating operation is being made
Inserting	beforeCreate	YES	Runs before the required operation over the database system only when an inserting operation is being made
Updating	afterUpdate	NO	Runs after the required operation over the database system only when an updating operation is being made
Inserting	afterCreate	NO	Runs after the required operation over the database system only when an inserting operation is being made
Inserting/Updating	afterSave	NO	Runs after the required operation over the database system

To make a model to react to an event, we must to implement a method with the same name of the event:

```
<?php

class Robots extends \Phalcon\Mvc\Collection
{

        public function beforeValidationOnCreate()
        {
                echo "This is executed before creating a Robot!";
        }

}
```

Events can be useful to assign values before performing a operation, for example:

```
<?php

class Products extends \Phalcon\Mvc\Collection
{

        public function beforeCreate()
        {
                // Set the creation date
                $this->created_at = date('Y-m-d H:i:s');
        }
```

```
        public function beforeUpdate()
        {
                // Set the modification date
                $this->modified_in = date('Y-m-d H:i:s');
        }

}
```

Additionally, this component is integrated with *Phalcon\Events\Manager*, this means we can create listeners that run when an event is triggered.

```
<?php

$eventsManager = new Phalcon\Events\Manager();

//Attach an anonymous function as a listener for "model" events
$eventsManager->attach('collection', function($event, $robot) {
        if ($event->getType() == 'beforeSave') {
                if ($robot->name == 'Scooby Doo') {
                        echo "Scooby Doo isn't a robot!";
                        return false;
                }
        }
        return true;
});

$robot = new Robots();
$robot->setEventsManager($eventsManager);
$robot->name = 'Scooby Doo';
$robot->year = 1969;
$robot->save();
```

In the example given above the EventsManager only acted as a bridge between an object and a listener (the anonymous function). If we want all objects created in our application use the same EventsManager, then we need to assign this to the Models Manager:

```
<?php

//Registering the collectionManager service
$di->set('collectionManager', function() {

        $eventsManager = new Phalcon\Events\Manager();

        // Attach an anonymous function as a listener for "model" events
        $eventsManager->attach('collection', function($event, $model) {
                if (get_class($model) == 'Robots') {
                        if ($event->getType() == 'beforeSave') {
                                if ($model->name == 'Scooby Doo') {
                                        echo "Scooby Doo isn't a robot!";
                                        return false;
                                }
                        }
                }
                return true;
        });

        // Setting a default EventsManager
        $modelsManager = new Phalcon\Mvc\Collection\Manager();
        $modelsManager->setEventsManager($eventsManager);
```

```
        return $modelsManager;

}, true);
```

Implementing a Business Rule

When an insert, update or delete is executed, the model verifies if there are any methods with the names of the events listed in the table above.

We recommend that validation methods are declared protected to prevent that business logic implementation from being exposed publicly.

The following example implements an event that validates the year cannot be smaller than 0 on update or insert:

```
<?php

class Robots extends \Phalcon\Mvc\Collection
{

        public function beforeSave()
        {
                if ($this->year < 0) {
                        echo "Year cannot be smaller than zero!";
                        return false;
                }
        }

}
```

Some events return false as an indication to stop the current operation. If an event doesn't return anything, *Phalcon\Mvc\Collection* will assume a true value.

Validating Data Integrity

Phalcon\Mvc\Collection provides several events to validate data and implement business rules. The special "validation" event allows us to call built-in validators over the record. Phalcon exposes a few built-in validators that can be used at this stage of validation.

The following example shows how to use it:

```
<?php

use Phalcon\Mvc\Model\Validator\InclusionIn;
use Phalcon\Mvc\Model\Validator\Uniqueness;

class Robots extends \Phalcon\Mvc\Collection
{

        public function validation()
        {

                $this->validate(new InclusionIn(
                        array(
                                "field"  => "type",
                                "domain" => array("Mechanical", "Virtual")
                        )
                ));
```

```
                $this->validate(new Uniqueness(
                        array(
                                "field"   => "name",
                                "message" => "The robot name must be unique"
                        )
                ));

                return $this->validationHasFailed() != true;
        }

}
```

The example given above performs a validation using the built-in validator "InclusionIn". It checks the value of the field "type" in a domain list. If the value is not included in the method, then the validator will fail and return false. The following built-in validators are available:

Name	Explanation	Example
Email	Validates that field contains a valid email format	*Example*
ExclusionIn	Validates that a value is not within a list of possible values	*Example*
InclusionIn	Validates that a value is within a list of possible values	*Example*
Numericality	Validates that a field has a numeric format	*Example*
Regex	Validates that the value of a field matches a regular expression	*Example*
StringLength	Validates the length of a string	*Example*

In addition to the built-in validatiors, you can create your own validators:

```
<?php

class UrlValidator extends \Phalcon\Mvc\Collection\Validator
{

        public function validate($model)
        {
                $field = $this->getOption('field');

                $value    = $model->$field;
                $filtered = filter_var($value, FILTER_VALIDATE_URL);
                if (!$filtered) {
                        $this->appendMessage("The URL is invalid", $field, "UrlValidator");
                        return false;
                }
                return true;
        }

}
```

Adding the validator to a model:

```
<?php

class Customers extends \Phalcon\Mvc\Collection
{

        public function validation()
        {
                $this->validate(new UrlValidator(array(
                        "field"  => "url",
                )));
```

```
                if ($this->validationHasFailed() == true) {
                        return false;
                }
        }

}
```

The idea of creating validators is make them reusable across several models. A validator can also be as simple as:

```
<?php

class Robots extends \Phalcon\Mvc\Collection
{

        public function validation()
        {
                if ($this->type == "Old") {
                        $message = new Phalcon\Mvc\Model\Message(
                                "Sorry, old robots are not allowed anymore",
                                "type",
                                "MyType"
                        );
                        $this->appendMessage($message);
                        return false;
                }
                return true;
        }

}
```

2.13.8 Deleting Records

The method Phalcon\Mvc\Collection::delete() allows to delete a document. You can use it as follows:

```
<?php

$robot = Robots::findFirst();
if ($robot != false) {
        if ($robot->delete() == false) {
                echo "Sorry, we can't delete the robot right now: \n";
                foreach ($robot->getMessages() as $message) {
                        echo $message, "\n";
                }
        } else {
                echo "The robot was deleted successfully!";
        }
}
```

You can also delete many documents by traversing a resultset with a foreach:

```
<?php

$robots = Robots::find(array(
        array("type" => "mechanical")
));
foreach ($robots as $robot) {
        if ($robot->delete() == false) {
                echo "Sorry, we can't delete the robot right now: \n";
```

```
                foreach ($robot->getMessages() as $message) {
                        echo $message, "\n";
                }
        } else {
                echo "The robot was deleted successfully!";
        }
}
```

The following events are available to define custom business rules that can be executed when a delete operation is performed:

Operation	Name	Can stop operation?	Explanation
Deleting	beforeDelete	YES	Runs before the delete operation is made
Deleting	afterDelete	NO	Runs after the delete operation was made

2.13.9 Validation Failed Events

Another type of events is available when the data validation process finds any inconsistency:

Operation	Name	Explanation
Insert or Update	notSave	Triggered when the insert/update operation fails for any reason
Insert, Delete or Update	onValidationFails	Triggered when any data manipulation operation fails

2.13.10 Implicit Ids vs. User Primary Keys

By default PhalconMvcCollection assumes that the _id attribute is automatically generated using MongoIds. If a model uses custom primary keys this behavior can be overriden:

```
<?php

class Robots extends Phalcon\Mvc\Collection
{
        public function initialize()
        {
                $this->useImplicitObjectIds(false);
        }
}
```

2.13.11 Setting multiple databases

In Phalcon, all models can belong to the same database connection or have an individual one. Actually, when *Phalcon\Mvc\Collection* needs to connect to the database it requests the "mongo" service in the application's services container. You can overwrite this service setting it in the initialize method:

```
<?php

// This service returns a mongo database at 192.168.1.100
$di->set('mongo1', function() {
        $mongo = new Mongo("mongodb://scott:nekhen@192.168.1.100");
        return $mongo->selectDb("management");
}, true);

// This service returns a mongo database at localhost
$di->set('mongo2', function() {
```

```
        $mongo = new Mongo("mongodb://localhost");
        return $mongo->selectDb("invoicing");
}, true);
```

Then, in the Initialize method, we define the connection service for the model:

```
<?php

class Robots extends \Phalcon\Mvc\Collection
{
        public function initialize()
        {
                $this->setConnectionService('mongo1');
        }

}
```

2.13.12 Injecting services into Models

You may be required to access the application services within a model, the following example explains how to do that:

```
<?php

class Robots extends \Phalcon\Mvc\Collection
{

        public function notSave()
        {
                // Obtain the flash service from the DI container
                $flash = $this->getDI()->getShared('flash');

                // Show validation messages
                foreach ($this->getMesages() as $message){
                        $flash->error((string) $message);
                }
        }

}
```

The "notSave" event is triggered whenever a "creating" or "updating" action fails. We're flashing the validation messages obtaining the "flash" service from the DI container. By doing this, we don't have to print messages after each saving.

2.14 Using Views

Views represent the user interface of your application. Views are often HTML files with embedded PHP code that perform tasks related solely to the presentation of the data. Views handle the job of providing data to the web browser or other tool that is used to make requests from your application.

The *Phalcon\Mvc\View* is responsible for the managing the view layer of your MVC application.

A hierarchy of files is supported by the component. This hierarchy allows for common layout points (commonly used views), as well as controller named folders defining respective view templates.

2.14.1 Integrating Views with Controllers

Phalcon automatically passes the execution to the view component as soon as a particular controller has completed its cycle. The view component will look in the views folder for a folder named as the same name of the last controller executed and then for a file named as the last action executed. For instance, if a request is made to the URL *http://127.0.0.1/blog/posts/show/301*, Phalcon will parse the URL as follows:

Server Address	127.0.0.1
Phalcon Directory	blog
Controller	posts
Action	show
Parameter	301

The dispatcher will look for a "PostsController" and its action "showAction". A simple controller file for this example:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function showAction($postId)
    {
        // Pass the $postId parameter to the view
        $this->view->setVar("postId", $postId);
    }

}
```

The setVar allows us to create view variables on demand so that they can be used in the view template. The example above demonstrates how to pass the $postId parameter to the respective view template.

Phalcon\Mvc\View uses PHP itself as the template engine, therefore views should have the .phtml extension. If the views directory is *app/views* then view component will find automatically for these 3 view files.

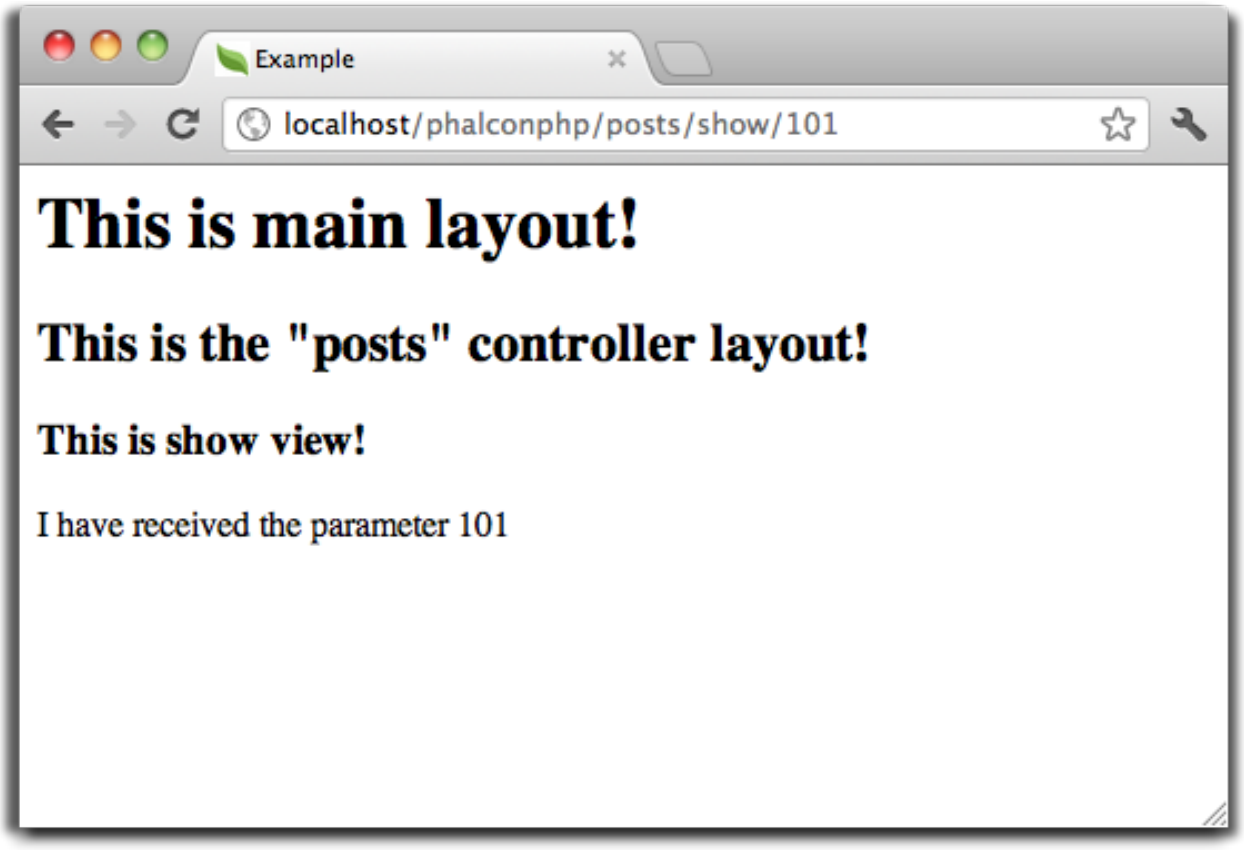
Name	File	Description
Action View	app/views/posts/show.phtml	This is the view related to the action. It only will be shown when the "show" action was executed.
Controller Layout	app/views/layouts/posts.phtml	This is the view related to the controller. It only will be shown for every action executed within the controller "posts". All the code implemented in the layout will be reused for all the actions in this controller.
Main Layout	app/views/index.phtml	This is main action it will be shown for every controller or action executed within the application.

You are not required to implement all of the files mentioned above. *Phalcon\Mvc\View* will simply move to the next view level in the hierarchy of files. If all three view files are implemented, they will be processed as follows:

```
<!-- app/views/posts/show.phtml -->

<h3>This is show view!</h3>

<p>I have received the parameter <?php $postId ?></p>

<!-- app/views/layouts/posts.phtml -->
```

```
<h2>This is the "posts" controller layout!</h2>

<?php echo $this->getContent() ?>

<!-- app/views/index.phtml -->
<html>
    <head>
        <title>Example</title>
    </head>
    <body>

        <h1>This is main layout!</h1>

        <?php echo $this->getContent() ?>

    </body>
</html>
```

Note the lines where the method *$this->getContent()* was called. This method instructs *Phalcon\Mvc\View* on where to inject the contents of the previous view executed in the hierarchy. For the example above, the output will be:



The generated HTML by the request will be:

```
<!-- app/views/index.phtml -->
<html>
    <head>
        <title>Example</title>
    </head>
```

```
    <body>

        <h1>This is main layout!</h1>

        <!-- app/views/layouts/posts.phtml -->

        <h2>This is the "posts" controller layout!</h2>

        <!-- app/views/posts/show.phtml -->

        <h3>This is show view!</h3>

        <p>I have received the parameter 101</p>

    </body>
</html>
```

2.14.2 Using Templates

Templates are views that can be used to share common view code. They act as controller layouts, so you need to place them in the layouts directory.

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{
    public function initialize()
    {
        $this->view->setTemplateAfter('common');
    }

    public function lastAction()
    {
        $this->flash->notice("These are the latest posts");
    }
}

<!-- app/views/index.phtml -->
<!DOCTYPE html>
<html>
    <head>
        <title>Blog's title</title>
    </head>
    <body>
        <?php echo $this->getContent() ?>
    </body>
</html>

<!-- app/views/layouts/common.phtml -->

<ul class="menu">
    <li><a href="/">Home</a></li>
    <li><a href="/articles">Articles</a></li>
    <li><a href="/contact">Contact us</a></li>
</ul>

<div class="content"><?php echo $this->getContent() ?></div>
```

```
<!-- app/views/layouts/posts.phtml -->

<h1>Blog Title</h1>

<?php echo $this->getContent() ?>

<!-- app/views/layouts/posts/last.phtml -->

<article>
    <h2>This is a title</h2>
    <p>This is the post content</p>
</article>

<article>
    <h2>This is another title</h2>
    <p>This is another post content</p>
</article>
```

The final output will be the following:

```
<!-- app/views/index.phtml -->
<!DOCTYPE html>
<html>
    <head>
        <title>Blog's title</title>
    </head>
    <body>

        <!-- app/views/layouts/common.phtml -->

        <ul class="menu">
            <li><a href="/">Home</a></li>
            <li><a href="/articles">Articles</a></li>
            <li><a href="/contact">Contact us</a></li>
        </ul>

        <div class="content">

            <!-- app/views/layouts/posts.phtml -->

            <h1>Blog Title</h1>

            <!-- app/views/layouts/posts/last.phtml -->

            <article>
                <h2>This is a title</h2>
                <p>This is the post content</p>
            </article>

            <article>
                <h2>This is another title</h2>
                <p>This is another post content</p>
            </article>

        </div>

    </body>
</html>
```

2.14.3 Using Partials

Partial templates are another way of breaking the rendering process into simpler more manageable chunks that can be reused by different parts of the application. With a partial, you can move the code for rendering a particular piece of a response to its own file.

One way to use partials is to treat them as the equivalent of subroutines: as a way to move details out of a view so that your code can be more easily understood. For example, you might have a view that looks like this:

```
<?php $this->partial("shared/ad_banner") ?>

<h1>Robots</h1>

<p>Check out our specials for robots:</p>
...

<?php $this->partial("shared/footer") ?>
```

2.14.4 Transfer values from the controller to views

Phalcon\Mvc\View is available in each controller using the view variable ($this->view). You can use that object to set variables directly to the view from a controller action by using the setVar() method.

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function showAction()
    {
        //Pass all the posts to the views
        $this->view->setVar("posts", Posts:find());
    }

}
```

A variable with the name of the first parameter of setView() will be created in the view, ready to be used. The variable can be of any type, from a simple string, integer etc. variable to a more complex structure such as array, collection etc.

```
<div class="post">
<?php

  foreach ($posts as $post) {
    echo "<h1>", $post->title, "</h1>";
  }

?>
</div>
```

2.14.5 Control Rendering Levels

As seen above, *Phalcon\Mvc\View* supports a view hierarchy. You might need to control the level of rendering produced by the view component. The method PhalconMvc\View::setRenderLevel() offers this functionality.

This method can be invoked from the controller or from a superior view layer to interfere with the rendering process.

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function findAction()
    {

        // This is an Ajax response so it doesn't generate any kind of view
        $this->view->setRenderLevel(\Phalcon\Mvc\View::LEVEL_NO_RENDER);

        //...
    }

    public function showAction($postId)
    {
        // Shows only the view related to the action
        $this->view->setRenderLevel(\Phalcon\Mvc\View::LEVEL_ACTION_VIEW);
    }

}
```

The available render levels are:

Class Constant	Description	Order
LEVEL_NO_RENDER	Indicates to avoid generating any kind of presentation.	
LEVEL_ACTION_VIEW	Generates the presentation to the view associated to the action.	1
LEVEL_BEFORE_TEMPLATE	Generates presentation templates prior to the controller layout.	2
LEVEL_LAYOUT	Generates the presentation to the controller layout.	3
LEVEL_AFTER_TEMPLATE	Generates the presentation to the templates after the controller layout.	4
LEVEL_MAIN_LAYOUT	Generates the presentation to the main layout. File views/index.phtml	5

Disabling render levels

You can permanently or temporarily disable render levels. A level could be permanently disabled if it isn't used at all in the whole application:

```
<?php

use Phalcon\Mvc\View;

$di->set('view', function(){

    $view = new View();

    //Disable several levels
```

```
    $view->disableLevel(array(
        View::LEVEL_LAYOUT => true,
        View::LEVEL_MAIN_LAYOUT => true
    ));

    return $view;

}, true);
```

Or disable temporarily in some part of the application:

```
<?php

use Phalcon\Mvc\View;

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function findAction()
    {
        $this->view->disableLevel(View::LEVEL_MAIN_LAYOUT);
    }

}
```

2.14.6 Using models in the view layer

Application models are always available at the view layer. The *Phalcon\Loader* will instantiate them at runtime automatically:

```
<div class="categories">
<?php

foreach (Catergories::find("status = 1") as $category) {
   echo "<span class='category'>", $category->name, "</span>";
}

?>
</div>
```

Although you may perform model manipulation operations such as insert() or update() in the view layer, it is not recommended since it is not possible to forward the execution flow to another controller in the case of an error or an exception.

2.14.7 Picking Views

As mentioned above, when *Phalcon\Mvc\View* is managed by *Phalcon\Mvc\Application* the view rendered is the one related with the last controller and action executed. You could override this by using the Phalcon\Mvc\View::pick() method:

```
<?php

class ProductsController extends \Phalcon\Mvc\Controller
{

    public function listAction()
    {
        // Pick "views-dir/products/search" as view to render
        $this->view->pick("products/search");
    }

}
```

2.14.8 Caching View Fragments

Sometimes when you develop dynamic websites and some areas of them are not updated very often, the output is exactly the same between requests. *Phalcon\Mvc\View* offers caching a part or the whole rendered output to increase performance.

Phalcon\Mvc\View integrates with *Phalcon\Cache* to provide an easier way to cache output fragments. You could manually set the cache handler or set a global handler:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function showAction()
    {
        //Cache the view using the default settings
        $this->view->cache(true);
    }

    public function showArticleAction()
    {
        // Cache this view for 1 hour
        $this->view->cache(array(
            "lifetime" => 3600
        ));
    }

    public function resumeAction()
    {
        //Cache this view for 1 day with the key "resume-cache"
        $this->view->cache(
            array(
                "lifetime" => 86400,
                "key"      => "resume-cache",
            )
        );
    }

    public function downloadAction()
    {
        //Passing a custom service
        $this->view->cache(
            array(
```

```
                "service"  => "myCache",
                "lifetime" => 86400,
                "key"      => "resume-cache",
            )
        );
    }

}
```

When we do not define a key to the cache, the component automatically creates one doing a md5 to view name is currently rendered. It is a good practice to define a key for each action so you can easily identify the cache associated with each view.

When the View component needs to cache something it will request a cache service to the services container. The service name convention for this service is "viewCache":

```
<?php

//Set the views cache service
$di->set('viewCache', function() {

    //Cache data for one day by default
    $frontCache = new \Phalcon\Cache\Frontend\Output(array(
        "lifetime" => 86400
    ));

    //Memcached connection settings
    $cache = new \Phalcon\Cache\Backend\Memcache($frontCache, array(
        "host" => "localhost",
        "port" => "11211"
    ));

    return $cache;
});
```

> The frontend must always be Phalcon\Cache\Frontend\Output and the service 'viewCache' must be registered as always open (not shared)

When using view caching is also useful to prevent that controllers perform the processes that produce the data to be displayed in the views.

To achieve this we must identify uniquely each cache with a key. First we verify that the cache does not exist or has expired to make the calculations/queries to display data in the view:

```
<?php

class DownloadController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

        //Check whether the cache with key "downloads" exists or has expired
        if ($this->view->getCache()->exists('downloads')) {

            //Query the latest downloads
            $latest = Downloads::find(array(
                'order' => 'created_at DESC'
            ));
```

```
            $this->view->setVar('latest', $latest);
        }

        //Enable the cache with the same key "downloads"
        $this->view->cache(array(
            'key' => 'downloads'
        ));
    }

}
```

The PHP alternative site is a example of implementing the caching of fragments.

2.14.9 Disabling the view

If your controller doesn't produce any output in the view (or not even have one) you may disable the view component avoiding unnecessary processing:

```
<?php

class UsersController extends \Phalcon\Mvc\Controller
{

    public function closeSessionAction()
    {
        //Disable the view
        $this->view->disable();
    }

}
```

2.14.10 Template Engines

Template Engines helps designers to create views without use a complicated syntax. Phalcon includes a powerful and fast templating engine called *Volt*.

Additionally, *Phalcon\Mvc\View* allows you to use other template engines instead of plain PHP or Volt.

Using a different template engine, usually requires complex text parsing using external PHP libraries in order to generate the final output for the user. This usually increases the number of resources that your application are using.

If an external template engine is used, *Phalcon\Mvc\View* provides exactly the same view hierarchy and it's still possible to access the API inside these templates with a little more effort.

This component uses adapters, these help Phalcon to speak with those external template engines in a unified way, let's see how to do that integration.

Creating your own Template Engine Adapter

There are many template engines, which you might want to integrate or create one of your own. The first step to start using an external template engine is create an adapter for it.

A template engine adapter is a class that acts as bridge between *Phalcon\Mvc\View* and the template engine itself. Usually it only needs two methods implemented: __construct() and render(). The first one receives the *Phalcon\Mvc\View* instance that creates the engine adapter and the DI container used by the application.

The method render() accepts an absolute path to the view file and the view parameters set using $this->view->setVar(). You could read or require it when it's necessary.

```
<?php

class MyTemplateAdapter extends \Phalcon\Mvc\View\Engine
{

    /**
     * Adapter constructor
     *
     * @param \Phalcon\Mvc\View $view
     * @param \Phalcon\DI $di
     */
    public function __construct($view, $di)
    {
        //Initiliaze here the adapter
        parent::__construct($view, $di);
    }

    /**
     * Renders a view using the template engine
     *
     * @param string $path
     * @param array $params
     */
    public function render($path, $params)
    {

        // Access view
        $view = $this->_view;

        // Access options
        $options = $this->_options;

        //Render the view
        //...
    }

}
```

Changing the Template Engine

You can replace or add more a template engine from the controller as follows:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {
        // Set the engine
        $this->view->registerEngines(
            array(
                ".my-html" => "MyTemplateAdapter"
            )
        );
```

```
    }

    public function showAction()
    {
        // Using more than one template engine
        $this->view->registerEngines(
            array(
                ".my-html" => 'MyTemplateAdapter'
                ".phtml" => 'Phalcon\Mvc\View\Engine\Php'
            )
        );
    }

}
```

You can replace the template engine completely or use more than one template engine at the same time. The method Phalcon\Mvc\View::registerEngines() accepts an array containing data that define the template engines. The key of each engine is an extension that aids in distinguishing one from another. Template files related to the particular engine must have those extensions.

The order that the template engines are defined with Phalcon\Mvc\View::registerEngines() defines the relevance of execution. If *Phalcon\Mvc\View* finds two views with the same name but different extensions, it will only render the first one.

If you want to register a template engine or a set of them for each request in the application. You could register it when the view service is created:

```
<?php

//Setting up the view component
$di->set('view', function() {

    $view = new \Phalcon\Mvc\View();

    //A trailing directory separator is required
    $view->setViewsDir('../app/views/');

    $view->registerEngines(array(
        ".my-html" => 'MyTemplateAdapter'
    ));

    return $view;

}, true);
```

There are adapters available for several template engines on the Phalcon Incubator

2.14.11 Injecting services in View

Every view executed is included inside a *Phalcon\DINjectable* instance, providing easy access to the application's service container.

The following example shows how to write a jQuery ajax request using a url with the framework conventions. The service "url" (usually *Phalcon\Mvc\Url*) is injected in the view by accessing a property with the same name:

```
<script type="text/javascript">

$.ajax({
```

```
    url: "<?php echo $this->url->get("cities/get") ?>"
})
.done(function() {
    alert("Done!");
});

</script>
```

2.14.12 Stand-Alone Component

All the components in Phalcon can be used as *glue* components individually because they are loosely coupled to each other. Using *Phalcon\Mvc\View* in a stand-alone mode can be demonstrated below:

```
<?php

$view = new \Phalcon\Mvc\View();

//A trailing directory separator is required
$view->setViewsDir("../app/views/");

// Passing variables to the views, these will be created as local variables
$view->setVar("someProducts", $products);
$view->setVar("someFeatureEnabled", true);

//Start the output buffering
$view->start();

//Render all the view hierarchy related to the view products/list.phtml
$view->render("products", "list");

//Finish the output buffering
$view->finish();

echo $view->getContent();
```

2.14.13 View Events

Phalcon\Mvc\View is able to send events to an *EventsManager* if it's present. Events are triggered using the type "view". Some events when returning boolean false could stop the active operation. The following events are supported:

Event Name	Triggered	Can stop operation?
beforeRender	Triggered before starting the render process	Yes
beforeRenderView	Triggered before rendering an existing view	Yes
afterRenderView	Triggered after rendering an existing view	No
afterRender	Triggered after completing the render process	No
notFoundView	Triggered when a view was not found	No

The following example demonstrates how to attach listeners to this component:

```
<?php

$di->set('view', function() {

    //Create an event manager
    $eventsManager = new Phalcon\Events\Manager();
```

```
    //Attach a listener for type "view"
    $eventsManager->attach("view", function($event, $view) {
        echo $event->getType(), ' - ', $view->getActiveRenderPath(), PHP_EOL;
    });

    $view = new \Phalcon\Mvc\View();
    $view->setViewsDir("../app/views/");

    //Bind the eventsManager to the view component
    $view->setEventsManager($eventManagers);

    return $view;

}, true);
```

The following example shows how to create a plugin that clean/repair the HTML produced by the render process using Tidy:

```
<?php

class TidyPlugin
{

    public function afterRender($event, $view)
    {

        $tidyConfig = array(
            'clean' => true,
            'output-xhtml' => true,
            'show-body-only' => true,
            'wrap' => 0,
        );

        $tidy = tidy_parse_string($view->getContent(), $tidyConfig, 'UTF8');
        $tidy->cleanRepair();

        $view->setContent((string) $tidy);
    }

}

//Attach the plugin as a listener
$eventsManager->attach("view:afterRender", new TidyPlugin());
```

2.15 View Helpers

Writing and maintaining HTML markup can quickly become a tedious task because of the naming conventions and numerous attributes that have to be taken into consideration. Phalcon deals with this complexity by offering *Phalcon\Tag*, which in turn offers view helpers to generate HTML markup.

This component can be used in a plain HTML+PHP view or in a *Volt* template.

> This guide is not intended to be a complete documentation of available helpers and their arguments. Please visit the *Phalcon\Tag* page in the API for a complete reference.

2.15.1 Using Name Aliasing

You could use name aliasing to get short names for classes. In this case, a Tag name can be used to alias the *Phalcon\Tag* class.

```
<?php use \Phalcon\Tag as Tag; ?>
```

2.15.2 Document Type of Content

Phalcon provides Phalcon\Tag::setDoctype() helper to set document type of the content. Document type setting may affect HTML output produced by other tag helpers. For example, if you set XHTML document type family, helpers that return or output HTML tags will produce self-closing tags to follow valid XHTML standard.

Available document type constants in Phalcon\Tag namespace are:

Constant	Document type
HTML32	HTML 3.2
HTML401_STRICT	HTML 4.01 Strict
HTML401_TRANSITIONAL	HTML 4.01 Transitional
HTML401_FRAMESET	HTML 4.01 Frameset
HTML5	HTML 5
XHTML10_STRICT	XHTML 1.0 Strict
XHTML10_TRANSITIONAL	XHTML 1.0 Transitional
XHTML10_FRAMESET	XHTML 1.0 Frameset
XHTML11	XHTML 1.1
XHTML20	XHTML 2.0
XHTML5	XHTML 5

Setting document type.

```
<?php \Phalcon\Tag::setDoctype(\Phalcon\Tag::HTML401_STRICT); ?>
```

Getting document type.

```
<?= \Phalcon\Tag::getDoctype() ?>
<html>
<!-- your HTML code -->
</html>
```

The following HTML will be produced.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
<html>
<!-- your HTML code -->
</html>
```

Volt syntax:

```
{{ get_doctype() }}
<html>
<!-- your HTML code -->
</html>
```

2.15.3 Generating Links

A real common task in any web application or website is to produce links that allow us to navigate from one page to another. When they are internal URLs we can create them in the following manner:

```
<!-- for the default route -->
<?= Tag::linkTo("products/search", "Search") ?>

<!-- with CSS attributes -->
<?= Tag::linkTo(array('products/edit/10', 'Edit', 'class' => 'edit-btn')) ?>

<!-- for a named route -->
<?= Tag::linkTo(array(array('for' => 'show-product', 'title' => 123, 'name' => 'carrots'), 'Show'))
```

Actually, all produced URLs are generated by the component *Phalcon\Mvc\Url* (or service "url" failing)

Same links generated with Volt:

```
<!-- for the default route -->
{{ link_to("products/search", "Search") }}

<!-- for a named route -->
{{ link_to(['for': 'show-product', 'id': 123, 'name': 'carrots'], 'Show') }}
```

2.15.4 Creating Forms

Forms in web applications play an essential part in retrieving user input. The following example shows how to implement a simple search form using view helpers:

```
<?php use \Phalcon\Tag as Tag; ?>

<!-- Sending the form by method POST -->
<?= Tag::form("products/search") ?>
    <label for="q">Search:</label>
    <?= Tag::textField("q") ?>
    <?= Tag::submitButton("Search") ?>
</form>

<!-- Specyfing another method or attributes for the FORM tag -->
<?= Tag::form(array("products/search", "method" => "get")); ?>
    <label for="q">Search:</label>
    <?= Tag::textField("q"); ?>
    <?= Tag::submitButton("Search"); ?>
</form>
```

This last code will generate the following HTML:

```
<form action="/store/products/search/" method="get">
     <label for="q">Search:</label>
     <input type="text" id="q" value="" name="q" />
     <input type="submit" value="Search" />
</endform>
```

Same form generated in Volt:

```
<!-- Specyfing another method or attributes for the FORM tag -->
{{ form("products/search", "method": "get") }}
    <label for="q">Search:</label>
```

```
    {{ text_field("q") }}
    {{ submit_button("Search") }}
</form>
```

2.15.5 Helpers to Generate Form Elements

Phalcon provides a series of helpers to generate form elements such as text fields, buttons and more. The first parameter of each helper is always the name of the element to be generated. When the form is submitted, the name will be passed along with the form data. In a controller you can get these values using the same name by using the getPost() and getQuery() methods on the request object ($this->request).

```
<?php echo Phalcon\Tag::textField("username") ?>

<?php echo Phalcon\Tag::textArea(array(
    "comment",
    "This is the content of the text-area",
    "cols" => "6",
    "rows" => 20
)) ?>

<?php echo Phalcon\Tag::passwordField(array(
    "password",
    "size" => 30
)) ?>

<?php echo Phalcon\Tag::hiddenField(array(
    "parent_id",
    "value"=> "5"
)) ?>
```

Volt syntax:

```
{{ text_field("username") }}

{{ text_area("comment", "This is the content", "cols": "6", "rows": 20) }}

{{ password_field("password", "size": 30) }}

{{ hidden_field("parent_id", "value": "5") }}
```

2.15.6 Making Select Boxes

Generating select boxes (select box) is easy, especially if the related data is stored in PHP associative arrays. The helpers for select elements are Phalcon\Tag::select() and Phalcon\Tag::selectStatic(). Phalcon\Tag::select() has been was specifically designed to work with *Phalcon\Mvc\Model*, while Phalcon\Tag::selectStatic() can with PHP arrays.

```
<?php

// Using data from a resultset
echo Phalcon\Tag::select(
    array(
        "productId",
        Products::find("type = 'vegetables'"),
        "using" => array("id", "name")
    )
```

```
);

// Using data from an array
echo Phalcon\Tag::selectStatic(
    array(
        "status",
        array(
            "A" => "Active",
            "I" => "Inactive",
        )
    )
);
```

The following HTML will generated:

```
<select id="productId" name="productId">
    <option value="101">Tomato</option>
    <option value="102">Lettuce</option>
    <option value="103">Beans</option>
</select>

<select id="status" name="status">
    <option value="A">Active</option>
    <option value="I">Inactive</option>
</select>
```

You can add an "empty" option to the generated HTML:

```
<?php

// Creating a Select Tag with an empty option
echo Phalcon\Tag::select(
    array(
        "productId",
        Products::find("type = 'vegetables'"),
        "using" => array("id", "name"),
        "useEmpty" => true
    )
);

<select id="productId" name="productId">
    <option value="">Choose..</option>
    <option value="101">Tomato</option>
    <option value="102">Lettuce</option>
    <option value="103">Beans</option>
</select>

<?php

// Creating a Select Tag with an empty option with default text
echo Phalcon\Tag::select(
    array(
        'productId',
        Products::find("type = 'vegetables'"),
        'using' => array('id', "name')
        'useEmpty' => true,
        'emptyText' => 'Please, choose one...',
        'emptyValue' => '@'
    ),
```

```
);

<select id="productId" name="productId">
    <option value="@">Please, choose one..</option>
    <option value="101">Tomato</option>
    <option value="102">Lettuce</option>
    <option value="103">Beans</option>
</select>
```

Volt syntax for above example:

```
{# Creating a Select Tag with an empty option with default text #}
{{ select('productId', products, 'using': ['id', 'name'],
    'useEmpty': true, 'emptyText': 'Please, choose one...', 'emptyValue': '@') }}
```

2.15.7 Assigning HTML attributes

All the helpers accept an array as their first parameter which can contain additional HTML attributes for the element generated.

```
<?php \Phalcon\Tag::textField(
    array(
        "price",
        "size"        => 20,
        "maxlength"   => 30,
        "placeholder" => "Enter a price",
    )
) ?>
```

or using Volt:

```
{{ text_field("price", "size": 20, "maxlength": 30, "placeholder": "Enter a price") }}
```

The following HTML is generated:

```
<input type="text" name="price" id="price" size="20" maxlength="30"
    placeholder="Enter a price" />
```

2.15.8 Setting Helper Values

From Controllers

It is a good programming principle for MVC frameworks to set specific values for form elements in the view. You can set those values directly from the controller using Phalcon\Tag::setDefault(). This helper preloads a value for any helpers present in the view. If any helper in the view has a name that matches the preloaded value, it will use it, unless a value is directly assigned on the helper in the view.

```
<?php

class ProductsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {
        Phalcon\Tag::setDefault("color", "Blue");
```

```
    }

}
```

At the view, a selectStatic helper matches the same index used to preset the value. In this case "color":

```
<?php

echo \Phalcon\Tag::selectStatic(
    array(
        "color",
        array(
            "Yellow" => "Yellow",
            "Blue"   => "Blue",
            "Red"    => "Red"
        )
    )
);
```

This will generate the following select tag with the value "Blue" selected:

```
<select id="color" name="color">
    <option value="Yellow">Yellow</option>
    <option value="Blue" selected="selected">Blue</option>
    <option value="Red">Red</option>
</select>
```

From the Request

A special feature that the *Phalcon\Tag* helpers have is that they keep the values of form helpers between requests. This way you can easily show validation messages without losing entered data.

Specifying values directly

Every form helper supports the parameter "value". With it you can specify a value for the helper directly. When this parameter is present, any preset value using setDefault() or via request will be ignored.

2.15.9 Changing dynamically the Document Title

Phalcon\Tag offers helpers to change dynamically the document title from the controller. The following example demonstrates just that:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function initialize()
    {
        Phalcon\Tag::setTitle("Your Website");
    }

    public function indexAction()
    {
        Phalcon\Tag::prependTitle("Index of Posts - ");
```

```
    }

}
```

```
<html>
    <head>
        <?php echo \Phalcon\Tag::getTitle(); ?>
    </head>
    <body>

    </body>
</html>
```

The following HTML will generated:

```
<html>
    <head>
        <title>Index of Posts - Your Website</title>
    </head>
      <body>

      </body>
</html>
```

2.15.10 Static Content Helpers

Phalcon\Tag also provide helpers to generate tags such as script, link or img. They aid in quick and easy generation of the static resources of your application

Images

```
<?php

// Generate <img src="/your-app/img/hello.gif">
echo \Phalcon\Tag::image("img/hello.gif");

// Generate <img alt="alternative text" src="/your-app/img/hello.gif">
echo \Phalcon\Tag::image(
    array(
       "img/hello.gif",
       "alt" => "alternative text"
    )
);
```

Volt syntax:

```
{# Generate <img src="/your-app/img/hello.gif"> #}
{{ image("img/hello.gif") }}

{# Generate <img alt="alternative text" src="/your-app/img/hello.gif"> #}
{{ image("img/hello.gif", "alt": "alternative text") }}
```

Stylesheets

```
<?php

// Generate <link rel="stylesheet" href="http://fonts.googleapis.com/css?family=Rosario" type="text/
echo \Phalcon\Tag::stylesheetLink("http://fonts.googleapis.com/css?family=Rosario", false);

// Generate <link rel="stylesheet" href="/your-app/css/styles.css" type="text/css">
echo \Phalcon\Tag::stylesheetLink("css/styles.css");
```

Volt syntax:

```
{# Generate <link rel="stylesheet" href="http://fonts.googleapis.com/css?family=Rosario" type="text/
{{ stylesheet_link("http://fonts.googleapis.com/css?family=Rosario", false) }}

{# Generate <link rel="stylesheet" href="/your-app/css/styles.css" type="text/css"> #}
{{ stylesheet_link("css/styles.css") }}
```

Javascript

```
<?php

// Generate <script src="http://localhost/javascript/jquery.min.js" type="text/javascript"></script>
echo \Phalcon\Tag::javascriptInclude("http://localhost/javascript/jquery.min.js", false);

// Generate <script src="/your-app/javascript/jquery.min.js" type="text/javascript"></script>
echo \Phalcon\Tag::javascriptInclude("javascript/jquery.min.js");
```

Volt syntax:

```
{# Generate <script src="http://localhost/javascript/jquery.min.js" type="text/javascript"></script>
{{ javascript_include("http://localhost/javascript/jquery.min.js", false) }}

{# Generate <script src="/your-app/javascript/jquery.min.js" type="text/javascript"></script> #}
{{ javascript_include("javascript/jquery.min.js") }}
```

2.15.11 Creating your own helpers

You can easily create your own helpers by extending the *Phalcon\Tag* and implementing your own helper. Below is a simple example of a custom helper:

```
<?php

class MyTags extends \Phalcon\Tag
{

    /**
    * Generates a widget to show a HTML5 audio tag
    *
    * @param array
    * @return string
    */
    static public function audioField($parameters)
    {

        // Converting parameters to array if it is not
```

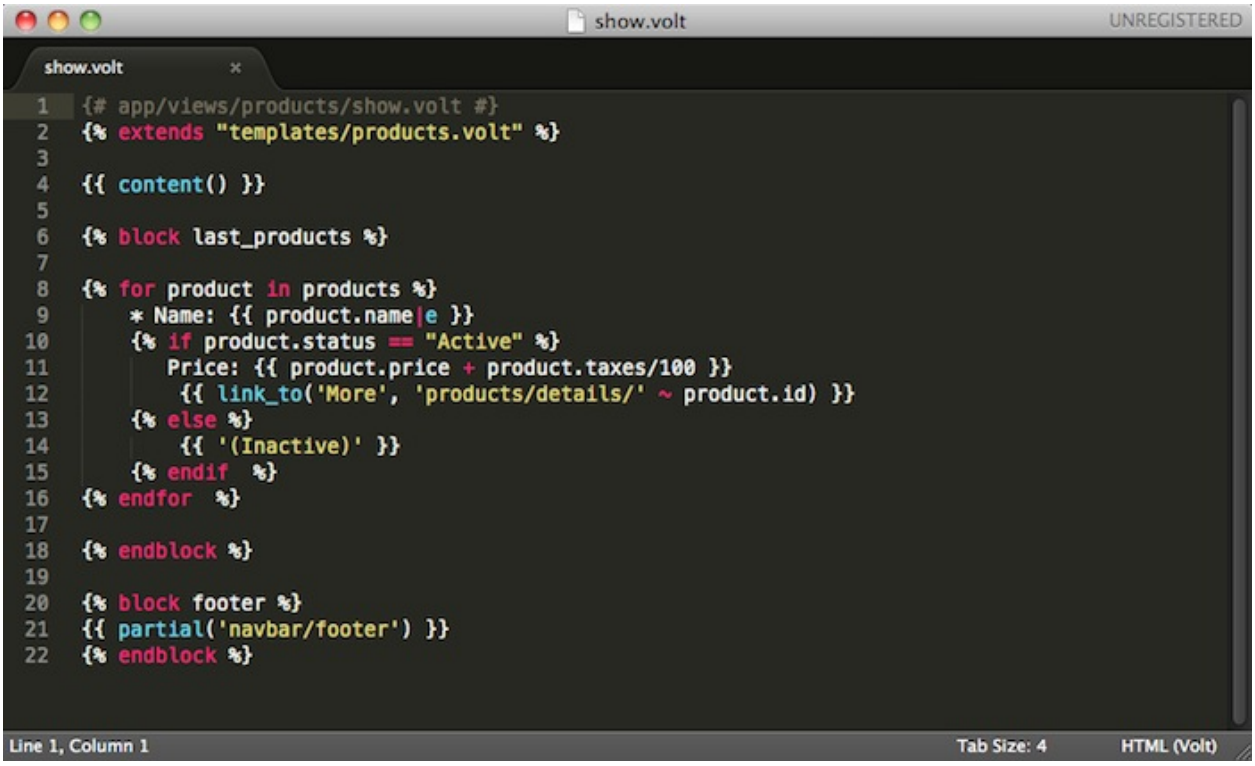
```
        if (!is_array($parameters)) {
            $parameters = array($parameters);
        }

        // Determining attributes "id" and "name"
        if (!isset($parameters[0])) {
            $parameters[0] = $parameters["id"];
        }

        $id = $parameters[0];
        if (!isset($parameters["name"])) {
            $parameters["name"] = $id;
        } else {
            if (!$parameters["name"]) {
                $parameters["name"] = $id;
            }
        }

        // Determining widget value,
        // \Phalcon\Tag::setDefault() allows to set the widget value
        if (isset($parameters["value"])) {
            $value = $parameters["value"];
            unset($parameters["value"]);
        } else {
            $value = self::getValue($id);
        }

        // Generate the tag code
        $code = '<audio id="'.$id.'" value="'.$value.'" ';
        foreach ($parameters as $key => $attributeValue) {
            if (!is_integer($key)) {
                $code.= $key.'="'.$attributeValue.'" ';
            }
        }
        $code.=" />";

        return $code;
    }

}
```

In next chapter, we'll talk about *Volt* a faster template engine for PHP, where you can use a more friendly syntax for using helpers provided by Phalcon\Tag.

2.16 Volt: Template Engine

Volt is an ultra-fast and designer friendly templating language written in C for PHP. It provides you a set of helpers to write views in an easy way. Volt is highly integrated with other components of Phalcon, just as you can use it as a stand-alone component in your applications.

Volt is inspired on Jinja, originally created by Armin Ronacher. Therefore many developers will be in familiar ground using the same syntax they have been using with similar template engines. Volt's syntax and features have been enhanced with more elements and of course with the performance that developers have been accustomed to while working with Phalcon.



2.16.1 Introduction

Volt views are compiled to pure PHP code, so basically they save the effort of writing PHP code manually:

```
{# app/views/products/show.volt #}

{% block last_products %}

{% for product in products %}
    * Name: {{ product.name|e }}
    {% if product.status == "Active" %}
       Price: {{ product.price + product.taxes/100 }}
    {% endif  %}
{% endfor  %}

{% endblock %}
```

2.16.2 Activating Volt

As other template engines, you may register Volt in the view component, using a new extension or reusing the standard .phtml:

```
<?php

//Registering Volt as template engine
$di->set('view', function() {

    $view = new \Phalcon\Mvc\View();

    $view->setViewsDir('../app/views/');
```

```
    $view->registerEngines(array(
        ".volt" => 'Phalcon\Mvc\View\Engine\Volt'
    ));

    return $view;
});
```

Use the standard ".phtml" extension:

```
<?php

$view->registerEngines(array(
    ".phtml" => 'Phalcon\Mvc\View\Engine\Volt'
));
```

2.16.3 Basic Usage

A view consists on Volt code, PHP and HTML. A set of special delimiters is available to enter in Volt mode. {% ... %} is used to execute statements such as for-loops or assign values and {{ ... }}, prints the result of an expression to the template.

Below is a minimal template that illustrates a few basics:

```
{# app/views/posts/show.phtml #}
<!DOCTYPE html>
<html>
    <head>
        <title>{{ title }} - A example blog</title>
    </head>
    <body>

        {% if show_navigation %}
            <ul id="navigation">
            {% for item in menu %}
                <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
            {% endfor %}
            </ul>
        {% endif %}

        <h1>{{ post.title }}</h1>

        <div class="content">
            {{ post.content }}
        </div>

    </body>
</html>
```

Using Phalcon\Mvc\View::setVar you can pass variables from the controller to the views. In the previous example, three variables were passed to the view: title, menu and post:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function showAction()
```

```
    {

        $post = Post::findFirst();

        $this->view->setVar("title", $post->title);
        $this->view->setVar("post", $post);
        $this->view->setVar("menu", Menu::find());
        $this->view->setVar("show_navigation", true);

    }

}
```

2.16.4 Variables

Variables may have attributes, those can be accessed using the syntax: foo.bar. If you are passing arrays, you can access using the curly braces syntax: foo['bar']

```
{{ post.title }}
{{ post['title'] }}
```

2.16.5 Filters

Variables can be formatted or modified using filters. The pipe operator | is used to apply filters to variables:

```
{{ post.title|e }}
{{ post.content|striptags }}
{{ name|capitalize|trim }}
```

The following is the list of available built-in filters in Volt:

Filter	Description
e	Applies Phalcon\Escaper->escapeHtml to the value
escape	Applies Phalcon\Escaper->escapeHtml to the value
escape_css	Applies Phalcon\Escaper->escapeCss to the value
escape_js	Applies Phalcon\Escaper->escapeJs to the value
escape_attr	Applies Phalcon\Escaper->escapeHtmlAttr to the value
trim	Applies the trim PHP function to the value. Removing extra spaces
striptags	Applies the striptags PHP function to the value. Removing HTML tags
slashes	Applies the slashes PHP function to the value. Escaping values
stripslashes	Applies the stripslashes PHP function to the value. Removing escaped quotes
capitalize	Capitalizes a string by applying the ucwords PHP function to the value
lower	Change the case of a string to lowercase
upper	Change the case of a string to uppercase
length	Counts the string length or how many items are in an array or object
nl2br	Changes newlines \n by line breaks ( ). Uses the PHP function nl2br
sort	Sorts an array using the PHP function asort
keys	Returns the array keys using array_keys
join	Joins the array parts using a separator join
format	Formats a string using sprintf.
json_encode	Converts a value into its JSON representation
json_decode	Converts a value from its JSON representation to a PHP representation
abs	Applies the abs PHP function to a value.
url_encode	Applies the urlencode PHP function to the value
default	Sets a default value in case that the evaluated expression is null
convert_encoding	Converts a string from one charset to another

Examples:

```
{# e or escape filter #}
{{ "<h1>Hello<h1>"|e }}
{{ "<h1>Hello<h1>"|escape }}

{# trim filter #}
{{ "   hello   "|trim }}

{# striptags filter #}
{{ "<h1>Hello<h1>"|striptags }}

{# slashes filter #}
{{ "'this is a string'"|slashes }}

{# stripslashes filter #}
{{ "\'this is a string\'"|stripslashes }}

{# capitalize filter #}
{{ "hello"|capitalize }}

{# lower filter #}
{{ "HELLO"|lower }}

{# upper filter #}
{{ "hello"|upper }}

{# length filter #}
{{ "robots"|length }}
{{ [1, 2, 3]|length }}
```

```
{# nl2br filter #}
{{ "some\ntext"|nl2br }}

{# sort filter #}
{{ [3, 1, 2]|sort }}

{# keys filter #}
{{ ['first': 1, 'second': 2, 'third': 3]|keys }}

{# json_encode filter #}
{{ robots|json_encode }}

{# json_decode filter #}
{{ '{"one":1,"two":2,"three":3}'|json_decode }}

{# url_encode filter #}
{{ post.permanent_link|url_encode }}

{# convert_encoding filter #}
{{ "désolé"|convert_encoding('utf8', 'latin1') }}
```

2.16.6 Comments

Comments may also be added to a template using the {# ... #} delimiters. All text inside them is just ignored in the final output:

```
{# note: this is a comment
    {% set price = 100; %}
#}
```

2.16.7 List of Control Structures

Volt provides a set of basic but powerful control structures for use in templates:

For

Loop over each item in a sequence. The following example shows how to traverse a set of "robots" and print his/her name:

```
<h1>Robots</h1>
<ul>
{% for robot in robots %}
  <li>{{ robot.name|e }}</li>
{% endfor %}
</ul>
```

for-loops can also be nested:

```
<h1>Robots</h1>
{% for robot in robots %}
  {% for part in robot.parts %}
  Robot: {{ robot.name|e }} Part: {{ part.name|e }} <br/>
  {% endfor %}
{% endfor %}
```

You can get the element "keys" as in the PHP counterpart using the following syntax:

```
{% set numbers = ['one': 1, 'two': 2, 'three': 3] %}

{% for name, value in numbers %}
  Name: {{ name }} Value: {{ value }}
{% endfor %}
```

An "if" evaluation can be optionally set:

```
{% set numbers = ['one': 1, 'two': 2, 'three': 3] %}

{% for value in numbers if value < 2 %}
  Name: {{ name }} Value: {{ value }}
{% endfor %}

{% for name, value in numbers if name != 'two' %}
  Name: {{ name }} Value: {{ value }}
{% endfor %}
```

If an 'else' is defined inside the 'for', it will be executed if the expression in the iterator result in zero iterations:

```
<h1>Robots</h1>
{% for robot in robots %}
    Robot: {{ robot.name|e }} Part: {{ part.name|e }} <br/>
{% else %}
    There are no robots to show
{% endfor %}
```

Loop Controls

The 'break' and 'continue' statements can be used to exit from a loop or force an iteration in the current block:

```
{# skip the even robots #}
{% for index, robot in robots %}
    {% if index is even %}
        {% continue %}
    {% endif %}
    ...
{% endfor %}

{# exit the foreach on the first even robot #}
{% for index, robot in robots %}
    {% if index is even %}
        {% break %}
    {% endif %}
    ...
{% endfor %}
```

If

As PHP, an "if" statement checks if an expression is evaluated as true or false:

```
<h1>Cyborg Robots</h1>
<ul>
{% for robot in robots %}
  {% if robot.type == "cyborg" %}
```

```
  <li>{{ robot.name|e }}</li>
  {% endif %}
{% endfor %}
</ul>
```

The else clause is also supported:

```
<h1>Robots</h1>
<ul>
{% for robot in robots %}
  {% if robot.type == "cyborg" %}
  <li>{{ robot.name|e }}</li>
  {% else %}
  <li>{{ robot.name|e }} (not a cyborg)</li>
  {% endif %}
{% endfor %}
</ul>
```

The 'elseif' control flow structure can be used together with if to emulate a 'switch' block:

```
{% if robot.type == "cyborg" %}
    Robot is a cyborg
{% elseif robot.type == "virtual" %}
    Robot is virtual
{% elseif robot.type == "mechanical" %}
    Robot is mechanical
{% endif %}
```

Loop Context

A special variable is available inside 'for' loops providing you information about

Variable	Description
loop.index	The current iteration of the loop. (1 indexed)
loop.index0	The current iteration of the loop. (0 indexed)
loop.revindex	The number of iterations from the end of the loop (1 indexed)
loop.revindex0	The number of iterations from the end of the loop (0 indexed)
loop.first	True if is the first iteration.
loop.last	True if is the last iteration.
loop.length	The number of items to itarate

```
{% for robot in robots %}
    {% if loop.first %}
            <table>
            <tr>
            <th>#</th>
                    <th>Id</th>
                    <th>Name</th>
            </tr>
    {% endif %}
            <tr>
            <td>{{ loop.index }}</td>
                    <td>{{ robot.id }}</td>
                    <td>{{ robot.name }}</td>
            </tr>
    {% if loop.last %}
            </table>
```

```
    {% endif %}
{% endfor %}
```

2.16.8 Assignments

Variables may be changed in a template using the instruction "set":

```
{% set fruits = ['Apple', 'Banana', 'Orange'] %}
{% set name = robot.name %}
```

2.16.9 Expressions

Volt provides a basic set of expression support, including literals and common operators.

A expression can be evaluated and printed using the '{{' and '}}' delimiters:

```
{{ (1 + 1) * 2 }}
```

If an expression needs to be evaluated without be printed the 'do' statement can be used:

```
{% do (1 + 1) * 2 %}
```

Literals

The following literals are supported:

Filter	Description
"this is a string"	Text between double quotes or single quotes are handled as strings
100.25	Numbers with a decimal part are handled as doubles/floats
100	Numbers without a decimal part are handled as integers
false	Constant "false" is the boolean false value
true	Constant "true" is the boolean true value
null	Constant "null" is the Null value

Arrays

Whether you're using PHP 5.3 or 5.4, you can create arrays by enclosing a list of values in square brackets:

```
{# Simple array #}
{{ ['Apple', 'Banana', 'Orange'] }}

{# Other simple array #}
{{ ['Apple', 1, 2.5, false, null] }}

{# Multi-Dimensional array #}
{{ [[1, 2], [3, 4], [5, 6]] }}

{# Hash-style array #}
{{ ['first': 1, 'second': 4/2, 'third': '3'] }}
```

Math

You may make calculations in templates using the following operators:

Operator	Description
+	Perform an adding operation. {{ 2 + 3 }} returns 5
-	Perform a substraction operation {{ 2 - 3 }} returns -1
*	Perform a multiplication operation {{ 2 * 3 }} returns 6
/	Perform a division operation {{ 10 / 2 }} returns 5
%	Calculate the remainder of an integer division {{ 10 % 3 }} returns 1

Comparisions

The following comparision operators are available:

Operator	Description
==	Check whether both operands are equal
!=	Check whether both operands aren't equal
<>	Check whether both operands aren't equal
>	Check whether left operand is greater than right operand
<	Check whether left operand is less than right operand
<=	Check whether left operand is less or equal than right operand
>=	Check whether left operand is greater or equal than right operand
===	Check whether both operands are identical
!==	Check whether both operands aren't identical

Logic

Logic operators are useful in the "if" expression evaluation to combine multiple tests:

Operator	Description
or	Return true if the left or right operand is evaluated as true
and	Return true if both left and right operands are evaluated as true
not	Negates an expression
(expr)	Parenthesis groups expressions

Other Operators

Additional operators seen the following operators are available:

Operator	Description
~	Concatenates both operands {{ "hello " ~ "world" }}
|	Applies a filter in the right operand to the left {{ "hello"|uppercase }}
..	Creates a range {{ 'a'..'z' }} {{ 1..10 }}
is	Same as == (equals), also performs tests
in	To check if a expression is contained into other expressions if "a" in "abc"
is not	Same as != (not equals)
is not	Same as != (not equals)
'a' ? 'b' : 'c'	Ternary operator. The same as the PHP ternary operator

The following example shows how to use operators:

```
{% set robots = ['Voltron', 'Astro Boy', 'Terminator', 'C3PO'] %}

{% for index in 0..robots|length %}
    {% if robots[index] is defined %}
        {{ "Name: " ~ robots[index] }}
    {% endif %}
{% endfor %}
```

2.16.10 Tests

Tests can be used to test if a variable has a valid expected value. The operator "is" is used to perform the tests:

```
{% set robots = ['1': 'Voltron', '2': 'Astro Boy', '3': 'Terminator', '4': 'C3PO'] %}

{% for position, name in robots %}
    {% if position is odd %}
        {{ value }}
    {% endif %}
{% endfor %}
```

The following built-in tests are available in Volt:

Test	Description
empty	Checks if a variable is empty
even	Checks if a numeric value is even
odd	Checks if a numeric value is odd
numeric	Checks if value is numeric
scalar	Checks if value is scalar (not an array or object)
iterable	Checks if a value is iterable. Can be traversed by a "for" statement
divisibleby	Checks if a value is divisible by other value
sameas	Checks if a value is identical to other value

More examples:

```
{% if robot is empty %}
    The robot is null or isn't defined
{% endif }

{% for key, name in [1: 'Voltron', 2: 'Astroy Boy', 3: 'Bender'] %}
    {% if key is even %}
        {{ name }}
    {% endif }
{% endfor %}

{% for key, name in [1: 'Voltron', 2: 'Astroy Boy', 3: 'Bender'] %}
    {% if key is odd %}
        {{ name }}
    {% endif }
{% endfor %}

{% for key, name in [1: 'Voltron', 2: 'Astroy Boy', 'third': 'Bender'] %}
    {% if key is numeric %}
        {{ name }}
    {% endif }
{% endfor %}

{% set robots = [1: 'Voltron', 2: 'Astroy Boy'] %}
```

```
{% if robots is iterable %}
    {% for robot in robots %}
        ...
    {% endfor %}
{% endif %}
```

2.16.11 Using Tag Helpers

Volt is highly integrated with *Phalcon\Tag*, so it's easy to use the helpers provided by that component in a Volt template:

```
{{ javascript_include("js/jquery.js") }}

{{ form('products/save', 'method': 'post') }}

    <label>Name</label>
    {{ text_field("name", "size": 32) }}

    <label>Type</label>
    {{ select("type", productTypes, 'using': ['id', 'name']) }}

    {{ submit_button('Send') }}

</form>
```

The following PHP is generated:

```
<?php echo Phalcon\Tag::javascriptInclude("js/jquery.js") ?>

<?php echo Phalcon\Tag::form(array('products/save', 'method' => 'post')); ?>

    <label>Name</label>
    <?php echo Phalcon\Tag::textField(array('name', 'size' => 32)); ?>

    <label>Type</label>
    <?php echo Phalcon\Tag::select(array('type', $productTypes, 'using' => array('id', 'name'))); ?>

    <?php echo Phalcon\Tag::submitButton('Send'); ?>

</form>
```

To call a PhalconTag helper, you only need to call an uncamelized version of the method:

Method	Volt function
Phalcon\Tag::linkTo	link_to
Phalcon\Tag::textField	text_field
Phalcon\Tag::passwordField	password_field
Phalcon\Tag::hiddenField	hidden_field
Phalcon\Tag::fileField	file_field
Phalcon\Tag::checkField	check_field
Phalcon\Tag::radioField	radio_field
Phalcon\Tag::submitButton	submit_button
Phalcon\Tag::selectStatic	select_static
Phalcon\Tag::select	select
Phalcon\Tag::textArea	text_area
Phalcon\Tag::form	form
Phalcon\Tag::endForm	end_form
Phalcon\Tag::getTitle	get_title
Phalcon\Tag::stylesheetLink	stylesheet_link
Phalcon\Tag::javascriptInclude	javascript_include
Phalcon\Tag::image	image
Phalcon\Tag::friendlyTitle	friendly_title

2.16.12 Functions

The following built-in functions are available in Volt:

Name	Description
content	Includes the content produced in a previous rendering stage
get_content	Same as 'content'
partial	Dynamically loads a partial view in the current template
super	Render the contents of the parent block
time	Calls the PHP function with the same name
date	Calls the PHP function with the same name
dump	Calls the PHP function 'var_dump'
version	Returns the current version of the framework
constant	Reads a PHP constant
url	Generate a URL using the 'url' service

2.16.13 View Integration

Also, Volt is integrated with *Phalcon\Mvc\View*, you can play with the view hierarchy and include partials as well:

```
{{ content() }}

<div id="footer">{{ partial("partials/footer") }}</div>
```

A partial is included in runtime, Volt also provides "include", this compiles the content of a view and returns its contents as part of the view which was included:

```
<div id="footer">{% include "partials/footer" %}</div>
```

Partial vs Include

Keep the following points in mind when choosing to use the "partial" function or "include":

- 'Partial' allows you to include templates made in Volt and in other template engines as well
- 'Partial' allows you to pass an expression like a variable allowing to include the content of other view dynamically
- 'Partial' is better if the content that you have to include changes frequently
- 'Include' copies the compiled content into the view which improves the performance
- 'Include' only allows to include templates made with Volt
- 'Include' requires an existing template at compile time

2.16.14 Template Inheritance

With template inheritance you can create base templates that can be extended by others templates allowing to reuse code. A base template define *blocks* than can be overridden by a child template. Let's pretend that we have the following base template:

```
{# templates/base.volt #}
<!DOCTYPE html>
<html>
    <head>
        {% block head %}
            <link rel="stylesheet" href="style.css" />
        {% endblock %}
        <title>{% block title %}{% endblock %} - My Webpage</title>
    </head>
    <body>
        <div id="content">{% block content %}{% endblock %}</div>
        <div id="footer">
            {% block footer %}&copy; Copyright 2012, All rights reserved.{% endblock %}
        </div>
    </body>
</html>
```

From other template we could extend the base template replacing the blocks:

```
{% extends "templates/base.volt" %}

{% block title %}Index{% endblock %}

{% block head %}<style type="text/css">.important { color: #336699; }</style>{% endblock %}

{% block content %}
    <h1>Index</h1>
    <p class="important">Welcome on my awesome homepage.</p>
{% endblock %}
```

Not all blocks must be replaced at a child template, only those that are needed. The final output produced will be the following:

```
<!DOCTYPE html>
<html>
    <head>
        <style type="text/css">.important { color: #336699; }</style>
        <title>Index - My Webpage</title>
    </head>
    <body>
        <div id="content">
```

```
            <h1>Index</h1>
            <p class="important">Welcome on my awesome homepage.</p>
        </div>
        <div id="footer">
            &copy; Copyright 2012, All rights reserved.
        </div>
    </body>
</html>
```

Multiple Inheritance

Extended templates can extend other templates. The following example illustrates this:

```
{# main.volt #}
<!DOCTYPE html>
<html>
    <head>
        <title>Title</title>
    </head>
    <body>
        {% block content %}{% endblock %}
    </body>
</html>
```

Template "layout.volt" extends "main.volt"

```
{# layout.volt #}
{% extends "main.volt" %}

{% block content %}

    <h1>Table of contents</h1>

{% endblock %}
```

Finally a view that extends "layout.volt":

```
{# index.volt #}
{% extends "layout.volt" %}

{% block content %}

    {{ super() }}

    <ul>
        <li>Some option</li>
        <li>Some other option</li>
    </ul>

{% endblock %}
```

Rendering "index.volt" produces:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Title</title>
    </head>
```

```
    <body>

        <h1>Table of contents</h1>

        <ul>
            <li>Some option</li>
            <li>Some other option</li>
        </ul>

    </body>
</html>
```

Note the call to the function "super()". With that function it's possible to render the contents of the parent block.

As partials, the path set to "extends" is a relative path under the current views directory (i.e. app/views/).

> By default, and for performance reasons, Volt only checks for changes in the children templates to know when to re-compile to plain PHP again, so it is recommended initialize Volt with the option 'compileAlways' => true. Thus, the templates are compiled always taking into account changes in the parent templates.

2.16.15 Autoescape mode

You can enable auto-escaping of all variables printed in a block using the autoescape mode:

```
Manually escaped: {{ robot.name|e }}

{% autoescape true %}
    Autoescaped: {{ robot.name }}
    {% autoescape false %}
        No Autoescaped: {{ robot.name }}
    {% endautoescape %}
{% endautoescape %}
```

2.16.16 Setting up the Volt Engine

Volt can be configured to alter its default behavior, the following example explain how to do that:

```
<?php

//Register Volt as a service
$di->set('voltService', function($view, $di) {

    $volt = new \Phalcon\Mvc\View\Engine\Volt($view, $di);

    $volt->setOptions(array(
        "compiledPath" => "../app/compiled-templates/",
        "compiledExtension" => ".compiled"
    ));

    return $volt;
});

//Register Volt as template engine
$di->set('view', function() {
```

```
    $view = new \Phalcon\Mvc\View();

    $view->setViewsDir('../app/views/');

    $view->registerEngines(array(
        ".volt" => 'voltService'
    ));

    return $view;
});
```

If you do not want to reuse Volt as a service you can pass an anonymous function to register the engine instead of a service name:

```
<?php

//Register Volt as template engine with an anonymous function
$di->set('view', function() {

    $view = new \Phalcon\Mvc\View();

    $view->setViewsDir('../app/views/');

    $view->registerEngines(array(
        ".volt" => function($view, $di) {
            $volt = new \Phalcon\Mvc\View\Engine\Volt($view, $di);

            //set some options here

            return $volt;
        }
    ));

    return $view;
});
```

The following options are available in Volt:

Option	Description	Default
compiledPath	A writable path where the compiled PHP templates will be placed	./
compiledExtension	An additional extension appended to the compiled PHP file	.php
compiledSeparator	Volt replaces the directory separators / and \ by this separator in order to create a single file in the compiled directory	%%
stat	Whether Phalcon must check if exists differences between the template file and its compiled path	true
compileAlways	Tell Volt if the templates must be compiled in each request or only when they change	false
prefix	Allows to prepend a prefix to the templates in the compilation path	null

2.16.17 Extending Volt

Unlike other template engines, Volt itself is not required to run the compiled templates. Once the templates are compiled there is no dependence on Volt. With performance independence in mind, Volt only acts as a compiler for PHP templates.

The Volt compiler allow you to extend it adding more functions, tests or filters to the existing ones.

Functions

Functions act as normal PHP functions, a valid string name is required as function name. Functions can be added using two strategies, returning a simple string or using an anonymous function. Always is required that the chosen strategy returns a valid PHP string expression:

```
<?php

$volt = new \Phalcon\Mvc\View\Engine\Volt($view, $di);

$compiler = $volt->getCompiler();

//This binds the function 'shuffle' in Volt to the PHP function 'str_shuffle'
$compiler->addFunction('shuffle', 'str_shuffle');
```

Register the function with an anonymous function. This case we use $resolvedArgs to pass the arguments exactly as were passed in the arguments:

```
<?php

$compiler->addFunction('widget', function($resolvedArgs, $exprArgs) {
    return 'MyLibrary\Widgets::get('.$resolvedArgs.')';
});
```

Treat the arguments independently and unresolved:

```
<?php

$compiler->addFunction('repeat', function($resolvedArgs, $exprArgs) use ($compiler) {

    //Resolve the first argument
    $firstArgument = $compiler->expression($exprArgs[0]['expr']);

    //Checks if the second argument was passed
    if (isset($exprArgs[1])) {
        $secondArgument = $compiler->expression($exprArgs[1]['expr']);
    } else {
        //Use '10' as default
        $secondArgument = '10';
    }

    return 'str_repeat('.$firstArgument.', '.$secondArgument.')';
});
```

Generate the code based on some function availability:

```
<?php

$compiler->addFunction('include_text', function($resolvedArgs, $exprArgs) {
    if (function_exists('mb_stripos')) {
        return 'mb_stripos('.$resolvedArgs.')';
    } else {
        return 'stripos('.$resolvedArgs.')';
    }
});
```

Built-in functions can be overrided adding a function with its name:

```
<?php

//Replace built-in function dump
$compiler->addFunction('dump', 'print_r');
```

Filters

A filter has the following form in a template: leftExpr|name(optional-args). Adding new filters is similar as seen with the functions:

```
<?php

//This creates a filter 'hash' that uses the PHP function 'md5'
$compiler->addFilter('hash', 'md5');
```

```
<?php

$compiler->addFilter('int', function($resolvedArgs, $exprArgs) {
    return 'intval('.$resolvedArgs.')';
});
```

Built-in filters can be overrided adding a function with its name:

```
<?php

//Replace built-in filter 'capitalize'
$compiler->addFilter('capitalize', 'lcfirst');
```

2.16.18 Caching view fragments

With Volt it's easy cache view fragments. This caching improves performance preventing that the contents of a block is executed by PHP each time the view is displayed:

```
{% cache "sidebar" %}
    <!-- generate this content is slow so we are going to cache it -->
{% endcache %}
```

Setting an specific number of seconds:

```
{# cache the sidebar by 1 hour #}
{% cache "sidebar" 3600 %}
    <!-- generate this content is slow so we are going to cache it -->
{% endcache %}
```

Any valid expression can be used as cache key:

```
{% cache ("article-" ~ post.id) 3600 %}

    <h1>{{ post.title }}</h1>

    <p>{{ post.content }}</p>

{% endcache %}
```

The caching is done by the *Phalcon\Cache* component via the view component. Learn more about how this integration works in the section *"Caching View Fragments"*.

2.16.19 Inject Services into a Template

If a service container (DI) is available for Volt, you can use the services by only accessing the name of the service in the template:

```
{# Inject the 'flash' service #}
<div id="messages">{{ flash.output() }}</div>

{# Inject the 'security' service #}
<input type="hidden" name="token" value="{{ security.getToken() }}">
```

2.16.20 Stand-alone component

Using Volt in a stand-alone mode can be demonstrated below:

```
<?php

//Create a compiler
$compiler = new \Phalcon\Mvc\View\Engine\Volt\Compiler();

//Optionally add some options
$compiler->setOptions(array(
    //...
));

//Compile a template string returning PHP code
echo $compiler->compileString('{{ "hello" }}');

//Compile a template in a file specifying the destination file
$compiler->compileFile('layouts/main.volt', 'cache/layouts/main.volt.php');

//Compile a template in a file based on the options passed to the compiler
$compiler->compile('layouts/main.volt');

//Require the compiled templated (optional)
require $compiler->getCompiledPath();
```

2.16.21 External Resources

- A bundle for Sublime/Textmate is available here
- Our website is running using Volt as template engine, check out its code on github
- Album-O-Rama is a sample application using Volt as template engine, check out its code on Github

2.17 MVC Applications

All the hard work behind orchestrating the operation of MVC in Phalcon is normally done by *Phalcon\Mvc\Application*. This component encapsulates all the complex operations required in the background, instantiating every component needed and integrating it with the project, to allow the MVC pattern to operate as desired.

2.17.1 Single or Multi Module Applications

With this component you can run various types of MVC structures:

Single Module

Single MVC applications consist of one module only. Namespaces can be used but are not necessary. An application like this would have the following file structure:

```
single/
    app/
        controllers/
        models/
        views/
    public/
        css/
        img/
        js/
```

If namespaces are not used, the following bootstrap file could be used to orchestrate the MVC flow:

```
<?php

$loader = new \Phalcon\Loader();

$loader->registerDirs(
    array(
        '../apps/controllers/',
        '../apps/models/'
    )
)->register();

$di = new \Phalcon\DI\FactoryDefault();

// Registering the view component
$di->set('view', function() {
    $view = new \Phalcon\Mvc\View();
    $view->setViewsDir('../apps/views/');
    return $view;
});

try {
    $application = new \Phalcon\Mvc\Application();
    $application->setDI($di);
    echo $application->handle()->getContent();
} catch(Phalcon\Exception $e) {
    echo $e->getMessage();
}
```

If namespaces are used, the following bootstrap can be used:

```
<?php

$loader = new \Phalcon\Loader();

// Use autoloading with namespaces prefixes
$loader->registerNamespaces(
    array(
```

```
        'Single\Controllers' => '../apps/controllers/',
        'Single\Models'      => '../apps/models/',
    )
)->register();

$di = new \Phalcon\DI\FactoryDefault();

// Register the dispatcher setting a Namespace for controllers
// Pay special attention to the double slashes at the end of the
// parameter used in the setDefaultNamespace function
$di->set('dispatcher', function() {
    $dispatcher = new \Phalcon\Mvc\Dispatcher();
    $dispatcher->setDefaultNamespace('Single\Controllers\\');
    return $dispatcher;
});

// Registering the view component
$di->set('view', function() {
    $view = new \Phalcon\Mvc\View();
    $view->setViewsDir('../apps/views/');
    return $view;
});

try {
    $application = new \Phalcon\Mvc\Application();
    $application->setDI($di);
    echo $application->handle()->getContent();
} catch(Phalcon\Exception $e){
    echo $e->getMessage();
}
```

Multi Module

A multi-module application uses the same document root for more than one module. In this case the following file structure can be used:

```
multiple/
  apps/
    frontend/
       controllers/
       models/
       views/
       Module.php
    backend/
       controllers/
       models/
       views/
       Module.php
  public/
    css/
    img/
    js/
```

Each directory in apps/ have its own MVC structure. A Module.php is present to configure specific settings of each module like autoloaders or custom services:

```php
<?php

namespace Multiple\Backend;

use Phalcon\Mvc\ModuleDefinitionInterface;

class Module implements ModuleDefinitionInterface
{

    /**
     * Register a specific autoloader for the module
     */
    public function registerAutoloaders()
    {

        $loader = new \Phalcon\Loader();

        $loader->registerNamespaces(
            array(
                'Multiple\Backend\Controllers' => '../apps/backend/controllers/',
                'Multiple\Backend\Models'      => '../apps/backend/models/',
            )
        );

        $loader->register();
    }

    /**
     * Register specific services for the module
     */
    public function registerServices($di)
    {

        //Registering a dispatcher
        $di->set('dispatcher', function() {
            $dispatcher = new \Phalcon\Mvc\Dispatcher();
            $dispatcher->setDefaultNamespace("Multiple\Backend\Controllers\\");
            return $dispatcher;
        });

        //Registering the view component
        $di->set('view', function() {
            $view = new \Phalcon\Mvc\View();
            $view->setViewsDir('../apps/backend/views/');
            return $view;
        });
    }

}
```

A special bootstrap file is required to load the a multi-module MVC architecture:

```php
<?php

$di = new \Phalcon\DI\FactoryDefault();

//Specify routes for modules
$di->set('router', function () {
```

```
    $router = new \Phalcon\Mvc\Router();

    $router->setDefaultModule("frontend");

    $router->add(
        "/login",
        array(
            'module'     => 'backend',
            'controller' => 'login',
            'action'     => 'index',
        )
    );

    $router->add(
        "/admin/products/:action",
        array(
            'module'     => 'backend',
            'controller' => 'products',
            'action'     => 1,
        )
    );

    $router->add(
        "/products/:action",
        array(
            'controller' => 'products',
            'action'     => 1,
        )
    );

    return $router;

});

try {

    //Create an application
    $application = new \Phalcon\Mvc\Application();
    $application->setDI($di);

    // Register the installed modules
    $application->registerModules(
        array(
            'frontend' => array(
                'className' => 'Multiple\Frontend\Module',
                'path'      => '../apps/frontend/Module.php',
            ),
            'backend'  => array(
                'className' => 'Multiple\Backend\Module',
                'path'      => '../apps/backend/Module.php',
            )
        )
    );

    //Handle the request
    echo $application->handle()->getContent();

} catch(Phalcon\Exception $e){
```

```
    echo $e->getMessage();
}
```

If you want to maintain the module configuration in the bootstrap file you can use an anonymous function to register the module:

```
<?php

//Creating a view component
$view = new \Phalcon\Mvc\View();

// Register the installed modules
$application->registerModules(
    array(
        'frontend' => function($di) use ($view) {
            $di->setShared('view', function() use ($view) {
                $view->setViewsDir('../apps/frontend/views/');
                return $view;
            });
        },
        'backend' => function($di) use ($view) {
            $di->setShared('view', function() use ($view) {
                $view->setViewsDir('../apps/frontend/views/');
                return $view;
            });
        }
    )
);
```

When *Phalcon\Mvc\Application* have modules registered, always is necessary that every matched route returns a valid module. Each registered module has an associated class offering functions to set the module itself up. Each module class definition must implement two methods: registerAutoloaders() and registerServices(), they will be called by *Phalcon\Mvc\Application* according to the module to be executed.

2.17.2 Understanding the default behavior

If you've been following the *tutorial* or have generated the code using *Phalcon Devtools*, you may recognize the following bootstrap file:

```
<?php

try {

    // Register autoloaders
    //...

    // Register services
    //...

    // Handle the request
    $application = new \Phalcon\Mvc\Application();
    $application->setDI($di);
    echo $application->handle()->getContent();

} catch (\Phalcon\Exception $e) {
    echo "PhalconException: ", $e->getMessage();
}
```

The core of all the work of the controller occurs when handle() is invoked:

```
<?php

echo $application->handle()->getContent();
```

If you do not wish to use *Phalcon\Mvc\Application*, the code above can be changed as follows:

```
<?php

// Request the services from the services container
$router = $di->get('router');
$router->handle();

$view = $di->getShared('view');

$dispatcher = $di->get('dispatcher');

// Pass the proccessed router parameters to the dispatcher
$dispatcher->setControllerName($router->getControllerName());
$dispatcher->setActionName($router->getActionName());
$dispatcher->setParams($router->getParams());

// Start the view
$view->start();

// Dispatch the request
$dispatcher->dispatch();

// Render the related views
$view->render(
    $dispatcher->getControllerName(),
    $dispatcher->getActionName(),
    $dispatcher->getParams()
);

// Finish the view
$view->finish();

$response = $di->get('response');

// Pass the output of the view to the response
$response->setContent($view->getContent());

// Send the request headers
$response->sendHeaders();

// Print the response
echo $response->getContent();
```

Although the above is a lot more verbose than the code needed while using *Phalcon\Mvc\Application*, it offers an alternative in boostraping your application. Depending on your needs, you might want to have full control of what should be instantiated or not, or replace certain components with those of your own to extend the default functionality.

2.17.3 Application Events

Phalcon\Mvc\Application is able to send events to the *EventsManager* (if it is present). Events are triggered using the type "application". The following events are supported:

Event Name	Triggered
beforeStartModule	Before initialize a module, only when modules are registered
afterStartModule	After initialize a module, only when modules are registered
beforeHandleRequest	Before execute the dispatch loop
afterHandleRequest	After execute the dispatch loop

The following example demonstrates how to attach listeners to this component:

```
<?php

$eventsManager = new Phalcon\Events\Manager();

$application->setEventsManager($eventsManager);

$eventsManager->attach(
    "application",
    function($event, $application) {
        // ...
    }
);
```

2.17.4 External Resources

- MVC examples on Github

2.18 Routing

The router component allows defining routes that are mapped to controllers or handlers that should receive the request. A router simply parses a URI to determine this information. The router has two modes: MVC mode and match-only mode. The first mode is ideal for working with MVC applications.

2.18.1 Defining Routes

Phalcon\Mvc\Router provides advanced routing capabilities. In MVC mode, you can define routes and map them to controllers/actions that you require. A route is defined as follows:

```
<?php

// Create the router
$router = new \Phalcon\Mvc\Router();

//Define a route
$router->add(
    "/admin/users/my-profile",
    array(
        "controller" => "users",
        "action"     => "profile",
    )
);

//Another route
$router->add(
    "/admin/users/change-password",
    array(
```

```
        "controller" => "users",
        "action"     => "changePassword",
    )
);

$router->handle();
```

The method add() receives as first parameter a pattern and optionally a set of paths as second parameter. In this case, if the URI is exactly: /admin/users/my-profile, then the "users" controller with its action "profile" will be executed. Currently, the router does not execute the controller and action, it only collects this information to inform the correct component (ie. *Phalcon\Mvc\Dispatcher*) that this is controller/action it should to execute.

An application can have many paths, define routes one by one can be a cumbersome task. In these cases we can create more flexible routes:

```
<?php

// Create the router
$router = new \Phalcon\Mvc\Router();

//Define a route
$router->add(
    "/admin/:controller/a/:action/:params",
    array(
        "controller" => 1,
        "action"     => 2,
        "params"     => 3,
    )
);
```

In the example above, using wildcards we make a route valid for many URIs. For example, by accessing the following URL (/admin/users/a/delete/dave/301) then:

Controller	users
Action	delete
Parameter	dave
Parameter	301

The method add() receives a pattern that optionally could have predefined placeholders and regular expression modifiers. All the routing patterns must start with a slash character (/). The regular expression syntax used is the same as the PCRE regular expressions. Note that, it is not necessary to add regular expression delimiters. All routes patterns are case-insensitive.

The second parameter defines how the matched parts should bind to the controller/action/parameters. Matching parts are placeholders or subpatterns delimited by parentheses (round brackets). In the example given above, the first subpattern matched (:controller) is the controller part of the route, the second the action and so on.

These placeholders help writing regular expressions that are more readable for developers and easier to understand. The following placeholders are supported:

Placeholder	Regular Expression	Usage
/:module	/([a-zA-Z0-9_-]+)	Matches a valid module name with alpha-numeric characters only
/:controller	/([a-zA-Z0-9_-]+)	Matches a valid controller name with alpha-numeric characters only
/:action	/([a-zA-Z0-9_]+)	Matches a valid action name with alpha-numeric characters only
/:params	(/.*)*	Matches a list of optional words separated by slashes
/:namespace	/([a-zA-Z0-9_-]+)	Matches a single level namespace name
/:int	/([0-9]+)	Matches an integer parameter

Controller names are camelized, this means that characters (-) and (_) are removed and the next character is uppercased.

For instance, some_controller is converted to SomeController.

Since you can add many routes as you need using add(), the order in which routes are added indicate their relevance, lastest routes added have more relevance than first added. Internally, all defined routes are traversed in reverse order until *Phalcon\Mvc\Router* finds the one that matches the given URI and processes it, while ignoring the rest.

Parameters with Names

The example below demonstrates how to define names to route parameters:

```
<?php

$router->add(
    "/news/([0-9]{4})/([0-9]{2})/([0-9]{2})/:params",
    array(
            "controller" => "posts",
            "action"     => "show",
            "year"       => 1, // ([0-9]{4})
            "month"      => 2, // ([0-9]{2})
            "day"        => 3, // ([0-9]{2})
            "params"     => 4, // :params
    )
);
```

In the above example, the route doesn't define a "controller" or "action" part. These parts are replaced with fixed values ("posts" and "show"). The user will not know the controller that is really dispatched by the request. Inside the controller, those named parameters can be accessed as follows:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function showAction()
    {

        // Return "year" parameter
        $year = $this->dispatcher->getParam("year");

        // Return "month" parameter
        $month = $this->dispatcher->getParam("month");

        // Return "day" parameter
        $day = $this->dispatcher->getParam("day");

    }

}
```

Note that the values of the parameters are obtained from the dispatcher. This happens because it is the component that finally interacts with the drivers of your application. Moreover, there is also another way to create named parameters as part of the pattern:

```
<?php

$router->add(
    "/documentation/{chapter}/{name}.{type:[a-z]+}",
    array(
        "controller" => "documentation",
        "action"     => "show"
    )
);
```

You can access their values in the same way as before:

```
<?php

class DocumentationController extends \Phalcon\Mvc\Controller
{

    public function showAction()
    {

        // Returns "name" parameter
        $year = $this->dispatcher->getParam("name");

        // Returns "type" parameter
        $year = $this->dispatcher->getParam("type");

    }

}
```

Short Syntax

If you don't like using an array to define the route paths, an alternative syntax is also available. The following examples produce the same result:

```
<?php

// Short form
$router->add("/posts/{year:[0-9]+}/{title:[a-z\-]+}", "Posts::show");

// Array form
$router->add(
    "/posts/([0-9]+)/([a-z\-]+)",
    array(
       "controller" => "posts",
       "action"     => "show",
       "year"       => 1,
       "title"      => 2,
    )
);
```

Mixing Array and Short Syntax

Array and short syntax can be mixed to define a route, in this case note that named parameters automatically are added to the route paths according to the position on which they were defined:

```
<?php

//First position must be skipped because it is used for
//the named parameter 'country'
$router->add('/news/{country:[a-z]{2}}/([a-z+])/([a-z\-+])',
    array(
        'section' => 2, //Positions start with 2
        'article' => 3
    )
);
```

Routing to Modules

You can define routes whose paths include modules. This is specially suitable to multi-module applications. It's possible define a default route that includes a module wildcard:

```
<?php

$router = new Phalcon\Mvc\Router(false);

$router->add('/:module/:controller/:action/:params', array(
    'module' => 1,
    'controller' => 2,
    'action' => 3,
    'params' => 4
));
```

In this case, the route always must have the module name as part of the URL. For example, the following URL: /admin/users/edit/sonny, will be processed as:

Module	admin
Controller	users
Action	edit
Parameter	sonny

Or you can bind specific routes to specific modules:

```
<?php

$router->add("/login", array(
    'module' => 'backend',
    'controller' => 'login',
    'action' => 'index',
));

$router->add("/products/:action", array(
    'module' => 'frontend',
    'controller' => 'products',
    'action' => 1,
));
```

Or bind them to specific namespaces:

```
<?php

$router->add("/:namespace/login", array(
    'namespace' => 1,
    'controller' => 'login',
```

```
    'action' => 'index'
));
```

Namespaces/class names must be passed separated:

```
<?php

$router->add("/login", array(
    'namespace' => 'Backend\Controllers',
    'controller' => 'login',
    'action' => 'index'
));
```

HTTP Method Restrictions

When you add a route using simply add(), the route will be enabled for any HTTP method. Sometimes we can restrict a route to a specific method, this is especially useful when creating RESTful applications:

```
<?php

// This route only will be matched if the HTTP method is GET
$router->addGet("/products/edit/{id}", "Posts::edit");

// This route only will be matched if the HTTP method is POST
$router->addPost("/products/save", "Posts::save");

// This route will be matched if the HTTP method is POST or PUT
$router->add("/products/update")->via(array("POST", "PUT"));
```

Using convertions

Convertions allow to freely transform the route's parameters before passing them to the dispatcher, the following examples show how to use them:

```
<?php

//The action name allows dashes, an action can be: /products/new-ipod-nano-4-generation
$router
    ->add('/products/{slug:[a-z\-]+}', array(
        'controller' => 'products',
        'action' => 'show'
    ))
    ->convert('slug', function($slug) {
        //Transform the slug removing the dashes
        return str_replace('-', '', $slug);
    });
```

Groups of Routes

If a set of routes have common paths they can be grouped to easily maintain them:

```
<?php

$router = new \Phalcon\Mvc\Router();
```

```
//Create a group with a common module and controller
$blog = new \Phalcon\Mvc\Router\Group(array(
    'module' => 'blog',
    'controller' => 'index'
));

//All the routes start with /blog
$blog->setPrefix('/blog');

//Add a route to the group
$blog->add('/save', array(
    'action' => 'save'
));

//Add another route to the group
$blog->add('/edit/{id}', array(
    'action' => 'edit'
));

//This route maps to a controller different than the default
$blog->add('/blog', array(
    'controller' => 'about',
    'action' => 'index'
));

//Add the group to the router
$router->mount($blog);
```

2.18.2 Matching Routes

A valid URI must be passed to Router in order to let it checks the route that matches that given URI. By default, the routing URI is taken from the $_GET['_url'] variable that is created by the rewrite engine module. A couple of rewrite rules that work very well with Phalcon are:

```
RewriteEngine On
RewriteCond   %{REQUEST_FILENAME} !-d
RewriteCond   %{REQUEST_FILENAME} !-f
RewriteRule   ^(.*)$ index.php?_url=/$1 [QSA,L]
```

The following example shows how to use this component in stand-alone mode:

```
<?php

// Creating a router
$router = new \Phalcon\Mvc\Router();

// Define routes here if any
// ...

// Taking URI from $_GET["_url"]
$router->handle();

// or Setting the URI value directly
$router->handle("/employees/edit/17");

// Getting the processed controller
echo $router->getControllerName();
```

```
// Getting the processed action
echo $router->getActionName();

//Get the matched route
$route = $router->getMatchedRoute();
```

2.18.3 Naming Routes

Each route that is added to the router is stored internally as an object *Phalcon\Mvc\Router\Route*. That class encapsulates all the details of each route. For instance, we can give a name to a path to identify it uniquely in our application. This is especially useful if you want to create URLs from it.

```
<?php

$route = $router->add("/posts/{year}/{title}", "Posts::show");

$route->setName("show-posts");

//or just

$router->add("/posts/{year}/{title}", "Posts::show")->setName("show-posts");
```

Then, using for example the component *Phalcon\Mvc\Url* we can build routes from its name:

```
<?php

// returns /posts/2012/phalcon-1-0-released
echo $url->get(array(
    "for" => "show-posts",
    "year" => "2012", "title" =>
    "phalcon-1-0-released"
));
```

2.18.4 Usage Examples

The following are examples of custom routes:

```
<?php

// matches "/system/admin/a/edit/7001"
$router->add(
    "/system/:controller/a/:action/:params",
    array(
        "controller" => 1,
        "action"     => 2,
        "params"     => 3
    )
);

// matches "/es/news"
$router->add(
    "/([a-z]{2})/:controller",
    array(
        "controller" => 2,
        "action"     => "index",
```

```
        "language"   => 1
    )
);

// matches "/es/news"
$router->add(
    "/{language:[a-z]{2}}/:controller",
    array(
        "controller" => 2,
        "action"     => "index"
    )
);

// matches "/admin/posts/edit/100"
$router->add(
    "/admin/:controller/:action/:int",
    array(
        "controller" => 1,
        "action"     => 2,
        "id"         => 3
    )
);

// matches "/posts/2010/02/some-cool-content"
$router->add(
    "/posts/([0-9]{4})/([0-9]{2})/([a-z\-]+)",
    array(
        "controller" => "posts",
        "action"     => "show",
        "year"       => 1,
        "month"      => 2,
        "title"      => 4
    )
);

// matches "/manual/en/translate.adapter.html"
$router->add(
    "/manual/([a-z]{2})/([a-z\.]+)\.html",
    array(
        "controller" => "manual",
        "action"     => "show",
        "language"   => 1,
        "file"       => 2
    )
);

// matches /feed/fr/le-robots-hot-news.atom
$router->add(
    "/feed/{lang:[a-z]+}/{blog:[a-z\-]+}\.{type:[a-z\-]+}",
    "Feed::get"
);

// matches /api/v1/users/peter.json
$router->add('/api/(v1|v2)/{method:[a-z]+}/{param:[a-z]+}\.(json|xml)', array(
    'controller' => 'api',
    'version' => 1,
    'format' => 4
));
```

> Beware of characters allowed in regular expression for controllers and namespaces. As these become class names and in turn they're passed through the file system could be used by attackers to read unauthorized files. A safe regular expression is: /([a-zA-Z0-9_-]+)

2.18.5 Default Behavior

Phalcon\Mvc\Router has a default behavior providing a very simple routing that always expects a URI that matches the following pattern: /:controller/:action/:params

For example, for a URL like this *http://phalconphp.com/documentation/show/about.html*, this router will translate it as follows:

Controller	documentation
Action	show
Parameter	about.html

If you don't want use this routes as default in your application, you must create the router passing false as parameter:

```
<?php

// Create the router without default routes
$router = new \Phalcon\Mvc\Router(false);
```

2.18.6 Setting the default route

When your application is accessed without any route, the '/' route is used to determine what paths must be used to show the initial page in your website/application:

```
<?php

$router->add("/", array(
    'controller' => 'index',
    'action' => 'index'
));
```

2.18.7 Not Found Paths

If none of the routes specified in the router are matched, you can define a group of paths to be used in this scenario:

```
<?php

//Set 404 paths
$router->notFound(array(
    "controller" => "index",
    "action" => "route404"
));
```

2.18.8 Setting default paths

It's possible to define default values for common paths like module, controller or action. When a route is missing any of those paths they can be automatically filled by the router:

```
<?php

//Individually
$router->setDefaultModule("backend");
$router->setDefaultNamespace('Backend\Controllers');
$router->setDefaultController("index");
$router->setDefaultAction("index");

//Using an array
$router->setDefaults(array(
    "controller" => "index",
    "action" => "index"
));
```

2.18.9 Dealing with extra/trailing slashes

Sometimes a route could be accessed with extra/trailing slashes and the end of the route, those extra slashes would lead to produce a not-found status in the dispatcher. You can set up the router to automatically remove the slashes from the end of handled route:

```
<?php

$router = new \Phalcon\Mvc\Router();

//Remove trailing slashes automatically
$router->removeExtraSlashes(true);
```

Or, you can modify specific routes to optionally accept trailing slashes:

```
<?php

$router->add(
    "/{language:[a-z]{2}}/:controller[/]{0,1}",
    array(
        "controller" => 2,
        "action"     => "index"
    )
);
```

2.18.10 URI Sources

By default the URI information is obtained from the $_GET['_url'] variable, this is passed by the Rewrite-Engine to Phalcon, you can also use $_SERVER['REQUEST_URI'] if required:

```
<?php

$router->setUriSource(Router::URI_SOURCE_GET_URL); // use $_GET['_url'] (default)
$router->setUriSource(Router::URI_SOURCE_SERVER_REQUEST_URI); // use $_SERVER['REQUEST_URI'] (defaul
```

Or you can manually pass a URI to the 'handle' method:

```
<?php

$router->handle('/some/route/to/handle');
```

2.18.11 Testing your routes

Since this component has no dependencies, you can create a file as shown below to test your routes:

```
<?php

//These routes simulate real URIs
$testRoutes = array(
    '/',
    '/index',
    '/index/index',
    '/index/test',
    '/products',
    '/products/index/',
    '/products/show/101',
);

$router = new Phalcon\Mvc\Router();

//Add here your custom routes
//...

//Testing each route
foreach ($testRoutes as $testRoute) {

    //Handle the route
    $router->handle($testRoute);

    echo 'Testing ', $testRoute, '<br>';

    //Check if some route was matched
    if ($router->wasMatched()) {
        echo 'Controller: ', $router->getControllerName(), '<br>';
        echo 'Action: ', $router->getActionName(), '<br>';
    } else {
        echo 'The route wasn\'t matched by any route<br>';
    }
    echo '<br>';

}
```

2.18.12 Annotations Router

This component provides a variant that's integrated with the *annotations* service. Using this strategy you can write the routes directly in the controllers instead of adding them in the service registration:

```
<?php

$di['router'] = function() {

    //Use the annotations router
    $router = new \Phalcon\Mvc\Router\Annotations(false);

    //Read the annotations from ProductsController if the uri starts with /api/products
    $router->addResource('Products', '/api/products');

    return $router;
```

```
};
```

The annotations can be defined in the following way:

```
<?php

/**
 * @RoutePrefix("/api/products")
 */
class ProductsController
{

    /**
     * @Get("/")
     */
    public function indexAction()
    {

    }

    /**
     * @Get("/edit/{id:[0-9]+}", name="edit-robot")
     */
    public function editAction($id)
    {

    }

    /**
     * @Route("/save", methods={"POST", "PUT"}, name="save-robot")
     */
    public function saveAction()
    {

    }

    /**
     * @Route("/delete/{id:[0-9]+}", methods="DELETE",
     *      conversors={id="MyConversors::checkId"})
     */
    public function deleteAction($id)
    {

    }

    public function infoAction($id)
    {

    }

}
```

Only methods marked with valid annotations are used as routes. List of annotations supported:

Name	Description	Usage
RoutePrefix	A prefix to be prepended to each route uri. This annotation must be placed at the class' docblock	@RoutePrefix("/api/products")
Route	This annotation marks a method as a route. This annotation must be placed in a method docblock	@Route("/api/products/show")
Get	This annotation marks a method as a route restricting the HTTP method to GET	@Get("/api/products/search")
Post	This annotation marks a method as a route restricting the HTTP method to POST	@Post("/api/products/save")
Put	This annotation marks a method as a route restricting the HTTP method to PUT	@Put("/api/products/save")
Delete	This annotation marks a method as a route restricting the HTTP method to DELETE	@Delete("/api/products/delete/{id}")
Options	This annotation marks a method as a route restricting the HTTP method to OPTIONS	@Option("/api/products/info")

For annotations that add routes, the following parameters are supported:

Name	Description	Usage
methods	Define one or more HTTP method that route must meet with	@Route("/api/products", methods={"GET", "POST"})
name	Define a name for the route	@Route("/api/products", name="get-products")
paths	An array of paths like the one passed to Phalcon\Mvc\Router::add	@Route("/posts/{id}/{slug}", paths={module="backend"})
conversors	A hash of conversors to be applied to the parameters	@Route("/posts/{id}/{slug}", conversors={id="MyConversor::getId"})

If routes map to controllers in modules is better use the addModuleResource method:

```
<?php

$di['router'] = function() {

    //Use the annotations router
    $router = new \Phalcon\Mvc\Router\Annotations(false);

    //Read the annotations from Backend\Controllers\ProductsController if the uri starts with /api/p
    $router->addModuleResource('backend', 'Products', '/api/products');

    return $router;
};
```

2.18.13 Implementing your own Router

The *Phalcon\Mvc\RouterInterface* interface must be implemented to create your own router replacing the one provided by Phalcon.

2.19 Dispatching Controllers

Phalcon\Mvc\Dispatcher is the component responsible of instantiate controllers and execute the required actions on them in an MVC application. Understand its operation and capabilities helps us get more out of the services provided by the framework.

2.19.1 The Dispatch Loop

This is an important process that has much to do with the MVC flow itself, especially with the controller part. The work occurs within the controller dispatcher. The controller files are read, loaded, instantiated, to then the required actions are executed. If an action forwards the flow to another controller/action, the controller dispatcher starts again. To better illustrate this, the following example shows approximately the process performed within *Phalcon\Mvc\Dispatcher*:

```
<?php

//Dispatch loop
while (!$finished) {

    $finished = true;

    $controllerClass = $controllerName."Controller";

    //Instantiating the controller class via autoloaders
    $controller = new $controllerClass();

    // Execute the action
    call_user_func_array(array($controller, $actionName . "Action"), $params);

    // Finished should be reloaded to check if the flow was forwarded to another controller
    // $finished = false;

}
```

The code above lacks validations, filters and additional checks, but it demonstrates the normal flow of operation in the dispatcher.

Dispatch Loop Events

Phalcon\Mvc\Dispatcher is able to send events to an *EventsManager* if it is present. Events are triggered using the type "dispatch". Some events when returning boolean false could stop the active operation. The following events are supported:

Event Name	Triggered	Can stop operation?
beforeDispatchLoop	Triggered before entering in the dispatch loop. At this point the dispatcher don't know if the controller or the actions to be executed exist. The Dispatcher only knows the information passed by the Router.	Yes
beforeDispatch	Triggered after entering in the dispatch loop. At this point the dispatcher don't know if the controller or the actions to be executed exist. The Dispatcher only knows the information passed by the Router.	Yes
beforeExecuteRoute	Triggered before executing the controller/action method. At this point the dispatcher has been initialized the controller and know if the action exist.	Yes
afterExecuteRoute	Triggered after executing the controller/action method. As operation cannot be stopped, only use this event to make clean up after execute the action	No
beforeNotFoundAction	Triggered when the action was not found in the controller	Yes
beforeException	Triggered before the dispatcher throws any exception	Yes
afterDispatch	Triggered after executing the controller/action method. As operation cannot be stopped, only use this event to make clean up after execute the action	Yes
afterDispatchLoop	Triggered after exiting the dispatch loop	No

The *INVO* tutorial shows how to take advantage of dispatching events implementing a security filter with *Acl*

The following example demonstrates how to attach listeners to this component:

```
<?php

$di->set('dispatcher', function(){

    //Create an event manager
    $eventsManager = new Phalcon\Events\Manager();

    //Attach a listener for type "dispatch"
    $eventsManager->attach("dispatch", function($event, $dispatcher) {
        //...
    });

    $dispatcher = new \Phalcon\Mvc\Dispatcher();

    //Bind the eventsManager to the view component
    $dispatcher->setEventsManager($eventsManager);

    return $dispatcher;

}, true);
```

An instantiated controller automatically acts as a listener for dispatch events, so you can implement methods as callbacks:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function beforeExecuteRoute($dispatcher)
    {
```

```
        // Executed before every found action
    }

    public function afterExecuteRoute($dispatcher)
    {
        // Executed after every found action
    }

}
```

2.19.2 Forwarding to other actions

The dispatch loop allows us to forward the execution flow to another controller/action. This is very useful to check if the user can access to certain options, redirect users to other screens or simply reuse code.

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function saveAction($year, $postTitle)
    {

        // .. store some product and forward the user

        // Forward flow to the index action
        $this->dispatcher->forward(array(
            "controller" => "post",
            "action" => "index"
        ));
    }

}
```

Keep in mind that making a "forward" is not the same as making an HTTP redirect. Although they apparently got the same result. The "forward" doesn't reload the current page, all the redirection occurs in a single request, while the HTTP redirect needs two requests to complete the process.

More forwarding examples:

```
<?php

// Forward flow to another action in the current controller
$this->dispatcher->forward(array(
    "action" => "search"
));

// Forward flow to another action in the current controller
// passing parameters
$this->dispatcher->forward(array(
    "action" => "search",
    "params" => array(1, 2, 3)
```

```
));
```

A forward action accepts the following parameters:

Parameter	Triggered
controller	A valid controller name to forward to.
action	A valid action name to forward to.
params	An array of parameters for the action
namespace	A valid namespace name where the controller is part of

2.19.3 Getting Parameters

When a route provides named parameters you can receive them in a controller, a view or any other component that extends *Phalcon\DI\Injectable*.

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function saveAction()
    {

        // Get the post's title passed in the URL as parameter
        $title = $this->dispatcher->getParam("title");

        // Get the post's year passed in the URL as parameter
        // also filtering it
        $year = $this->dispatcher->getParam("year", "int");
    }

}
```

2.19.4 Handling Not-Found Exceptions

Using the *EventsManager* it's possible to insert a hook point before the dispatcher throws an exception when a controller/action wasn't found.

```
<?php

$di->setShared('dispatcher', function() {

    //Create/Get an EventManager
    $eventsManager = new Phalcon\Events\Manager();

    //Attach a listener
    $eventsManager->attach("dispatch", function($event, $dispatcher, $exception) {

        //The controller exists but the action not
        if ($event->getType() == 'beforeNotFoundAction') {
```

```
            $dispatcher->forward(array(
                'controller' => 'index',
                'action' => 'show404'
            ));
            return false;
        }

        //Alternative way, controller or action doesn't exist
        if ($event->getType() == 'beforeException') {
            switch ($exception->getCode()) {
                case Phalcon\Dispatcher::EXCEPTION_HANDLER_NOT_FOUND:
                case Phalcon\Dispatcher::EXCEPTION_ACTION_NOT_FOUND:
                    $dispatcher->forward(array(
                        'controller' => 'index',
                        'action' => 'show404'
                    ));
                    return false;
            }
        }
    });

    $dispatcher = new Phalcon\Mvc\Dispatcher();

    //Bind the EventsManager to the dispatcher
    $dispatcher->setEventsManager($eventsManager);

    return $dispatcher;

}, true);
```

2.19.5 Implementing your own Dispatcher

The *Phalcon\Mvc\DispatcherInterface* interface must be implemented to create your own dispatcher replacing the one provided by Phalcon.

2.20 Micro Applications

With Phalcon you can create "Micro-Framework like" applications. By doing this, you only need to write a minimal amount of code to create a PHP application. Micro applications are suitable to implement small applications, APIs and prototypes in a practical way.

```
<?php

$app = new Phalcon\Mvc\Micro();

$app->get('/say/welcome/{name}', function ($name) {
    echo "<h1>Welcome $name!</h1>";
});

$app->handle();
```

2.20.1 Creating a Micro Application

Phalcon\Mvc\Micro is the class responsible for implementing a micro application.

```
<?php

$app = new Phalcon\Mvc\Micro();
```

2.20.2 Defining routes

After instantiating the object, you will need to add some routes. *Phalcon\Mvc\Router* manages routing internally. Routes must always start with /. A HTTP method constraint is optionally required when defining routes, so as to instruct the router to match only if the request also matches the HTTP methods. The following example shows how to define a route for the method GET:

```
<?php

$app->get('/say/hello/{name}', function ($name) {
    echo "<h1>Hello! $name</h1>";
});
```

The "get" method indicates that the associated HTTP method is GET. The route /say/hello/{name} also has a parameter {$name} that is passed directly to the route handler (the anonymous function). Handlers are executed when a route is matched. A handler could be any callable item in the PHP userland. The following example shows how to define different types of handlers:

```
<?php

// With a function
function say_hello($name) {
    echo "<h1>Hello! $name</h1>";
}

$app->get('/say/hello/{name}', "say_hello");

// With a static method
$app->get('/say/hello/{name}', "SomeClass::someSayMethod");

// With a method in an object
$myController = new MyController();
$app->get('/say/hello/{name}', array($myController, "someAction"));

//Anonymous function
$app->get('/say/hello/{name}', function ($name) {
    echo "<h1>Hello! $name</h1>";
});
```

Phalcon\Mvc\Micro provides a set of methods to define the HTTP method (or methods) which the route is constrained for:

```
<?php

//Matches if the HTTP method is GET
$app->get('/api/products', "get_products");

//Matches if the HTTP method is POST
$app->post('/api/products/add', "add_product");
```

```
//Matches if the HTTP method is PUT
$app->put('/api/products/update/{id}', "update_product");

//Matches if the HTTP method is DELETE
$app->put('/api/products/remove/{id}', "delete_product");

//Matches if the HTTP method is OPTIONS
$app->options('/api/products/info/{id}', "info_product");

//Matches if the HTTP method is PATCH
$app->patch('/api/products/update/{id}', "info_product");

//Matches if the HTTP method is GET or POST
$app->map('/repos/store/refs')->via(array('GET', 'POST'));
```

Routes with Parameters

Defining parameters in routes is very easy as demonstrated above. The name of the parameter has to be enclosed in brackets. Parameter formatting is also available using regular expressions to ensure consistency of data. This is demonstrated in the example below:

```
<?php

//This route have two parameters and each of them have a format
$app->get('/posts/{year:[0-9]+}/{title:[a-zA-Z\-]+}', function ($year, $title) {
    echo "<h1>Title: $title</h1>";
    echo "<h2>Year: $year</h2>";
});
```

Starting Route

Normally, the starting route in an application is the route /, and it will more frequent to be accessed by the method GET. This scenario is coded as follows:

```
<?php

//This is the start route
$app->get('/', function () {
    echo "<h1>Welcome!</h1>";
});
```

Rewrite Rules

The following rules can be used together with Apache to rewrite the URis:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^(.*)$ index.php?_url=/$1 [QSA,L]
</IfModule>
```

2.20.3 Working with Responses

You are free to produce any kind of response in a handler: directly make an output, use a template engine, include a view, return a json, etc.:

```
<?php

//Direct output
$app->get('/say/hello', function () {
    echo "<h1>Hello! $name</h1>";
});

//Requiring another file
$app->get('/show/results', function () {
    require 'views/results.php';
});

//Returning a JSON
$app->get('/get/some-json', function () {
    echo json_encode(array("some", "important", "data"));
});
```

In addition to that, you have access to the service *"response"*, with which you can manipulate better the response:

```
<?php

$app->get('/show/data', function () use ($app) {

    //Set the Content-Type header
    $app->response->setContentType('text/plain')->sendHeaders();

    //Print a file
    readfile("data.txt");

});
```

2.20.4 Making redirections

Redirections could be performed to forward the execution flow to another route:

```
<?php

//This route makes a redirection to another route
$app->post('/old/welcome', function () use ($app) {
    $app->response->redirect("new/welcome");
});

$app->post('/new/welcome', function () use ($app) {
    echo 'This is the new Welcome';
});
```

2.20.5 Generating URLs for Routes

Phalcon\Mvc\Url can be used to produce URLs based on the defined routes. You need to set up a name for the route; by this way the "url" service can produce the corresponding URL:

```
<?php

//Set a route with the name "show-post"
$app->get('/blog/{year}/{title}', function ($year, $title) use ($app) {

    //.. show the post here

})->setName('show-post');

//produce a url somewhere
$app->get('/', function() use ($app) {

    echo '<a href="', $app->url->get(array(
        'for' => 'show-post',
        'title' => 'php-is-a-great-framework',
        'year' => 2012
    )), '">Show the post</a>';

});
```

2.20.6 Interacting with the Dependency Injector

In the micro application, a *Phalcon\DI\FactoryDefault* services container is created implicitly; additionally you can create outside the application a container to manipulate its services:

```
<?php

$di = new \Phalcon\DI\FactoryDefault();

$di->set('config', function() {
    return new \Phalcon\Config\Adapter\Ini("config.ini");
});

$app = new Phalcon\Mvc\Micro();

$app->setDI($di);

$app->get('/', function () use ($app) {
    //Read a setting from the config
    echo $app->config->app_name;
});

$app->post('/contact', function () use ($app) {
    $app->flash->success('Yes!, the contact was made!');
});
```

The array-syntax is allowed to easily set/get services in the internal services container:

```
<?php

$app = new Phalcon\Mvc\Micro();

//Setup the database service
$app['db'] = function() {
    return new \Phalcon\Db\Adapter\Pdo\Mysql(array(
        "host" => "localhost",
        "username" => "root",
```

```
        "password" => "secret",
        "dbname" => "test_db"
    ));
};

$app->get('/blog', function () use ($app) {
    $news = $app['db']->query('SELECT * FROM news');
    foreach ($news as $new) {
        echo $new->title;
    }
});
```

2.20.7 Not-Found Handler

When a user tries to access a route that is not defined, the micro application will try to execute the "Not-Found" handler. An example of that behavior is below:

```
<?php

$app->notFound(function () use ($app) {
    $app->response->setStatusCode(404, "Not Found")->sendHeaders();
    echo 'This is crazy, but this page was not found!';
});
```

2.20.8 Models in Micro Applications

Models can be used transparently in Micro Applications, only is required an autoloader to load models:

```
<?php

$loader = new \Phalcon\Loader();

$loader->registerDirs(array(
    __DIR__ . '/models/'
))->register();

$app = new \Phalcon\Mvc\Micro();

$app->get('/products/find', function(){

    foreach (Products::find() as $product) {
        echo $product->name, '<br>';
    }

});

$app->handle();
```

2.20.9 Micro Application Events

Phalcon\Mvc\Micro is able to send events to the *EventsManager* (if it is present). Events are triggered using the type "micro". The following events are supported:

Event Name	Triggered	Can stop operation?
beforeHandleRoute	The main method is just called, at this point the application doesn't know if there is some matched route	Yes
beforeExecuteRoute	A route has been matched and it contains a valid handler, at this point the handler has not been executed	Yes
afterExecuteRoute	Triggered after running the handler	No
beforeNotFound	Triggered when any of the defined routes match the requested URI	Yes
afterHandleRoute	Triggered after completing the whole process in a successful way	Yes

In the following example, we explain how to control the application security using events:

```
<?php

//Create a events manager
$eventManager = \Phalcon\Events\Manager();

//Listen all the application events
$eventManager->attach('micro', function($event, $app) {

    if ($event->getType() == 'beforeExecuteRoute') {
        if ($app->session->get('auth') == false) {

            $app->flashSession->error("The user isn't authenticated");
            $app->response->redirect("/");

            //Return (false) stop the operation
            return false;
        }
    }

});

$app = new Phalcon\Mvc\Micro();

//Bind the events manager to the app
$app->setEventsManager($eventsManager);
```

2.20.10 Middleware events

In addition to the events manager, events can be added using the methods 'before', 'after' and 'finish':

```
<?php

$app = new Phalcon\Mvc\Micro();

//Executed before every route executed
//Return false cancels the route execution
$app->before(function() use ($app) {
    if ($app['session']->get('auth') == false) {
        return false;
    }
    return true;
});
```

```
$app->map('/api/robots', function(){
    return array(
        'status' => 'OK'
    );
});

$app->after(function() use ($app) {
    //This is executed after the route is executed
    echo json_encode($app->getReturnedValue());
});

$app->finish(function() use ($app) {
    //This is executed when is the request has been served
});
```

You can call the methods several times to add more events of the same type. The following table explains the events:

Event Name	Triggered	Can stop operation?
before	Before executing the handler. It can be used to control the access to the application	Yes
after	Executed after the handler is executed. It can be used to prepare the response	No
finish	Executed after sending the response. It can be used to perform clean-up	No

2.20.11 Returning Responses

Handlers may return raw responses using *Phalcon\Http\Response* or a component that implements the relevant interface.

```
<?php

$app = new Phalcon\Mvc\Micro();

//Return a response
$app->get('/welcome/index', function() {

    $response = new Phalcon\Http\Response();

    $response->setStatusCode(401, "Unauthorized");

    $response->setContent("Access is not authorized");

    return $response;
});
```

2.20.12 Rendering Views

Phalcon\Mvc\View can be used to render views, the following example shows how to do that:

```
<?php

$app = new Phalcon\Mvc\Micro();

$app['view'] = function() {
```

```
    $view = new \Phalcon\Mvc\View();
    $view->setViewsDir('app/views/');
    return $view;
};

//Return a rendered view
$app->get('/products/show', function() use ($app) {

    // Render app/views/products/show.phtml passing some variables
    echo $app['view']->getRender('products', 'show', array(
        'id' => 100,
        'name' => 'Artichoke'
    ));

});
```

Creating a Simple REST API is a tutorial that explains how to create a micro application to implement a RESTful web service.

2.21 Working with Namespaces

Namespaces can be used to avoid class name collisions; this means that if you have two controllers in an application with the same name, a namespace can be used to differentiate them. Namespaces are also useful for creating bundles or modules.

2.21.1 Setting up the framework

Using namespaces has some implications when loading the appropriate controller. To adjust the framework behavior to namespaces is necessary to perform one or all of the following tasks:

Use an autoload strategy that takes into account the namespaces, for example with Phalcon\Loader:

```
<?php

$loader->registerNamespaces(
    array(
       'Store\Admin\Controllers'    => "../bundles/admin/controllers/",
       'Store\Admin\Models'    => "../bundles/admin/models/",
    )
);
```

Specify it in the routes as a separate parameter in the route's paths:

```
<?php

$router->add(
    "/admin/users/my-profile",
    array(
        "namespace"  => "Store\Admin",
        "controller" => "Users",
        "action"     => "profile",
    )
);
```

Passing it as part of the route:

```
<?php

$router->add(
    "/:namespace/admin/users/my-profile",
    array(
        "namespace"  => 1,
        "controller" => "Users",
        "action"     => "profile",
    )
);
```

If you are only working with the same namespace for every controller in your application, then you can define a default namespace in the Dispatcher, by doing this, you don't need to specify a full class name in the router path:

```
<?php

//Registering a dispatcher
$di->set('dispatcher', function() {
    $dispatcher = new \Phalcon\Mvc\Dispatcher();
    $dispatcher->setDefaultNamespace('Store\Admin\Controllers\\');
    return $dispatcher;
});
```

2.21.2 Controllers with Namespaces

The following example shows how to implement a controller that use namespaces:

```
<?php

namespace Store\Admin\Controllers;

class UsersController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function profileAction()
    {

    }

}
```

2.22 Events Manager

The purpose of this component is to intercept the execution of most of the components of the framework by creating "hooks point". These hook points allow the developer to obtain status information, manipulate data or change the flow of execution during the process of a component.

2.22.1 Usage Example

In the following example, we use the EventsManager to listen for events produced in a MySQL connection managed by *Phalcon\Db*. First, we need a listener object to do this. We created a class whose methods are the events we want to listen:

```
<?php

class MyDbListener
{

    public function afterConnect()
    {

    }

    public function beforeQuery()
    {

    }

    public function afterQuery()
    {

    }

}
```

This new class can be as verbose as we need it to. The EventsManager will interface between the component and our listener class, offering hook points based on the methods we defined in our listener class:

```
<?php

$eventsManager = new \Phalcon\Events\Manager();

//Create a database listener
$dbListener = new MyDbListener()

//Listen all the database events
$eventsManager->attach('db', $dbListener);

$connection = new \Phalcon\Db\Adapter\Pdo\Mysql(array(
    "host" => "localhost",
    "username" => "root",
    "password" => "secret",
    "dbname" => "invo"
));

//Assign the eventsManager to the db adapter instance
$connection->setEventsManager($eventsManager);

//Send a SQL command to the database server
$connection->query("SELECT * FROM products p WHERE p.status = 1");
```

In order to log all the SQL statements executed by our application, we need to use the event "afterQuery". The first parameter passed to the event listener contains contextual information about the event that is running, the second is the connection itself.

```php
<?php

class MyDbListener
{

    protected $_logger;

    public function __construct()
    {
        $this->_logger = new \Phalcon\Logger\Adapter\File("../apps/logs/db.log");
    }

    public function afterQuery($event, $connection)
    {
        $this->_logger->log($connection->getSQLStatement(), \Phalcon\Logger::INFO);
    }

}
```

As part of this example, we will also implement the Phalcon\Db\Profiler to detect the SQL statements that are taking longer to execute than expected:

```php
<?php

class MyDbListener
{

    protected $_profiler;

    protected $_logger;

    public function __construct()
    {
        $this->_profiler = new \Phalcon\Db\Profiler();
        $this->_logger = new \Phalcon\Logger\Adapter\File("../apps/logs/db.log");
    }

    public function beforeQuery($event, $connection)
    {
        $this->_profiler->startProfile($connection->getSQLStatement());
    }

    public function afterQuery($event, $connection)
    {
        $this->_logger->log($connection->getSQLStatement(), \Phalcon\Logger::INFO);
        $this->_profiler->stopProfile();
    }

    public function getProfiler()
    {
        return $this->_profiler;
    }

}
```

The resulting profile data can be obtained from the listener:

```php
<?php
```

```
//Send a SQL command to the database server
$connection->query("SELECT * FROM products p WHERE p.status = 1");

foreach($dbListener->getProfiler()->getProfiles() as $profile){
    echo "SQL Statement: ", $profile->getSQLStatement(), "\n";
    echo "Start Time: ", $profile->getInitialTime(), "\n"
    echo "Final Time: ", $profile->getFinalTime(), "\n";
    echo "Total Elapsed Time: ", $profile->getTotalElapsedSeconds(), "\n";
}
```

In a similar manner we can register an lambda function to perform the task instead of a separate listener class (as seen above):

```
<?php

//Listen all the database events
$eventManager->attach('db', function($event, $connection) {
    if ($event->getType() == 'afterQuery') {
        echo $connection->getSQLStatement();
    }
});
```

2.22.2 Creating components that trigger Events

You can create components in your application that trigger events to an EventsManager. As a consequence, there may exist listeners that react to these events when generated. In the following example we're creating a component called "MyComponent". This component is EventsManager aware; when its method "someTask" is executed it triggers two events to any listener in the EventsManager:

```
<?php

class MyComponent implements \Phalcon\Events\EventsAwareInterface
{

    protected $_eventsManager;

    public function setEventsManager($eventsManager)
    {
        $this->_eventsManager = $eventsManager;
    }

    public function getEventsManager()
    {
        return $this->_eventsManager
    }

    public function someTask()
    {
        $this->_eventsManager->fire("my-component:beforeSomeTask", $this);

        // do some task

        $this->_eventsManager->fire("my-component:afterSomeTask", $this);
    }

}
```

Note that events produced by this component are prefixed with "my-component". This is a unique word that helps us identify events that are generated from certain component. You can even generate events outside the component with the same name. Now let's create a listener to this component:

```
<?php

class SomeListener
{

    public function beforeSomeTask($event, $myComponent)
    {
        echo "Here, beforeSomeTask\n";
    }

    public function afterSomeTask($event, $myComponent)
    {
        echo "Here, afterSomeTask\n";
    }

}
```

A listener is simply a class that implements any of all the events triggered by the component. Now let's make everything work together:

```
<?php

//Create an Events Manager
$eventsManager = new Phalcon\Events\Manager();

//Create the MyComponent instance
$myComponent = new MyComponent();

//Bind the eventsManager to the instance
$myComponent->setEventsManager($myComponent);

//Attach the listener to the EventsManager
$eventsManager->attach('my-component', new SomeListener());

//Execute methods in the component
$myComponent->someTask();
```

As "someTask" is executed, the two methods in the listener will be executed, producing the following output:

```
Here, beforeSomeTask
Here, afterSomeTask
```

Additional data may also passed when triggering an event using the third parameter of "fire":

```
<?php

$eventsManager->fire("my-component:afterSomeTask", $this, $extraData);
```

In a listener the third parameter also receives this data:

```
<?php

//Receiving the data in the third parameter
$eventManager->attach('my-component', function($event, $component, $data) {
    print_r($data);
});
```

```
//Receiving the data from the event context
$eventManager->attach('my-component', function($event, $component) {
    print_r($event->getData());
});
```

If a listener it is only interested in listening a specific type of event you can attach a listener directly:

```
<?php

//The handler will only be executed if the event triggered is "beforeSomeTask"
$eventManager->attach('my-component:beforeSomeTask', function($event, $component) {
    //...
});
```

2.22.3 Event Propagation/Cancelation

Many listeners may be added to the same event manager, this means that for the same type of event many listeners can be notified. The listeners are notified in the order they were registered in the EventsManager. Some events are cancelable, indicating that these may be stopped preventing other listeners are notified about the event:

```
<?php

$eventsManager->attach('db', function($event, $connection){

    //We stop the event if it is cancelable
    if ($event->isCancelable()) {
        //Stop the event, so other listeners will not be notified about this
        $event->stop();
    }

    //...

});
```

By default events are cancelable, even most of events produced by the framework are cancelables. You can fire a not-cancelable event by passing "false" in the fourth parameter of fire:

```
<?php

$eventsManager->fire("my-component:afterSomeTask", $this, $extraData, false);
```

2.22.4 Listener Priorities

When attaching listeners you can set a specifical priority. With this feature you can attach listeners indicating the order in which they must be called:

```
<?php

$evManager->attach('db', new DbListener(), 150); //More priority
$evManager->attach('db', new DbListener(), 100); //Normal priority
$evManager->attach('db', new DbListener(), 50); //Less priority
```

2.22.5 Implementing your own EventsManager

The *Phalcon\Events\ManagerInterface* interface must be implemented to create your own EventsManager replacing the one provided by Phalcon.

2.23 Request Environment

Every HTTP request (usually originated by a browser) contains additional information regarding the request such as header data, files, variables, etc. A web based application needs to parse that information so as to provide the correct response back to the requester. *Phalcon\HTTP\Request* encapsulates the information of the request, allowing you to access it in an object-oriented way.

```
<?php

// Getting a request instance
$request = new \Phalcon\Http\Request();

// Check whether the request was made with method POST
if ($request->isPost() == true) {
    // Check whether the request was made with Ajax
    if ($request->isAjax() == true) {
        echo "Request was made using POST and AJAX";
    }
}
```

2.23.1 Getting Values

PHP automatically fills the superglobal arrays $_GET and $_POST depending on the type of the request. These arrays contain the values present in forms submitted or the parameters sent via the URL. The variables in the arrays are never sanitized and can contain illegal characters or even malicious code, which can lead to SQL injection or Cross Site Scripting (XSS) attacks.

Phalcon\HTTP\Request allows you to access the values stored in the $_REQUEST, $_GET and $_POST arrays and sanitize or filter them with the 'filter' service, (by default *Phalcon\Filter*). The following examples offer the same behavior:

```
<?php

// Manually applying the filter
$filter = new Phalcon\Filter();

$email  = $filter->sanitize($_POST["user_email"], "email");

// Manually applying the filter to the value
$filter = new Phalcon\Filter();
$email  = $filter->sanitize($request->getPost("user_email"), "email");

// Automatically applying the filter
$email = $request->getPost("user_email", "email");

// Setting a default value if the param is null
$email = $request->getPost("user_email", "email", "some@example.com");

// Setting a default value if the param is null without filtering
$email = $request->getPost("user_email", null, "some@example.com");
```

2.23.2 Accessing the Request from Controllers

The most common place to access the request environment is in an action of a controller. To access the *Phalcon\HTTP\Request* object from a controller you will need to use the $this->request public property of the controller:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function saveAction()
    {

        // Check if request has made with POST
        if ($this->request->isPost() == true) {

            // Access POST data
            $customerName = $this->request->getPost("name");
            $customerBorn = $this->request->getPost("born");

        }

    }

}
```

2.23.3 Uploading Files

Another common task is file uploading. *Phalcon\HTTP\Request* offers an object-oriented way to achieve this task:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function uploadAction()
    {
        // Check if the user has uploaded files
        if ($this->request->hasFiles() == true) {
            // Print the real file names and sizes
            foreach ($this->request->getUploadedFiles() as $file) {

                //Print file details
                echo $file->getName(), " ", $file->getSize(), "\n";


                //Move the file into the application
                $file->moveTo('files/');
            }
        }
    }
```

```
}
```

Each object returned by Phalcon\Http\Request::getUploadedFiles() is an instance of the *Phalcon\Http\Request\File* class. Using the $_FILES superglobal array offers the same behavior. *Phalcon\Http\Request\File* encapsulates only the information related to each file uploaded with the request.

2.23.4 Working with Headers

As mentioned above, request headers contain useful information that allow us to send the proper response back to the user. The following examples show usages of that information:

```
<?php

// get the Http-X-Requested-With header
$requestedWith = $response->getHeader("X_REQUESTED_WITH");
if ($requestedWith == "XMLHttpRequest") {
    echo "The request was made with Ajax";
}

// Same as above
if ($request->isAjax()) {
    echo "The request was made with Ajax";
}

// Check the request layer
if ($request->isSecureRequest() == true) {
    echo "The request was made using a secure layer";
}

// Get the servers's ip address. ie. 192.168.0.100
$ipAddress = $request->getServerAddress();

// Get the client's ip address ie. 201.245.53.51
$ipAddress = $request->getClientAddress();

// Get the User Agent (HTTP_USER_AGENT)
$userAgent = $request->getUserAgent();

// Get the best acceptable content by the browser. ie text/xml
$contentType = $request->getAcceptableContent();

// Get the best charset accepted by the browser. ie. utf-8
$charset = $request->getBestCharset();

// Get the best language accepted configured in the browser. ie. en-us
$language = $request->getBestLanguage();
```

2.24 Returning Responses

Part of the HTTP cycle is return responses to the clients. *Phalcon\HTTP\Response* is the Phalcon component designed to achieve this task. HTTP responses are usually composed by headers and body. The basic usage is the following:

```
<?php

// Getting a response instance
```

```
$response = new \Phalcon\Http\Response();

//Set status code
$response->setRawHeader(404, "Not Found");

//Set the content of the response
$response->setContent("Sorry, the page doesn't exist");

//Send response to the client
$response->send();
```

Keep in mind that if you're using the full MVC stack there is no need to create responses manually. However, if you need to return a responde directly from a controller's action follow this example:

```
<?php

class FeedController extends Phalcon\Mvc\Controller
{

    public function getAction()
    {
        // Getting a response instance
        $response = new \Phalcon\Http\Request();

        $feed = //.. load here the feed

        //Set the content of the response
        $response->setContent($feed->asString());

        //Return the response
        return $response;
    }

}
```

2.24.1 Working with Headers

Headers are an important part of the whole HTTP response. It contains useful information about the response state like the HTTP status, type of response and much more.

You can set headers in the following way:

```
<?php

//Setting it by its name
$response->setHeader("Content-Type", "application/pdf");
$response->setHeader("Content-Disposition", 'attachment; filename="downloaded.pdf"');

//Setting a raw header
$response->setRawHeader("HTTP/1.1 200 OK");
```

A *Phalcon\HTTP\Response\Headers* bag internally manages headers. This class allows to manage headers before sending it to client:

```
<?php

//Get the headers bag
$headers = $response->getHeaders();
```

```
//Get a header by its name
$contentType = $response->getHeaders()->get("Content-Type");
```

2.24.2 Making Redirections

With *Phalcon\HTTP\Response* you can also make HTTP redirections:

```
<?php

//Making a redirection to the default URI
$response->redirect();

//Making a redirection using the local base URI
$response->redirect("posts/index");

//Making a redirection to an external URL
$response->redirect("http://en.wikipedia.org", true);

//Making a redirection specifyng the HTTP status code
$response->redirect("http://www.example.com/new-location", true, 301);
```

All internal URIs are generated using the 'url' service (by default *Phalcon\Mvc\Url*), in this way you can make redirections based on the routes you've currently defined in the application:

```
<?php

//Making a redirection based on a named route
$response->redirect(array(
    "for" => "index-lang",
    "lang" => "jp",
    "controller" => "index"
));
```

Note that making a redirection doesn't disable the view component, so if there is a view asociated with the current action it will be executed anyway. You can disable the view from a controller by executing $this->view->disable();

2.24.3 HTTP Cache

One of the easiest ways to improve the performance in your applications also reducing the traffic is the HTTP Cache. Most modern browsers support HTTP caching and is one of the reasons why many websites are currently fast.

The secret are the headers sent by the application when serving a page for the first time, these headers are:

- *Expires:* With this header the application can set a date in the future or the past telling the browser when the page must expire.
- *Cache-Control:* This header allows to specify how much time a page should be considered fresh in the browser.
- *Last-Modified:* This header tells the browser which was the last time the site was updated avoiding page re-loads
- *ETag:* An etag is a unique identifier that must be created including the modification timestamp of the current page

Setting an Expiration Time

The expiration date is one of the most easy and effective ways to cache a page in the client (browser). Starting from the current date we add over time, then, this will maintain the page stored in the browser cache until this date expires without requesting the content to the server again:

```
<?php

$expireDate = new DateTime();
$expireDate->modify('+2 months');

$response->setExpires($expireDate);
```

The Response component automatically shows the date in GMT timezone in order as is expected in an Expires header.

Moreover if we set a date in the past this will tell the browser to always refresh the requested page:

```
<?php

$expireDate = new DateTime();
$expireDate->modify('-10 minutes');

$response->setExpires($expireDate);
```

Browsers relies on the client's clock to assess if this date has passed or not, the client clock can be modified to make pages expire, this may represent a limitation for this cache mechanism.

Cache-Control

This header provides a safer way to cache the pages served. We simply must specify a time in seconds telling the browser how much time it must keep the page in its cache:

```
<?php

//Starting from now, cache the page for one day
$response->setHeader('Cache-Control', 'max-age=86400');
```

The opposite effect (avoid page caching) is achieved in this way:

```
<?php

//Never cache the served page
$response->setHeader('Cache-Control', 'private, max-age=0, must-revalidate');
```

E-Tag

A "entity-tag" or "E-tag" is a unique identifier that helps the browser to realize if the page has changed or not between two requests. The identifier must be calculated taking into account that this must change if the content has changed previously served:

```
<?php

//Calculate the E-Tag based on the modification time of the latest news
$recentDate = News::maximum(array('column' => 'created_at'));
$eTag = md5($recentDate);
```

```
//Send a E-Tag header
$response->setHeader('E-Tag', $eTag);
```

2.25 Generating URLs and Paths

Phalcon\Mvc\Url is the component responsible of generate urls in a Phalcon application. It's capable of produce independent urls based on routes.

2.25.1 Setting a base URI

Dependending of which directory of your document root your application is installed, it may have a base uri or not.

For example, if your document root is /var/www/htdocs and your application is installed in /var/www/htdocs/invo then your baseUri will be /invo/. If you are using a VirtualHost or your application is installed on the document root, then your baseUri is /. Execute the following code to know the base uri detected by Phalcon:

```
<?php

$url = new Phalcon\Mvc\Url();
echo $url->getBaseUri();
```

By default, Phalcon automatically may detect your baseUri, but if you want to increase the performance of your application is recommended setting up it manually:

```
<?php

$url = new Phalcon\Mvc\Url();

$url->setBaseUri('/invo/');
```

Usually, this component must be registered in the Dependency Injector container, so you can set up it there:

```
<?php

$di->set('url', function(){
    $url = new Phalcon\Mvc\Url();
    $url->setBaseUri('/invo/');
    return $url;
});
```

2.25.2 Generating URIs

If you are using the *Router* with its default behavior. Your application is able to match routes based on the following pattern: /:controller/:action/:params. Accordingly it is easy to create routes that satisfy that pattern (or any other pattern defined in the router) passing a string to the method "get":

```
<?php echo $url->get("products/save") ?>
```

Note that isn't necessary to prepend the base uri. If you have named routes you can easily change it creating it dynamically. For Example if you have the following route:

```
<?php

$route->add('/blog/{$year}/{month}/{title}', array(
```

```
    'controller' => 'posts',
    'action' => 'show'
))->setName('show-post');
```

A URL can be generated in the following way:

```
<?php

//This produces: /blog/2012/01/some-blog-post
$url->get(array(
    'for' => 'show-post',
    'year' => 2012,
    'month' => '01',
    'title' => 'some-blog-post'
));
```

2.25.3 Producing URLs without Mod-Rewrite

You can use this component also to create urls without mod-rewrite:

```
<?php

$url = new Phalcon\Mvc\Url();

//Pass the URI in $_GET["_url"]
$url->setBaseUri('/invo/index.php?_url=/');

//This produce: /invo/index.php?_url=/products/save
echo $url->get("products/save");
```

You can also use $_SERVER[”REQUEST_URI”]:

```
<?php

$url = new Phalcon\Mvc\Url();

//Pass the URI using $_SERVER["REQUEST_URI"]
$url->setBaseUri('/invo/index.php?_url=/');

//Pass the URI in $_GET["_url"]
$url->setBaseUri('/invo/index.php/');
```

In this case, it’s necessary to manually handle the required URI in the Router:

```
<?php

$router = new Phalcon\Mvc\Router();

// ... define routes

$uri = str_replace($_SERVER["SCRIPT_NAME"], '', $_SERVER["REQUEST_URI"]);
$router->handle($uri);
```

The produced routes would look like:

```
<?php
```

```
//This produce: /invo/index.php/products/save
echo $url->get("products/save");
```

2.25.4 Producing URLs from Volt

The function "url" is available in volt to generate URLs using this component:

```
<a href="{{ url("posts/edit/1002") }}">Edit</a>
```

2.25.5 Implementing your own Url Generator

The *Phalcon\Mvc\UrlInterface* interface must be implemented to create your own URL generator replacing the one provided by Phalcon.

2.26 Flashing Messages

Flash messages are used to notify the user about the state of actions he/she made or simply show information to the users. These kind of messages can be generated using this component.

2.26.1 Adapters

This component makes use of adapters to define the behavior of the messages after being passed to the Flasher:

Adapter	Description	API
Direct	Directly outputs the messages passed to the flasher	*Phalcon\Flash\Direct*
Session	Temporarily stores the messages in session, then messages can be printed in the next request	*Phalcon\Flash\Session*

2.26.2 Usage

Usually the Flash Messaging service is requested from the services container, if you're using *Phalcon\DI\FactoryDefault* then *Phalcon\Flash\Direct* is automatically registered as "flash" service:

```
<?php

//Set up the flash service
$di->set('flash', function() {
    return new \Phalcon\Flash\Direct();
});
```

This way, you can use it in controllers or views by injecting the service in the required scope:

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {
```

```
    }

    public function saveAction()
    {
        $this->flash->success("The post was correctly saved!");
    }

}
```

There are four built-in message types supported:

```
<?php

$this->flash->error("too bad! the form had errors");
$this->flash->success("yes!, everything went very smoothly");
$this->flash->notice("this a very important information");
$this->flash->warning("best check yo self, you're not looking too good.");
```

You can add messages with your own types:

```
<?php

$this->flash->message("debug", "this is debug message, you don't say");
```

2.26.3 Printing Messages

Messages sent to the flasher are automatically formatted with html:

```
<div class="errorMessage">too bad! the form had errors</div>
<div class="successMessage">yes!, everything went very smoothly</div>
<div class="noticeMessage">this a very important information</div>
<div class="warningMessage">best check yo self, you're not looking too good.</div>
```

As can be seen, also some CSS classes are added automatically to the DIVs. These classes allow you to define the graphical presentation of the messages in the browser. The CSS classes can be overridden, for example, if you're using Twitter bootstrap, classes can be configured as:

```
<?php

//Register the flash service with custom CSS classes
$di->set('flash', function(){
    $flash = new \Phalcon\Flash\Direct(array(
        'error' => 'alert alert-error',
        'success' => 'alert alert-success',
        'notice' => 'alert alert-info',
    ));
    return $flash;
});
```

Then the messages would be printed as follows:

```
<div class="alert alert-error">too bad! the form had errors</div>
<div class="alert alert-success">yes!, everything went very smoothly</div>
<div class="alert alert-info">this a very important information</div>
```

2.26.4 Implicit Flush vs. Session

Depending on the adapter used to send the messages, it could be producing output directly, or be temporarily storing the messages in session to be shown later. When should you use each? That usually depends on the type of redirection you do after sending the messages. For example, if you make a "forward" is not necessary to store the messages in session, but if you do a HTTP redirect then, they need to be stored in session:

```
<?php

class ContactController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function saveAction()
    {

        //store the post

        //Using direct flash
        $this->flash->success("Your information were stored correctly!");

        //Forward to the index action
        return $this->dispatcher->forward(array("action" => "index"));
    }

}
```

Or using a HTTP redirection:

```
<?php

class ContactController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function saveAction()
    {

        //store the post

        //Using session flash
        $this->flashSession->success("Your information were stored correctly!");

        //Make a full HTTP redirection
        return $this->response->redirect("contact/index");
    }

}
```

In this case you need to print manually the messages in the corresponding view:

```
<!-- app/views/contact/index.phtml -->

<p><?php $this->flashSession->output() ?></p>
```

The attribute 'flashSession' is how the flash was previously set into the dependency injection container. You need to start the *session* first to successfully use the flashSession messenger.

2.27 Storing data in Session

The *Phalcon\Session* provides object-oriented wrappers to access session data.

2.27.1 Starting the Session

Some applications are session-intensive, almost any action that performs requires access to session data. There are others who access session data casually. Thanks to the service container, we can ensure that the session is accessed only when it's clearly needed:

```
<?php

//Start the session the first time when some component request the session service
$di->setShared('session', function() {
    $session = new Phalcon\Session\Adapter\Files();
    $session->start();
    return $session;
});
```

2.27.2 Storing/Retrieving data in Session

From a controller, a view or any other component that extends *Phalcon\DI\Injectable* you can access the session service and store items and retrieve them in the following way:

```
<?php

class UserController extends Phalcon\Mvc\Controller
{

    public function indexAction()
    {
        //Set a session variable
        $this->session->set("user-name", "Michael");
    }

    public function welcomeAction()
    {

        //Check if the variable is defined
        if ($this->session->has("user-name")) {

            //Retrieve its value
            $name = $this->session->get("user-name");
        }
    }
```

```
}
```

2.27.3 Removing/Destroying Sessions

It's also possible remove specific variables or destroy the whole session:

```
<?php

class UserController extends Phalcon\Mvc\Controller
{

    public function removeAction()
    {
        //Remove a session variable
        $this->session->remove("user-name");
    }

    public function logoutAction()
    {
        //Destroy the whole session
        $this->session->destroy();
    }

}
```

2.27.4 Isolating Session Data between Applications

Sometimes a user can use the same application twice, on the same server, in the same session. Surely, if we use variables in session, we want that every application have separate session data (even though the same code and same variable names). To solve this, you can add a prefix for every session variable created in a certain application:

```
<?php

//Isolating the session data
$di->set('session', function(){

    //All variables created will prefixed with "my-app-1"
    $session = new Phalcon\Session\Adapter\Files(
        array(
            'uniqueId' => 'my-app-1'
        )
    );

    $session->start();

    return $session;
});
```

2.27.5 Session Bags

Phalcon\Session\Bag is a component helps that helps separing session data into "namespaces". Working by this way you can easily create groups of session variables into the application. By only setting the variables in the "bag", it's automatically stored in session:

```
<?php

$user       = new Phalcon\Session\Bag('user');
$user->setDI($di);
$user->name = "Kimbra Johnson";
$user->age  = 22;
```

2.27.6 Persistent Data in Components

Controller, components and classes thats extends *Phalcon\DI\Injectable* may inject a *Phalcon\Session\Bag*. This class isolates variables for every class. Thanks to this you can persist data between requests in every class in an independent way.

```
<?php

class UserController extends Phalcon\Mvc\Controller
{

    public function indexAction()
    {
        // Create a persistent variable "name"
        $this->persistent->name = "Laura";
    }

    public function welcomeAction()
    {
        if (isset($this->persistent->name))
        {
            echo "Welcome, ", $this->persistent->name;
        }
    }

}
```

In a component:

```
<?php

class Security extends Phalcon\Mvc\User\Component
{

    public function auth()
    {
        // Create a persistent variable "name"
        $this->persistent->name = "Laura";
    }

    public function getAuthName()
    {
        return $this->persistent->name;
    }

}
```

The data added to the session ($this->session) are available throughout the application, while persistent ($this->persistent) can only be accessed in the scope of the current class.

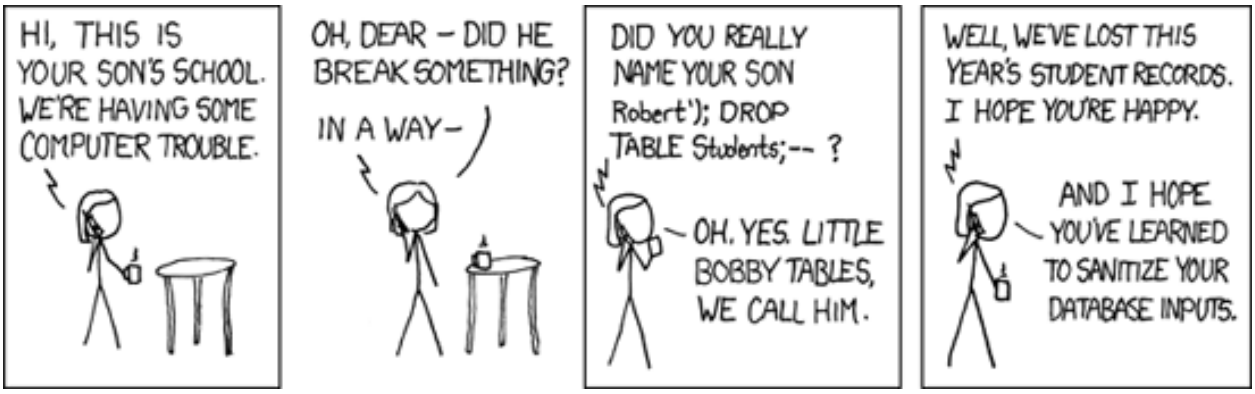
2.27.7 Implementing your own adapters

The *Phalcon\Session\AdapterInterface* interface must be implemented in order to create your own session adapters or extend the existing ones.

There are more adapters available for this components in the Phalcon Incubator

2.28 Filtering and Sanitizing

Sanitizing user input is a critical part of software development. Trusting or neglecting to sanitize user input could lead to unauthorized access to the content of your application, mainly user data, or even the server your application is hosted.



Full image (from xkcd)

The *Phalcon\Filter* component provides a set of commonly used filters and data sanitizing helpers. It provides object-oriented wrappers around the PHP filter extension.

2.28.1 Sanitizing data

Sanitizing is the process which removes specific characters from a value, that are not required or desired by the user or application. By sanitizing input we ensure that application integrity will be intact.

```
<?php

$filter = new \Phalcon\Filter();

// returns "someone@example.com"
$filter->sanitize("some(one)@exa\mple.com", "email");

// returns "hello"
$filter->sanitize("hello<<", "string");

// returns "100019"
$filter->sanitize("!100a019", "int");

// returns "100019.01"
$filter->sanitize("!100a019.01a", "float");
```

2.28.2 Sanitizing from Controllers

You can access a *Phalcon\Filter* object from your controllers when accessing GET or POST input data (through the request object). The first parameter is the name of the variable to be obtained; the second is the filter to be applied on it.

```
<?php

class ProductsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function saveAction()
    {

        // Sanitizing price from input
        $price = $this->request->getPost("price", "double");

        // Sanitizing email from input
        $email = $this->request->getPost("customerEmail", "email");

    }

}
```

2.28.3 Filtering Action Parameters

The next example shows you how to sanitize the action parameters within a controller action:

```
<?php

class ProductsController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function showAction($productId)
    {
        $productId = $this->filter->sanitize($productId, "int");
    }

}
```

2.28.4 Filtering data

In addition to sanitizing, *Phalcon\Filter* also provides filtering by removing or modifying input data to the format we expect.

```
<?php

$filter = new \Phalcon\Filter();

// returns "Hello"
$filter->filter("<h1>Hello</h1>", "striptags");

// returns "Hello"
$filter->filter("  Hello   ", "trim");
```

2.28.5 Types of Built-in Filters

The following are the built-in filters provided by this component:

Name	Description
string	Strip tags
email	Remove all characters except letters, digits and !#$%&*+-/=?^_‘{|}~@.[].
int	Remove all characters except digits, plus and minus sign.
float	Remove all characters except digits, dot, plus and minus sign.
alphanum	Remove all characters except [a-zA-Z0-9]
striptags	Applies the strip_tags function
trim	Applies the trim function
lower	Applies the strtolower function
upper	Applies the strtoupper function

2.28.6 Creating your own Filters

You can add your own filters to *Phalcon\Filter*. The filter function could be an anonomyous function:

```
<?php

$filter = new \Phalcon\Filter();

//Using an anonymous function
$filter->add('md5', function($value) {
    return preg_replace('/[^0-9a-f]/', '', $value);
});

//Sanitize with the "md5" filter
$filtered = $filter->sanitize($possibleMd5, "md5");
```

Or, if you prefer, you can implement the filter in a class:

```
<?php

class IPv4Filter
{

    public function filter($value)
    {
        return filter_var($value, FILTER_VALIDATE_IP, FILTER_FLAG_IPV4);
    }

}
```

```
$filter = new \Phalcon\Filter();

//Using an object
$filter->add('ipv4', new IPv4Filter());

//Sanitize with the "ipv4" filter
$filteredIp = $filter->sanitize("127.0.0.1", "ipv4");
```

2.28.7 Complex Sanitizing and Filtering

PHP itself provides an excellent filter extension you can use. Check out its documentation: Data Filtering at PHP Documentation

2.28.8 Implementing your own Filter

The *Phalcon\FilterInterface* interface must be implemented to create your own filtering service replacing the one provided by Phalcon.

2.29 Contextual Escaping

Websites and Web applications are vulnerable to XSS attacks, despite PHP provides escaping functionality, in some contexts those are not sufficient/appropriate. *Phalcon\Escaper* provides contextual escaping, this component is written in C providing the minimal overhead when escaping different kinds of texts.

We designed this component based on the XSS (Cross Site Scripting) Prevention Cheat Sheet created by the OWASP

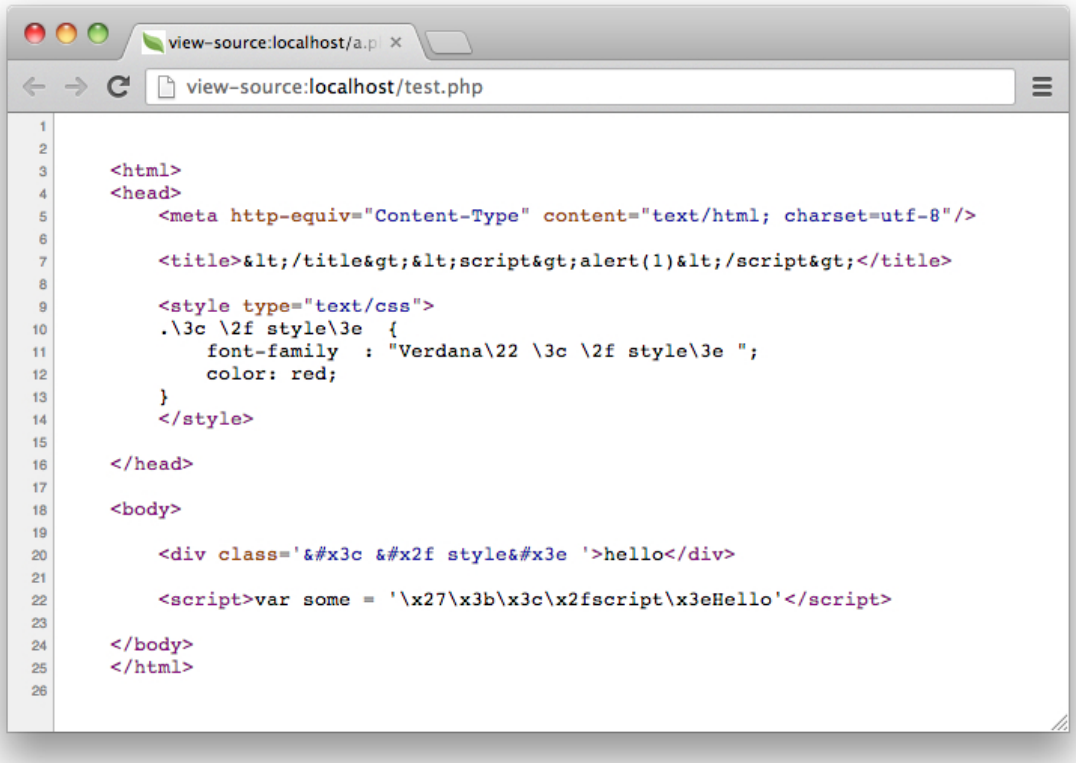
Additionally, this component relies on mbstring to support almost any charset.

To illustrate how this component works and why it is important, consider the following example:

```
<?php

    //Document title with malicious extra HTML tags
    $maliciousTitle = '</title><script>alert(1)</script>';

    //Malicious CSS class name
    $className = ';`(';

    //Malicious CSS font name
    $fontName = 'Verdana"</style>';

    //Malicious Javascript text
    $javascriptText = "';</script>Hello";

    //Create a escaper
    $e = new Phalcon\Escaper();

?>

<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>

    <title><?php echo $e->escapeHtml($maliciousTitle) ?></title>
```

```
    <style type="text/css">
    .<?php echo $e->escapeCss($className) ?> {
        font-family  : "<?php echo $e->escapeCss($fontName) ?>";
        color: red;
    }
    </style>

</head>

<body>

    <div class='<?php echo $e->escapeHtmlAttr($className) ?>'>hello</div>

    <script>var some = '<?php echo $e->escapeJs($javascriptText) ?>'</script>

</body>
</html>
```

Which produces the following:



Every text was escaped according to its context. Use the appropriate context is important to avoid XSS attacks.

2.29.1 Escaping HTML

The most common situation when inserting unsafe data is between HTML tags:

```
<div class="comments"><!-- Escape unstrusted data here! --></div>
```

You can escape those data using the escapeHtml method:

```
<div class="comments"><?php echo $e->escapeHtml('></div><h1>myattack</h1>'); ?></div>
```

Which produces:

```
<div class="comments">&gt;&lt;/div&gt;&lt;h1&gt;myattack&lt;/h1&gt;</div>
```

2.29.2 Escaping HTML Attributes

Escape HTML attributes is different from escape a full HTML content. The escape works by changing every non-alphanumeric character to the form. This kind of escaping is intended to most simpler attributes excluding complex ones like 'href' or 'url':

```
<table width="Escape unstrusted data here!"><tr><td>Hello</td></tr></table>
```

You can escape an HTML attribute by using the escapeHtmlAttr method:

```
<table width="<?php echo $e->escapeHtmlAttr('"><h1>Hello</table'); ?>"><tr><td>Hello</td></tr></table
```

Which produces:

```
<table width="&#x22;&#x3e;&#x3c;h1&#x3e;Hello&#x3c;&#x2f;table"><tr><td>Hello</td></tr></table>
```

2.29.3 Escaping URLs

Some HTML attributes like 'href' or 'url' need to be escaped differently:

```
<a href="Escape unstrusted data here!">Some link</a>
```

You can escape an HTML attribute by using the escapeUrl method:

```
<a href="<?php echo $e->escapeUrl('"><script>alert(1)</script><a href="#'); ?>">Some link</a>
```

Which produces:

```
<a href="%22%3E%3Cscript%3Ealert%281%29%3C%2Fscript%3E%3Ca%20href%3D%22%23">Some link</a>
```

2.29.4 Escaping CSS

CSS identifiers/values can be escaped too:

```
<a style="color: Escape unstrusted data here">Some link</a>
```

You can escape an HTML attribute by using the escapeCss method:

```
<a style="color: <?php echo $e->escapeCss('"><script>alert(1)</script><a href="#'); ?>">Some link</a
```

Which produces:

```
<a style="color: \22 \3e \3c script\3e alert\28 1\29 \3c \2f script\3e \3c a\20 href\3d \22 \23 ">Son
```

2.29.5 Escaping Javascript

Strings to be inserted into javascript code also must be properly escaped:

```
<script>document.title = 'Escape unstrusted data here'</script>
```

You can escape an HTML attribute by using the escapeJs method:

```
<script>document.title = '<?php echo $e->escapejs("'; alert(100); var x='"); ?>'</script>

<script>alert('\x27\x3b\x20alert\x28100\x29\x3b\x20var\x20x\x3d\x27')</script>
```

2.30 Validation

PhalconValidation is an independent validation component to validate an arbitrary set of data. This component can be used to implement validation rules that does not belong to a model or collection.

The following example shows its basic usage:

```
use Phalcon\Validation\Validator\PresenceOf,
        Phalcon\Validation\Validator\Email;

$validation = new Phalcon\Validation();

$validation->add('name', new PresenceOf(
        'message' => 'The name is required'
));

$validation->add('email', new PresenceOf(
        'message' => 'The e-mail is required'
));

$validation->add('email', new Email(
        'message' => 'The e-mail is not valid'
));

$messages = $validation->validate($_POST);
if (count($messages)) {
        foreach ($messages as $message) {
                echo $message, '<br>';
        }
}
```

2.30.1 Validators

Phalcon exposes a set of built-in validators for this component:

Name	Explanation	Example
PresenceOf	Validates that a field's value isn't null or empty string.	*Example*
Email	Validates that field contains a valid email format	*Example*
ExclusionIn	Validates that a value is not within a list of possible values	`Example`
InclusionIn	Validates that a value is within a list of possible values	`Example`
Regex	Validates that the value of a field matches a regular expression	*Example*
StringLength	Validates the length of a string	*Example*

Additional validators can be created by the developer. The following class explains how to create a validator for this component:

```
use Phalcon\Validation\Validator,
        Phalcon\Validation\ValidatorInterface,
        Phalcon\Validation\Message;

class IpValidator extends Validator implements ValidatorInterface
{

        /**
         * Executes the validation
         *
         * @param Phalcon\Validation $validator
         * @param string $attribute
         * @return boolean
         */
        public function validate($validator, $attribute)
        {
                $value = $validator->getValue($attribute);

                if (filter_var($value, FILTER_VALIDATE_URL, FILTER_FLAG_PATH_REQUIRED)) {

                        $message = $this->getOption('message');
                        if (!$message) {
                                $message = 'The IP is not valid';
                        }

                        $validator->appendMessage(new Message($message, $attribute, 'Ip'));

                        return false;
                }

                return true;
        }

}
```

2.30.2 Validation Messages

Phalcon\Validation has a messaging subsystem that provides a flexible way to output or store the validation messages generated during the validation processes.

Each message consists of an instance of the class *Phalcon\Validation\Message*. The set of messages generated can be retrieved with the method getMessages(). Each message provides extended information like the attribute that generated the message or the message type:

```
<?php

$messages = $validation->validate();
if (count($messages)) {
    foreach ($validation->getMessages() as $message) {
        echo "Message: ", $message->getMessage(), "\n";
        echo "Field: ", $message->getField(), "\n";
        echo "Type: ", $message->getType(), "\n";
    }
}
```

The method getMessages() can be overriden in a validation class to replace/translate the default messages generated automatically by the validators:

```
<?php

class MyValidation extends Phalcon\Validation
{
    public function getMessages()
    {
        $messages = array();
        foreach (parent::getMessages() as $message) {
            switch ($message->getType()) {
                case 'PresenceOf':
                    $messages[] = 'The field ' . $message->getField() . ' is mandatory';
                    break;
            }
        }
        return $messages;
    }
}
```

Or you can pass a parameter 'message' to change the default message in each validator:

```
$validation->add('email', new Phalcon\Validation\Validator\Email(
        'message' => 'The e-mail is not valid'
));
```

2.31 Forms

Phalcon\Forms is a component aids the developer in the creation and maintenance of forms in web applications.

The following example shows its basic usage:

```
<?php

use Phalcon\Forms\Form,
        Phalcon\Forms\Element\Text,
        Phalcon\Forms\Element\Select;

$form = new Form();

$form->add(new Text("name"));

$form->add(new Text("telephone"));

$form->add(new Select("telephoneType", array(
        'H' => 'Home',
        'C' => 'Cell'
)));
```

Forms can be rendered based on the form definition:

```
<h1>Contacts</h1>

<form method="post">

        <p>
                <label>Name</label>
```

```
                <?php echo $form->render("name") ?>
        </p>

        <p>
                <label>Telephone</label>
                <?php echo $form->render("telephone") ?>
        </p>

        <p>
                <label>Type</label>
                <?php echo $form->render("telephoneType") ?>
        </p>

        <p>
                <input type="submit" value="Save" />
        </p>

</form>
```

Each element in the form can be rendered as required by the developer. Internally, *Phalcon\Tag* is used to produce the right HTML for each element, you can pass additional html attributes as second parameter for render:

```
<p>
        <label>Name</label>
        <?php echo $form->render("name", array('maxlength' => 30, 'placeholder' => 'Type your name'))
</p>
```

HTML Attributes also can be set in the element's definition:

```
<?php

$form->add(new Text("name", array(
        'maxlength' => 30,
        'placeholder' => 'Type your name'
)));
```

2.31.1 Initializing forms

As seen before, forms can be initialized outside the form class by adding elements to it. You can re-use code or organize your form classes implementing the form in a separated file:

```
<?php

use Phalcon\Forms\Form,
        Phalcon\Forms\Element\Text,
        Phalcon\Forms\Element\Select;

class ContactsForm extends Form
{
        public function initialize()
        {
                $this->add(new Text("name"));

                $this->add(new Text("telephone"));

                $this->add(new Select("telephoneType", TelephoneTypes::find(), array(
                        'using' => array('id', 'name')
```

```
                ));
        }
}
```

2.31.2 Validation

Phalcon forms are integrated with the *validation* component to offer instant validation. Built-in or custom validators could be set to each element:

```
<?php

use Phalcon\Forms\Element\Text,
        Phalcon\Validation\Validator\PresenceOf,
        Phalcon\Validation\Validator\StringLength;

$name = new Text("name");

$name->addValidator(new PresenceOf(array(
        'message' => 'The name is required'
)));

$name->addValidator(new StringLength(array(
        'min' => 10,
        'messageMinimum' => 'The name is too short'
)));

$form->add($name);
```

Then you can validate the form according to the input entered by the user:

```
<?php

if (!$form->isValid($_POST)) {
        foreach ($form->getMessages() as $message) {
                echo $message, '<br>';
        }
}
```

Validators are executed in the same order as they were registered.

By default messages generated by all the elements in the form are joined so they can be traversed using a single foreach, you can change this behavior to get the messages separated by the field:

```
<?php

foreach ($form->getMessages(false) as $attribute => $messages) {
        echo 'Messages generated by ', $attribute, ':', "\n";
        foreach ($messages as $message) {
                echo $message, '<br>;
        }
}
```

Or get specific messages for an element:

```
<?php

foreach ($form->getMessagesFor('name') as $message) {
```

```
        echo $message, '<br>';
}
```

2.31.3 Forms + Entities

An entity such as a model/collection instance or just a plain PHP class can be linked to the form in order to set default values in the form's elements or assign the values from the form to the entity easily:

```
<?php

$robot = Robots::findFirst();

$form = new Form($robot);

$form->add(new Text("name"));

$form->add(new Text("year"));
```

Once the form is rendered if there is no default values assigned to the elements it will use the ones provided by the entiy:

```
<?php echo $form->render('name') ?>
```

You can validate the form and assign the values from the user input in the following way:

```
<?php

$form->bind($_POST, $robot);

//Check if the form is valid
if ($form->isValid()) {

        //Save the entity
        $robot->save();
}
```

2.31.4 Form Elements

Phalcon provides a set of built-in elements to use in your forms:

Name	Description	Example
Text	Generate INPUT[type=text] elements	*Example*
Password	Generate INPUT[type=password] elements	*Example*
Select	Generate SELECT tag (combo lists) elements based on choices	*Example*
Check	Generate INPUT[type=check] elements	*Example*
Textarea	Generate TEXTAREA elements	*Example*

2.31.5 Rendering Forms

You can render the form with total flexibility, the following example shows how to render each element using an standard procedure:

```
<?php

<form method="post">
        <?php
                //Traverse the form
                foreach ($form as $element) {

                        //Get any generated messages for the current element
                        $messages = $form->getMessagesFor($element->getName());

                        if (count($messages)) {
                                //Print each element
                                echo '<div class="messages">';
                                foreach ($messages as $message) {
                                        echo $message;
                                }
                                echo '</div>';
                        }

                        echo '<p>';
                        echo '<label for="', $element->getName(), '">', $element->getLabel(), '</labe
                        echo $element;
                        echo '</p>';

                }
        ?>
        <input type="submit" value="Send"/>
</form>
```

Or reuse the logic in your form class:

```
<?php

class ContactForm extends Phalcon\Forms\Form
{
        public function initialize()
        {
                //...
        }

        public function renderDecorated($name)
        {
                $element = $this->get($name);

                //Get any generated messages for the current element
                $messages = $this->getMessagesFor($element->getName());

                if (count($messages)) {
                        //Print each element
                        echo '<div class="messages">';
                        foreach ($messages as $message) {
                                echo $message;
                        }
                        echo '</div>';
                }

                echo '<p>';
                echo '<label for="', $element->getName(), '">', $element->getLabel(), '</label>';
                echo $element;
```

```
                echo '</p>';
        }

}
```

2.31.6 Creating Form Elements

In addition to the form elements provided by Phalcon you can create your own custom elements:

```
<?php

class MyElement extends Phalcon\Forms\Element
{
        public function render($attributes=null)
        {
                $html = //... produce some html
                return $html;
        }
}
```

2.32 Reading Configurations

Phalcon\Config is a component used to read configuration files of various formats (using adapters) into PHP objects for use in an application.

2.32.1 File Adapters

The adapters available are:

File Type	Description
Ini	Uses INI files to store settings. Internally the adapter uses the PHP function parse_ini_file.
Array	Uses PHP multidimensional arrays to store settings. This adapter offers the best performance.

2.32.2 Native Arrays

The next example shows how to convert native arrays into Phalcon\Config objects. This option offers the best performance since no files are read during this request.

```
<?php

$settings = array(
    "database" => array(
        "adapter"  => "Mysql",
        "host"     => "localhost",
        "username" => "scott",
        "password" => "cheetah",
        "name"     => "test_db",
    ),
     "app" => array(
        "controllersDir" => "../app/controllers/",
        "modelsDir"      => "../app/models/",
        "viewsDir"       => "../app/views/",
```

```
    ),
    "mysetting" => "the-value"
);

$config = new \Phalcon\Config($settings);

echo $config->app->controllersDir, "\n";
echo $config->database->username, "\n";
echo $config->mysetting, "\n";
```

If you want to better organize your project you can save the array in another file and then read it.

```
<?php

require "config/config.php";
$config = new \Phalcon\Config($settings);
```

2.32.3 Reading INI Files

Ini files are a common way to store settings. Phalcon\Config uses the optimized PHP function parse_ini_file to read these files. Files sections are parsed into sub-settings for easy access.

```
[database]
adapter  = Mysql
host     = localhost
username = scott
password = cheetah
name     = test_db

[phalcon]
controllersDir = "../app/controllers/"
modelsDir      = "../app/models/"
viewsDir       = "../app/views/"

[models]
metadata.adapter  = "Memory"
```

You can read the file as follows:

```
<?php

$config = new \Phalcon\Config\Adapter\Ini("path/config.ini");

echo $config->phalcon->controllersDir, "\n";
echo $config->database->username, "\n";
echo $config->models->metadata->adapter, "\n";
```

2.32.4 Merging Configurations

Phalcon\Config allows to merge a configuration object into another one recursively:

```
<?php

$config = new \Phalcon\Config(array(
    'database' => array(
        'host' => 'localhost',
```

```
        'name' => 'test_db'
    ),
    'debug' => 1
));

$config2 = new \Phalcon\Config(array(
    'database' => array(
        'username' => 'scott',
        'password' => 'secret',
    )
));

$config->merge($config2);

print_r($config);
```

The above code produces the following:

```
Phalcon\Config Object
(
    [database] => Phalcon\Config Object
        (
            [host] => localhost
            [name] => test_db
            [username] => scott
            [password] => secret
        )
    [debug] => 1
)
```

2.33 Data Pagination

The process of pagination takes place when we need to present big groups of arbitrary data gradually. Phalcon\Paginator offers a fast and convenient way to split these sets of data browsable pages.

2.33.1 Data Adapters

This component makes use of adapters to encapsulate different sources of data:

Adapter	Description
NativeArray	Use a PHP array as source data
Model	Use a Phalcon\Mvc\Model\Resultset object as source data

2.33.2 Using Paginators

In the example below, the paginator will use as its source data the result of a query from a model, and limit the displayed data to 10 records per page:

```
<?php

// Current page to show
// In a controller this can be:
// $this->request->getQuery('page', 'int'); // GET
```

```
// $this->request->getPost('page', 'int'); // POST
$currentPage = (int) $_GET["page"];

// The data set to paginate
$robots = Robots::find();

// Create a Model paginator, show 10 rows by page starting from $currentPage
$paginator = new \Phalcon\Paginator\Adapter\Model(
    array(
        "data" => $robots,
        "limit"=> 10,
        "page" => $currentPage
    )
);

// Get the paginated results
$page = $paginator->getPaginate();
```

Variable $currentPage controls the page to be displayed. The $paginator->getPaginate() returns a $page object that contains the paginated data. It can be used for generating the pagination:

```
<table>
    <tr>
        <th>Id</th>
        <th>Name</th>
        <th>Type</th>
    </tr>
    <?php foreach ($page->items as $item) { ?>
    <tr>
        <td><?php echo $item->id; ?></td>
        <td><?php echo $item->name; ?></td>
        <td><?php echo $item->type; ?></td>
    </tr>
    <?php } ?>
</table>
```

The $page object also contains navigation data:

```
<a href="/robots/search">First</a>
<a href="/robots/search?page=<?= $page->before; ?>">Previous</a>
<a href="/robots/search?page=<?= $page->next; ?>">Next</a>
<a href="/robots/search?page=<?= $page->last; ?>">Last</a>

<?php echo "You are in page ", $page->current, " of ", $page->total_pages; ?>
```

2.33.3 Page Attributes

The $page object has the following attributes:

Adapter	Description
items	The set of records to be displayed at the current page
before	The previous page to the current one
next	The next page to the current one
last	The last page in the set of records

2.33.4 Implementing your own adapters

The *Phalcon\Paginator\AdapterInterface* interface must be implemented in order to create your own paginator adapters or extend the existing ones:

```
<?php

class MyPaginator implements Phalcon\Paginator\AdapterInterface  {

    /**
     * Adapter constructor
     *
     * @param array $config
     */
    public function __construct($config);

    /**
     * Set the current page number
     *
     * @param int $page
     */
    public function setCurrentPage($page);

    /**
     * Returns a slice of the resultset to show in the pagination
     *
     * @return stdClass
     */
    public function getPaginate();

}
```

2.34 Improving Performance with Cache

Phalcon provides the *Phalcon\Cache* class allowing faster access to frequently used or already processed data. *Phalcon\Cache* is written in C, achieving higher performance and reducing the overhead when getting items from the backends. This class uses an internal structure of frontend and backend components. Front-end components act as input sources or interfaces, while backend components offer storage options to the class.

2.34.1 When to implement cache?

Although this component is very fast, implementing it in cases that is not needed could lead to loss of performance than gain. We recommend you check this cases before using a cache:

- You are making complex calculations that every time return the same result (changing infrequently)
- You are using a lot of helpers and the output generated is almost always the same
- You are accessing database data constantly and these data rarely change

  NOTE Even after implementing the cache, you should check the hit ratio of your cache over a period of time. This can easily be done, especially in the case of Memcache or Apc, with the relevant tools that the backends provide.

2.34.2 Caching Behavior

The caching process is divided into 2 parts:

- **Frontend**: This part is responsible for checking if a key has expired and perform additional transformations to the data before storing and after retrieving them from the backend-
- **Backend**: This part is responsible for communicating, writing/reading the data required by the frontend.

2.34.3 Caching Output Fragments

An output fragment is a piece of HTML or text that is cached as is and returned as is. The output is automatically captured from the ob_* functions or the PHP output so that it can be saved in the cache. The following example demonstrates such usage. It receives the output generated by PHP and stores it into a file. The contents of the file are refreshed every 172800 seconds (2 days).

The implementation of this caching mechanism allows us to gain performance by not executing the helper Phalcon\Tag::linkTo call whenever this piece of code is called.

```
<?php

//Create an Output frontend. Cache the files for 2 days
$frontCache = new Phalcon\Cache\Frontend\Output(array(
    "lifetime" => 172800
));

// Create the component that will cache from the "Output" to a "File" backend
// Set the cache file directory - it's important to keep the "/" at the end of
// the value for the folder
$cache = new Phalcon\Cache\Backend\File($frontCache, array(
    "cacheDir" => "../app/cache/"
));

// Get/Set the cache file to ../app/cache/my-cache.html
$content = $cache->start("my-cache.html");

// If $content is null then the content will be generated for the cache
if ($content === null) {

    //Print date and time
    echo date("r");

    //Generate a link to the sign-up action
    echo Phalcon\Tag::linkTo(
        array(
            "user/signup",
            "Sign Up",
            "class" => "signup-button"
        )
    );

    // Store the output into the cache file
    $cache->save();

} else {

    // Echo the cached output
```

```
    echo $content;
}
```

NOTE In the example above, our code remains the same, echoing output to the user as it has been doing before. Our cache component transparently captures that output and stores it in the cache file (when the cache is generated) or it sends it back to the user pre-compiled from a previous call, thus avoiding expensive operations.

2.34.4 Caching Arbitrary Data

Caching just data is equally important for your application. Caching can reduce database load by reusing commonly used (but not updated) data, thus speeding up your application.

File Backend Example

One of the caching adapters is 'File'. The only key area for this adapter is the location of where the cache files will be stored. This is controlled by the cacheDir option which *must* have a backslash at the end of it.

```
<?php

// Cache the files for 2 days using a Data frontend
$frontCache = new Phalcon\Cache\Frontend\Data(array(
    "lifetime" => 172800
));

// Create the component that will cache "Data" to a "File" backend
// Set the cache file directory - important to keep the "/" at the end of
// of the value for the folder
$cache = new Phalcon\Cache\Backend\File($frontCache, array(
    "cacheDir" => "../app/cache/"
));

// Try to get cached records
$cacheKey = 'robots_order_id.cache';
$robots    = $cache->get($cacheKey);
if ($robots === null) {

    // $robots is null because of cache expiration or data does not exist
    // Make the database call and populate the variable
    $robots = Robots::find(array("order" => "id"));

    // Store it in the cache
    $cache->save($cacheKey, $robots);
}

// Use $robots :)
foreach ($robots as $robot) {
   echo $robot->name, "\n";
}
```

Memcached Backend Example

The above example changes slightly (especially in terms of configuration) when we are using a Memcached backend.

```
<?php

//Cache data for one hour
$frontCache = new Phalcon\Cache\Frontend\Data(array(
    "lifetime" => 3600
));

// Create the component that will cache "Data" to a "Memcached" backend
// Memcached connection settings
$cache = new Phalcon\Cache\Backend\Memcache($frontCache, array(
    "host" => "localhost",
    "port" => "11211"
));

// Try to get cached records
$cacheKey = 'robots_order_id.cache';
$robots    = $cache->get($cacheKey);
if ($robots === null) {

    // $robots is null because of cache expiration or data does not exist
    // Make the database call and populate the variable
    $robots = Robots::find(array("order" => "id"));

    // Store it in the cache
    $cache->save($cacheKey, $robots);
}

// Use $robots :)
foreach ($robots as $robot) {
   echo $robot->name, "\n";
}
```

2.34.5 Querying the cache

The elements added to the cache are uniquely identified by a key. In the case of the File backend, the key is the actual filename. To retrieve data from the cache, we just have to call it using the unique key. If the key does not exist, the get method will return null.

```
<?php

// Retrieve products by key "myProducts"
$products = $cache->get("myProducts");
```

If you want to know which keys are stored in the cache you could call the queryKeys method:

```
<?php

// Query all keys used in the cache
$keys = $cache->queryKeys();
foreach ($keys as $key) {
    $data = $cache->get($key);
    echo "Key=", $key, " Data=", $data;
}

//Query keys in the cache that begins with "my-prefix"
$keys = $cache->queryKeys("my-prefix");
```

2.34.6 Deleting data from the cache

There are times where you will need to forcibly invalidate a cache entry (due to an update in the cached data). The only requirement is to know the key that the data have been stored with.

```
<?php

// Delete an item with a specific key
$cache->queryKeys("someKey");

// Delete all items from the cache
$keys = $cache->queryKeys();
foreach ($keys as $key) {
    $cache->delete($key);
}
```

2.34.7 Checking cache existance

It is possible to check if cache is already exists with given key.

```
<?php

if ($cache->exists("someKey")) {
    echo $cache->get("someKey");
}
else {
    echo "Cache does not exists!";
}
```

2.34.8 Lifetime

A "lifetime" is a time in seconds that a cache could live without expire. By default, all the created caches use the lifetime set in the frontend creation. You can set a specific lifetime in the creation or retrieving of the data from the cache:

Setting the lifetime when retrieving:

```
<?php

$cacheKey = 'my.cache';

//Setting the cache when getting a result
$robots = $cache->get($cacheKey, 3600);
if ($robots === null) {

    $robots = "some robots";

    // Store it in the cache
    $cache->save($cacheKey, $robots);
}
```

Setting the lifetime when saving:

```
<?php

$cacheKey = 'my.cache';
```

```
$robots = $cache->get($cacheKey);
if ($robots === null) {

    $robots = "some robots";

    //Setting the cache when saving data
    $cache->save($cacheKey, $robots, 3600);
}
```

Due to the different nature of the backends maybe is required to use some form or another. For example, the file adapter requires that the "lifetime" will be defined when retrieving, while "Apc" when saving.

A cross-backend way to do this is the following:

```
<?php

$lifetime = 3600;
$cacheKey = 'my.cache';

$robots = $cache->get($cacheKey, $lifetime);
if ($robots === null) {

    $robots = "some robots";

    $cache->save($cacheKey, $robots, $lifetime);
}
```

2.34.9 Multi-Level Cache

This feature of the cache component, allows the developer to implement a multi-level cache. This new feature is very useful because you can save the same data in several cache locations with different lifetimes, reading first from the one with the faster adapter and ending with the slowest one until the data expires:

```
<?php

$ultraFastFrontend = new Phalcon\Cache\Frontend\Data(array(
    "lifetime" => 3600
));

$fastFrontend = new Phalcon\Cache\Frontend\Data(array(
    "lifetime" => 86400
));

$slowFrontend = new Phalcon\Cache\Frontend\Data(array(
    "lifetime" => 604800
));

//Backends are registered from the fastest to the slower
$cache = new \Phalcon\Cache\Multiple(array(
    new Phalcon\Cache\Backend\Apc($ultraFastFrontend, array(
        "prefix" => 'cache',
    )),
    new Phalcon\Cache\Backend\Memcache($fastFrontend, array(
        "prefix" => 'cache',
        "host" => "localhost",
        "port" => "11211"
    )),
```

```
    new Phalcon\Cache\Backend\File($slowFrontend, array(
        "prefix" => 'cache',
        "cacheDir" => "../app/cache/"
    ))
));

//Save, saves in every backend
$cache->save('my-key', $data);
```

2.34.10 Frontend Adapters

The available frontend adapters that are used as interfaces or input sources to the cache are:

Adapter	Description	Example
Output	Read input data from standard PHP output	*Phalcon\Cache\Frontend\Output*
Data	It's used to cache any kind of PHP data (big arrays, objects, text, etc). The data is serialized before stored in the backend.	*Phalcon\Cache\Frontend\Data*
Base64	It's used to cache binary data. The data is serialized using base64_encode before be stored in the backend.	*Phalcon\Cache\Frontend\Base64*
None	It's used to cache any kind of PHP data without serializing them.	*Phalcon\Cache\Frontend\None*

Implementing your own Frontend adapters

The *Phalcon\Cache\FrontendInterface* interface must be implemented in order to create your own frontend adapters or extend the existing ones.

2.34.11 Backend Adapters

The backend adapters available to store cache data are:

Adapter	Description	Info	Required Extensions	Example
File	Stores data to local plain files			*Phalcon\Cache\Backend\File*
Memcached	Stores data to a memcached server	Memcached	memcache	*Phalcon\Cache\Backend\Memcache*
APC	Stores data to the Alternative PHP Cache (APC)	APC	APC extension	*Phalcon\Cache\Backend\Apc*
Mongo	Stores data to Mongo Database	MongoDb	Mongo	*Phalcon\Cache\Backend\Mongo*

Implementing your own Backend adapters

The *Phalcon\Cache\BackendInterface* interface must be implemented in order to create your own backend adapters or extend the existing ones.

File Backend Options

This backend will store cached content into files in the local server. The available options for this backend are:

Option	Description
cacheDir	A writable directory on which cached files will be placed

Memcached Backend Options

This backend will store cached content on a memcached server. The available options for this backend are:

Option	Description
host	memcached host
port	memcached port
persistent	create a persitent connection to memcached?

APC Backend Options

This backend will store cached content on Alternative PHP Cache (APC). This cache backend does not require any additional configuration options.

Mongo Backend Options

This backend will store cached content on a MongoDB server. The available options for this backend are:

Option	Description
server	A MongoDB connection string
db	Mongo database name
collection	Mongo collection in the database

There are more adapters available for this components in the Phalcon Incubator

2.35 Security

This component aids the developer in common security tasks such as password hashing and Cross-Site Request Forgery protection (CSRF).

2.35.1 Password Hashing

Storing passwords in plain text is a bad security practice. Anyone with access to the database will immediately have access to all user accounts thus being able to engage in unauthorized activities. To combat that, many applications use the familiar one way hashing methods "md5" and "sha1". However, hardware evolves each day, and becomes faster, these algorithms are becoming vulnerable to brute force attacks. These attacks are also known as rainbow tables.

To solve this problem we can use hash algorithms as bcrypt. Why bcrypt? Thanks to its "Eksblowfish" key setup algorithm we could make the password encryption as "slow" as we want. Slow algorithms make the process to calculate the real password behind a hash extremely difficult if not impossible. This will protect your for a long time from a possible attack using rainbow tables.

This component gives you the ability to use this algorithm in a simple way:

```php
<?php

    class UsersController extends Phalcon\Mvc\Controller
    {

        public function registerAction()
        {

            $user = new Users();

            $login = $this->request->getPost('login');
            $password = $this->request->getPost('password');

            $user->login = $login;

            //Store the password hashed
            $user->password = $this->security->hash($password);

            $user->save();
        }

    }
```

We saved the password hashed with a default work factor. A higher work factor will make the password less vulnerable as its encryption will be slow. We can check if the password is correct as follows:

```php
<?php

    class SessionController extends Phalcon\Mvc\Controller
    {

        public function loginAction()
        {

            $login = $this->request->getPost('login');
            $password = $this->request->getPost('password');

            $user = Users::findFirst(array(
                "login = ?0",
                "bind" => array($login)
            ));
            if ($user) {
                if ($this->security->checkHash($password, $user->password)) {
                    //The password is valid
                }
            }

            //The validation failed
        }

    }
```

The salt is generated using pseudo-random bytes with the PHP's function openssl_random_pseudo_bytes so is required to have the openssl extension loaded.

2.35.2 Cross-Site Request Forgery (CSRF) protection

This is another common attack against web sites and applications. Forms designed to perform tasks such as user registration or adding comments are vulnerable to this attack.

The idea is to prevent the form values from being sent outside our application. To fix this, we generate a random nonce (token) in each form, add the token in the session and then validate the token once the form posts data back to our application by comparing the stored token in the session to the one submitted by the form:

```
<?php echo Tag::form('session/login') ?>

        <!-- login and password inputs ... -->

        <input type="hidden" name="<?php echo $this->security->getTokenKey() ?>"
               value="<?php echo $this->security->getToken() ?>"/>

</form>
```

Then in the controller's action you can check if the CSRF token is valid:

```
<?php

class SessionController extends Phalcon\Mvc\Controller
{

    public function loginAction()
    {
        if ($this->request->isPost()) {
            if ($this->security->checkToken()) {
                //The token is ok
            }
        }
    }

}
```

Adding a captcha to the form is also recommended to completely avoid the risks of this attack.

2.35.3 Setting up the component

This component is automatically registered in the services container as 'security', you can re-register it to setup it's options:

```
<?php

$di->set('security', function(){

        $security = new Phalcon\Security();

        //Set the password hashing factor to 12 rounds
        $security->setWorkFactor(12);

        return $security;
}, true);
```

2.36 Access Control Lists ACL

Phalcon\Acl provides an easy and lightweight management of ACLs as well as the permissions attached to them. Access Control Lists (ACL) allow an application to control access to its areas and the underlying objects from requests. You are encouraged to read more about the ACL methodology so as to be familiar with its concepts.

In summary, ACLs have roles and resources. Resources are objects which abide by the permissions defined to them by the ACLs. Roles are objects that request access to resources and can be allowed or denied access by the ACL mechanism.

2.36.1 Creating an ACL

This component is designed to initially work in memory. This provides ease of use and speed in accessing every aspect of the list. The *Phalcon\Acl* constructor takes as its first parameter an adapter used to retriever the information related to the control list. An example using the memory adapter is below:

```
<?php $acl = new \Phalcon\Acl\Adapter\Memory();
```

By default *Phalcon\Acl* allows access to action on resources that have not been yet defined. To increase the security level of the access list we can define a "deny" level as a default access level.

```
<?php

// Default action is deny access
$acl->setDefaultAction(Phalcon\Acl::DENY);
```

2.36.2 Adding Roles to the ACL

A role is an object that can or cannot access certain resources in the access list. As an example, we will define roles as groups of people in an organization. The *Phalcon\Acl\Role* class is available to create roles in a more structured way. Let's add some roles to our recently created list:

```
<?php

// Create some roles
$roleAdmins = new \Phalcon\Acl\Role("Administrators", "Super-User role");
$roleGuests = new \Phalcon\Acl\Role("Guests");

// Add "Guests" role to acl
acl->addRole($roleGuests);

// Add "Designers" role to acl without a Phalcon\Acl\Role
$acl->addRole("Designers");
```

As you can see, roles are defined directly without using a instance.

2.36.3 Adding Resources

Resources are objects where access is controlled. Normally in MVC applications resources refer to controllers. Although this is not mandatory, the *Phalcon\Acl\Resource* class can be used in defining resources. It's important to add related actions or operations to a resource so that the ACL can understand what it should to control.

```
<?php

// Define the "Customers" resource
$customersResource = new \Phalcon\Acl\Resource("Customers");

// Add "customers" resource with a couple of operations
$acl->addResource($customersResource, "search");
$acl->addResource($customersResource, array("create", "update"));
```

2.36.4 Defining Access Controls

Now we've roles and resources. It's time to define the ACL i.e. which roles can access which resources. This part is very important especially taking in consideration your default access level "allow" or "deny".

```
<?php

// Set access level for roles into resources
$acl->allow("Guests", "Customers", "search");
$acl->allow("Guests", "Customers", "create");
$acl->deny("Guests", "Customers", "update");
```

The allow method designates that a particular role has granted access to access a particular resource. The deny method does the opposite.

2.36.5 Querying an ACL

Once the list has been completely defined. We can query it to check if a role has a given permission or not.

```
<?php

// Check whether role has access to the operations
$acl->isAllowed("Guests", "Customers", "edit");   //Returns 0
$acl->isAllowed("Guests", "Customers", "search"); //Returns 1
$acl->isAllowed("Guests", "Customers", "create"); //Returns 1
```

2.36.6 Roles Inheritance

You can build complex role structures using the inheritance that *Phalcon\Acl\Role* provides. Roles can inherit from other roles, thus allowing access to supersets or subsets of resources. To use role inheritance, you need to pass the inherited role as the second parameter of the function call, when adding that role in the list.

```
<?php

// Create some roles
$roleAdmins = new \Phalcon\Acl\Role("Administrators", "Super-User role");
$roleGuests = new \Phalcon\Acl\Role("Guests");

// Add "Guests" role to acl
$acl->addRole($roleGuests);

// Add "Administrators" role inheriting from "Guests" its accesses
$acl->addRole($roleAdmins, $roleGuests);
```

2.36.7 Serializing ACL lists

To improve performance *Phalcon\Acl* instances can be serialized and stored in APC, session, text files or a database table so that they can be loaded at will without having to redefine the whole list. You can do that as follows:

```
<?php

//Check whether acl data already exist
if (!file_exists("app/security/acl.data")) {

    $acl = new \Phalcon\Acl("Memory");

    //... Define roles, resources, access, etc

    // Store serialized list into plain file
    file_put_contents("app/security/acl.data", serialize($acl));

} else {

     //Restore acl object from serialized file
     $acl = unserialize(file_get_contents("app/security/acl.data"));
}

// Use acl list as needed
if ($acl->isAllowed("Guests", "Customers", "edit")) {
    echo "Access granted!";
} else {
    echo "Access denied :(";
}
```

2.36.8 Acl Events

Phalcon\Acl is able to send events to a *EventsManager* if it's present. Events are triggered using the type "acl". Some events when returning boolean false could stop the active operation. The following events are supported:

Event Name	Triggered	Can stop operation?
beforeCheckAccess	Triggered before checking if a role/resource has access	Yes
afterCheckAccess	Triggered after checking if a role/resource has access	No

The following example demonstrates how to attach listeners to this component:

```
<?php

//Create an event manager
$eventsManager = new Phalcon\Events\Manager();

//Attach a listener for type "acl"
$eventsManager->attach("acl", function($event, $acl) {
    if ($event->getType() == 'beforeCheckAccess') {
         echo   $acl->getActiveRole(),
                $acl->getActiveResource(),
                $acl->getActiveAccess();
    }
});

$acl = new \Phalcon\Acl\Adapter\Memory();

//Setup the $acl
```

```
//...

//Bind the eventsManager to the acl component
$acl->setEventsManager($eventManagers);
```

2.36.9 Implementing your own adapters

The *Phalcon\Acl\AdapterInterface* interface must be implemented in order to create your own ACL adapters or extend the existing ones.

2.37 Multi-lingual Support

The component *Phalcon\Translate* aids in creating multilingual applications. Applications using this component, display content in different languages, based on the user's chosen language supported by the application.

2.37.1 Adapters

This component makes use of adapters to read translation messages from different sources in a unified way.

Adapter	Description
NativeArray	Uses PHP arrays to store the messages. This is the best option in terms of performance.

2.37.2 Component Usage

Translation strings are stored in files. The structure of these files could vary depending of the adapter used. Phalcon gives you the freedom to organize your translation strings. A simple structure could be:

```
app/messages/en.php
app/messages/es.php
app/messages/fr.php
app/messages/zh.php
```

Each file contains an array of the translations in a key/value manner. For each translation file, keys are unique. The same array is used in different files, where keys remain the same and values contain the translated strings depending on each language.

```
<?php

//app/messages/es.php
$messages = array(
    "hi"       => "Hello",
    "bye"      => "Good Bye",
    "hi-name" => "Hello %name%",
    "song"     => "This song is %song%"
);

<?php

//app/messages/fr.php
$messages = array(
    "hi"       => "Bonjour",
    "bye"      => "Au revoir",
```

```
    "hi-name" => "Bonjour %name%",
    "song"    => "La chanson est %song%"
);
```

Implementing the translation mechanism in your application is trivial but depends on how you wish to implement it. You can use an automatic detection of the language from the user's browser or you can provide a settings page where the user can select their language.

A simple way of detecting the user's language is to parse the $_SERVER['HTTP_ACCEPT_LANGUAGE'] contents, or if you wish, access it directly by calling $this->request->getBestLanguage() from an action/controller:

```
<?php

class UserController extends \Phalcon\Mvc\Controller
{

  protected function _getTranslation()
  {

    //Ask browser what is the best language
    $language = $this->request->getBestLanguage();

    //Check if we have a translation file for that lang
    if (file_exists("app/messages/".$language.".php")) {
       require "app/messages/".$language.".php";
    } else {
       // fallback to some default
       require "app/messages/en.php";
    }

    //Return a translation object
    return new \Phalcon\Translate\Adapter\NativeArray(array(
       "content" => $messages
    ));

  }

  public function indexAction()
  {
    $this->view->setVar("name", "Mike");
    $this->view->setVar("t", $this->_getTranslation());
  }

}
```

The _getTranslation method is available for all actions that require translations. The $t variable is passed to the views, and with it, we can translate strings in that layer:

```
<!-- welcome -->
<!-- String: hi => 'Hello' -->
<p><?php echo $t->_("hi"), " ", $name; ?></p>
```

The "_" function is returning the translated string based on the index passed. Some strings need to incorporate placeholders for calculated data i.e. Hello %name%. These placeholders can be replaced with passed parameters in the "_ function. The passed parameters are in the form of a key/value array, where the key matches the placeholder name and the value is the actual data to be replaced:

```
<!-- welcome -->
<!-- String: hi-user => 'Hello %name%' -->
```

```
<p><?php echo $t->_("hi-user", array("name" => $name)); ?></p>
```

Some applications implement multilingual on the URL such as http://www.mozilla.org/**es-ES**/firefox/. Phalcon can implement this by using a *Router*.

2.37.3 Implementing your own adapters

The *Phalcon\Translate\AdapterInterface* interface must be implemented in order to create your own translate adapters or extend the existing ones:

```
<?php

class MyTranslateAdapter implements Phalcon\Translate\AdapterInterface
{

    /**
     * Adapter constructor
     *
     * @param array $data
     */
    public function __construct($options);

    /**
     * Returns the translation string of the given key
     *
     * @param    string $translateKey
     * @param    array $placeholders
     * @return   string
     */
    public function _($translateKey, $placeholders=null);

    /**
     * Returns the translation related to the given key
     *
     * @param    string $index
     * @param    array $placeholders
     * @return   string
     */
    public function query($index, $placeholders=null);

    /**
     * Check whether is defined a translation key in the internal array
     *
     * @param    string $index
     * @return   bool
     */
    public function exists($index);

}
```

There are more adapters available for this components in the Phalcon Incubator

2.38 Universal Class Loader

Phalcon\Loader is a component that allows you to load project classes automatically, based on some predefined rules.

Since this component is written in C, it provides the lowest overhead in reading and interpreting external PHP files.

The behavior of this component is based on the PHP's capability of autoloading classes. If a class that does not exist is used in any part of the code, a special handler will try to load it. *Phalcon\Loader* serves as the special handler for this operation. By loading classes on a need to load basis, the overall performance is increased since the only file reads that occur are for the files needed. This technique is called lazy initialization.

With this component you can load files from other projects or vendors, this autoloader is PSR-0 compliant.

Phalcon\Loader offers four options to autoload classes. You can use them one at a time or combine them.

2.38.1 Registering Namespaces

If you're organizing your code using namespaces, or external libraries do so, the registerNamespaces() provides the autoloading mechanism. It takes an associative array, which keys are namespace prefixes and their values are directories where the classes are located in. The namespace separator will be replaced by the directory separator when the loader try to find the classes. Remember always to add a trailing slash at the end of the paths.

```
<?php

// Creates the autoloader
$loader = new \Phalcon\Loader();

//Register some namespaces
$loader->registerNamespaces(
    array(
        "Example\Base"    => "vendor/example/base/",
        "Example\Adapter" => "vendor/example/adapter/",
        "Example"         => "vendor/example/",
    )
);

// register autoloader
$loader->register();

// The required class will automatically include the
// file vendor/example/adapter/Some.php
$some = new Example\Adapter\Some();
```

2.38.2 Registering Prefixes

This strategy is similar to the namespaces strategy. It takes an associative array, which keys are prefixes and their values are directories where the classes are located in. The namespace separator and the "_" underscore character will be replaced by the directory separator when the loader try to find the classes. Remember always to add a trailing slash at the end of the paths.

```
<?php

// Creates the autoloader
$loader = new \Phalcon\Loader();

//Register some prefixes
$loader->registerPrefixes(
    array(
        "Example_Base"    => "vendor/example/base/",
        "Example_Adapter" => "vendor/example/adapter/",
```

```
        "Example_"          => "vendor/example/",
    )
);

// register autoloader
$loader->register();

// The required class will automatically include the
// file vendor/example/adapter/Some.php
$some = new Example_Adapter_Some();
```

2.38.3 Registering Directories

The third option is to register directories, in which classes could be found. This option is not recommended in terms of performance, since Phalcon will need to perform a significant number of file stats on each folder, looking for the file with the same name as the class. It's important to register the directories in relevance order. Remember always add a trailing slash at the end of the paths.

```
<?php

// Creates the autoloader
$loader = new \Phalcon\Loader();

// Register some directories
$loader->registerDirs(
    array(
        "library/MyComponent/",
        "library/OtherComponent/Other/",
        "vendor/example/adapters/",
        "vendor/example/"
    )
);

// register autoloader
$loader->register();

// The required class will automatically include the file from
// the first directory where it has been located
// i.e. library/OtherComponent/Other/Some.php
$some = new Some();
```

2.38.4 Registering Classes

The last option is to register the class name and its path. This autoloader can be very useful when the folder convention of the project does not allow for easy retrieval of the file using the path and the class name. This is the fastest method of autoloading. However the more your application grows, the more classes/files need to be added to this autoloader, which will effectively make maintenance of the class list very cumbersome and it is not recommended.

```
<?php

// Creates the autoloader
$loader = new \Phalcon\Loader();

// Register some classes
$loader->registerClasses(
```

```
    array(
        "Some"         => "library/OtherComponent/Other/Some.php",
        "Example\Base" => "vendor/example/adapters/Example/BaseClass.php",
    )
);

// register autoloader
$loader->register();

// Requiring a class will automatically include the file it references
// in the associative array
// i.e. library/OtherComponent/Other/Some.php
$some = new Some();
```

2.38.5 Additional file extensions

Some autoloading strategies such as "prefixes", "namespaces" or "directories" automatically append the "php" extension at the end of the checked file. If you are using additional extensions you could set it with the method "setExtensions". Files are checked in the order as it were defined:

```
<?php

 // Creates the autoloader
$loader = new \Phalcon\Loader();

//Set file extensions to check
$loader->setExtensions(array("php", "inc", "phb"));
```

2.38.6 Modifying current strategies

Additional data could be added to the existing values for strategies in the following way:

```
<?php

// Adding more directories
$loader->registerDirs(
    array(
        "../app/library/"
        "../app/plugins/"
    ),
    true
);
```

Passing "true" as second parameter will merge the current values with new ones in any strategy.

2.38.7 Autoloading Events

In the following example, the EventsManager is working with the class loader, allowing us to obtain debugging information regarding the flow of operation:

```
<?php

$eventsManager = new \Phalcon\Events\Manager();
```

```
$loader = new \Phalcon\Loader();

$loader->registerNamespaces(array(
   'Example\\Base' => 'vendor/example/base/',
   'Example\\Adapter' => 'vendor/example/adapter/',
   'Example' => 'vendor/example/'
));

//Listen all the loader events
$eventsManager->attach('loader', function($event, $loader) {
    if ($event->getType() == 'beforeCheckPath') {
        echo $loader->getCheckedPath();
    }
});

$loader->setEventsManager($eventsManager);

$loader->register();
```

Some events when returning boolean false could stop the active operation. The following events are supported:

Event Name	Triggered
beforeCheckClass	Triggered before starting the autoloading process
pathFound	Triggered when the loader locate a class
afterCheckClass	Triggered after finish the autoloading process. If this event is launched the autoloader didn't find the class file

2.38.8 Troubleshooting

Some things to keep in mind when using the universal autoloader:

- Auto-loading process is case-sensitive, the class will be loaded as it is written in the code
- Strategies based on namespaces/prefixes are faster than the directories strategy
- If a cache bytecode like APC is installed this will used to retrieve the requested file (an implicit caching of the file is performed)

2.39 Logging

Phalcon\Logger is a component whose purpose is to provide logging services for applications. It offers logging to different backends using different adapters. It also offers transaction logging, configuration options, different formats and filters. You can use the *Phalcon\Logger* for every logging need your application has, from debugging processes to tracing application flow.

2.39.1 Adapters

This component makes use of adapters to store the logged messages. The use of adapters allows for a common interface for logging while switching backends if necessary. The adapters supported are:

Adapter	Description	API
File	Logs to a plain text file	*Phalcon\Logger\Adapter\File*
Stream	Logs to a PHP Streams	*Phalcon\Logger\Adapter\Stream*
Syslog	Logs to the system logger	*Phalcon\Logger\Adapter\Syslog*

2.39.2 Creating a Log

The example below shows how to create a log and add messages to it:

```
<?php

$logger = new \Phalcon\Logger\Adapter\File("app/logs/test.log");
$logger->log("This is a message");
$logger->log("This is an error", \Phalcon\Logger::ERROR);
$logger->error("This is another error");
```

The log generated is below:

```
[Tue, 17 Apr 12 22:09:02 -0500][DEBUG] This is a message
[Tue, 17 Apr 12 22:09:02 -0500][ERROR] This is an error
[Tue, 17 Apr 12 22:09:02 -0500][ERROR] This is another error
```

2.39.3 Transactions

Logging data to an adapter i.e. File (file system) is always an expensive operation in terms of performance. To combat that, you can take advantage of logging transactions. Transactions store log data temporarily in memory and later on write the data to the relevant adapter (File in this case) in a single atomic operation.

```
<?php

// Create the logger
$logger = new \Phalcon\Logger\Adapter\File("app/logs/test.log");

// Start a transaction
$logger->begin();

// Add messages
$logger->alert("This is an alert");
$logger->error("This is another error");

// Commit messages to file
$logger->commit();
```

2.39.4 Logging to Multiple Handlers

Phalcon\Logger allows to send messages to multiple handlers with a just single call:

```
<?php

$logger = new \Phalcon\Logger\Multiple();

$logger->push(new \Phalcon\Logger\Adapter\File('test.log'));
$logger->push(new \Phalcon\Logger\Adapter\Stream('php://stdout'));

$logger->log("This is a message");
$logger->log("This is an error", \Phalcon\Logger::ERROR);
$logger->error("This is another error");
```

The messages are sent to the handlers in the order they where registered.

2.39.5 Message Formatting

This component makes use of 'formatters' to format messages before sent them to the backend. The formatters available are:

Adapter	Description	API
Line	Formats the messages using a one-line string	*Phalcon\Logger\Formatter\Line*
Json	Prepares a message to be encoded with JSON	*Phalcon\Logger\Formatter\Json*
Syslog	Prepares a message to be sent to syslog	*Phalcon\Logger\Formatter\Syslog*

Line Formatter

Formats the messages using a one-line string. The default logging format is:

[%date%][%type%] %message%

You can change the default format using setFormat(), this allows you to change the format of the logged messages by defining your own. The log format variables allowed are:

Variable	Description
%message%	The message itself expected to be logged
%date%	Date the message was added
%type%	Uppercase string with message type

The example below shows how to change the log format:

```
<?php

//Changing the logger format
$formatter = new Phalcon\Logger\Formatter\Line("%date% - %message%");
$logger->setFormatter($formatter);
```

Implementing your own formatters

The *Phalcon\Logger\FormatterInterface* interface must be implemented in order to create your own logger formatter or extend the existing ones.

2.39.6 Adapters

The following examples show the basic use of each adapter:

Stream Logger

The stream logger writes messages to a valid registered stream in PHP. A list of streams is available here:

```
<?php

// Opens a stream using zlib compression
$logger = new \Phalcon\Logger\Adapter\Stream("compress.zlib://week.log.gz");

// Writes the logs to stderr
$logger = new \Phalcon\Logger\Adapter\Stream("php://stderr");
```

File Logger

This logger uses plain files to log any kind of data. By default all logger files are open using append mode which open the files for writing only; placing the file pointer at the end of the file. If the file does not exist, attempt to create it. You can change this mode passing additional options to the constructor:

```
<?php

// Create the file logger in 'w' mode
$logger = new \Phalcon\Logger\Adapter\File("app/logs/test.log", array(
    'mode' => 'w'
));
```

Syslog Logger

This logger sends messages to the system logger. The syslog behavior may vary from one operating system to another.

```
<?php

// Basic Usage
$logger = new \Phalcon\Logger\Adapter\Syslog(null);

// Setting ident/mode/facility
$logger = new \Phalcon\Logger\Adapter\Syslog("ident-name", array(
    'option' => LOG_NDELAY,
    'facility' => LOG_MAIL
));
```

Implementing your own adapters

The *Phalcon\Logger\AdapterInterface* interface must be implemented in order to create your own logger adapters or extend the existing ones.

2.40 Annotations Parser

It is the first time that an annotations parser component is written in C for the PHP world. Phalcon\Annotations is a general purpose component that provides ease of parsing and caching annotations in PHP classes to be used in applications.

Annotations are read from docblocks in classes, methods and properties. An annotation can be placed at any position in the docblock:

```
<?php

/**
 * This is the class description
 *
 * @AmazingClass(true)
 */
class Example
{

        /**
         * This a property with a special feature
```

```
         *
         * @SpecialFeature
         */
        protected $someProperty;

        /**
         * This is a method
         *
         * @SpecialFeature
         */
        public function someMethod()
        {
                // ...
        }

}
```

In the above example we find some annotations in the comments, an annotation has the following syntax:

@Annotation-Name[(param1, param2, ...)]

Also, an annotation could be placed at any part of a docblock:

```
<?php

/**
 * This a property with a special feature
 *
 * @SpecialFeature
 *
 * More comments
 *
 * @AnotherSpecialFeature(true)
 */
```

The parser is highly flexible, the following docblock is valid:

```
<?php

/**
 * This a property with a special feature @SpecialFeature({
someParameter="the value", false

 })  More comments @AnotherSpecialFeature(true) @MoreAnnotations
 **/
```

However, to make the code more maintainable and understandable it is recommended to place annotations at the end of the docblock:

```
<?php

/**
 * This a property with a special feature
 * More comments
 *
 * @SpecialFeature({someParameter="the value", false})
 * @AnotherSpecialFeature(true)
 */
```

2.40.1 Reading Annotations

A reflector is implemented to easily get the annotations defined on a class using an object-oriented interface:

```
<?php

$reader = new \Phalcon\Annotations\Adapter\Memory();

//Reflect the annotations in the class Example
$reflector = $reader->get('Example');

//Read the annotations in the class' docblock
$annotations = $reflector->getClassAnnotations();

//Traverse the annotations
foreach ($annotations as $annotation) {

        //Print the annotation name
        echo $annotation->getName(), PHP_EOL;

        //Print the number of arguments
        echo $annotation->numberArguments(), PHP_EOL;

        //Print the arguments
        print_r($annotation->getArguments());
}
```

The annotation reading process is very fast, however, for performance reasons it is recommended to store the parsed annotations using an adapter. Adapters cache the processed annotations avoiding the need of parse the annotations again and again.

Phalcon\Annotations\Adapter\Memory was used in the above example. This adapter only caches the annotations while the request is running, for this reason th adapter is more suitable for development. There are other adapters to swap out when the application is in production stage.

2.40.2 Types of Annotations

Annotations may have parameters or not. A parameter could be a simple literal (strings, number, boolean, null), an array, a hashed list or other annotation:

```
<?php

/**
 * Simple Annotation
 *
 * @SomeAnnotation
 */

/**
 * Annotation with parameters
 *
 * @SomeAnnotation("hello", "world", 1, 2, 3, false, true)
 */

/**
 * Annotation with named parameters
 *
 * @SomeAnnotation(first="hello", second="world", third=1)
```

```
 * @SomeAnnotation(first: "hello", second: "world", third: 1)
 */

/**
 * Passing an array
 *
 * @SomeAnnotation([1, 2, 3, 4])
 * @SomeAnnotation({1, 2, 3, 4})
 */

/**
 * Passing a hash as parameter
 *
 * @SomeAnnotation({first=1, second=2, third=3})
 * @SomeAnnotation({'first'=1, 'second'=2, 'third'=3})
 * @SomeAnnotation({'first': 1, 'second': 2, 'third': 3})
 * @SomeAnnotation(['first': 1, 'second': 2, 'third': 3])
 */

/**
 * Nested arrays/hashes
 *
 * @SomeAnnotation({"name"="SomeName", "other"={
 *               "foo1": "bar1", "foo2": "bar2", {1, 2, 3},
 * }})
 */

/**
 * Nested Annotations
 *
 * @SomeAnnotation(first=@AnotherAnnotation(1, 2, 3))
 */
```

2.40.3 Practical Usage

Let's pretend we've the following controller and the developer wants to create a plugin that automatically starts the cache if the latest action executed is marked as cacheable. First off all we register a plugin in the Dispatcher service to be notified when a route is executed:

```
<?php

$di['dispatcher'] = function() {

        $eventsManager = new \Phalcon\Events\Manager();

        //Attach the plugin to 'dispatch' events
        $eventsManager->attach('dispatch', new CacheEnablerPlugin());

        $dispatcher = new \Phalcon\Mvc\Dispatcher();
        $dispatcher->setEventsManager($eventsManager);
        return $dispatcher;
};
```

CacheEnablerPlugin is a plugin that intercepts every action executed in the dispatcher enabling the cache if needed:

```
<?php
```

```
/**
 * Enables the cache for a view if the latest
 * executed action has the annotation @Cache
 */
class CacheEnablerPlugin extends \Phalcon\Mvc\User\Plugin
{

        /**
         * This event is executed before every route is executed in the dispatcher
         *
         */
        public function beforeExecuteRoute($event, $dispatcher)
        {

                //Parse the annotations in the method currently executed
                $annotations = $this->annotations->getMethod(
                        $dispatcher->getActiveController(),
                        $dispatcher->getActiveMethod()
                );

                //Check if the method has an annotation 'Cache'
                if ($annotations->has('Cache')) {

                        //The method has the annotation 'Cache'
                        $annotation = $annotations->get('Cache');

                        //Get the lifetime
                        $lifetime = $annotation->getNamedParameter('lifetime');

                        $options = array('lifetime' => $lifetime);

                        //Check if there is a user defined cache key
                        if ($annotation->hasNamedParameter('key')) {
                                $options['key'] = $annotation->getNamedParameter('key');
                        }

                        //Enable the cache for the current method
                        $this->view->cache($options);
                }

        }

}
```

Now, we can use the annotation in a controller:

```
<?php

class NewsController extends \Phalcon\Mvc\Controller
{

        public function indexAction()
        {

        }

        /**
         * This is a comment
         *
```

```
         * @Cache(lifetime=86400)
         */
        public function showAllAction()
        {
                $this->view->article = Articles::find();
        }

        /**
         * This is a comment
         *
         * @Cache(key="my-key", lifetime=86400)
         */
        public function showAction($slug)
        {
                $this->view->article = Articles::findFirstByTitle($slug);
        }

}
```

2.41 Command Line Applications

CLI applications are executed from the command line. They are useful to create cron jobs, scripts, command utilities and more.

2.41.1 Tasks

Tasks are similar to controllers, on them can be implemented

```
<?php

class MonitoringTask extends \Phalcon\CLI\Task
{

    public function mainAction()
    {

    }

}
```

```
<?php

//Using the CLI factory default services container
$di = new Phalcon\DI\FactoryDefault\CLI();

//Create a console application
$console = new \Phalcon\CLI\Console();
$console->setDI($di);

//
$console->handle(array('shell_script_name', 'echo'));
```

2.42 Database Abstraction Layer

Phalcon\Db is the component behind *Phalcon\Mvc\Model* that powers the model layer in the framework. It consists of an independent high-level abstraction layer for database systems completely written in C.

This component allows for a lower level database manipulation than using traditional models.

> This guide is not intended to be a complete documentation of available methods and their arguments. Please visit the *API* for a complete reference.

2.42.1 Database Adapters

This component makes use of adapters to encapsulate specific database system details. Phalcon uses PDO to connect to databases. The following database engines are supported:

Name	Description	API
MySQL	Is the world's most used relational database management system (RDBMS) that runs as a server providing multi-user access to a number of databases	*Phalcon\Db\Adapter\Pdo\Mysql*
PostgreSQL	PostgreSQL is a powerful, open source relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness.	*Phalcon\Db\Adapter\Pdo\Postgresql*
SQLite	SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine	*Phalcon\Db\Adapter\Pdo\Sqlite*

Implementing your own adapters

The *Phalcon\Db\AdapterInterface* interface must be implemented in order to create your own database adapters or extend the existing ones.

2.42.2 Database Dialects

Phalcon encapsulates the specific details of each database engine in dialects. Those provide common functions and SQL generator to the adapters.

Name	Description	API
MySQL	SQL specific dialect for MySQL database system	*Phalcon\Db\Dialect\Mysql*
PostgreSQL	SQL specific dialect for PostgreSQL database system	*Phalcon\Db\Dialect\Postgresql*
SQLite	SQL specific dialect for SQLite database system	*Phalcon\Db\Dialect\Sqlite*

2.42.3 Connecting to Databases

To create a connection it's neccesary instantiate the adapter class. It only requires an array with the connection parameters. The example below shows how to create a connection passing both required and optional parameters:

Implementing your own dialects

The *Phalcon\Db\DialectInterface* interface must be implemented in order to create your own database dialects or extend the existing ones.

```php
<?php

// Required
$config = array(
    "host" => "127.0.0.1",
    "username" => "mike",
    "password" => "sigma",
    "dbname" => "test_db"
);

// Optional
$config["persistent"] = false;

// Create a connection
$connection = new \Phalcon\Db\Adapter\Pdo\Mysql($config);

<?php

// Required
$config = array(
    "host" => "localhost",
    "username" => "postgres",
    "password" => "secret1",
    "dbname" => "template"
);

// Optional
$config["schema"] = "public";

// Create a connection
$connection = new \Phalcon\Db\Adapter\Pdo\Postgresql($config);

<?php

// Required
$config = array(
    "dbname" => "/path/to/database.db"
);

// Create a connection
$connection = new \Phalcon\Db\Adapter\Pdo\Sqlite($config);
```

2.42.4 Finding Rows

Phalcon\Db provides several methods to query rows from tables. The specific SQL syntax of the target database engine is required in this case:

```php
<?php

$sql = "SELECT id, name FROM robots ORDER BY name";

// Send a SQL statement to the database system
$result = $connection->query($sql);

// Print each robot name
while ($robot = $result->fetch()) {
   echo $robot["name"];
```

```
}

// Get all rows in an array
$robots = $connection->fetchAll($sql);
foreach ($robots as $robot) {
   echo $robot["name"];
}

// Get only the first row
$robot = $connection->fetchOne($sql);
```

By default these calls create arrays with both associative and numeric indexes. You can change this behavior by using Phalcon\Db\Result::setFetchMode(). This method receives a constant, defining which kind of index is required.

Constant	Description
Phalcon\Db::FETCH_NUM	Return an array with numeric indexes
Phalcon\Db::FETCH_ASSOC	Return an array with associative indexes
Phalcon\Db::FETCH_BOTH	Return an array with both associative and numeric indexes
Phalcon\Db::FETCH_OBJ	Return an object instead of an array

```
<?php

$sql = "SELECT id, name FROM robots ORDER BY name";
$result = $connection->query($sql);

$result->setFetchMode(Phalcon\Db::DB_NUM);
while ($robot = $result->fetch()) {
   echo $robot[0];
}
```

The Phalcon\Db::query() returns an instance of *Phalcon\Db\Result\Pdo*. These objects encapsulate all the functionality related to the returned resultset i.e. traversing, seeking specific records, count etc.

```
<?php

$sql = "SELECT id, name FROM robots";
$result = $connection->query($sql);

// Traverse the resultset
while ($robot = $result->fetch()) {
   echo $robot["name"];
}

// Seek to the third row
$result->seek(2);
$robot = $result->fetch();

// Count the resultset
echo $result->numRows();
```

2.42.5 Binding Parameters

Bound parameters is also supported in *Phalcon\Db*. Although there is a minimal performance impact by using bound parameters, you are encouraged to use this methodology so as to eliminate the possibility of your code being subject to SQL injection attacks. Both string and integer placeholders are supported. Binding parameters can simply be achieved as follows:

```
<?php

// Binding with numeric placeholders
$sql     = "SELECT * FROM robots WHERE name = ?1 ORDER BY name";
$sql     = $connection->bindParams($sql, array(1 => "Wall-E"));
$result = $connection->query($sql);

// Binding with named placeholders
$sql     = "INSERT INTO `robots`(name`, year) VALUES (:name:, :year:)";
$sql     = $connection->bindParams($sql, array("name" => "Astro Boy", "year" => 1952));
$success = $connection->query($sql);
```

When using numeric placeholders, you will need to define them as integers i.e. 1 or 2. In this case "1" or "2" are considered strings and not numbers, so the placeholder could not be successfully replaced. With any adapter data are automatically escaped using PDO Quote.

This function takes into account the connection charset, so its recommended to define the correct charset in the connection parameters or in your database server configuration, as a wrong charset will produce undesired effects when storing or retrieving data.

Also, you can pass your parameterers directly to the execute/query methods. In this case bound parameters are directly passed to PDO:

```
<?php

// Binding with PDO placeholders
$sql     = "SELECT * FROM robots WHERE name = ? ORDER BY name";
$result = $connection->query($sql, array(1 => "Wall-E"));
```

2.42.6 Inserting/Updating/Deleting Rows

To insert, update or delete rows, you can use raw SQL or use the preset functions provided by the class:

```
<?php

// Inserting data with a raw SQL statement
$sql     = "INSERT INTO `robots`(`name`, `year`) VALUES ('Astro Boy', 1952)";
$success = $connection->execute($sql);

//With placeholders
$sql     = "INSERT INTO `robots`(`name`, `year`) VALUES (?, ?)";
$success = $connection->execute($sql, array('Astroy Boy', 1952));

// Generating dynamically the necessary SQL
$success = $connection->insert(
   "robots",
   array("Astro Boy", 1952),
   array("name", "year")
);

// Updating data with a raw SQL statement
$sql     = "UPDATE `robots` SET `name` = 'Astro boy' WHERE `id` = 101";
$success = $connection->execute($sql);

//With placeholders
$sql     = "UPDATE `robots` SET `name` = ? WHERE `id` = ?";
$success = $connection->execute($sql, array('Astroy Boy', 101));
```

```
// Generating dynamically the necessary SQL
$success = $connection->update(
   "robots",
   array("name")
   array("New Astro Boy"),
   "id = 101"
);

// Deleting data with a raw SQL statement
$sql     = "DELETE `robots` WHERE `id` = 101";
$success = $connection->execute($sql);

//With placeholders
$sql     = "DELETE `robots` WHERE `id` = ?";
$success = $connection->execute($sql, array(101));

// Generating dynamically the necessary SQL
$success = $connection->delete("robots", "id = 101");
```

2.42.7 Database Events

Phalcon\Db is able to send events to a *EventsManager* if it's present. Some events when returning boolean false could stop the active operation. The following events are supported:

Event Name	Triggered	Can stop operation?
afterConnect	After a successfully connection to a database system	No
beforeQuery	Before send a SQL statement to the database system	Yes
afterQuery	After send a SQL statement to database system	No
beforeDisconnect	Before close a temporal database connection	No
beginTransaction	Before a transaction is going to be started	No
rollbackTransaction	Before a transaction in the transaction	No
commitTransaction	Before a transaction the transaction is commite | No	

Bind an EventsManager to a connection is simple, Phalcon\Db will trigger the events with the type "db":

```
<?php

$eventsManager = new Phalcon\Events\Manager();

//Listen all the database events
$eventsManager->attach('db', $dbListener);

$connection = new \Phalcon\Db\Adapter\Pdo\Mysql(array(
    "host" => "localhost",
    "username" => "root",
    "password" => "secret",
    "dbname" => "invo"
));

//Assign the eventsManager to the db adapter instance
$connection->setEventsManager($eventsManager);
```

2.42.8 Profiling SQL Statements

Phalcon\Db includes a profiling component called *Phalcon\Db\Profiler*, that is used to analyze the performance of

database operations so as to diagnose performance problems and discover bottlenecks.

Database profiling is really easy With *Phalcon\Db\Profiler*:

```
<?php

$eventsManager = new \Phalcon\Events\Manager();

$profiler = new \Phalcon\Db\Profiler();

//Listen all the database events
$eventsManager->attach('db', function($event, $connection) use ($profiler) {
    if ($event->getType() == 'beforeQuery') {
        //Start a profile with the active connection
        $profiler->startProfile($connection->getSQLStatement());
    }
    if ($event->getType() == 'afterQuery') {
        //Stop the active profile
        $profiler->stopProfile();
    }
});

//Assign the events manager to the connection
$connection->setEventsManager($eventsManager);

$sql = "SELECT buyer_name, quantity, product_name "
     . "FROM buyers "
     . "LEFT JOIN products ON buyers.pid = products.id";

// Execute a SQL statement
$connection->query($sql);

// Get the last profile in the profiler
$profile = $profiler->getLastProfile();

echo "SQL Statement: ", $profile->getSQLStatement(), "\n";
echo "Start Time: ", $profile->getInitialTime(), "\n";
echo "Final Time: ", $profile->getFinalTime(), "\n";
echo "Total Elapsed Time: ", $profile->getTotalElapsedSeconds(), "\n";
```

You can also create your own profile class based on *Phalcon\Db\Profiler* to record real time statistics of the statements sent to the database system:

```
<?php

use \Phalcon\Db\Profiler as Profiler;
use \Phalcon\Db\Profiler\Item as Item;

class DbProfiler extends Profiler
{

    /**
     * Executed before the SQL statement will sent to the db server
     */
    public function beforeStartProfile(Item $profile)
    {
        echo $profile->getSQLStatement();
    }
```

```
    /**
     * Executed after the SQL statement was sent to the db server
     */
    public function afterEndProfile(Item $profile)
    {
        echo $profile->getTotalElapsedSeconds();
    }

}

//Create a EventsManager
$eventsManager = new Phalcon\Events\Manager();

//Create a listener
$dbProfiler = new DbProfiler();

//Attach the listener listening for all database events
$eventsManager->attach('db', $dbProfiler);
```

2.42.9 Logging SQL Statements

Using high-level abstraction components such as *Phalcon\Db* to access a database, it is difficult to understand which statements are sent to the database system. *Phalcon\Logger* interacts with *Phalcon\Db*, providing logging capabilities on the database abstraction layer.

```
<?php

$eventsManager = new Phalcon\Events\Manager();

$logger = new \Phalcon\Logger\Adapter\File("app/logs/db.log");

//Listen all the database events
$eventsManager->attach('db', function($event, $connection) use ($logger) {
    if ($event->getType() == 'beforeQuery') {
        $logger->log($connection->getSQLStatement(), \Phalcon\Logger::INFO);
    }
});

//Assign the eventsManager to the db adapter instance
$connection->setEventsManager($eventsManager);

//Execute some SQL statement
$connection->insert(
    "products",
    array("Hot pepper", 3.50),
    array("name", "price")
);
```

As above, the file *app/logs/db.log* will contain something like this:

```
[Sun, 29 Apr 12 22:35:26 -0500][DEBUG][Resource Id #77] INSERT INTO products
(name, price) VALUES ('Hot pepper', 3.50)
```

Implementing your own Logger

You can implement your own logger class for database queries, by creating a class that implements a single method called "log". The method needs to accept a string as the first argument. You can then pass your logging object to Phalcon\Db::setLogger(), and from then on any SQL statement executed will call that method to log the results.

2.42.10 Describing Tables and Databases

Phalcon\Db also provides methods to retrieve detailed information about tables and databases.

```
<?php

// Get tables on the test_db database
$tables = $connection->listTables("test_db");

// Is there a table robots in the database?
$exists = $connection->tableExists("robots");

// Get name, data types and special features of robots fields
$fields = $connection->describeColumns("robots");
foreach ($fields as $field) {
    echo "Column Type: ", $field["Type"];
}

// Get indexes on the robots table
$indexes = $connection->describeIndexes("robots");
foreach ($indexes as $index) {
    print_r($index->getColumns());
}

// Get foreign keys on the robots table
$references = $connection->describeReferences("robots");
foreach ($references as $reference) {
    // Print referenced columns
    print_r($reference->getReferencedColumns());
}
```

A table description is very similar to the MySQL describe command, it contains the following information:

Index	Description
Field	Field's name
Type	Column Type
Key	Is the column part of the primary key or an index?
Null	Does the column allow null values?

2.42.11 Creating/Altering/Dropping Tables

Different database systems (MySQL, Postgresql etc.) offer the ability to create, alter or drop tables with the use of commands such as CREATE, ALTER or DROP. The SQL syntax differs based on which database system is used. *Phalcon\Db* offers a unified interface to alter tables, without the need to differentiate the SQL syntax based on the target storage system.

Creating Tables

The following example shows how to create a table:

```
<?php

use \Phalcon\Db\Column as Column;

$connection->createTable(
    "robots",
    null,
    array(
       "columns" => array(
            new Column(
                "id",
                array(
                    "type"          => Column::TYPE_INTEGER,
                    "size"          => 10,
                    "notNull"       => true,
                    "autoIncrement" => true,
                )
            ),
            new Column(
                "name",
                array(
                    "type"    => Column::TYPE_VARCHAR,
                    "size"    => 70,
                    "notNull" => true,
                )
            ),
            new Column(
                "year",
                array(
                    "type"    => Column::TYPE_INTEGER,
                    "size"    => 11,
                    "notNull" => true,
                )
            )
        )
    )
);
```

Phalcon\Db::createTable() accepts an associative array describing the table. Columns are defined with the class *Phalcon\Db\Column*. The table below shows the options available to define a column:

Option	Description	Optional
"type"	Column type. Must be a Phalcon\Db\Column constant (see below for a list)	No
"primary"	True if the table is part of the table's primary key	Yes
"size"	Some type of columns like VARCHAR or INTEGER may have a specific size	Yes
"scale"	DECIMAL or NUMBER columns may be have a scale to specify how many decimals should be stored	Yes
"unsigned"	INTEGER columns may be signed or unsigned. This option does not apply to other types of columns	Yes
"notNull"	Column can store null values?	Yes
"autoIncrement"	With this attribute column will filled automatically with an auto-increment integer. Only one column in the table can have this attribute.	Yes
"bind"	One of the BIND_TYPE_* constants telling how the column must be binded before save it	Yes
"first"	Column must be placed at first position in the column order	Yes
"after"	Column must be placed after indicated column	Yes

Phalcon\Db supports the following database column types:

- Phalcon\DbColumn::TYPE_INTEGER
- Phalcon\DbColumn::TYPE_DATE
- Phalcon\Db\Column::TYPE_VARCHAR
- Phalcon\Db\Column::TYPE_DECIMAL
- Phalcon\Db\Column::TYPE_DATETIME
- Phalcon\Db\Column::TYPE_CHAR
- Phalcon\Db\Column::TYPE_TEXT

The associative array passed in Phalcon\Db::createTable() can have the possible keys:

Index	Description	Optional
"columns"	An array with a set of table columns defined with *Phalcon\Db\Column*	No
"indexes"	An array with a set of table indexes defined with *Phalcon\Db\Index*	Yes
"references"	An array with a set of table references (foreign keys) defined with *Phalcon\Db\Reference*	Yes
"options"	An array with a set of table creation options. These options often relate to the database system in which the migration was generated.	Yes

Altering Tables

As your application grows, you might need to alter your database, as part of a refactoring or adding new features. Not all database systems allow to modify existing columns or add columns between two existing ones. *Phalcon\Db* is limited by these constraints.

```
<?php

use \Phalcon\Db\Column as Column;

// Adding a new column
$connection->addColumn(
    "robots",
    null,
    new Column(
```

```
        "robot_type",
        array(
            "type"    => Column::TYPE_VARCHAR,
            "size"    => 32,
            "notNull" => true,
            "after"   => "name",
        )
    )
);

// Modifying an existing column
$connection->modifyColumn(
    "robots",
    null,
    new Column(
        "name",
        array(
            "type" => Column::TYPE_VARCHAR,
            "size" => 40,
            "notNull" => true,
        )
    )
);

// Deleting the column "name"
$connection->deleteColumn("robots", null, "name");
```

Dropping Tables

Examples on dropping tables:

```
<?php

// Drop table robot from active database
$connection->dropTable("robots");

//Drop table robot from database "machines"
$connection->dropTable("robots", "machines");
```

2.43 Internationalization

Phalcon is written in C as an extension for PHP. There is a PECL extension that offers internationalization functions to PHP applications called intl. Starting from PHP 5.4 this extension is bundled with PHP. Its documentation can be found in the pages of the official PHP manual.

Phalcon does not offer this functionality, since creating such a component would be replicating existing code.

In the examples below, we will show you how to implement the intl extension's functionality into Phalcon powered applications.

> This guide is not intended to be a complete documentation of the intl extension. Please visit its the documentation of the extension for a reference.

2.43.1 Find out best available Locale

There are several ways to find out the best available locale using intl. One of them is to check the HTTP "Accept-Language" header:

```
<?php

$locale = Locale::acceptFromHttp($_SERVER["HTTP_ACCEPT_LANGUAGE"]);

// Locale could be something like "en_GB" or "en"
echo $locale;
```

Below method returns a locale identified. It is used to get language, culture, or regionally-specific behavior from the Locale API. Examples of identifiers include:

- en-US (English, United States)
- zh-Hant-TW (Chinese, Traditional Script, Taiwan)
- fr-CA, fr-FR (French for Canada and France respectively)

2.43.2 Formatting messages based on Locale

Part of creating a localized application is to produce concatenated, language-neutral messages. The MessageFormatter allows for the production of those messages.

Printing numbers formatted based on some locale:

```
<?php

// Prints € 4 560
$formatter = new MessageFormatter("fr_FR", "€ {0, number, integer}");
echo $formatter->format(array(4560));

// Prints USD$ 4,560.5
$formatter = new MessageFormatter("en_US", "USD$ {0, number}");
echo $formatter->format(array(4560.50));

// Prints ARS$ 1.250,25
$formatter = new MessageFormatter("es_AR", "ARS$ {0, number}");
echo $formatter->format(array(1250.25));
```

Message formatting using time and date patterns:

```
<?php

//Setting parameters
$time   = time();
$values = array(7, $time, $time);

// Prints "At 3:50:31 PM on Apr 19, 2012, there was a disturbance on planet 7."
$pattern   = "At {1, time} on {1, date}, there was a disturbance on planet {0, number}.";
$formatter = new MessageFormatter("en_US", $pattern);
echo $formatter->format($values);

// Prints "À 15:53:01 le 19 avr. 2012, il y avait une perturbation sur la planète 7."
$pattern   = "À {1, time} le {1, date}, il y avait une perturbation sur la planète {0, number}.";
$formatter = new MessageFormatter("fr_FR", $pattern);
echo $formatter->format($values);
```

2.43.3 Locale-Sensitive comparison

The Collator class provides string comparison capability with support for appropriate locale-sensitive sort orderings. Check the examples below on the usage of this class:

```
<?php

// Create a collator using Spanish locale
$collator = new Collator("es");

// Returns that the strings are equal, in spite of the emphasis on the "o"
$collator->setStrength(Collator::PRIMARY);
var_dump($collator->compare("una canción", "una cancion"));

// Returns that the strings are not equal
$collator->setStrength(Collator::DEFAULT);
var_dump($collator->compare("una canción", "una cancion"));
```

2.43.4 Transliteration

Transliterator provides transliteration of strings:

```
<?php

$id = "Any-Latin; NFD; [:Nonspacing Mark:] Remove; NFC; [:Punctuation:] Remove; Lower();";
$transliterator = Transliterator::create($id);

$string = "garçon-étudiant-où-L'école";
echo $transliterator->transliterate($string); // garconetudiantoulecole
```

2.44 Database Migrations

Migrations are a convenient way for you to alter your database in a structured and organized manner.

> **Important:** Migrations are available on *Phalcon Developer Tools* You need at least Phalcon Framework version 0.5.0 to use developer tools. Also is recommended to have PHP 5.3.11 or greater installed.

Often in development we need to update changes in production environments. Some of these changes could be database modifications like new fields, new tables, removing indexes, etc.

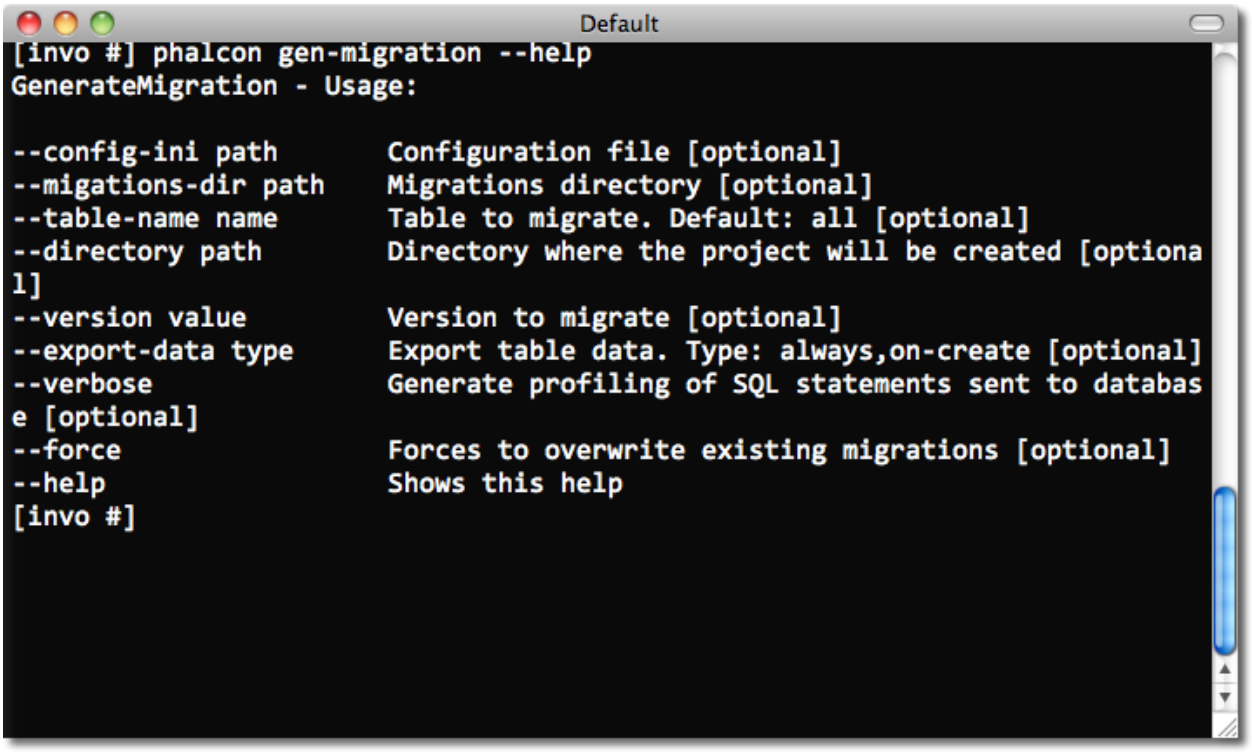
When a migration is generated a set of classes are created to describe how your database is structured at that moment. These classes can be used to synchronize the schema structure on remote databases setting your database ready to work with the new changes that your application implements. Migrations describe these transformations using plain PHP.

2.44.1 Schema Dumping

The *Phalcon Developer Tools* provides scripts to manage migrations (generation, running and rollback).

The available options for generating migrations are:

Running this script without any parameters will simply dump every object (tables and views) from your database in migration classes.



Each migration has a version identifier associated to it. The version number allows us to identify if the migration is newer or older than the current 'version' of our database. Versions also inform Phalcon of the running order when executing a migration.

When a migration is generated, instructions are displayed on the console to describe the different steps of the migration and the execution time of those statements. At the end, a migration version is generated.

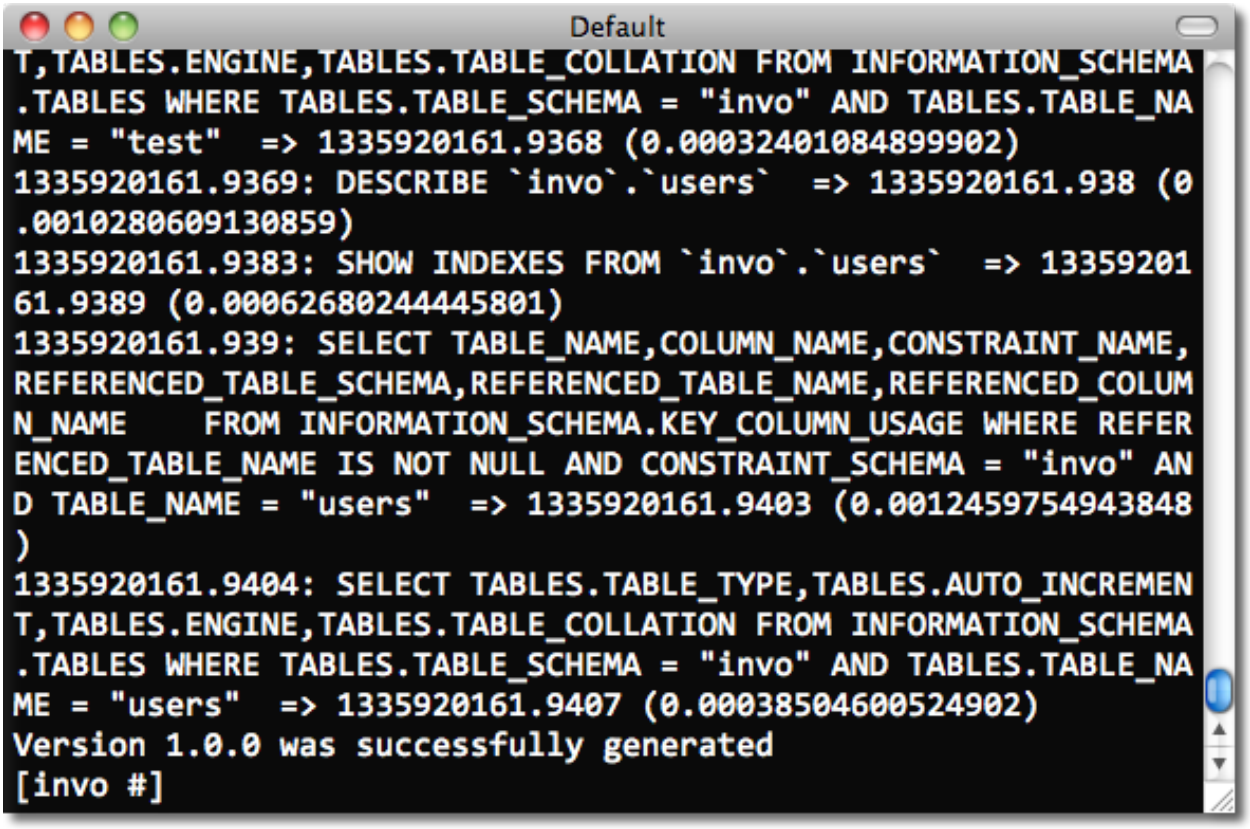
By default *Phalcon Developer Tools* use the *app/migrations* directory to dump the migration files. You can change the location by setting one of the parameters on the generation script. Each table in the database has its respective class generated in a separated file under a directory referring its version:

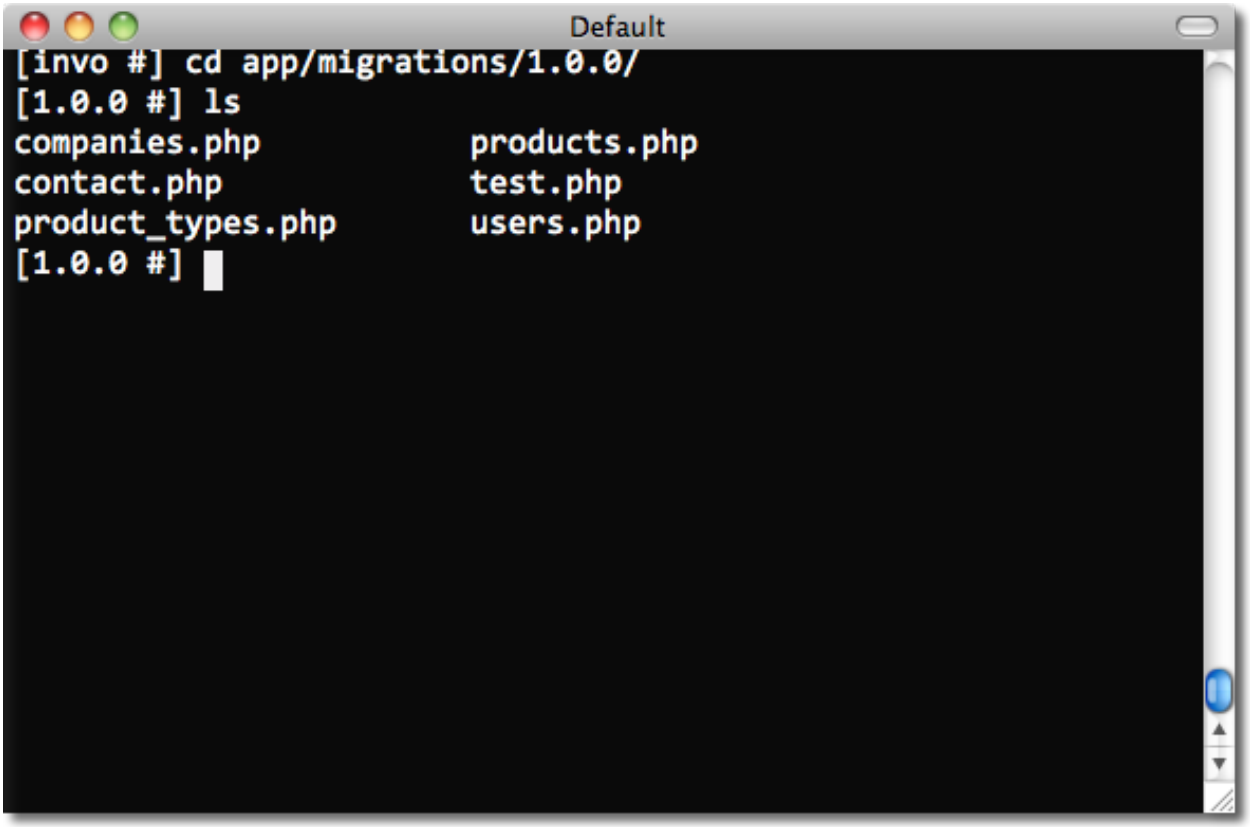
2.44.2 Migration Class Anatomy

Each file contains a unique class that extends the Phalcon\Mvc\Model\Migration These classes normally have two methods: up() and down(). Up() performs the migration, while down() rolls it back.

Up() also contains the *magic* method morphTable(). The magic comes when it recognizes the changes needed to synchronize the actual table in the database to the description given.

```
<?php

use Phalcon\Db\Column as Column;
use Phalcon\Db\Index as Index;
use Phalcon\Db\Reference as Reference;

class ProductsMigration_100 extends \Phalcon\Mvc\Model\Migration
{

    public function up()
    {
        $this->morphTable(
```


Default
T,TABLES.ENGINE,TABLES.TABLE_COLLATION FROM INFORMATION_SCHEMA
.TABLES WHERE TABLES.TABLE_SCHEMA = "invo" AND TABLES.TABLE_NA
ME = "test" => 1335920161.9368 (0.00032401084899902)
1335920161.9369: DESCRIBE `invo`.`users` => 1335920161.938 (0
.0010280609130859)
1335920161.9383: SHOW INDEXES FROM `invo`.`users` => 13359201
61.9389 (0.00062680244445801)
1335920161.939: SELECT TABLE_NAME,COLUMN_NAME,CONSTRAINT_NAME,
REFERENCED_TABLE_SCHEMA,REFERENCED_TABLE_NAME,REFERENCED_COLUM
N_NAME   FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE WHERE REFER
ENCED_TABLE_NAME IS NOT NULL AND CONSTRAINT_SCHEMA = "invo" AN
D TABLE_NAME = "users" => 1335920161.9403 (0.0012459754943848
)
1335920161.9404: SELECT TABLES.TABLE_TYPE,TABLES.AUTO_INCREMEN
T,TABLES.ENGINE,TABLES.TABLE_COLLATION FROM INFORMATION_SCHEMA
.TABLES WHERE TABLES.TABLE_SCHEMA = "invo" AND TABLES.TABLE_NA
ME = "users" => 1335920161.9407 (0.00038504600524902)
Version 1.0.0 was successfully generated
[invo #]


Default
[invo #] cd app/migrations/1.0.0/
[1.0.0 #] ls
companies.php
contact.php
product_types.php
products.php
test.php
users.php
[1.0.0 #]

```
    "products",
    array(
        "columns" => array(
            new Column(
                "id",
                array(
                    "type"          => Column::TYPE_INTEGER,
                    "size"          => 10,
                    "unsigned"      => true,
                    "notNull"       => true,
                    "autoIncrement" => true,
                    "first"         => true,
                )
            ),
            new Column(
                "product_types_id",
                array(
                    "type"     => Column::TYPE_INTEGER,
                    "size"     => 10,
                    "unsigned" => true,
                    "notNull"  => true,
                    "after"    => "id",
                )
            ),
            new Column(
                "name",
                array(
                    "type"    => Column::TYPE_VARCHAR,
                    "size"    => 70,
                    "notNull" => true,
                    "after"   => "product_types_id",
                )
            ),
            new Column(
                "price",
                array(
                    "type"    => Column::TYPE_DECIMAL,
                    "size"    => 16,
                    "scale"   => 2,
                    "notNull" => true,
                    "after"   => "name",
                )
            ),
        ),
        "indexes" => array(
            new Index(
                "PRIMARY",
                array("id")
            ),
            new Index(
                "product_types_id",
                array("product_types_id")
            )
        ),
        "references" => array(
            new Reference(
                "products_ibfk_1",
                array(
```

```
                        "referencedSchema"  => "invo",
                        "referencedTable"   => "product_types",
                        "columns"           => array("product_types_id"),
                        "referencedColumns" => array("id"),
                    )
                )
            ),
            "options" => array(
                "TABLE_TYPE"      => "BASE TABLE",
                "ENGINE"          => "InnoDB",
                "TABLE_COLLATION" => "utf8_general_ci",
            )
        )
    );
    }

}
```

The class is called "ProductsMigration_100". Suffix 100 refers to the version 1.0.0. morphTable() receives an associative array with 4 possible sections:

Index	Description	Optional
"columns"	An array with a set of table columns	No
"indexes"	An array with a set of table indexes.	Yes
"references"	An array with a set of table references (foreign keys).	Yes
"options"	An array with a set of table creation options. These options are often related to the database system in which the migration was generated.	Yes

Defining Columns

Phalcon\Db\Column is used to define table columns. It encapsulates a wide variety of column related features. Its constructor receives as first parameter the column name and an array describing the column. The following options are available when describing columns:

Option	Description	Optional
"type"	Column type. Must be a *Phalcon_Db_Column* constant (see below)	No
"size"	Some type of columns like VARCHAR or INTEGER may have a specific size	Yes
"scale"	DECIMAL or NUMBER columns may be have a scale to specify how much decimals it must store	Yes
"unsigned"	INTEGER columns may be signed or unsigned. This option does not apply to other types of columns	Yes
"notNull"	Column can store null values?	Yes
"autoIncrement"	With this attribute column will filled automatically with an auto-increment integer. Only one column in the table can have this attribute.	Yes
"first"	Column must be placed at first position in the column order	Yes
"after"	Column must be placed after indicated column	Yes

Database migrations support the following database column types:

- Phalcon\Db\Column::TYPE_INTEGER
- Phalcon\Db\Column::TYPE_DATE

- Phalcon\Db\Column::TYPE_VARCHAR
- Phalcon\Db\Column::TYPE_DECIMAL
- Phalcon\Db\Column::TYPE_DATETIME
- Phalcon\Db\Column::TYPE_CHAR
- Phalcon\Db\Column::TYPE_TEXT

Defining Indexes

Phalcon\Db\Index defines table indexes. An index only requires that you define a name for it and a list of its columns. Note that if any index has the name PRIMARY, Phalcon will create a primary key index in that table.

Defining References

Phalcon\Db\Reference defines table references (also called foreign keys). The following options can be used to define a reference:

Index	Description	Optional
"referencedTable"	It's auto-descriptive. It refers to the name of the referenced table.	No
"columns"	An array with the name of the columns at the table that have the reference	No
"referencedColumns"	An array with the name of the columns at the referenced table	No
"referencedTable"	The referenced table maybe is on another schema or database. This option allows you to define that.	Yes

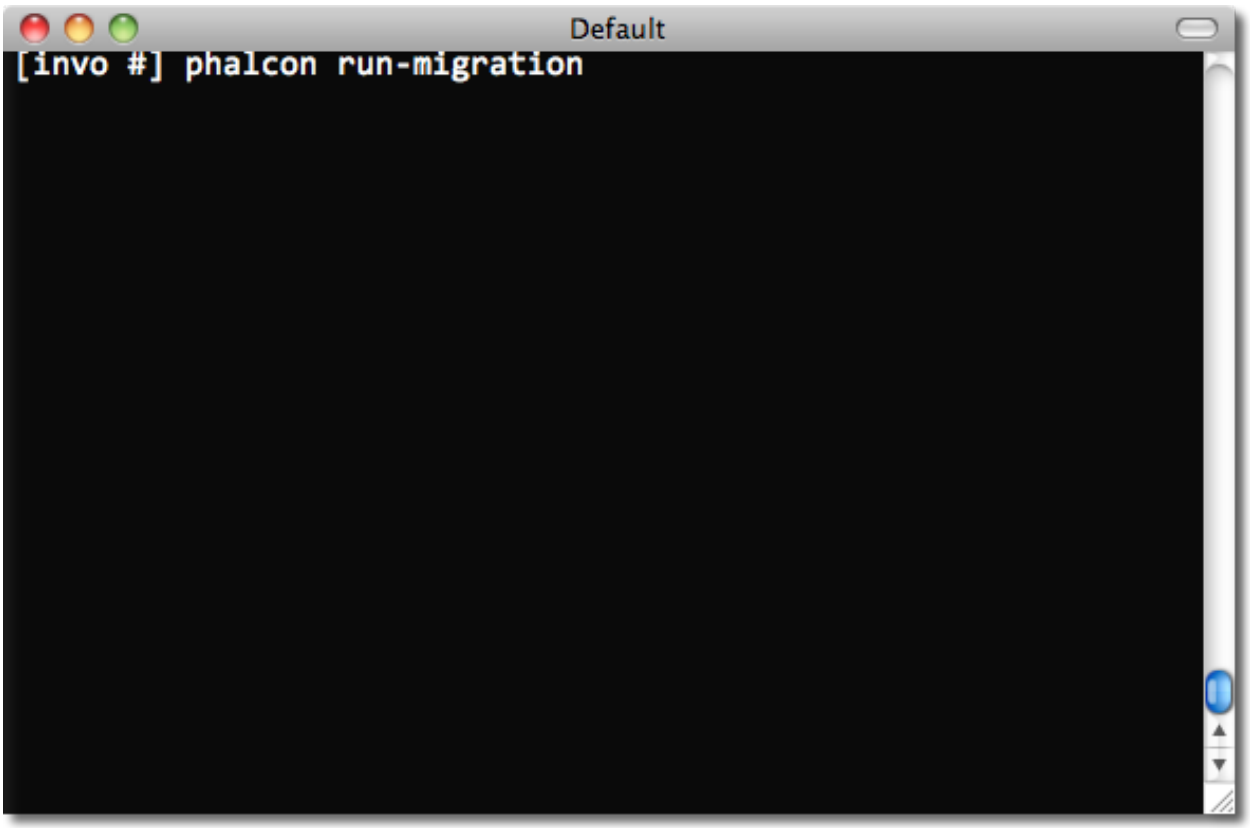
2.44.3 Writing Migrations

Migrations aren't only designed to "morph" table. A migration is just a regular PHP class so you're not limited to these functions. For example after adding a column you could write code to set the value of that column for existing records. For more details and examples of individual methods, check the *database component*.

```
<?php

class ProductsMigration_100 extends \Phalcon\Mvc\Model\Migration
{

    public function up()
    {
        //...
        self::$_connection->insert(
            "products",
            array("Malabar spinach", 14.50),
            array("name", "price")
        );
    }

}
```

2.44.4 Running Migrations

Once the generated migrations are uploaded on the target server, you can easily run them as shown in the following example:



Depending on how outdated is the database with respect to migrations, Phalcon may run multiple migration versions in the same migration process. If you specify a target version, Phalcon will run the required migrations until it reaches the specified version.

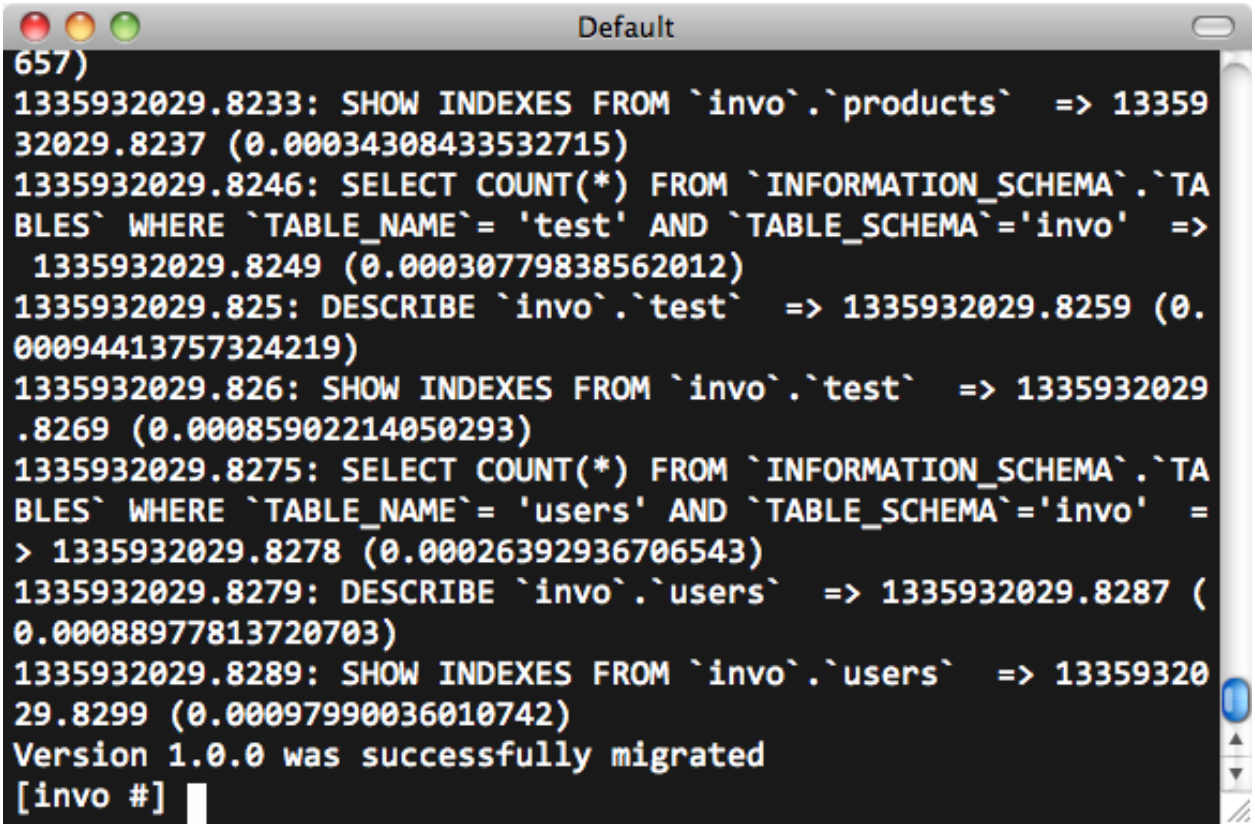
2.45 Debugging Applications

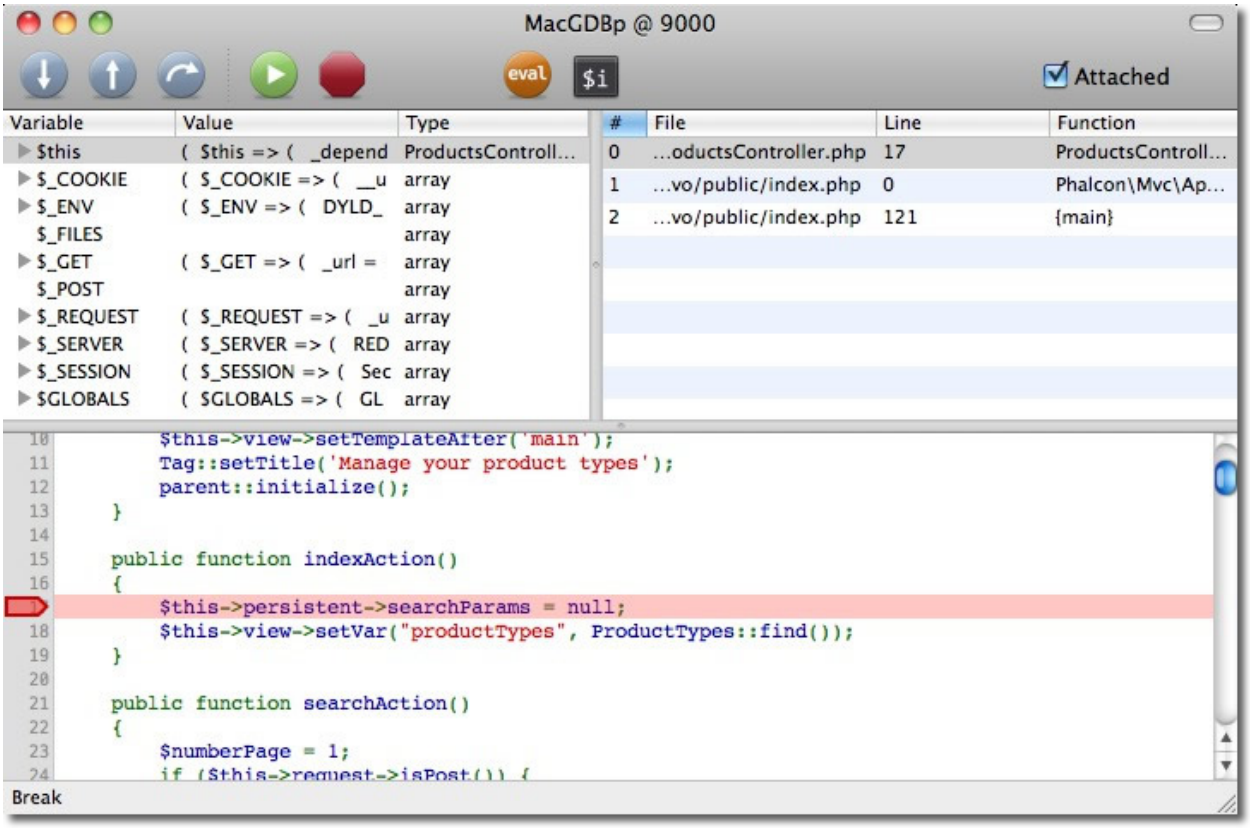
PHP offers tools to debug applications with notices, warnings, errors and exceptions. The Exception class offers information such as the file, line, message, numeric code, backtrace etc. on where an error occurred. OOP frameworks like Phalcon mainly use this class to encapsulate this functionality and provide information back to the developer or user.

Despite being written in C, Phalcon runs methods in the PHP userland, providing the debug capability that any other application or framework written in PHP has.

2.45.1 Catching Exceptions

Throughout the tutorials and examples of the Phalcon documentation, there is a common element that is catching exceptions. This is a try/catch block:


Default
657)
1335932029.8233: SHOW INDEXES FROM `invo`.`products` => 13359
32029.8237 (0.00034308433532715)
1335932029.8246: SELECT COUNT(*) FROM `INFORMATION_SCHEMA`.`TA
BLES` WHERE `TABLE_NAME`= 'test' AND `TABLE_SCHEMA`='invo' =>
1335932029.8249 (0.00030779838562012)
1335932029.825: DESCRIBE `invo`.`test` => 1335932029.8259 (0.
00094413757324219)
1335932029.826: SHOW INDEXES FROM `invo`.`test` => 1335932029
.8269 (0.00085902214050293)
1335932029.8275: SELECT COUNT(*) FROM `INFORMATION_SCHEMA`.`TA
BLES` WHERE `TABLE_NAME`= 'users' AND `TABLE_SCHEMA`='invo' =
> 1335932029.8278 (0.00026392936706543)
1335932029.8279: DESCRIBE `invo`.`users` => 1335932029.8287 (
0.00088977813720703)
1335932029.8289: SHOW INDEXES FROM `invo`.`users` => 13359320
29.8299 (0.00097990036010742)
Version 1.0.0 was successfully migrated
[invo #]


MacGDBp @ 9000
eval
$i
Attached
Variable
Value
Type
$this
$_COOKIE
$_ENV
$_FILES
$_GET
$_POST
$_REQUEST
$_SERVER
$_SESSION
$GLOBALS
#
File
Line
Function
0 ...oductsController.php 17 ProductsControll...
1 ...vo/public/index.php 0 Phalcon\Mvc\Ap...
2 ...vo/public/index.php 121 {main}
$this->view->setTemplateAfter('main');
Tag::setTitle('Manage your product types');
parent::initialize();
public function indexAction()
$this->persistent->searchParams = null;
$this->view->setVar("productTypes", ProductTypes::find());
public function searchAction()
$numberPage = 1;
Break

```
<?php

try {

    //... some phalcon code

} catch(\Phalcon\Exception $e) {

}
```

Any exception thrown within the block is captured in the variable $e. A *Phalcon\Exception* extends the PHP Exception class and is used to understand whether the exception came from Phalcon or PHP itself.

All exceptions generated by PHP are based on the Exception class, and have at least the following elements:

```
<?php

class Exception
{

    /* Properties */
    protected string $message;
    protected int $code;
    protected string $file;
    protected int $line;

    /* Methods */
    public __construct ([ string $message = "" [, int $code = 0 [, Exception $previous = NULL ]]])
    final public string getMessage ( void )
    final public Exception getPrevious ( void )
    final public mixed getCode ( void )
    final public string getFile ( void )
    final public int getLine ( void )
    final public array getTrace ( void )
    final public string getTraceAsString ( void )
    public string __toString ( void )
    final private void __clone ( void )

}
```

Retrieving information from *Phalcon\Exception* is the same as PHP's Exception class:

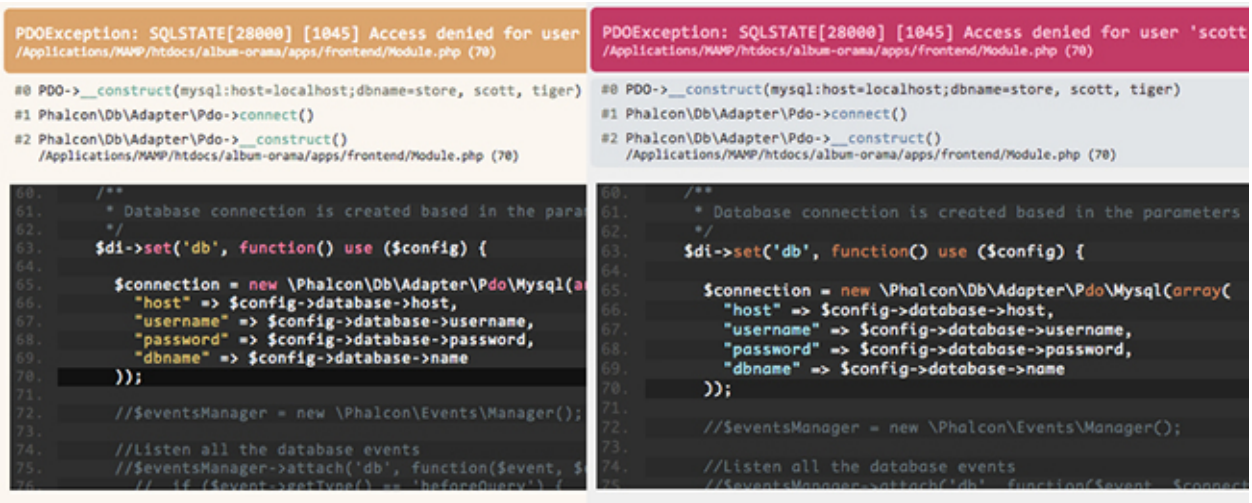
```
<?php

try {

    //... app code ...

} catch(\Phalcon\Exception $e) {
    echo get_class($e), ": ", $e->getMessage(), "\n";
    echo " File=", $e->getFile(), "\n";
    echo " Line=", $e->getLine(), "\n";
    echo $e->getTraceAsString();
}
```

It's therefore easy to find which file and line of the application's code generated the exception, as well as the components involved in generating the exception:

```
PDOException: SQLSTATE[28000] [1045] Access denied for user 'root'@'localhost'
    (using password: NO)
 File=/Applications/MAMP/htdocs/invo/public/index.php
 Line=74
#0 [internal function]: PDO->__construct('mysql:host=loca...', 'root', '', Array)
#1 [internal function]: Phalcon\Db\Adapter\Pdo->connect(Array)
#2 /Applications/MAMP/htdocs/invo/public/index.php(74):
    Phalcon\Db\Adapter\Pdo->__construct(Array)
#3 [internal function]: {closure}()
#4 [internal function]: call_user_func_array(Object(Closure), Array)
#5 [internal function]: Phalcon\DI->_factory(Object(Closure), Array)
#6 [internal function]: Phalcon\DI->get('db', Array)
#7 [internal function]: Phalcon\DI->getShared('db')
#8 [internal function]: Phalcon\Mvc\Model->getConnection()
#9 [internal function]: Phalcon\Mvc\Model::_getOrCreateResultset('Users', Array, true)
#10 /Applications/MAMP/htdocs/invo/app/controllers/SessionController.php(83):
    Phalcon\Mvc\Model::findFirst('email='demo@pha...')
#11 [internal function]: SessionController->startAction()
#12 [internal function]: call_user_func_array(Array, Array)
#13 [internal function]: Phalcon\Mvc\Dispatcher->dispatch()
#14 /Applications/MAMP/htdocs/invo/public/index.php(114): Phalcon\Mvc\Application->handle()
#15 {main}
```

As you can see from the above output the Phalcon's classes and methods are displayed just like any other component, and even showing the parameters that were invoked in every call. The method Exception::getTrace provides additional information if needed.

By installing the 'Pretty Exceptions' utility in your appplication you can print exceptions with a nicely presentation:



2.45.2 Reflection and Instrospection

Any instance of a Phalcon class offers exactly the same behavior than a PHP normal one. It's possible to use the Reflection API or simply print any object to show how is its internal state:

```
<?php

$router = new Phalcon\Mvc\Router();
print_r($router);
```

It's easy to know the internal state of any object. The above example prints the following:

```
Phalcon\Mvc\Router Object
(
    [_dependencyInjector:protected] =>
    [_module:protected] =>
    [_controller:protected] =>
    [_action:protected] =>
    [_params:protected] => Array
        (
        )
    [_routes:protected] => Array
        (
            [0] => Phalcon\Mvc\Router\Route Object
                (
                    [_pattern:protected] => #^/([a-zA-Z0-9\_]+)[/]{0,1}$#
                    [_compiledPattern:protected] => #^/([a-zA-Z0-9\_]+)[/]{0,1}$#
                    [_paths:protected] => Array
                        (
                            [controller] => 1
                        )

                    [_methods:protected] =>
                    [_id:protected] => 0
                    [_name:protected] =>
                )

            [1] => Phalcon\Mvc\Router\Route Object
                (
                    [_pattern:protected] => #^/([a-zA-Z0-9\_]+)/([a-zA-Z0-9\_]+)(/.*)*$#
                    [_compiledPattern:protected] => #^/([a-zA-Z0-9\_]+)/([a-zA-Z0-9\_]+)(/.*)*$#
                    [_paths:protected] => Array
                        (
                            [controller] => 1
                            [action] => 2
                            [params] => 3
                        )
                    [_methods:protected] =>
                    [_id:protected] => 1
                    [_name:protected] =>
                )
        )
    [_matchedRoute:protected] =>
    [_matches:protected] =>
    [_wasMatched:protected] =>
    [_defaultModule:protected] =>
    [_defaultController:protected] =>
    [_defaultAction:protected] =>
    [_defaultParams:protected] => Array
        (
        )
)
```

2.45.3 Using XDebug

XDebug is an amazing tool that complements the debugging of PHP applications. It is also a C extension for PHP, and you can use it together with Phalcon without additional configuration or side effects.

Once you have xdebug installed, you can use its API to get a more detailed information about exceptions and messages. The following example implements xdebug_print_function_stack to stop the execution and generate a backtrace:

```
<?php

class SignupController extends \Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

    public function registerAction()
    {

        // Request variables from html form
        $name  = $this->request->getPost("name", "string");
        $email = $this->request->getPost("email", "email");

        // Stop execution and show a backtrace
        return xdebug_print_function_stack("stop here!");

        $user        = new Users();
        $user->name  = $name;
        $user->email = $email;

        // Store and check for errors
        $user->save();
    }

}
```

In this instance, Xdebug will also show us the variables in the local scope, and a backtrace as well:

```
Xdebug: stop here! in /Applications/MAMP/htdocs/tutorial/app/controllers/SignupController.php
    on line 19

Call Stack:
    0.0383     654600   1. {main}() /Applications/MAMP/htdocs/tutorial/public/index.php:0
    0.0392     663864   2. Phalcon\Mvc\Application->handle()
        /Applications/MAMP/htdocs/tutorial/public/index.php:37
    0.0418     738848   3. SignupController->registerAction()
        /Applications/MAMP/htdocs/tutorial/public/index.php:0
    0.0419     740144   4. xdebug_print_function_stack()
        /Applications/MAMP/htdocs/tutorial/app/controllers/SignupController.php:19
```

Xdebug provides several ways to get debug and trace information regarding the execution of your application using Phalcon. You can check the XDebug documentation for more information.

2.46 Phalcon Developer Tools

These tools are a collection of useful scripts to generate skeleton code. Core components of your application can be generated with a simple command, allowing you to easily develop applications using Phalcon.

> **Important:** Phalcon Framework version 0.5.0 or greater is needed to use developer tools. It is highly recommended to use PHP 5.3.6 or greater. If you prefer to use the web version instead of the console, this

blog post offers more information.

2.46.1 Download

You can download or clone a cross platform package containing the developer tools from Github.

Installation

These are detailed instructions on how to install the developer tools on different platforms:

Phalcon Developer Tools on Windows

These steps will guide you through the process of installing Phalcon Developer Tools for Windows.

Prerequisites The Phalcon PHP extension is required to run Phalcon Tools. If you haven't installed it yet, please see the *Installation* section for instructions.

Download You can download a cross platform package containing the developer tools from the Download section. Also you can clone it from Github.

On the Windows platform, you need to configure the system PATH to include Phalcon tools as well as the PHP executable. If you download the Phalcon tools as a zip archive, extract it on any path of your local drive i.e. *c:\phalcon-tools*. You will need this path in the steps below. Edit the file "phalcon.bat" by right clicking on the file and selecting "Edit":

Change the path to the one you installed the Phalcon tools:

Save the changes.

Adding PHP and Tools to your system PATH Because the scripts are written in PHP, you need to install it on your machine. Depending on your PHP installation, the executable can be located in various places. Search for the file php.exe and copy the path it is located in. For instance, if using the latest WAMP stack, PHP is located in: *C:\wampbin\php\php5.3.10\php.exe*.

From the Windows start menu, right mouse click on the "My Computer" icon and select "Properties":

Click the "Advanced" tab and then the button "Environment Variables":

At the bottom, look for the section "System variables" and edit the variable "Path":

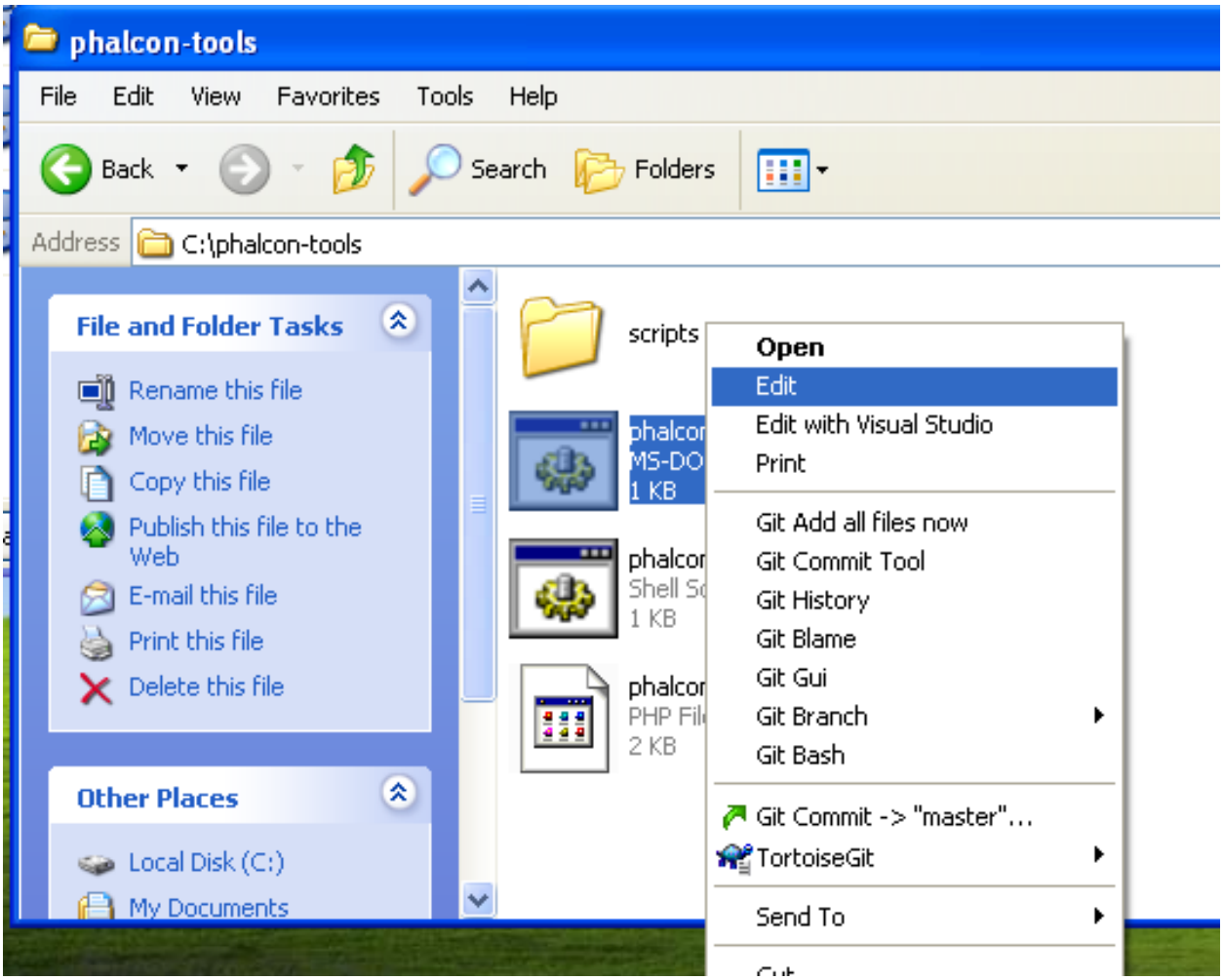
Be very careful on this step! You need to append at the end of the long string the path where your php.exe was located and the path where Phalcon tools are installed. Use the ";" character to separate the differents paths in the variable:

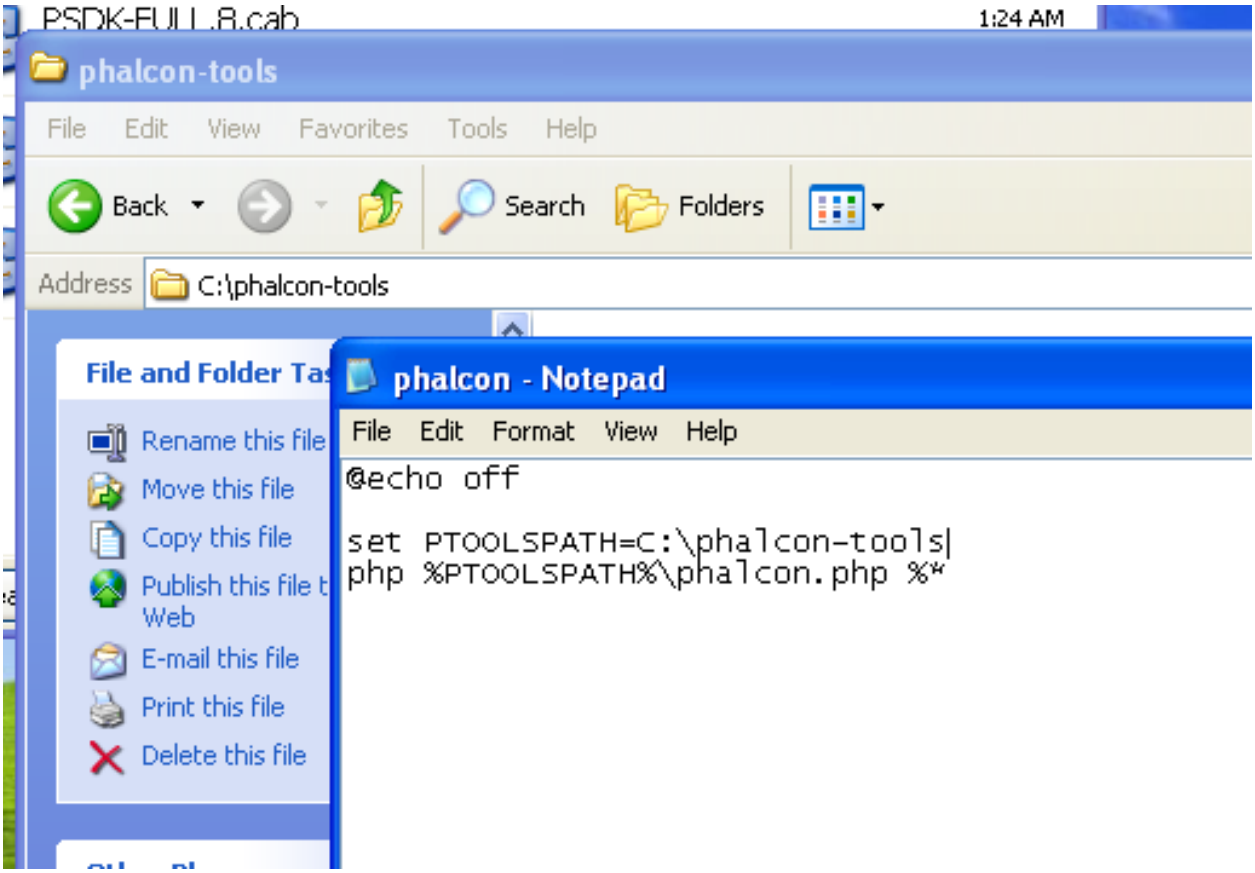
Accept the changes made by clicking "OK" and close the dialogs opened. From the start menu click on the option "Run". If you can't find this option, press "Windows Key" + "R".

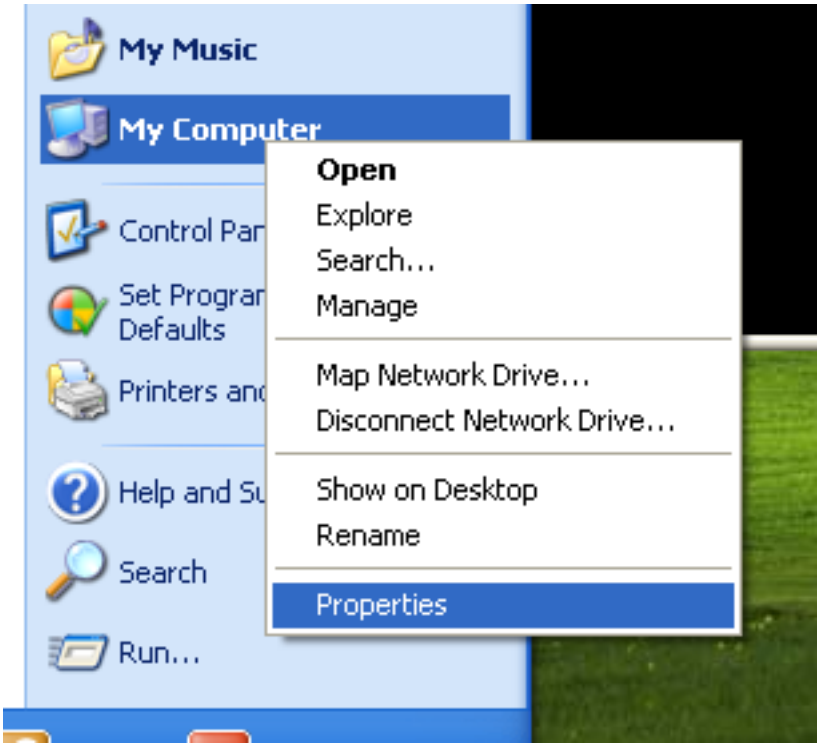
Type "cmd" and press enter to open the windows command line utility:

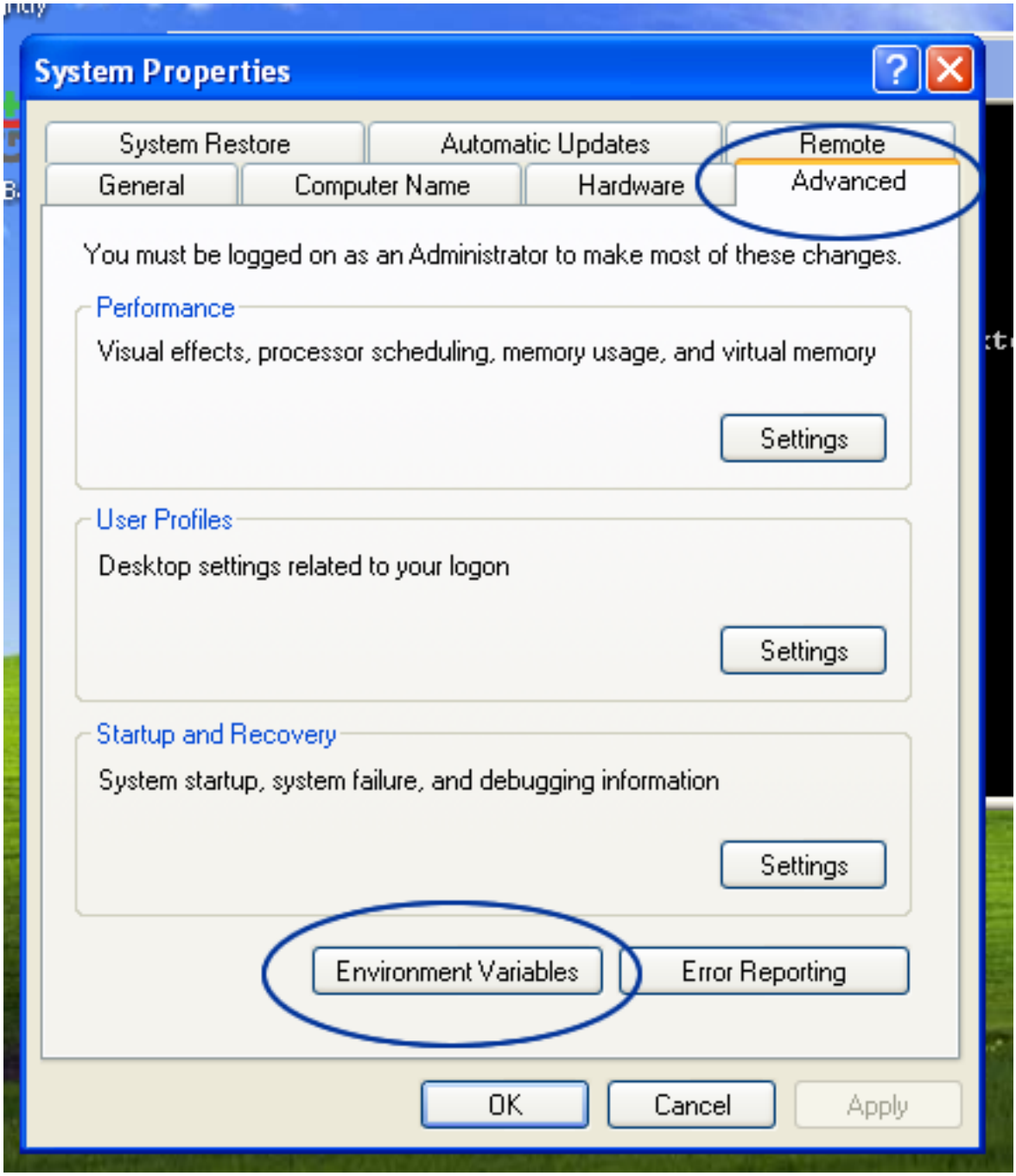
Type the commands "php -v" and "phalcon" and you will see something like this:

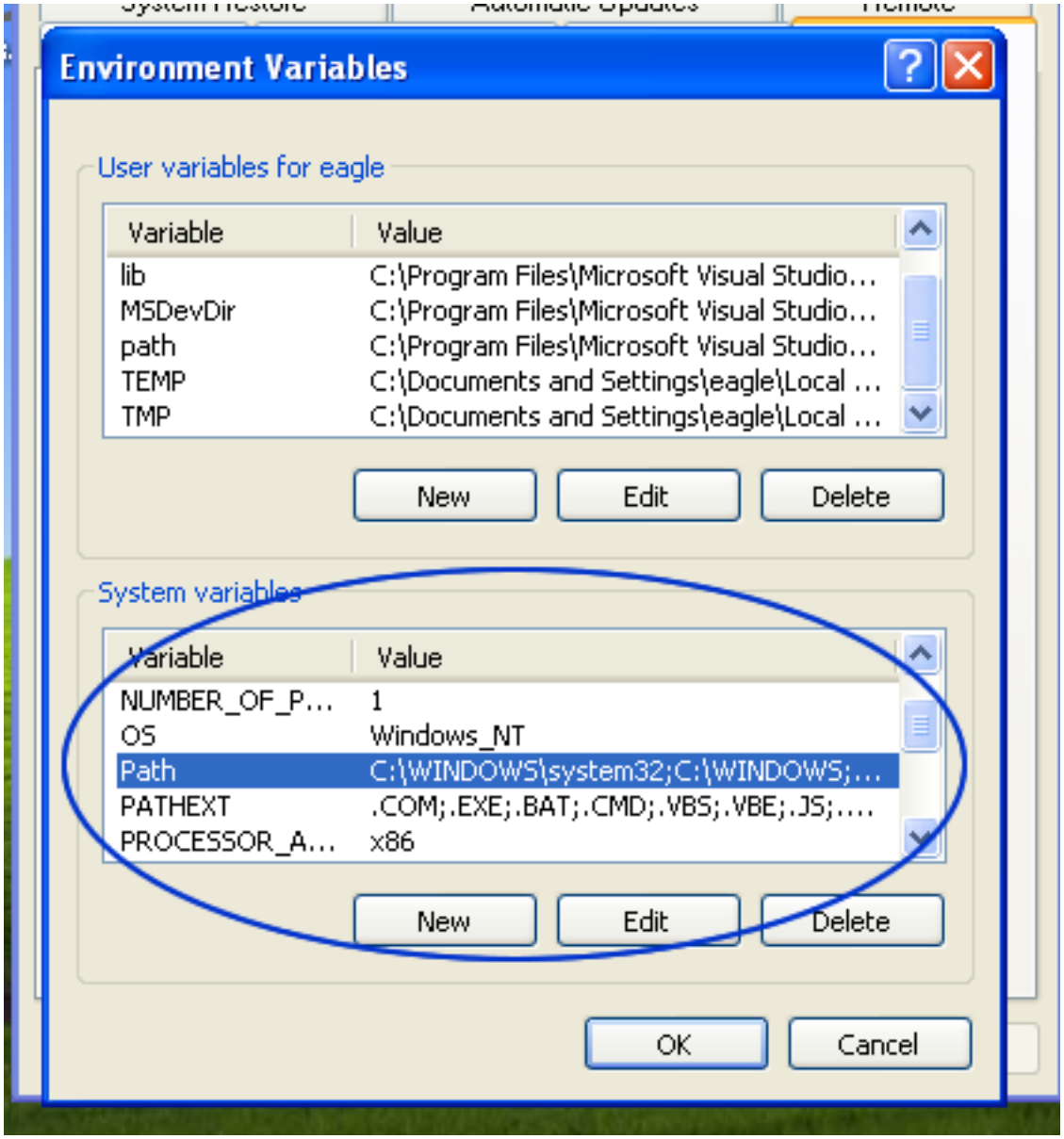
Congratulations you now have Phalcon tools installed!


phalcon-tools
File Edit View Favorites Tools Help
Back Search Folders
Address C:\phalcon-tools
File and Folder Tasks
Rename this file
Move this file
Copy this file
Publish this file to the Web
E-mail this file
Print this file
Delete this file
Other Places
Local Disk (C:)
My Documents
scripts
1 KB
Shell Sc
1 KB
PHP Fil
2 KB
Open
Edit
Edit with Visual Studio
Print
Git Add all files now
Git Commit Tool
Git History
Git Blame
Git Gui
Git Branch
Git Bash
Git Commit -> "master"...
TortoiseGit
Send To

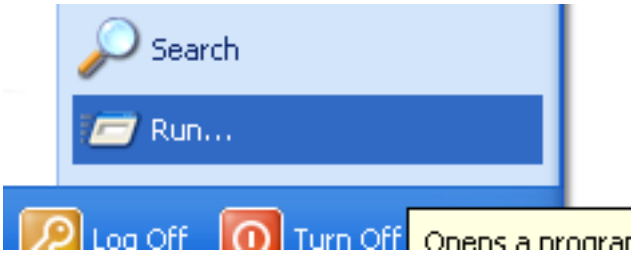

PSDK-FULL.8.cab
1:24 AM
phalcon-tools
File Edit View Favorites Tools Help
Back
Search
Folders
Address
C:\phalcon-tools
File and Folder Tas
Rename this file
Move this file
Copy this file
Publish this file t
Web
E-mail this file
Print this file
Delete this file
phalcon - Notepad
File Edit Format View Help
@echo off
set PTOOLSPATH=C:\phalcon-tools
php %PTOOLSPATH%\phalcon.php %*

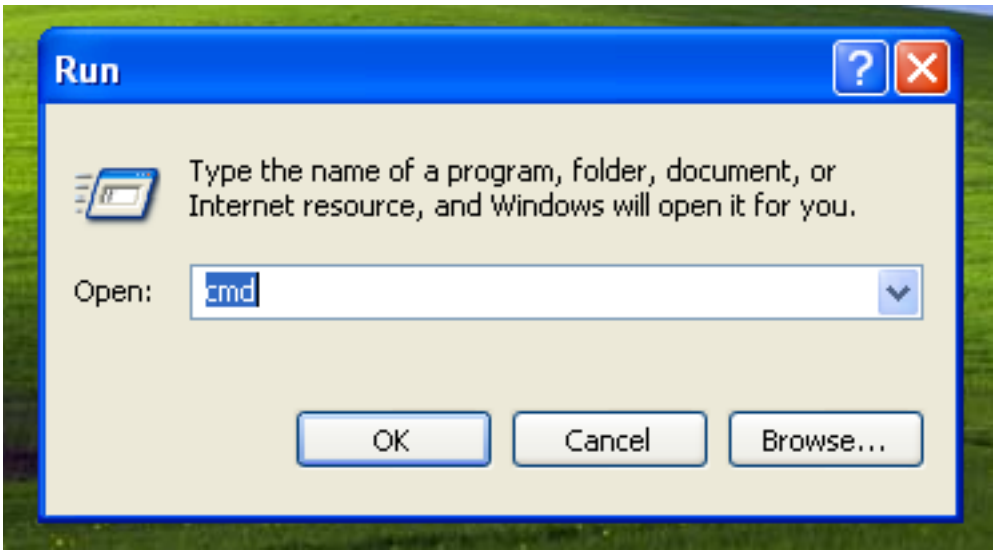

My Music
My Computer
Control Par
Set Prograr
Defaults
Printers and
Help and Su
Search
Run...
Open
Explore
Search...
Manage
Map Network Drive...
Disconnect Network Drive...
Show on Desktop
Rename
Properties

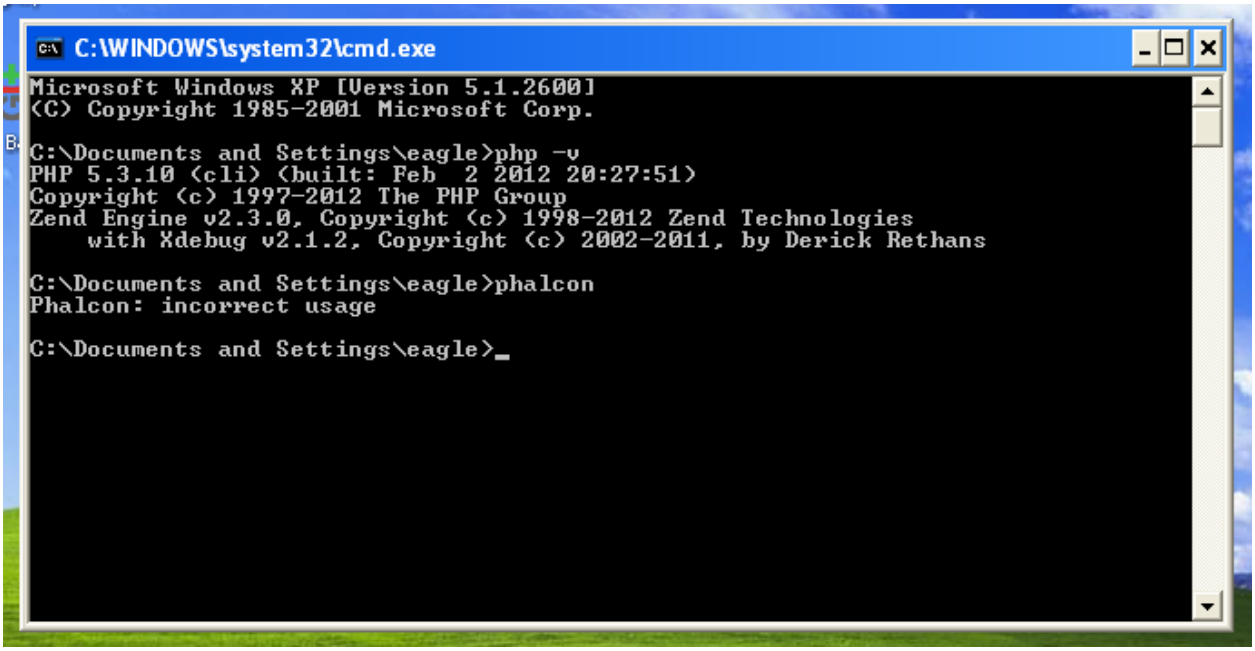

System Properties
System Restore
Automatic Updates
Remote
General
Computer Name
Hardware
Advanced
You must be logged on as an Administrator to make most of these changes.
Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings
User Profiles
Desktop settings related to your logon
Settings
Startup and Recovery
System startup, system failure, and debugging information
Settings
Environment Variables
Error Reporting
OK
Cancel
Apply


Environment Variables
User variables for eagle
Variable
Value
lib
C:\Program Files\Microsoft Visual Studio...
MSDevDir
C:\Program Files\Microsoft Visual Studio...
path
C:\Program Files\Microsoft Visual Studio...
TEMP
C:\Documents and Settings\eagle\Local ...
TMP
C:\Documents and Settings\eagle\Local ...
New
Edit
Delete
System variables
Variable
Value
NUMBER_OF_P...
1
OS
Windows_NT
Path
C:\WINDOWS\system32;C:\WINDOWS;...
PATHEXT
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...
x86
New
Edit
Delete
OK
Cancel


Edit System Variable
Variable name:
Path
Variable value:
wamp\bin\php\php5.3.10;C:\phalcon-tools
OK
Cancel


Search
Run...
Log Off
Turn Off


Run
Type the name of a program, folder, document, or
Internet resource, and Windows will open it for you.
Open:
cmd
OK
Cancel
Browse...


C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\eagle>php -v
PHP 5.3.10 (cli) (built: Feb 2 2012 20:27:51)
Copyright (c) 1997-2012 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2012 Zend Technologies
with Xdebug v2.1.2, Copyright (c) 2002-2011, by Derick Rethans
C:\Documents and Settings\eagle>phalcon
Phalcon: incorrect usage
C:\Documents and Settings\eagle>

Related Guides

- *Using Developer Tools*
- *Installation on OS X*
- *Installation on Linux*

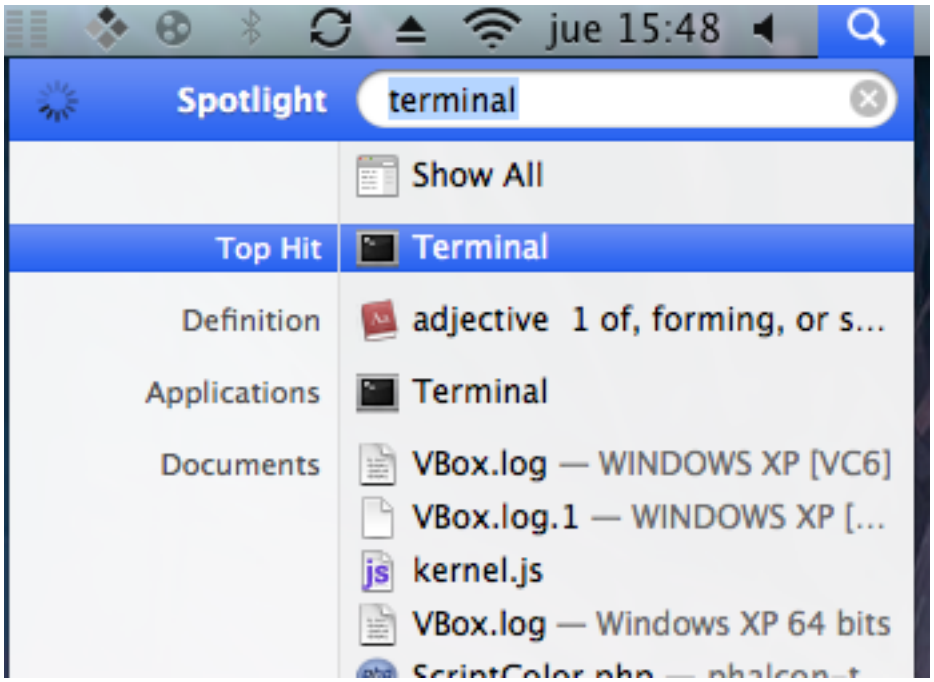
Phalcon Developer Tools on Mac OS X

These steps will guide you through the process of installing Phalcon Developer Tools for OS/X.

Prerequisites The Phalcon PHP extension is required to run Phalcon Tools. If you haven't installed it yet, please see the *Installation* section for instructions.

Download You can download a cross platform package containing the developer tools from the Download section. You can also clone it from Github.

Open the terminal application:



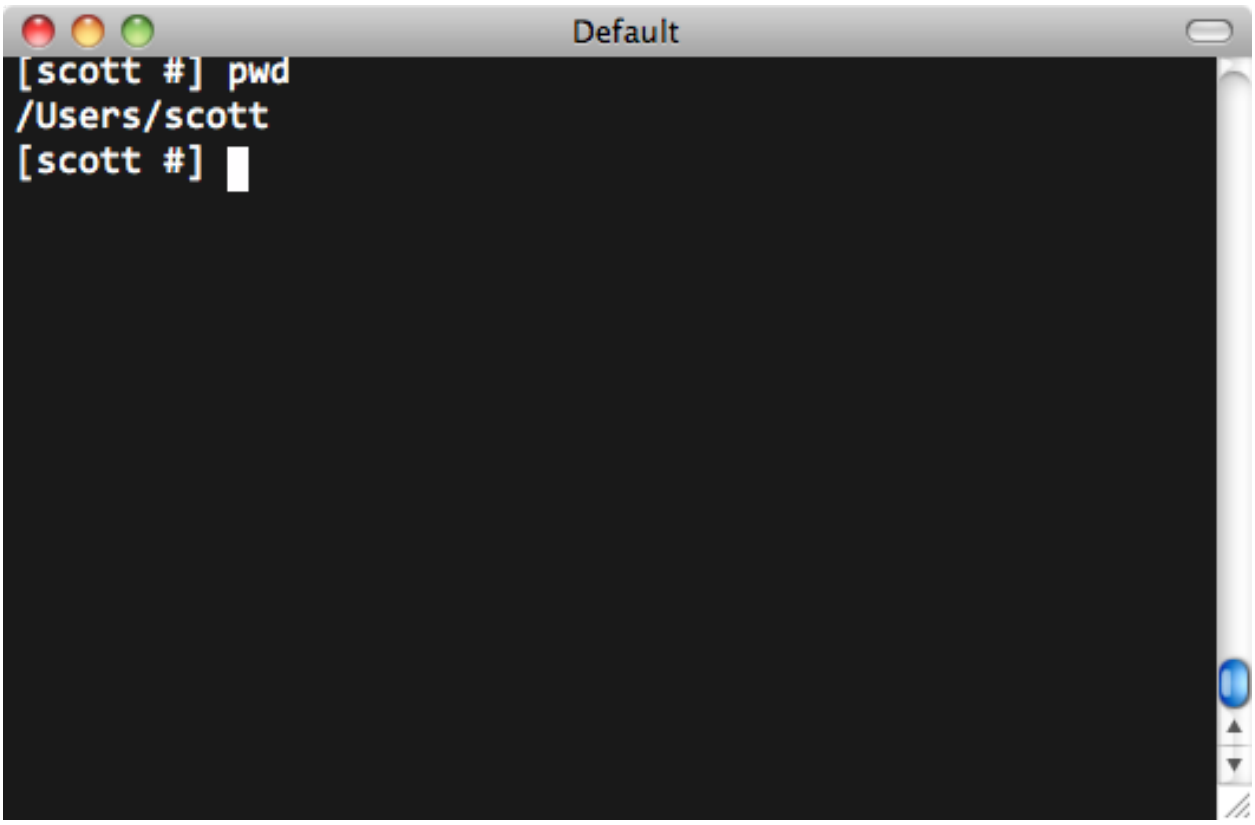
Copy & Paste the commands below in your terminal:

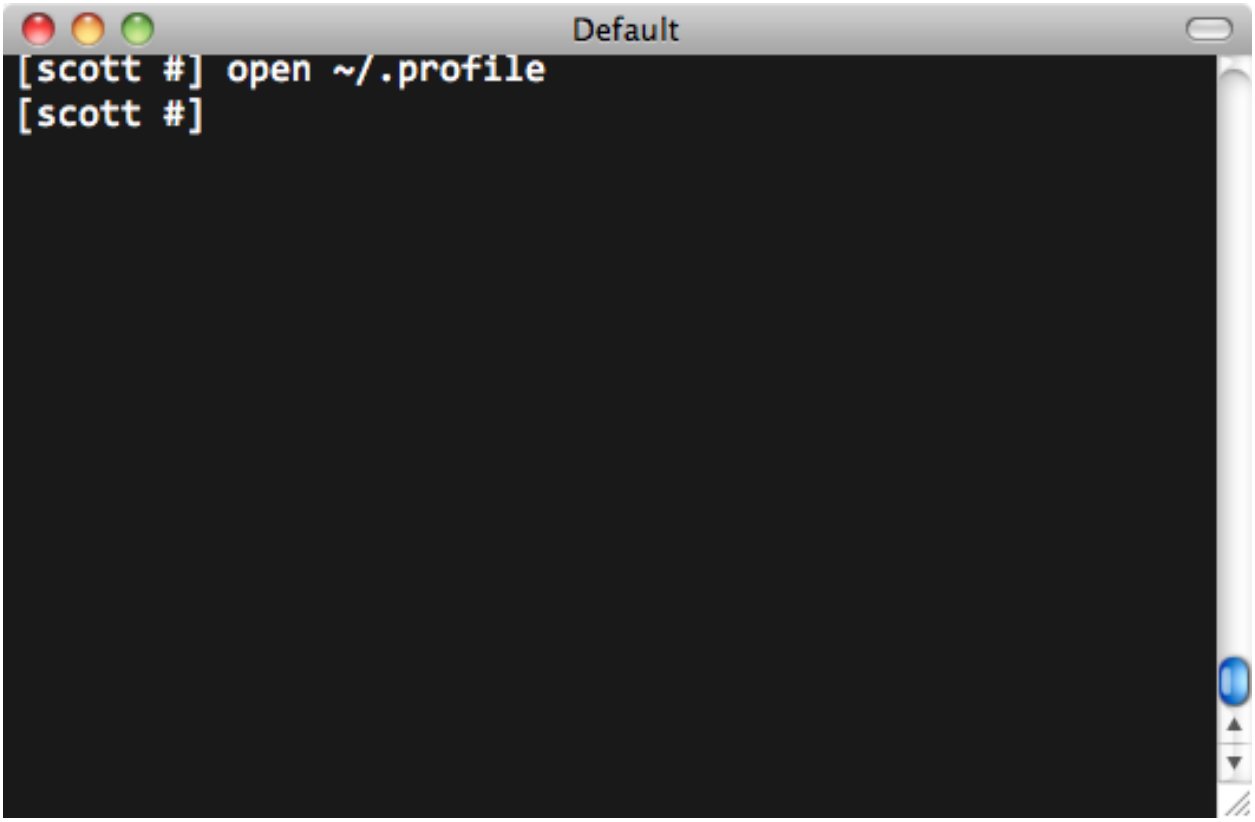
```
wget -q --no-check-certificate -O phalcon-tools.zip http://github.com/phalcon/phalcon-devtools/zipbal
unzip -q phalcon-tools.zip
mv phalcon-phalcon-devtools-* phalcon-tools
```

Check where the phalcon-tools directory was installed using a *pwd* command in your terminal:

On the Mac platform, you need to configure your user PATH to include Phalcon tools. Edit your .profile and append the Phalcon tools path to the environment variable PATH:

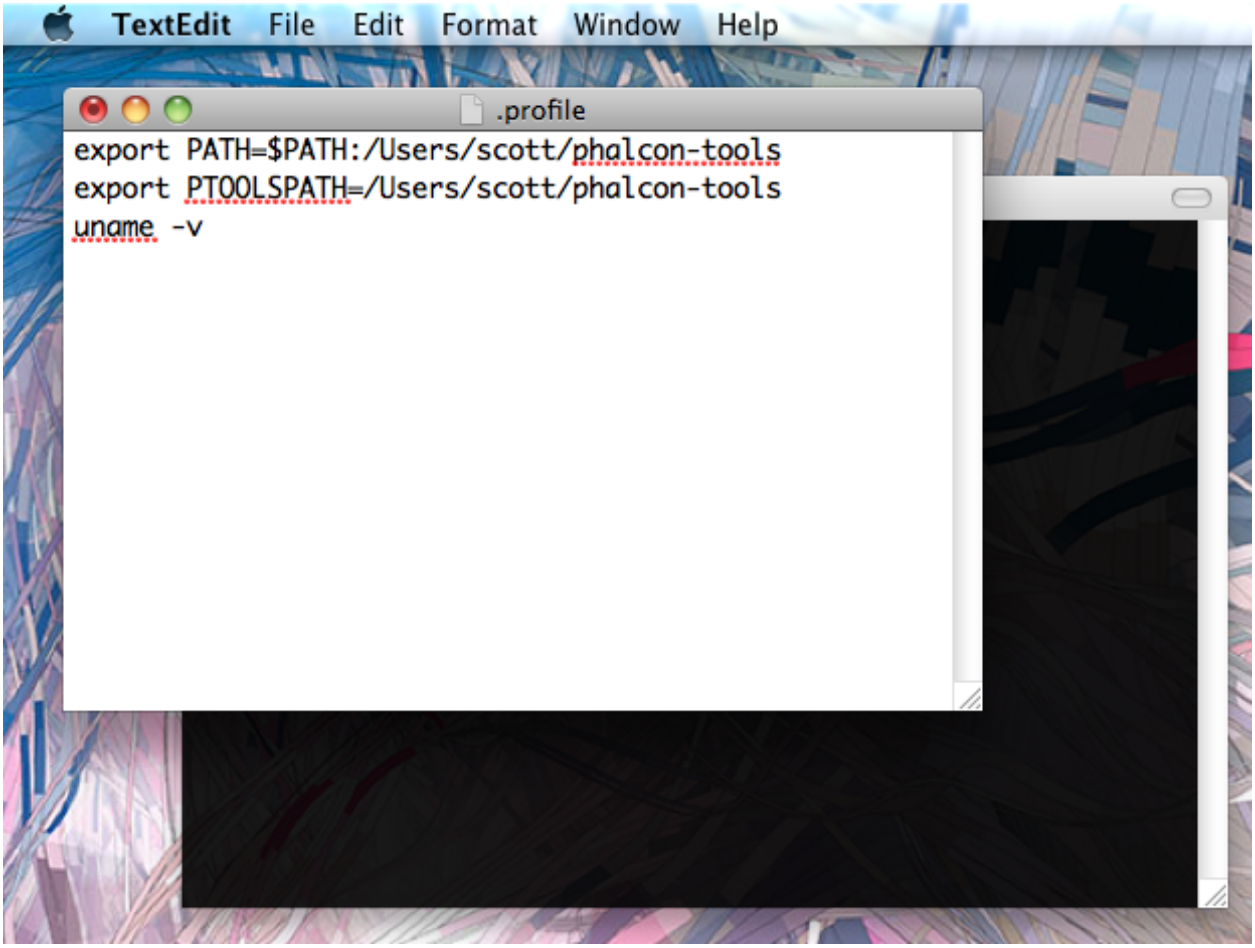
Insert these two lines at the end of the file:


Default
[scott #] pwd
/Users/scott
[scott #]


Default
[scott #] open ~/.profile
[scott #]

```
export PATH=$PATH:/Users/scott/phalcon-tools
export PTOOLSPATH=/Users/scott/phalcon-tools
```

The .profile should look like this:



Save your changes and close the editor. In the terminal window, type the following commands to create a symbolic link to the phalcon.sh script:

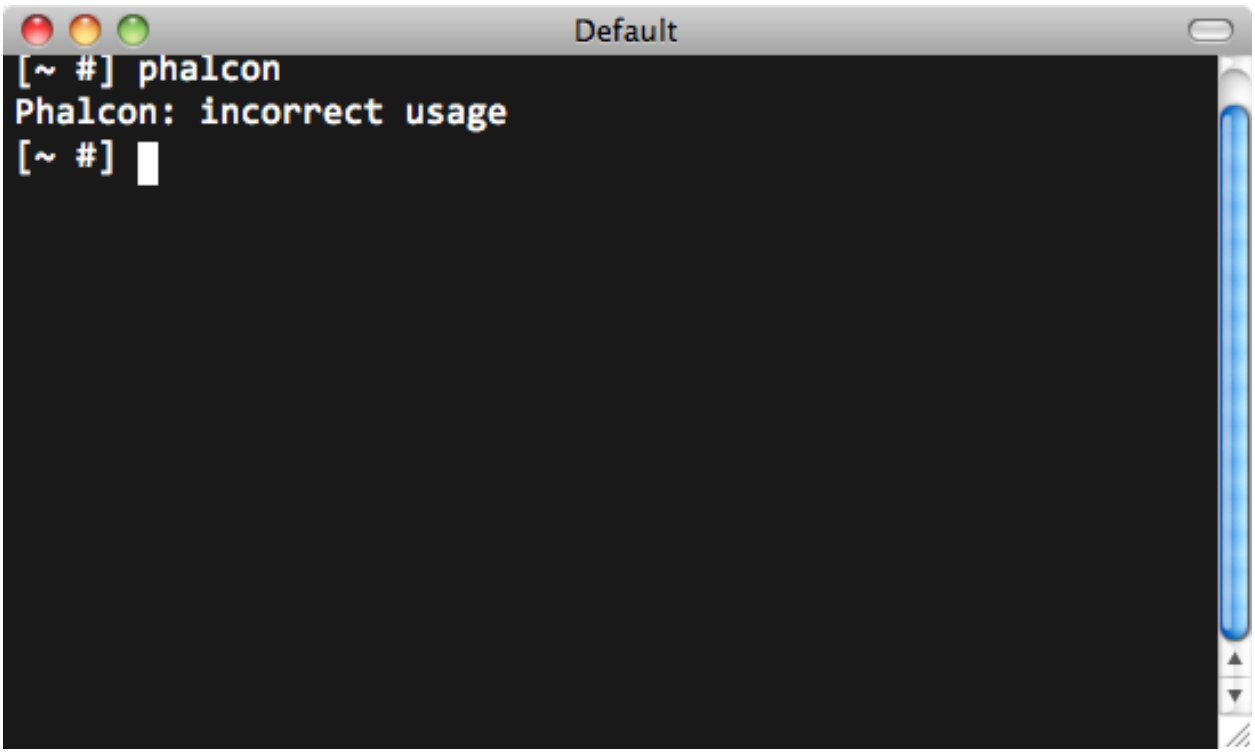
```
ln -s ~/phalcon-tools/phalcon.sh ~/phalcon-tools/phalcon
chmod +x ~/phalcon-tools/phalcon
```

Type the command "phalcon" and you will see something like this:

Congratulations you now have Phalcon tools installed!

Related Guides

- *Using Developer Tools*
- *Installation on Windows*
- *Installation on Linux*



Phalcon Developer Tools on Linux

These steps will guide you through the process of installing Phalcon Developer Tools for Linux.

Prerequisites The Phalcon PHP extension is required to run Phalcon Tools. If you haven't installed it yet, please see the *Installation* section for instructions.

Download You can download a cross platform package containing the developer tools from the Download section. Also you can clone it from Github.

Open a terminal and type the commands below:

Then enter the folder where the tools were cloned and execute ". ./phalcon.sh", (don't forget the dot at beginning of the command):

Congratulations you now have Phalcon tools installed!

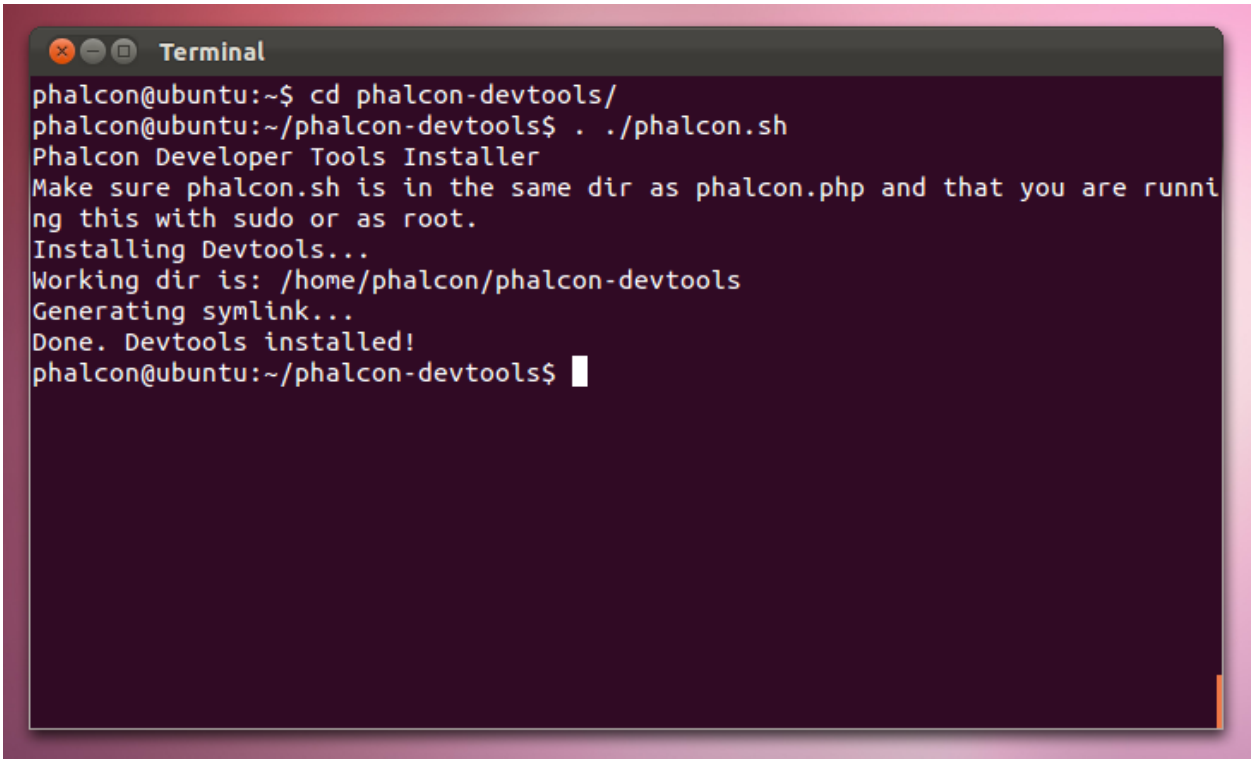
Related Guides

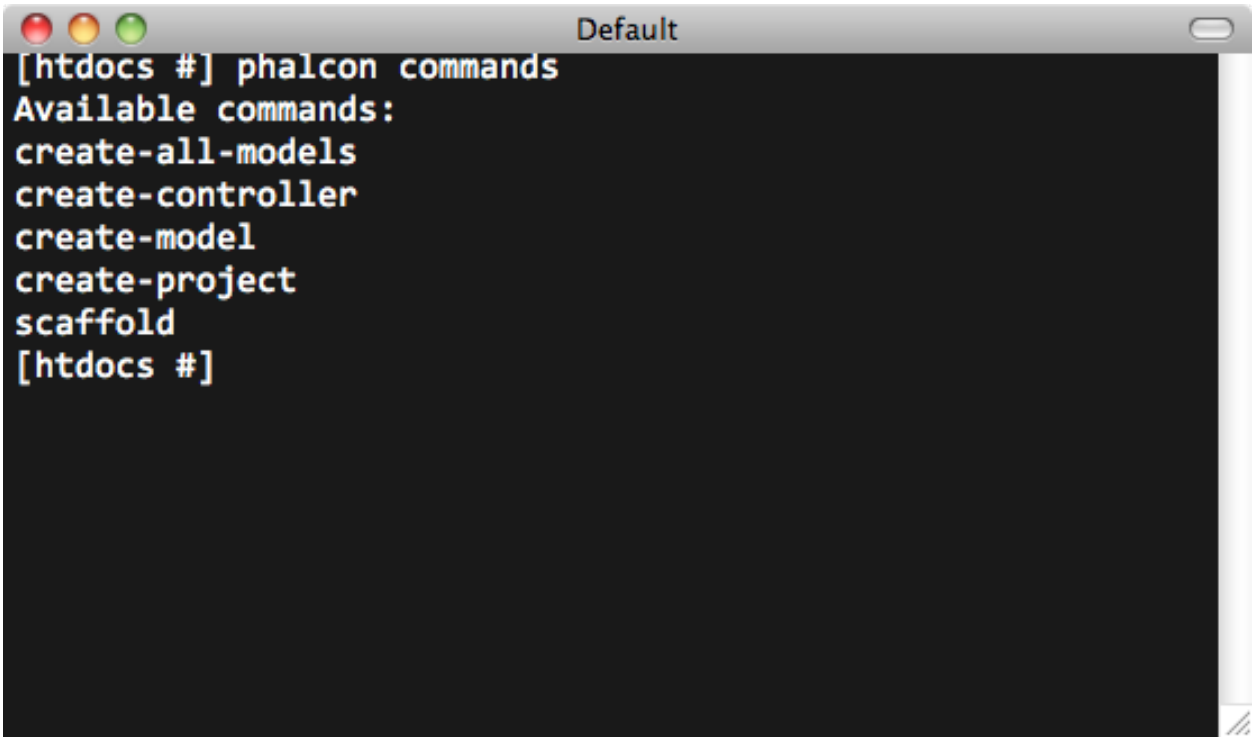
- *Using Developer Tools*
- *Installation on Windows*
- *Installation on Mac*

2.46.2 Getting Available Commands

You can get a list of available commands in Phalcon tools by typing: phalcon commands


Terminal
phalcon@ubuntu:~$ git clone git://github.com/phalcon/phalcon-devtools.git
Cloning into phalcon-devtools...
remote: Counting objects: 788, done.
remote: Compressing objects: 100% (597/597), done.
Receiving objects: 100% (788/788), 421.64 KiB | 230 KiB/s, done.
remote: Total 788 (delta 339), reused 590 (delta 141)
Resolving deltas: 100% (339/339), done.
phalcon@ubuntu:~$


Terminal
phalcon@ubuntu:~$ cd phalcon-devtools/
phalcon@ubuntu:~/phalcon-devtools$ . ./phalcon.sh
Phalcon Developer Tools Installer
Make sure phalcon.sh is in the same dir as phalcon.php and that you are runni
ng this with sudo or as root.
Installing Devtools...
Working dir is: /home/phalcon/phalcon-devtools
Generating symlink...
Done. Devtools installed!
phalcon@ubuntu:~/phalcon-devtools$



2.46.3 Generating a Project Skeleton

You can use Phalcon tools to generate pre-defined project skeletons for your applications with Phalcon framework. By default the project skeleton generator will use mod_rewrite for Apache. Type the following command on your web server document root:

The above recommended project structure was generated:

You could add the parameter *–help* to get help on the usage of a certain script:

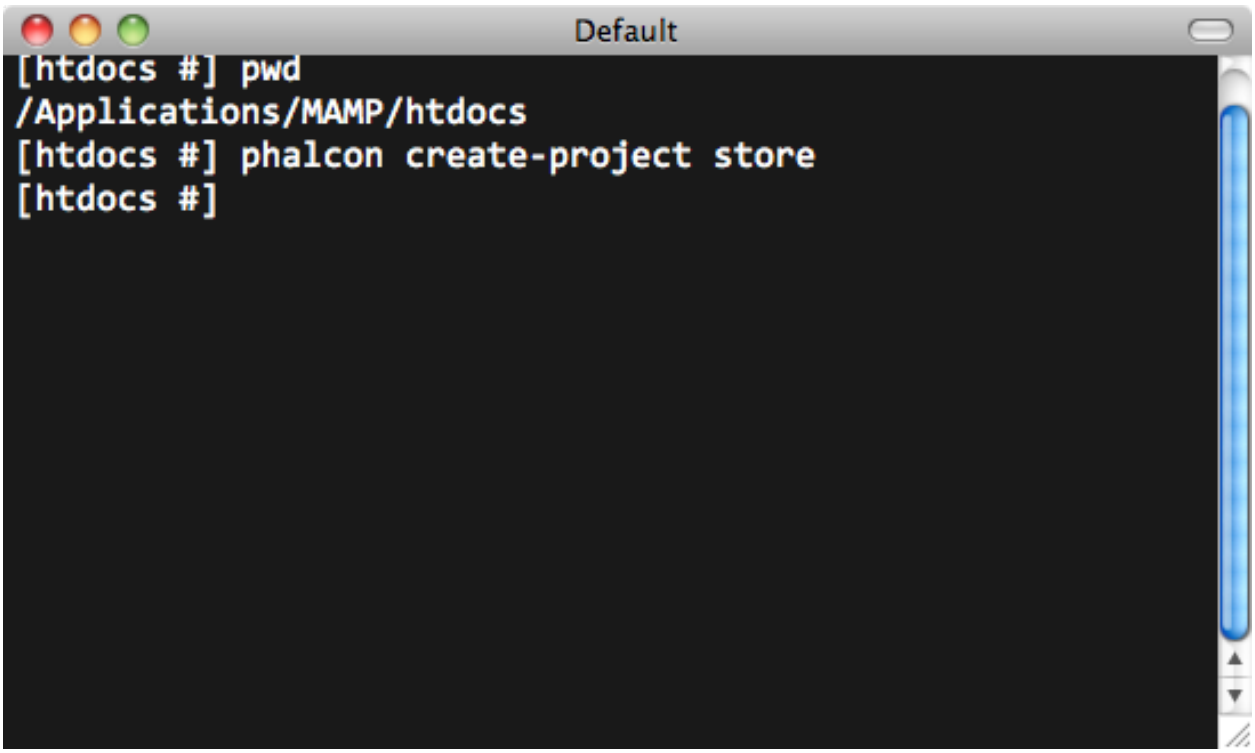
Accessing the project from the web server will show you:

2.46.4 Generating Controllers

The command "create-controller" generates controller skeleton structures. It's important to invoke this command inside a directory that already has a Phalcon project.

The following code is generated by the script:

```
<?php

class TestController extends Phalcon\Mvc\Controller
{

    public function indexAction()
    {

    }

}
```



2.46.5 Preparing Database Settings

When a project is generated using developer tools. A configuration file can be found in *app/config/config.ini* To generate models or scaffold, you will need to change the settings used to connect to your database.

Change the database section in your config.ini file:

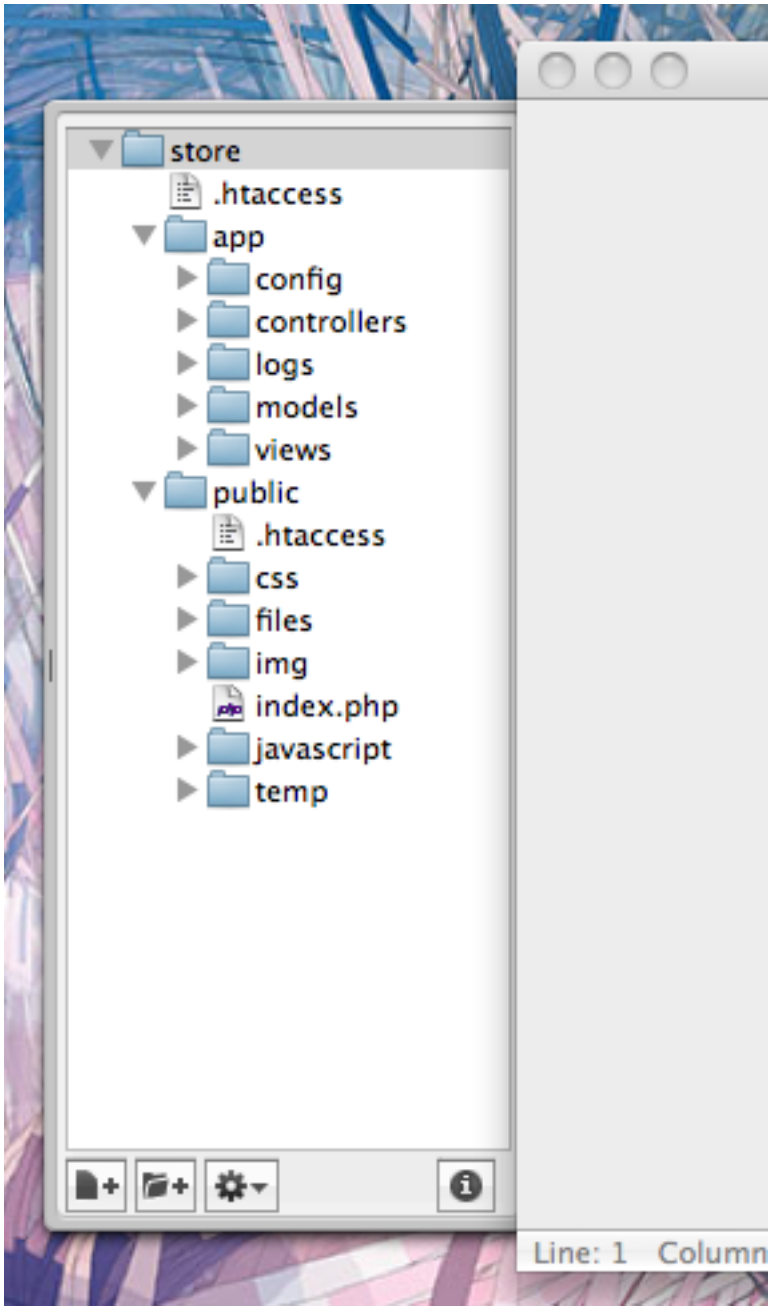
```
[database]
adapter  = Mysql
host     = "127.0.0.1"
username = "root"
password = "secret"
name     = "store_db"

[phalcon]
controllersDir = "../app/controllers/"
modelsDir      = "../app/models/"
viewsDir       = "../app/views/"
baseUri        = "/store/"
```

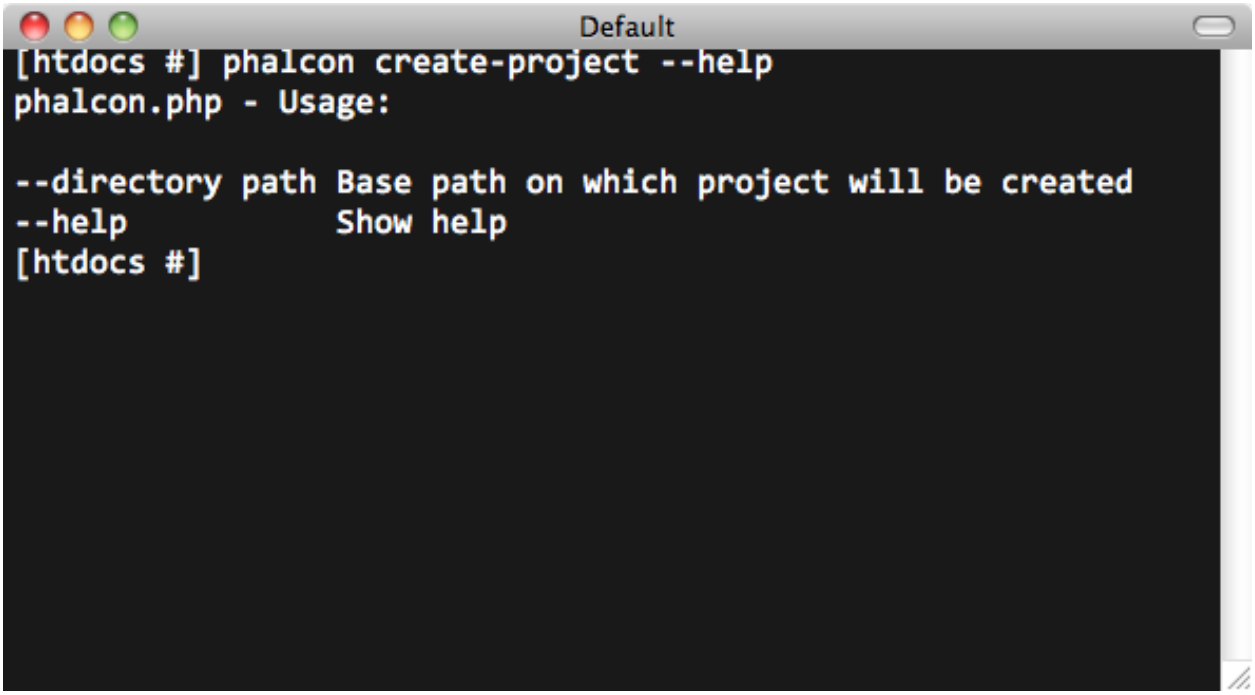
2.46.6 Generating Models

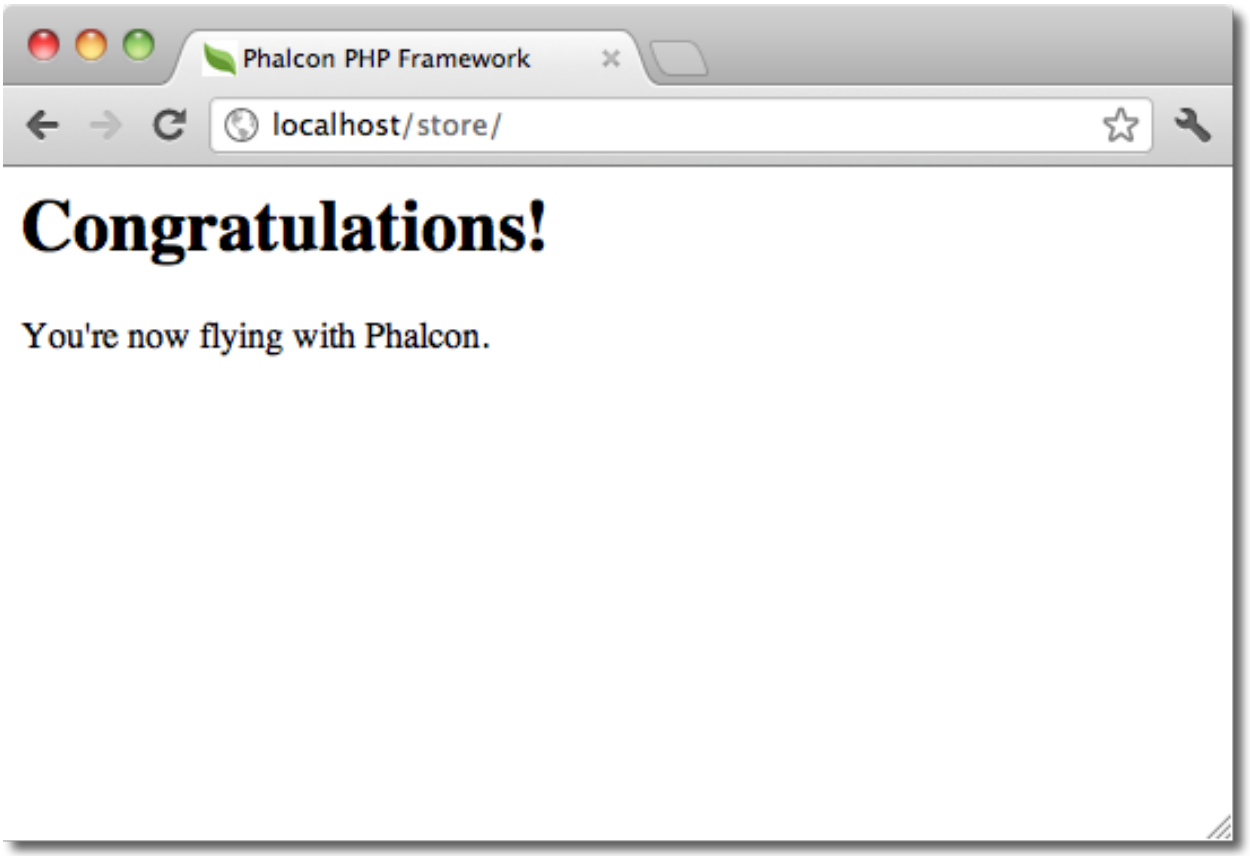
There are several ways to create models. You can create all models from the default database connection or some selectively. Models can have public attributes for the field representations or setters/getters can be used. The simplest way to generate a model is:

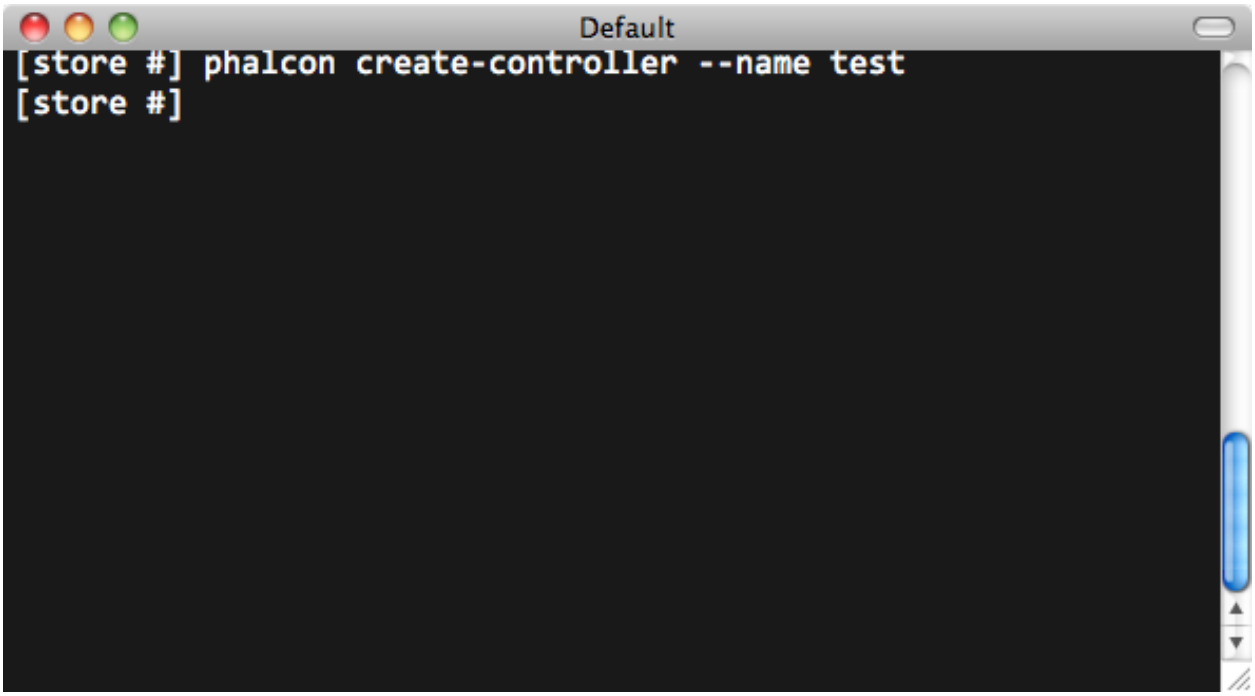
All table fields are declared public for direct access.

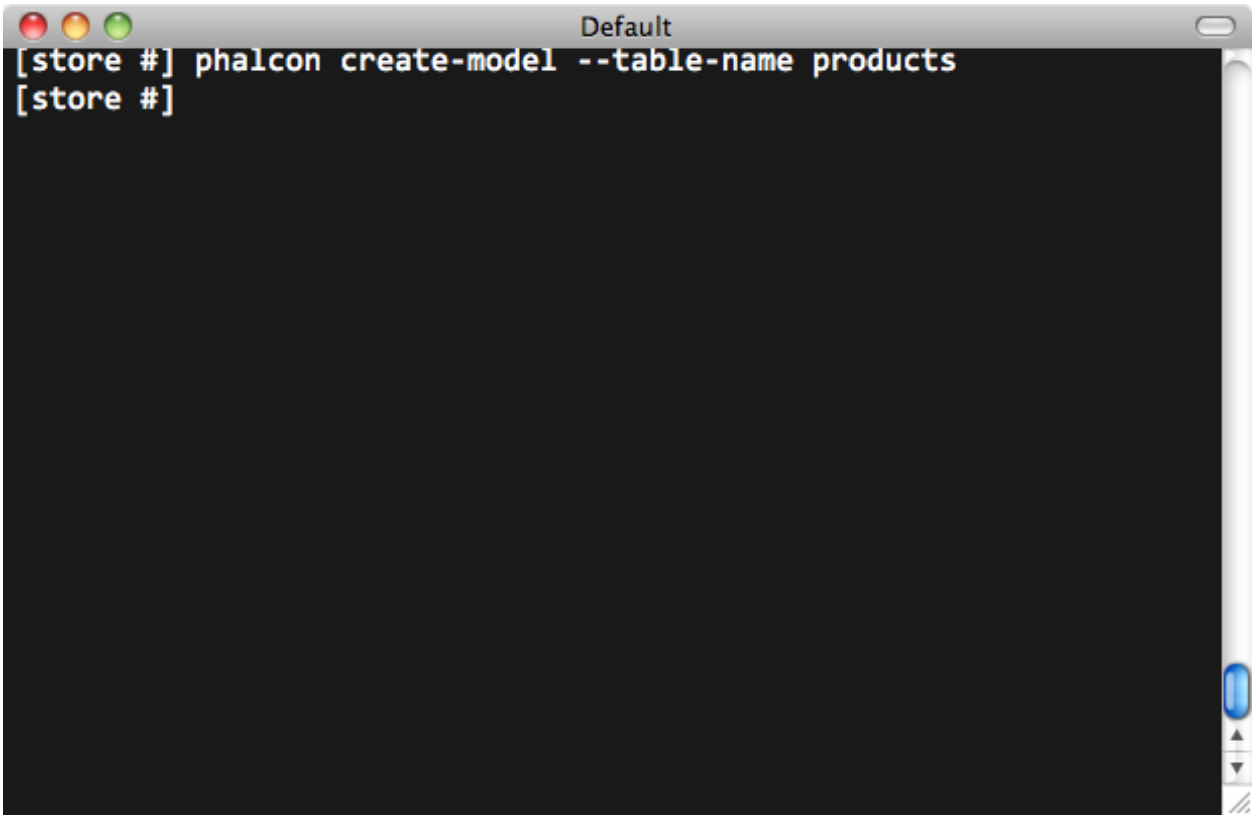
```
<?php

class Products extends \Phalcon\Mvc\Model
```


store
.htaccess
app
config
controllers
logs
models
views
public
.htaccess
css
files
img
index.php
javascript
temp
Line: 1 Column


Default
[htdocs #] phalcon create-project --help
phalcon.php - Usage:
--directory path Base path on which project will be created
--help Show help
[htdocs #]


Phalcon PHP Framework
localhost/store/
Congratulations!
You're now flying with Phalcon.


Default
[store #] phalcon create-controller --name test
[store #]


Default
[store #] phalcon create-model --table-name products
[store #]

```
{

    /**
     * @var integer
     */
    public $id;

    /**
     * @var integer
     */
    public $types_id;

    /**
     * @var string
     */
    public $name;

    /**
     * @var string
     */
    public $price;

    /**
     * @var integer
     */
    public $quantity;

    /**
     * @var string
     */
    public $status;

}
```

By adding the *–get-set* you can generate the fields with protected variables and public setter/getter methods. Those methods can help in business logic implementation within the setter/getter methods.

```
<?php

class Products extends \Phalcon\Mvc\Model
{

    /**
     * @var integer
     */
    protected $id;

    /**
     * @var integer
     */
    protected $types_id;

    /**
     * @var string
     */
    protected $name;

    /**
```

```
     * @var string
     */
    protected $price;

    /**
     * @var integer
     */
    protected $quantity;

    /**
     * @var string
     */
    protected $status;


    /**
     * Method to set the value of field id
     * @param integer $id
     */
    public function setId($id)
    {
        $this->id = $id;
    }

    /**
     * Method to set the value of field types_id
     * @param integer $types_id
     */
    public function setTypesId($types_id)
    {
        $this->types_id = $types_id;
    }

    ...

    /**
     * Returns the value of field status
     * @return string
     */
    public function getStatus()
    {
        return $this->status;
    }

}
```

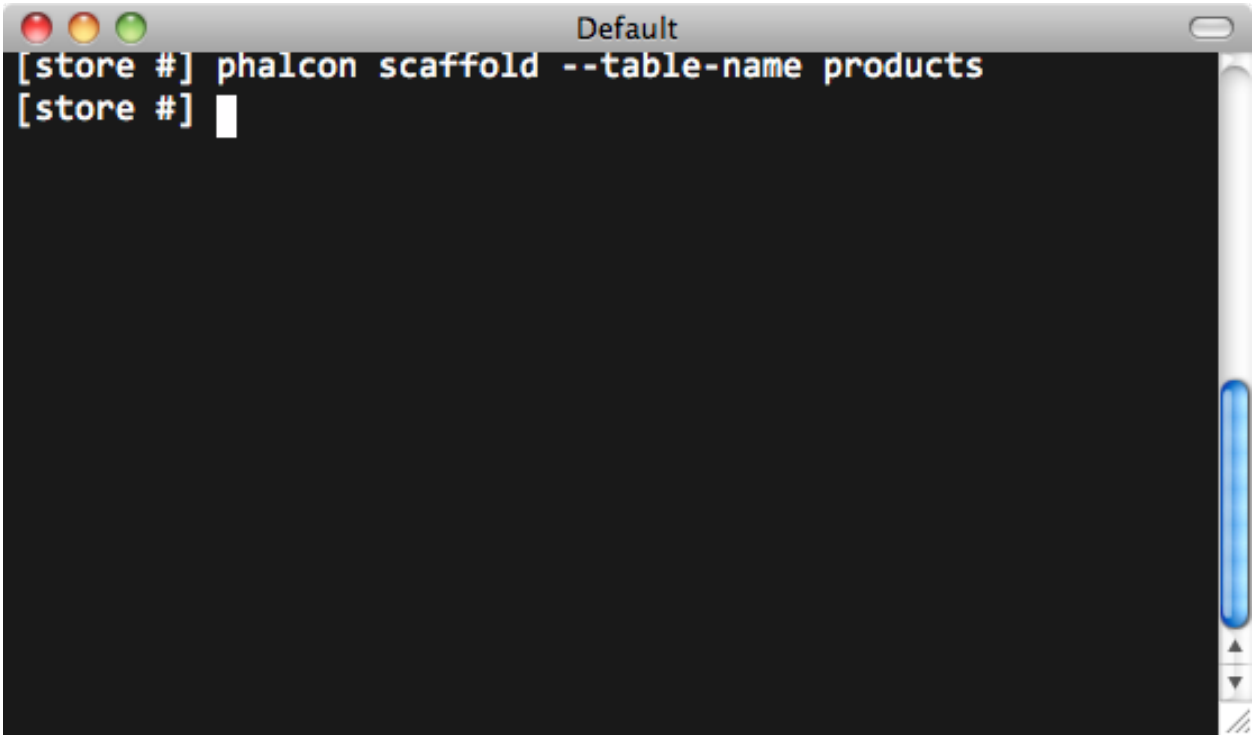
A nice feature of the model generator is that it keeps changes made by the developer between code generations. This allows the addition or removal of fields and properties, without worrying about losing changes made to the model itself. The following screencast shows you how it works:

2.46.7 Scaffold a CRUD

Scaffolding is a quick way to generate some of the major pieces of an application. If you want to create the models, views, and controllers for a new resource in a single operation, scaffolding is the tool for the job.

Once the code is generated, it will have to be customized to meet your needs. Many developers avoid scaffolding entirely, opting to write all or most of their source code from scratch. The generated code can serve as a guide to better

understand of how the framework works or develop prototypes. The screenshot below shows a scaffold based on the table "products":



The scaffold generator will build several files in your application, along with some folders. Here's a quick overview of what will be generated:

File	Purpose
app/controllers/ProductsController.php	The Products controller
app/models/Products.php	The Products model
app/views/layout/products.phtml	Controller layout for Products
app/views/products/new.phtml	View for the action "new"
app/views/products/edit.phtml	View for the action "edit"
app/views/products/search.phtml	View for the action "search"
app/views/products/edit.phtml	View for the action "edit"

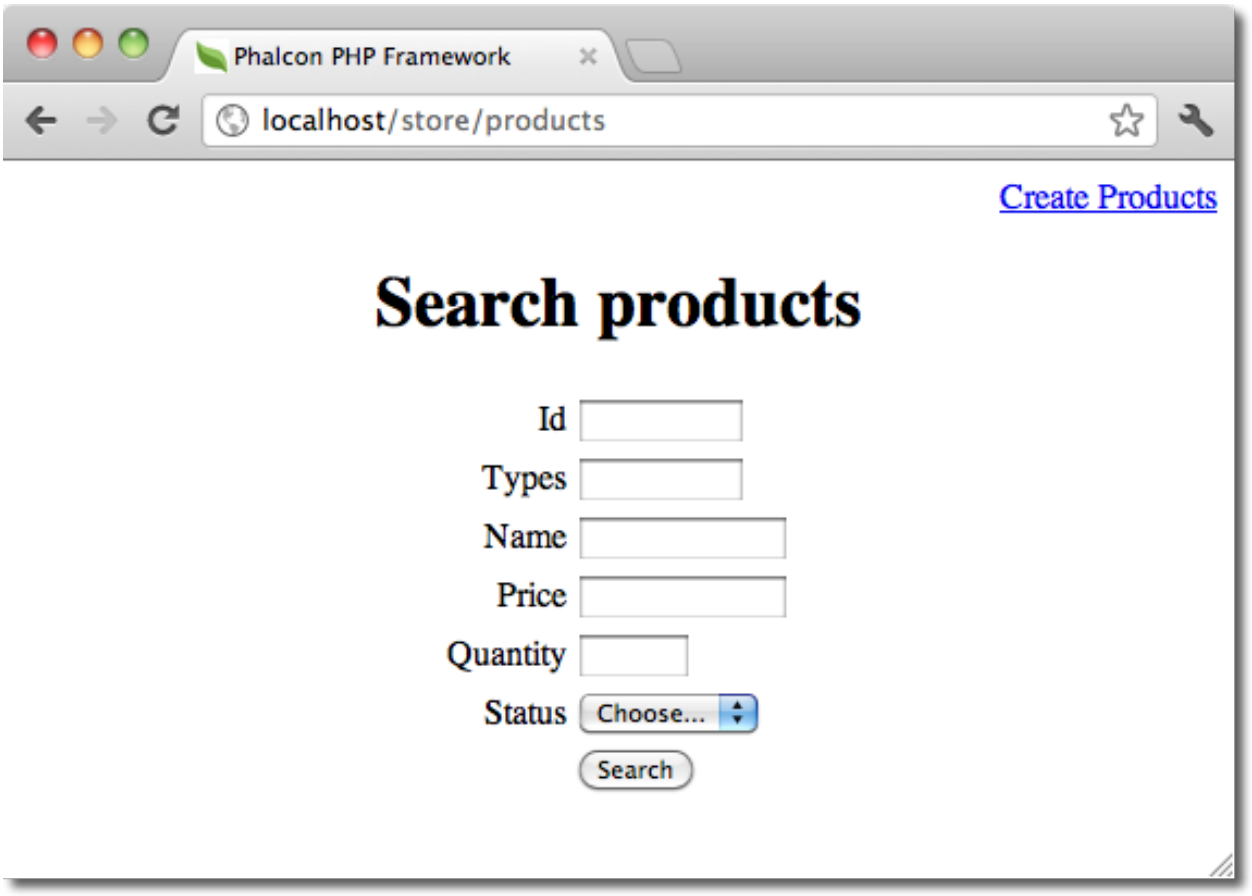
When browsing the recently generated controller, you will see a search form and a link to create a new Product:

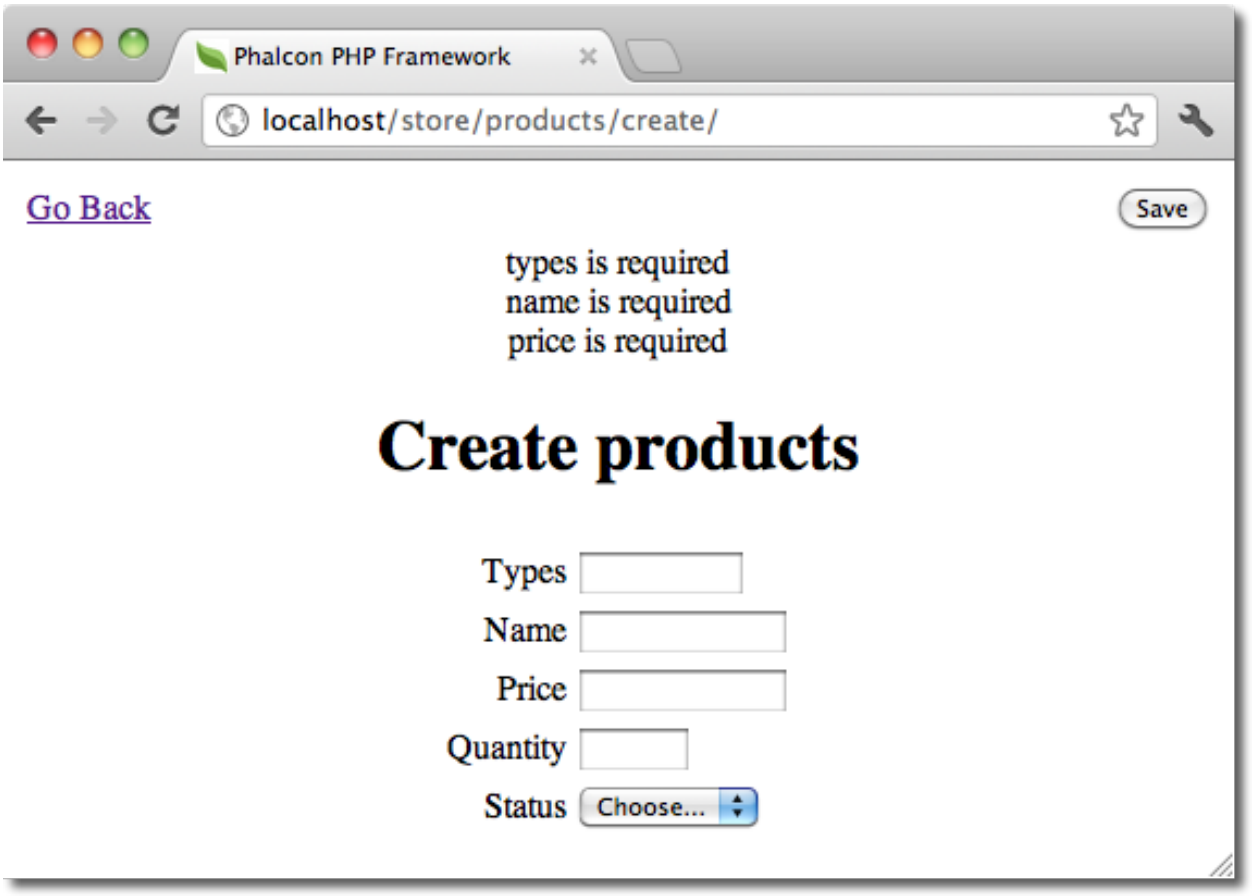
The "create page" allows you to create products applying validations on the Products model. Phalcon will automatically validate not null fields producing warnings if any of them is required.

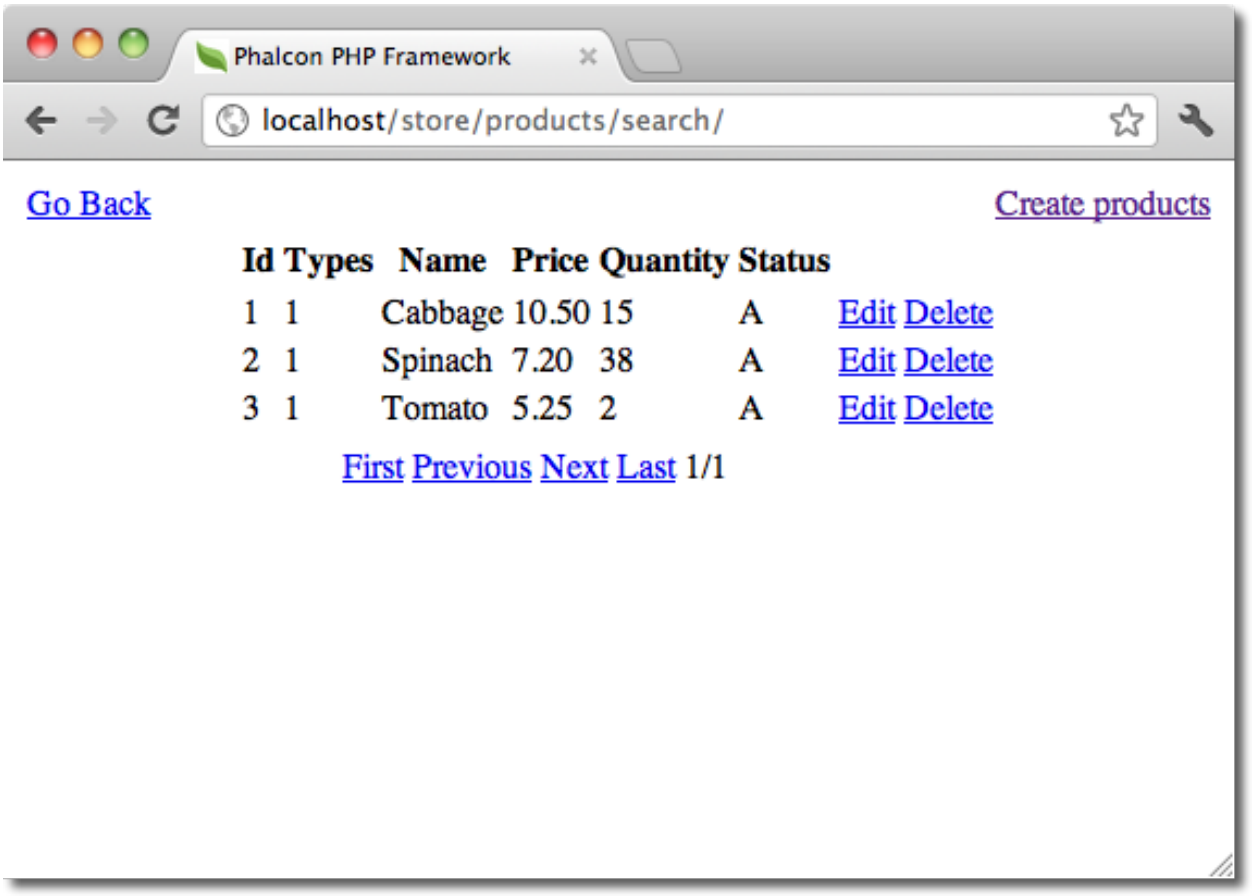
After performing a search, a pager component is available to show paged results. Use the "Edit" or "Delete" links in front of each result to perform such actions.

2.46.8 Web Interface to Tools

Also, if you prefer, it's possible to use Phalcon Developer Tools from a web interface. Check out the following screencast to figure out how it works:


Phalcon PHP Framework
localhost/store/products
Create Products
Search products
Id
Types
Name
Price
Quantity
Status
Choose...
Search


Phalcon PHP Framework
localhost/store/products/create/
Go Back
Save
types is required
name is required
price is required
Create products
Types
Name
Price
Quantity
Status
Choose...


Phalcon PHP Framework
localhost/store/products/search/
Go Back
Create products
Id Types Name Price Quantity Status
1 1 Cabbage 10.50 15 A Edit Delete
2 1 Spinach 7.20 38 A Edit Delete
3 1 Tomato 5.25 2 A Edit Delete
First Previous Next Last 1/1

2.46.9 Integrating Tools with PhpStorm IDE

The screencast below shows how to integrate developer tools with the PhpStorm IDE. The configuration steps could be easily adapted to other IDEs for PHP.

2.46.10 Conclusion

Phalcon Developer Tools provides an easy way to generate code for your application, reducing development time and potential coding errors.

2.47 Increasing Performance: What's next?

Get faster applications requires refine many aspects: server, client, network, database, web server, static sources, etc. In this chapter we highlight scenarios where you can improve performance and how detect what is really slow in your application.

2.47.1 Profile on the Server

Each application is different, the permanent profiling is important to understand where performance can be increased. Profiling gives us a real picture on what is really slow and what do not. Profiles can vary between a request and another, so it is important to make enough measurements to make conclusions.

Profiling with XDebug

Xdebug provides an easier way to profile PHP applications, just install the extension and enable profiling in the php.ini:

```
xdebug.profiler_enable = On
```

Using a tool like Webgrind you can see which functions/methods are slower than others:

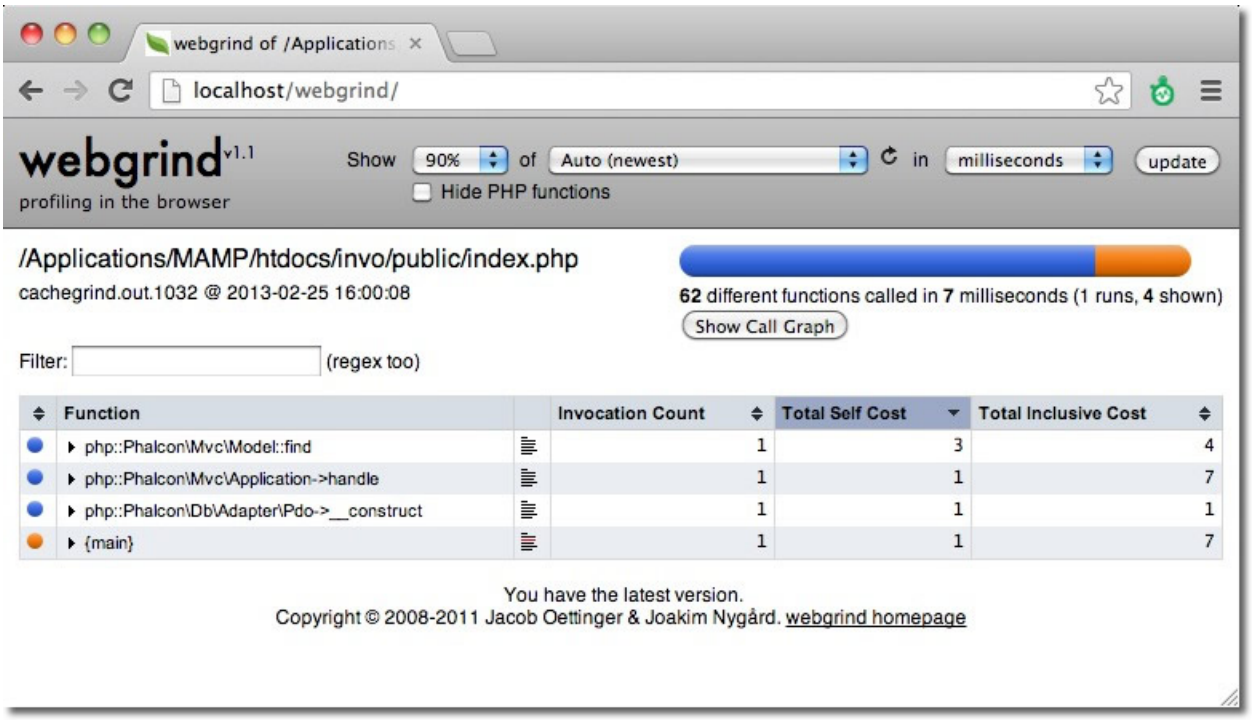
Profiling with Xhprof

Xhprof is another interesting extension to profile PHP applications. Add the following line to the start of the bootstrap file:

```
<?php

xhprof_enable(XHPROF_FLAGS_CPU + XHPROF_FLAGS_MEMORY);
```

Then at the end of the file save the profiling data:

```
<?php

$xhprof_data = xhprof_disable('/tmp');

$XHPROF_ROOT = "/var/www/xhprof/";
include_once $XHPROF_ROOT . "/xhprof_lib/utils/xhprof_lib.php";
include_once $XHPROF_ROOT . "/xhprof_lib/utils/xhprof_runs.php";

$xhprof_runs = new XHProfRuns_Default();
$run_id = $xhprof_runs->save_run($xhprof_data, "xhprof_testing");
```




```
echo "http://localhost/xhprof/xhprof_html/index.php?run={$run_id}&source=xhprof_testing\n";
```

Xhprof provides a built-in html viewer to analize the profile data:

Profiling SQL Statements

Most database systems provide tools to identify slow SQL statements. Detecting and fixing slow queries is very important to increase the performance in the server side. In the Mysql case, you can use the slow query log to know what SQL queries are taking more time than expected:

```
log-slow-queries = /var/log/slow-queries.log
long_query_time = 1.5
```

2.47.2 Profile on the Client

Sometimes we may need to improve the loading of static elements such as images, javascript and css to improve performance. The following tools are useful to detect common bottlenecks in the client side:

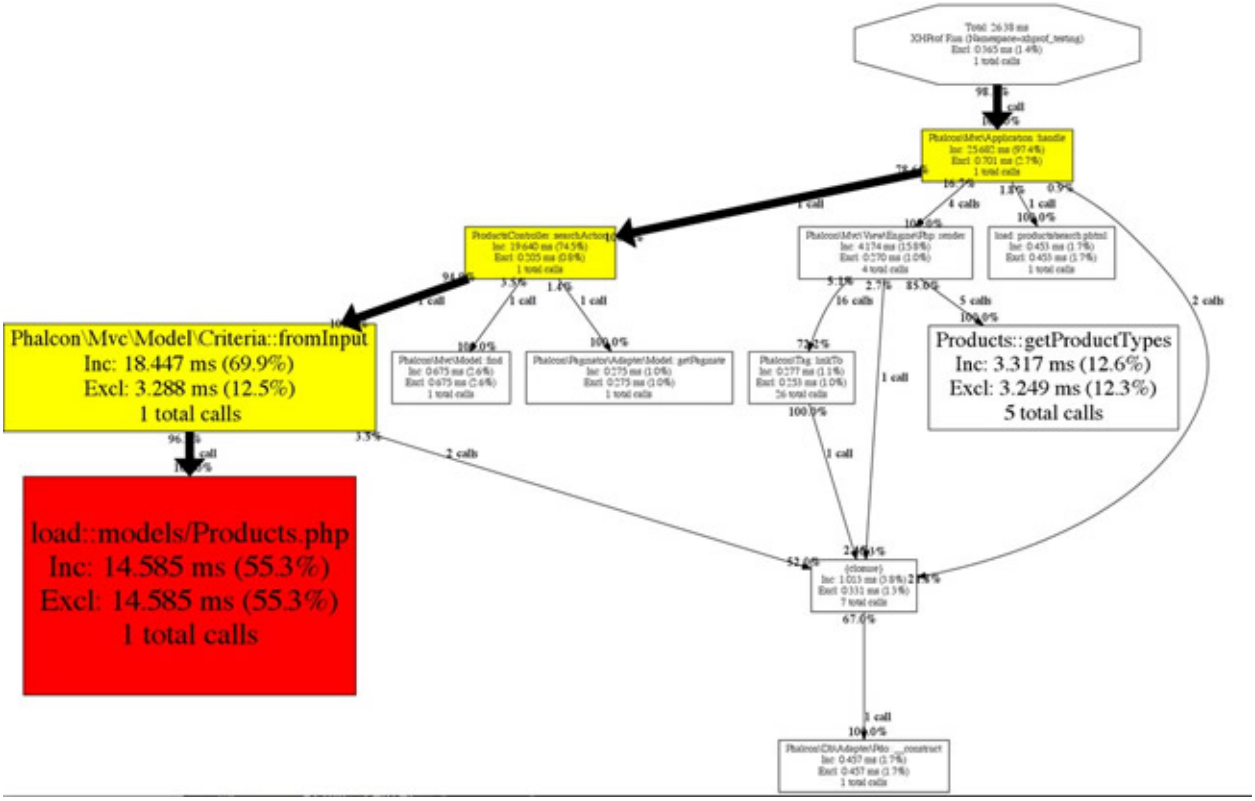
Profile with Chrome/Firefox

Most modern browsers have tools to profile the page loading time. In Chrome you can use the web inspector to know how much time is taking the loading of the different resources required by a single page:

Firebug provides a similar functionality:

Displaying top 100 functions: Sorted by Excl. Wall Time (microsec) [display all]

Function Name	Calls	Calls%	Incl. Wall Time (microsec)	IWall%	Excl. Wall Time (microsec)	EWall%	Incl. CPU (microsecs)	ICpu%	Excl. CPU (microsec)	ECPU%	Incl. MemUse (bytes)
load::models/Products.php	1	0.8%	14,585	55.3%	14,585	55.3%	346	5.4%	346	5.4%	43,528
Phalcon\Mvc\Model\Criteria::fromInput	1	0.8%	18,447	69.9%	3,288	12.5%	1,515	23.7%	831	13.0%	87,544
Products::getProductTypes	5	3.8%	3,317	12.6%	3,249	12.3%	1,029	16.1%	958	15.0%	57,992
Phalcon\Mvc\Application::handle	1	0.8%	25,682	97.4%	701	2.7%	5,981	93.4%	683	10.7%	355,296
Phalcon\Mvc\Model::find	1	0.8%	675	2.6%	675	2.6%	457	7.1%	457	7.1%	16,288
Phalcon\Db\Adapter\Pdo::__construct	1	0.8%	457	1.7%	457	1.7%	217	3.4%	217	3.4%	8,664
load::products/search.phtml	1	0.8%	453	1.7%	453	1.7%	288	4.5%	288	4.5%	26,144
main()	1	0.8%	26,380	100.0%	365	1.4%	6,401	100.0%	128	2.0%	402,568
{closure}	7	5.4%	1,013	3.8%	331	1.3%	777	12.1%	308	4.8%	99,544
Phalcon\Paginator\Adapter\Model::getPaginate	1	0.8%	275	1.0%	275	1.0%	276	4.3%	276	4.3%	24,928
Phalcon\Mvc\View\Engine\Php::render	4	3.1%	4,174	15.8%	270	1.0%	1,864	29.1%	231	3.6%	99,664
Phalcon\Tag::linkTo	26	20.0%	277	1.1%	253	1.0%	292	4.6%	268	4.2%	9,024
ProductsController::searchAction	1	0.8%	19,640	74.5%	205	0.8%	2,479	38.7%	184	2.9%	142,392
Phalcon\Loader::autoLoad	5	3.8%	174	0.7%	128	0.5%	122	1.9%	89	1.4%	14,032
Phalcon\Config\Adapter\Ini::__construct	1	0.8%	122	0.5%	122	0.5%	123	1.9%	123	1.9%	4,480
Phalcon\Session\Adapter::start	1	0.8%	115	0.4%	115	0.4%	117	1.8%	117	1.8%	34,328
Phalcon\DI::set	8	6.2%	80	0.3%	80	0.3%	65	1.0%	65	1.0%	2,592
Elements::getMenu	1	0.8%	107	0.4%	59	0.2%	108	1.7%	52	0.8%	5,632
Security::beforeDispatch	1	0.8%	243	0.9%	52	0.2%	244	3.8%	46	0.7%	47,208
Phalcon\Loader::__construct	1	0.8%	51	0.2%	51	0.2%	15	0.2%	15	0.2%	1,160
Elements::getTabs	1	0.8%	96	0.4%	51	0.2%	97	1.5%	45	0.7%	4,504
Phalcon\DI\FactoryDefault::__construct	1	0.8%	47	0.2%	47	0.2%	49	0.8%	49	0.8%	17,736



2.47.3 Yahoo! YSlow

YSlow analyzes web pages and suggests ways to improve their performance based on a set of rules for high performance web pages

Profile with Speed Tracer

Speed Tracer is a tool to help you identify and fix performance problems in your web applications. It visualizes metrics that are taken from low level instrumentation points inside of the browser and analyzes them as your application runs. Speed Tracer is available as a Chrome extension and works on all platforms where extensions are currently supported (Windows and Linux).

This tool is very useful because it help you to get the real time used to render the whole page including HTML parsing, Javascript evaluation and CSS styling.

2.47.4 Use a PHP Bytecode Cache

APC as many other bytecode caches help an application to reduce the overhead of read, tokenize and parse PHP files in each request. Once the extension is installed use the following setting to enable APC:

```
apc.enabled = On
```

2.47.5 Google Page Speed

mod_pagespeed speeds up your site and reduces page load time. This open-source Apache HTTP server module automatically applies web performance best practices to pages, and associated assets (CSS, JavaScript, images) without requiring that you modify your existing content or workflow.

2.48 API Indice

2.48.1 Class Phalcon\Acl

Constants

integer **ALLOW**

integer **DENY**

2.48.2 Class Phalcon\Acl\Adapter

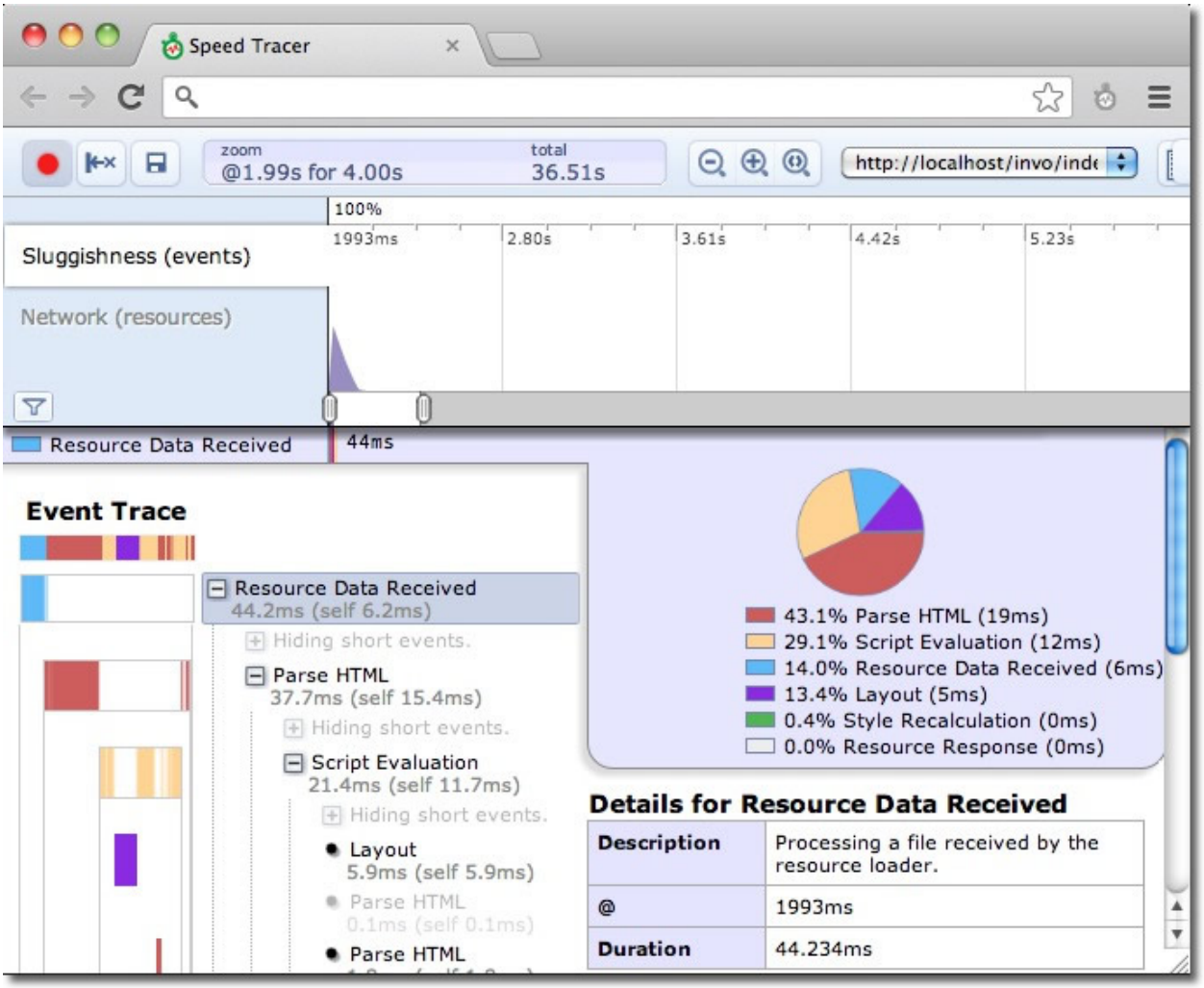
implements Phalcon\Events\EventsAwareInterface

Adapter for Phalcon\Acl adapters

Methods

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager)

Sets the events manager


Speed Tracer
zoom
@1.99s for 4.00s
total
36.51s
http://localhost/invo/inde
100%
1993ms
2.80s
3.61s
4.42s
5.23s
Sluggishness (events)
Network (resources)
Resource Data Received
44ms
Event Trace
Resource Data Received
44.2ms (self 6.2ms)
Hiding short events.
Parse HTML
37.7ms (self 15.4ms)
Hiding short events.
Script Evaluation
21.4ms (self 11.7ms)
Hiding short events.
Layout
5.9ms (self 5.9ms)
Parse HTML
0.1ms (self 0.1ms)
Parse HTML
43.1% Parse HTML (19ms)
29.1% Script Evaluation (12ms)
14.0% Resource Data Received (6ms)
13.4% Layout (5ms)
0.4% Style Recalculation (0ms)
0.0% Resource Response (0ms)
Details for Resource Data Received
Description
Processing a file received by the resource loader.
@
1993ms
Duration
44.234ms

public *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the internal event manager

public **setDefaultAction** (*int* $defaultAccess)

Sets the default access level (Phalcon\Acl::ALLOW or Phalcon\Acl::DENY)

public *int* **getDefaultAction** ()

Returns the default ACL access level

public *string* **getActiveRole** ()

Returns the role which the list is checking if it's allowed to certain resource/access

public *string* **getActiveResource** ()

Returns the resource which the list is checking if some role can access it

public *string* **getActiveAccess** ()

Returns the access which the list is checking if some role can access it

2.48.3 Class Phalcon\Acl\Adapter\Memory

extends Phalcon\Acl\Adapter

implements Phalcon\Events\EventsAwareInterface, *Phalcon\Acl\AdapterInterface*

Manages ACL lists in memory

```
<?php

$acl = new Phalcon\Acl\Adapter\Memory();

$acl->setDefaultAction(Phalcon\Acl::DENY);

//Register roles
$roles = array(
    'users' => new Phalcon\Acl\Role('Users'),
    'guests' => new Phalcon\Acl\Role('Guests')
);
foreach ($roles as $role) {
    $acl->addRole($role);
}

//Private area resources
  $privateResources = array(
    'companies' => array('index', 'search', 'new', 'edit', 'save', 'create', 'delete'),
    'products' => array('index', 'search', 'new', 'edit', 'save', 'create', 'delete'),
    'invoices' => array('index', 'profile')
);
foreach ($privateResources as $resource => $actions) {
    $acl->addResource(new Phalcon\Acl\Resource($resource), $actions);
}

//Public area resources
$publicResources = array(
    'index' => array('index'),
    'about' => array('index'),
    'session' => array('index', 'register', 'start', 'end'),
```

```
    'contact' => array('index', 'send')
);
  foreach ($publicResources as $resource => $actions) {
    $acl->addResource(new Phalcon\Acl\Resource($resource), $actions);
}

  //Grant access to public areas to both users and guests
foreach ($roles as $role){
    foreach ($publicResources as $resource => $actions) {
            $acl->allow($role->getName(), $resource, '*');
    }
}

//Grant acess to private area to role Users
  foreach ($privateResources as $resource => $actions) {
            foreach ($actions as $action) {
            $acl->allow('Users', $resource, $action);
    }
}
```

Methods

public **__construct** ()

Phalcon\Acl\Adapter\Memory constructor

public *boolean* **addRole** (*Phalcon\Acl\RoleInterface* $role, [*array|string* $accessInherits])

Adds a role to the ACL list. Second parameter allows inheriting access data from other existing role Example:

```
<?php

    $acl->addRole(new Phalcon\Acl\Role('administrator'), 'consultant');
    $acl->addRole('administrator', 'consultant');
```

public **addInherit** (*string* $roleName, *string* $roleToInherit)

Do a role inherit from another existing role

public *boolean* **isRole** (*string* $roleName)

Check whether role exist in the roles list

public *boolean* **isResource** (*string* $resourceName)

Check whether resource exist in the resources list

public *boolean* **addResource** (*Phalcon\Acl\Resource* $resource, [*array* $accessList])

Adds a resource to the ACL list Access names can be a particular action, by example search, update, delete, etc or a list of them Example:

```
<?php

 //Add a resource to the the list allowing access to an action
 $acl->addResource(new Phalcon\Acl\Resource('customers'), 'search');
 $acl->addResource('customers', 'search');

 //Add a resource  with an access list
 $acl->addResource(new Phalcon\Acl\Resource('customers'), array('create', 'search'));
 $acl->addResource('customers', array('create', 'search'));
```

public **addResourceAccess** (*string* $resourceName, *mixed* $accessList)

Adds access to resources

public **dropResourceAccess** (*string* $resourceName, *mixed* $accessList)

Removes an access from a resource

protected **_allowOrDeny** ()

Checks if a role has access to a resource

public **allow** (*string* $roleName, *string* $resourceName, *mixed* $access)

Allow access to a role on a resource You can use '*' as wildcard Example:

```
<?php

 //Allow access to guests to search on customers
 $acl->allow('guests', 'customers', 'search');

 //Allow access to guests to search or create on customers
 $acl->allow('guests', 'customers', array('search', 'create'));

 //Allow access to any role to browse on products
 $acl->allow('*', 'products', 'browse');

 //Allow access to any role to browse on any resource
 $acl->allow('*', '*', 'browse');
```

public *boolean* **deny** (*string* $roleName, *string* $resourceName, *mixed* $access)

Deny access to a role on a resource You can use '*' as wildcard Example:

```
<?php

 //Deny access to guests to search on customers
 $acl->deny('guests', 'customers', 'search');

 //Deny access to guests to search or create on customers
 $acl->deny('guests', 'customers', array('search', 'create'));

 //Deny access to any role to browse on products
 $acl->deny('*', 'products', 'browse');

 //Deny access to any role to browse on any resource
 $acl->deny('*', '*', 'browse');
```

public *boolean* **isAllowed** (*string* $role, *string* $resource, *string* $access)

Check whether a role is allowed to access an action from a resource

```
<?php

 //Does andres have access to the customers resource to create?
 $acl->isAllowed('andres', 'Products', 'create');

 //Do guests have access to any resource to edit?
 $acl->isAllowed('guests', '*', 'edit');
```

public *Phalcon\Acl\Role* [] **getRoles** ()

Return an array with every role registered in the list

public *Phalcon\Acl\Resource* [] **getResources** ()

Return an array with every resource registered in the list

protected **_rebuildAccessList** ()

Rebuild the list of access from the inherit lists

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\Acl\Adapter

Sets the events manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\Acl\Adapter

Returns the internal event manager

public **setDefaultAction** (*int* $defaultAccess) inherited from Phalcon\Acl\Adapter

Sets the default access level (Phalcon\Acl::ALLOW or Phalcon\Acl::DENY)

public *int* **getDefaultAction** () inherited from Phalcon\Acl\Adapter

Returns the default ACL access level

public *string* **getActiveRole** () inherited from Phalcon\Acl\Adapter

Returns the role which the list is checking if it's allowed to certain resource/access

public *string* **getActiveResource** () inherited from Phalcon\Acl\Adapter

Returns the resource which the list is checking if some role can access it

public *string* **getActiveAccess** () inherited from Phalcon\Acl\Adapter

Returns the access which the list is checking if some role can access it

2.48.4 Class Phalcon\Acl\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.5 Class Phalcon\Acl\Resource

implements Phalcon\Acl\ResourceInterface

This class defines resource entity and its description

Methods

public **__construct** (*string* $name, [*string* $description])

Phalcon\Acl\Resource constructor

public *string* **getName** ()

Returns the resource name

public *string* **getDescription** ()

Returns resource description

public *string* **__toString** ()

Magic method __toString

2.48.6 Class Phalcon\Acl\Role

implements Phalcon\Acl\RoleInterface

This class defines role entity and its description

Methods

public **__construct** (*string* $name, [*string* $description])

Phalcon\Acl\Role description

public *string* **getName** ()

Returns the role name

public *string* **getDescription** ()

Returns role description

public *string* **__toString** ()

Magic method __toString

2.48.7 Class Phalcon\Annotations\Adapter

This is the base class for Phalcon\Annotations adapters

Methods

public **setReader** (*Phalcon\Annotations\ReaderInterface* $reader)

Sets the annotations parser

public *Phalcon\Annotations\ReaderInterface* **getReader** ()

Returns the annotation reader

public *Phalcon\Annotations\Reflection* **get** (*string|object* $className)

Parses or retrieves all the annotations found in a class

public *array* **getMethods** (*string* $className)

Returns the annotations found in all the class' methods

public *Phalcon\Annotations\Collection* **getMethod** (*string* $className, *string* $methodName)

Returns the annotations found in a specific method

public *array* **getProperties** (*string* $className)

Returns the annotations found in all the class' methods

public *Phalcon\Annotations\Collection* **getProperty** (*string* $className, *string* $propertyName)

Returns the annotations found in a specific property

2.48.8 Class Phalcon\Annotations\Adapter\Apc

extends Phalcon\Annotations\Adapter

implements Phalcon\Annotations\AdapterInterface

Phalcon\Annotations\Adapter\Files Stores the parsed annotations in APC. This adapter is the suitable for production

```
<?php

 $annotations = new \Phalcon\Annotations\Adapter\Apc();
```

Methods

public *array* **read** (*string* $key)

Reads parsed annotations from Apc

public **write** (*string* $key, *array* $data)

Writes parsed annotations to APC

public **setReader** (*Phalcon\Annotations\ReaderInterface* $reader) inherited from Phalcon\Annotations\Adapter

Sets the annotations parser

public *Phalcon\Annotations\ReaderInterface* **getReader** () inherited from Phalcon\Annotations\Adapter

Returns the annotation reader

public *Phalcon\Annotations\Reflection* **get** (*string|object* $className) inherited from Phalcon\Annotations\Adapter

Parses or retrieves all the annotations found in a class

public *array* **getMethods** (*string* $className) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in all the class' methods

public *Phalcon\Annotations\Collection* **getMethod** (*string* $className, *string* $methodName) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in a specific method

public *array* **getProperties** (*string* $className) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in all the class' methods

public *Phalcon\Annotations\Collection* **getProperty** (*string* $className, *string* $propertyName) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in a specific property

2.48.9 Class Phalcon\Annotations\Adapter\Files

extends Phalcon\Annotations\Adapter

implements Phalcon\Annotations\AdapterInterface

Stores the parsed annotations in diles. This adapter is the suitable for production

```
<?php

$annotations = new \Phalcon\Annotations\Adapter\Files(array(
   'metaDataDir' => 'app/cache/metadata/'
));
```

Methods

public **__construct** ([*array* $options])

Phalcon\Annotations\Adapter\Files constructor

public *array* **read** (*string* $key)

Reads parsed annotations from files

public **write** (*string* $key, *array* $data)

Writes parsed annotations to files

public **setReader** (*Phalcon\Annotations\ReaderInterface* $reader) inherited from Phalcon\Annotations\Adapter

Sets the annotations parser

public *Phalcon\Annotations\ReaderInterface* **getReader** () inherited from Phalcon\Annotations\Adapter

Returns the annotation reader

public *Phalcon\Annotations\Reflection* **get** (*string|object* $className) inherited from Phalcon\Annotations\Adapter

Parses or retrieves all the annotations found in a class

public *array* **getMethods** (*string* $className) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in all the class' methods

public *Phalcon\Annotations\Collection* **getMethod** (*string* $className, *string* $methodName) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in a specific method

public *array* **getProperties** (*string* $className) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in all the class' methods

public *Phalcon\Annotations\Collection* **getProperty** (*string* $className, *string* $propertyName) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in a specific property

2.48.10 Class Phalcon\Annotations\Adapter\Memory

extends Phalcon\Annotations\Adapter

implements Phalcon\Annotations\AdapterInterface

Stores the parsed annotations in memory. This adapter is the suitable for development/testing

Methods

public *array* **read** (*string* $key)

Reads meta-data from memory

public **write** (*string* $key, *array* $data)

Writes the meta-data to files

public **setReader** (*Phalcon\Annotations\ReaderInterface* $reader) inherited from Phalcon\Annotations\Adapter

Sets the annotations parser

public *Phalcon\Annotations\ReaderInterface* **getReader** () inherited from Phalcon\Annotations\Adapter

Returns the annotation reader

public *Phalcon\Annotations\Reflection* **get** (*string|object* $className) inherited from Phalcon\Annotations\Adapter

Parses or retrieves all the annotations found in a class

public *array* **getMethods** (*string* $className) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in all the class' methods

public *Phalcon\Annotations\Collection* **getMethod** (*string* $className, *string* $methodName) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in a specific method

public *array* **getProperties** (*string* $className) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in all the class' methods

public *Phalcon\Annotations\Collection* **getProperty** (*string* $className, *string* $propertyName) inherited from Phalcon\Annotations\Adapter

Returns the annotations found in a specific property

2.48.11 Class Phalcon\Annotations\Annotation

Represents a single annotation in an annotations collection

Methods

public **__construct** (*array* $reflectionData)

Phalcon\Annotations\Annotation constructor

public *string* **getName** ()

Returns the annotation's name

public *mixed* **getExpression** (*array* $expr)

Resolves an annotation expression

public *array* **getExprArguments** ()

Returns the expression arguments without resolving

public *array* **getArguments** ()

Returns the expression arguments

public *int* **numberArguments** ()

Returns the number of arguments that the annotation has

public *mixed* **getArgument** (*unknown* $position)

Returns an argument in an specific position

public *mixed* **hasArgument** (*unknown* $position)

Returns an argument in an specific position

public *mixed* **getNamedParameter** (*string* $name)

Returns a named argument

public *boolean* **hasNamedArgument** (*unknown* $name)

Checks if the annotation has a specific named argument

2.48.12 Class Phalcon\Annotations\Collection

implements Iterator, Traversable, Countable

Represents a collection of annotations. This class allows to traverse a group of annotations easily

```
<?php

 //Traverse annotations
 foreach ($classAnnotations as $annotation) {
     echo 'Name=', $annotation->getName(), PHP_EOL;
 }

 //Check if the annotations has an specific
 var_dump($classAnnotations->has('Cacheable'));

 //Get an specific annotation in the collection
 $annotation = $classAnnotations->get('Cacheable');
```

Methods

public **__construct** ([*array* $reflectionData])

Phalcon\Annotations\Collection constructor

public *int* **count** ()

Returns the number of annotations in the collection

public **rewind** ()

Rewinds the internal iterator

public *Phalcon\Annotations\Annotation* **current** ()

Returns the current annotation in the iterator

public *int* **key** ()

Returns the current position/key in the iterator

public **next** ()

Moves the internal iteration pointer to the next position

public *boolean* **valid** ()

Check if the current annotation in the iterator is valid

public *Phalcon\Annotations\Annotation* [] **getAnnotations** ()

Returns the internal annotations as an array

public *Phalcon\Annotations\Annotation* **get** (*string* $name)

Returns an annotation by its name

public *boolean* **has** (*string* $name)

Check if an annotation exists in a collection

2.48.13 Class Phalcon\Annotations\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.14 Class Phalcon\Annotations\Reader

implements Phalcon\Annotations\ReaderInterface

Parses docblocks returning an array with the found annotations

Methods

public *array* **parse** (*string* $className)

Reads annotations from the class dockblocks, its methods and/or properties

public static *array* **parseDocBlock** (*string* $docBlock, [*unknown* $file], [*unknown* $line])

Parses a raw doc block returning the annotations found

2.48.15 Class Phalcon\Annotations\Reflection

Allows to manipulate the annotations reflection in an OO manner

```
<?php

 //Parse the annotations in a class
 $reader = new \Phalcon\Annotations\Reader();
 $parsing = $reader->parse('MyComponent');

 //Create the reflection
 $reflection = new \Phalcon\Annotations\Reflection($parsing);

 //Get the annotations in the class docblock
 $classAnnotations = $reflection->getClassAnnotations();
```

Methods

public **__construct** ([*array* $reflectionData])

Phalcon\Annotations\Reflection constructor

public *Phalcon\Annotations\Collection* **getClassAnnotations** ()

Returns the annotations found in the class docblock

public *Phalcon\Annotations\Collection* [] **getMethodsAnnotations** ()

Returns the annotations found in the methods' docblocks

public *Phalcon\Annotations\Collection* [] **getPropertiesAnnotations** ()

Returns the annotations found in the properties' docblocks

public *array* **getReflectionData** ()

Returns the raw parsing intermediate definitions used to construct the reflection

public static *array $data* **__set_state** (*unknown* $data)

Restores the state of a Phalcon\Annotations\Reflection variable export

2.48.16 Class Phalcon\CLI\Console

implements Phalcon\DINjectionAwareInterface, *Phalcon\Events\EventsAwareInterface*

This component allows to create CLI applications using Phalcon

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager)

Sets the events manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the internal event manager

public **registerModules** (*array* $modules)

Register an array of modules present in the console

```
<?php

$application->registerModules(array(
    'frontend' => array(
            'className' => 'Multiple\Frontend\Module',
            'path' => '../apps/frontend/Module.php'
    ),
    'backend' => array(
            'className' => 'Multiple\Backend\Module',
            'path' => '../apps/backend/Module.php'
```

```
    )
));
```

public **addModules** (*array* $modules)

Merge modules with the existing ones

```
<?php

$application->addModules(array(
    'admin' => array(
            'className' => 'Multiple\Admin\Module',
            'path' => '../apps/admin/Module.php'
    )
));
```

public *array* **getModules** ()

Return the modules registered in the console

public *mixed* **handle** ([*array* $arguments])

Handle the whole command-line tasks

2.48.17 Class Phalcon\CLI\Console\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.18 Class Phalcon\CLI\Dispatcher

extends Phalcon\Dispatcher

implements Phalcon\Events\EventsAwareInterface, Phalcon\DINjectionAwareInterface, Phalcon\DispatcherInterface

Dispatching is the process of taking the command-line arguments, extracting the module name, task name, action name, and optional parameters contained in it, and then instantiating a task and calling an action on it.

```
<?php

$di = new Phalcon\DI();

$dispatcher = new Phalcon\CLI\Dispatcher();

  $dispatcher->setDI($di);

$dispatcher->setTaskName('posts');
$dispatcher->setActionName('index');
$dispatcher->setParams(array());

$handle = $dispatcher->dispatch();
```

Constants

integer **EXCEPTION_NO_DI**

integer **EXCEPTION_CYCLIC_ROUTING**

integer **EXCEPTION_HANDLER_NOT_FOUND**

integer **EXCEPTION_INVALID_HANDLER**

integer **EXCEPTION_INVALID_PARAMS**

integer **EXCEPTION_ACTION_NOT_FOUND**

Methods

public **setTaskSuffix** (*string* $taskSuffix)

Sets the default task suffix

public **setDefaultTask** (*string* $taskName)

Sets the default task name

public **setTaskName** (*string* $taskName)

Sets the task name to be dispatched

public *string* **getTaskName** ()

Gets last dispatched task name

protected **_throwDispatchException** ()

Throws an internal exception

public *Phalcon\CLI\Task* **getLastTask** ()

Returns the lastest dispatched controller

public *Phalcon\CLI\Task* **getActiveTask** ()

Returns the active task in the dispatcher

public **__construct** () inherited from Phalcon\Dispatcher

Phalcon\Dispatcher constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\Dispatcher

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\Dispatcher

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\Dispatcher

Sets the events manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\Dispatcher

Returns the internal event manager

public **setActionSuffix** (*string* $actionSuffix) inherited from Phalcon\Dispatcher

Sets the default action suffix

public **setNamespaceName** (*string* $namespaceName) inherited from Phalcon\Dispatcher

Sets a namespace to be prepended to the handler name

public *string* **getNamespaceName** () inherited from Phalcon\Dispatcher

Gets a namespace to be prepended to the current handler name

public **setDefaultNamespace** (*string* $namespace) inherited from Phalcon\Dispatcher

Sets the default namespace

public *string* **getDefaultNamespace** () inherited from Phalcon\Dispatcher

Returns the default namespace

public **setDefaultAction** (*string* $actionName) inherited from Phalcon\Dispatcher

Sets the default action name

public **setActionName** (*string* $actionName) inherited from Phalcon\Dispatcher

Sets the action name to be dispatched

public *string* **getActionName** () inherited from Phalcon\Dispatcher

Gets the lastest dispatched action name

public **setParams** (*array* $params) inherited from Phalcon\Dispatcher

Sets action params to be dispatched

public *array* **getParams** () inherited from Phalcon\Dispatcher

Gets action params

public **setParam** (*mixed* $param, *mixed* $value) inherited from Phalcon\Dispatcher

Set a param by its name or numeric index

public *mixed* **getParam** (*mixed* $param, [*string|array* $filters], [*mixed* $defaultValue]) inherited from Phalcon\Dispatcher

Gets a param by its name or numeric index

public *string* **getActiveMethod** () inherited from Phalcon\Dispatcher

Returns the current method to be/executed in the dispatcher

public *boolean* **isFinished** () inherited from Phalcon\Dispatcher

Checks if the dispatch loop is finished or has more pendent controllers/tasks to disptach

public **setReturnedValue** (*mixed* $value) inherited from Phalcon\Dispatcher

Sets the latest returned value by an action manually

public *mixed* **getReturnedValue** () inherited from Phalcon\Dispatcher

Returns value returned by the lastest dispatched action

public *object* **dispatch** () inherited from Phalcon\Dispatcher

Dispatches a handle action taking into account the routing parameters

public **forward** (*array* $forward) inherited from Phalcon\Dispatcher

Forwards the execution flow to another controller/action

2.48.19 Class Phalcon\CLI\Dispatcher\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.20 Class Phalcon\CLI\Router

implements Phalcon\DINjectionAwareInterface

Phalcon\CLI\Router is the standard framework router. Routing is the process of taking a command-line arguments and decomposing it into parameters to determine which module, task, and action of that task should receive the request

```
<?php

$router = new Phalcon\CLI\Router();
$router->handle(array());
echo $router->getTaskName();
```

Methods

public **__construct** ()

Phalcon\CLI\Router constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

public **setDefaultModule** (*string* $moduleName)

Sets the name of the default module

public **setDefaultTask** (*string* $taskName)

Sets the default controller name

public **setDefaultAction** (*string* $actionName)

Sets the default action name

public **handle** ([*array* $arguments])

Handles routing information received from command-line arguments

public *string* **getModuleName** ()

Returns proccesed module name

public *string* **getTaskName** ()

Returns proccesed task name

public *string* **getActionName** ()

Returns proccesed action name

public *array* **getParams** ()

Returns proccesed extra params

2.48.21 Class Phalcon\CLI\Router\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.22 Class Phalcon\CLI\Task

extends Phalcon\DI\Injectable

implements Phalcon\Events\EventsAwareInterface, *Phalcon\DI\InjectionAwareInterface*

Every command-line task should extend this class that encapsulates all the task functionality A task can be used to run "tasks" such as migrations, cronjobs, unit-tests, or anything that you want. The Task class should at least have a "mainAction" method

```
<?php

class HelloTask extends \Phalcon\CLI\Task
{

  //This action will be executed by default
  public function mainAction()
  {

  }

  public function findAction()
  {

  }

}
```

Methods

final public **__construct** ()

Phalcon\CLI\Task constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI\Injectable

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

public **__get** (*string* $propertyName) inherited from Phalcon\DI\Injectable

Magic method __get

2.48.23 Class Phalcon\Cache\Backend

This class implements common functionality for backend adapters. A backend cache adapter may extend this class

Methods

public **__construct** (*Phalcon\Cache\FrontendInterface* $frontend, [*array* $options])

Phalcon\Cache\Backend constructor

public *mixed* **start** (*int|string* $keyName, [*long* $lifetime])

Starts a cache. The $keyname allows to identify the created fragment

public **stop** ([*boolean* $stopBuffer])

Stops the frontend without store any cached content

public *mixed* **getFrontend** ()

Returns front-end instance adapter related to the back-end

public *array* **getOptions** ()

Returns the backend options

public *boolean* **isFresh** ()

Checks whether the last cache is fresh or cached

public *boolean* **isStarted** ()

Checks whether the cache has starting buffering or not

public **setLastKey** (*string* $lastKey)

Sets the last key used in the cache

public *string* **getLastKey** ()

Gets the last key stored by the cache

2.48.24 Class Phalcon\Cache\Backend\Apc

extends Phalcon\Cache\Backend

implements Phalcon\Cache\BackendInterface

Allows to cache output fragments, PHP data and raw data using a memcache backend

```
<?php

//Cache data for 2 days
$frontCache = new Phalcon\Cache\Frontend\Data(array(
    'lifetime' => 172800
));

  $cache = new Phalcon\Cache\Backend\Apc($frontCache, array(
      'prefix' => 'app-data'
  ));

//Cache arbitrary data
$cache->save('my-data', array(1, 2, 3, 4, 5));

//Get data
$data = $cache->get('my-data');
```

Methods

public *mixed* **get** (*string* $keyName, [*long* $lifetime])

Returns a cached content

public **save** ([*string* $keyName], [*string* $content], [*long* $lifetime], [*boolean* $stopBuffer])

Stores cached content into the APC backend and stops the frontend

public *boolean* **delete** (*string* $keyName)

Deletes a value from the cache by its key

public *array* **queryKeys** ([*string* $prefix])

Query the existing cached keys

public *boolean* **exists** ([*string* $keyName], [*long* $lifetime])

Checks if cache exists and it hasn't expired

public **__construct** (*Phalcon\Cache\FrontendInterface* $frontend, [*array* $options]) inherited from Phalcon\Cache\Backend

Phalcon\Cache\Backend constructor

public *mixed* **start** (*int|string* $keyName, [*long* $lifetime]) inherited from Phalcon\Cache\Backend

Starts a cache. The $keyname allows to identify the created fragment

public **stop** ([*boolean* $stopBuffer]) inherited from Phalcon\Cache\Backend

Stops the frontend without store any cached content

public *mixed* **getFrontend** () inherited from Phalcon\Cache\Backend

Returns front-end instance adapter related to the back-end

public *array* **getOptions** () inherited from Phalcon\Cache\Backend

Returns the backend options

public *boolean* **isFresh** () inherited from Phalcon\Cache\Backend

Checks whether the last cache is fresh or cached

public *boolean* **isStarted** () inherited from Phalcon\Cache\Backend

Checks whether the cache has starting buffering or not

public **setLastKey** (*string* $lastKey) inherited from Phalcon\Cache\Backend

Sets the last key used in the cache

public *string* **getLastKey** () inherited from Phalcon\Cache\Backend

Gets the last key stored by the cache

2.48.25 Class Phalcon\Cache\Backend\File

extends Phalcon\Cache\Backend

implements Phalcon\Cache\BackendInterface

Allows to cache output fragments using a file backend

```
<?php

//Cache the file for 2 days
$frontendOptions = array(
    'lifetime' => 172800
);

  //Create a output cache
  $frontCache = \Phalcon\Cache\Frontend\Output($frontOptions);

//Set the cache directory
```

```
$backendOptions = array(
    'cacheDir' => '../app/cache/'
);

  //Create the File backend
  $cache = new \Phalcon\Cache\Backend\File($frontCache, $backendOptions);

$content = $cache->start('my-cache');
if ($content === null) {
    echo '<h1>', time(), '</h1>';
    $cache->save();
} else {
    echo $content;
}
```

Methods

public **__construct** (*Phalcon\Cache\FrontendInterface* $frontend, [*array* $options])

Phalcon\Cache\Backend\File constructor

public *mixed* **get** (*int|string* $keyName, [*long* $lifetime])

Returns a cached content

public **save** ([*int|string* $keyName], [*string* $content], [*long* $lifetime], [*boolean* $stopBuffer])

Stores cached content into the file backend and stops the frontend

public *boolean* **delete** (*int|string* $keyName)

Deletes a value from the cache by its key

public *array* **queryKeys** ([*string* $prefix])

Query the existing cached keys

public *boolean* **exists** ([*string* $keyName], [*long* $lifetime])

Checks if cache exists and it isn't expired

public *mixed* **start** (*int|string* $keyName, [*long* $lifetime]) inherited from Phalcon\Cache\Backend

Starts a cache. The $keyname allows to identify the created fragment

public **stop** ([*boolean* $stopBuffer]) inherited from Phalcon\Cache\Backend

Stops the frontend without store any cached content

public *mixed* **getFrontend** () inherited from Phalcon\Cache\Backend

Returns front-end instance adapter related to the back-end

public *array* **getOptions** () inherited from Phalcon\Cache\Backend

Returns the backend options

public *boolean* **isFresh** () inherited from Phalcon\Cache\Backend

Checks whether the last cache is fresh or cached

public *boolean* **isStarted** () inherited from Phalcon\Cache\Backend

Checks whether the cache has starting buffering or not

public **setLastKey** (*string* $lastKey) inherited from Phalcon\Cache\Backend

Sets the last key used in the cache

public *string* **getLastKey** () inherited from Phalcon\Cache\Backend

Gets the last key stored by the cache

2.48.26 Class Phalcon\Cache\Backend\Memcache

extends Phalcon\Cache\Backend

implements Phalcon\Cache\BackendInterface

Allows to cache output fragments, PHP data or raw data to a memcache backend This adapter uses the special memcached key "_PHCM" to store all the keys internally used by the adapter

```
<?php

 // Cache data for 2 days
 $frontCache = new Phalcon\Cache\Frontend\Data(array(
    "lifetime" => 172800
 ));

 //Create the Cache setting memcached connection options
 $cache = new Phalcon\Cache\Backend\Memcache($frontCache, array(
    'host' => 'localhost',
    'port' => 11211,
    'persistent' => false
 ));

 //Cache arbitrary data
 $cache->save('my-data', array(1, 2, 3, 4, 5));

 //Get data
 $data = $cache->get('my-data');
```

Methods

public **__construct** (*Phalcon\Cache\FrontendInterface* $frontend, [*array* $options])

Phalcon\Cache\Backend\Memcache constructor

protected **_connect** ()

Create internal connection to memcached

public *mixed* **get** (*int|string* $keyName, [*long* $lifetime])

Returns a cached content

public **save** ([*int|string* $keyName], [*string* $content], [*long* $lifetime], [*boolean* $stopBuffer])

Stores cached content into the Memcached backend and stops the frontend

public *boolean* **delete** (*int|string* $keyName)

Deletes a value from the cache by its key

public *array* **queryKeys** ([*string* $prefix])

Query the existing cached keys

public *boolean* **exists** ([*string* $keyName], [*long* $lifetime])

Checks if cache exists and it hasn't expired

public *mixed* **start** (*int|string* $keyName, [*long* $lifetime]) inherited from Phalcon\Cache\Backend

Starts a cache. The $keyname allows to identify the created fragment

public **stop** ([*boolean* $stopBuffer]) inherited from Phalcon\Cache\Backend

Stops the frontend without store any cached content

public *mixed* **getFrontend** () inherited from Phalcon\Cache\Backend

Returns front-end instance adapter related to the back-end

public *array* **getOptions** () inherited from Phalcon\Cache\Backend

Returns the backend options

public *boolean* **isFresh** () inherited from Phalcon\Cache\Backend

Checks whether the last cache is fresh or cached

public *boolean* **isStarted** () inherited from Phalcon\Cache\Backend

Checks whether the cache has starting buffering or not

public **setLastKey** (*string* $lastKey) inherited from Phalcon\Cache\Backend

Sets the last key used in the cache

public *string* **getLastKey** () inherited from Phalcon\Cache\Backend

Gets the last key stored by the cache

2.48.27 Class Phalcon\Cache\Backend\Memory

extends Phalcon\Cache\Backend

implements Phalcon\Cache\BackendInterface

Stores content in memory. Data is lost when the request is finished

```
<?php

//Cache data
$frontCache = new Phalcon\Cache\Frontend\Data();

  $cache = new Phalcon\Cache\Backend\Memory($frontCache);

//Cache arbitrary data
$cache->save('my-data', array(1, 2, 3, 4, 5));

//Get data
$data = $cache->get('my-data');
```

Methods

public *mixed* **get** (*string* $keyName, [*long* $lifetime])

Returns a cached content

public **save** ([*string* $keyName], [*string* $content], [*long* $lifetime], [*boolean* $stopBuffer])

Stores cached content into the APC backend and stops the frontend

public *boolean* **delete** (*string* $keyName)

Deletes a value from the cache by its key

public *array* **queryKeys** ([*string* $prefix])

Query the existing cached keys

public *boolean* **exists** ([*string* $keyName], [*long* $lifetime])

Checks if cache exists and it hasn't expired

public **__construct** (*Phalcon\Cache\FrontendInterface* $frontend, [*array* $options]) inherited from Phalcon\Cache\Backend

Phalcon\Cache\Backend constructor

public *mixed* **start** (*int|string* $keyName, [*long* $lifetime]) inherited from Phalcon\Cache\Backend

Starts a cache. The $keyname allows to identify the created fragment

public **stop** ([*boolean* $stopBuffer]) inherited from Phalcon\Cache\Backend

Stops the frontend without store any cached content

public *mixed* **getFrontend** () inherited from Phalcon\Cache\Backend

Returns front-end instance adapter related to the back-end

public *array* **getOptions** () inherited from Phalcon\Cache\Backend

Returns the backend options

public *boolean* **isFresh** () inherited from Phalcon\Cache\Backend

Checks whether the last cache is fresh or cached

public *boolean* **isStarted** () inherited from Phalcon\Cache\Backend

Checks whether the cache has starting buffering or not

public **setLastKey** (*string* $lastKey) inherited from Phalcon\Cache\Backend

Sets the last key used in the cache

public *string* **getLastKey** () inherited from Phalcon\Cache\Backend

Gets the last key stored by the cache

2.48.28 Class Phalcon\Cache\Backend\Mongo

extends Phalcon\Cache\Backend

implements Phalcon\Cache\BackendInterface

Allows to cache output fragments, PHP data or raw data to a MongoDb backend

```
<?php

 // Cache data for 2 days
 $frontCache = new Phalcon\Cache\Frontend\Base64(array(
    "lifetime" => 172800
 ));

 //Create a MongoDB cache
 $cache = new Phalcon\Cache\Backend\Mongo($frontCache, array(
```

```
    'server' => "mongodb://localhost",
      'db' => 'caches',
    'collection' => 'images'
));

//Cache arbitrary data
$cache->save('my-data', file_get_contents('some-image.jpg'));

//Get data
$data = $cache->get('my-data');
```

Methods

public **__construct** (*Phalcon\Cache\FrontendInterface* $frontend, [*array* $options])

Phalcon\Cache\Backend\Mongo constructor

protected *MongoCollection* **_getCollection** ()

Returns a MongoDb collection based on the backend parameters

public *mixed* **get** (*int|string* $keyName, [*long* $lifetime])

Returns a cached content

public **save** ([*int|string* $keyName], [*string* $content], [*long* $lifetime], [*boolean* $stopBuffer])

Stores cached content into the Mongo backend and stops the frontend

public *boolean* **delete** (*int|string* $keyName)

Deletes a value from the cache by its key

public *array* **queryKeys** ([*string* $prefix])

Query the existing cached keys

public *boolean* **exists** ([*string* $keyName], [*long* $lifetime])

Checks if cache exists and it hasn't expired

public *mixed* **start** (*int|string* $keyName, [*long* $lifetime]) inherited from Phalcon\Cache\Backend

Starts a cache. The $keyname allows to identify the created fragment

public **stop** ([*boolean* $stopBuffer]) inherited from Phalcon\Cache\Backend

Stops the frontend without store any cached content

public *mixed* **getFrontend** () inherited from Phalcon\Cache\Backend

Returns front-end instance adapter related to the back-end

public *array* **getOptions** () inherited from Phalcon\Cache\Backend

Returns the backend options

public *boolean* **isFresh** () inherited from Phalcon\Cache\Backend

Checks whether the last cache is fresh or cached

public *boolean* **isStarted** () inherited from Phalcon\Cache\Backend

Checks whether the cache has starting buffering or not

public **setLastKey** (*string* $lastKey) inherited from Phalcon\Cache\Backend

Sets the last key used in the cache

public *string* **getLastKey** () inherited from Phalcon\Cache\Backend

Gets the last key stored by the cache

2.48.29 Class Phalcon\Cache\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.30 Class Phalcon\Cache\Frontend\Base64

implements Phalcon\Cache\FrontendInterface

Allows to cache data converting/deconverting them to base64. This adapters uses the base64_encode/base64_decode PHP's functions

```php
<?php

 // Cache the files for 2 days using a Base64 frontend
 $frontCache = new Phalcon\Cache\Frontend\Base64(array(
    "lifetime" => 172800
 ));

 //Create a MongoDB cache
 $cache = new Phalcon\Cache\Backend\Mongo($frontCache, array(
    'server' => "mongodb://localhost",
      'db' => 'caches',
    'collection' => 'images'
 ));

 // Try to get cached image
 $cacheKey = 'some-image.jpg.cache';
 $image    = $cache->get($cacheKey);
 if ($image === null) {

     // Store the image in the cache
     $cache->save($cacheKey, file_put_contents('tmp-dir/some-image.jpg'));
 }

 header('Content-Type: image/jpeg');
 echo $image;
```

Methods

public **__construct** ([*array* $frontendOptions])

Phalcon\Cache\Frontend\Base64 constructor

public *integer* **getLifetime** ()

Returns the cache lifetime

public *boolean* **isBuffering** ()

Check whether if frontend is buffering output

public **start** ()

Starts output frontend. Actually, does nothing

public *string* **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (*mixed* $data)

Serializes data before storing it

public **afterRetrieve** (*mixed* $data)

Unserializes data after retrieving it

2.48.31 Class Phalcon\Cache\Frontend\Data

implements Phalcon\Cache\FrontendInterface

Allows to cache native PHP data in a serialized form

```
<?php

 // Cache the files for 2 days using a Data frontend
 $frontCache = new Phalcon\Cache\Frontend\Data(array(
    "lifetime" => 172800
 ));

 // Create the component that will cache "Data" to a "File" backend
 // Set the cache file directory - important to keep the "/" at the end of
 // of the value for the folder
 $cache = new Phalcon\Cache\Backend\File($frontCache, array(
     "cacheDir" => "../app/cache/"
 ));

 // Try to get cached records
 $cacheKey = 'robots_order_id.cache';
 $robots     = $cache->get($cacheKey);
 if ($robots === null) {

     // $robots is null due to cache expiration or data does not exist
     // Make the database call and populate the variable
     $robots = Robots::find(array("order" => "id"));

     // Store it in the cache
     $cache->save($cacheKey, $robots);
 }

 // Use $robots :)
 foreach ($robots as $robot) {
    echo $robot->name, "\n";
 }
```

Methods

public **__construct** ([*array* $frontendOptions])

Phalcon\Cache\Frontend\Data constructor

public *int* **getLifetime** ()

Returns cache lifetime

public *boolean* **isBuffering** ()

Check whether if frontend is buffering output

public **start** ()

Starts output frontend. Actually, does nothing

public *string* **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (*mixed* $data)

Serializes data before storing it

public **afterRetrieve** (*mixed* $data)

Unserializes data after retrieving it

2.48.32 Class Phalcon\Cache\Frontend\None

implements Phalcon\Cache\FrontendInterface

Discards any kind of frontend data input. This frontend does not have expiration time or any other options

```
<?php

 //Create a None Cache
 $frontCache = new Phalcon\Cache\Frontend\None();

 // Create the component that will cache "Data" to a "Memcached" backend
 // Memcached connection settings
 $cache = new Phalcon\Cache\Backend\Memcached($frontCache, array(
     "host" => "localhost",
     "port" => "11211"
 ));

 // This Frontend always return the data as it's returned by the backend
 $cacheKey = 'robots_order_id.cache';
 $robots    = $cache->get($cacheKey);
 if ($robots === null) {

     // This cache doesn't perform any expiration checking, so the data is always expired
     // Make the database call and populate the variable
     $robots = Robots::find(array("order" => "id"));

     $cache->save($cacheKey, $robots);
 }

 // Use $robots :)
 foreach ($robots as $robot) {
    echo $robot->name, "\n";
 }
```

Methods

public **__construct** ([*array* $frontendOptions])

Phalcon\Cache\Frontend\None constructor

public *int* **getLifetime** ()

Returns cache lifetime, always one second expiring content

public *boolean* **isBuffering** ()

Check whether if frontend is buffering output, always false

public **start** ()

Starts output frontend

public *string* **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (*mixed* $data)

Prepare data to be stored

public **afterRetrieve** (*mixed* $data)

Prepares data to be retrieved to user

2.48.33 Class Phalcon\Cache\Frontend\Output

implements Phalcon\Cache\FrontendInterface

Allows to cache output fragments captured with ob_* functions

```
<?php

 //Create an Output frontend. Cache the files for 2 days
 $frontCache = new Phalcon\Cache\Frontend\Output(array(
   "lifetime" => 172800
 ));

 // Create the component that will cache from the "Output" to a "File" backend
 // Set the cache file directory - it's important to keep the "/" at the end of
 // the value for the folder
 $cache = new Phalcon\Cache\Backend\File($frontCache, array(
     "cacheDir" => "../app/cache/"
 ));

 // Get/Set the cache file to ../app/cache/my-cache.html
 $content = $cache->start("my-cache.html");

 // If $content is null then the content will be generated for the cache
 if ($content === null) {

     //Print date and time
     echo date("r");

     //Generate a link to the sign-up action
     echo Phalcon\Tag::linkTo(
         array(
             "user/signup",
             "Sign Up",
             "class" => "signup-button"
         )
     );

     // Store the output into the cache file
     $cache->save();

 } else {
```

```
    // Echo the cached output
    echo $content;
}
```

Methods

public **__construct** ([*array* $frontendOptions])

Phalcon\Cache\Frontend\Output constructor

public *integer* **getLifetime** ()

Returns cache lifetime

public *boolean* **isBuffering** ()

Check whether if frontend is buffering output

public **start** ()

Starts output frontend

public *string* **getContent** ()

Returns output cached content

public **stop** ()

Stops output frontend

public **beforeStore** (*mixed* $data)

Prepare data to be stored

public **afterRetrieve** (*mixed* $data)

Prepares data to be retrieved to user

2.48.34 Class Phalcon\Cache\Multiple

Allows to read to chained backends writing to multiple backends

Methods

public **__construct** ([*Phalcon\Cache\BackendInterface[]* $backends])

Phalcon\Cache\Multiple constructor

public *Phalcon\Cache\Multiple* **push** (*Phalcon\Cache\BackendInterface* $backend)

Adds a backend

public *mixed* **get** (*string* $keyName, [*long* $lifetime])

Returns a cached content reading the internal backends

public *mixed* **start** (*int|string* $keyName, [*long* $lifetime])

Starts every backend

public **save** ([*string* $keyName], [*string* $content], [*long* $lifetime], [*boolean* $stopBuffer])

Stores cached content into the APC backend and stops the frontend

public *boolean* **delete** (*int|string* $keyName)

Deletes a value from each backend

public *boolean* **exists** ([*string* $keyName], [*long* $lifetime])

Checks if cache exists in at least one backend

2.48.35 Class Phalcon\Config

implements ArrayAccess

Phalcon\Config is designed to simplify the access to, and the use of, configuration data within applications. It provides a nested object property based user interface for accessing this configuration data within application code.

```
<?php

$config = new Phalcon\Config(array(
    "database" => array(
            "adapter" => "Mysql",
            "host" => "localhost",
            "username" => "scott",
            "password" => "cheetah",
            "name" => "test_db"
    ),
    "phalcon" => array(
            "controllersDir" => "../app/controllers/",
            "modelsDir" => "../app/models/",
            "viewsDir" => "../app/views/"
    )
 ));
```

Methods

public **__construct** ([*array* $arrayConfig])

Phalcon\Config constructor

public *boolean* **offsetExists** (*string* $index)

Allows to check whether an attribute is defined using the array-syntax

```
<?php

 var_dump(isset($config['database']));
```

public *mixed* **get** (*string* $index, [*mixed* $defaultValue])

Gets an attribute from the configuration, if the attribute isn't defined returns null If the value is exactly null or is not defined the default value will be used instead

```
<?php

 echo $config->get('controllersDir', '../app/controllers/');
```

public *string* **offsetGet** (*string* $index)

Gets an attribute using the array-syntax

```
<?php

 print_r($config['database']);
```

public **offsetSet** (*string* $index, *mixed* $value)

Sets an attribute using the array-syntax

```
<?php

 $config['database'] = array('type' => 'Sqlite');
```

public **offsetUnset** (*string* $index)

Unsets an attribute using the array-syntax

```
<?php

 unset($config['database']);
```

public **merge** (*Phalcon\Config* $config)

Merges a configuration into the current one

```
<?php

$appConfig = new Phalcon\Config(array('database' => array('host' => 'localhost')));
$globalConfig->merge($config2);
```

public *array* **toArray** ()

Converts recursively the object to an array

```
<?php

print_r($config->toArray());
```

public static *Phalcon\Config* **__set_state** (*array* $data)

Restores the state of a Phalcon\Config object

2.48.36 Class Phalcon\Config\Adapter\Ini

extends Phalcon\Config

implements ArrayAccess

Reads ini files and convert it to Phalcon\Config objects. Given the next configuration file:

```
<?php

[database]
adapter = Mysql
host = localhost
username = scott
password = cheetah
name = test_db

[phalcon]
controllersDir = "../app/controllers/"
modelsDir = "../app/models/"
```

```
  viewsDir = "../app/views/"

You can read it as follows:

<?php

$config = new Phalcon\Config\Adapter\Ini("path/config.ini");
echo $config->phalcon->controllersDir;
echo $config->database->username;
```

Methods

public **__construct** (*string* $filePath)

Phalcon\Config\Adapter\Ini constructor

public *boolean* **offsetExists** (*string* $index) inherited from Phalcon\Config

Allows to check whether an attribute is defined using the array-syntax

```
<?php

 var_dump(isset($config['database']));
```

public *mixed* **get** (*string* $index, [*mixed* $defaultValue]) inherited from Phalcon\Config

Gets an attribute from the configuration, if the attribute isn't defined returns null If the value is exactly null or is not defined the default value will be used instead

```
<?php

 echo $config->get('controllersDir', '../app/controllers/');
```

public *string* **offsetGet** (*string* $index) inherited from Phalcon\Config

Gets an attribute using the array-syntax

```
<?php

 print_r($config['database']);
```

public **offsetSet** (*string* $index, *mixed* $value) inherited from Phalcon\Config

Sets an attribute using the array-syntax

```
<?php

 $config['database'] = array('type' => 'Sqlite');
```

public **offsetUnset** (*string* $index) inherited from Phalcon\Config

Unsets an attribute using the array-syntax

```
<?php

 unset($config['database']);
```

public **merge** (*Phalcon\Config* $config) inherited from Phalcon\Config

Merges a configuration into the current one

```
<?php

$appConfig = new Phalcon\Config(array('database' => array('host' => 'localhost')));
$globalConfig->merge($config2);
```

public *array* **toArray** () inherited from Phalcon\Config

Converts recursively the object to an array

```
<?php

print_r($config->toArray());
```

public static *Phalcon\Config* **__set_state** (*array* $data) inherited from Phalcon\Config

Restores the state of a Phalcon\Config object

2.48.37 Class Phalcon\Config\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.38 Class Phalcon\DI

implements Phalcon\DiInterface

Phalcon\DI is a component that implements Dependency Injection of services and it's itself a container for them. Since Phalcon is highly decoupled, Phalcon\DI is essential to integrate the different components of the framework. The developer can also use this component to inject dependencies and manage global instances of the different classes used in the application. Basically, this component implements the *Inversion of Control* pattern. Applying this, the objects do not receive their dependencies using setters or constructors, but requesting a service dependency injector. This reduces the overall complexity, since there is only one way to get the required dependencies within a component. Additionally, this pattern increases testability in the code, thus making it less prone to errors.

```
<?php

 $di = new Phalcon\DI();

 //Using a string definition
 $di->set('request', 'Phalcon\Http\Request', true);

 //Using an anonymous function
 $di->set('request', function(){
  return new Phalcon\Http\Request();
 }, true);

 $request = $di->getRequest();
```

Methods

public **__construct** ()

Phalcon\DI constructor

public *Phalcon\DI\ServiceInterface* **set** (*string* $name, *mixed* $definition, [*boolean* $shared])

Registers a service in the services container

public *Phalcon\DI\ServiceInterface* **setShared** (*string* $name, *mixed* $definition)

Registers an "always shared" service in the services container

public **remove** (*string* $name)

Removes a service in the services container

public *Phalcon\DI\ServiceInterface* **attempt** (*string* $name, *mixed* $definition, [*boolean* $shared])

Attempts to register a service in the services container Only is successful if a service hasn't been registered previously with the same name

public *Phalcon\DI\ServiceInterface* **setRaw** (*string* $name, *Phalcon\DI\ServiceInterface* $rawDefinition)

Sets a service using a raw Phalcon\DI\Service definition

public *mixed* **getRaw** (*string* $name)

Returns a service definition without resolving

public *Phalcon\DI\ServiceInterface* **getService** (*string* $name)

Returns a Phalcon\DI\Service instance

public *mixed* **get** (*string* $name, [*array* $parameters])

Resolves the service based on its configuration

public *mixed* **getShared** (*string* $name, [*array* $parameters])

Resolves a service, the resolved service is stored in the DI, subsequent requests for this service will return the same instance

public *boolean* **has** (*string* $name)

Check whether the DI contains a service by a name

public *boolean* **wasFreshInstance** ()

Check whether the last service obtained via getShared produced a fresh instance or an existing one

public *Phalcon\DI\Service* [] **getServices** ()

Return the services registered in the DI

public *boolean* **offsetExists** (*string* $alias)

Check if a service is registered using the array syntax

public **offsetSet** (*string* $alias, *mixed* $definition)

Allows to register a shared service using the array syntax

```
<?php

$di['request'] = new Phalcon\Http\Request();
```

public *mixed* **offsetGet** (*string* $alias)

Allows to obtain a shared service using the array syntax

```
<?php

var_dump($di['request']);
```

public **offsetUnset** (*string* $alias)

Removes a service from the services container using the array syntax

public *mixed* **__call** (*string* $method, [*array* $arguments])

Magic method to get or set services using setters/getters

public static **setDefault** (*Phalcon\DiInterface* $dependencyInjector)

Set a default dependency injection container to be obtained into static methods

public static *Phalcon\DiInterface* **getDefault** ()

Return the lastest DI created

public static **reset** ()

Resets the internal default DI

2.48.39 Class Phalcon\DI\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.40 Class Phalcon\DI\FactoryDefault

extends Phalcon\DI

implements Phalcon\DiInterface

This is a variant of the standard Phalcon\DI. By default it automatically registers all the services provided by the framework. Thanks to this, the developer does not need to register each service individually providing a full stack framework

Methods

public **__construct** ()

Phalcon\DI\FactoryDefault constructor

public *Phalcon\DI\ServiceInterface* **set** (*string* $name, *mixed* $definition, [*boolean* $shared]) inherited from Phalcon\DI

Registers a service in the services container

public *Phalcon\DI\ServiceInterface* **setShared** (*string* $name, *mixed* $definition) inherited from Phalcon\DI

Registers an “always shared” service in the services container

public **remove** (*string* $name) inherited from Phalcon\DI

Removes a service in the services container

public *Phalcon\DI\ServiceInterface* **attempt** (*string* $name, *mixed* $definition, [*boolean* $shared]) inherited from Phalcon\DI

Attempts to register a service in the services container Only is successful if a service hasn't been registered previously with the same name

public *Phalcon\DI\ServiceInterface* **setRaw** (*string* $name, *Phalcon\DI\ServiceInterface* $rawDefinition) inherited from Phalcon\DI

Sets a service using a raw Phalcon\DI\Service definition

public *mixed* **getRaw** (*string* $name) inherited from Phalcon\DI

Returns a service definition without resolving

public *Phalcon\DI\ServiceInterface* **getService** (*string* $name) inherited from Phalcon\DI

Returns a Phalcon\DI\Service instance

public *mixed* **get** (*string* $name, [*array* $parameters]) inherited from Phalcon\DI

Resolves the service based on its configuration

public *mixed* **getShared** (*string* $name, [*array* $parameters]) inherited from Phalcon\DI

Resolves a service, the resolved service is stored in the DI, subsequent requests for this service will return the same instance

public *boolean* **has** (*string* $name) inherited from Phalcon\DI

Check whether the DI contains a service by a name

public *boolean* **wasFreshInstance** () inherited from Phalcon\DI

Check whether the last service obtained via getShared produced a fresh instance or an existing one

public *Phalcon\DI\Service* [] **getServices** () inherited from Phalcon\DI

Return the services registered in the DI

public *boolean* **offsetExists** (*string* $alias) inherited from Phalcon\DI

Check if a service is registered using the array syntax

public **offsetSet** (*string* $alias, *mixed* $definition) inherited from Phalcon\DI

Allows to register a shared service using the array syntax

```
<?php

$di['request'] = new Phalcon\Http\Request();
```

public *mixed* **offsetGet** (*string* $alias) inherited from Phalcon\DI

Allows to obtain a shared service using the array syntax

```
<?php

var_dump($di['request']);
```

public **offsetUnset** (*string* $alias) inherited from Phalcon\DI

Removes a service from the services container using the array syntax

public *mixed* **__call** (*string* $method, [*array* $arguments]) inherited from Phalcon\DI

Magic method to get or set services using setters/getters

public static **setDefault** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI

Set a default dependency injection container to be obtained into static methods

public static *Phalcon\DiInterface* **getDefault** () inherited from Phalcon\DI

Return the lastest DI created

public static **reset** () inherited from Phalcon\DI

Resets the internal default DI

2.48.41 Class Phalcon\DI\FactoryDefault\CLI

extends Phalcon\DI\FactoryDefault

implements Phalcon\DiInterface

This is a variant of the standard Phalcon\DI. By default it automatically registers all the services provided by the framework. Thanks to this, the developer does not need to register each service individually. This class is specially suitable for CLI applications

Methods

public **__construct** ()

Phalcon\DI\FactoryDefault\CLI constructor

public *Phalcon\DI\ServiceInterface* **set** (*string* $name, *mixed* $definition, [*boolean* $shared]) inherited from Phalcon\DI

Registers a service in the services container

public *Phalcon\DI\ServiceInterface* **setShared** (*string* $name, *mixed* $definition) inherited from Phalcon\DI

Registers an “always shared” service in the services container

public **remove** (*string* $name) inherited from Phalcon\DI

Removes a service in the services container

public *Phalcon\DI\ServiceInterface* **attempt** (*string* $name, *mixed* $definition, [*boolean* $shared]) inherited from Phalcon\DI

Attempts to register a service in the services container Only is successful if a service hasn’t been registered previously with the same name

public *Phalcon\DI\ServiceInterface* **setRaw** (*string* $name, *Phalcon\DI\ServiceInterface* $rawDefinition) inherited from Phalcon\DI

Sets a service using a raw Phalcon\DI\Service definition

public *mixed* **getRaw** (*string* $name) inherited from Phalcon\DI

Returns a service definition without resolving

public *Phalcon\DI\ServiceInterface* **getService** (*string* $name) inherited from Phalcon\DI

Returns a Phalcon\DI\Service instance

public *mixed* **get** (*string* $name, [*array* $parameters]) inherited from Phalcon\DI

Resolves the service based on its configuration

public *mixed* **getShared** (*string* $name, [*array* $parameters]) inherited from Phalcon\DI

Resolves a service, the resolved service is stored in the DI, subsequent requests for this service will return the same instance

public *boolean* **has** (*string* $name) inherited from Phalcon\DI

Check whether the DI contains a service by a name

public *boolean* **wasFreshInstance** () inherited from Phalcon\DI

Check whether the last service obtained via getShared produced a fresh instance or an existing one

public *Phalcon\DI\Service* [] **getServices** () inherited from Phalcon\DI

Return the services registered in the DI

public *boolean* **offsetExists** (*string* $alias) inherited from Phalcon\DI

Check if a service is registered using the array syntax

public **offsetSet** (*string* $alias, *mixed* $definition) inherited from Phalcon\DI

Allows to register a shared service using the array syntax

```
<?php

$di['request'] = new Phalcon\Http\Request();
```

public *mixed* **offsetGet** (*string* $alias) inherited from Phalcon\DI

Allows to obtain a shared service using the array syntax

```
<?php

var_dump($di['request']);
```

public **offsetUnset** (*string* $alias) inherited from Phalcon\DI

Removes a service from the services container using the array syntax

public *mixed* **__call** (*string* $method, [*array* $arguments]) inherited from Phalcon\DI

Magic method to get or set services using setters/getters

public static **setDefault** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI

Set a default dependency injection container to be obtained into static methods

public static *Phalcon\DiInterface* **getDefault** () inherited from Phalcon\DI

Return the lastest DI created

public static **reset** () inherited from Phalcon\DI

Resets the internal default DI

2.48.42 Class Phalcon\DI\Injectable

implements Phalcon\DI\InjectionAwareInterface, *Phalcon\Events\EventsAwareInterface*

This class allows to access services in the services container by just only accessing a public property with the same name of a registered service

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager)

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the internal event manager

public **__get** (*string* $propertyName)

Magic method __get

2.48.43 Class Phalcon\DI\Service

implements Phalcon\DI\ServiceInterface

Represents individually a service in the services container

```
<?php

 $service = new Phalcon\DI\Service('request', 'Phalcon\Http\Request');
 $request = $service->resolve();

<?php
```

Methods

public **__construct** (*string* $name, *mixed* $definition, [*boolean* $shared])

public **getName** ()

Returns the service's name

public **setShared** (*boolean* $shared)

Sets if the service is shared or not

public *boolean* **isShared** ()

Check whether the service is shared or not

public **setSharedInstance** (*mixed* $sharedInstance)

Sets/Resets the shared instance related to the service

public **setDefinition** (*mixed* $definition)

Set the service definition

public *mixed* **getDefinition** ()

Returns the service definition

public *mixed* **resolve** ([*array* $parameters], [*Phalcon\DiInterface* $dependencyInjector])

Resolves the service

public *Phalcon\DI\Service* **setParameter** (*long* $position, *array* $parameter)

Changes a parameter in the definition without resolve the service

public *array* **getParameter** (*int* $position)

Returns a parameter in an specific position

public static *Phalcon\DI\Service* **__set_state** (*array* $attributes)

Restore the internal state of a service

2.48.44 Class Phalcon\DI\Service\Builder

This class builds instances based on complex definitions

Methods

protected *mixed* **_buildParameter** ()

Resolves a constructor/call parameter

protected *arguments* **_buildParameters** ()

Resolves an array of parameters

public *mixed* **build** (*Phalcon\DiInterface* $dependencyInjector, *array* $definition, [*array* $parameters])

Builds a service using a complex service definition

2.48.45 Class Phalcon\Db

Phalcon\Db and its related classes provide a simple SQL database interface for Phalcon Framework. The Phalcon\Db is the basic class you use to connect your PHP application to an RDBMS. There is a different adapter class for each brand of RDBMS. This component is intended to lower level database operations. If you want to interact with databases using higher level of abstraction use Phalcon\Mvc\Model. Phalcon\Db is an abstract class. You only can use it with a database adapter like Phalcon\Db\Adapter\Pdo

```
<?php

try {

  $connection = new Phalcon\Db\Adapter\Pdo\Mysql(array(
     'host' => '192.168.0.11',
     'username' => 'sigma',
     'password' => 'secret',
     'dbname' => 'blog',
     'port' => '3306',
  ));
```

```
  $result = $connection->query("SELECT * FROM robots LIMIT 5");
  $result->setFetchMode(Phalcon\Db::FETCH_NUM);
  while($robot = $result->fetch()){
    print_r($robot);
  }

} catch(Phalcon\Db\Exception $e){
echo $e->getMessage(), PHP_EOL;
}
```

Constants

integer **FETCH_ASSOC**

integer **FETCH_BOTH**

integer **FETCH_NUM**

integer **FETCH_OBJ**

Methods

public static **setup** (*array* $options)

Enables/disables options in the Database component

2.48.46 Class Phalcon\Db\Adapter

implements Phalcon\Events\EventsAwareInterface

Base class for Phalcon\Db adapters

Methods

protected **__construct** ()

Phalcon\Db\Adapter constructor

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager)

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the internal event manager

public *array* **fetchOne** (*string* $sqlQuery, [*int* $fetchMode], [*array* $bindParams], [*array* $bindTypes])

Returns the first row in a SQL query result

```
<?php

//Getting first robot
$robot = $connection->fecthOne("SELECT * FROM robots");
print_r($robot);

//Getting first robot with associative indexes only
```

```
$robot = $connection->fecthOne("SELECT * FROM robots", Phalcon\Db::FETCH_ASSOC);
print_r($robot);
```

public *array* **fetchAll** (*string* $sqlQuery, [*int* $fetchMode], [*array* $bindParams], [*array* $bindTypes])

Dumps the complete result of a query into an array

```
<?php

//Getting all robots
$robots = $connection->fetchAll("SELECT * FROM robots");
foreach($robots as $robot){
    print_r($robot);
}

//Getting all robots with associative indexes only
$robots = $connection->fetchAll("SELECT * FROM robots", Phalcon\Db::FETCH_ASSOC);
foreach($robots as $robot){
    print_r($robot);
}
```

public *boolean* **insert** (*string* $table, *array* $values, [*array* $fields], [*array* $dataTypes])

Inserts data into a table using custom RBDM SQL syntax

```
<?php

 //Inserting a new robot
 $success = $connection->insert(
     "robots",
     array("Astro Boy", 1952),
     array("name", "year")
 );

 //Next SQL sentence is sent to the database system
 INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **update** (*string* $table, *array* $fields, *array* $values, [*string* $whereCondition], [*array* $dataTypes])

Updates data on a table using custom RBDM SQL syntax

```
<?php

 //Updating existing robot
 $success = $connection->update(
     "robots",
     array("name")
     array("New Astro Boy"),
     "id = 101"
 );

 //Next SQL sentence is sent to the database system
 UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101
```

public *boolean* **delete** (*string* $table, [*string* $whereCondition], [*array* $placeholders], [*array* $dataTypes])

Deletes data from a table using custom RBDM SQL syntax

```
<?php

 //Deleting existing robot
```

```
$success = $connection->delete(
    "robots",
    "id = 101"
);

//Next SQL sentence is generated
DELETE FROM `robots` WHERE `id` = 101
```

public *string* **getColumnList** (*array* $columnList)

Gets a list of columns

public *string* **limit** (*string* $sqlQuery, *int* $number)

Appends a LIMIT clause to $sqlQuery argument

```
<?php

    echo $connection->limit("SELECT * FROM robots", 5);
```

public *string* **tableExists** (*string* $tableName, [*string* $schemaName])

Generates SQL checking for the existence of a schema.table

```
<?php

    var_dump($connection->tableExists("blog", "posts"));
```

public *string* **viewExists** (*string* $viewName, [*string* $schemaName])

Generates SQL checking for the existence of a schema.view

```
<?php

 var_dump($connection->viewExists("active_users", "posts"));
```

public *string* **forUpdate** (*string* $sqlQuery)

Returns a SQL modified with a FOR UPDATE clause

public *string* **sharedLock** (*string* $sqlQuery)

Returns a SQL modified with a LOCK IN SHARE MODE clause

public *boolean* **createTable** (*string* $tableName, *string* $schemaName, *array* $definition)

Creates a table

public *boolean* **dropTable** (*string* $tableName, *string* $schemaName, [*boolean* $ifExists])

Drops a table from a schema/database

public *boolean* **addColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Adds a column to a table

public *boolean* **modifyColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Modifies a table column based on a definition

public *boolean* **dropColumn** (*string* $tableName, *string* $schemaName, *string* $columnName)

Drops a column from a table

public *boolean* **addIndex** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index)

Adds an index to a table

public *boolean* **dropIndex** (*string* $tableName, *string* $schemaName, *string* $indexName)

Drop an index from a table

public *boolean* **addPrimaryKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index)

Adds a primary key to a table

public *boolean* **dropPrimaryKey** (*string* $tableName, *string* $schemaName)

Drops a table's primary key

public *boolean true* **addForeignKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ReferenceInterface* $reference)

Adds a foreign key to a table

public *boolean true* **dropForeignKey** (*string* $tableName, *string* $schemaName, *string* $referenceName)

Drops a foreign key from a table

public *string* **getColumnDefinition** (*Phalcon\Db\ColumnInterface* $column)

Returns the SQL column definition from a column

public *array* **listTables** ([*string* $schemaName])

List all tables on a database

```
<?php

    print_r($connection->listTables("blog");
```

public *array* **getDescriptor** ()

Return descriptor used to connect to the active database

public *string* **getConnectionId** ()

Gets the active connection unique identifier

public *string* **getSQLStatement** ()

Active SQL statement in the object

public *string* **getRealSQLStatement** ()

Active SQL statement in the object without replace bound paramters

public *array* **getSQLVariables** ()

Active SQL statement in the object

public *array* **getSQLBindTypes** ()

Active SQL statement in the object

public *string* **getType** ()

Returns type of database system the adapter is used for

public *string* **getDialectType** ()

Returns the name of the dialect used

public *Phalcon\Db\DialectInterface* **getDialect** ()

Returns internal dialect instance

2.48.47 Class Phalcon\Db\Adapter\Pdo

extends Phalcon\Db\Adapter

implements Phalcon\Events\EventsAwareInterface

Phalcon\Db\Adapter\Pdo is the Phalcon\Db that internally uses PDO to connect to a database

```
<?php

 $connection = new Phalcon\Db\Adapter\Pdo\Mysql(array(
  'host' => '192.168.0.11',
  'username' => 'sigma',
  'password' => 'secret',
  'dbname' => 'blog',
  'port' => '3306',
 ));
```

Methods

public **__construct** (*array* $descriptor)

Constructor for Phalcon\Db\Adapter\Pdo

public *boolean* **connect** ([*array* $descriptor])

This method is automatically called in Phalcon\Db\Adapter\Pdo constructor. Call it when you need to restore a database connection

```
<?php

 //Make a connection
 $connection = new Phalcon\Db\Adapter\Pdo\Mysql(array(
  'host' => '192.168.0.11',
  'username' => 'sigma',
  'password' => 'secret',
  'dbname' => 'blog',
 ));

 //Reconnect
 $connection->connect();
```

public *PDOStatement* **prepare** (*string* $sqlStatement)

Returns a PDO prepared statement to be executed with 'executePrepared'

```
<?php

 $statement = $db->prepare('SELECT * FROM robots WHERE name = :name');
 $result = $connection->executePrepared($statement, array('name' => 'Voltron'));
```

public *PDOStatement* **executePrepared** (*PDOStatement* $statement, *array* $placeholders, *array* $dataTypes)

Executes a prepared statement binding. This function uses integer indexes starting from zero

```
<?php

 $statement = $db->prepare('SELECT * FROM robots WHERE name = :name');
 $result = $connection->executePrepared($statement, array('name' => 'Voltron'));
```

public *Phalcon\Db\ResultInterface* **query** (*string* $sqlStatement, [*array* $bindParams], [*array* $bindTypes])

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server is returning rows

```
<?php

//Querying data
$resultset = $connection->query("SELECT * FROM robots WHERE type='mechanical'");
$resultset = $connection->query("SELECT * FROM robots WHERE type=?", array("mechanical"));
```

public *boolean* **execute** (*string* $sqlStatement, [*array* $bindParams], [*array* $bindTypes])

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server doesn't return any row

```
<?php

//Inserting data
$success = $connection->execute("INSERT INTO robots VALUES (1, 'Astro Boy')");
$success = $connection->execute("INSERT INTO robots VALUES (?, ?)", array(1, 'Astro Boy'));
```

public *int* **affectedRows** ()

Returns the number of affected rows by the lastest INSERT/UPDATE/DELETE executed in the database system

```
<?php

$connection->execute("DELETE FROM robots");
echo $connection->affectedRows(), ' were deleted';
```

public *boolean* **close** ()

Closes the active connection returning success. Phalcon automatically closes and destroys active connections when the request ends

public *string* **escapeIdentifier** (*string* $identifier)

Escapes a column/table/schema name

```
<?php

$escapedTable = $connection->escapeIdentifier('robots');
```

public *string* **escapeString** (*string* $str)

Escapes a value to avoid SQL injections

```
<?php

$escapedStr = $connection->escapeString('some dangerous value');
```

public *string* **bindParams** (*string* $sqlStatement, *array* $params)

Manually bind params to a SQL statement. This method requires an active connection to a database system

```
<?php

$sql = $connection->bindParams('SELECT * FROM robots WHERE name = ?0', array('Bender'));
  echo $sql; // SELECT * FROM robots WHERE name = 'Bender'
```

public *array* **convertBoundParams** (*string* $sql, *array* $params)

Converts bound parameters such as :name: or ?1 into PDO bind params ?

```
<?php

 print_r($connection->convertBoundParams('SELECT * FROM robots WHERE name = :name:', array('Bender'))
```

public *int* **lastInsertId** ([*string* $sequenceName])

Returns the insert id for the auto_increment/serial column inserted in the lastest executed SQL statement

```
<?php

 //Inserting a new robot
 $success = $connection->insert(
     "robots",
     array("Astro Boy", 1952),
     array("name", "year")
 );

 //Getting the generated id
 $id = $connection->lastInsertId();
```

public *boolean* **begin** ()

Starts a transaction in the connection

public *boolean* **rollback** ()

Rollbacks the active transaction in the connection

public *boolean* **commit** ()

Commits the active transaction in the connection

public *boolean* **isUnderTransaction** ()

Checks whether the connection is under a transaction

```
<?php

 $connection->begin();
 var_dump($connection->isUnderTransaction()); //true
```

public *PDO* **getInternalHandler** ()

Return internal PDO handler

public *Phalcon\Db\Index* [] **describeIndexes** (*string* $table, [*string* $schema])

Lists table indexes

```
<?php

 print_r($connection->describeIndexes('robots_parts'));
```

public *Phalcon\Db\Reference* [] **describeReferences** (*string* $table, [*string* $schema])

Lists table references

```
<?php

 print_r($connection->describeReferences('robots_parts'));
```

public *array* **tableOptions** (*string* $tableName, [*string* $schemaName])

Gets creation options from a table

```
<?php

 print_r($connection->tableOptions('robots'));
```

public *Phalcon\Db\RawValue* **getDefaultIdValue** ()

Returns the default identity value to be inserted in an identity column

```
<?php

 //Inserting a new robot with a valid default value for the column 'id'
 $success = $connection->insert(
     "robots",
     array($connection->getDefaultIdValue(), "Astro Boy", 1952),
     array("id", "name", "year")
 );
```

public *boolean* **supportSequences** ()

Check whether the database system requires a sequence to produce auto-numeric values

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\Db\Adapter

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\Db\Adapter

Returns the internal event manager

public *array* **fetchOne** (*string* $sqlQuery, [*int* $fetchMode], [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter

Returns the first row in a SQL query result

```
<?php

//Getting first robot
$robot = $connection->fecthOne("SELECT * FROM robots");
print_r($robot);

//Getting first robot with associative indexes only
$robot = $connection->fecthOne("SELECT * FROM robots", Phalcon\Db::FETCH_ASSOC);
print_r($robot);
```

public *array* **fetchAll** (*string* $sqlQuery, [*int* $fetchMode], [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter

Dumps the complete result of a query into an array

```
<?php

//Getting all robots
$robots = $connection->fetchAll("SELECT * FROM robots");
foreach($robots as $robot){
    print_r($robot);
}

//Getting all robots with associative indexes only
$robots = $connection->fetchAll("SELECT * FROM robots", Phalcon\Db::FETCH_ASSOC);
foreach($robots as $robot){
    print_r($robot);
}
```

public *boolean* **insert** (*string* $table, *array* $values, [*array* $fields], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Inserts data into a table using custom RBDM SQL syntax

```
<?php

 //Inserting a new robot
 $success = $connection->insert(
     "robots",
     array("Astro Boy", 1952),
     array("name", "year")
 );

 //Next SQL sentence is sent to the database system
 INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **update** (*string* $table, *array* $fields, *array* $values, [*string* $whereCondition], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Updates data on a table using custom RBDM SQL syntax

```
<?php

 //Updating existing robot
 $success = $connection->update(
     "robots",
     array("name")
     array("New Astro Boy"),
     "id = 101"
 );

 //Next SQL sentence is sent to the database system
 UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101
```

public *boolean* **delete** (*string* $table, [*string* $whereCondition], [*array* $placeholders], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Deletes data from a table using custom RBDM SQL syntax

```
<?php

 //Deleting existing robot
 $success = $connection->delete(
     "robots",
     "id = 101"
 );

 //Next SQL sentence is generated
 DELETE FROM `robots` WHERE `id` = 101
```

public *string* **getColumnList** (*array* $columnList) inherited from Phalcon\Db\Adapter

Gets a list of columns

public *string* **limit** (*string* $sqlQuery, *int* $number) inherited from Phalcon\Db\Adapter

Appends a LIMIT clause to $sqlQuery argument

```
<?php

    echo $connection->limit("SELECT * FROM robots", 5);
```

public *string* **tableExists** (*string* $tableName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter

Generates SQL checking for the existence of a schema.table

```
<?php

    var_dump($connection->tableExists("blog", "posts"));
```

public *string* **viewExists** (*string* $viewName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter

Generates SQL checking for the existence of a schema.view

```
<?php

 var_dump($connection->viewExists("active_users", "posts"));
```

public *string* **forUpdate** (*string* $sqlQuery) inherited from Phalcon\Db\Adapter

Returns a SQL modified with a FOR UPDATE clause

public *string* **sharedLock** (*string* $sqlQuery) inherited from Phalcon\Db\Adapter

Returns a SQL modified with a LOCK IN SHARE MODE clause

public *boolean* **createTable** (*string* $tableName, *string* $schemaName, *array* $definition) inherited from Phalcon\Db\Adapter

Creates a table

public *boolean* **dropTable** (*string* $tableName, *string* $schemaName, [*boolean* $ifExists]) inherited from Phalcon\Db\Adapter

Drops a table from a schema/database

public *boolean* **addColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Adds a column to a table

public *boolean* **modifyColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Modifies a table column based on a definition

public *boolean* **dropColumn** (*string* $tableName, *string* $schemaName, *string* $columnName) inherited from Phalcon\Db\Adapter

Drops a column from a table

public *boolean* **addIndex** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index) inherited from Phalcon\Db\Adapter

Adds an index to a table

public *boolean* **dropIndex** (*string* $tableName, *string* $schemaName, *string* $indexName) inherited from Phalcon\Db\Adapter

Drop an index from a table

public *boolean* **addPrimaryKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index) inherited from Phalcon\Db\Adapter

Adds a primary key to a table

public *boolean* **dropPrimaryKey** (*string* $tableName, *string* $schemaName) inherited from Phalcon\Db\Adapter

Drops a table's primary key

public *boolean true* **addForeignKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ReferenceInterface* $reference) inherited from Phalcon\Db\Adapter

Adds a foreign key to a table

public *boolean true* **dropForeignKey** (*string* $tableName, *string* $schemaName, *string* $referenceName) inherited from Phalcon\Db\Adapter

Drops a foreign key from a table

public *string* **getColumnDefinition** (*Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Returns the SQL column definition from a column

public *array* **listTables** ([*string* $schemaName]) inherited from Phalcon\Db\Adapter

List all tables on a database

```
<?php

    print_r($connection->listTables("blog");
```

public *array* **getDescriptor** () inherited from Phalcon\Db\Adapter

Return descriptor used to connect to the active database

public *string* **getConnectionId** () inherited from Phalcon\Db\Adapter

Gets the active connection unique identifier

public *string* **getSQLStatement** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *string* **getRealSQLStatement** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object without replace bound paramters

public *array* **getSQLVariables** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *array* **getSQLBindTypes** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *string* **getType** () inherited from Phalcon\Db\Adapter

Returns type of database system the adapter is used for

public *string* **getDialectType** () inherited from Phalcon\Db\Adapter

Returns the name of the dialect used

public *Phalcon\Db\DialectInterface* **getDialect** () inherited from Phalcon\Db\Adapter

Returns internal dialect instance

2.48.48 Class Phalcon\Db\Adapter\Pdo\Mysql

extends Phalcon\Db\Adapter\Pdo

implements Phalcon\Events\EventsAwareInterface, *Phalcon\Db\AdapterInterface*

Specific functions for the Mysql database system

```php
<?php

$config = array(
    "host" => "192.168.0.11",
    "dbname" => "blog",
    "port" => 3306,
    "username" => "sigma",
    "password" => "secret"
);

$connection = new Phalcon\Db\Adapter\Pdo\Mysql($config);
```

Methods

public *string* **escapeIdentifier** (*string* $identifier)

Escapes a column/table/schema name

public *Phalcon\Db\Column* [] **describeColumns** (*string* $table, [*string* $schema])

Returns an array of Phalcon\Db\Column objects describing a table

```php
<?php

 print_r($connection->describeColumns("posts")); ?>
```

public **__construct** (*array* $descriptor) inherited from Phalcon\Db\Adapter\Pdo

Constructor for Phalcon\Db\Adapter\Pdo

public *boolean* **connect** ([*array* $descriptor]) inherited from Phalcon\Db\Adapter\Pdo

This method is automatically called in Phalcon\Db\Adapter\Pdo constructor. Call it when you need to restore a database connection

```php
<?php

 //Make a connection
 $connection = new Phalcon\Db\Adapter\Pdo\Mysql(array(
  'host' => '192.168.0.11',
  'username' => 'sigma',
  'password' => 'secret',
  'dbname' => 'blog',
 ));

 //Reconnect
 $connection->connect();
```

public *PDOStatement* **prepare** (*string* $sqlStatement) inherited from Phalcon\Db\Adapter\Pdo

Returns a PDO prepared statement to be executed with 'executePrepared'

```php
<?php

 $statement = $db->prepare('SELECT * FROM robots WHERE name = :name');
 $result = $connection->executePrepared($statement, array('name' => 'Voltron'));
```

public *PDOStatement* **executePrepared** (*PDOStatement* $statement, *array* $placeholders, *array* $dataTypes) inherited from Phalcon\Db\Adapter\Pdo

Executes a prepared statement binding. This function uses integer indexes starting from zero

```
<?php

 $statement = $db->prepare('SELECT * FROM robots WHERE name = :name');
 $result = $connection->executePrepared($statement, array('name' => 'Voltron'));
```

public *Phalcon\Db\ResultInterface* **query** (*string* $sqlStatement, [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter\Pdo

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server is returning rows

```
<?php

//Querying data
$resultset = $connection->query("SELECT * FROM robots WHERE type='mechanical'");
$resultset = $connection->query("SELECT * FROM robots WHERE type=?", array("mechanical"));
```

public *boolean* **execute** (*string* $sqlStatement, [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter\Pdo

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server doesn't return any row

```
<?php

//Inserting data
$success = $connection->execute("INSERT INTO robots VALUES (1, 'Astro Boy')");
$success = $connection->execute("INSERT INTO robots VALUES (?, ?)", array(1, 'Astro Boy'));
```

public *int* **affectedRows** () inherited from Phalcon\Db\Adapter\Pdo

Returns the number of affected rows by the lastest INSERT/UPDATE/DELETE executed in the database system

```
<?php

$connection->execute("DELETE FROM robots");
echo $connection->affectedRows(), ' were deleted';
```

public *boolean* **close** () inherited from Phalcon\Db\Adapter\Pdo

Closes the active connection returning success. Phalcon automatically closes and destroys active connections when the request ends

public *string* **escapeString** (*string* $str) inherited from Phalcon\Db\Adapter\Pdo

Escapes a value to avoid SQL injections

```
<?php

$escapedStr = $connection->escapeString('some dangerous value');
```

public *string* **bindParams** (*string* $sqlStatement, *array* $params) inherited from Phalcon\Db\Adapter\Pdo

Manually bind params to a SQL statement. This method requires an active connection to a database system

```
<?php

$sql = $connection->bindParams('SELECT * FROM robots WHERE name = ?0', array('Bender'));
  echo $sql; // SELECT * FROM robots WHERE name = 'Bender'
```

public *array* **convertBoundParams** (*string* $sql, *array* $params) inherited from Phalcon\Db\Adapter\Pdo

Converts bound parameters such as :name: or ?1 into PDO bind params ?

```
<?php

 print_r($connection->convertBoundParams('SELECT * FROM robots WHERE name = :name:', array('Bender'))
```

public *int* **lastInsertId** ([*string* $sequenceName]) inherited from Phalcon\Db\Adapter\Pdo

Returns the insert id for the auto_increment/serial column inserted in the lastest executed SQL statement

```
<?php

 //Inserting a new robot
 $success = $connection->insert(
     "robots",
     array("Astro Boy", 1952),
     array("name", "year")
 );

 //Getting the generated id
 $id = $connection->lastInsertId();
```

public *boolean* **begin** () inherited from Phalcon\Db\Adapter\Pdo

Starts a transaction in the connection

public *boolean* **rollback** () inherited from Phalcon\Db\Adapter\Pdo

Rollbacks the active transaction in the connection

public *boolean* **commit** () inherited from Phalcon\Db\Adapter\Pdo

Commits the active transaction in the connection

public *boolean* **isUnderTransaction** () inherited from Phalcon\Db\Adapter\Pdo

Checks whether the connection is under a transaction

```
<?php

 $connection->begin();
 var_dump($connection->isUnderTransaction()); //true
```

public *PDO* **getInternalHandler** () inherited from Phalcon\Db\Adapter\Pdo

Return internal PDO handler

public *Phalcon\Db\Index* [] **describeIndexes** (*string* $table, [*string* $schema]) inherited from Phalcon\Db\Adapter\Pdo

Lists table indexes

```
<?php

 print_r($connection->describeIndexes('robots_parts'));
```

public *Phalcon\Db\Reference* [] **describeReferences** (*string* $table, [*string* $schema]) inherited from Phalcon\Db\Adapter\Pdo

Lists table references

```
<?php

 print_r($connection->describeReferences('robots_parts'));
```

public *array* **tableOptions** (*string* $tableName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter\Pdo

Gets creation options from a table

```
<?php

 print_r($connection->tableOptions('robots'));
```

public *Phalcon\Db\RawValue* **getDefaultIdValue** () inherited from Phalcon\Db\Adapter\Pdo

Returns the default identity value to be inserted in an identity column

```
<?php

 //Inserting a new robot with a valid default value for the column 'id'
 $success = $connection->insert(
     "robots",
     array($connection->getDefaultIdValue(), "Astro Boy", 1952),
     array("id", "name", "year")
 );
```

public *boolean* **supportSequences** () inherited from Phalcon\Db\Adapter\Pdo

Check whether the database system requires a sequence to produce auto-numeric values

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\Db\Adapter

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\Db\Adapter

Returns the internal event manager

public *array* **fetchOne** (*string* $sqlQuery, [*int* $fetchMode], [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter

Returns the first row in a SQL query result

```
<?php

//Getting first robot
$robot = $connection->fecthOne("SELECT * FROM robots");
print_r($robot);

//Getting first robot with associative indexes only
$robot = $connection->fecthOne("SELECT * FROM robots", Phalcon\Db::FETCH_ASSOC);
print_r($robot);
```

public *array* **fetchAll** (*string* $sqlQuery, [*int* $fetchMode], [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter

Dumps the complete result of a query into an array

```
<?php

//Getting all robots
$robots = $connection->fetchAll("SELECT * FROM robots");
foreach($robots as $robot){
    print_r($robot);
}

//Getting all robots with associative indexes only
$robots = $connection->fetchAll("SELECT * FROM robots", Phalcon\Db::FETCH_ASSOC);
```

```
foreach($robots as $robot){
    print_r($robot);
}
```

public *boolean* **insert** (*string* $table, *array* $values, [*array* $fields], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Inserts data into a table using custom RBDM SQL syntax

```
<?php

 //Inserting a new robot
 $success = $connection->insert(
     "robots",
     array("Astro Boy", 1952),
     array("name", "year")
 );

 //Next SQL sentence is sent to the database system
 INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **update** (*string* $table, *array* $fields, *array* $values, [*string* $whereCondition], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Updates data on a table using custom RBDM SQL syntax

```
<?php

 //Updating existing robot
 $success = $connection->update(
     "robots",
     array("name")
     array("New Astro Boy"),
     "id = 101"
 );

 //Next SQL sentence is sent to the database system
 UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101
```

public *boolean* **delete** (*string* $table, [*string* $whereCondition], [*array* $placeholders], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Deletes data from a table using custom RBDM SQL syntax

```
<?php

 //Deleting existing robot
 $success = $connection->delete(
     "robots",
     "id = 101"
 );

 //Next SQL sentence is generated
 DELETE FROM `robots` WHERE `id` = 101
```

public *string* **getColumnList** (*array* $columnList) inherited from Phalcon\Db\Adapter

Gets a list of columns

public *string* **limit** (*string* $sqlQuery, *int* $number) inherited from Phalcon\Db\Adapter

Appends a LIMIT clause to $sqlQuery argument

```
<?php

    echo $connection->limit("SELECT * FROM robots", 5);
```

public *string* **tableExists** (*string* $tableName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter

Generates SQL checking for the existence of a schema.table

```
<?php

    var_dump($connection->tableExists("blog", "posts"));
```

public *string* **viewExists** (*string* $viewName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter

Generates SQL checking for the existence of a schema.view

```
<?php

 var_dump($connection->viewExists("active_users", "posts"));
```

public *string* **forUpdate** (*string* $sqlQuery) inherited from Phalcon\Db\Adapter

Returns a SQL modified with a FOR UPDATE clause

public *string* **sharedLock** (*string* $sqlQuery) inherited from Phalcon\Db\Adapter

Returns a SQL modified with a LOCK IN SHARE MODE clause

public *boolean* **createTable** (*string* $tableName, *string* $schemaName, *array* $definition) inherited from Phalcon\Db\Adapter

Creates a table

public *boolean* **dropTable** (*string* $tableName, *string* $schemaName, [*boolean* $ifExists]) inherited from Phalcon\Db\Adapter

Drops a table from a schema/database

public *boolean* **addColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Adds a column to a table

public *boolean* **modifyColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Modifies a table column based on a definition

public *boolean* **dropColumn** (*string* $tableName, *string* $schemaName, *string* $columnName) inherited from Phalcon\Db\Adapter

Drops a column from a table

public *boolean* **addIndex** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index) inherited from Phalcon\Db\Adapter

Adds an index to a table

public *boolean* **dropIndex** (*string* $tableName, *string* $schemaName, *string* $indexName) inherited from Phalcon\Db\Adapter

Drop an index from a table

public *boolean* **addPrimaryKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index) inherited from Phalcon\Db\Adapter

Adds a primary key to a table

public *boolean* **dropPrimaryKey** (*string* $tableName, *string* $schemaName) inherited from Phalcon\Db\Adapter

Drops a table's primary key

public *boolean true* **addForeignKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ReferenceInterface* $reference) inherited from Phalcon\Db\Adapter

Adds a foreign key to a table

public *boolean true* **dropForeignKey** (*string* $tableName, *string* $schemaName, *string* $referenceName) inherited from Phalcon\Db\Adapter

Drops a foreign key from a table

public *string* **getColumnDefinition** (*Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Returns the SQL column definition from a column

public *array* **listTables** ([*string* $schemaName]) inherited from Phalcon\Db\Adapter

List all tables on a database

```
<?php

    print_r($connection->listTables("blog");
```

public *array* **getDescriptor** () inherited from Phalcon\Db\Adapter

Return descriptor used to connect to the active database

public *string* **getConnectionId** () inherited from Phalcon\Db\Adapter

Gets the active connection unique identifier

public *string* **getSQLStatement** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *string* **getRealSQLStatement** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object without replace bound paramters

public *array* **getSQLVariables** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *array* **getSQLBindTypes** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *string* **getType** () inherited from Phalcon\Db\Adapter

Returns type of database system the adapter is used for

public *string* **getDialectType** () inherited from Phalcon\Db\Adapter

Returns the name of the dialect used

public *Phalcon\Db\DialectInterface* **getDialect** () inherited from Phalcon\Db\Adapter

Returns internal dialect instance

2.48.49 Class Phalcon\Db\Adapter\Pdo\Postgresql

extends Phalcon\Db\Adapter\Pdo

implements Phalcon\Events\EventsAwareInterface, *Phalcon\Db\AdapterInterface*

Specific functions for the Postgresql database system

```
<?php

 $config = array(
  "host" => "192.168.0.11",
  "dbname" => "blog",
  "username" => "postgres",
  "password" => ""
 );

 $connection = new Phalcon\Db\Adapter\Pdo\Postgresql($config);
```

Methods

public *boolean* **connect** ([*array* $descriptor])

This method is automatically called in Phalcon\Db\Adapter\Pdo constructor. Call it when you need to restore a database connection. Support set search_path after connectted if schema is specified in config.

public *Phalcon\Db\Column* [] **describeColumns** (*string* $table, [*string* $schema])

Returns an array of Phalcon\Db\Column objects describing a table <code>print_r($connection->describeColumns("posts")); ?>

public *Phalcon\Db\RawValue* **getDefaultIdValue** ()

Return the default identity value to insert in an identity column

public *boolean* **supportSequences** ()

Check whether the database system requires a sequence to produce auto-numeric values

public **__construct** (*array* $descriptor) inherited from Phalcon\Db\Adapter\Pdo

Constructor for Phalcon\Db\Adapter\Pdo

public *PDOStatement* **prepare** (*string* $sqlStatement) inherited from Phalcon\Db\Adapter\Pdo

Returns a PDO prepared statement to be executed with 'executePrepared'

```
<?php

 $statement = $db->prepare('SELECT * FROM robots WHERE name = :name');
 $result = $connection->executePrepared($statement, array('name' => 'Voltron'));
```

public *PDOStatement* **executePrepared** (*PDOStatement* $statement, *array* $placeholders, *array* $dataTypes) inherited from Phalcon\Db\Adapter\Pdo

Executes a prepared statement binding. This function uses integer indexes starting from zero

```
<?php

 $statement = $db->prepare('SELECT * FROM robots WHERE name = :name');
 $result = $connection->executePrepared($statement, array('name' => 'Voltron'));
```

public *Phalcon\Db\ResultInterface* **query** (*string* $sqlStatement, [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter\Pdo

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server is returning rows

```
<?php

//Querying data
$resultset = $connection->query("SELECT * FROM robots WHERE type='mechanical'");
$resultset = $connection->query("SELECT * FROM robots WHERE type=?", array("mechanical"));
```

public *boolean* **execute** (*string* $sqlStatement, [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter\Pdo

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server doesn't return any row

```
<?php

//Inserting data
$success = $connection->execute("INSERT INTO robots VALUES (1, 'Astro Boy')");
$success = $connection->execute("INSERT INTO robots VALUES (?, ?)", array(1, 'Astro Boy'));
```

public *int* **affectedRows** () inherited from Phalcon\Db\Adapter\Pdo

Returns the number of affected rows by the lastest INSERT/UPDATE/DELETE executed in the database system

```
<?php

$connection->execute("DELETE FROM robots");
echo $connection->affectedRows(), ' were deleted';
```

public *boolean* **close** () inherited from Phalcon\Db\Adapter\Pdo

Closes the active connection returning success. Phalcon automatically closes and destroys active connections when the request ends

public *string* **escapeIdentifier** (*string* $identifier) inherited from Phalcon\Db\Adapter\Pdo

Escapes a column/table/schema name

```
<?php

$escapedTable = $connection->escapeIdentifier('robots');
```

public *string* **escapeString** (*string* $str) inherited from Phalcon\Db\Adapter\Pdo

Escapes a value to avoid SQL injections

```
<?php

$escapedStr = $connection->escapeString('some dangerous value');
```

public *string* **bindParams** (*string* $sqlStatement, *array* $params) inherited from Phalcon\Db\Adapter\Pdo

Manually bind params to a SQL statement. This method requires an active connection to a database system

```
<?php

$sql = $connection->bindParams('SELECT * FROM robots WHERE name = ?0', array('Bender'));
  echo $sql; // SELECT * FROM robots WHERE name = 'Bender'
```

public *array* **convertBoundParams** (*string* $sql, *array* $params) inherited from Phalcon\Db\Adapter\Pdo

Converts bound parameters such as :name: or ?1 into PDO bind params ?

```
<?php

 print_r($connection->convertBoundParams('SELECT * FROM robots WHERE name = :name:', array('Bender'))
```

public *int* **lastInsertId** ([*string* $sequenceName]) inherited from Phalcon\Db\Adapter\Pdo

Returns the insert id for the auto_increment/serial column inserted in the lastest executed SQL statement

```
<?php

 //Inserting a new robot
 $success = $connection->insert(
     "robots",
     array("Astro Boy", 1952),
     array("name", "year")
 );

 //Getting the generated id
 $id = $connection->lastInsertId();
```

public *boolean* **begin** () inherited from Phalcon\Db\Adapter\Pdo

Starts a transaction in the connection

public *boolean* **rollback** () inherited from Phalcon\Db\Adapter\Pdo

Rollbacks the active transaction in the connection

public *boolean* **commit** () inherited from Phalcon\Db\Adapter\Pdo

Commits the active transaction in the connection

public *boolean* **isUnderTransaction** () inherited from Phalcon\Db\Adapter\Pdo

Checks whether the connection is under a transaction

```
<?php

 $connection->begin();
 var_dump($connection->isUnderTransaction()); //true
```

public *PDO* **getInternalHandler** () inherited from Phalcon\Db\Adapter\Pdo

Return internal PDO handler

public *Phalcon\Db\Index* [] **describeIndexes** (*string* $table, [*string* $schema]) inherited from Phalcon\Db\Adapter\Pdo

Lists table indexes

```
<?php

 print_r($connection->describeIndexes('robots_parts'));
```

public *Phalcon\Db\Reference* [] **describeReferences** (*string* $table, [*string* $schema]) inherited from Phalcon\Db\Adapter\Pdo

Lists table references

```
<?php

 print_r($connection->describeReferences('robots_parts'));
```

public *array* **tableOptions** (*string* $tableName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter\Pdo

Gets creation options from a table

```
<?php

 print_r($connection->tableOptions('robots'));
```

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\Db\Adapter

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\Db\Adapter

Returns the internal event manager

public *array* **fetchOne** (*string* $sqlQuery, [*int* $fetchMode], [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter

Returns the first row in a SQL query result

```
<?php

//Getting first robot
$robot = $connection->fecthOne("SELECT * FROM robots");
print_r($robot);

//Getting first robot with associative indexes only
$robot = $connection->fecthOne("SELECT * FROM robots", Phalcon\Db::FETCH_ASSOC);
print_r($robot);
```

public *array* **fetchAll** (*string* $sqlQuery, [*int* $fetchMode], [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter

Dumps the complete result of a query into an array

```
<?php

//Getting all robots
$robots = $connection->fetchAll("SELECT * FROM robots");
foreach($robots as $robot){
    print_r($robot);
}

//Getting all robots with associative indexes only
$robots = $connection->fetchAll("SELECT * FROM robots", Phalcon\Db::FETCH_ASSOC);
foreach($robots as $robot){
    print_r($robot);
}
```

public *boolean* **insert** (*string* $table, *array* $values, [*array* $fields], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Inserts data into a table using custom RBDM SQL syntax

```
<?php

 //Inserting a new robot
 $success = $connection->insert(
     "robots",
     array("Astro Boy", 1952),
     array("name", "year")
```

```
);

//Next SQL sentence is sent to the database system
INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **update** (*string* $table, *array* $fields, *array* $values, [*string* $whereCondition], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Updates data on a table using custom RBDM SQL syntax

```
<?php

//Updating existing robot
$success = $connection->update(
    "robots",
    array("name")
    array("New Astro Boy"),
    "id = 101"
);

//Next SQL sentence is sent to the database system
UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101
```

public *boolean* **delete** (*string* $table, [*string* $whereCondition], [*array* $placeholders], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Deletes data from a table using custom RBDM SQL syntax

```
<?php

//Deleting existing robot
$success = $connection->delete(
    "robots",
    "id = 101"
);

//Next SQL sentence is generated
DELETE FROM `robots` WHERE `id` = 101
```

public *string* **getColumnList** (*array* $columnList) inherited from Phalcon\Db\Adapter

Gets a list of columns

public *string* **limit** (*string* $sqlQuery, *int* $number) inherited from Phalcon\Db\Adapter

Appends a LIMIT clause to $sqlQuery argument

```
<?php

    echo $connection->limit("SELECT * FROM robots", 5);
```

public *string* **tableExists** (*string* $tableName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter

Generates SQL checking for the existence of a schema.table

```
<?php

    var_dump($connection->tableExists("blog", "posts"));
```

public *string* **viewExists** (*string* $viewName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter

Generates SQL checking for the existence of a schema.view

```
<?php

 var_dump($connection->viewExists("active_users", "posts"));
```

public *string* **forUpdate** (*string* $sqlQuery) inherited from Phalcon\Db\Adapter

Returns a SQL modified with a FOR UPDATE clause

public *string* **sharedLock** (*string* $sqlQuery) inherited from Phalcon\Db\Adapter

Returns a SQL modified with a LOCK IN SHARE MODE clause

public *boolean* **createTable** (*string* $tableName, *string* $schemaName, *array* $definition) inherited from Phalcon\Db\Adapter

Creates a table

public *boolean* **dropTable** (*string* $tableName, *string* $schemaName, [*boolean* $ifExists]) inherited from Phalcon\Db\Adapter

Drops a table from a schema/database

public *boolean* **addColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Adds a column to a table

public *boolean* **modifyColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Modifies a table column based on a definition

public *boolean* **dropColumn** (*string* $tableName, *string* $schemaName, *string* $columnName) inherited from Phalcon\Db\Adapter

Drops a column from a table

public *boolean* **addIndex** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index) inherited from Phalcon\Db\Adapter

Adds an index to a table

public *boolean* **dropIndex** (*string* $tableName, *string* $schemaName, *string* $indexName) inherited from Phalcon\Db\Adapter

Drop an index from a table

public *boolean* **addPrimaryKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index) inherited from Phalcon\Db\Adapter

Adds a primary key to a table

public *boolean* **dropPrimaryKey** (*string* $tableName, *string* $schemaName) inherited from Phalcon\Db\Adapter

Drops a table's primary key

public *boolean true* **addForeignKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ReferenceInterface* $reference) inherited from Phalcon\Db\Adapter

Adds a foreign key to a table

public *boolean true* **dropForeignKey** (*string* $tableName, *string* $schemaName, *string* $referenceName) inherited from Phalcon\Db\Adapter

Drops a foreign key from a table

public *string* **getColumnDefinition** (*Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Returns the SQL column definition from a column

public *array* **listTables** ([*string* $schemaName]) inherited from Phalcon\Db\Adapter

List all tables on a database

```
<?php

    print_r($connection->listTables("blog");
```

public *array* **getDescriptor** () inherited from Phalcon\Db\Adapter

Return descriptor used to connect to the active database

public *string* **getConnectionId** () inherited from Phalcon\Db\Adapter

Gets the active connection unique identifier

public *string* **getSQLStatement** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *string* **getRealSQLStatement** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object without replace bound paramters

public *array* **getSQLVariables** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *array* **getSQLBindTypes** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *string* **getType** () inherited from Phalcon\Db\Adapter

Returns type of database system the adapter is used for

public *string* **getDialectType** () inherited from Phalcon\Db\Adapter

Returns the name of the dialect used

public *Phalcon\Db\DialectInterface* **getDialect** () inherited from Phalcon\Db\Adapter

Returns internal dialect instance

2.48.50 Class Phalcon\Db\Adapter\Pdo\Sqlite

extends Phalcon\Db\Adapter\Pdo

implements Phalcon\Events\EventsAwareInterface, *Phalcon\Db\AdapterInterface*

Specific functions for the Sqlite database system

```
<?php

 $config = array(
  "dbname" => "/tmp/test.sqlite"
 );

 $connection = new Phalcon\Db\Adapter\Pdo\Sqlite($config);
```

Methods

public *boolean* **connect** ([*array* $descriptor])

This method is automatically called in Phalcon\Db\Adapter\Pdo constructor. Call it when you need to restore a database connection.

public *Phalcon\Db\Column* [] **describeColumns** (*string* $table, [*string* $schema])

Returns an array of Phalcon\Db\Column objects describing a table

```
<?php

 print_r($connection->describeColumns("posts")); ?>
```

public *Phalcon\Db\Index* [] **describeIndexes** (*string* $table, [*string* $schema])

Lists table indexes

public *Phalcon\Db\Reference* [] **describeReferences** (*string* $table, [*string* $schema])

Lists table references

public **__construct** (*array* $descriptor) inherited from Phalcon\Db\Adapter\Pdo

Constructor for Phalcon\Db\Adapter\Pdo

public *PDOStatement* **prepare** (*string* $sqlStatement) inherited from Phalcon\Db\Adapter\Pdo

Returns a PDO prepared statement to be executed with 'executePrepared'

```
<?php

 $statement = $db->prepare('SELECT * FROM robots WHERE name = :name');
 $result = $connection->executePrepared($statement, array('name' => 'Voltron'));
```

public *PDOStatement* **executePrepared** (*PDOStatement* $statement, *array* $placeholders, *array* $dataTypes) inherited from Phalcon\Db\Adapter\Pdo

Executes a prepared statement binding. This function uses integer indexes starting from zero

```
<?php

 $statement = $db->prepare('SELECT * FROM robots WHERE name = :name');
 $result = $connection->executePrepared($statement, array('name' => 'Voltron'));
```

public *Phalcon\Db\ResultInterface* **query** (*string* $sqlStatement, [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter\Pdo

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server is returning rows

```
<?php

//Querying data
$resultset = $connection->query("SELECT * FROM robots WHERE type='mechanical'");
$resultset = $connection->query("SELECT * FROM robots WHERE type=?", array("mechanical"));
```

public *boolean* **execute** (*string* $sqlStatement, [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter\Pdo

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server doesn't return any row

```
<?php

//Inserting data
$success = $connection->execute("INSERT INTO robots VALUES (1, 'Astro Boy')");
$success = $connection->execute("INSERT INTO robots VALUES (?, ?)", array(1, 'Astro Boy'));
```

public *int* **affectedRows** () inherited from Phalcon\Db\Adapter\Pdo

Returns the number of affected rows by the lastest INSERT/UPDATE/DELETE executed in the database system

```
<?php

$connection->execute("DELETE FROM robots");
echo $connection->affectedRows(), ' were deleted';
```

public *boolean* **close** () inherited from Phalcon\Db\Adapter\Pdo

Closes the active connection returning success. Phalcon automatically closes and destroys active connections when the request ends

public *string* **escapeIdentifier** (*string* $identifier) inherited from Phalcon\Db\Adapter\Pdo

Escapes a column/table/schema name

```
<?php

$escapedTable = $connection->escapeIdentifier('robots');
```

public *string* **escapeString** (*string* $str) inherited from Phalcon\Db\Adapter\Pdo

Escapes a value to avoid SQL injections

```
<?php

$escapedStr = $connection->escapeString('some dangerous value');
```

public *string* **bindParams** (*string* $sqlStatement, *array* $params) inherited from Phalcon\Db\Adapter\Pdo

Manually bind params to a SQL statement. This method requires an active connection to a database system

```
<?php

$sql = $connection->bindParams('SELECT * FROM robots WHERE name = ?0', array('Bender'));
  echo $sql; // SELECT * FROM robots WHERE name = 'Bender'
```

public *array* **convertBoundParams** (*string* $sql, *array* $params) inherited from Phalcon\Db\Adapter\Pdo

Converts bound parameters such as :name: or ?1 into PDO bind params ?

```
<?php

 print_r($connection->convertBoundParams('SELECT * FROM robots WHERE name = :name:', array('Bender'))
```

public *int* **lastInsertId** ([*string* $sequenceName]) inherited from Phalcon\Db\Adapter\Pdo

Returns the insert id for the auto_increment/serial column inserted in the lastest executed SQL statement

```
<?php

 //Inserting a new robot
 $success = $connection->insert(
     "robots",
     array("Astro Boy", 1952),
```

```
    array("name", "year")
);

//Getting the generated id
$id = $connection->lastInsertId();
```

public *boolean* **begin** () inherited from Phalcon\Db\Adapter\Pdo

Starts a transaction in the connection

public *boolean* **rollback** () inherited from Phalcon\Db\Adapter\Pdo

Rollbacks the active transaction in the connection

public *boolean* **commit** () inherited from Phalcon\Db\Adapter\Pdo

Commits the active transaction in the connection

public *boolean* **isUnderTransaction** () inherited from Phalcon\Db\Adapter\Pdo

Checks whether the connection is under a transaction

```
<?php

$connection->begin();
var_dump($connection->isUnderTransaction()); //true
```

public *PDO* **getInternalHandler** () inherited from Phalcon\Db\Adapter\Pdo

Return internal PDO handler

public *array* **tableOptions** (*string* $tableName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter\Pdo

Gets creation options from a table

```
<?php

print_r($connection->tableOptions('robots'));
```

public *Phalcon\Db\RawValue* **getDefaultIdValue** () inherited from Phalcon\Db\Adapter\Pdo

Returns the default identity value to be inserted in an identity column

```
<?php

//Inserting a new robot with a valid default value for the column 'id'
$success = $connection->insert(
    "robots",
    array($connection->getDefaultIdValue(), "Astro Boy", 1952),
    array("id", "name", "year")
);
```

public *boolean* **supportSequences** () inherited from Phalcon\Db\Adapter\Pdo

Check whether the database system requires a sequence to produce auto-numeric values

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\Db\Adapter

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\Db\Adapter

Returns the internal event manager

public *array* **fetchOne** (*string* $sqlQuery, [*int* $fetchMode], [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter

Returns the first row in a SQL query result

```
<?php

//Getting first robot
$robot = $connection->fecthOne("SELECT * FROM robots");
print_r($robot);

//Getting first robot with associative indexes only
$robot = $connection->fecthOne("SELECT * FROM robots", Phalcon\Db::FETCH_ASSOC);
print_r($robot);
```

public *array* **fetchAll** (*string* $sqlQuery, [*int* $fetchMode], [*array* $bindParams], [*array* $bindTypes]) inherited from Phalcon\Db\Adapter

Dumps the complete result of a query into an array

```
<?php

//Getting all robots
$robots = $connection->fetchAll("SELECT * FROM robots");
foreach($robots as $robot){
    print_r($robot);
}

//Getting all robots with associative indexes only
$robots = $connection->fetchAll("SELECT * FROM robots", Phalcon\Db::FETCH_ASSOC);
foreach($robots as $robot){
    print_r($robot);
}
```

public *boolean* **insert** (*string* $table, *array* $values, [*array* $fields], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Inserts data into a table using custom RBDM SQL syntax

```
<?php

 //Inserting a new robot
 $success = $connection->insert(
     "robots",
     array("Astro Boy", 1952),
     array("name", "year")
 );

 //Next SQL sentence is sent to the database system
 INSERT INTO `robots` (`name`, `year`) VALUES ("Astro boy", 1952);
```

public *boolean* **update** (*string* $table, *array* $fields, *array* $values, [*string* $whereCondition], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Updates data on a table using custom RBDM SQL syntax

```
<?php

 //Updating existing robot
 $success = $connection->update(
     "robots",
     array("name")
     array("New Astro Boy"),
     "id = 101"
```

```
);

//Next SQL sentence is sent to the database system
UPDATE `robots` SET `name` = "Astro boy" WHERE id = 101
```

public *boolean* **delete** (*string* $table, [*string* $whereCondition], [*array* $placeholders], [*array* $dataTypes]) inherited from Phalcon\Db\Adapter

Deletes data from a table using custom RBDM SQL syntax

```
<?php

//Deleting existing robot
$success = $connection->delete(
    "robots",
    "id = 101"
);

//Next SQL sentence is generated
DELETE FROM `robots` WHERE `id` = 101
```

public *string* **getColumnList** (*array* $columnList) inherited from Phalcon\Db\Adapter

Gets a list of columns

public *string* **limit** (*string* $sqlQuery, *int* $number) inherited from Phalcon\Db\Adapter

Appends a LIMIT clause to $sqlQuery argument

```
<?php

    echo $connection->limit("SELECT * FROM robots", 5);
```

public *string* **tableExists** (*string* $tableName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter

Generates SQL checking for the existence of a schema.table

```
<?php

    var_dump($connection->tableExists("blog", "posts"));
```

public *string* **viewExists** (*string* $viewName, [*string* $schemaName]) inherited from Phalcon\Db\Adapter

Generates SQL checking for the existence of a schema.view

```
<?php

 var_dump($connection->viewExists("active_users", "posts"));
```

public *string* **forUpdate** (*string* $sqlQuery) inherited from Phalcon\Db\Adapter

Returns a SQL modified with a FOR UPDATE clause

public *string* **sharedLock** (*string* $sqlQuery) inherited from Phalcon\Db\Adapter

Returns a SQL modified with a LOCK IN SHARE MODE clause

public *boolean* **createTable** (*string* $tableName, *string* $schemaName, *array* $definition) inherited from Phalcon\Db\Adapter

Creates a table

public *boolean* **dropTable** (*string* $tableName, *string* $schemaName, [*boolean* $ifExists]) inherited from Phalcon\Db\Adapter

Drops a table from a schema/database

public *boolean* **addColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Adds a column to a table

public *boolean* **modifyColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Modifies a table column based on a definition

public *boolean* **dropColumn** (*string* $tableName, *string* $schemaName, *string* $columnName) inherited from Phalcon\Db\Adapter

Drops a column from a table

public *boolean* **addIndex** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index) inherited from Phalcon\Db\Adapter

Adds an index to a table

public *boolean* **dropIndex** (*string* $tableName, *string* $schemaName, *string* $indexName) inherited from Phalcon\Db\Adapter

Drop an index from a table

public *boolean* **addPrimaryKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index) inherited from Phalcon\Db\Adapter

Adds a primary key to a table

public *boolean* **dropPrimaryKey** (*string* $tableName, *string* $schemaName) inherited from Phalcon\Db\Adapter

Drops a table's primary key

public *boolean true* **addForeignKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ReferenceInterface* $reference) inherited from Phalcon\Db\Adapter

Adds a foreign key to a table

public *boolean true* **dropForeignKey** (*string* $tableName, *string* $schemaName, *string* $referenceName) inherited from Phalcon\Db\Adapter

Drops a foreign key from a table

public *string* **getColumnDefinition** (*Phalcon\Db\ColumnInterface* $column) inherited from Phalcon\Db\Adapter

Returns the SQL column definition from a column

public *array* **listTables** ([*string* $schemaName]) inherited from Phalcon\Db\Adapter

List all tables on a database

```
<?php

    print_r($connection->listTables("blog");
```

public *array* **getDescriptor** () inherited from Phalcon\Db\Adapter

Return descriptor used to connect to the active database

public *string* **getConnectionId** () inherited from Phalcon\Db\Adapter

Gets the active connection unique identifier

public *string* **getSQLStatement** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *string* **getRealSQLStatement** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object without replace bound paramters

public *array* **getSQLVariables** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *array* **getSQLBindTypes** () inherited from Phalcon\Db\Adapter

Active SQL statement in the object

public *string* **getType** () inherited from Phalcon\Db\Adapter

Returns type of database system the adapter is used for

public *string* **getDialectType** () inherited from Phalcon\Db\Adapter

Returns the name of the dialect used

public *Phalcon\Db\DialectInterface* **getDialect** () inherited from Phalcon\Db\Adapter

Returns internal dialect instance

2.48.51 Class Phalcon\Db\Column

implements Phalcon\Db\ColumnInterface

Allows to define columns to be used on create or alter table operations

```
<?php

use Phalcon\Db\Column as Column;

 //column definition
 $column = new Column("id", array(
   "type" => Column::TYPE_INTEGER,
   "size" => 10,
   "unsigned" => true,
   "notNull" => true,
   "autoIncrement" => true,
   "first" => true
 ));

 //add column to existing table
 $connection->addColumn("robots", null, $column);
```

Constants

integer **TYPE_INTEGER**

integer **TYPE_DATE**

integer **TYPE_VARCHAR**

integer **TYPE_DECIMAL**

integer **TYPE_DATETIME**

integer **TYPE_CHAR**

integer **TYPE_TEXT**

integer **TYPE_FLOAT**

integer **TYPE_BOOLEAN**

integer **BIND_PARAM_NULL**

integer **BIND_PARAM_INT**

integer **BIND_PARAM_STR**

integer **BIND_PARAM_BOOL**

integer **BIND_PARAM_DECIMAL**

integer **BIND_SKIP**

Methods

public **__construct** (*string* $columnName, *array* $definition)

Phalcon\Db\Column constructor

public *string* **getSchemaName** ()

Returns schema's table related to column

public *string* **getName** ()

Returns column name

public *int* **getType** ()

Returns column type

public *int* **getSize** ()

Returns column size

public *int* **getScale** ()

Returns column scale

public *boolean* **isUnsigned** ()

Returns true if number column is unsigned

public *boolean* **isNotNull** ()

Not null

public *boolean* **isPrimary** ()

Column is part of the primary key?

public *boolean* **isAutoIncrement** ()

Auto-Increment

public *boolean* **isNumeric** ()

Check whether column have an numeric type

public *boolean* **isFirst** ()

Check whether column have first position in table

public *string* **getAfterPosition** ()

Check whether field absolute to position in table

public *int* **getBindType** ()

Returns the type of bind handling

public static *PhalconDbColumn* **__set_state** (*array* $data)

Restores the internal state of a Phalcon\Db\Column object

2.48.52 Class Phalcon\Db\Dialect

This is the base class to each database dialect. This implements common methods to transform intermediate code into its RDBM related syntax

Methods

public *string* **limit** (*string* $sqlQuery, *int* $number)

Generates the SQL for LIMIT clause

```
<?php

 $sql = $dialect->limit('SELECT * FROM robots', 10);
 echo $sql; // SELECT * FROM robots LIMIT 10
```

public *string* **forUpdate** (*string* $sqlQuery)

Returns a SQL modified with a FOR UPDATE clause

```
<?php

 $sql = $dialect->forUpdate('SELECT * FROM robots');
 echo $sql; // SELECT * FROM robots FOR UPDATE
```

public *string* **sharedLock** (*string* $sqlQuery)

Returns a SQL modified with a LOCK IN SHARE MODE clause

```
<?php

 $sql = $dialect->sharedLock('SELECT * FROM robots');
 echo $sql; // SELECT * FROM robots LOCK IN SHARE MODE
```

public *string* **getColumnList** (*array* $columnList)

Gets a list of columns with escaped identifiers

```
<?php

 echo $dialect->getColumnList(array('column1', 'column'));
```

public *string* **getSqlExpression** (*array* $expression, [*string* $escapeChar])

Transforms an intermediate representation for a expression into a database system valid expression

public *string* **getSqlTable** (*array* $table, [*string* $escapeChar])

Transform an intermediate representation for a schema/table into a database system valid expression

public *string* **select** (*array* $definition)

Builds a SELECT statement

2.48.53 Class Phalcon\Db\Dialect\Mysql

extends Phalcon\Db\Dialect

implements Phalcon\Db\DialectInterface

Generates database specific SQL for the MySQL RBDM

Methods

public *string* **getColumnDefinition** (*Phalcon\Db\ColumnInterface* $column)

Gets the column name in MySQL

public *string* **addColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Generates SQL to add a column to a table

public *string* **modifyColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Generates SQL to modify a column in a table

public *string* **dropColumn** (*string* $tableName, *string* $schemaName, *string* $columnName)

Generates SQL to delete a column from a table

public *string* **addIndex** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index)

Generates SQL to add an index to a table

public *string* **dropIndex** (*string* $tableName, *string* $schemaName, *string* $indexName)

Generates SQL to delete an index from a table

public *string* **addPrimaryKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index)

Generates SQL to add the primary key to a table

public *string* **dropPrimaryKey** (*string* $tableName, *string* $schemaName)

Generates SQL to delete primary key from a table

public *string* **addForeignKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ReferenceInterface* $reference)

Generates SQL to add an index to a table

public *string* **dropForeignKey** (*string* $tableName, *string* $schemaName, *string* $referenceName)

Generates SQL to delete a foreign key from a table

protected *array* **_getTableOptions** ()

Generates SQL to add the table creation options

public *string* **createTable** (*string* $tableName, *string* $schemaName, *array* $definition)

Generates SQL to create a table in MySQL

public *string* **dropTable** (*string* $tableName, *string* $schemaName, [*boolean* $ifExists])

Generates SQL to drop a table

public *string* **tableExists** (*string* $tableName, [*string* $schemaName])

Generates SQL checking for the existence of a schema.table

```
<?php

 echo $dialect->tableExists("posts", "blog");
 echo $dialect->tableExists("posts");
```

public *string* **describeColumns** (*string* $table, [*string* $schema])

Generates SQL describing a table

```
<?php

print_r($dialect->describeColumns("posts")) ?>
```

public *array* **listTables** ([*string* $schemaName])

List all tables on database

```
<?php

print_r($dialect->listTables("blog")) ?>
```

public *string* **describeIndexes** (*string* $table, [*string* $schema])

Generates SQL to query indexes on a table

public *string* **describeReferences** (*string* $table, [*string* $schema])

Generates SQL to query foreign keys on a table

public *string* **tableOptions** (*string* $table, [*string* $schema])

Generates the SQL to describe the table creation options

public *string* **limit** (*string* $sqlQuery, *int* $number) inherited from Phalcon\Db\Dialect

Generates the SQL for LIMIT clause

```
<?php

 $sql = $dialect->limit('SELECT * FROM robots', 10);
 echo $sql; // SELECT * FROM robots LIMIT 10
```

public *string* **forUpdate** (*string* $sqlQuery) inherited from Phalcon\Db\Dialect

Returns a SQL modified with a FOR UPDATE clause

```
<?php

 $sql = $dialect->forUpdate('SELECT * FROM robots');
 echo $sql; // SELECT * FROM robots FOR UPDATE
```

public *string* **sharedLock** (*string* $sqlQuery) inherited from Phalcon\Db\Dialect

Returns a SQL modified with a LOCK IN SHARE MODE clause

```
<?php

 $sql = $dialect->sharedLock('SELECT * FROM robots');
 echo $sql; // SELECT * FROM robots LOCK IN SHARE MODE
```

public *string* **getColumnList** (*array* $columnList) inherited from Phalcon\Db\Dialect

Gets a list of columns with escaped identifiers

```
<?php

 echo $dialect->getColumnList(array('column1', 'column'));
```

public *string* **getSqlExpression** (*array* $expression, [*string* $escapeChar]) inherited from Phalcon\Db\Dialect

Transforms an intermediate representation for a expression into a database system valid expression

public *string* **getSqlTable** (*array* $table, [*string* $escapeChar]) inherited from Phalcon\Db\Dialect

Transform an intermediate representation for a schema/table into a database system valid expression

public *string* **select** (*array* $definition) inherited from Phalcon\Db\Dialect

Builds a SELECT statement

2.48.54 Class Phalcon\Db\Dialect\Postgresql

extends Phalcon\Db\Dialect

implements Phalcon\Db\DialectInterface

Generates database specific SQL for the PostgreSQL RBDM

Methods

public *string* **getColumnDefinition** (*Phalcon\Db\ColumnInterface* $column)

Gets the column name in PostgreSQL

public *string* **addColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Generates SQL to add a column to a table

public *string* **modifyColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Generates SQL to modify a column in a table

public *string* **dropColumn** (*string* $tableName, *string* $schemaName, *string* $columnName)

Generates SQL to delete a column from a table

public *string* **addIndex** (*string* $tableName, *string* $schemaName, *Phalcon\Db\Index* $index)

Generates SQL to add an index to a table

public *string* **dropIndex** (*string* $tableName, *string* $schemaName, *string* $indexName)

Generates SQL to delete an index from a table

public *string* **addPrimaryKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\Index* $index)

Generates SQL to add the primary key to a table

public *string* **dropPrimaryKey** (*string* $tableName, *string* $schemaName)

Generates SQL to delete primary key from a table

public *string* **addForeignKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ReferenceInterface* $reference)

Generates SQL to add an index to a table

public *string* **dropForeignKey** (*string* $tableName, *string* $schemaName, *string* $referenceName)

Generates SQL to delete a foreign key from a table

protected *array* **_getTableOptions** ()

Generates SQL to add the table creation options

public *string* **createTable** (*string* $tableName, *string* $schemaName, *array* $definition)

Generates SQL to create a table in PostgreSQL

public *boolean* **dropTable** (*string* $tableName, *string* $schemaName, [*boolean* $ifExists])

Generates SQL to drop a table

public *string* **tableExists** (*string* $tableName, [*string* $schemaName])

Generates SQL checking for the existence of a schema.table <code>echo $dialect->tableExists("posts", "blog") <code>echo $dialect->tableExists("posts")

public *string* **describeColumns** (*string* $table, [*string* $schema])

Generates a SQL describing a table <code>print_r($dialect->describeColumns("posts") ?>

public *array* **listTables** ([*string* $schemaName])

List all tables on database <code>print_r($dialect->listTables("blog") ?>

public *string* **describeIndexes** (*string* $table, [*string* $schema])

Generates SQL to query indexes on a table

public *string* **describeReferences** (*string* $table, [*string* $schema])

Generates SQL to query foreign keys on a table

public *string* **tableOptions** (*string* $table, [*string* $schema])

Generates the SQL to describe the table creation options

public *string* **limit** (*string* $sqlQuery, *int* $number) inherited from Phalcon\Db\Dialect

Generates the SQL for LIMIT clause

```
<?php

 $sql = $dialect->limit('SELECT * FROM robots', 10);
 echo $sql; // SELECT * FROM robots LIMIT 10
```

public *string* **forUpdate** (*string* $sqlQuery) inherited from Phalcon\Db\Dialect

Returns a SQL modified with a FOR UPDATE clause

```
<?php

 $sql = $dialect->forUpdate('SELECT * FROM robots');
 echo $sql; // SELECT * FROM robots FOR UPDATE
```

public *string* **sharedLock** (*string* $sqlQuery) inherited from Phalcon\Db\Dialect

Returns a SQL modified with a LOCK IN SHARE MODE clause

```
<?php

 $sql = $dialect->sharedLock('SELECT * FROM robots');
 echo $sql; // SELECT * FROM robots LOCK IN SHARE MODE
```

public *string* **getColumnList** (*array* $columnList) inherited from Phalcon\Db\Dialect

Gets a list of columns with escaped identifiers

```
<?php

 echo $dialect->getColumnList(array('column1', 'column'));
```

public *string* **getSqlExpression** (*array* $expression, [*string* $escapeChar]) inherited from Phalcon\Db\Dialect

Transforms an intermediate representation for a expression into a database system valid expression

public *string* **getSqlTable** (*array* $table, [*string* $escapeChar]) inherited from Phalcon\Db\Dialect

Transform an intermediate representation for a schema/table into a database system valid expression

public *string* **select** (*array* $definition) inherited from Phalcon\Db\Dialect

Builds a SELECT statement

2.48.55 Class Phalcon\Db\Dialect\Sqlite

extends Phalcon\Db\Dialect

implements Phalcon\Db\DialectInterface

Generates database specific SQL for the Sqlite RBDM

Methods

public *string* **getColumnDefinition** (*Phalcon\Db\ColumnInterface* $column)

Gets the column name in Sqlite

public *string* **addColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Generates SQL to add a column to a table

public *string* **modifyColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Generates SQL to modify a column in a table

public *string* **dropColumn** (*string* $tableName, *string* $schemaName, *string* $columnName)

Generates SQL to delete a column from a table

public *string* **addIndex** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index)

Generates SQL to add an index to a table

public *string* **dropIndex** (*string* $tableName, *string* $schemaName, *string* $indexName)

Generates SQL to delete an index from a table

public *string* **addPrimaryKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index)

Generates SQL to add the primary key to a table

public *string* **dropPrimaryKey** (*string* $tableName, *string* $schemaName)

Generates SQL to delete primary key from a table

public *string* **addForeignKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\Reference* $reference)

Generates SQL to add an index to a table

public *string* **dropForeignKey** (*string* $tableName, *string* $schemaName, *string* $referenceName)

Generates SQL to delete a foreign key from a table

protected *array* **_getTableOptions** ()

Generates SQL to add the table creation options

public *string* **createTable** (*string* $tableName, *string* $schemaName, *array* $definition)

Generates SQL to create a table in Sqlite

public *boolean* **dropTable** (*string* $tableName, *string* $schemaName, [*boolean* $ifExists])

Generates SQL to drop a table

public *string* **tableExists** (*string* $tableName, [*string* $schemaName])

Generates SQL checking for the existence of a schema.table <code>echo $dialect->tableExists("posts", "blog") <code>echo $dialect->tableExists("posts")

public *string* **describeColumns** (*string* $table, [*string* $schema])

Generates a SQL describing a table <code>print_r($dialect->describeColumns("posts") ?>

public *array* **listTables** ([*string* $schemaName])

List all tables on database <code>print_r($dialect->listTables("blog") ?>

public *string* **describeIndexes** (*string* $table, [*string* $schema])

Generates SQL to query indexes on a table

public *string* **describeIndex** (*string* $indexName)

Generates SQL to query indexes detail on a table

public *string* **describeReferences** (*string* $table, [*string* $schema])

Generates SQL to query foreign keys on a table

public *string* **tableOptions** (*string* $table, [*string* $schema])

Generates the SQL to describe the table creation options

public *string* **limit** (*string* $sqlQuery, *int* $number) inherited from Phalcon\Db\Dialect

Generates the SQL for LIMIT clause

```
<?php

 $sql = $dialect->limit('SELECT * FROM robots', 10);
 echo $sql; // SELECT * FROM robots LIMIT 10
```

public *string* **forUpdate** (*string* $sqlQuery) inherited from Phalcon\Db\Dialect

Returns a SQL modified with a FOR UPDATE clause

```
<?php

 $sql = $dialect->forUpdate('SELECT * FROM robots');
 echo $sql; // SELECT * FROM robots FOR UPDATE
```

public *string* **sharedLock** (*string* $sqlQuery) inherited from Phalcon\Db\Dialect

Returns a SQL modified with a LOCK IN SHARE MODE clause

```
<?php

 $sql = $dialect->sharedLock('SELECT * FROM robots');
 echo $sql; // SELECT * FROM robots LOCK IN SHARE MODE
```

public *string* **getColumnList** (*array* $columnList) inherited from Phalcon\Db\Dialect

Gets a list of columns with escaped identifiers

```
<?php

 echo $dialect->getColumnList(array('column1', 'column'));
```

public *string* **getSqlExpression** (*array* $expression, [*string* $escapeChar]) inherited from Phalcon\Db\Dialect

Transforms an intermediate representation for a expression into a database system valid expression

public *string* **getSqlTable** (*array* $table, [*string* $escapeChar]) inherited from Phalcon\Db\Dialect

Transform an intermediate representation for a schema/table into a database system valid expression

public *string* **select** (*array* $definition) inherited from Phalcon\Db\Dialect

Builds a SELECT statement

2.48.56 Class Phalcon\Db\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.57 Class Phalcon\Db\Index

implements Phalcon\Db\IndexInterface

Allows to define indexes to be used on tables. Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index

Methods

public **__construct** (*string* $indexName, *array* $columns)

Phalcon\Db\Index constructor

public *string* **getName** ()

Gets the index name

public *array* **getColumns** ()

Gets the columns that comprends the index

public static *Phalcon\Db\IndexInterface* **__set_state** (*array* $data)

Restore a Phalcon\Db\Index object from export

2.48.58 Class Phalcon\Db\Profiler

Instances of Phalcon\Db can generate execution profiles on SQL statements sent to the relational database. Profiled information includes execution time in miliseconds. This helps you to identify bottlenecks in your applications.

```
<?php

$profiler = new Phalcon\Db\Profiler();

//Set the connection profiler
$connection->setProfiler($profiler);

$sql = "SELECT buyer_name, quantity, product_name
FROM buyers LEFT JOIN products ON
buyers.pid=products.id";

//Execute a SQL statement
$connection->query($sql);

//Get the last profile in the profiler
$profile = $profiler->getLastProfile();

echo "SQL Statement: ", $profile->getSQLStatement(), "\n";
echo "Start Time: ", $profile->getInitialTime(), "\n";
echo "Final Time: ", $profile->getFinalTime(), "\n";
echo "Total Elapsed Time: ", $profile->getTotalElapsedSeconds(), "\n";
```

Methods

public **__construct** ()

Phalcon\Db\Profiler constructor

public *Phalcon\Db\Profiler* **startProfile** (*string* $sqlStatement)

Starts the profile of a SQL sentence

public *Phalcon\Db\Profiler* **stopProfile** ()

Stops the active profile

public *integer* **getNumberTotalStatements** ()

Returns the total number of SQL statements processed

public *double* **getTotalElapsedSeconds** ()

Returns the total time in seconds spent by the profiles

public *Phalcon\Db\Profiler\Item* [] **getProfiles** ()

Returns all the processed profiles

public *Phalcon\Db\Profiler* **reset** ()

Resets the profiler, cleaning up all the profiles

public *Phalcon\Db\Profiler\Item* **getLastProfile** ()

Returns the last profile executed in the profiler

2.48.59 Class Phalcon\Db\Profiler\Item

This class identifies each profile in a Phalcon\Db\Profiler

Methods

public **setSQLStatement** (*string* $sqlStatement)

Sets the SQL statement related to the profile

public *string* **getSQLStatement** ()

Returns the SQL statement related to the profile

public **setInitialTime** (*int* $initialTime)

Sets the timestamp on when the profile started

public **setFinalTime** (*int* $finalTime)

Sets the timestamp on when the profile ended

public *double* **getInitialTime** ()

Returns the initial time in milseconds on when the profile started

public *double* **getFinalTime** ()

Returns the initial time in milseconds on when the profile ended

public *double* **getTotalElapsedSeconds** ()

Returns the total time in seconds spent by the profile

2.48.60 Class Phalcon\Db\RawValue

This class allows to insert/update raw data without quoting or formating. The next example shows how to use the MySQL now() function as a field value.

```
<?php

$subscriber = new Subscribers();
$subscriber->email = 'andres@phalconphp.com';
$subscriber->created_at = new Phalcon\Db\RawValue('now()');
$subscriber->save();
```

Methods

public **__construct** (*string* $value)

Phalcon\Db\RawValue constructor

public *string* **getValue** ()

Returns internal raw value without quoting or formating

public **__toString** ()

Magic method __toString returns raw value without quoting or formating

2.48.61 Class Phalcon\Db\Reference

implements Phalcon\Db\ReferenceInterface

Allows to define reference constraints on tables

```
<?php

$reference = new Phalcon\Db\Reference("field_fk", array(
    'referencedSchema' => "invoicing",
    'referencedTable' => "products",
    'columns' => array("product_type", "product_code"),
    'referencedColumns' => array("type", "code")
));
```

Methods

public **__construct** (*string* $referenceName, *array* $definition)

Phalcon\Db\Reference constructor

public *string* **getName** ()

Gets the index name

public *string* **getSchemaName** ()

Gets the schema where referenced table is

public *string* **getReferencedSchema** ()

Gets the schema where referenced table is

public *array* **getColumns** ()

Gets local columns which reference is based

public *string* **getReferencedTable** ()

Gets the referenced table

public *array* **getReferencedColumns** ()

Gets referenced columns

public static *Phalcon\Db\Reference* **__set_state** (*array* $data)

Restore a Phalcon\Db\Reference object from export

2.48.62 Class Phalcon\Db\Result\Pdo

Encapsulates the resultset internals

```
<?php

$result = $connection->query("SELECT * FROM robots ORDER BY name");
$result->setFetchMode(Phalcon\Db::FETCH_NUM);
while($robot = $result->fetchArray()){
    print_r($robot);
}
```

Methods

public **__construct** (*Phalcon\Db\AdapterInterface* $connection, *PDOStatement* $result, [*string* $sqlStatement], [*array* $bindParams], [*array* $bindTypes])

Phalcon\Db\Result\Pdo constructor

public *boolean* **execute** ()

Allows to executes the statement again. Some database systems don't support scrollable cursors, So, as cursors are forward only, we need to execute the cursor again to fetch rows from the begining

public *mixed* **fetch** ()

Fetches an array/object of strings that corresponds to the fetched row, or FALSE if there are no more rows. This method is affected by the active fetch flag set using Phalcon\Db\Result\Pdo::setFetchMode

```
<?php

$result = $connection->query("SELECT * FROM robots ORDER BY name");
$result->setFetchMode(Phalcon\Db::FETCH_OBJ);
while($robot = $result->fetch()){
    echo $robot->name;
}
```

public *mixed* **fetchArray** ()

Returns an array of strings that corresponds to the fetched row, or FALSE if there are no more rows. This method is affected by the active fetch flag set using Phalcon\Db\Result\Pdo::setFetchMode

```
<?php

$result = $connection->query("SELECT * FROM robots ORDER BY name");
$result->setFetchMode(Phalcon\Db::FETCH_NUM);
while($robot = $result->fetchArray()){
```

```
    print_r($robot);
}
```

public *array* **fetchAll** ()

Returns an array of arrays containing all the records in the result This method is affected by the active fetch flag set using Phalcon\Db\Result\Pdo::setFetchMode

```
<?php

$result = $connection->query("SELECT * FROM robots ORDER BY name");
$robots = $result->fetchAll();
```

public *int* **numRows** ()

Gets number of rows returned by a resulset

```
<?php

$result = $connection->query("SELECT * FROM robots ORDER BY name");
echo 'There are ', $result->numRows(), ' rows in the resulset';
```

public **dataSeek** (*int* $number)

Moves internal resulset cursor to another position letting us to fetch a certain row

```
<?php

$result = $connection->query("SELECT * FROM robots ORDER BY name");
$result->dataSeek(2); // Move to third row on result
$row = $result->fetch(); // Fetch third row
```

public **setFetchMode** (*int* $fetchMode)

Changes the fetching mode affecting Phalcon\Db\Result\Pdo::fetch()

```
<?php

//Return array with integer indexes
$result->setFetchMode(Phalcon\Db::FETCH_NUM);

//Return associative array without integer indexes
$result->setFetchMode(Phalcon\Db::FETCH_ASSOC);

//Return associative array together with integer indexes
$result->setFetchMode(Phalcon\Db::FETCH_BOTH);

//Return an object
$result->setFetchMode(Phalcon\Db::FETCH_OBJ);
```

public *PDOStatement* **getInternalResult** ()

Gets the internal PDO result object

2.48.63 Class Phalcon\Dispatcher

implements Phalcon\DispatcherInterface, Phalcon\DI\InjectionAwareInterface, Phalcon\Events\EventsAwareInterface

This is the base class for Phalcon\Mvc\Dispatcher and Phalcon\CLI\Dispatcher. This class can't be instantiated directly, you can use it to create your own dispatchers

Constants

integer **EXCEPTION_NO_DI**

integer **EXCEPTION_CYCLIC_ROUTING**

integer **EXCEPTION_HANDLER_NOT_FOUND**

integer **EXCEPTION_INVALID_HANDLER**

integer **EXCEPTION_INVALID_PARAMS**

integer **EXCEPTION_ACTION_NOT_FOUND**

Methods

public **__construct** ()

Phalcon\Dispatcher constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager)

Sets the events manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the internal event manager

public **setActionSuffix** (*string* $actionSuffix)

Sets the default action suffix

public **setNamespaceName** (*string* $namespaceName)

Sets a namespace to be prepended to the handler name

public *string* **getNamespaceName** ()

Gets a namespace to be prepended to the current handler name

public **setDefaultNamespace** (*string* $namespace)

Sets the default namespace

public *string* **getDefaultNamespace** ()

Returns the default namespace

public **setDefaultAction** (*string* $actionName)

Sets the default action name

public **setActionName** (*string* $actionName)

Sets the action name to be dispatched

public *string* **getActionName** ()

Gets the lastest dispatched action name

public **setParams** (*array* $params)

Sets action params to be dispatched

public *array* **getParams** ()

Gets action params

public **setParam** (*mixed* $param, *mixed* $value)

Set a param by its name or numeric index

public *mixed* **getParam** (*mixed* $param, [*string|array* $filters], [*mixed* $defaultValue])

Gets a param by its name or numeric index

public *string* **getActiveMethod** ()

Returns the current method to be/executed in the dispatcher

public *boolean* **isFinished** ()

Checks if the dispatch loop is finished or has more pendent controllers/tasks to disptach

public **setReturnedValue** (*mixed* $value)

Sets the latest returned value by an action manually

public *mixed* **getReturnedValue** ()

Returns value returned by the lastest dispatched action

public *object* **dispatch** ()

Dispatches a handle action taking into account the routing parameters

public **forward** (*array* $forward)

Forwards the execution flow to another controller/action

2.48.64 Class Phalcon\Escaper

implements Phalcon\EscaperInterface

Escapes different kinds of text securing them. By using this component you may prevent XSS attacks. This component only works with UTF-8. The PREG extension needs to be compiled with UTF-8 support.

```
<?php

$escaper = new Phalcon\Escaper();
$escaped = $escaper->escapeCss("font-family: <Verdana>");
echo $escaped; // font\2D family\3A \20 \3C Verdana\3E
```

Methods

public **setEncoding** (*string* $encoding)

Sets the encoding to be used by the escaper

```
<?php

 $escaper->setEncoding('utf-8');
```

public *string* **getEncoding** ()

Returns the internal encoding used by the escaper

public **setHtmlQuoteType** (*int* $quoteType)

Sets the HTML quoting type for htmlspecialchars

```
<?php

 $escaper->setHtmlQuoteType(ENT_XHTML);
```

public *string* **detectEncoding** (*string* $str)

Detect the character encoding of a string to be handled by an encoder Special-handling for chr(172) and chr(128) to chr(159) which fail to be detected by mb_detect_encoding()

public *string* **normalizeEncoding** (*string* $str)

Utility to normalize a string's encoding to UTF-32.

public *string* **escapeHtml** (*string* $text)

Escapes a HTML string. Internally uses htmlspeciarchars

public *string* **escapeHtmlAttr** (*string* $attribute)

Escapes a HTML attribute string

public *string* **escapeCss** (*string* $css)

Escape CSS strings by replacing non-alphanumeric chars by their hexadecimal escaped representation

public *string* **escapeJs** (*string* $js)

Escape javascript strings by replacing non-alphanumeric chars by their hexadecimal escaped representation

public *string* **escapeUrl** (*string* $url)

Escapes a URL. Internally uses rawurlencode

2.48.65 Class Phalcon\Escaper\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.66 Class Phalcon\Events\Event

This class offers contextual information of a fired event in the EventsManager

Methods

public **__construct** (*string* $type, *object* $source, [*mixed* $data], [*boolean* $cancelable])

Phalcon\Events\Event constructor

public **setType** (*string* $eventType)

Set the event's type

public *string* **getType** ()

Returns the event's type

public *object* **getSource** ()

Returns the event's source

public **setData** (*string* $data)

Set the event's data

public *mixed* **getData** ()

Returns the event's data

public **setCancelable** (*boolean* $cancelable)

Sets if the event is cancelable

public *boolean* **getCancelable** ()

Check whether the event is cancelable

public **stop** ()

Stops the event preventing propagation

public **isStopped** ()

Check whether the event is currently stopped

2.48.67 Class Phalcon\Events\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.68 Class Phalcon\Events\Manager

implements Phalcon\Events\ManagerInterface

Phalcon Events Manager, offers an easy way to intercept and manipulate, if needed, the normal flow of operation. With the EventsManager the developer can create hooks or plugins that will offer monitoring of data, manipulation, conditional execution and much more.

Methods

public **attach** (*string* $eventType, *object* $handler, [*int* $priority])

Attach a listener to the events manager

public **collectResponses** (*boolean* $collect)

Tells the event manager if it needs to collect all the responses returned by every registered listener in a single fire

public **isCollecting** ()

Check if the events manager is collecting all all the responses returned by every registered listener in a single fire

public *array* **getResponses** ()

Returns all the responses returned by every handler executed by the last 'fire' executed

public **dettachAll** ([*string* $type])

Removes all events from the EventsManager

public *mixed* **fireQueue** (*SplPriorityQueue* $queue, *Phalcon\Events\Event* $event)

Internal handler to call a queue of events

public *mixed* **fire** (*string* $eventType, *object* $source, [*mixed* $data], [*int* $cancelable])

Fires an event in the events manager causing that active listeners be notified about it

```
<?php

$eventsManager->fire('db', $connection);
```

public *boolean* **hasListeners** (*string* $type)

Check whether certain type of event has listeners

public *array* **getListeners** (*string* $type)

Returns all the attached listeners of a certain type

2.48.69 Class Phalcon\Exception

extends Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.70 Class Phalcon\Filter

implements Phalcon\FilterInterface

The Phalcon\Filter component provides a set of commonly needed data filters. It provides object oriented wrappers to the php filter extension. Also allows the developer to define his/her own filters

```
<?php

$filter = new Phalcon\Filter();
$filter->sanitize("some(one)@exa\\mple.com", "email"); // returns "someone@example.com"
$filter->sanitize("hello<<", "string"); // returns "hello"
$filter->sanitize("!100a019", "int"); // returns "100019"
$filter->sanitize("!100a019.01a", "float"); // returns "100019.01"
```

Methods

public **__construct** ()

Phalcon\Filter constructor

public *Phalcon\Filter* **add** (*string* $name, *callable* $handler)

Adds a user-defined filter

public *mixed* **sanitize** (*mixed* $value, *mixed* $filters)

Sanitizes a value with a specified single or set of filters

protected *mixed* **_sanitize** ()

Internal sanitize wrapper to filter_var

public *object[]* **getFilters** ()

Return the user-defined filters in the instance

2.48.71 Class Phalcon\Filter\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.72 Class Phalcon\Flash

Shows HTML notifications related to different circumstances. Classes can be stylized using CSS

```
<?php

 $flash->success("The record was successfully deleted");
 $flash->error("Cannot open the file");
```

Methods

public **__construct** ([*array* $cssClasses])

Phalcon\Flash constructor

public *Phalcon\FlashInterface* **setImplicitFlush** (*boolean* $implicitFlush)

Set the if the output must be implictly flushed to the output or returned as string

public *Phalcon\FlashInterface* **setAutomaticHtml** (*boolean* $automaticHtml)

Set the if the output must be implictly formatted with HTML

public *Phalcon\FlashInterface* **setCssClasses** (*array* $cssClasses)

Set an array with CSS classes to format the messages

public *string* **error** (*string* $message)

Shows a HTML error message

```
<?php

 $flash->error('This is an error');
```

public *string* **notice** (*string* $message)

Shows a HTML notice/information message

```
<?php

 $flash->notice('This is an information');
```

public *string* **success** (*string* $message)

Shows a HTML success message

```
<?php

 $flash->success('The process was finished successfully');
```

public *string* **warning** (*string* $message)

Shows a HTML warning message

```
<?php

 $flash->warning('Hey, this is important');
```

public **outputMessage** (*string* $type, *string* $message)

Outputs a message formatting it with HTML

```
<?php

 $flash->outputMessage('error', $message);
```

2.48.73 Class Phalcon\Flash\Direct

extends Phalcon\Flash

implements Phalcon\FlashInterface

This is a variant of the Phalcon\Flash that inmediately outputs any message passed to it

Methods

public *string* **message** (*string* $type, *string* $message)

Outputs a message

public **__construct** ([*array* $cssClasses]) inherited from Phalcon\Flash

Phalcon\Flash constructor

public *Phalcon\FlashInterface* **setImplicitFlush** (*boolean* $implicitFlush) inherited from Phalcon\Flash

Set the if the output must be implictly flushed to the output or returned as string

public *Phalcon\FlashInterface* **setAutomaticHtml** (*boolean* $automaticHtml) inherited from Phalcon\Flash

Set the if the output must be implictly formatted with HTML

public *Phalcon\FlashInterface* **setCssClasses** (*array* $cssClasses) inherited from Phalcon\Flash

Set an array with CSS classes to format the messages

public *string* **error** (*string* $message) inherited from Phalcon\Flash

Shows a HTML error message

```
<?php

 $flash->error('This is an error');
```

public *string* **notice** (*string* $message) inherited from Phalcon\Flash

Shows a HTML notice/information message

```
<?php

 $flash->notice('This is an information');
```

public *string* **success** (*string* $message) inherited from Phalcon\Flash

Shows a HTML success message

```
<?php

 $flash->success('The process was finished successfully');
```

public *string* **warning** (*string* $message) inherited from Phalcon\Flash

Shows a HTML warning message

```
<?php

 $flash->warning('Hey, this is important');
```

public **outputMessage** (*string* $type, *string* $message) inherited from Phalcon\Flash

Outputs a message formatting it with HTML

```
<?php

 $flash->outputMessage('error', $message);
```

2.48.74 Class Phalcon\Flash\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.75 Class Phalcon\Flash\Session

extends Phalcon\Flash

implements Phalcon\FlashInterface, *Phalcon\DINjectionAwareInterface*

Temporarily stores the messages in session, then messages can be printed in the next request

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

protected *array* **_getSessionMessages** ()

Returns the messages stored in session

protected **_setSessionMessages** ()

Stores the messages in session

public **message** (*string* $type, *string* $message)

Adds a message to the session flasher

public *array* **getMessages** ([*string* $type], [*boolean* $remove])

Returns the messages in the session flasher

public **output** ([*boolean* $remove])

Prints the messages in the session flasher

public **__construct** ([*array* $cssClasses]) inherited from Phalcon\Flash

Phalcon\Flash constructor

public *Phalcon\FlashInterface* **setImplicitFlush** (*boolean* $implicitFlush) inherited from Phalcon\Flash

Set the if the output must be implictly flushed to the output or returned as string

public *Phalcon\FlashInterface* **setAutomaticHtml** (*boolean* $automaticHtml) inherited from Phalcon\Flash

Set the if the output must be implictly formatted with HTML

public *Phalcon\FlashInterface* **setCssClasses** (*array* $cssClasses) inherited from Phalcon\Flash

Set an array with CSS classes to format the messages

public *string* **error** (*string* $message) inherited from Phalcon\Flash

Shows a HTML error message

```
<?php

 $flash->error('This is an error');
```

public *string* **notice** (*string* $message) inherited from Phalcon\Flash

Shows a HTML notice/information message

```
<?php

 $flash->notice('This is an information');
```

public *string* **success** (*string* $message) inherited from Phalcon\Flash

Shows a HTML success message

```
<?php

 $flash->success('The process was finished successfully');
```

public *string* **warning** (*string* $message) inherited from Phalcon\Flash

Shows a HTML warning message

```
<?php

 $flash->warning('Hey, this is important');
```

public **outputMessage** (*string* $type, *string* $message) inherited from Phalcon\Flash

Outputs a message formatting it with HTML

```
<?php

 $flash->outputMessage('error', $message);
```

2.48.76 Class Phalcon\Forms\Element

This is a base class for form elements

Methods

public **__construct** (*string* $name, [*array* $attributes])

Phalcon\Forms\Element constructor

public *Phalcon\Forms\ElementInterface* **setForm** (*Phalcon\Forms\Form* $form)

Sets the parent form to the element

public *Phalcon\Forms\ElementInterface* **getForm** ()

Returns the parent form to the element

public *Phalcon\Forms\ElementInterface* **setName** (*string* $name)

Sets the element's name

public *string* **getName** ()

Returns the element's name

public *Phalcon\Forms\ElementInterface* **addValidators** (*unknown* $validators, [*unknown* $merge])

Adds a group of validators

public *Phalcon\Forms\ElementInterface* **addValidator** (*unknown* $validator)

Adds a validator to the element

public *Phalcon\Validation\ValidatorInterface* [] **getValidators** ()

Returns the validators registered for the element

public *array* **prepareAttributes** (*array* $attributes)

Returns an array of attributes for Phalcon\Tag helpers prepared according to the element's parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* $attribute, *mixed* $value)

Sets a default attribute for the element

public *Phalcon\Forms\ElementInterface* **setAttributes** (*array* $attributes)

Sets default attributes for the element

public *array* **getAttributes** ()

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setLabel** (*string* $label)

Sets the element label

public *string* **getLabel** ()

Returns the element's label

public *string* **__toString** ()

Magic method __toString renders the widget without atttributes

2.48.77 Class Phalcon\Forms\Element\Check

extends Phalcon\Forms\Element

Component INPUT[type=check] for forms

Methods

public *string* **render** ([*array* $attributes])

Renders the element widget

public **__construct** (*string* $name, [*array* $attributes]) inherited from Phalcon\Forms\Element

Phalcon\Forms\Element constructor

public *Phalcon\Forms\ElementInterface* **setForm** (*Phalcon\Forms\Form* $form) inherited from Phalcon\Forms\Element

Sets the parent form to the element

public *Phalcon\Forms\ElementInterface* **getForm** () inherited from Phalcon\Forms\Element

Returns the parent form to the element

public *Phalcon\Forms\ElementInterface* **setName** (*string* $name) inherited from Phalcon\Forms\Element

Sets the element's name

public *string* **getName** () inherited from Phalcon\Forms\Element

Returns the element's name

public *Phalcon\Forms\ElementInterface* **addValidators** (*unknown* $validators, [*unknown* $merge]) inherited from Phalcon\Forms\Element

Adds a group of validators

public *Phalcon\Forms\ElementInterface* **addValidator** (*unknown* $validator) inherited from Phalcon\Forms\Element

Adds a validator to the element

public *Phalcon\Validation\ValidatorInterface* [] **getValidators** () inherited from Phalcon\Forms\Element

Returns the validators registered for the element

public *array* **prepareAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Returns an array of attributes for Phalcon\Tag helpers prepared according to the element's parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* $attribute, *mixed* $value) inherited from Phalcon\Forms\Element

Sets a default attribute for the element

public *Phalcon\Forms\ElementInterface* **setAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Sets default attributes for the element

public *array* **getAttributes** () inherited from Phalcon\Forms\Element

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setLabel** (*string* $label) inherited from Phalcon\Forms\Element

Sets the element label

public *string* **getLabel** () inherited from Phalcon\Forms\Element

Returns the element's label

public *string* **__toString** () inherited from Phalcon\Forms\Element

Magic method __toString renders the widget without atttributes

2.48.78 Class Phalcon\Forms\Element\File

extends Phalcon\Forms\Element

Component INPUT[type=file] for forms

Methods

public *string* **render** ([*array* $attributes])

Renders the element widget

public **__construct** (*string* $name, [*array* $attributes]) inherited from Phalcon\Forms\Element

Phalcon\Forms\Element constructor

public *Phalcon\Forms\ElementInterface* **setForm** (*Phalcon\Forms\Form* $form) inherited from Phalcon\Forms\Element

Sets the parent form to the element

public *Phalcon\Forms\ElementInterface* **getForm** () inherited from Phalcon\Forms\Element

Returns the parent form to the element

public *Phalcon\Forms\ElementInterface* **setName** (*string* $name) inherited from Phalcon\Forms\Element

Sets the element's name

public *string* **getName** () inherited from Phalcon\Forms\Element

Returns the element's name

public *Phalcon\Forms\ElementInterface* **addValidators** (*unknown* $validators, [*unknown* $merge]) inherited from Phalcon\Forms\Element

Adds a group of validators

public *Phalcon\Forms\ElementInterface* **addValidator** (*unknown* $validator) inherited from Phalcon\Forms\Element

Adds a validator to the element

public *Phalcon\Validation\ValidatorInterface* [] **getValidators** () inherited from Phalcon\Forms\Element

Returns the validators registered for the element

public *array* **prepareAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Returns an array of attributes for Phalcon\Tag helpers prepared according to the element's parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* $attribute, *mixed* $value) inherited from Phalcon\Forms\Element

Sets a default attribute for the element

public *Phalcon\Forms\ElementInterface* **setAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Sets default attributes for the element

public *array* **getAttributes** () inherited from Phalcon\Forms\Element

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setLabel** (*string* $label) inherited from Phalcon\Forms\Element

Sets the element label

public *string* **getLabel** () inherited from Phalcon\Forms\Element

Returns the element's label

public *string* **__toString** () inherited from Phalcon\Forms\Element

Magic method __toString renders the widget without atttributes

2.48.79 Class Phalcon\Forms\Element\Hidden

extends Phalcon\Forms\Element

Component INPUT[type=hidden] for forms

Methods

public *string* **render** ([*array* $attributes])

Renders the element widget

public **__construct** (*string* $name, [*array* $attributes]) inherited from Phalcon\Forms\Element

Phalcon\Forms\Element constructor

public *Phalcon\Forms\ElementInterface* **setForm** (*Phalcon\Forms\Form* $form) inherited from Phalcon\Forms\Element

Sets the parent form to the element

public *Phalcon\Forms\ElementInterface* **getForm** () inherited from Phalcon\Forms\Element

Returns the parent form to the element

public *Phalcon\Forms\ElementInterface* **setName** (*string* $name) inherited from Phalcon\Forms\Element

Sets the element's name

public *string* **getName** () inherited from Phalcon\Forms\Element

Returns the element's name

public *Phalcon\Forms\ElementInterface* **addValidators** (*unknown* $validators, [*unknown* $merge]) inherited from Phalcon\Forms\Element

Adds a group of validators

public *Phalcon\Forms\ElementInterface* **addValidator** (*unknown* $validator) inherited from Phalcon\Forms\Element

Adds a validator to the element

public *Phalcon\Validation\ValidatorInterface* [] **getValidators** () inherited from Phalcon\Forms\Element

Returns the validators registered for the element

public *array* **prepareAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Returns an array of attributes for Phalcon\Tag helpers prepared according to the element's parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* $attribute, *mixed* $value) inherited from Phalcon\Forms\Element

Sets a default attribute for the element

public *Phalcon\Forms\ElementInterface* **setAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Sets default attributes for the element

public *array* **getAttributes** () inherited from Phalcon\Forms\Element

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setLabel** (*string* $label) inherited from Phalcon\Forms\Element

Sets the element label

public *string* **getLabel** () inherited from Phalcon\Forms\Element

Returns the element's label

public *string* **__toString** () inherited from Phalcon\Forms\Element

Magic method __toString renders the widget without atttributes

2.48.80 Class Phalcon\Forms\Element\Password

extends Phalcon\Forms\Element

Component INPUT[type=password] for forms

Methods

public *string* **render** ([*array* $attributes])

Renders the element widget

public **__construct** (*string* $name, [*array* $attributes]) inherited from Phalcon\Forms\Element

Phalcon\Forms\Element constructor

public *Phalcon\Forms\ElementInterface* **setForm** (*Phalcon\Forms\Form* $form) inherited from Phalcon\Forms\Element

Sets the parent form to the element

public *Phalcon\Forms\ElementInterface* **getForm** () inherited from Phalcon\Forms\Element

Returns the parent form to the element

public *Phalcon\Forms\ElementInterface* **setName** (*string* $name) inherited from Phalcon\Forms\Element

Sets the element's name

public *string* **getName** () inherited from Phalcon\Forms\Element

Returns the element's name

public *Phalcon\Forms\ElementInterface* **addValidators** (*unknown* $validators, [*unknown* $merge]) inherited from Phalcon\Forms\Element

Adds a group of validators

public *Phalcon\Forms\ElementInterface* **addValidator** (*unknown* $validator) inherited from Phalcon\Forms\Element

Adds a validator to the element

public *Phalcon\Validation\ValidatorInterface* [] **getValidators** () inherited from Phalcon\Forms\Element

Returns the validators registered for the element

public *array* **prepareAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Returns an array of attributes for Phalcon\Tag helpers prepared according to the element's parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* $attribute, *mixed* $value) inherited from Phalcon\Forms\Element

Sets a default attribute for the element

public *Phalcon\Forms\ElementInterface* **setAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Sets default attributes for the element

public *array* **getAttributes** () inherited from Phalcon\Forms\Element

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setLabel** (*string* $label) inherited from Phalcon\Forms\Element

Sets the element label

public *string* **getLabel** () inherited from Phalcon\Forms\Element

Returns the element's label

public *string* **__toString** () inherited from Phalcon\Forms\Element

Magic method __toString renders the widget without atttributes

2.48.81 Class Phalcon\Forms\Element\Select

extends Phalcon\Forms\Element

Component SELECT (choice) for forms

Methods

public **__construct** (*string* $name, [*object|array* $options], [*array* $attributes])

Phalcon\Forms\Element constructor

public *Phalcon\Forms\Element* **setOptions** (*array|object* $options)

Set the choice's options

public *array|object* **getOptions** ()

Returns the choices' options

public *$this;* **addOption** (*array* $option)

Adds an option to the current options

public *string* **render** ([*array* $attributes])

Renders the element widget

public *Phalcon\Forms\ElementInterface* **setForm** (*Phalcon\Forms\Form* $form) inherited from Phalcon\Forms\Element

Sets the parent form to the element

public *Phalcon\Forms\ElementInterface* **getForm** () inherited from Phalcon\Forms\Element

Returns the parent form to the element

public *Phalcon\Forms\ElementInterface* **setName** (*string* $name) inherited from Phalcon\Forms\Element

Sets the element's name

public *string* **getName** () inherited from Phalcon\Forms\Element

Returns the element's name

public *Phalcon\Forms\ElementInterface* **addValidators** (*unknown* $validators, [*unknown* $merge]) inherited from Phalcon\Forms\Element

Adds a group of validators

public *Phalcon\Forms\ElementInterface* **addValidator** (*unknown* $validator) inherited from Phalcon\Forms\Element

Adds a validator to the element

public *Phalcon\Validation\ValidatorInterface* [] **getValidators** () inherited from Phalcon\Forms\Element

Returns the validators registered for the element

public *array* **prepareAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Returns an array of attributes for Phalcon\Tag helpers prepared according to the element's parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* $attribute, *mixed* $value) inherited from Phalcon\Forms\Element

Sets a default attribute for the element

public *Phalcon\Forms\ElementInterface* **setAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Sets default attributes for the element

public *array* **getAttributes** () inherited from Phalcon\Forms\Element

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setLabel** (*string* $label) inherited from Phalcon\Forms\Element

Sets the element label

public *string* **getLabel** () inherited from Phalcon\Forms\Element

Returns the element's label

public *string* **__toString** () inherited from Phalcon\Forms\Element

Magic method __toString renders the widget without atttributes

2.48.82 Class Phalcon\Forms\Element\Submit

extends Phalcon\Forms\Element

Component INPUT[type=submit] for forms

Methods

public *string* **render** ([*array* $attributes])

Renders the element widget

public **__construct** (*string* $name, [*array* $attributes]) inherited from Phalcon\Forms\Element

Phalcon\Forms\Element constructor

public *Phalcon\Forms\ElementInterface* **setForm** (*Phalcon\Forms\Form* $form) inherited from Phalcon\Forms\Element

Sets the parent form to the element

public *Phalcon\Forms\ElementInterface* **getForm** () inherited from Phalcon\Forms\Element

Returns the parent form to the element

public *Phalcon\Forms\ElementInterface* **setName** (*string* $name) inherited from Phalcon\Forms\Element

Sets the element's name

public *string* **getName** () inherited from Phalcon\Forms\Element

Returns the element's name

public *Phalcon\Forms\ElementInterface* **addValidators** (*unknown* $validators, [*unknown* $merge]) inherited from Phalcon\Forms\Element

Adds a group of validators

public *Phalcon\Forms\ElementInterface* **addValidator** (*unknown* $validator) inherited from Phalcon\Forms\Element

Adds a validator to the element

public *Phalcon\Validation\ValidatorInterface* [] **getValidators** () inherited from Phalcon\Forms\Element

Returns the validators registered for the element

public *array* **prepareAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Returns an array of attributes for Phalcon\Tag helpers prepared according to the element's parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* $attribute, *mixed* $value) inherited from Phalcon\Forms\Element

Sets a default attribute for the element

public *Phalcon\Forms\ElementInterface* **setAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Sets default attributes for the element

public *array* **getAttributes** () inherited from Phalcon\Forms\Element

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setLabel** (*string* $label) inherited from Phalcon\Forms\Element

Sets the element label

public *string* **getLabel** () inherited from Phalcon\Forms\Element

Returns the element's label

public *string* **__toString** () inherited from Phalcon\Forms\Element

Magic method __toString renders the widget without atttributes

2.48.83 Class Phalcon\Forms\Element\Text

extends Phalcon\Forms\Element

Component INPUT[type=text] for forms

Methods

public *string* **render** ([*array* $attributes])

Renders the element widget

public **__construct** (*string* $name, [*array* $attributes]) inherited from Phalcon\Forms\Element

Phalcon\Forms\Element constructor

public *Phalcon\Forms\ElementInterface* **setForm** (*Phalcon\Forms\Form* $form) inherited from Phalcon\Forms\Element

Sets the parent form to the element

public *Phalcon\Forms\ElementInterface* **getForm** () inherited from Phalcon\Forms\Element

Returns the parent form to the element

public *Phalcon\Forms\ElementInterface* **setName** (*string* $name) inherited from Phalcon\Forms\Element

Sets the element's name

public *string* **getName** () inherited from Phalcon\Forms\Element

Returns the element's name

public *Phalcon\Forms\ElementInterface* **addValidators** (*unknown* $validators, [*unknown* $merge]) inherited from Phalcon\Forms\Element

Adds a group of validators

public *Phalcon\Forms\ElementInterface* **addValidator** (*unknown* $validator) inherited from Phalcon\Forms\Element

Adds a validator to the element

public *Phalcon\Validation\ValidatorInterface* [] **getValidators** () inherited from Phalcon\Forms\Element

Returns the validators registered for the element

public *array* **prepareAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Returns an array of attributes for Phalcon\Tag helpers prepared according to the element's parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* $attribute, *mixed* $value) inherited from Phalcon\Forms\Element

Sets a default attribute for the element

public *Phalcon\Forms\ElementInterface* **setAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Sets default attributes for the element

public *array* **getAttributes** () inherited from Phalcon\Forms\Element

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setLabel** (*string* $label) inherited from Phalcon\Forms\Element

Sets the element label

public *string* **getLabel** () inherited from Phalcon\Forms\Element

Returns the element's label

public *string* **__toString** () inherited from Phalcon\Forms\Element

Magic method __toString renders the widget without atttributes

2.48.84 Class Phalcon\Forms\Element\TextArea

extends Phalcon\Forms\Element

Component TEXTAREA for forms

Methods

public *string* **render** ([*array* $attributes])

Renders the element widget

public **__construct** (*string* $name, [*array* $attributes]) inherited from Phalcon\Forms\Element

Phalcon\Forms\Element constructor

public *Phalcon\Forms\ElementInterface* **setForm** (*Phalcon\Forms\Form* $form) inherited from Phalcon\Forms\Element

Sets the parent form to the element

public *Phalcon\Forms\ElementInterface* **getForm** () inherited from Phalcon\Forms\Element

Returns the parent form to the element

public *Phalcon\Forms\ElementInterface* **setName** (*string* $name) inherited from Phalcon\Forms\Element

Sets the element's name

public *string* **getName** () inherited from Phalcon\Forms\Element

Returns the element's name

public *Phalcon\Forms\ElementInterface* **addValidators** (*unknown* $validators, [*unknown* $merge]) inherited from Phalcon\Forms\Element

Adds a group of validators

public *Phalcon\Forms\ElementInterface* **addValidator** (*unknown* $validator) inherited from Phalcon\Forms\Element

Adds a validator to the element

public *Phalcon\Validation\ValidatorInterface* [] **getValidators** () inherited from Phalcon\Forms\Element

Returns the validators registered for the element

public *array* **prepareAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Returns an array of attributes for Phalcon\Tag helpers prepared according to the element's parameters

public *Phalcon\Forms\ElementInterface* **setAttribute** (*string* $attribute, *mixed* $value) inherited from Phalcon\Forms\Element

Sets a default attribute for the element

public *Phalcon\Forms\ElementInterface* **setAttributes** (*array* $attributes) inherited from Phalcon\Forms\Element

Sets default attributes for the element

public *array* **getAttributes** () inherited from Phalcon\Forms\Element

Returns the default attributes for the element

public *Phalcon\Forms\ElementInterface* **setLabel** (*string* $label) inherited from Phalcon\Forms\Element

Sets the element label

public *string* **getLabel** () inherited from Phalcon\Forms\Element

Returns the element's label

public *string* **__toString** () inherited from Phalcon\Forms\Element

Magic method __toString renders the widget without atttributes

2.48.85 Class Phalcon\Forms\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.86 Class Phalcon\Forms\Form

implements Countable, Iterator, Traversable

This component allows to build forms

Methods

public **__construct** ([*object* $entity])

Phalcon\Forms\Form constructor

public *Phalcon\Forms\Form* **setEntity** (*object* $entity)

Sets the entity related to the model

public *object* **getEntity** ()

Returns the entity related to the model

public *Phalcon\Forms\ElementInterface[]* **getElements** ()

Returns the form elements added to the form

public *Phalcon\Forms\Form* **bind** (*array* $data, *object* $entity, [*unknown* $whitelist])

Binds data to the entity

public *boolean* **isValid** ([*array* $data], [*object* $entity])

Validates the form

public *array* **getMessages** ([*boolean* $byItemName])

Returns the messages generated in the validation

public *Phalcon\Validation\Message\Group* [] **getMessagesFor** (*unknown* $name)

Returns the messages generated by

public *Phalcon\Forms\Form* **add** (*Phalcon\Forms\ElementInterface* $element)

Adds an element to the form

public *string* **render** (*string* $name, [*array* $attributes])

Renders an specific item in the form

public *Phalcon\Forms\ElementInterface* **get** (*string* $name)

Returns an element added to the form by its name

public *string* **label** (*string* $name)

Generate the label of a element added to the form including HTML

public *string* **getLabel** (*string* $name)

Returns the label

public *mixed* **getValue** (*string* $name)

Gets a value from the the internal related entity or from the default value

public *int* **count** ()

Returns the number of elements in the form

public **rewind** ()

Rewinds the internal iterator

public *Phalcon\Validation\Message* **current** ()

Returns the current element in the iterator

public *int* **key** ()

Returns the current position/key in the iterator

public **next** ()

Moves the internal iteration pointer to the next position

public *boolean* **valid** ()

Check if the current element in the iterator is valid

2.48.87 Class Phalcon\Forms\Manager

Manages forms whithin the application. Allowing the developer to access them from any part of the application

Methods

public **create** ([*unknown* $name], [*unknown* $entity])

...

public *Phalcon\Forms\Form* **get** ()

Returns a form by its name

2.48.88 Class Phalcon\Http\Cookie

implements Phalcon\DINjectionAwareInterface

Provide OO wrappers to manage a HTTP cookie

Methods

public **__construct** (*string* $name, [*mixed* $value], [*int* $expire], [*string* $path])

Phalcon\Http\Cookie constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

public *Phalcon\Http\CookieInterface* **setValue** (*string* $value)

Sets the cookie's value

public *mixed* **getValue** ([*string|array* $filters], [*string* $defaultValue])

Returns the cookie's value

public *Phalcon\Http\Cookie* **setExpiration** (*int* $expire)

Sets the cookie's expiration time

public *string* **getExpiration** ()

Returns the current expiration time

public *Phalcon\Http\Cookie* **setPath** (*string* $path)

Sets the cookie's expiration time

public *string* **getPath** ()

Returns the current cookie's path

public *Phalcon\Http\Cookie* **setSecure** (*boolean* $secure)

Sets if the cookie must only be sent when the connection is secure (HTTPS)

public *boolean* **getSecure** ()

Returns whether the cookie must only be sent when the connection is secure (HTTPS)

2.48.89 Class Phalcon\Http\Cookie\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.90 Class Phalcon\Http\Request

implements Phalcon\Http\RequestInterface, Phalcon\DI\InjectionAwareInterface

Encapsulates request information for easy and secure access from application controllers. The request object is a simple value object that is passed between the dispatcher and controller classes. It packages the HTTP request environment.

```
<?php

$request = new Phalcon\Http\Request();
if ($request->isPost() == true) {
    if ($request->isAjax() == true) {
            echo 'Request was made using POST and AJAX';
```

```
    }
}
```

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

public *mixed* **get** (*string* $name, [*string|array* $filters], [*mixed* $defaultValue])

Gets a variable from the $_REQUEST superglobal applying filters if needed

```
<?php

//Returns value from $_REQUEST["user_email"] without sanitizing
$userEmail = $request->get("user_email");

//Returns value from $_REQUEST["user_email"] with sanitizing
$userEmail = $request->get("user_email", "email");
```

public *mixed* **getPost** (*string* $name, [*string|array* $filters], [*mixed* $defaultValue])

Gets a variable from the $_POST superglobal applying filters if needed

```
<?php

//Returns value from $_POST["user_email"] without sanitizing
$userEmail = $request->getPost("user_email");

//Returns value from $_POST["user_email"] with sanitizing
$userEmail = $request->getPost("user_email", "email");
```

public *mixed* **getQuery** (*string* $name, [*string|array* $filters], [*mixed* $defaultValue])

Gets variable from $_GET superglobal applying filters if needed

```
<?php

//Returns value from $_GET["id"] without sanitizing
$id = $request->getQuery("id");

//Returns value from $_GET["id"] with sanitizing
$id = $request->getQuery("id", "int");

//Returns value from $_GET["id"] with a default value
$id = $request->getQuery("id", null, 150);
```

public *mixed* **getServer** (*string* $name)

Gets variable from $_SERVER superglobal

public *boolean* **has** (*string* $name)

Checks whether $_SERVER superglobal has certain index

public *boolean* **hasPost** (*string* $name)

Checks whether $_POST superglobal has certain index

public *boolean* **hasQuery** (*string* $name)

Checks whether $_SERVER superglobal has certain index

public *mixed* **hasServer** (*string* $name)

Checks whether $_SERVER superglobal has certain index

public *string* **getHeader** (*string* $header)

Gets HTTP header from request data

public *string* **getScheme** ()

Gets HTTP schema (http/https)

public *boolean* **isAjax** ()

Checks whether request has been made using ajax. Checks if $_SERVER['HTTP_X_REQUESTED_WITH']=='XMLHttpRequest'

public *boolean* **isSoapRequested** ()

Checks whether request has been made using SOAP

public *boolean* **isSecureRequest** ()

Checks whether request has been made using any secure layer

public *string* **getRawBody** ()

Gets HTTP raw request body

public *string* **getServerAddress** ()

Gets active server address IP

public *string* **getServerName** ()

Gets active server name

public *string* **getHttpHost** ()

Gets information about schema, host and port used by the request

public *string* **getClientAddress** ([*boolean* $trustForwardedHeader])

Gets most possible client IPv4 Address. This method search in $_SERVER['REMOTE_ADDR'] and optionally in $_SERVER['HTTP_X_FORWARDED_FOR']

public *string* **getMethod** ()

Gets HTTP method which request has been made

public *string* **getUserAgent** ()

Gets HTTP user agent used to made the request

public *boolean* **isMethod** (*string|array* $methods)

Check if HTTP method match any of the passed methods

public *boolean* **isPost** ()

Checks whether HTTP method is POST. if $_SERVER['REQUEST_METHOD']=='POST'

public *boolean* **isGet** ()

Checks whether HTTP method is GET. if $_SERVER['REQUEST_METHOD']=='GET'

public *boolean* **isPut** ()

Checks whether HTTP method is PUT. if $_SERVER['REQUEST_METHOD']=='PUT'

public *boolean* **isPatch** ()

Checks whether HTTP method is PATCH. if $_SERVER['REQUEST_METHOD']=='PATCH'

public *boolean* **isHead** ()

Checks whether HTTP method is HEAD. if $_SERVER['REQUEST_METHOD']=='HEAD'

public *boolean* **isDelete** ()

Checks whether HTTP method is DELETE. if $_SERVER['REQUEST_METHOD']=='DELETE'

public *boolean* **isOptions** ()

Checks whether HTTP method is OPTIONS. if $_SERVER['REQUEST_METHOD']=='OPTIONS'

public *boolean* **hasFiles** ()

Checks whether request include attached files

public *Phalcon\Http\Request\File* [] **getUploadedFiles** ()

Gets attached files as Phalcon\Http\Request\File instances

public *string* **getHTTPReferer** ()

Gets web page that refers active request. ie: http://www.google.com

protected *array* **_getQualityHeader** ()

Process a request header and return an array of values with their qualities

protected *string* **_getBestQuality** ()

Process a request header and return the one with best quality

public *array* **getAcceptableContent** ()

Gets array with mime/types and their quality accepted by the browser/client from $_SERVER['HTTP_ACCEPT']

public *array* **getBestAccept** ()

Gets best mime/type accepted by the browser/client from $_SERVER['HTTP_ACCEPT']

public *array* **getClientCharsets** ()

Gets charsets array and their quality accepted by the browser/client from $_SERVER['HTTP_ACCEPT_CHARSET']

public *string* **getBestCharset** ()

Gets best charset accepted by the browser/client from $_SERVER['HTTP_ACCEPT_CHARSET']

public *array* **getLanguages** ()

Gets languages array and their quality accepted by the browser/client from $_SERVER['HTTP_ACCEPT_LANGUAGE']

public *string* **getBestLanguage** ()

Gets best language accepted by the browser/client from $_SERVER['HTTP_ACCEPT_LANGUAGE']

2.48.91 Class Phalcon\Http\Request\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.92 Class Phalcon\Http\Request\File

implements Phalcon\Http\Request\FileInterface

Provides OO wrappers to the $_FILES superglobal

```
<?php

class PostsController extends \Phalcon\Mvc\Controller
{

    public function uploadAction()
    {
            //Check if the user has uploaded files
            if ($this->request->hasFiles() == true) {
                    //Print the real file names and their sizes
                    foreach ($this->request->getUploadedFiles() as $file){
                            echo $file->getName(), " ", $file->getSize(), "\n";
                    }
            }
    }

}
```

Methods

public **__construct** (*array* $file)

Phalcon\Http\Request\File constructor

public *int* **getSize** ()

Returns the file size of the uploaded file

public *string* **getName** ()

Returns the real name of the uploaded file

public *string* **getTempName** ()

Returns the temporal name of the uploaded file

public *boolean* **moveTo** (*string* $destination)

Move the temporary file to a destination whithin the application

2.48.93 Class Phalcon\Http\Response

implements Phalcon\Http\ResponseInterface, *Phalcon\DI\InjectionAwareInterface*

Part of the HTTP cycle is return responses to the clients. Phalcon\HTTP\Response is the Phalcon component responsible to achieve this task. HTTP responses are usually composed by headers and body.

```
<?php

$response = new Phalcon\Http\Response();
$response->setStatusCode(200, "OK");
$response->setContent("<html><body>Hello</body></html>");
$response->send();
```

Methods

public **__construct** ([*string* $content], [*int* $code], [*string* $status])

Phalcon\Http\Response constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

public *Phalcon\Http\ResponseInterface* **setStatusCode** (*int* $code, *string* $message)

Sets the HTTP response code

```
<?php

$response->setStatusCode(404, "Not Found");
```

public *Phalcon\Http\ResponseInterface* **setHeaders** (*Phalcon\Http\Response\HeadersInterface* $headers)

Sets a headers bag for the response externally

public *Phalcon\Http\Response\HeadersInterface* **getHeaders** ()

Returns headers set by the user

public *Phalcon\Http\ResponseInterface* **setCookies** (*Phalcon\Http\Response\CookiesInterface* $cookies)

Sets a cookies bag for the response externally

public *Phalcon\Http\Response\CookiesInterface* **getCookies** ()

Returns coookies set by the user

public *Phalcon\Http\ResponseInterface* **setHeader** (*string* $name, *string* $value)

Overwrites a header in the response

```
<?php

$response->setHeader("Content-Type", "text/plain");
```

public *Phalcon\Http\ResponseInterface* **setRawHeader** (*string* $header)

Send a raw header to the response

```
<?php

$response->setRawHeader("HTTP/1.1 404 Not Found");
```

public *Phalcon\Http\ResponseInterface* **resetHeaders** ()

Resets all the stablished headers

public *Phalcon\Http\ResponseInterface* **setExpires** (*DateTime* $datetime)

Sets a Expires header to use HTTP cache

```
<?php

$this->response->setExpires(new DateTime());
```

public *Phalcon\Http\ResponseInterface* **setNotModified** ()

Sends a Not-Modified response

public *Phalcon\Http\ResponseInterface* **setContentType** (*string* $contentType, [*string* $charset])

Sets the response content-type mime, optionally the charset

```
<?php

$response->setContentType('application/pdf');
$response->setContentType('text/plain', 'UTF-8');
```

public *Phalcon\Http\ResponseInterface* **redirect** ([*string* $location], [*boolean* $externalRedirect], [*int* $statusCode])

Redirect by HTTP to another action or URL

```
<?php

  //Using a string redirect (internal/external)
$response->redirect("posts/index");
$response->redirect("http://en.wikipedia.org", true);
$response->redirect("http://www.example.com/new-location", true, 301);
```

public *Phalcon\Http\ResponseInterface* **setContent** (*string* $content)

Sets HTTP response body

```
<?php

$response->setContent("<h1>Hello!</h1>");
```

public *Phalcon\Http\ResponseInterface* **appendContent** (*string* $content)

Appends a string to the HTTP response body

public *string* **getContent** ()

Gets the HTTP response body

public *boolean* **isSent** ()

Check if the response is already sent

public *Phalcon\Http\ResponseInterface* **sendHeaders** ()

Sends headers to the client

public *Phalcon\Http\ResponseInterface* **send** ()

Prints out HTTP response to the client

2.48.94 Class Phalcon\Http\Response\Cookies

implements Phalcon\DINjectionAwareInterface

This class is a bag to manage the cookies

Methods

public **__construct** ()

Phalcon\Http\Response\Cookies constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

public **set** (*string* $name, [*mixed* $value], [*int* $expire], [*string* $path])

Sets a header to be sent at the end of the request

public *Phalcon\Http\Cookie* **get** (*string* $name)

Gets a cookie from the bag

public **reset** ()

Reset set cookies

2.48.95 Class Phalcon\Http\Response\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.96 Class Phalcon\Http\Response\Headers

implements Phalcon\Http\Response\HeadersInterface

This class is a bag to manage the response headers

Methods

public **__construct** ()

Phalcon\Http\Response\Headers constructor

public **set** (*string* $name, *string* $value)

Sets a header to be sent at the end of the request

public *string* **get** (*string* $name)

Gets a header value from the internal bag

public **setRaw** (*string* $header)

Sets a raw header to be sent at the end of the request

public *boolean* **send** ()

Sends the headers to the client

public **reset** ()

Reset set headers

public static *Phalcon\Http\Response\Headers* **__set_state** (*array* $data)

Restore a Phalcon\Http\Response\Headers object

2.48.97 Class Phalcon\Kernel

Methods

public static **preComputeHashKey** (*unknown* $arrKey)

...

2.48.98 Class Phalcon\Loader

implements Phalcon\Events\EventsAwareInterface

This component helps to load your project classes automatically based on some conventions

```
<?php

 //Creates the autoloader
 $loader = new Phalcon\Loader();

 //Register some namespaces
 $loader->registerNamespaces(array(
   'Example\Base' => 'vendor/example/base/',
   'Example\Adapter' => 'vendor/example/adapter/',
   'Example' => 'vendor/example/'
 ));

 //register autoloader
 $loader->register();

 //Requiring this class will automatically include file vendor/example/adapter/Some.php
 $adapter = Example\Adapter\Some();
```

Methods

public **__construct** ()

Phalcon\Loader constructor

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager)

Sets the events manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the internal event manager

public *Phalcon\Loader* **setExtensions** (*array* $extensions)

Sets an array of extensions that the Loader must check together with the path

public *boolean* **getExtensions** ()

Return file extensions registered in the loader

public *Phalcon\Loader* **registerNamespaces** (*array* $namespaces, [*boolean* $merge])

Register namespaces and their related directories

public **getNamespaces** ()

Return current namespaces registered in the autoloader

public *Phalcon\Loader* **registerPrefixes** (*array* $prefixes, [*boolean* $merge])

Register directories on which "not found" classes could be found

public **getPrefixes** ()

Return current prefixes registered in the autoloader

public *Phalcon\Loader* **registerDirs** (*array* $directories, [*boolean* $merge])

Register directories on which "not found" classes could be found

public **getDirs** ()

Return current directories registered in the autoloader

public *Phalcon\Loader* **registerClasses** (*array* $classes, [*boolean* $merge])

Register classes and their locations

public **getClasses** ()

Return the current class-map registered in the autoloader

public *Phalcon\Loader* **register** ()

Register the autoload method

public *Phalcon\Loader* **unregister** ()

Unregister the autoload method

public *boolean* **autoLoad** (*string* $className)

Makes the work of autoload registered classes

public *string* **getFoundPath** ()

Get the path when a class was found

public *string* **getCheckedPath** ()

Get the path the loader is checking for a path

2.48.99 Class Phalcon\Loader\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.100 Class Phalcon\Logger

Constants

integer **SPECIAL**

integer **CUSTOM**

integer **DEBUG**

integer **INFO**

integer **NOTICE**

integer **WARNING**

integer **ERROR**

integer **ALERT**

integer **CRITICAL**

integer **EMERGENCE**

2.48.101 Class Phalcon\Logger\Adapter

Base class for Phalcon\Logger adapters

Methods

public **setLogLevel** (*int* $level)

Filters the logs sent to the handlers to be less or equals than a specific level

public **getLogLevel** ()

Returns the current log level

public **setFormatter** (*Phalcon\Logger\FormatterInterface* $formatter)

Sets the message formatter

public **begin** ()

Starts a transaction

public **commit** ()

Commits the internal transaction

public **rollback** ()

Rollbacks the internal transaction

public **emergence** (*string* $message)

Sends/Writes an emergence message to the log

public **debug** (*string* $message)

Sends/Writes a debug message to the log

public **error** (*string* $message)

Sends/Writes an error message to the log

public **info** (*string* $message)

Sends/Writes an info message to the log

public **notice** (*string* $message)

Sends/Writes a notice message to the log

public **warning** (*string* $message)

Sends/Writes a warning message to the log

public **alert** (*string* $message)

Sends/Writes an alert message to the log

public **log** (*string* $message, [*int* $type])

Logs messages to the internal loggger. Appends logs to the

2.48.102 Class Phalcon\Logger\Adapter\File

extends Phalcon\Logger\Adapter

implements Phalcon\Logger\AdapterInterface

Adapter to store logs in plain text files

```
<?php

$logger = new \Phalcon\Logger\Adapter\File("app/logs/test.log");
$logger->log("This is a message");
$logger->log("This is an error", \Phalcon\Logger::ERROR);
$logger->error("This is another error");
$logger->close();
```

Methods

public **__construct** (*string* $name, [*array* $options])

Phalcon\Logger\Adapter\File constructor

public *Phalcon\Logger\Formatter\Line* **getFormatter** ()

Returns the internal formatter

public **logInternal** (*string* $message, *int* $type, *int* $time)

Writes the log to the file itself

public *boolean* **close** ()

Closes the logger

public **__wakeup** ()

Opens the internal file handler after unserialization

public **setLogLevel** (*int* $level) inherited from Phalcon\Logger\Adapter

Filters the logs sent to the handlers to be less or equals than a specific level

public **getLogLevel** () inherited from Phalcon\Logger\Adapter

Returns the current log level

public **setFormatter** (*Phalcon\Logger\FormatterInterface* $formatter) inherited from Phalcon\Logger\Adapter

Sets the message formatter

public **begin** () inherited from Phalcon\Logger\Adapter

Starts a transaction

public **commit** () inherited from Phalcon\Logger\Adapter

Commits the internal transaction

public **rollback** () inherited from Phalcon\Logger\Adapter

Rollbacks the internal transaction

public **emergence** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an emergence message to the log

public **debug** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes a debug message to the log

public **error** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an error message to the log

public **info** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an info message to the log

public **notice** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes a notice message to the log

public **warning** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes a warning message to the log

public **alert** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an alert message to the log

public **log** (*string* $message, [*int* $type]) inherited from Phalcon\Logger\Adapter

Logs messages to the internal loggger. Appends logs to the

2.48.103 Class Phalcon\Logger\Adapter\Stream

extends Phalcon\Logger\Adapter

implements Phalcon\Logger\AdapterInterface

Sends logs to a valid PHP stream

```
<?php

$logger = new \Phalcon\Logger\Adapter\Stream("php://stderr");
$logger->log("This is a message");
$logger->log("This is an error", \Phalcon\Logger::ERROR);
$logger->error("This is another error");
```

Methods

public **__construct** (*string* $name, [*array* $options])

Phalcon\Logger\Adapter\Stream constructor

public *Phalcon\Logger\Formatter\Line* **getFormatter** ()

Returns the internal formatter

public **logInternal** (*string* $message, *int* $type, *int* $time)

Writes the log to the stream itself

public *boolean* **close** ()

Closes the logger

public **setLogLevel** (*int* $level) inherited from Phalcon\Logger\Adapter

Filters the logs sent to the handlers to be less or equals than a specific level

public **getLogLevel** () inherited from Phalcon\Logger\Adapter

Returns the current log level

public **setFormatter** (*Phalcon\Logger\FormatterInterface* $formatter) inherited from Phalcon\Logger\Adapter

Sets the message formatter

public **begin** () inherited from Phalcon\Logger\Adapter

Starts a transaction

public **commit** () inherited from Phalcon\Logger\Adapter

Commits the internal transaction

public **rollback** () inherited from Phalcon\Logger\Adapter

Rollbacks the internal transaction

public **emergence** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an emergence message to the log

public **debug** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes a debug message to the log

public **error** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an error message to the log

public **info** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an info message to the log

public **notice** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes a notice message to the log

public **warning** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes a warning message to the log

public **alert** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an alert message to the log

public **log** (*string* $message, [*int* $type]) inherited from Phalcon\Logger\Adapter

Logs messages to the internal loggger. Appends logs to the

2.48.104 Class Phalcon\Logger\Adapter\Syslog

extends Phalcon\Logger\Adapter

implements Phalcon\Logger\AdapterInterface

Sends logs to the system logger

```
<?php

$logger = new \Phalcon\Logger\Adapter\Syslog("ident", array(
    'option' => LOG_NDELAY,
    'facility' => LOG_MAIL
));
$logger->log("This is a message");
$logger->log("This is an error", \Phalcon\Logger::ERROR);
$logger->error("This is another error");
```

Methods

public **__construct** (*string* $name, [*array* $options])

Phalcon\Logger\Adapter\Syslog constructor

public *Phalcon\Logger\Formatter\Line* **getFormatter** ()

Returns the internal formatter

public **logInternal** (*string* $message, *int* $type, *int* $time)

Writes the log to the stream itself

public *boolean* **close** ()

Closes the logger

public **setLogLevel** (*int* $level) inherited from Phalcon\Logger\Adapter

Filters the logs sent to the handlers to be less or equals than a specific level

public **getLogLevel** () inherited from Phalcon\Logger\Adapter

Returns the current log level

public **setFormatter** (*Phalcon\Logger\FormatterInterface* $formatter) inherited from Phalcon\Logger\Adapter

Sets the message formatter

public **begin** () inherited from Phalcon\Logger\Adapter

Starts a transaction

public **commit** () inherited from Phalcon\Logger\Adapter

Commits the internal transaction

public **rollback** () inherited from Phalcon\Logger\Adapter

Rollbacks the internal transaction

public **emergence** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an emergence message to the log

public **debug** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes a debug message to the log

public **error** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an error message to the log

public **info** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an info message to the log

public **notice** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes a notice message to the log

public **warning** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes a warning message to the log

public **alert** (*string* $message) inherited from Phalcon\Logger\Adapter

Sends/Writes an alert message to the log

public **log** (*string* $message, [*int* $type]) inherited from Phalcon\Logger\Adapter

Logs messages to the internal loggger. Appends logs to the

2.48.105 Class Phalcon\Logger\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.106 Class Phalcon\Logger\Formatter

This is a base class for logger formatters

Methods

public *string* **getTypeString** (*integer* $type)

Returns the string meaning of a logger constant

2.48.107 Class Phalcon\Logger\Formatter\Json

extends Phalcon\Logger\Formatter

implements Phalcon\Logger\FormatterInterface

Formats messages using JSON format

Methods

public **format** (*string* $message, *int* $type, *int* $timestamp)

Applies a format to a message before sent it to the internal log

public *string* **getTypeString** (*integer* $type) inherited from Phalcon\Logger\Formatter

Returns the string meaning of a logger constant

2.48.108 Class Phalcon\Logger\Formatter\Line

extends Phalcon\Logger\Formatter

implements Phalcon\Logger\FormatterInterface

Formats messages using a one-line string

Methods

public **__construct** ([*string* $format], [*string* $dateFormat])

Phalcon\Logger\Formatter\Line construct

public **setFormat** (*string* $format)

Set the log format

public *format* **getFormat** ()

Returns the log format

public **setDateFormat** (*string* $date)

Sets the internal date format

public *string* **getDateFormat** ()

Returns the internal date format

public **format** (*string* $message, *int* $type, *int* $timestamp)

Applies a format to a message before sent it to the internal log

public *string* **getTypeString** (*integer* $type) inherited from Phalcon\Logger\Formatter

Returns the string meaning of a logger constant

2.48.109 Class Phalcon\Logger\Formatter\Syslog

extends Phalcon\Logger\Formatter

implements Phalcon\Logger\FormatterInterface

Prepares a message to be used in a Syslog backend

Methods

public **format** (*string* $message, *int* $type, *int* $timestamp)

Applies a format to a message before sent it to the internal log

public *string* **getTypeString** (*integer* $type) inherited from Phalcon\Logger\Formatter

Returns the string meaning of a logger constant

2.48.110 Class Phalcon\Logger\Item

Represents each item in a logging transaction

Methods

public **__construct** (*string* $message, *integer* $type, [*integer* $time])

Phalcon\Logger\Item constructor

public *string* **getMessage** ()

Returns the message

public *integer* **getType** ()

Returns the log type

public *integer* **getTime** ()

Returns log timestamp

2.48.111 Class Phalcon\Logger\Multiple

Handles multiples logger handlers

Methods

public **push** (*Phalcon\Logger\AdapterInterface* $logger)

Pushes a logger to the logger tail

public *Phalcon\Logger\AdapterInterface* [] **getLoggers** ()

Returns the registered loggers

public **setFormatter** (*Phalcon\Logger\FormatterInterface* $formatter)

Sets a global formatter

public *Phalcon\Logger\FormatterInterface* **getFormatter** ()

Returns a formatter

public **log** (*string* $message, [*int* $type])

Sends a message to each registered logger

public **emergence** (*string* $message)

Sends/Writes an emergence message to the log

public **debug** (*string* $message)

Sends/Writes a debug message to the log

public **error** (*string* $message)

Sends/Writes an error message to the log

public **info** (*string* $message)

Sends/Writes an info message to the log

public **notice** (*string* $message)

Sends/Writes a notice message to the log

public **warning** (*string* $message)

Sends/Writes a warning message to the log

public **alert** (*string* $message)

Sends/Writes an alert message to the log

2.48.112 Class Phalcon\Mvc\Application

extends Phalcon\DINjectable

implements Phalcon\Events\EventsAwareInterface, *Phalcon\DINjectionAwareInterface*

This component encapsulates all the complex operations behind instantiating every component needed and integrating it with the rest to allow the MVC pattern to operate as desired.

```
<?php

 class Application extends \Phalcon\Mvc\Application
 {

    /**
     * Register the services here to make them general or register
     * in the ModuleDefinition to make them module-specific
     */
    protected function _registerServices()
    {

    }

    /**
     * This method registers all the modules in the application
     */
    public function main()
    {
            $this->registerModules(array(
                    'frontend' => array(
```

```
                        'className' => 'Multiple\Frontend\Module',
                        'path' => '../apps/frontend/Module.php'
                ),
                'backend' => array(
                        'className' => 'Multiple\Backend\Module',
                        'path' => '../apps/backend/Module.php'
                )
        ));
    }
}

$application = new Application();
$application->main();
```

Methods

public **registerModules** (*array* $modules, [*boolean* $merge])

Register an array of modules present in the application

```
<?php

$this->registerModules(array(
    'frontend' => array(
            'className' => 'Multiple\Frontend\Module',
            'path' => '../apps/frontend/Module.php'
    ),
    'backend' => array(
            'className' => 'Multiple\Backend\Module',
            'path' => '../apps/backend/Module.php'
    )
));
```

public *array* **getModules** ()

Return the modules registered in the application

public **setDefaultModule** (*string* $defaultModule)

Sets the module name to be used if the router doesn't return a valid module

public *string* **getDefaultModule** ()

Returns the default module name

public *Phalcon\Http\ResponseInterface* **handle** ([*string* $uri])

Handles a MVC request

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI\Injectable

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

public **__get** (*string* $propertyName) inherited from Phalcon\DI\Injectable

Magic method __get

2.48.113 Class Phalcon\Mvc\Application\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.114 Class Phalcon\Mvc\Collection

implements Phalcon\Mvc\CollectionInterface, *Phalcon\DI\InjectionAwareInterface*, Serializable

This component implements a high level abstraction for NoSQL databases which works with documents

Constants

integer **OP_NONE**

integer **OP_CREATE**

integer **OP_UPDATE**

integer **OP_DELETE**

Methods

final public **__construct** ([*Phalcon\DiInterface* $dependencyInjector], [*Phalcon\Mvc\Collection\ManagerInterface* $modelsManager])

Phalcon\Mvc\Model constructor

public **setId** (*mixed* $id)

Sets a value for the _id property, creates a MongoId object if needed

public *MongoId* **getId** ()

Returns the value of the _id property

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injection container

public *Phalcon\DiInterface* **getDI** ()

Returns the dependency injection container

protected **setEventsManager** ()

Sets a custom events manager

protected *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the custom events manager

public *Phalcon\Mvc\Model\ManagerInterface* **getModelsManager** ()

Returns the models manager related to the entity instance

public *array* **getReservedAttributes** ()

Returns an array with reserved properties that cannot be part of the insert/update

protected **useImplicitObjectIds** ()

Sets if a model must use implicit objects ids

protected *Phalcon\Mvc\Collection* **setSource** ()

Sets collection name which model should be mapped

public *string* **getSource** ()

Returns collection name mapped in the model

public *Phalcon\Mvc\Model* **setConnectionService** (*string* $connectionService)

Sets the DependencyInjection connection service name

public *string* **getConnectionService** ()

Returns DependencyInjection connection service

public *MongoDb* **getConnection** ()

Retrieves a database connection

public *mixed* **readAttribute** (*string* $attribute)

Reads an attribute value by its name

```
<?php

echo $robot->readAttribute('name');
```

public **writeAttribute** (*string* $attribute, *mixed* $value)

Writes an attribute value by its name

```
<?php

$robot->writeAttribute('name', 'Rosey');
```

public static *Phalcon\Mvc\Collection* **cloneResult** (*Phalcon\Mvc\Collection* $collection, *array* $document)

Returns a cloned collection

protected static *array* **_getResultset** ()

Returns a collection resultset

protected static *int* **_getGroupResultset** ()

Perform a count over a resultset

protected *boolean* **_preSave** ()

Executes internal hooks before save a document

protected *boolean* **_postSave** ()

Executes internal events after save a document

protected **validate** ()

Executes validators on every validation call

```
<?php

use Phalcon\Mvc\Model\Validator\ExclusionIn as ExclusionIn;

class Subscriptors extends Phalcon\Mvc\Collection
{

public function validation()
{
    $this->validate(new ExclusionIn(array(
            'field' => 'status',
            'domain' => array('A', 'I')
    )));
    if ($this->validationHasFailed() == true) {
            return false;
    }
}

}
```

public *boolean* **validationHasFailed** ()

Check whether validation process has generated any messages

```
<?php

use Phalcon\Mvc\Model\Validator\ExclusionIn as ExclusionIn;

class Subscriptors extends Phalcon\Mvc\Collection
{

public function validation()
{
    $this->validate(new ExclusionIn(array(
            'field' => 'status',
            'domain' => array('A', 'I')
    )));
    if ($this->validationHasFailed() == true) {
            return false;
    }
}

}
```

public *boolean* **fireEvent** (*string* $eventName)

Fires an internal event

public *boolean* **fireEventCancel** (*string* $eventName)

Fires an internal event that cancels the operation

protected *boolean* **_cancelOperation** ()

Cancel the current operation

protected **_exists** ()

Checks if the document exists in the collection

public *Phalcon\Mvc\Model\MessageInterface* [] **getMessages** ()

Returns all the validation messages

```
<?php

$robot = new Robots();
$robot->type = 'mechanical';
$robot->name = 'Astro Boy';
$robot->year = 1952;
if ($robot->save() == false) {
echo "Umh, We can't store robots right now ";
foreach ($robot->getMessages() as $message) {
    echo $message;
}
} else {
echo "Great, a new robot was saved successfully!";
}
```

public **appendMessage** (*Phalcon\Mvc\Model\MessageInterface* $message)

Appends a customized message on the validation process

```
<?php

use \Phalcon\Mvc\Model\Message as Message;

class Robots extends Phalcon\Mvc\Model
{

    public function beforeSave()
    {
            if (this->name == 'Peter') {
                    $message = new Message("Sorry, but a robot cannot be named Peter");
                    $this->appendMessage($message);
            }
    }
}
```

public *boolean* **save** ()

Creates/Updates a collection based on the values in the atributes

public static *Phalcon\Mvc\Collection* **findById** (*string|MongoId* $id)

Find a document by its id (_id)

public static *array* **findFirst** ([*array* $parameters])

Allows to query the first record that match the specified conditions

```
<?php

 //What's the first robot in the robots table?
 $robot = Robots::findFirst();
 echo "The robot name is ", $robot->name, "\n";

 //What's the first mechanical robot in robots table?
 $robot = Robots::findFirst(array(
     array("type" => "mechanical")
 ));
 echo "The first mechanical robot name is ", $robot->name, "\n";

 //Get first virtual robot ordered by name
 $robot = Robots::findFirst(array(
     array("type" => "mechanical"),
     "order" => array("name" => 1)
 ));
 echo "The first virtual robot name is ", $robot->name, "\n";
```

public static *array* **find** ([*array* $parameters])

Allows to query a set of records that match the specified conditions

```
<?php

 //How many robots are there?
 $robots = Robots::find();
 echo "There are ", count($robots), "\n";

 //How many mechanical robots are there?
 $robots = Robots::find(array(
     array("type" => "mechanical")
 ));
```

```
echo "There are ", count($robots), "\n";

//Get and print virtual robots ordered by name
$robots = Robots::findFirst(array(
    array("type" => "virtual"),
    "order" => array("name" => 1)
));
foreach ($robots as $robot) {
  echo $robot->name, "\n";
}

//Get first 100 virtual robots ordered by name
$robots = Robots::find(array(
    array("type" => "virtual"),
    "order" => array("name" => 1),
    "limit" => 100
));
foreach ($robots as $robot) {
  echo $robot->name, "\n";
}
```

public static *array* **count** ([*array* $parameters])

Perform a count over a collection

```
<?php

 echo 'There are ', Robots::count(), ' robots';
```

public static *array* **aggregate** (*array* $parameters)

Perform an aggregation using the Mongo aggregation framework

```
<?php

 echo 'There are ', Robots::aggregate(), ' robots';
```

public *boolean* **delete** ()

Deletes a model instance. Returning true on success or false otherwise.

```
<?php

$robot = Robots::findFirst();
$robot->delete();

foreach (Robots::find() as $robot) {
    $robot->delete();
}
```

public *array* **toArray** ()

Returns the instance as an array representation

```
<?php

 print_r($robot->toArray());
```

public *string* **serialize** ()

Serializes the object ignoring connections or protected properties

public **unserialize** (*string* $data)

Unserializes the object from a serialized string

2.48.115 Class Phalcon\Mvc\Collection\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.116 Class Phalcon\Mvc\Collection\Manager

implements Phalcon\DI\InjectionAwareInterface, Phalcon\Events\EventsAwareInterface

This components controls the initialization of models, keeping record of relations between the different models of the application. A CollectionManager is injected to a model via a Dependency Injector Container such as Phalcon\DI.

```
<?php

 $di = new Phalcon\DI();

 $di->set('collectionManager', function(){
      return new Phalcon\Mvc\Collection\Manager();
```

```
});

$robot = new Robots($di);
```

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** ()

Returns the DependencyInjector container

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager)

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the internal event manager

public **setCustomEventsManager** (*Phalcon\Mvc\CollectionInterface* $model, *Phalcon\Events\ManagerInterface* $eventsManager)

Sets a custom events manager for a specific model

public *Phalcon\Events\ManagerInterface* **getCustomEventsManager** (*Phalcon\Mvc\CollectionInterface* $model)

Returns a custom events manager related to a model

public **initialize** (*Phalcon\Mvc\CollectionInterface* $model)

Initializes a model in the models manager

public *bool* **isInitialized** (*string* $modelName)

Check whether a model is already initialized

public *Phalcon\Mvc\CollectionInterface* **getLastInitialized** ()

Get the latest initialized model

public **setConnectionService** (*Phalcon\Mvc\CollectionInterface* $model, *string* $connectionService)

Sets a connection service for a specific model

public **useImplicitObjectIds** (*Phalcon\Mvc\CollectionInterface* $model, *boolean* $useImplicitObjectIds)

Sets if a model must use implicit objects ids

public *boolean* **isUsingImplicitObjectIds** (*Phalcon\Mvc\CollectionInterface* $model)

Checks if a model is using implicit object ids

public *Phalcon\Db\AdapterInterface* **getConnection** (*Phalcon\Mvc\CollectionInterface* $model)

Returns the connection related to a model

public **notifyEvent** (*string* $eventName, *Phalcon\Mvc\CollectionInterface* $model)

Receives events generated in the models and dispatches them to a events-manager if available Notify the behaviors that are listening in the model

2.48.117 Class Phalcon\Mvc\Controller

extends Phalcon\DINjectable

implements Phalcon\Events\EventsAwareInterface, *Phalcon\DINjectionAwareInterface*

Every application controller should extend this class that encapsulates all the controller functionality The controllers provide the "flow" between models and views. Controllers are responsible for processing the incoming requests from the web browser, interrogating the models for data, and passing that data on to the views for presentation.

```
<?php

class PeopleController extends \Phalcon\Mvc\Controller
{

  //This action will be executed by default
  public function indexAction()
  {

  }

  public function findAction()
  {

  }

  public function saveAction()
  {
   //Forwards flow to the index action
   return $this->dispatcher->forward(array('controller' => 'people', 'action' => 'index'));
  }

  //This action will be executed when a non existent action is requested
  public function notFoundAction()
  {

  }

}
```

Methods

final public **__construct** ()

Phalcon\Mvc\Controller constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI\Injectable

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

public **__get** (*string* $propertyName) inherited from Phalcon\DI\Injectable

Magic method __get

2.48.118 Class Phalcon\Mvc\Dispatcher

extends Phalcon\Dispatcher

implements Phalcon\Events\EventsAwareInterface, Phalcon\DINjectionAwareInterface, Phalcon\DispatcherInterface, Phalcon\Mvc\DispatcherInterface

Dispatching is the process of taking the request object, extracting the module name, controller name, action name, and optional parameters contained in it, and then instantiating a controller and calling an action of that controller.

```
<?php

$di = new Phalcon\DI();

$dispatcher = new Phalcon\Mvc\Dispatcher();

  $dispatcher->setDI($di);

$dispatcher->setControllerName('posts');
$dispatcher->setActionName('index');
$dispatcher->setParams(array());

$controller = $dispatcher->dispatch();
```

Constants

integer **EXCEPTION_NO_DI**

integer **EXCEPTION_CYCLIC_ROUTING**

integer **EXCEPTION_HANDLER_NOT_FOUND**

integer **EXCEPTION_INVALID_HANDLER**

integer **EXCEPTION_INVALID_PARAMS**

integer **EXCEPTION_ACTION_NOT_FOUND**

Methods

public **setControllerSuffix** (*string* $controllerSuffix)

Sets the default controller suffix

public **setDefaultController** (*string* $controllerName)

Sets the default controller name

public **setControllerName** (*string* $controllerName)

Sets the controller name to be dispatched

public *string* **getControllerName** ()

Gets last dispatched controller name

protected **_throwDispatchException** ()

Throws an internal exception

public *Phalcon\Mvc\ControllerInterface* **getLastController** ()

Returns the lastest dispatched controller

public *Phalcon\Mvc\ControllerInterface* **getActiveController** ()

Returns the active controller in the dispatcher

public **__construct** () inherited from Phalcon\Dispatcher

Phalcon\Dispatcher constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\Dispatcher

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\Dispatcher

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\Dispatcher

Sets the events manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\Dispatcher

Returns the internal event manager

public **setActionSuffix** (*string* $actionSuffix) inherited from Phalcon\Dispatcher

Sets the default action suffix

public **setNamespaceName** (*string* $namespaceName) inherited from Phalcon\Dispatcher

Sets a namespace to be prepended to the handler name

public *string* **getNamespaceName** () inherited from Phalcon\Dispatcher

Gets a namespace to be prepended to the current handler name

public **setDefaultNamespace** (*string* $namespace) inherited from Phalcon\Dispatcher

Sets the default namespace

public *string* **getDefaultNamespace** () inherited from Phalcon\Dispatcher

Returns the default namespace

public **setDefaultAction** (*string* $actionName) inherited from Phalcon\Dispatcher

Sets the default action name

public **setActionName** (*string* $actionName) inherited from Phalcon\Dispatcher

Sets the action name to be dispatched

public *string* **getActionName** () inherited from Phalcon\Dispatcher

Gets the lastest dispatched action name

public **setParams** (*array* $params) inherited from Phalcon\Dispatcher

Sets action params to be dispatched

public *array* **getParams** () inherited from Phalcon\Dispatcher

Gets action params

public **setParam** (*mixed* $param, *mixed* $value) inherited from Phalcon\Dispatcher

Set a param by its name or numeric index

public *mixed* **getParam** (*mixed* $param, [*string|array* $filters], [*mixed* $defaultValue]) inherited from Phalcon\Dispatcher

Gets a param by its name or numeric index

public *string* **getActiveMethod** () inherited from Phalcon\Dispatcher

Returns the current method to be/executed in the dispatcher

public *boolean* **isFinished** () inherited from Phalcon\Dispatcher

Checks if the dispatch loop is finished or has more pendent controllers/tasks to disptach

public **setReturnedValue** (*mixed* $value) inherited from Phalcon\Dispatcher

Sets the latest returned value by an action manually

public *mixed* **getReturnedValue** () inherited from Phalcon\Dispatcher

Returns value returned by the lastest dispatched action

public *object* **dispatch** () inherited from Phalcon\Dispatcher

Dispatches a handle action taking into account the routing parameters

public **forward** (*array* $forward) inherited from Phalcon\Dispatcher

Forwards the execution flow to another controller/action

2.48.119 Class Phalcon\Mvc\Dispatcher\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.120 Class Phalcon\Mvc\Micro

extends Phalcon\DINjectable

implements Phalcon\Events\EventsAwareInterface, *Phalcon\DINjectionAwareInterface*, ArrayAccess

With Phalcon you can create "Micro-Framework like" applications. By doing this, you only need to write a minimal amount of code to create a PHP application. Micro applications are suitable to small applications, APIs and prototypes in a practical way.

```
<?php

$app = new Phalcon\Mvc\Micro();

$app->get('/say/welcome/{name}', function ($name) {
    echo "<h1>Welcome $name!</h1>";
});

$app->handle();
```

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the DependencyInjector container

public *Phalcon\Mvc\Router\RouteInterface* **map** (*string* $routePattern, *callable* $handler)

Maps a route to a handler without any HTTP method constraint

public *Phalcon\Mvc\Router\RouteInterface* **get** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is GET

public *Phalcon\Mvc\Router\RouteInterface* **post** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is POST

public *Phalcon\Mvc\Router\RouteInterface* **put** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is PUT

public *Phalcon\Mvc\Router\RouteInterface* **patch** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is PATCH

public *Phalcon\Mvc\Router\RouteInterface* **head** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is HEAD

public *Phalcon\Mvc\Router\RouteInterface* **delete** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is DELETE

public *Phalcon\Mvc\Router\RouteInterface* **options** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is OPTIONS

public **notFound** (*callable* $handler)

Sets a handler that will be called when the router doesn't match any of the defined routes

public *Phalcon\Mvc\RouterInterface* **getRouter** ()

Returns the internal router used by the application

public *Phalcon\DI\ServiceInterface* **setService** (*string* $serviceName, *mixed* $definition, [*boolean* $shared])

Sets a service from the DI

public *boolean* **hasService** (*string* $serviceName)

Checks if a service is registered in the DI

public *object* **getService** (*string* $serviceName)

Obtains a service from the DI

public *mixed* **getSharedService** (*string* $serviceName)

Obtains a shared service from the DI

public *mixed* **handle** ([*string* $uri])

Handle the whole request

public **setActiveHandler** (*callable* $activeHandler)

Sets externally the handler that must be called by the matched route

public *callable* **getActiveHandler** ()

Return the handler that will be called for the matched route

public *mixed* **getReturnedValue** ()

Returns the value returned by the executed handler

public *boolean* **offsetExists** (*string* $alias)

Check if a service is registered in the internal services container using the array syntax

public **offsetSet** (*string* $alias, *mixed* $definition)

Allows to register a shared service in the internal services container using the array syntax

```
<?php

$app['request'] = new Phalcon\Http\Request();
```

public *mixed* **offsetGet** (*string* $alias)

Allows to obtain a shared service in the internal services container using the array syntax

```
<?php

var_dump($di['request']);
```

public **offsetUnset** (*string* $alias)

Removes a service from the internal services container using the array syntax

public *Phalcon\Mvc\Micro* **before** (*callable* $handler)

Appends a before middleware to be called before execute the route

public *Phalcon\Mvc\Micro* **after** (*callable* $handler)

Appends an 'after' middleware to be called after execute the route

public *Phalcon\Mvc\Micro* **finish** (*callable* $handler)

Appends an 'finish' middleware to be called when the request is finished

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

public **__get** (*string* $propertyName) inherited from Phalcon\DI\Injectable

Magic method __get

2.48.121 Class Phalcon\Mvc\Micro\Collection

Groups handlers as controllers

Methods

protected **_addMap** ()

...

public *Phalcon\Mvc\Router\RouteInterface* **get** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is GET

public *Phalcon\Mvc\Router\RouteInterface* **post** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is POST

public *Phalcon\Mvc\Router\RouteInterface* **put** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is PUT

public *Phalcon\Mvc\Router\RouteInterface* **patch** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is PATCH

public *Phalcon\Mvc\Router\RouteInterface* **head** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is HEAD

public *Phalcon\Mvc\Router\RouteInterface* **delete** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is DELETE

public *Phalcon\Mvc\Router\RouteInterface* **options** (*string* $routePattern, *callable* $handler)

Maps a route to a handler that only matches if the HTTP method is OPTIONS

2.48.122 Class Phalcon\Mvc\Micro\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.123 Class Phalcon\Mvc\Model

implements Phalcon\Mvc\ModelInterface, *Phalcon\Mvc\Model\ResultInterface*, *Phalcon\DI\InjectionAwareInterface*, Serializable

Phalcon\Mvc\Model connects business objects and database tables to create a persistable domain model where logic and data are presented in one wrapping. It's an implementation of the object-relational mapping (ORM). A model represents the information (data) of the application and the rules to manipulate that data. Models are primarily used for managing the rules of interaction with a corresponding database table. In most cases, each table in your database will correspond to one model in your application. The bulk of your application's business logic will be concentrated in the models. Phalcon\Mvc\Model is the first ORM written in C-language for PHP, giving to developers high performance when interacting with databases while is also easy to use.

```php
<?php

 $robot = new Robots();
 $robot->type = 'mechanical'
 $robot->name = 'Astro Boy';
 $robot->year = 1952;
 if ($robot->save() == false) {
  echo "Umh, We can store robots: ";
  foreach ($robot->getMessages() as $message) {
    echo $message;
  }
 } else {
  echo "Great, a new robot was saved successfully!";
 }
```

Constants

integer **OP_NONE**

integer **OP_CREATE**

integer **OP_UPDATE**

integer **OP_DELETE**

integer **DIRTY_STATE_PERSISTENT**

integer **DIRTY_STATE_TRANSIENT**

integer **DIRTY_STATE_DETACHED**

Methods

final public **__construct** ([*Phalcon\DiInterface* $dependencyInjector], [*Phalcon\Mvc\Model\ManagerInterface* $modelsManager])

Phalcon\Mvc\Model constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injection container

public *Phalcon\DiInterface* **getDI** ()

Returns the dependency injection container

protected **setEventsManager** ()

Sets a custom events manager

protected *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the custom events manager

public *Phalcon\Mvc\Model\MetaDataInterface* **getModelsMetaData** ()

Returns the models meta-data service related to the entity instance

public *Phalcon\Mvc\Model\ManagerInterface* **getModelsManager** ()

Returns the models manager related to the entity instance

public *Phalcon\Mvc\Model* **setTransaction** (*Phalcon\Mvc\Model\TransactionInterface* $transaction)

Sets a transaction related to the Model instance

```
<?php

use Phalcon\Mvc\Model\Transaction\Manager as TxManager;
use Phalcon\Mvc\Model\Transaction\Failed as TxFailed;

try {

  $txManager = new TxManager();

  $transaction = $txManager->get();

  $robot = new Robots();
  $robot->setTransaction($transaction);
  $robot->name = 'WALL·E';
  $robot->created_at = date('Y-m-d');
  if ($robot->save() == false) {
    $transaction->rollback("Can't save robot");
  }

  $robotPart = new RobotParts();
  $robotPart->setTransaction($transaction);
  $robotPart->type = 'head';
  if ($robotPart->save() == false) {
    $transaction->rollback("Robot part cannot be saved");
  }

  $transaction->commit();

} catch (TxFailed $e) {
  echo 'Failed, reason: ', $e->getMessage();
}
```

protected *Phalcon\Mvc\Model* **setSource** ()

Sets table name which model should be mapped

public *string* **getSource** ()

Returns table name mapped in the model

protected *Phalcon\Mvc\Model* **setSchema** ()

Sets schema name where table mapped is located

public *string* **getSchema** ()

Returns schema name where table mapped is located

public *Phalcon\Mvc\Model* **setConnectionService** (*string* $connectionService)

Sets the DependencyInjection connection service name

public *Phalcon\Mvc\Model* **setReadConnectionService** (*string* $connectionService)

Sets the DependencyInjection connection service name used to read data

public *Phalcon\Mvc\Model* **setWriteConnectionService** (*string* $connectionService)

Sets the DependencyInjection connection service name used to write data

public *string* **getReadConnectionService** ()

Returns the DependencyInjection connection service name used to read data related the model

public *string* **getWriteConnectionService** ()

Returns the DependencyInjection connection service name used to write data related to the model

public *Phalcon\Mvc\Model* **setDirtyState** (*int* $dirtyState)

Sets the dirty state of the object using one of the DIRTY_STATE_* constants

public *int* **getDirtyState** ()

Returns one of the DIRTY_STATE_* constants telling if the record exists in the database or not

public *Phalcon\Db\AdapterInterface* **getReadConnection** ()

Gets the connection used to read data for the model

public *Phalcon\Db\AdapterInterface* **getWriteConnection** ()

Gets the connection used to write data to the model

public *Phalcon\Mvc\Model* **assign** (*array* $data, [*array* $columnMap])

Assigns values to a model from an array

```
<?php

$robot->assign(array(
  'type' => 'mechanical',
  'name' => 'Astro Boy',
  'year' => 1952
));
```

public static *Phalcon\Mvc\Model* **cloneResultMap** (*Phalcon\Mvc\Model* $base, *array* $data, *array* $columnMap, [*int* $dirtyState], [*boolean* $keepSnapshots])

Assigns values to a model from an array returning a new model.

```
<?php

$robot = \Phalcon\Mvc\Model::cloneResultMap(new Robots(), array(
  'type' => 'mechanical',
  'name' => 'Astro Boy',
  'year' => 1952
));
```

public static *mixed* **cloneResultMapHydrate** (*array* $data, *array* $columnMap, *int* $hydrationMode)

Returns an hydrated result based on the data and the column map

public static *Phalcon\Mvc\Model* **cloneResult** (*Phalcon\Mvc\Model* $base, *array* $data, [*int* $dirtyState])

Assigns values to a model from an array returning a new model

```
<?php

$robot = Phalcon\Mvc\Model::cloneResult(new Robots(), array(
  'type' => 'mechanical',
  'name' => 'Astro Boy',
  'year' => 1952
));
```

public static *Phalcon\Mvc\Model\ResultsetInterface* **find** ([*array* $parameters])

Allows to query a set of records that match the specified conditions

```
<?php

 //How many robots are there?
 $robots = Robots::find();
 echo "There are ", count($robots), "\n";

 //How many mechanical robots are there?
 $robots = Robots::find("type='mechanical'");
 echo "There are ", count($robots), "\n";

 //Get and print virtual robots ordered by name
 $robots = Robots::find(array("type='virtual'", "order" => "name"));
 foreach ($robots as $robot) {
   echo $robot->name, "\n";
 }

 //Get first 100 virtual robots ordered by name
 $robots = Robots::find(array("type='virtual'", "order" => "name", "limit" => 100));
 foreach ($robots as $robot) {
   echo $robot->name, "\n";
 }
```

public static *Phalcon\Mvc\Model* **findFirst** ([*array* $parameters])

Allows to query the first record that match the specified conditions

```
<?php

 //What's the first robot in robots table?
 $robot = Robots::findFirst();
 echo "The robot name is ", $robot->name;

 //What's the first mechanical robot in robots table?
 $robot = Robots::findFirst("type='mechanical'");
 echo "The first mechanical robot name is ", $robot->name;

 //Get first virtual robot ordered by name
 $robot = Robots::findFirst(array("type='virtual'", "order" => "name"));
 echo "The first virtual robot name is ", $robot->name;
```

public static *Phalcon\Mvc\Model\Criteria* **query** ([*Phalcon\DiInterface* $dependencyInjector])

Create a criteria for a specific model

protected *boolean* **_exists** ()

Checks if the current record already exists or not

protected static *Phalcon\Mvc\Model\ResultsetInterface* **_groupResult** ()

Generate a PHQL SELECT statement for an aggregate

public static *int* **count** ([*array* $parameters])

Allows to count how many records match the specified conditions

```
<?php

 //How many robots are there?
 $number = Robots::count();
 echo "There are ", $number, "\n";
```

```
//How many mechanical robots are there?
$number = Robots::count("type='mechanical'");
echo "There are ", $number, " mechanical robots\n";
```

public static *double* **sum** ([*array* $parameters])

Allows to calculate a summatory on a column that match the specified conditions

```
<?php

//How much are all robots?
$sum = Robots::sum(array('column' => 'price'));
echo "The total price of robots is ", $sum, "\n";

//How much are mechanical robots?
$sum = Robots::sum(array("type='mechanical'", 'column' => 'price'));
echo "The total price of mechanical robots is  ", $sum, "\n";
```

public static *mixed* **maximum** ([*array* $parameters])

Allows to get the maximum value of a column that match the specified conditions

```
<?php

//What is the maximum robot id?
$id = Robots::maximum(array('column' => 'id'));
echo "The maximum robot id is: ", $id, "\n";

//What is the maximum id of mechanical robots?
$sum = Robots::maximum(array("type='mechanical'", 'column' => 'id'));
echo "The maximum robot id of mechanical robots is ", $id, "\n";
```

public static *mixed* **minimum** ([*array* $parameters])

Allows to get the minimum value of a column that match the specified conditions

```
<?php

//What is the minimum robot id?
$id = Robots::minimum(array('column' => 'id'));
echo "The minimum robot id is: ", $id;

//What is the minimum id of mechanical robots?
$sum = Robots::minimum(array("type='mechanical'", 'column' => 'id'));
echo "The minimum robot id of mechanical robots is ", $id;
```

public static *double* **average** ([*array* $parameters])

Allows to calculate the average value on a column matching the specified conditions

```
<?php

//What's the average price of robots?
$average = Robots::average(array('column' => 'price'));
echo "The average price is ", $average, "\n";

//What's the average price of mechanical robots?
$average = Robots::average(array("type='mechanical'", 'column' => 'price'));
echo "The average price of mechanical robots is ", $average, "\n";
```

public *boolean* **fireEvent** (*string* $eventName)

Fires an event, implicitly calls behaviors and listeners in the events manager are notified

public *boolean* **fireEventCancel** (*string* $eventName)

Fires an event, implicitly calls behaviors and listeners in the events manager are notified This method stops if one of the callbacks/listeners returns boolean false

protected *boolean* **_cancelOperation** ()

Cancel the current operation

public **appendMessage** (*Phalcon\Mvc\Model\MessageInterface* $message)

Appends a customized message on the validation process

```
<?php

 use \Phalcon\Mvc\Model\Message as Message;

 class Robots extends Phalcon\Mvc\Model
 {

   public function beforeSave()
   {
     if (this->name == 'Peter') {
        $message = new Message("Sorry, but a robot cannot be named Peter");
        $this->appendMessage($message);
     }
   }
 }
```

protected **validate** ()

Executes validators on every validation call

```
<?php

use Phalcon\Mvc\Model\Validator\ExclusionIn as ExclusionIn;

class Subscriptors extends Phalcon\Mvc\Model
{

public function validation()
  {
            $this->validate(new ExclusionIn(array(
            'field' => 'status',
            'domain' => array('A', 'I')
    )));
    if ($this->validationHasFailed() == true) {
            return false;
    }
}

}
```

public *boolean* **validationHasFailed** ()

Check whether validation process has generated any messages

```
<?php

use Phalcon\Mvc\Model\Validator\ExclusionIn as ExclusionIn;
```

```
class Subscriptors extends Phalcon\Mvc\Model
{

public function validation()
  {
            $this->validate(new ExclusionIn(array(
            'field' => 'status',
            'domain' => array('A', 'I')
    )));
    if ($this->validationHasFailed() == true) {
            return false;
    }
}

}
```

public *Phalcon\Mvc\Model\MessageInterface* [] **getMessages** ()

Returns all the validation messages

```
<?php

$robot = new Robots();
$robot->type = 'mechanical';
$robot->name = 'Astro Boy';
$robot->year = 1952;
if ($robot->save() == false) {
    echo "Umh, We can't store robots right now ";
    foreach ($robot->getMessages() as $message) {
            echo $message;
    }
} else {
    echo "Great, a new robot was saved successfully!";
}
```

protected *boolean* **_checkForeignKeys** ()

Reads "belongs to" relations and check the virtual foreign keys when inserting or updating records

protected *boolean* **_checkForeignKeysReverse** ()

Reads both "hasMany" and "hasOne" relations and checks the virtual foreign keys when deleting records

protected *boolean* **_preSave** ()

Executes internal hooks before save a record

protected *boolean* **_postSave** ()

Executes internal events after save a record

protected *boolean* **_doLowInsert** ()

Sends a pre-build INSERT SQL statement to the relational database system

protected *boolean* **_doLowUpdate** ()

Sends a pre-build UPDATE SQL statement to the relational database system

protected **_preSaveRelatedRecords** ()

protected **_postSaveRelatedRecords** ()

Save the related records assigned in the has-one/has-many relations

public *boolean* **save** ([*array* $data], [*array* $whiteList])

Inserts or updates a model instance. Returning true on success or false otherwise.

```
<?php

//Creating a new robot
$robot = new Robots();
$robot->type = 'mechanical'
$robot->name = 'Astro Boy';
$robot->year = 1952;
$robot->save();

//Updating a robot name
$robot = Robots::findFirst("id=100");
$robot->name = "Biomass";
$robot->save();
```

public *boolean* **create** ([*array* $data], [*array* $whiteList])

Inserts a model instance. If the instance already exists in the persistance it will throw an exception Returning true on success or false otherwise.

```
<?php

//Creating a new robot
$robot = new Robots();
$robot->type = 'mechanical'
$robot->name = 'Astro Boy';
$robot->year = 1952;
$robot->create();

  //Passing an array to create
  $robot = new Robots();
  $robot->create(array(
      'type' => 'mechanical',
      'name' => 'Astroy Boy',
      'year' => 1952
  ));
```

public *boolean* **update** ([*array* $data], [*array* $whiteList])

Updates a model instance. If the instance doesn't exist in the persistance it will throw an exception Returning true on success or false otherwise.

```
<?php

//Updating a robot name
$robot = Robots::findFirst("id=100");
$robot->name = "Biomass";
$robot->update();
```

public *boolean* **delete** ()

Deletes a model instance. Returning true on success or false otherwise.

```
<?php

$robot = Robots::findFirst("id=100");
$robot->delete();
```

```
foreach(Robots::find("type = 'mechanical'") as $robot){
   $robot->delete();
}
```

public *int* **getOperationMade** ()

Returns the type of the latest operation performed by the ORM Returns one of the OP_* class constants

public **refresh** ()

Refreshes the model attributes re-querying the record from the database

public **skipOperation** (*boolean* $skip)

Skips the current operation forcing a success state

public *mixed* **readAttribute** (*string* $attribute)

Reads an attribute value by its name

```
<?php

 echo $robot->readAttribute('name');
```

public **writeAttribute** (*string* $attribute, *mixed* $value)

Writes an attribute value by its name

```
<?php

    $robot->writeAttribute('name', 'Rosey');
```

protected **skipAttributes** ()

Sets a list of attributes that must be skipped from the generated INSERT/UPDATE statement

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

   public function initialize()
   {
       $this->skipAttributes(array('price'));
   }

}
```

protected **skipAttributesOnCreate** ()

Sets a list of attributes that must be skipped from the generated INSERT statement

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

   public function initialize()
   {
       $this->skipAttributesOnCreate(array('created_at'));
   }

}
```

protected **skipAttributesOnUpdate** ()

Sets a list of attributes that must be skipped from the generated UPDATE statement

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function initialize()
    {
        $this->skipAttributesOnUpdate(array('modified_in'));
    }

}
```

protected **hasOne** ()

Setup a 1-1 relation between two models

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function initialize()
    {
        $this->hasOne('id', 'RobotsDescription', 'robots_id');
    }

}
```

protected **belongsTo** ()

Setup a relation reverse 1-1 between two models

```
<?php

class RobotsParts extends \Phalcon\Mvc\Model
{

    public function initialize()
    {
        $this->belongsTo('robots_id', 'Robots', 'id');
    }

}
```

protected **hasMany** ()

Setup a relation 1-n between two models

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

    public function initialize()
    {
        $this->hasMany('id', 'RobotsParts', 'robots_id');
    }
```

```
}
```

protected **hasManyThrough** ()

Setup a relation n-n between two models through an intermediate relation

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

   public function initialize()
   {
       //A reference relation must be set
       $this->hasMany('id', 'RobotsParts', 'robots_id');

       //Setup a many-to-many relation to Parts through RobotsParts
       $this->hasManyThrough('Parts', 'RobotsParts');
   }

}
```

protected **addBehavior** ()

Setups a behavior in a model

```
<?php

use Phalcon\Mvc\Model\Behaviors\Timestampable;

class Robots extends \Phalcon\Mvc\Model
{

   public function initialize()
   {
    $this->addBehavior(new Timestampable(
            'onCreate' => array(
                    'field' => 'created_at',
                    'format' => 'Y-m-d'
            )
    ));
   }

}
```

protected **keepSnapshots** ()

Sets if the model must keep the original record snapshot in memory

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

   public function initialize()
   {
    $this->keepSnapshots(true);
   }

}
```

public **setSnapshotData** (*array* $data, [*array* $columnMap])

Sets the record's snapshot data. This method is used internally to set snapshot data when the model was set up to keep snapshot data

public *boolean* **hasSnapshotData** ()

Checks if the object has internal snapshot data

public *array* **getSnapshotData** ()

Returns the internal snapshot data

public **hasChanged** ([*boolean* $fieldName])

Check if an specific attribute has changed This only works if the model is keeping data snapshots

public *array* **getChangedFields** ()

Returns a list of changed values

protected **useDynamicUpdate** ()

Sets if a model must use dynamic update instead of the all-field update

```
<?php

class Robots extends \Phalcon\Mvc\Model
{

   public function initialize()
   {
    $this->useDynamicUpdate(true);
   }

}
```

public *Phalcon\Mvc\Model\ResultsetInterface* **getRelated** (*string* $alias, [*array* $arguments])

Returns related records based on defined relations

protected *mixed* **_getRelatedRecords** ()

Returns related records defined relations depending on the method name

public *mixed* **__call** (*string* $method, [*array* $arguments])

Handles method calls when a method is not implemented

public static *mixed* **__callStatic** (*string* $method, [*array* $arguments])

Handles method calls when a static method is not implemented

public **__set** (*string* $property, *mixed* $value)

Magic method to assign values to the the model

public *Phalcon\Mvc\Model\Resultset* **__get** (*string* $property)

Magic method to get related records using the relation alias as a property

public **__isset** (*string* $property)

Magic method to check if a property is a valid relation

public *string* **serialize** ()

Serializes the object ignoring connections or static properties

public **unserialize** (*string* $data)

Unserializes the object from a serialized string

public *array* **dump** ()

Returns a simple representation of the object that can be used with var_dump

```
<?php

 var_dump($robot->dump());
```

public *array* **toArray** ()

Returns the instance as an array representation

```
<?php

 print_r($robot->toArray());
```

public static **setup** (*array* $options)

Enables/disables options in the ORM

2.48.124 Class Phalcon\Mvc\Model\Behavior

This is an optional base class for ORM behaviors

Methods

public **__construct** ([*array* $options])

protected **mustTakeAction** ()

Checks whether the behavior must take action on certain event

protected *array* **getOptions** ()

Returns the behavior options related to an event

public **notify** (*string* $type, *Phalcon\Mvc\ModelInterface* $model)

This method receives the notifications from the EventsManager

public **missingMethod** (*Phalcon\Mvc\ModelInterface* $model, *string* $method, [*array* $arguments])

Acts as fallbacks when a missing method is called on the model

2.48.125 Class Phalcon\Mvc\Model\Behavior\SoftDelete

extends Phalcon\Mvc\Model\Behavior

implements Phalcon\Mvc\Model\BehaviorInterface

Instead of permanently delete a record it marks the record as deleted changing the value of a flag column

Methods

public **notify** (*string* $type, *Phalcon\Mvc\ModelInterface* $model)

Listens for notifications from the models manager

public **__construct** ([*array* $options]) inherited from Phalcon\Mvc\Model\Behavior

Phalcon\Mvc\Model\Behavior

protected **mustTakeAction** () inherited from Phalcon\Mvc\Model\Behavior

Checks whether the behavior must take action on certain event

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Behavior

Returns the behavior options related to an event

public **missingMethod** (*Phalcon\Mvc\ModelInterface* $model, *string* $method, [*array* $arguments]) inherited from Phalcon\Mvc\Model\Behavior

Acts as fallbacks when a missing method is called on the model

2.48.126 Class Phalcon\Mvc\Model\Behavior\Timestampable

extends Phalcon\Mvc\Model\Behavior

implements Phalcon\Mvc\Model\BehaviorInterface

Allows to automatically update a model's attribute saving the datetime when a record is created or updated

Methods

public **notify** (*string* $type, *Phalcon\Mvc\ModelInterface* $model)

Listens for notifications from the models manager

public **__construct** ([*array* $options]) inherited from Phalcon\Mvc\Model\Behavior

Phalcon\Mvc\Model\Behavior

protected **mustTakeAction** () inherited from Phalcon\Mvc\Model\Behavior

Checks whether the behavior must take action on certain event

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Behavior

Returns the behavior options related to an event

public **missingMethod** (*Phalcon\Mvc\ModelInterface* $model, *string* $method, [*array* $arguments]) inherited from Phalcon\Mvc\Model\Behavior

Acts as fallbacks when a missing method is called on the model

2.48.127 Class Phalcon\Mvc\Model\Criteria

implements Phalcon\Mvc\Model\CriteriaInterface, *Phalcon\DI\InjectionAwareInterface*

This class allows to build the array parameter required by Phalcon\Mvc\Model::find and Phalcon\Mvc\Model::findFirst, using an object-oriented interfase

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** ()

Returns the DependencyInjector container

public *Phalcon\Mvc\Model\Criteria* **setModelName** (*string* $modelName)

Set a model on which the query will be executed

public *string* **getModelName** ()

Returns an internal model name on which the criteria will be applied

public *Phalcon\Mvc\Model\Criteria* **bind** (*string* $bindParams)

Adds the bind parameter to the criteria

public *Phalcon\Mvc\Model\Criteria* **where** (*string* $conditions)

Adds the conditions parameter to the criteria

public *Phalcon\Mvc\Model\Criteria* **addWhere** (*string* $conditions)

Appends a condition to the current conditions using an AND operator (deprecated)

public *Phalcon\Mvc\Model\Criteria* **andWhere** (*string* $conditions)

Appends a condition to the current conditions using an AND operator

public *Phalcon\Mvc\Model\Criteria* **orWhere** (*string* $conditions)

Appends a condition to the current conditions using an OR operator

public *Phalcon\Mvc\Model\Criteria* **conditions** (*string* $conditions)

Adds the conditions parameter to the criteria

public *Phalcon\Mvc\Model\Criteria* **order** (*string* $orderColumns)

Adds the order-by parameter to the criteria

public *Phalcon\Mvc\Model\Criteria* **limit** (*int* $limit, [*int* $offset])

Adds the limit parameter to the criteria

public *Phalcon\Mvc\Model\Criteria* **forUpdate** ([*boolean* $forUpdate])

Adds the “for_update” parameter to the criteria

public *Phalcon\Mvc\Model\Criteria* **sharedLock** ([*boolean* $sharedLock])

Adds the “shared_lock” parameter to the criteria

public *string* **getWhere** ()

Returns the conditions parameter in the criteria

public *string* **getConditions** ()

Returns the conditions parameter in the criteria

public *string* **getLimit** ()

Returns the limit parameter in the criteria

public *string* **getOrder** ()

Returns the order parameter in the criteria

public *string* **getParams** ()

Returns all the parameters defined in the criteria

public static *static* **fromInput** (*Phalcon\DiInterface* $dependencyInjector, *string* $modelName, *array* $data)

Builds a Phalcon\Mvc\Model\Criteria based on an input array like $_POST

public *Phalcon\Mvc\Model\ResultsetInterface* **execute** ()

Executes a find using the parameters built with the criteria

2.48.128 Class Phalcon\Mvc\Model\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.129 Class Phalcon\Mvc\Model\Manager

implements Phalcon\Mvc\Model\ManagerInterface, Phalcon\DI\InjectionAwareInterface, Phalcon\Events\EventsAwareInterface

This components controls the initialization of models, keeping record of relations between the different models of the application. A ModelsManager is injected to a model via a Dependency Injector Container such as Phalcon\DI.

```
<?php

 $dependencyInjector = new Phalcon\DI();

 $dependencyInjector->set('modelsManager', function(){
      return new Phalcon\Mvc\Model\Manager();
 });

 $robot = new Robots($dependencyInjector);
```

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** ()

Returns the DependencyInjector container

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager)

Sets a global events manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the internal event manager

public **setCustomEventsManager** (*Phalcon\Mvc\ModelInterface* $model, *Phalcon\Events\ManagerInterface* $eventsManager)

Sets a custom events manager for a specific model

public *Phalcon\Events\ManagerInterface* **getCustomEventsManager** (*Phalcon\Mvc\ModelInterface* $model)

Returns a custom events manager related to a model

public **initialize** (*Phalcon\Mvc\ModelInterface* $model)

Initializes a model in the model manager

public *bool* **isInitialized** (*string* $modelName)

Check whether a model is already initialized

public *Phalcon\Mvc\ModelInterface* **getLastInitialized** ()

Get last initialized model

public *Phalcon\Mvc\ModelInterface* **load** (*string* $modelName, [*boolean* $newInstance])

Loads a model throwing an exception if it doesn't exist

public *string* **setModelSource** (*Phalcon\Mvc\Model* $model, *string* $source)

Sets the mapped source for a model

public *string* **getModelSource** (*Phalcon\Mvc\Model* $model)

Returns the mapped source for a model

public *string* **setModelSchema** (*Phalcon\Mvc\Model* $model, *string* $schema)

Sets the mapped schema for a model

public *string* **getModelSchema** (*Phalcon\Mvc\Model* $model)

Returns the mapped schema for a model

public **setConnectionService** (*Phalcon\Mvc\ModelInterface* $model, *string* $connectionService)

Sets both write and read connection service for a model

public **setWriteConnectionService** (*Phalcon\Mvc\ModelInterface* $model, *string* $connectionService)

Sets write connection service for a model

public **setReadConnectionService** (*Phalcon\Mvc\ModelInterface* $model, *string* $connectionService)

Sets read connection service for a model

public *Phalcon\Db\AdapterInterface* **getWriteConnection** (*Phalcon\Mvc\ModelInterface* $model)

Returns the connection to write data related to a model

public *Phalcon\Db\AdapterInterface* **getReadConnection** (*Phalcon\Mvc\ModelInterface* $model)

Returns the connection to read data related to a model

public **getReadConnectionService** (*Phalcon\Mvc\ModelInterface* $model)

Returns the connection service name used to read data related to a model

public **getWriteConnectionService** (*Phalcon\Mvc\ModelInterface* $model)

Returns the connection service name used to write data related to a model

public **notifyEvent** (*string* $eventName, *Phalcon\Mvc\ModelInterface* $model)

Receives events generated in the models and dispatches them to a events-manager if available Notify the behaviors that are listening in the model

public *boolean* **missingMethod** (*Phalcon\Mvc\ModelInterface* $model, *string* $eventName, *array* $data)

Dispatch a event to the listeners and behaviors This method expects that the endpoint listeners/behaviors returns true meaning that a least one is implemented

public **addBehavior** (*Phalcon\Mvc\ModelInterface* $model, *Phalcon\Mvc\Model\BehaviorInterface* $behavior)

Binds a behavior to a model

public **keepSnapshots** (*Phalcon\Mvc\Model* $model, *boolean* $keepSnapshots)

Sets if a model must keep snapshots

public *boolean* **isKeepingSnapshots** (*unknown* $model)

Checks if a model is keeping snapshots for the queried records

public **useDynamicUpdate** (*Phalcon\Mvc\Model* $model, *boolean* $dynamicUpdate)

Sets if a model must use dynamic update instead of the all-field update

public *boolean* **isUsingDynamicUpdate** (*unknown* $model)

Checks if a model is using dynamic update instead of all-field update

public *Phalcon\Mvc\Model\Relation* **addHasOne** (*Phalcon\Mvc\Model* $model, *mixed* $fields, *string* $referencedModel, *mixed* $referencedFields, [*array* $options])

Setup a 1-1 relation between two models

public *Phalcon\Mvc\Model\Relation* **addBelongsTo** (*Phalcon\Mvc\Model* $model, *mixed* $fields, *string* $referencedModel, *mixed* $referencedFields, [*array* $options])

Setup a relation reverse many to one between two models

public **addHasMany** (*Phalcon\Mvc\ModelInterface* $model, *mixed* $fields, *string* $referencedModel, *mixed* $referencedFields, [*array* $options])

Setup a relation 1-n between two models

public **addHasManyThrough** ()

...

public *boolean* **existsBelongsTo** (*string* $modelName, *string* $modelRelation)

Checks whether a model has a belongsTo relation with another model

public *boolean* **existsHasMany** (*string* $modelName, *string* $modelRelation)

Checks whether a model has a hasMany relation with another model

public *boolean* **existsHasOne** (*string* $modelName, *string* $modelRelation)

Checks whether a model has a hasOne relation with another model

public *Phalcon\Mvc\Model\Relation* **getRelationByAlias** (*string* $modelName, *string* $alias)

Returns a relation by its alias

public *Phalcon\Mvc\Model\Resultset\Simple* **getRelationRecords** (*Phalcon\Mvc\Model\Relation* $relation, *string* $method, *Phalcon\Mvc\ModelInterface* $record, [*array* $parameters])

Helper method to query records based on a relation definition

public *object* **getReusableRecords** (*string* $modelName, *string* $key)

Returns a reusable object from the internal list

public **setReusableRecords** (*string* $modelName, *string* $key, *mixed* $records)

Stores a reusable record in the internal list

public **clearReusableObjects** ()

Clears the internal reusable list

public *Phalcon\Mvc\Model\ResultsetInterface* **getBelongsToRecords** (*string* $method, *string* $modelName, *string* $modelRelation, *Phalcon\Mvc\Model* $record, [*array* $parameters])

Gets belongsTo related records from a model

public *Phalcon\Mvc\Model\ResultsetInterface* **getHasManyRecords** (*string* $method, *string* $modelName, *string* $modelRelation, *Phalcon\Mvc\Model* $record, [*array* $parameters])

Gets hasMany related records from a model

public *Phalcon\Mvc\Model\ResultsetInterface* **getHasOneRecords** (*string* $method, *string* $modelName, *string* $modelRelation, *Phalcon\Mvc\Model* $record, [*array* $parameters])

Gets belongsTo related records from a model

public *Phalcon\Mvc\Model\RelationInterface* [] **getBelongsTo** (*Phalcon\Mvc\ModelInterface* $model)

Gets all the belongsTo relations defined in a model

```
<?php

$relations = $modelsManager->getBelongsTo(new Robots());
```

public *Phalcon\Mvc\Model\RelationInterface* [] **getHasMany** (*Phalcon\Mvc\ModelInterface* $model)

Gets hasMany relations defined on a model

public *array* **getHasOne** (*Phalcon\Mvc\ModelInterface* $model)

Gets hasOne relations defined on a model

public *array* **getHasOneAndHasMany** (*Phalcon\Mvc\ModelInterface* $model)

Gets hasOne relations defined on a model

public *Phalcon\Mvc\Model\RelationInterface* [] **getRelations** (*string* $modelName)

Query all the relationships defined on a model

public *Phalcon\Mvc\Model\RelationInterface* **getRelationsBetween** (*string* $first, *string* $second)

Query the first relationship defined between two models

public *Phalcon\Mvc\Model\QueryInterface* **createQuery** (*string* $phql)

Creates a Phalcon\Mvc\Model\Query without execute it

public *Phalcon\Mvc\Model\QueryInterface* **executeQuery** (*string* $phql, [*array* $placeholders])

Creates a Phalcon\Mvc\Model\Query and execute it

public *Phalcon\Mvc\Model\Query\BuilderInterface* **createBuilder** ([*string* $params])

Creates a Phalcon\Mvc\Model\Query\Builder

public *Phalcon\Mvc\Model\QueryInterface* **getLastQuery** ()

Returns the last query created or executed in the models manager

2.48.130 Class Phalcon\Mvc\Model\Message

implements Phalcon\Mvc\Model\MessageInterface

Encapsulates validation info generated before save/delete records fails

```
<?php

  use Phalcon\Mvc\Model\Message as Message;

  class Robots extends Phalcon\Mvc\Model
  {

    public function beforeSave()
    {
      if (this->name == 'Peter') {
        $text = "A robot cannot be named Peter";
        $field = "name";
        $type = "InvalidValue";
        $message = new Message($text, $field, $type);
        $this->appendMessage($message);
     }
```

```
    }

}
```

Methods

public **__construct** (*string* $message, [*string* $field], [*string* $type], [*Phalcon\Mvc\ModelInterface* $model])

Phalcon\Mvc\Model\Message constructor

public *Phalcon\Mvc\Model\Message* **setType** (*string* $type)

Sets message type

public *string* **getType** ()

Returns message type

public *Phalcon\Mvc\Model\Message* **setMessage** (*string* $message)

Sets verbose message

public *string* **getMessage** ()

Returns verbose message

public *Phalcon\Mvc\Model\Message* **setField** (*string* $field)

Sets field name related to message

public *string* **getField** ()

Returns field name related to message

public *Phalcon\Mvc\Model\Message* **setModel** (*Phalcon\Mvc\ModelInterface* $model)

Set the model who generates the message

public *Phalcon\Mvc\ModelInterface* **getModel** ()

Returns the model that produced the message

public *string* **__toString** ()

Magic __toString method returns verbose message

public static *Phalcon\Mvc\Model\Message* **__set_state** (*array* $message)

Magic __set_state helps to re-build messages variable exporting

2.48.131 Class Phalcon\Mvc\Model\MetaData

implements Phalcon\DI\InjectionAwareInterface

Because Phalcon\Mvc\Model requires meta-data like field names, data types, primary keys, etc. this component collect them and store for further querying by Phalcon\Mvc\Model. Phalcon\Mvc\Model\MetaData can also use adapters to store temporarily or permanently the meta-data. A standard Phalcon\Mvc\Model\MetaData can be used to query model attributes:

```
<?php

$metaData = new Phalcon\Mvc\Model\MetaData\Memory();
$attributes = $metaData->getAttributes(new Robots());
print_r($attributes);
```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

protected **_initialize** ()

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** ()

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\Strategy\Introspection* $strategy)

Set the meta-data extraction strategy

public *Phalcon\Mvc\Model\MetaData\Strategy\Introspection* **getStrategy** ()

Return the strategy to obtain the meta-data

public *array* **readMetaData** (*Phalcon\Mvc\ModelInterface* $model)

Reads the complete meta-data for certain model

```
<?php

print_r($metaData->readMetaData(new Robots());
```

public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index)

Reads meta-data for certain model using a MODEL_* constant

```
<?php

print_r($metaData->writeColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP, array('leNa
```

public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index, *mixed* $data)

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r($metaData->writeColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP, array('leNa
```

public *array* **readColumnMap** (*Phalcon\Mvc\ModelInterface* $model)

Reads the ordered/reversed column map for certain model

```
<?php

print_r($metaData->readColumnMap(new Robots()));
```

public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index)

Reads column-map information for certain model using a MODEL_* constant

```
<?php

print_r($metaData->readColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP));
```

public *array* **getAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns table attributes names (fields)

```
<?php

print_r($metaData->getAttributes(new Robots()));
```

public *array* **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns an array of fields which are part of the primary key

```
<?php

print_r($metaData->getPrimaryKeyAttributes(new Robots()));
```

public *array* **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns an arrau of fields which are not part of the primary key

```
<?php

print_r($metaData->getNonPrimaryKeyAttributes(new Robots()));
```

public *array* **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns an array of not null attributes

```
<?php

print_r($metaData->getNotNullAttributes(new Robots()));
```

public *array* **getDataTypes** (*Phalcon\Mvc\ModelInterface* $model)

Returns attributes and their data types

```
<?php

print_r($metaData->getDataTypes(new Robots()));
```

public *array* **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* $model)

Returns attributes which types are numerical

```
<?php

print_r($metaData->getDataTypesNumeric(new Robots()));
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* $model)

Returns the name of identity field (if one is present)

```
<?php

print_r($metaData->getIdentityField(new Robots()));
```

public *array* **getBindTypes** (*Phalcon\Mvc\ModelInterface* $model)

Returns attributes and their bind data types

```
<?php

print_r($metaData->getBindTypes(new Robots()));
```

public *array* **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r($metaData->getAutomaticCreateAttributes(new Robots()));
```

public *array* **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r($metaData->getAutomaticUpdateAttributes(new Robots()));
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes)

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes(new Robots(), array('created_at' => true));
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes)

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes(new Robots(), array('modified_at' => true));
```

public *array* **getColumnMap** (*Phalcon\Mvc\ModelInterface* $model)

Returns the column map if any

```
<?php

print_r($metaData->getColumnMap(new Robots()));
```

public *array* **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* $model)

Returns the reverse column map if any

```
<?php

print_r($metaData->getReverseColumnMap(new Robots()));
```

public *boolean* **hasAttribute** (*Phalcon\Mvc\ModelInterface* $model, *string* $attribute)

Check if a model has certain attribute

```
<?php

var_dump($metaData->hasAttribute(new Robots(), 'name'));
```

public *boolean* **isEmpty** ()

Checks if the internal meta-data container is empty

```
<?php

var_dump($metaData->isEmpty());
```

public **reset** ()

Resets internal meta-data in order to regenerate it

```
<?php

$metaData->reset();
```

2.48.132 Class Phalcon\Mvc\Model\MetaData\Apc

extends Phalcon\Mvc\Model\MetaData

implements Phalcon\DI\InjectionAwareInterface, *Phalcon\Mvc\Model\MetaDataInterface*

Stores model meta-data in the APC cache. Data will erased if the web server is restarted By default meta-data is stored for 48 hours (172800 seconds) You can query the meta-data by printing apc_fetch('PMM') or apc_fetch('PMMmy-app-id')

```
<?php

$metaData = new Phalcon\Mvc\Model\Metadata\Apc(array(
    'prefix' => 'my-app-id',
    'lifetime' => 86400
));
```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([*array* $options])

Phalcon\Mvc\Model\MetaData\Apc constructor

public *array* **read** (*string* $key)

Reads meta-data from APC

public **write** (*string* $key, *array* $data)

Writes the meta-data to APC

protected **_initialize** () inherited from Phalcon\Mvc\Model\MetaData

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\Mvc\Model\MetaData

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\Mvc\Model\MetaData

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\Strategy\Introspection* $strategy) inherited from Phalcon\Mvc\Model\MetaData

Set the meta-data extraction strategy

public *Phalcon\Mvc\Model\MetaData\Strategy\Introspection* **getStrategy** () inherited from Phalcon\Mvc\Model\MetaData

Return the strategy to obtain the meta-data

public *array* **readMetaData** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Reads the complete meta-data for certain model

```
<?php

print_r($metaData->readMetaData(new Robots());
```

public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index) inherited from Phalcon\Mvc\Model\MetaData

Reads meta-data for certain model using a MODEL_* constant

```
<?php

print_r($metaData->writeColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP, array('leNa
```

public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index, *mixed* $data) inherited from Phalcon\Mvc\Model\MetaData

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r($metaData->writeColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP, array('leNa
```

public *array* **readColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Reads the ordered/reversed column map for certain model

```
<?php

print_r($metaData->readColumnMap(new Robots()));
```

public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index) inherited from Phalcon\Mvc\Model\MetaData

Reads column-map information for certain model using a MODEL_* constant

```
<?php

print_r($metaData->readColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP));
```

public *array* **getAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns table attributes names (fields)

```
<?php

print_r($metaData->getAttributes(new Robots()));
```

public *array* **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an array of fields which are part of the primary key

```
<?php

print_r($metaData->getPrimaryKeyAttributes(new Robots()));
```

public *array* **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an arrau of fields which are not part of the primary key

```
<?php

print_r($metaData->getNonPrimaryKeyAttributes(new Robots()));
```

public *array* **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an array of not null attributes

```
<?php

print_r($metaData->getNotNullAttributes(new Robots()));
```

public *array* **getDataTypes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes and their data types

```
<?php

print_r($metaData->getDataTypes(new Robots()));
```

public *array* **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes which types are numerical

```
<?php

print_r($metaData->getDataTypesNumeric(new Robots()));
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the name of identity field (if one is present)

```
<?php

print_r($metaData->getIdentityField(new Robots()));
```

public *array* **getBindTypes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes and their bind data types

```
<?php

print_r($metaData->getBindTypes(new Robots()));
```

public *array* **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r($metaData->getAutomaticCreateAttributes(new Robots()));
```

public *array* **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r($metaData->getAutomaticUpdateAttributes(new Robots()));
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes) inherited from Phalcon\Mvc\Model\MetaData

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes(new Robots(), array('created_at' => true));
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes) inherited from Phalcon\Mvc\Model\MetaData

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes(new Robots(), array('modified_at' => true));
```

public *array* **getColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the column map if any

```
<?php

print_r($metaData->getColumnMap(new Robots()));
```

public *array* **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the reverse column map if any

```
<?php

print_r($metaData->getReverseColumnMap(new Robots()));
```

public *boolean* **hasAttribute** (*Phalcon\Mvc\ModelInterface* $model, *string* $attribute) inherited from Phalcon\Mvc\Model\MetaData

Check if a model has certain attribute

```
<?php

var_dump($metaData->hasAttribute(new Robots(), 'name'));
```

public *boolean* **isEmpty** () inherited from Phalcon\Mvc\Model\MetaData

Checks if the internal meta-data container is empty

```
<?php

var_dump($metaData->isEmpty());
```

public **reset** () inherited from Phalcon\Mvc\Model\MetaData

Resets internal meta-data in order to regenerate it

```
<?php

$metaData->reset();
```

2.48.133 Class Phalcon\Mvc\Model\MetaData\Files

extends Phalcon\Mvc\Model\MetaData

implements Phalcon\DI\InjectionAwareInterface, *Phalcon\Mvc\Model\MetaDataInterface*

Stores model meta-data in PHP files.

```
<?php

 $metaData = new Phalcon\Mvc\Model\Metadata\Files(array(
    'metaDataDir' => 'app/cache/metadata/'
 ));
```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([*array* $options])

Phalcon\Mvc\Model\MetaData\Files constructor

public *array* **read** (*string* $key)

Reads meta-data from files

public **write** (*string* $key, *array* $data)

Writes the meta-data to files

protected **_initialize** () inherited from Phalcon\Mvc\Model\MetaData

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\Mvc\Model\MetaData

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\Mvc\Model\MetaData

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\Strategy\Introspection* $strategy) inherited from Phalcon\Mvc\Model\MetaData

Set the meta-data extraction strategy

public *Phalcon\Mvc\Model\MetaData\Strategy\Introspection* **getStrategy** () inherited from Phalcon\Mvc\Model\MetaData

Return the strategy to obtain the meta-data

public *array* **readMetaData** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Reads the complete meta-data for certain model

```
<?php

print_r($metaData->readMetaData(new Robots());
```

public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index) inherited from Phalcon\Mvc\Model\MetaData

Reads meta-data for certain model using a MODEL_* constant

```
<?php

print_r($metaData->writeColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP, array('leNa
```

public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index, *mixed* $data) inherited from Phalcon\Mvc\Model\MetaData

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r($metaData->writeColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP, array('leNa
```

public *array* **readColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Reads the ordered/reversed column map for certain model

```
<?php

print_r($metaData->readColumnMap(new Robots()));
```

public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index) inherited from Phalcon\Mvc\Model\MetaData

Reads column-map information for certain model using a MODEL_* constant

```
<?php

print_r($metaData->readColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP));
```

public *array* **getAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns table attributes names (fields)

```
<?php

print_r($metaData->getAttributes(new Robots()));
```

public *array* **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an array of fields which are part of the primary key

```
<?php

print_r($metaData->getPrimaryKeyAttributes(new Robots()));
```

public *array* **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an arrau of fields which are not part of the primary key

```
<?php

print_r($metaData->getNonPrimaryKeyAttributes(new Robots()));
```

public *array* **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an array of not null attributes

```
<?php

print_r($metaData->getNotNullAttributes(new Robots()));
```

public *array* **getDataTypes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes and their data types

```
<?php

print_r($metaData->getDataTypes(new Robots()));
```

public *array* **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes which types are numerical

```
<?php

print_r($metaData->getDataTypesNumeric(new Robots()));
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the name of identity field (if one is present)

```
<?php

print_r($metaData->getIdentityField(new Robots()));
```

public *array* **getBindTypes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes and their bind data types

```
<?php

print_r($metaData->getBindTypes(new Robots()));
```

public *array* **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r($metaData->getAutomaticCreateAttributes(new Robots()));
```

public *array* **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r($metaData->getAutomaticUpdateAttributes(new Robots()));
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes) inherited from Phalcon\Mvc\Model\MetaData

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes(new Robots(), array('created_at' => true));
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes) inherited from Phalcon\Mvc\Model\MetaData

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes(new Robots(), array('modified_at' => true));
```

public *array* **getColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the column map if any

```
<?php

print_r($metaData->getColumnMap(new Robots()));
```

public *array* **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the reverse column map if any

```
<?php

print_r($metaData->getReverseColumnMap(new Robots()));
```

public *boolean* **hasAttribute** (*Phalcon\Mvc\ModelInterface* $model, *string* $attribute) inherited from Phalcon\Mvc\Model\MetaData

Check if a model has certain attribute

```
<?php

var_dump($metaData->hasAttribute(new Robots(), 'name'));
```

public *boolean* **isEmpty** () inherited from Phalcon\Mvc\Model\MetaData

Checks if the internal meta-data container is empty

```
<?php

var_dump($metaData->isEmpty());
```

public **reset** () inherited from Phalcon\Mvc\Model\MetaData

Resets internal meta-data in order to regenerate it

```
<?php

$metaData->reset();
```

2.48.134 Class Phalcon\Mvc\Model\MetaData\Memory

extends Phalcon\Mvc\Model\MetaData

implements Phalcon\DI\InjectionAwareInterface, *Phalcon\Mvc\Model\MetaDataInterface*

Stores model meta-data in memory. Data will be erased when the request finishes

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([*array* $options])

Phalcon\Mvc\Model\MetaData\Memory constructor

public *array* **read** (*string* $key)

Reads the meta-data from temporal memory

public **write** (*string* $key, *array* $metaData)

Writes the meta-data to temporal memory

protected **_initialize** () inherited from Phalcon\Mvc\Model\MetaData

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\Mvc\Model\MetaData

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\Mvc\Model\MetaData

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\Strategy\Introspection* $strategy) inherited from Phalcon\Mvc\Model\MetaData

Set the meta-data extraction strategy

public *Phalcon\Mvc\Model\MetaData\Strategy\Introspection* **getStrategy** () inherited from Phalcon\Mvc\Model\MetaData

Return the strategy to obtain the meta-data

public *array* **readMetaData** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Reads the complete meta-data for certain model

```
<?php

print_r($metaData->readMetaData(new Robots());
```

public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index) inherited from Phalcon\Mvc\Model\MetaData

Reads meta-data for certain model using a MODEL_* constant

```
<?php

print_r($metaData->writeColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP, array('leNa
```

public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index, *mixed* $data) inherited from Phalcon\Mvc\Model\MetaData

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r($metaData->writeColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP, array('leNa
```

public *array* **readColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Reads the ordered/reversed column map for certain model

```
<?php

print_r($metaData->readColumnMap(new Robots()));
```

public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index) inherited from Phalcon\Mvc\Model\MetaData

Reads column-map information for certain model using a MODEL_* constant

```
<?php

print_r($metaData->readColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP));
```

public *array* **getAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns table attributes names (fields)

```
<?php

print_r($metaData->getAttributes(new Robots()));
```

public *array* **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an array of fields which are part of the primary key

```
<?php

print_r($metaData->getPrimaryKeyAttributes(new Robots()));
```

public *array* **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an arrau of fields which are not part of the primary key

```
<?php

print_r($metaData->getNonPrimaryKeyAttributes(new Robots()));
```

public *array* **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an array of not null attributes

```
<?php

print_r($metaData->getNotNullAttributes(new Robots()));
```

public *array* **getDataTypes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes and their data types

```
<?php

print_r($metaData->getDataTypes(new Robots()));
```

public *array* **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes which types are numerical

```
<?php

print_r($metaData->getDataTypesNumeric(new Robots()));
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the name of identity field (if one is present)

```
<?php

print_r($metaData->getIdentityField(new Robots()));
```

public *array* **getBindTypes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes and their bind data types

```
<?php

print_r($metaData->getBindTypes(new Robots()));
```

public *array* **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r($metaData->getAutomaticCreateAttributes(new Robots()));
```

public *array* **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r($metaData->getAutomaticUpdateAttributes(new Robots()));
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes) inherited from Phalcon\Mvc\Model\MetaData

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes(new Robots(), array('created_at' => true));
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes) inherited from Phalcon\Mvc\Model\MetaData

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes(new Robots(), array('modified_at' => true));
```

public *array* **getColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the column map if any

```
<?php

print_r($metaData->getColumnMap(new Robots()));
```

public *array* **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the reverse column map if any

```
<?php

print_r($metaData->getReverseColumnMap(new Robots()));
```

public *boolean* **hasAttribute** (*Phalcon\Mvc\ModelInterface* $model, *string* $attribute) inherited from Phalcon\Mvc\Model\MetaData

Check if a model has certain attribute

```
<?php

var_dump($metaData->hasAttribute(new Robots(), 'name'));
```

public *boolean* **isEmpty** () inherited from Phalcon\Mvc\Model\MetaData

Checks if the internal meta-data container is empty

```
<?php

var_dump($metaData->isEmpty());
```

public **reset** () inherited from Phalcon\Mvc\Model\MetaData

Resets internal meta-data in order to regenerate it

```
<?php

$metaData->reset();
```

2.48.135 Class Phalcon\Mvc\Model\MetaData\Session

extends Phalcon\Mvc\Model\MetaData

implements Phalcon\DI\InjectionAwareInterface, *Phalcon\Mvc\Model\MetaDataInterface*

Stores model meta-data in session. Data will erased when the session finishes. Meta-data are permanent while the session is active. You can query the meta-data by printing $_SESSION['$PMM$']

```
<?php

 $metaData = new Phalcon\Mvc\Model\Metadata\Session(array(
    'prefix' => 'my-app-id'
 ));
```

Constants

integer **MODELS_ATTRIBUTES**

integer **MODELS_PRIMARY_KEY**

integer **MODELS_NON_PRIMARY_KEY**

integer **MODELS_NOT_NULL**

integer **MODELS_DATA_TYPES**

integer **MODELS_DATA_TYPES_NUMERIC**

integer **MODELS_DATE_AT**

integer **MODELS_DATE_IN**

integer **MODELS_IDENTITY_COLUMN**

integer **MODELS_DATA_TYPES_BIND**

integer **MODELS_AUTOMATIC_DEFAULT_INSERT**

integer **MODELS_AUTOMATIC_DEFAULT_UPDATE**

integer **MODELS_COLUMN_MAP**

integer **MODELS_REVERSE_COLUMN_MAP**

Methods

public **__construct** ([*array* $options])

Phalcon\Mvc\Model\MetaData\Session constructor

public *array* **read** (*string* $key)

Reads meta-data from $_SESSION

public **write** (*string* $key, *array* $data)

Writes the meta-data to $_SESSION

protected **_initialize** () inherited from Phalcon\Mvc\Model\MetaData

Initialize the metadata for certain table

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\Mvc\Model\MetaData

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\Mvc\Model\MetaData

Returns the DependencyInjector container

public **setStrategy** (*Phalcon\Mvc\Model\MetaData\Strategy\Introspection* $strategy) inherited from Phalcon\Mvc\Model\MetaData

Set the meta-data extraction strategy

public *Phalcon\Mvc\Model\MetaData\Strategy\Introspection* **getStrategy** () inherited from Phalcon\Mvc\Model\MetaData

Return the strategy to obtain the meta-data

public *array* **readMetaData** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Reads the complete meta-data for certain model

```
<?php

print_r($metaData->readMetaData(new Robots());
```

public **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index) inherited from Phalcon\Mvc\Model\MetaData

Reads meta-data for certain model using a MODEL_* constant

```
<?php

print_r($metaData->writeColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP, array('leNa
```

public **writeMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index, *mixed* $data) inherited from Phalcon\Mvc\Model\MetaData

Writes meta-data for certain model using a MODEL_* constant

```
<?php

print_r($metaData->writeColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP, array('leNa
```

public *array* **readColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Reads the ordered/reversed column map for certain model

```
<?php

print_r($metaData->readColumnMap(new Robots()));
```

public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index) inherited from Phalcon\Mvc\Model\MetaData

Reads column-map information for certain model using a MODEL_* constant

```
<?php

print_r($metaData->readColumnMapIndex(new Robots(), MetaData::MODELS_REVERSE_COLUMN_MAP));
```

public *array* **getAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns table attributes names (fields)

```
<?php

print_r($metaData->getAttributes(new Robots()));
```

public *array* **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an array of fields which are part of the primary key

```
<?php

print_r($metaData->getPrimaryKeyAttributes(new Robots()));
```

public *array* **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an arrau of fields which are not part of the primary key

```
<?php

print_r($metaData->getNonPrimaryKeyAttributes(new Robots()));
```

public *array* **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns an array of not null attributes

```
<?php

print_r($metaData->getNotNullAttributes(new Robots()));
```

public *array* **getDataTypes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes and their data types

```
<?php

print_r($metaData->getDataTypes(new Robots()));
```

public *array* **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes which types are numerical

```
<?php

print_r($metaData->getDataTypesNumeric(new Robots()));
```

public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the name of identity field (if one is present)

```
<?php

print_r($metaData->getIdentityField(new Robots()));
```

public *array* **getBindTypes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes and their bind data types

```
<?php

print_r($metaData->getBindTypes(new Robots()));
```

public *array* **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes that must be ignored from the INSERT SQL generation

```
<?php

print_r($metaData->getAutomaticCreateAttributes(new Robots()));
```

public *array* **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns attributes that must be ignored from the UPDATE SQL generation

```
<?php

print_r($metaData->getAutomaticUpdateAttributes(new Robots()));
```

public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes) inherited from Phalcon\Mvc\Model\MetaData

Set the attributes that must be ignored from the INSERT SQL generation

```
<?php

$metaData->setAutomaticCreateAttributes(new Robots(), array('created_at' => true));
```

public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes) inherited from Phalcon\Mvc\Model\MetaData

Set the attributes that must be ignored from the UPDATE SQL generation

```
<?php

$metaData->setAutomaticUpdateAttributes(new Robots(), array('modified_at' => true));
```

public *array* **getColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the column map if any

```
<?php

print_r($metaData->getColumnMap(new Robots()));
```

public *array* **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* $model) inherited from Phalcon\Mvc\Model\MetaData

Returns the reverse column map if any

```
<?php

print_r($metaData->getReverseColumnMap(new Robots()));
```

public *boolean* **hasAttribute** (*Phalcon\Mvc\ModelInterface* $model, *string* $attribute) inherited from Phalcon\Mvc\Model\MetaData

Check if a model has certain attribute

```
<?php

var_dump($metaData->hasAttribute(new Robots(), 'name'));
```

public *boolean* **isEmpty** () inherited from Phalcon\Mvc\Model\MetaData

Checks if the internal meta-data container is empty

```
<?php

var_dump($metaData->isEmpty());
```

public **reset** () inherited from Phalcon\Mvc\Model\MetaData

Resets internal meta-data in order to regenerate it

```
<?php

$metaData->reset();
```

2.48.136 Class Phalcon\Mvc\Model\MetaData\Strategy\Annotations

Queries the table meta-data in order to instrospect the model's metadata

Methods

public *array* **getMetaData** (*Phalcon\Mvc\ModelInterface* $model, *Phalcon\DiInterface* $dependencyInjector)

The meta-data is obtained by reading the column descriptions from the database information schema

public *array* **getColumnMaps** ()

Read the model's column map, this can't be infered

2.48.137 Class Phalcon\Mvc\Model\MetaData\Strategy\Introspection

Phalcon\Mvc\Model\MetaData\Strategy\Instrospection Queries the table meta-data in order to instrospect the model's metadata

Methods

public *array* **getMetaData** (*Phalcon\Mvc\ModelInterface* $model, *Phalcon\DiInterface* $dependencyInjector)

The meta-data is obtained by reading the column descriptions from the database information schema

public *array* **getColumnMaps** (*Phalcon\Mvc\ModelInterface* $model, *Phalcon\DiInterface* $dependencyInjector)

Read the model's column map, this can't be infered

2.48.138 Class Phalcon\Mvc\Model\Query

implements Phalcon\Mvc\Model\QueryInterface, *Phalcon\DI\InjectionAwareInterface*

This class takes a PHQL intermediate representation and executes it.

```
<?php

 $phql = "SELECT c.price*0.16 AS taxes, c.* FROM Cars AS c JOIN Brands AS b
          WHERE b.name = :name: ORDER BY c.name";

 $result = $manager->executeQuery($phql, array(
   'name' => 'Lamborghini'
 ));

 foreach ($result as $row) {
   echo "Name: ", $row->cars->name, "\n";
   echo "Price: ", $row->cars->price, "\n";
   echo "Taxes: ", $row->taxes, "\n";
 }
```

Constants

integer **TYPE_SELECT**

integer **TYPE_INSERT**

integer **TYPE_UPDATE**

integer **TYPE_DELETE**

Methods

public **__construct** ([*string* $phql])

Phalcon\Mvc\Model\Query constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injection container

public *Phalcon\DiInterface* **getDI** ()

Returns the dependency injection container

public *Phalcon\Mvc\Model\Query* **setUniqueRow** (*boolean* $uniqueRow)

Tells to the query if only the first row in the resultset must be resturned

public *boolean* **getUniqueRow** ()

Check if the query is programmed to get only the first row in the resultset

protected *string* **_getQualified** ()

Replaces the model's name to its source name in a qualifed-name expression

protected *string* **_getCallArgument** ()

Resolves a expression in a single call argument

protected *string* **_getFunctionCall** ()

Resolves a expression in a single call argument

protected *string* **_getExpression** ()

Resolves an expression from its intermediate code into a string

protected *array* **_getSelectColumn** ()

Resolves a column from its intermediate representation into an array used to determine if the resulset produced is simple or complex

protected *string* **_getTable** ()

Resolves a table in a SELECT statement checking if the model exists

protected *array* **_getJoin** ()

Resolves a JOIN clause checking if the associated models exist

protected *string* **_getJoinType** ()

Resolves a JOIN type

protected *array* **_getJoins** ()

Resolves all the JOINS in a SELECT statement

protected *string* **_getOrderClause** ()

Returns a processed order clause for a SELECT statement

protected *string* **_getGroupClause** ()

Returns a processed group clause for a SELECT statement

protected *array* **_prepareSelect** ()

Analyzes a SELECT intermediate code and produces an array to be executed later

protected *array* **_prepareInsert** ()

Analyzes an INSERT intermediate code and produces an array to be executed later

protected *array* **_prepareUpdate** ()

Analyzes an UPDATE intermediate code and produces an array to be executed later

protected *array* **_prepareDelete** ()

Analyzes a DELETE intermediate code and produces an array to be executed later

public *array* **parse** ()

Parses the intermediate code produced by Phalcon\Mvc\Model\Query\Lang generating another intermediate representation that could be executed by Phalcon\Mvc\Model\Query

public *Phalcon\Mvc\Model\Query* **cache** (*array* $cacheOptions)

Sets the cache parameters of the query

public **getCacheOptions** ()

Returns the current cache options

public *Phalcon\Cache\BackendInterface* **getCache** ()

Returns the current cache backend instance

protected *Phalcon\Mvc\Model\ResultsetInterface* **_executeSelect** ()

Executes the SELECT intermediate representation producing a Phalcon\Mvc\Model\Resultset

protected *Phalcon\Mvc\Model\Query\StatusInterface* **_executeInsert** ()

Executes the INSERT intermediate representation producing a Phalcon\Mvc\Model\Query\Status

protected *Phalcon\Mvc\Model\ResultsetInterface* **_getRelatedRecords** ()

Query the records on which the UPDATE/DELETE operation well be done

protected *Phalcon\Mvc\Model\Query\StatusInterface* **_executeUpdate** ()

Executes the UPDATE intermediate representation producing a Phalcon\Mvc\Model\Query\Status

protected *Phalcon\Mvc\Model\Query\StatusInterface* **_executeDelete** ()

Executes the DELETE intermediate representation producing a Phalcon\Mvc\Model\Query\Status

public *mixed* **execute** ([*array* $bindParams], [*array* $bindTypes])

Executes a parsed PHQL statement

public *halconMvcModelInterface* **getSingleResult** ([*array* $bindParams], [*array* $bindTypes])

Executes the query returning the first result

public *Phalcon\Mvc\Model\Query* **setType** (*int* $type)

Sets the type of PHQL statement to be executed

public *int* **getType** ()

Gets the type of PHQL statement executed

public *Phalcon\Mvc\Model\Query* **setIntermediate** (*array* $intermediate)

Allows to set the IR to be executed

public *array* **getIntermediate** ()

Returns the intermediate representation of the PHQL statement

2.48.139 Class Phalcon\Mvc\Model\Query\Builder

implements Phalcon\Mvc\Model\Query\BuilderInterface, *Phalcon\DI\InjectionAwareInterface*

Helps to create PHQL queries using an OO interface

```
<?php

$resultset = $this->modelsManager->createBuilder()
   ->from('Robots')
   ->join('RobotsParts')
   ->limit(20)
   ->orderBy('Robots.name')
   ->getQuery()
   ->execute();
```

Methods

public **__construct** ([*array* $params])

Phalcon\Mvc\Model\Query\Builder constructor

public *Phalcon\Mvc\Model\Query\Builder* **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** ()

Returns the DependencyInjector container

public *Phalcon\Mvc\Model\Query\Builder* **columns** (*string|array* $columns)

Sets the columns to be queried

```
<?php

$builder->columns(array('id', 'name'));
```

public *string|array* **getColumns** ()

Return the columns to be queried

public *Phalcon\Mvc\Model\Query\Builder* **from** (*string|array* $models)

Sets the models who makes part of the query

```
<?php

$builder->from(array('Robots', 'RobotsParts'));
```

public *Phalcon\Mvc\Model\Query\Builder* **addFrom** (*string* $model, [*string* $alias])

Add a model to take part of the query

```
<?php

$builder->addFrom('Robots', 'r');
```

public *string|array* **getFrom** ()

Return the models who makes part of the query

public *Phalcon\Mvc\Model\Query\Builder* **join** (*string* $model, [*string* $conditions], [*string* $alias], [*string* $type])

Adds a join to the query

```
<?php

$builder->join('Robots', 'r.id = RobotsParts.robots_id', 'r');
```

public *Phalcon\Mvc\Model\Query\Builder* **leftJoin** (*string* $model, [*string* $conditions], [*string* $alias])

Adds a LEFT join to the query

```
<?php

$builder->leftJoin('Robots', 'r.id = RobotsParts.robots_id', 'r');
```

public *Phalcon\Mvc\Model\Query\Builder* **rightJoin** (*string* $model, [*string* $conditions], [*string* $alias])

Adds a RIGHT join to the query

```
<?php

$builder->rightJoin('Robots', 'r.id = RobotsParts.robots_id', 'r');
```

public *Phalcon\Mvc\Model\Query\Builder* **where** (*string* $conditions)

Sets the query conditions

```
<?php

$builder->where('name = :name: AND id > :id:');
```

public *Phalcon\Mvc\Model\Query\Builder* **andWhere** (*string* $conditions)

Appends a condition to the current conditions using a AND operator

```
<?php

$builder->andWhere('name = :name: AND id > :id:');
```

public *Phalcon\Mvc\Model\Query\Builder* **orWhere** (*string* $conditions)

Appends a condition to the current conditions using a OR operator

```
<?php

$builder->orWhere('name = :name: AND id > :id:');
```

public *string|array* **getWhere** ()

Return the conditions for the query

public *Phalcon\Mvc\Model\Query\Builder* **orderBy** (*string* $orderBy)

Sets a ORDER BY condition clause

```
<?php

$builder->orderBy('Robots.name');
$builder->orderBy(array('1', 'Robots.name'));
```

public *string|array* **getOrderBy** ()

Returns the set ORDER BY clause

public *Phalcon\Mvc\Model\Query\Builder* **having** (*string* $having)

Sets a HAVING condition clause. You need to escape PHQL reserved words using [and] delimiters

```
<?php

$builder->having('SUM(Robots.price) > 0');
```

public *string|array* **getHaving** ()

Return the current having clause

public *Phalcon\Mvc\Model\Query\Builder* **limit** (*int* $limit, [*int* $offset])

Sets a LIMIT clause, optionally a offset clause

```
<?php

$builder->limit(100);
$builder->limit(100, 20);
```

public *string|array* **getLimit** ()

Returns the current LIMIT clause

public *Phalcon\Mvc\Model\Query\Builder* **offset** (*int* $offset)

Sets an OFFSET clause

```
<?php

$builder->offset(30);
```

public *string|array* **getOffset** ()

Returns the current OFFSET clause

public *Phalcon\Mvc\Model\Query\Builder* **groupBy** (*string* $group)

Sets a GROUP BY clause

```
<?php

$builder->groupBy(array('Robots.name'));
```

public *string* **getGroupBy** ()

Returns the GROUP BY clause

public *string* **getPhql** ()

Returns a PHQL statement built based on the builder parameters

public *Phalcon\Mvc\Model\Query* **getQuery** ()

Returns the query built

2.48.140 Class Phalcon\Mvc\Model\Query\Lang

PHQL is implemented as a parser (written in C) that translates syntax in that of the target RDBMS. It allows Phalcon to offer a unified SQL language to the developer, while internally doing all the work of translating PHQL instructions to the most optimal SQL instructions depending on the RDBMS type associated with a model. To achieve the highest performance possible, we wrote a parser that uses the same technology as SQLite. This technology provides a small in-memory parser with a very low memory footprint that is also thread-safe.

```
<?php

 $intermediate = Phalcon\Mvc\Model\Query\Lang::parsePHQL("SELECT r.* FROM Robots r LIMIT 10");
```

Methods

public static *string* **parsePHQL** (*string* $phql)

Parses a PHQL statement returning an intermediate representation (IR)

2.48.141 Class Phalcon\Mvc\Model\Query\Status

implements Phalcon\Mvc\Model\Query\StatusInterface

This class represents the status returned by a PHQL statement like INSERT, UPDATE or DELETE. It offers context information and the related messages produced by the model which finally executes the operations when it fails

```
<?php

$phql = "UPDATE Robots SET name = :name:, type = :type:, year = :year: WHERE id = :id:";
$status = $app->modelsManager->executeQuery($phql, array(
   'id' => 100,
   'name' => 'Astroy Boy',
   'type' => 'mechanical',
   'year' => 1959
));

\//Check if the update was successful
if ($status->success() == true) {
   echo 'OK';
}
```

Methods

public **__construct** (*boolean* $success, *Phalcon\Mvc\ModelInterface* $model)

public *Phalcon\Mvc\ModelInterface* **getModel** ()

Returns the model that executed the action

public *Phalcon\Mvc\Model\MessageInterface* [] **getMessages** ()

Returns the messages produced by a failed operation

public *boolean* **success** ()

Allows to check if the executed operation was successful

2.48.142 Class Phalcon\Mvc\Model\Relation

implements Phalcon\Mvc\Model\RelationInterface

This class represents each relationship between two models

Constants

integer **BELONGS_TO**

integer **HAS_ONE**

integer **HAS_MANY**

integer **HAS_ONE_THROUGH**

integer **HAS_MANY_THROUGH**

integer **MANY_TO_MANY**

Methods

public **__construct** (*int* $type, *string* $referencedModel, *string|array* $fields, *string|array* $referencedFields, [*array* $options])

Phalcon\Mvc\Model\Relation constructor

public *int* **getType** ()

Returns the relation's type

public *string* **getReferencedModel** ()

Returns the referenced model

public *string|array* **getFields** ()

Returns the fields

public *string|array* **getReferencedFields** ()

Returns the referenced fields

public *string|array* **getOptions** ()

Returns the options

public *string|array* **isForeingKey** ()

Check whether the relation act as a foreign key

public *string|array* **getForeignKey** ()

Returns the foreign key configuration

public *boolean* **hasThrough** ()

Check whether the relation

public *string* **getThrough** ()

Returns the 'through' relation if any

public *boolean* **isReusable** ()

Check if records in belongs-to/has-many are implicitly cached during the current request

2.48.143 Class Phalcon\Mvc\Model\Resultset

implements Phalcon\Mvc\Model\ResultsetInterface, Iterator, Traversable, SeekableIterator, Countable, ArrayAccess, Serializable

This component allows to Phalcon\Mvc\Model returns large resulsets with the minimum memory consumption Resulsets can be traversed using a standard foreach or a while statement. If a resultset is serialized it will dump all the rows into a big array. Then unserialize will retrieve the rows as they were before serializing.

```
<?php

 //Using a standard foreach
 $robots = Robots::find(array("type='virtual'", "order" => "name"));
 foreach ($robots as $robot) {
  echo $robot->name, "\n";
 }

 //Using a while
 $robots = Robots::find(array("type='virtual'", "order" => "name"));
 $robots->rewind();
 while ($robots->valid()) {
  $robot = $robots->current();
  echo $robot->name, "\n";
  $robots->next();
 }
```

Constants

integer **TYPE_RESULT_FULL**

integer **TYPE_RESULT_PARTIAL**

integer **HYDRATE_RECORDS**

integer **HYDRATE_OBJECTS**

integer **HYDRATE_ARRAYS**

Methods

public **next** ()

Moves cursor to next row in the resultset

public *int* **key** ()

Gets pointer number of active row in the resultset

public **rewind** ()

Rewinds resultset to its beginning

public **seek** (*int* $position)

Changes internal pointer to a specific position in the resultset

public *int* **count** ()

Counts how many rows are in the resultset

public *boolean* **offsetExists** (*int* $index)

Checks whether offset exists in the resultset

public *Phalcon\Mvc\ModelInterface* **offsetGet** (*int* $index)

Gets row in a specific position of the resultset

public **offsetSet** (*int* $index, *Phalcon\Mvc\ModelInterface* $value)

Resulsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **offsetUnset** (*int* $offset)

Resulsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public *int* **getType** ()

Returns the internal type of data retrieval that the resultset is using

public *Phalcon\Mvc\ModelInterface* **getFirst** ()

Get first row in the resultset

public *Phalcon\Mvc\ModelInterface* **getLast** ()

Get last row in the resultset

public *Phalcon\Mvc\Model\Resultset* **setIsFresh** (*boolean* $isFresh)

Set if the resultset is fresh or an old one cached

public *boolean* **isFresh** ()

Tell if the resultset if fresh or an old one cached

public *Phalcon\Mvc\Model\Resultset* **setHydrateMode** (*int* $hydrateMode)

Sets the hydration mode in the resultset

public *int* **getHydrateMode** ()

Returns the current hydration mode

public *Phalcon\Cache\BackendInterface* **getCache** ()

Returns the associated cache for the resultset

public *Phalcon\Mvc\ModelInterface* **current** ()

Returns current row in the resultset

public *Phalcon\Mvc\Model\MessageInterface* [] **getMessages** ()

Returns the error messages produced by a batch operation

public *boolean* **delete** ([*Closure* $conditionCallback])

Delete every record in the resultset

abstract public *array* **toArray** () inherited from Phalcon\Mvc\Model\ResultsetInterface

Returns a complete resultset as an array, if the resultset has a big number of rows it could consume more memory than currently it does.

abstract public **valid** () inherited from Iterator

...

abstract public **serialize** () inherited from Serializable

...

abstract public **unserialize** (*unknown* $serialized) inherited from Serializable

...

2.48.144 Class Phalcon\Mvc\Model\Resultset\Complex

extends Phalcon\Mvc\Model\Resultset

implements Serializable, ArrayAccess, Countable, SeekableIterator, Traversable, Iterator, *Phalcon\Mvc\Model\ResultsetInterface*

Complex resultsets may include complete objects and scalar values. This class builds every complex row as the're required

Constants

integer **TYPE_RESULT_FULL**

integer **TYPE_RESULT_PARTIAL**

integer **HYDRATE_RECORDS**

integer **HYDRATE_OBJECTS**

integer **HYDRATE_ARRAYS**

Methods

public **__construct** (*array* $columnsTypes, *Phalcon\Db\ResultInterface* $result, [*Phalcon\Cache\BackendInterface* $cache])

Phalcon\Mvc\Model\Resultset\Complex constructor

public *boolean* **valid** ()

Check whether internal resource has rows to fetch

public *array* **toArray** ()

Returns a complete resultset as an array, if the resultset has a big number of rows it could consume more memory than currently it does.

public *string* **serialize** ()

Serializing a resultset will dump all related rows into a big array

public **unserialize** (*string* $data)

Unserializing a resultset will allow to only works on the rows present in the saved state

public **next** () inherited from Phalcon\Mvc\Model\Resultset

Moves cursor to next row in the resultset

public *int* **key** () inherited from Phalcon\Mvc\Model\Resultset

Gets pointer number of active row in the resultset

public **rewind** () inherited from Phalcon\Mvc\Model\Resultset

Rewinds resultset to its beginning

public **seek** (*int* $position) inherited from Phalcon\Mvc\Model\Resultset

Changes internal pointer to a specific position in the resultset

public *int* **count** () inherited from Phalcon\Mvc\Model\Resultset

Counts how many rows are in the resultset

public *boolean* **offsetExists** (*int* $index) inherited from Phalcon\Mvc\Model\Resultset

Checks whether offset exists in the resultset

public *Phalcon\Mvc\ModelInterface* **offsetGet** (*int* $index) inherited from Phalcon\Mvc\Model\Resultset

Gets row in a specific position of the resultset

public **offsetSet** (*int* $index, *Phalcon\Mvc\ModelInterface* $value) inherited from Phalcon\Mvc\Model\Resultset

Resulsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **offsetUnset** (*int* $offset) inherited from Phalcon\Mvc\Model\Resultset

Resulsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public *int* **getType** () inherited from Phalcon\Mvc\Model\Resultset

Returns the internal type of data retrieval that the resultset is using

public *Phalcon\Mvc\ModelInterface* **getFirst** () inherited from Phalcon\Mvc\Model\Resultset

Get first row in the resultset

public *Phalcon\Mvc\ModelInterface* **getLast** () inherited from Phalcon\Mvc\Model\Resultset

Get last row in the resultset

public *Phalcon\Mvc\Model\Resultset* **setIsFresh** (*boolean* $isFresh) inherited from Phalcon\Mvc\Model\Resultset

Set if the resultset is fresh or an old one cached

public *boolean* **isFresh** () inherited from Phalcon\Mvc\Model\Resultset

Tell if the resultset if fresh or an old one cached

public *Phalcon\Mvc\Model\Resultset* **setHydrateMode** (*int* $hydrateMode) inherited from Phalcon\Mvc\Model\Resultset

Sets the hydration mode in the resultset

public *int* **getHydrateMode** () inherited from Phalcon\Mvc\Model\Resultset

Returns the current hydration mode

public *Phalcon\Cache\BackendInterface* **getCache** () inherited from Phalcon\Mvc\Model\Resultset

Returns the associated cache for the resultset

public *Phalcon\Mvc\ModelInterface* **current** () inherited from Phalcon\Mvc\Model\Resultset

Returns current row in the resultset

public *Phalcon\Mvc\Model\MessageInterface* [] **getMessages** () inherited from Phalcon\Mvc\Model\Resultset

Returns the error messages produced by a batch operation

public *boolean* **delete** ([*Closure* $conditionCallback]) inherited from Phalcon\Mvc\Model\Resultset

Delete every record in the resultset

2.48.145 Class Phalcon\Mvc\Model\Resultset\Simple

extends Phalcon\Mvc\Model\Resultset

implements Serializable, ArrayAccess, Countable, SeekableIterator, Traversable, Iterator, *Phalcon\Mvc\Model\ResultsetInterface*

Simple resultsets only contains a complete objects This class builds every complete object as it is required

Constants

integer **TYPE_RESULT_FULL**

integer **TYPE_RESULT_PARTIAL**

integer **HYDRATE_RECORDS**

integer **HYDRATE_OBJECTS**

integer **HYDRATE_ARRAYS**

Methods

public **__construct** (*array* $columnMap, *Phalcon\Mvc\ModelInterface* $model, *Phalcon\Db\Result\Pdo* $result, [*Phalcon\Cache\BackendInterface* $cache], [*boolean* $keepSnapshots])

Phalcon\Mvc\Model\Resultset\Simple constructor

public *boolean* **valid** ()

Check whether internal resource has rows to fetch

public *array* **toArray** ()

Returns a complete resultset as an array, if the resultset has a big number of rows it could consume more memory than currently it does.

public *string* **serialize** ()

Serializing a resultset will dump all related rows into a big array

public **unserialize** (*string* $data)

Unserializing a resultset will allow to only works on the rows present in the saved state

public **next** () inherited from Phalcon\Mvc\Model\Resultset

Moves cursor to next row in the resultset

public *int* **key** () inherited from Phalcon\Mvc\Model\Resultset

Gets pointer number of active row in the resultset

public **rewind** () inherited from Phalcon\Mvc\Model\Resultset

Rewinds resultset to its beginning

public **seek** (*int* $position) inherited from Phalcon\Mvc\Model\Resultset

Changes internal pointer to a specific position in the resultset

public *int* **count** () inherited from Phalcon\Mvc\Model\Resultset

Counts how many rows are in the resultset

public *boolean* **offsetExists** (*int* $index) inherited from Phalcon\Mvc\Model\Resultset

Checks whether offset exists in the resultset

public *Phalcon\Mvc\ModelInterface* **offsetGet** (*int* $index) inherited from Phalcon\Mvc\Model\Resultset

Gets row in a specific position of the resultset

public **offsetSet** (*int* $index, *Phalcon\Mvc\ModelInterface* $value) inherited from Phalcon\Mvc\Model\Resultset

Resulsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **offsetUnset** (*int* $offset) inherited from Phalcon\Mvc\Model\Resultset

Resulsets cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public *int* **getType** () inherited from Phalcon\Mvc\Model\Resultset

Returns the internal type of data retrieval that the resultset is using

public *Phalcon\Mvc\ModelInterface* **getFirst** () inherited from Phalcon\Mvc\Model\Resultset

Get first row in the resultset

public *Phalcon\Mvc\ModelInterface* **getLast** () inherited from Phalcon\Mvc\Model\Resultset

Get last row in the resultset

public *Phalcon\Mvc\Model\Resultset* **setIsFresh** (*boolean* $isFresh) inherited from Phalcon\Mvc\Model\Resultset

Set if the resultset is fresh or an old one cached

public *boolean* **isFresh** () inherited from Phalcon\Mvc\Model\Resultset

Tell if the resultset if fresh or an old one cached

public *Phalcon\Mvc\Model\Resultset* **setHydrateMode** (*int* $hydrateMode) inherited from Phalcon\Mvc\Model\Resultset

Sets the hydration mode in the resultset

public *int* **getHydrateMode** () inherited from Phalcon\Mvc\Model\Resultset

Returns the current hydration mode

public *Phalcon\Cache\BackendInterface* **getCache** () inherited from Phalcon\Mvc\Model\Resultset

Returns the associated cache for the resultset

public *Phalcon\Mvc\ModelInterface* **current** () inherited from Phalcon\Mvc\Model\Resultset

Returns current row in the resultset

public *Phalcon\Mvc\Model\MessageInterface* [] **getMessages** () inherited from Phalcon\Mvc\Model\Resultset

Returns the error messages produced by a batch operation

public *boolean* **delete** ([*Closure* $conditionCallback]) inherited from Phalcon\Mvc\Model\Resultset

Delete every record in the resultset

2.48.146 Class Phalcon\Mvc\Model\Row

implements ArrayAccess, *Phalcon\Mvc\Model\ResultInterface*

This component allows Phalcon\Mvc\Model to return rows without an associated entity. This objects implements the ArrayAccess interface to allow access the object as object->x or array[x].

Methods

public **setDirtyState** (*int* $dirtyState)

Set the current object's state

public *boolean* **offsetExists** (*int* $index)

Checks whether offset exists in the row

public *string|PhalconMvcModelInterface* **offsetGet** (*int* $index)

Gets row in a specific position of the row

public **offsetSet** (*int* $index, *Phalcon\Mvc\ModelInterface* $value)

Rows cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

public **offsetUnset** (*int* $offset)

Rows cannot be changed. It has only been implemented to meet the definition of the ArrayAccess interface

2.48.147 Class Phalcon\Mvc\Model\Transaction

implements Phalcon\Mvc\Model\TransactionInterface

Transactions are protective blocks where SQL statements are only permanent if they can all succeed as one atomic action. Phalcon\Transaction is intended to be used with Phalcon_Model_Base. Phalcon Transactions should be created using Phalcon\Transaction\Manager.

```php
<?php

try {

  $manager = new Phalcon\Mvc\Model\Transaction\Manager();

  $transaction = $manager->get();

  $robot = new Robots();
  $robot->setTransaction($transaction);
  $robot->name = 'WALL·E';
  $robot->created_at = date('Y-m-d');
  if ($robot->save() == false) {
    $transaction->rollback("Can't save robot");
  }

  $robotPart = new RobotParts();
  $robotPart->setTransaction($transaction);
  $robotPart->type = 'head';
  if ($robotPart->save() == false) {
    $transaction->rollback("Can't save robot part");
  }

  $transaction->commit();

} catch(Phalcon\Mvc\Model\Transaction\Failed $e) {
  echo 'Failed, reason: ', $e->getMessage();
}
```

Methods

public **__construct** (*Phalcon\DiInterface* $dependencyInjector, [*boolean* $autoBegin], [*string* $service])

Phalcon\Mvc\Model\Transaction constructor

public **setTransactionManager** (*Phalcon\Mvc\Model\Transaction\ManagerInterface* $manager)

Sets transaction manager related to the transaction

public *boolean* **begin** ()

Starts the transaction

public *boolean* **commit** ()

Commits the transaction

public *boolean* **rollback** ([*string* $rollbackMessage], [*Phalcon\Mvc\ModelInterface* $rollbackRecord])

Rollbacks the transaction

public *Phalcon\Db\AdapterInterface* **getConnection** ()

Returns the connection related to transaction

public **setIsNewTransaction** (*boolean* $isNew)

Sets if is a reused transaction or new once

public **setRollbackOnAbort** (*boolean* $rollbackOnAbort)

Sets flag to rollback on abort the HTTP connection

public *boolean* **isManaged** ()

Checks whether transaction is managed by a transaction manager

public *array* **getMessages** ()

Returns validations messages from last save try

public *boolean* **isValid** ()

Checks whether internal connection is under an active transaction

public **setRollbackedRecord** (*Phalcon\Mvc\ModelInterface* $record)

Sets object which generates rollback action

2.48.148 Class Phalcon\Mvc\Model\Transaction\Exception

extends Phalcon\Mvc\Model\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.149 Class Phalcon\Mvc\Model\Transaction\Failed

extends Exception

This class will be thrown to exit a try/catch block for isolated transactions

Methods

public **__construct** (*string* $message, *Phalcon\Mvc\ModelInterface* $record)

Phalcon\Mvc\Model\Transaction\Failed constructor

public *Phalcon\Mvc\Model\MessageInterface* [] **getRecordMessages** ()

Returns validation record messages which stop the transaction

public *Phalcon\Mvc\ModelInterface* **getRecord** ()

Returns validation record messages which stop the transaction

final private *Exception* **__clone** () inherited from Exception

Clone the exception

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.150 Class Phalcon\Mvc\Model\Transaction\Manager

implements Phalcon\Mvc\Model\Transaction\ManagerInterface, *Phalcon\DI\InjectionAwareInterface*

A transaction acts on a single database connection. If you have multiple class-specific databases, the transaction will not protect interaction among them. This class manages the objects that compose a transaction. A trasaction produces a unique connection that is passed to every object part of the transaction.

```
<?php

try {

  use Phalcon\Mvc\Model\Transaction\Manager as TransactionManager;

  $transactionManager = new TransactionManager();

  $transaction = $transactionManager->get();

  $robot = new Robots();
  $robot->setTransaction($transaction);
  $robot->name = 'WALL·E';
  $robot->created_at = date('Y-m-d');
  if($robot->save()==false){
    $transaction->rollback("Can't save robot");
  }

  $robotPart = new RobotParts();
  $robotPart->setTransaction($transaction);
  $robotPart->type = 'head';
  if($robotPart->save()==false){
    $transaction->rollback("Can't save robot part");
  }

  $transaction->commit();

}
catch(Phalcon\Mvc\Model\Transaction\Failed $e){
  echo 'Failed, reason: ', $e->getMessage();
}
```

Methods

public **__construct** ([*Phalcon\DiInterface* $dependencyInjector])

Phalcon\Mvc\Model\Transaction\Manager constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injection container

public *Phalcon\DiInterface* **getDI** ()

Returns the dependency injection container

public *Phalcon\Mvc\Model\Transaction\Manager* **setDbService** (*string* $service)

Sets the database service used to run the isolated transactions

public *string* **getDbService** ()

Returns the database service used to isolate the transaction

public *Phalcon\Mvc\Model\Transaction\Manager* **setRollbackPendent** (*boolean* $rollbackPendent)

Set if the transaction manager must register a shutdown function to clean up pendent transactions

public *boolean* **getRollbackPendent** ()

Check if the transaction manager is registering a shutdown function to clean up pendent transactions

public *boolean* **has** ()

Checks whether the manager has an active transaction

public *Phalcon\Mvc\Model\TransactionInterface* **get** ([*boolean* $autoBegin])

Returns a new Phalcon\Mvc\Model\Transaction or an already created once This method registers a shutdown function to rollback active connections

public *Phalcon\Mvc\Model\TransactionInterface* **getOrCreateTransaction** ([*boolean* $autoBegin])

Create/Returns a new transaction or an existing one

public **rollbackPendent** ()

Rollbacks active transactions within the manager

public **commit** ()

Commmits active transactions within the manager

public **rollback** ([*boolean* $collect])

Rollbacks active transactions within the manager Collect will remove transaction from the manager

public **notifyRollback** (*Phalcon\Mvc\Model\TransactionInterface* $transaction)

Notifies the manager about a rollbacked transaction

public **notifyCommit** (*Phalcon\Mvc\Model\TransactionInterface* $transaction)

Notifies the manager about a commited transaction

protected **_collectTransaction** ()

Removes transactions from the TransactionManager

public **collectTransactions** ()

Remove all the transactions from the manager

2.48.151 Class Phalcon\Mvc\Model\Validator

This is a base class for Phalcon\Mvc\Model validators

Methods

public **__construct** (*array* $options)

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** ()

Appends a message to the validator

public *array* **getMessages** ()

Returns messages generated by the validator

protected *array* **getOptions** ()

Returns all the options from the validator

protected *mixed* **getOption** ()

Returns an option

protected *boolean* **isSetOption** ()

Check whether a option has been defined in the validator options

2.48.152 Class Phalcon\Mvc\Model\Validator\Email

extends Phalcon\Mvc\Model\Validator

implements Phalcon\Mvc\Model\ValidatorInterface

Allows to validate if email fields has correct values

```
<?php

use Phalcon\Mvc\Model\Validator\Email as EmailValidator;

class Subscriptors extends Phalcon\Mvc\Model
{

    public function validation()
    {
            $this->validate(new EmailValidator(array(
                    'field' => 'electronic_mail'
            )));
            if ($this->validationHasFailed() == true) {
                    return false;
            }
    }

}
```

Methods

public *boolean* **validate** (*Phalcon\Mvc\ModelInterface* $record)

Executes the validator

public **__construct** (*array* $options) inherited from Phalcon\Mvc\Model\Validator

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** () inherited from Phalcon\Mvc\Model\Validator

Appends a message to the validator

public *array* **getMessages** () inherited from Phalcon\Mvc\Model\Validator

Returns messages generated by the validator

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Validator

Returns all the options from the validator

protected *mixed* **getOption** () inherited from Phalcon\Mvc\Model\Validator

Returns an option

protected *boolean* **isSetOption** () inherited from Phalcon\Mvc\Model\Validator

Check whether a option has been defined in the validator options

2.48.153 Class Phalcon\Mvc\Model\Validator\Exclusionin

extends Phalcon\Mvc\Model\Validator

implements Phalcon\Mvc\Model\ValidatorInterface

Phalcon\Mvc\Model\Validator\ExclusionIn Check if a value is not included into a list of values

```
<?php

use Phalcon\Mvc\Model\Validator\ExclusionIn as ExclusionInValidator;

class Subscriptors extends Phalcon\Mvc\Model
{

    public function validation()
    {
            $this->validate(new ExclusionInValidator(array(
                    'field' => 'status',
                    'domain' => array('A', 'I')
            )));
            if ($this->validationHasFailed() == true) {
                    return false;
            }
    }

}
```

Methods

public *boolean* **validate** (*Phalcon\Mvc\ModelInterface* $record)

Executes the validator

public **__construct** (*array* $options) inherited from Phalcon\Mvc\Model\Validator

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** () inherited from Phalcon\Mvc\Model\Validator

Appends a message to the validator

public *array* **getMessages** () inherited from Phalcon\Mvc\Model\Validator

Returns messages generated by the validator

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Validator

Returns all the options from the validator

protected *mixed* **getOption** () inherited from Phalcon\Mvc\Model\Validator

Returns an option

protected *boolean* **isSetOption** () inherited from Phalcon\Mvc\Model\Validator

Check whether a option has been defined in the validator options

2.48.154 Class Phalcon\Mvc\Model\Validator\Inclusionin

extends Phalcon\Mvc\Model\Validator

implements Phalcon\Mvc\Model\ValidatorInterface

Phalcon\Mvc\Model\Validator\InclusionIn Check if a value is included into a list of values

```
<?php

use Phalcon\Mvc\Model\Validator\InclusionIn as InclusionInValidator;

class Subscriptors extends Phalcon\Mvc\Model
{

    public function validation()
    {
            $this->validate(new InclusionInValidator(array(
                    'field' => 'status',
                    'domain' => array('A', 'I')
            )));
            if ($this->validationHasFailed() == true) {
                    return false;
            }
    }

}
```

Methods

public *boolean* **validate** (*Phalcon\Mvc\ModelInterface* $record)

Executes validator

public **__construct** (*array* $options) inherited from Phalcon\Mvc\Model\Validator

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** () inherited from Phalcon\Mvc\Model\Validator

Appends a message to the validator

public *array* **getMessages** () inherited from Phalcon\Mvc\Model\Validator

Returns messages generated by the validator

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Validator

Returns all the options from the validator

protected *mixed* **getOption** () inherited from Phalcon\Mvc\Model\Validator

Returns an option

protected *boolean* **isSetOption** () inherited from Phalcon\Mvc\Model\Validator

Check whether a option has been defined in the validator options

2.48.155 Class Phalcon\Mvc\Model\Validator\Numericality

extends Phalcon\Mvc\Model\Validator

implements Phalcon\Mvc\Model\ValidatorInterface

Allows to validate if a field has a valid numeric format

```
<?php

use Phalcon\Mvc\Model\Validator\Numericality as NumericalityValidator;

class Products extends Phalcon\Mvc\Model
{

  public function validation()
  {
      $this->validate(new NumericalityValidator(array(
          'field' => 'price'
      )));
      if ($this->validationHasFailed() == true) {
          return false;
      }
  }

}
```

Methods

public *boolean* **validate** (*Phalcon\Mvc\ModelInterface* $record)

Executes the validator

public **__construct** (*array* $options) inherited from Phalcon\Mvc\Model\Validator

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** () inherited from Phalcon\Mvc\Model\Validator

Appends a message to the validator

public *array* **getMessages** () inherited from Phalcon\Mvc\Model\Validator

Returns messages generated by the validator

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Validator

Returns all the options from the validator

protected *mixed* **getOption** () inherited from Phalcon\Mvc\Model\Validator

Returns an option

protected *boolean* **isSetOption** () inherited from Phalcon\Mvc\Model\Validator

Check whether a option has been defined in the validator options

2.48.156 Class Phalcon\Mvc\Model\Validator\PresenceOf

extends Phalcon\Mvc\Model\Validator

implements Phalcon\Mvc\Model\ValidatorInterface

Allows to validate if a filed have a value different of null and empty string (“”)

```
<?php

use Phalcon\Mvc\Model\Validator\PresenceOf;
```

```
class Subscriptors extends Phalcon\Mvc\Model
{

  public function validation()
  {
      $this->validate(new PresenceOf(array(
          'field' => 'name',
          'message' => 'The name is required'
      )));
      if ($this->validationHasFailed() == true) {
          return false;
      }
  }

}
```

Methods

public *boolean* **validate** (*Phalcon\Mvc\ModelInterface* $record)

Executes the validator

public **__construct** (*array* $options) inherited from Phalcon\Mvc\Model\Validator

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** () inherited from Phalcon\Mvc\Model\Validator

Appends a message to the validator

public *array* **getMessages** () inherited from Phalcon\Mvc\Model\Validator

Returns messages generated by the validator

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Validator

Returns all the options from the validator

protected *mixed* **getOption** () inherited from Phalcon\Mvc\Model\Validator

Returns an option

protected *boolean* **isSetOption** () inherited from Phalcon\Mvc\Model\Validator

Check whether a option has been defined in the validator options

2.48.157 Class Phalcon\Mvc\Model\Validator\Regex

extends Phalcon\Mvc\Model\Validator

implements Phalcon\Mvc\Model\ValidatorInterface

Allows validate if the value of a field matches a regular expression

```
<?php

use Phalcon\Mvc\Model\Validator\Regex as RegexValidator;

class Subscriptors extends Phalcon\Mvc\Model
{
```

```
  public function validation()
  {
      $this->validate(new RegexValidator(array(
          'field' => 'created_at',
          'pattern' => '/^[0-9]{4}[-\/](0[1-9]|1[12])[-\/](0[1-9]|[12][0-9]|3[01])$/'
      )));
      if ($this->validationHasFailed() == true) {
          return false;
      }
  }

}
```

Methods

public *boolean* **validate** (*Phalcon\Mvc\ModelInterface* $record)

Executes the validator

public **__construct** (*array* $options) inherited from Phalcon\Mvc\Model\Validator

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** () inherited from Phalcon\Mvc\Model\Validator

Appends a message to the validator

public *array* **getMessages** () inherited from Phalcon\Mvc\Model\Validator

Returns messages generated by the validator

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Validator

Returns all the options from the validator

protected *mixed* **getOption** () inherited from Phalcon\Mvc\Model\Validator

Returns an option

protected *boolean* **isSetOption** () inherited from Phalcon\Mvc\Model\Validator

Check whether a option has been defined in the validator options

2.48.158 Class Phalcon\Mvc\Model\Validator\StringLength

extends Phalcon\Mvc\Model\Validator

implements Phalcon\Mvc\Model\ValidatorInterface

Simply validates specified string length constraints

```
<?php

use Phalcon\Mvc\Model\Validator\StringLength as StringLengthValidator;

class Subscriptors extends Phalcon\Mvc\Model
{

public function validation()
{
    $this->validate(new StringLengthValidator(array(
```

```
            'field' => 'name_last',
            'max' => 50,
            'min' => 2,
          'messageMaximum' => 'We don't like really long names',
          'messageMinimum' => 'We want more than just their initials'
    )));
    if ($this->validationHasFailed() == true) {
            return false;
    }
}

}
```

Methods

public *boolean* **validate** (*Phalcon\Mvc\ModelInterface* $record)

Executes the validator

public **__construct** (*array* $options) inherited from Phalcon\Mvc\Model\Validator

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** () inherited from Phalcon\Mvc\Model\Validator

Appends a message to the validator

public *array* **getMessages** () inherited from Phalcon\Mvc\Model\Validator

Returns messages generated by the validator

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Validator

Returns all the options from the validator

protected *mixed* **getOption** () inherited from Phalcon\Mvc\Model\Validator

Returns an option

protected *boolean* **isSetOption** () inherited from Phalcon\Mvc\Model\Validator

Check whether a option has been defined in the validator options

2.48.159 Class Phalcon\Mvc\Model\Validator\Uniqueness

extends Phalcon\Mvc\Model\Validator

implements Phalcon\Mvc\Model\ValidatorInterface

Validates that a field or a combination of a set of fields are not present more than once in the existing records of the related table

```
<?php

use Phalcon\Mvc\Model\Validator\Uniqueness as UniquenessValidator;

class Subscriptors extends Phalcon\Mvc\Model
{

  public function validation()
  {
```

```
        $this->validate(new UniquenessValidator(array(
            'field' => 'email'
        )));
        if ($this->validationHasFailed() == true) {
            return false;
        }
    }

}
```

Methods

public *boolean* **validate** (*Phalcon\Mvc\ModelInterface* $record)

Executes the validator

public **__construct** (*array* $options) inherited from Phalcon\Mvc\Model\Validator

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** () inherited from Phalcon\Mvc\Model\Validator

Appends a message to the validator

public *array* **getMessages** () inherited from Phalcon\Mvc\Model\Validator

Returns messages generated by the validator

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Validator

Returns all the options from the validator

protected *mixed* **getOption** () inherited from Phalcon\Mvc\Model\Validator

Returns an option

protected *boolean* **isSetOption** () inherited from Phalcon\Mvc\Model\Validator

Check whether a option has been defined in the validator options

2.48.160 Class Phalcon\Mvc\Model\Validator\Url

extends Phalcon\Mvc\Model\Validator

implements Phalcon\Mvc\Model\ValidatorInterface

Allows to validate if a field has a url format

```
<?php

use Phalcon\Mvc\Model\Validator\Url as UrlValidator;

class Posts extends Phalcon\Mvc\Model
{

    public function validation()
    {
        $this->validate(new UrlValidator(array(
            'field' => 'source_url'
        )));
        if ($this->validationHasFailed() == true) {
```

```
            return false;
        }
    }

}
```

Methods

public *boolean* **validate** (*Phalcon\Mvc\ModelInterface* $record)

Executes the validator

public **__construct** (*array* $options) inherited from Phalcon\Mvc\Model\Validator

Phalcon\Mvc\Model\Validator constructor

protected **appendMessage** () inherited from Phalcon\Mvc\Model\Validator

Appends a message to the validator

public *array* **getMessages** () inherited from Phalcon\Mvc\Model\Validator

Returns messages generated by the validator

protected *array* **getOptions** () inherited from Phalcon\Mvc\Model\Validator

Returns all the options from the validator

protected *mixed* **getOption** () inherited from Phalcon\Mvc\Model\Validator

Returns an option

protected *boolean* **isSetOption** () inherited from Phalcon\Mvc\Model\Validator

Check whether a option has been defined in the validator options

2.48.161 Class Phalcon\Mvc\Router

implements Phalcon\Mvc\RouterInterface, *Phalcon\DI\InjectionAwareInterface*

Phalcon\Mvc\Router is the standard framework router. Routing is the process of taking a URI endpoint (that part of the URI which comes after the base URL) and decomposing it into parameters to determine which module, controller, and action of that controller should receive the request

```
<?php

$router = new Phalcon\Mvc\Router();

  $router->add(
    "/documentation/{chapter}/{name}.{type:[a-z]+}",
    array(
            "controller" => "documentation",
            "action"     => "show"
    )
);

$router->handle();

echo $router->getControllerName();
```

Constants

integer **URI_SOURCE_GET_URL**

integer **URI_SOURCE_SERVER_REQUEST_URI**

Methods

public **__construct** ([*boolean* $defaultRoutes])

Phalcon\Mvc\Router constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

protected *string* **_getRewriteUri** ()

Get rewrite info. This info is read from $_GET['_url']. This returns '/' if the rewrite information cannot be read

public **setUriSource** (*string* $uriSource)

Sets the URI source. One of the URI_SOURCE_* constants

```
<?php

$router->setUriSource(Router::URI_SOURCE_SERVER_REQUEST_URI);
```

public **removeExtraSlashes** (*boolean* $remove)

Set whether router must remove the extra slashes in the handled routes

public **setDefaultNamespace** (*string* $namespaceName)

Sets the name of the default namespace

public **setDefaultModule** (*string* $moduleName)

Sets the name of the default module

public **setDefaultController** (*string* $controllerName)

Sets the default controller name

public **setDefaultAction** (*string* $actionName)

Sets the default action name

public **setDefaults** (*array* $defaults)

Sets an array of default paths. If a route is missing a path the router will use the defined here This method must not be used to set a 404 route

```
<?php

 $router->setDefaults(array(
    'module' => 'common',
    'action' => 'index'
 ));
```

public **handle** ([*string* $uri])

Handles routing information received from the rewrite engine

```
<?php

 //Read the info from the rewrite engine
 $router->handle();

 //Manually passing an URL
 $router->handle('/posts/edit/1');
```

public *Phalcon\Mvc\Router\Route* **add** (*string* $pattern, [*string/array* $paths], [*string* $httpMethods])

Adds a route to the router without any HTTP constraint

```
<?php

 $router->add('/about', 'About::index');
```

public *Phalcon\Mvc\Router\Route* **addGet** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is GET

public *Phalcon\Mvc\Router\Route* **addPost** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is POST

public *Phalcon\Mvc\Router\Route* **addPut** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is PUT

public *Phalcon\Mvc\Router\Route* **addPatch** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is PATCH

public *Phalcon\Mvc\Router\Route* **addDelete** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is DELETE

public *Phalcon\Mvc\Router\Route* **addOptions** (*string* $pattern, [*string/array* $paths])

Add a route to the router that only match if the HTTP method is OPTIONS

public *Phalcon\Mvc\Router\Route* **addHead** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is HEAD

public **mount** (*unknown* $group)

Mounts a group of routes in the router

public **notFound** (*unknown* $paths)

A set of paths used to

public **clear** ()

Removes all the pre-defined routes

public *string* **getNamespaceName** ()

Returns the processed namespace name

public *string* **getModuleName** ()

Returns the processed module name

public *string* **getControllerName** ()

Returns the processed controller name

public *string* **getActionName** ()

Returns the processed action name

public *array* **getParams** ()

Returns the processed parameters

public *Phalcon\Mvc\Router\Route* **getMatchedRoute** ()

Returns the route that matchs the handled URI

public *array* **getMatches** ()

Returns the sub expressions in the regular expression matched

public *bool* **wasMatched** ()

Checks if the router macthes any of the defined routes

public *Phalcon\Mvc\Router\Route* [] **getRoutes** ()

Returns all the routes defined in the router

public *Phalcon\Mvc\Router\Route* **getRouteById** (*string* $id)

Returns a route object by its id

public *Phalcon\Mvc\Router\Route* **getRouteByName** (*string* $name)

Returns a route object by its name

2.48.162 Class Phalcon\Mvc\Router\Annotations

extends Phalcon\Mvc\Router

implements Phalcon\DI\InjectionAwareInterface, *Phalcon\Mvc\RouterInterface*

A router that reads routes annotations from classes/resources

```
<?php

 $di['router'] = function() {

    //Use the annotations router
    $router = new \Phalcon\Mvc\Router\Annotations(false);

    //This will do the same as above but only if the handled uri starts with /robots
            $router->addResource('Robots', '/robots');

            return $router;
};
```

Constants

integer **URI_SOURCE_GET_URL**

integer **URI_SOURCE_SERVER_REQUEST_URI**

Methods

public *Phalcon\Mvc\Router\Annotations* **addResource** (*string* $handler, [*string* $prefix])

Adds a resource to the annotations handler A resource is a class that contains routing annotations

public *Phalcon\Mvc\Router\Annotations* **addModuleResource** (*string* $module, *string* $handler, [*string* $prefix])

Adds a resource to the annotations handler A resource is a class that contains routing annotations The class is located in a module

public **handle** ([*string* $uri])

Produce the routing parameters from the rewrite information

public **processControllerAnnotation** (*string* $handler, *unknown* $annotation)

Checks for annotations in the controller docblock

public **processActionAnnotation** (*string* $module, *string* $namespace, *string* $controller, *string* $action, *Phalcon\Annotations\Annotation* $annotation)

Checks for annotations in the public methods of the controller

public **setControllerSuffix** (*string* $controllerSuffix)

Changes the controller class suffix

public **setActionSuffix** (*string* $actionSuffix)

Changes the action method suffix

public *array* **getResources** ()

Return the registered resources

public **__construct** ([*boolean* $defaultRoutes]) inherited from Phalcon\Mvc\Router

Phalcon\Mvc\Router constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\Mvc\Router

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\Mvc\Router

Returns the internal dependency injector

protected *string* **_getRewriteUri** () inherited from Phalcon\Mvc\Router

Get rewrite info. This info is read from $_GET['_url']. This returns '/' if the rewrite information cannot be read

public **setUriSource** (*string* $uriSource) inherited from Phalcon\Mvc\Router

Sets the URI source. One of the URI_SOURCE_* constants

```
<?php

$router->setUriSource(Router::URI_SOURCE_SERVER_REQUEST_URI);
```

public **removeExtraSlashes** (*boolean* $remove) inherited from Phalcon\Mvc\Router

Set whether router must remove the extra slashes in the handled routes

public **setDefaultNamespace** (*string* $namespaceName) inherited from Phalcon\Mvc\Router

Sets the name of the default namespace

public **setDefaultModule** (*string* $moduleName) inherited from Phalcon\Mvc\Router

Sets the name of the default module

public **setDefaultController** (*string* $controllerName) inherited from Phalcon\Mvc\Router

Sets the default controller name

public **setDefaultAction** (*string* $actionName) inherited from Phalcon\Mvc\Router

Sets the default action name

public **setDefaults** (*array* $defaults) inherited from Phalcon\Mvc\Router

Sets an array of default paths. If a route is missing a path the router will use the defined here This method must not be used to set a 404 route

```
<?php

 $router->setDefaults(array(
    'module' => 'common',
    'action' => 'index'
 ));
```

public *Phalcon\Mvc\Router\Route* **add** (*string* $pattern, [*string/array* $paths], [*string* $httpMethods]) inherited from Phalcon\Mvc\Router

Adds a route to the router without any HTTP constraint

```
<?php

 $router->add('/about', 'About::index');
```

public *Phalcon\Mvc\Router\Route* **addGet** (*string* $pattern, [*string/array* $paths]) inherited from Phalcon\Mvc\Router

Adds a route to the router that only match if the HTTP method is GET

public *Phalcon\Mvc\Router\Route* **addPost** (*string* $pattern, [*string/array* $paths]) inherited from Phalcon\Mvc\Router

Adds a route to the router that only match if the HTTP method is POST

public *Phalcon\Mvc\Router\Route* **addPut** (*string* $pattern, [*string/array* $paths]) inherited from Phalcon\Mvc\Router

Adds a route to the router that only match if the HTTP method is PUT

public *Phalcon\Mvc\Router\Route* **addPatch** (*string* $pattern, [*string/array* $paths]) inherited from Phalcon\Mvc\Router

Adds a route to the router that only match if the HTTP method is PATCH

public *Phalcon\Mvc\Router\Route* **addDelete** (*string* $pattern, [*string/array* $paths]) inherited from Phalcon\Mvc\Router

Adds a route to the router that only match if the HTTP method is DELETE

public *Phalcon\Mvc\Router\Route* **addOptions** (*string* $pattern, [*string/array* $paths]) inherited from Phalcon\Mvc\Router

Add a route to the router that only match if the HTTP method is OPTIONS

public *Phalcon\Mvc\Router\Route* **addHead** (*string* $pattern, [*string/array* $paths]) inherited from Phalcon\Mvc\Router

Adds a route to the router that only match if the HTTP method is HEAD

public **mount** (*unknown* $group) inherited from Phalcon\Mvc\Router

Mounts a group of routes in the router

public **notFound** (*unknown* $paths) inherited from Phalcon\Mvc\Router

A set of paths used to

public **clear** () inherited from Phalcon\Mvc\Router

Removes all the pre-defined routes

public *string* **getNamespaceName** () inherited from Phalcon\Mvc\Router

Returns the processed namespace name

public *string* **getModuleName** () inherited from Phalcon\Mvc\Router

Returns the processed module name

public *string* **getControllerName** () inherited from Phalcon\Mvc\Router

Returns the processed controller name

public *string* **getActionName** () inherited from Phalcon\Mvc\Router

Returns the processed action name

public *array* **getParams** () inherited from Phalcon\Mvc\Router

Returns the processed parameters

public *Phalcon\Mvc\Router\Route* **getMatchedRoute** () inherited from Phalcon\Mvc\Router

Returns the route that matchs the handled URI

public *array* **getMatches** () inherited from Phalcon\Mvc\Router

Returns the sub expressions in the regular expression matched

public *bool* **wasMatched** () inherited from Phalcon\Mvc\Router

Checks if the router macthes any of the defined routes

public *Phalcon\Mvc\Router\Route* [] **getRoutes** () inherited from Phalcon\Mvc\Router

Returns all the routes defined in the router

public *Phalcon\Mvc\Router\Route* **getRouteById** (*string* $id) inherited from Phalcon\Mvc\Router

Returns a route object by its id

public *Phalcon\Mvc\Router\Route* **getRouteByName** (*string* $name) inherited from Phalcon\Mvc\Router

Returns a route object by its name

2.48.163 Class Phalcon\Mvc\Router\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.164 Class Phalcon\Mvc\Router\Group

Helper class to create a group of routes with common attributes

```
<?php

 $router = new Phalcon\Mvc\Router();

 //Create a group with a common module and controller
 $blog = new Phalcon\Mvc\Router\Group(array(
    'module' => 'blog',
    'controller' => 'index'
 ));

 //All the routes start with /blog
 $blog->setPrefix('/blog');

 //Add a route to the group
 $blog->add('/save', array(
    'action' => 'save'
 ));

 //Add another route to the group
 $blog->add('/edit/{id}', array(
    'action' => 'edit'
 ));

 //This route maps to a controller different than the default
 $blog->add('/blog', array(
    'controller' => 'about',
    'action' => 'index'
 ));
```

```
//Add the group to the router
$router->mount($blog);
```

Methods

public **__construct** ([*array* $paths])

Phalcon\Mvc\Router\Group constructor

public *Phalcon\Mvc\Router\Group* **setPrefix** (*string* $prefix)

Set a common uri prefix for all the routes in this group

public *string* **getPrefix** ()

Returns the common prefix for all the routes

public *Phalcon\Mvc\Router\Group* **setPaths** (*array* $paths)

Set common paths for all the routes in the group

public *array|string* **getPaths** ()

Returns the common paths defined for this group

public *Phalcon\Mvc\Router\Route* [] **getRoutes** ()

Returns the routes added to the group

protected *Phalcon\Mvc\Router\Route* **_addRoute** ()

Adds a route applying the common attributes

public *Phalcon\Mvc\Router\Route* **add** (*string* $pattern, [*string/array* $paths], [*string* $httpMethods])

Adds a route to the router on any HTTP method

```
<?php

$router->add('/about', 'About::index');
```

public *Phalcon\Mvc\Router\Route* **addGet** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is GET

public *Phalcon\Mvc\Router\Route* **addPost** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is POST

public *Phalcon\Mvc\Router\Route* **addPut** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is PUT

public *Phalcon\Mvc\Router\Route* **addPatch** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is PATCH

public *Phalcon\Mvc\Router\Route* **addDelete** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is DELETE

public *Phalcon\Mvc\Router\Route* **addOptions** (*string* $pattern, [*string/array* $paths])

Add a route to the router that only match if the HTTP method is OPTIONS

public *Phalcon\Mvc\Router\Route* **addHead** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is HEAD

public **clear** ()

Removes all the pre-defined routes

2.48.165 Class Phalcon\Mvc\Router\Route

implements Phalcon\Mvc\Router\RouteInterface

This class represents every route added to the router

Methods

public **__construct** (*string* $pattern, [*array* $paths], [*array*|*string* $httpMethods])

Phalcon\Mvc\Router\Route constructor

public *string* **compilePattern** (*string* $pattern)

Replaces placeholders from pattern returning a valid PCRE regular expression

public *Phalcon\Mvc\Router\RouteInterface* **via** (*string*|*array* $httpMethods)

Set one or more HTTP methods that constraint the matching of the route

public **reConfigure** (*string* $pattern, [*array* $paths])

Reconfigure the route adding a new pattern and a set of paths

public *string* **getName** ()

Returns the route's name

public *Phalcon\Mvc\Router\RouteInterface* **setName** (*string* $name)

Sets the route's name

public *Phalcon\Mvc\Router\RouteInterface* **setHttpMethods** (*string*|*array* $httpMethods)

Sets a set of HTTP methods that constraint the matching of the route

public *string* **getRouteId** ()

Returns the route's id

public *string* **getPattern** ()

Returns the route's pattern

public *string* **getCompiledPattern** ()

Returns the route's compiled pattern

public *array* **getPaths** ()

Returns the paths

public *array* **getReversedPaths** ()

Returns the paths using positions as keys and names as values

public *string*|*array* **getHttpMethods** ()

Returns the HTTP methods that constraint matching the route

public *Phalcon\Mvc\Router\RouteInterface* **convert** (*string* $name, *callable* $converter)

Adds a converter to perform an additional transformation for certain parameter

public *array* **getConverters** ()

Returns the router converter

public static **reset** ()

Resets the internal route id generator

2.48.166 Class Phalcon\Mvc\Url

implements Phalcon\Mvc\UrlInterface, Phalcon\DI\InjectionAwareInterface

This components aids in the generation of: URIs, URLs and Paths

```
<?php

 //Generate a url appending a uri to the base Uri
 echo $url->get('products/edit/1');

 //Generate a url for a predefined route
 echo $url->get(array('for' => 'blog-post', 'title' => 'some-cool-stuff', 'year' => '2012'));
```

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** ()

Returns the DependencyInjector container

public **setBaseUri** (*string* $baseUri)

Sets a prefix to all the urls generated

```
<?php

$url->setBaseUri('/invo/');
```

public *string* **getBaseUri** ()

Returns the prefix for all the generated urls. By default /

public **setBasePath** (*string* $basePath)

Sets a base paths for all the generated paths

```
<?php

$url->setBasePath('/var/www/');
```

public *string* **getBasePath** ()

Returns a base path

public *string* **get** ([*string|array* $uri])

Generates a URL

public *string* **path** ([*string* $path])

Generates a local path

2.48.167 Class Phalcon\Mvc\Url\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.168 Class Phalcon\Mvc\User\Component

extends Phalcon\DINjectable

implements Phalcon\Events\EventsAwareInterface, *Phalcon\DINjectionAwareInterface*

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI\Injectable

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

public **__get** (*string* $propertyName) inherited from Phalcon\DI\Injectable

Magic method __get

2.48.169 Class Phalcon\Mvc\User\Module

extends Phalcon\DI\Injectable

implements Phalcon\Events\EventsAwareInterface, *Phalcon\DI\InjectionAwareInterface*

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI\Injectable

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

public **__get** (*string* $propertyName) inherited from Phalcon\DI\Injectable

Magic method __get

2.48.170 Class Phalcon\Mvc\User\Plugin

extends Phalcon\DI\Injectable

implements Phalcon\Events\EventsAwareInterface, *Phalcon\DI\InjectionAwareInterface*

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI\Injectable

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

public **__get** (*string* $propertyName) inherited from Phalcon\DI\Injectable

Magic method __get

2.48.171 Class Phalcon\Mvc\View

extends Phalcon\DI\Injectable

implements Phalcon\Events\EventsAwareInterface, *Phalcon\DI\InjectionAwareInterface*, *Phalcon\Mvc\ViewInterface*

Phalcon\Mvc\View is a class for working with the "view" portion of the model-view-controller pattern. That is, it exists to help keep the view script separate from the model and controller scripts. It provides a system of helpers, output filters, and variable escaping.

```
<?php

 //Setting views directory
 $view = new Phalcon\Mvc\View();
 $view->setViewsDir('app/views/');

 $view->start();
 //Shows recent posts view (app/views/posts/recent.phtml)
 $view->render('posts', 'recent');
 $view->finish();

 //Printing views output
 echo $view->getContent();
```

Constants

integer **LEVEL_MAIN_LAYOUT**

integer **LEVEL_AFTER_TEMPLATE**

integer **LEVEL_LAYOUT**

integer **LEVEL_BEFORE_TEMPLATE**

integer **LEVEL_ACTION_VIEW**

integer **LEVEL_NO_RENDER**

Methods

public **__construct** ([*array* $options])

Phalcon\Mvc\View constructor

public **setViewsDir** (*string* $viewsDir)

Sets views directory. Depending of your platform, always add a trailing slash or backslash

public *string* **getViewsDir** ()

Gets views directory

public **setLayoutsDir** (*string* $layoutsDir)

Sets the layouts sub-directory. Must be a directory under the views directory. Depending of your platform, always add a trailing slash or backslash

```
<?php

 $view->setLayoutsDir('../common/layouts/');
```

public *string* **getLayoutsDir** ()

Gets the current layouts sub-directory

public **setPartialsDir** (*string* $partialsDir)

Sets a partials sub-directory. Must be a directory under the views directory. Depending of your platform, always add a trailing slash or backslash

```
<?php

*
 $view->setPartialsDir('../common/partials/');
```

public *string* **getPartialsDir** ()

Gets the current partials sub-directory

public **setBasePath** (*string* $basePath)

Sets base path. Depending of your platform, always add a trailing slash or backslash

```
<?php

    $view->setBasePath(__DIR__.'/');
```

public **setRenderLevel** (*string* $level)

Sets the render level for the view

```
<?php

    //Render the view related to the controller only
    $this->view->setRenderLevel(View::LEVEL_VIEW);
```

public **disableLevel** (*int|array* $level)

Disables an specific level of rendering

```
<?php

 //Render all levels except ACTION level
 $this->view->disableLevel(View::LEVEL_ACTION_VIEW);
```

public **setMainView** (*string* $viewPath)

Sets default view name. Must be a file without extension in the views directory

```
<?php

    //Renders as main view views-dir/inicio.phtml
    $this->view->setMainView('inicio');
```

public *string* **getMainView** ()

Returns the name of the main view

public **setLayout** (*string* $layout)

Change the layout to be used instead of using the name of the latest controller name

```
<?php

    $this->view->setLayout('main');
```

public *string* **getLayout** ()

Returns the name of the main view

public **setTemplateBefore** (*string|array* $templateBefore)

Appends template before controller layout

public **cleanTemplateBefore** ()

Resets any template before layouts

public **setTemplateAfter** (*string|array* $templateAfter)

Appends template after controller layout

public **cleanTemplateAfter** ()

Resets any template before layouts

public **setParamToView** (*string* $key, *mixed* $value)

Adds parameters to views (alias of setVar)

```
<?php

$this->view->setParamToView('products', $products);
```

public **setVars** (*array* $params, [*boolean* $merge])

Set all the render params

```
<?php

$this->view->setVars(array('products' => $products));
```

public **setVar** (*string* $key, *mixed* $value)

Set a single view parameter

```
<?php

$this->view->setVar('products', $products);
```

public *mixed* **getVar** (*string* $key)

Returns a parameter previously set in the view

public *array* **getParamsToView** ()

Returns parameters to views

public *string* **getControllerName** ()

Gets the name of the controller rendered

public *string* **getActionName** ()

Gets the name of the action rendered

public *array* **getParams** ()

Gets extra parameters of the action rendered

public **start** ()

Starts rendering process enabling the output buffering

protected *array* **_loadTemplateEngines** ()

Loads registered template engines, if none is registered it will use Phalcon\Mvc\View\Engine\Php

protected **_engineRender** ()

Checks whether view exists on registered extensions and render it

public **registerEngines** (*array* $engines)

Register templating engines

```
<?php

$this->view->registerEngines(array(
  ".phtml" => "Phalcon\Mvc\View\Engine\Php",
  ".volt" => "Phalcon\Mvc\View\Engine\Volt",
  ".mhtml" => "MyCustomEngine"
));
```

public **render** (*string* $controllerName, *string* $actionName, [*array* $params])

Executes render process from dispatching data

```
<?php

 $view->start();
 //Shows recent posts view (app/views/posts/recent.phtml)
 $view->render('posts', 'recent');
 $view->finish();
```

public **pick** (*string* $renderView)

Choose a different view to render instead of last-controller/last-action

```
<?php

 class ProductsController extends Phalcon\Mvc\Controller
 {

    public function saveAction()
    {

         //Do some save stuff...

         //Then show the list view
         $this->view->pick("products/list");
    }
 }
```

public *string* **partial** (*string* $partialPath)

Renders a partial view

```
<?php

    //Show a partial inside another view
    $this->partial('shared/footer');
```

public *string* **getRender** (*string* $controllerName, *string* $actionName, [*array* $params])

Perform the automatic rendering returning the output as a string

```
<?php

    $template = $this->view->getRender('products', 'show', array('products' => $products));
```

public **finish** ()

Finishes the render process by stopping the output buffering

protected *Phalcon\Cache\BackendInterface* **_createCache** ()

Create a Phalcon\Cache based on the internal cache options

public *boolean* **isCaching** ()

Check if the component is currently caching the output content

public *Phalcon\Cache\BackendInterface* **getCache** ()

Returns the cache instance used to cache

public **cache** ([*boolean|array* $options])

Cache the actual view render to certain level

public **setContent** (*string* $content)

Externally sets the view content

```
<?php

$this->view->setContent("<h1>hello</h1>");
```

public *string* **getContent** ()

Returns cached ouput from another view stage

public *string* **getActiveRenderPath** ()

Returns the path of the view that is currently rendered

public **disable** ()

Disables the auto-rendering process

public **enable** ()

Enables the auto-rendering process

public **reset** ()

Resets the view component to its factory default values

public **__set** (*string* $key, *mixed* $value)

Magic method to pass variables to the views

```
<?php

$this->view->products = $products;
```

public *mixed* **__get** (*string* $key)

Magic method to retrieve a variable passed to the view

```
<?php

echo $this->view->products;
```

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI\Injectable

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

2.48.172 Class Phalcon\Mvc\View\Engine

extends Phalcon\DI\Injectable

implements Phalcon\Events\EventsAwareInterface, *Phalcon\DI\InjectionAwareInterface*

All the template engine adapters must inherit this class. This provides basic interfacing between the engine and the Phalcon\Mvc\View component.

Methods

public **__construct** (*Phalcon\Mvc\ViewInterface* $view, [*Phalcon\DiInterface* $dependencyInjector])

Phalcon\Mvc\View\Engine constructor

public *array* **getContent** ()

Returns cached ouput on another view stage

public *string* **partial** (*string* $partialPath)

Renders a partial inside another view

public *Phalcon\Mvc\ViewInterface* **getView** ()

Returns the view component related to the adapter

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI\Injectable

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

public **__get** (*string* $propertyName) inherited from Phalcon\DI\Injectable

Magic method __get

2.48.173 Class Phalcon\Mvc\View\Engine\Php

extends Phalcon\Mvc\View\Engine

implements Phalcon\DI\InjectionAwareInterface, Phalcon\Events\EventsAwareInterface, Phalcon\Mvc\View\EngineInterface

Adapter to use PHP itself as templating engine

Methods

public **render** (*string* $path, *array* $params, [*boolean* $mustClean])

Renders a view using the template engine

public **__construct** (*Phalcon\Mvc\ViewInterface* $view, [*Phalcon\DiInterface* $dependencyInjector]) inherited from Phalcon\Mvc\View\Engine

Phalcon\Mvc\View\Engine constructor

public *array* **getContent** () inherited from Phalcon\Mvc\View\Engine

Returns cached ouput on another view stage

public *string* **partial** (*string* $partialPath) inherited from Phalcon\Mvc\View\Engine

Renders a partial inside another view

public *Phalcon\Mvc\ViewInterface* **getView** () inherited from Phalcon\Mvc\View\Engine

Returns the view component related to the adapter

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI\Injectable

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

public **__get** (*string* $propertyName) inherited from Phalcon\DI\Injectable

Magic method __get

2.48.174 Class Phalcon\Mvc\View\Engine\Volt

extends Phalcon\Mvc\View\Engine

implements Phalcon\DI\InjectionAwareInterface, Phalcon\Events\EventsAwareInterface, Phalcon\Mvc\View\EngineInterface

Designer friendly and fast template engine for PHP written in C

Methods

public **setOptions** (*array* $options)

Set Volt's options

public *array* **getOptions** ()

Return Volt's options

public *Phalcon\Mvc\View\Engine\Volt\Compiler* **getCompiler** ()

Returns the Volt's compiler

public **render** (*string* $templatePath, *array* $params, [*boolean* $mustClean])

Renders a view using the template engine

public *int* **length** (*mixed* $item)

Length filter. If an array/object is passed a count is performed otherwise a strlen/mb_strlen

public *boolean* **isIncluded** (*mixed* $needle, *mixed* $haystack)

Checks if the needle is included in the haystack

public *string* **converEncoding** (*string* $text, *string* $from, *string* $to)

Performs a string conversion

public **slice** (*mixed* $value, *unknown* $start, [*unknown* $end])

Extracts a slice from an string/array/traversable object value

public **__construct** (*Phalcon\Mvc\ViewInterface* $view, [*Phalcon\DiInterface* $dependencyInjector]) inherited from Phalcon\Mvc\View\Engine

Phalcon\Mvc\View\Engine constructor

public *array* **getContent** () inherited from Phalcon\Mvc\View\Engine

Returns cached ouput on another view stage

public *string* **partial** (*string* $partialPath) inherited from Phalcon\Mvc\View\Engine

Renders a partial inside another view

public *Phalcon\Mvc\ViewInterface* **getView** () inherited from Phalcon\Mvc\View\Engine

Returns the view component related to the adapter

public **setDI** (*Phalcon\DiInterface* $dependencyInjector) inherited from Phalcon\DI\Injectable

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** () inherited from Phalcon\DI\Injectable

Returns the internal dependency injector

public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager) inherited from Phalcon\DI\Injectable

Sets the event manager

public *Phalcon\Events\ManagerInterface* **getEventsManager** () inherited from Phalcon\DI\Injectable

Returns the internal event manager

public **__get** (*string* $propertyName) inherited from Phalcon\DI\Injectable

Magic method __get

2.48.175 Class Phalcon\Mvc\View\Engine\Volt\Compiler

implements Phalcon\DINjectionAwareInterface

This class reads and compiles Volt templates into PHP plain code

```
<?php

$compiler = new \Phalcon\Mvc\View\Engine\Volt\Compiler();
$compiler->compile('views/partials/header.volt');
require $compiler->getCompiledTemplatePath();
```

Methods

public **__construct** ([*Phalcon\Mvc\ViewInterface* $view])

public **setOptions** (*array* $options)

Sets the compiler options

public *array* **getOptions** ()

Returns the compiler options

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

public *Phalcon\Mvc\View\Engine\Volt\Compiler* **addFunction** (*string* $name, *Closure|string* $definition)

Register a new function in the compiler

public *array* **getFunctions** ()

Register the user registered functions

public *Phalcon\Mvc\View\Engine\Volt\Compiler* **addFilter** (*string* $name, *Closure|string* $definition)

Register a new filter in the compiler

public *array* **getFilters** ()

Register the user registered filters

public **setUniquePrefix** (*string* $prefix)

Set a unique prefix to be used as prefix for compiled variables

public *string* **getUniquePrefix** ()

Return a unique prefix to be used as prefix for compiled variables and contexts

public *string* **attributeReader** (*array* $expr)

Resolves attribute reading

public *string* **functionCall** (*array* $expr)

Resolves function intermediate code into PHP function calls

public *string* **resolveTest** (*array* $test, *string* $left)

Resolves filter intermediate code into a valid PHP expression

protected *string* **resolveFilter** ()

Resolves filter intermediate code into PHP function calls

public *string* **expression** (*array* $expr)

Resolves an expression node in an AST volt tree

protected **_statementListOrExtends** ()

...

public *string* **compileForeach** (*array* $statement, [*boolean* $extendsMode])

Compiles a 'foreach' intermediate code representation into plain PHP code

public *string* **compileForElse** ()

Generates a 'forelse' PHP code

public **compileIf** (*unknown* $statement, [*unknown* $extendsMode])

...

public **compileElseIf** (*unknown* $statement)

...

public **compileCache** (*unknown* $statement, [*unknown* $extendsMode])

...

public **compileEcho** (*unknown* $statement)

...

public **compileInclude** (*unknown* $statement)

...

public **compileSet** (*unknown* $statement)

public **compileDo** (*unknown* $statement)

public **compileAutoEscape** (*unknown* $statement, *unknown* $extendsMode)

...

protected *string* **_statementList** ()

Traverses a statement list compiling each of its nodes

protected *string* **_compileSource** ()

Compiles a Volt source code returning a PHP plain version

public *string* **compileString** (*string* $viewCode, [*boolean* $extendsMode])

Compiles a template into a string

```
<?php

 echo $compiler->compileString('{{ "hello world" }}');
```

public *string|array* **compileFile** (*string* $path, *string* $compiledPath, [*boolean* $extendsMode])

Compiles a template into a file forcing the destination path

```
<?php

$compiler->compile('views/layouts/main.volt', 'views/layouts/main.volt.php');
```

public *string|array* **compile** (*string* $templatePath, [*boolean* $extendsMode])

Compiles a template into a file applying the compiler options This method does not return the compiled path if the template was not compiled

```
<?php

$compiler->compile('views/layouts/main.volt');
require $compiler->getCompiledTemplatePath();
```

public *string* **getTemplatePath** ()

Returns the path that is currently beign compiled

public *string* **getCompiledTemplatePath** ()

Returns the path to the last compiled template

public *array* **parse** (*string* $viewCode)

Parses a Volt template returning its intermediate representation

```
<?php

print_r($compiler->parse('{{ 3 + 2 }}'));
```

2.48.176 Class Phalcon\Mvc\View\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.177 Class Phalcon\Paginator\Adapter\Model

implements Phalcon\Paginator\AdapterInterface

This adapter allows to paginate data using a Phalcon\Mvc\Model resultset as base

Methods

public **__construct** (*array* $config)

Phalcon\Paginator\Adapter\Model constructor

public **setCurrentPage** (*int* $page)

Set the current page number

public *stdClass* **getPaginate** ()

Returns a slice of the resultset to show in the pagination

2.48.178 Class Phalcon\Paginator\Adapter\NativeArray

implements Phalcon\Paginator\AdapterInterface

Component of pagination by array data

Methods

public **__construct** (*array* $config)

Phalcon\Paginator\Adapter\NativeArray constructor

public **setCurrentPage** (*int* $page)

Set the current page number

public *stdClass* **getPaginate** ()

Returns a slice of the resultset to show in the pagination

2.48.179 Class Phalcon\Paginator\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.180 Class Phalcon\Queue\Beanstalk

Class to access the beanstalk queue service. Partially implements the protocol version 1.2

Methods

public **__construct** ([*array* $options])

public **connect** ()

...

public **put** (*string* $data, [*array* $options])

Inserts jobs into the queue

public *boolean|PhalconQueueBeanstalkJob* **reserve** ([*unknown* $timeout])

public *string|boolean* **choose** (*string* $tube)

Change the active tube. By default the tube is 'default'

public *string|boolean* **watch** (*string* $tube)

Change the active tube. By default the tube is 'default'

public *boolean|PhalconQueueBeanstalkJob* **peekReady** ()

Inspect the next ready job.

protected **readStatus** ()

...

public *string|boolean Data or 'false' on error.* **read** ([*unknown* $length])

Reads a packet from the socket. Prior to reading from the socket will check for availability of the connection.

protected *integer|boolean* **write** ()

Writes data to the socket. Performs a connection if none is available

public *boolean* **disconnect** ()

Closes the connection to the beanstalk server.

2.48.181 Class Phalcon\Queue\Beanstalk\Job

Phalcon\Queue\Beanstalk\Job initializer

Methods

public **__construct** (*unknown* $queue, *unknown* $id, *unknown* $body)

Phalcon\Queue\Beanstalk\Job constructor

public *boolean* **delete** ()

Removes a job from the server entirely

2.48.182 Class Phalcon\Security

implements Phalcon\DINjectionAwareInterface

This component provides a set of functions to improve the security in Phalcon applications

Methods

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

public **setRandomBytes** (*string* $randomBytes)

Sets a number of bytes to be generated by the openssl pseudo random generator

public *string* **getRandomBytes** ()

Returns a number of bytes to be generated by the openssl pseudo random generator

public **setWorkFactor** (*int* $workFactor)

Sets the default working factor for bcrypts password's salts

public *int* **getWorkFactor** ()

Returns the default working factor for bcrypts password's salts

public *string* **getSaltBytes** ()

Generate a >22-length pseudo random string to be used as salt for passwords

public *string* **hash** (*string* $password, [*int* $workFactor])

Creates a password hash using bcrypt with a pseudo random salt

public *boolean* **checkHash** (*string* $password, *string* $passwordHash)

Checks a plain text password and its hash version to check if the password matches

public *boolean* **isLegacyHash** (*string* $passwordHash)

Checks a plain text password and its hash version to check if the password matches

public *string* **getTokenKey** ([*int* $numberBytes])

Generates a pseudo random token key to be used as input's name in a CSRF check

public *string* **getToken** ([*int* $numberBytes])

Generates a pseudo random token value to be used as input's value in a CSRF check

public *boolean* **checkToken** ([*string* $tokenKey], [*string* $tokenValue])

Check if the CSRF token sent in the request is the same that the current in session

public *string* **getSessionToken** ()

Returns the value of the CSRF token in session

2.48.183 Class Phalcon\Security\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.184 Class Phalcon\Session

2.48.185 Class Phalcon\Session\Adapter

Base class for Phalcon\Session adapters

Methods

public **__construct** ([*array* $options])

Phalcon\Session\Adapter constructor

public *boolean* **start** ()

Starts the session (if headers are already sent the session will not started)

public **setOptions** (*array* $options)

Sets session's options

```
<?php

$session->setOptions(array(
    'uniqueId' => 'my-private-app'
));
```

public *array* **getOptions** ()

Get internal options

public *mixed* **get** (*string* $index, [*mixed* $defaultValue])

Gets a session variable from an application context

public **set** (*string* $index, *string* $value)

Sets a session variable in an application context

```
<?php

$session->set('auth', 'yes');
```

public **has** (*string* $index)

Check whether a session variable is set in an application context

```
<?php

var_dump($session->has('auth'));
```

public **remove** (*string* $index)

Removes a session variable from an application context

```
<?php

$session->remove('auth');
```

public *string* **getId** ()

Returns active session id

```
<?php

echo $session->getId();
```

public *boolean* **isStarted** ()

Check whether the session has been started

```
<?php

var_dump($session->isStarted());
```

public *boolean* **destroy** ()

Destroys the active session

```
<?php

var_dump($session->destroy());
```

2.48.186 Class Phalcon\Session\Adapter\Files

extends Phalcon\Session\Adapter

implements Phalcon\Session\AdapterInterface

Methods

public **__construct** ([*array* $options]) inherited from Phalcon\Session\Adapter

Phalcon\Session\Adapter constructor

public *boolean* **start** () inherited from Phalcon\Session\Adapter

Starts the session (if headers are already sent the session will not started)

public **setOptions** (*array* $options) inherited from Phalcon\Session\Adapter

Sets session's options

```
<?php

$session->setOptions(array(
    'uniqueId' => 'my-private-app'
));
```

public *array* **getOptions** () inherited from Phalcon\Session\Adapter

Get internal options

public *mixed* **get** (*string* $index, [*mixed* $defaultValue]) inherited from Phalcon\Session\Adapter

Gets a session variable from an application context

public **set** (*string* $index, *string* $value) inherited from Phalcon\Session\Adapter

Sets a session variable in an application context

```
<?php

$session->set('auth', 'yes');
```

public **has** (*string* $index) inherited from Phalcon\Session\Adapter

Check whether a session variable is set in an application context

```
<?php

var_dump($session->has('auth'));
```

public **remove** (*string* $index) inherited from Phalcon\Session\Adapter

Removes a session variable from an application context

```
<?php

$session->remove('auth');
```

public *string* **getId** () inherited from Phalcon\Session\Adapter

Returns active session id

```
<?php

echo $session->getId();
```

public *boolean* **isStarted** () inherited from Phalcon\Session\Adapter

Check whether the session has been started

```
<?php

var_dump($session->isStarted());
```

public *boolean* **destroy** () inherited from Phalcon\Session\Adapter

Destroys the active session

```
<?php

var_dump($session->destroy());
```

2.48.187 Class Phalcon\Session\Bag

implements Phalcon\DINjectionAwareInterface, *Phalcon\Session\BagInterface*

This component helps to separate session data into "namespaces". Working by this way you can easily create groups of session variables into the application

```
<?php

$user = new \Phalcon\Session\Bag();
$user->name = "Kimbra Johnson";
$user->age = 22;
```

Methods

public **__construct** (*string* $name)

Phalcon\Session\Bag constructor

public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the DependencyInjector container

public *Phalcon\DiInterface* **getDI** ()

Returns the DependencyInjector container

public **initialize** ()

Initializes the session bag. This method must not be called directly, the class calls it when its internal data is accesed

public **destroy** ()

Destroyes the session bag

```
<?php

 $user->destroy();
```

public **set** (*string* $property, *string* $value)

Sets a value in the session bag

```
<?php

 $user->set('name', 'Kimbra');
```

public **__set** (*string* $property, *string* $value)

Magic setter to assign values to the session bag

```
<?php

 $user->name = Kimbra;
```

public *mixed* **get** (*string* $property, [*string* $defaultValue])

Obtains a value from the session bag optionally setting a default value

```
<?php

 echo $user->get('name', 'Kimbra');
```

public *string* **__get** (*string* $property)

Magic getter to obtain values from the session bag

```
<?php

 echo $user->name;
```

public *boolean* **has** (*string* $property)

Check whether a property is defined in the internal bag

```
<?php

 var_dump($user->has('name'));
```

public *boolean* **__isset** (*string* $property)

Magic isset to check whether a property is defined in the bag

```
<?php

 var_dump(isset($user['name']));
```

public *boolean* **remove** (*string* $property)

Removes a property from the internal bag

```
<?php

 $user->remove('name'));
```

public *boolean* **__unset** (*string* $property)

Magic unset to remove items using the array syntax

```
<?php

 unset($user['name']);
```

2.48.188 Class Phalcon\Session\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.189 Class Phalcon\Tag

Phalcon\Tag is designed to simplify building of HTML tags. It provides a set of helpers to generate HTML in a dynamic way. This component is an abstract class that you can extend to add more helpers.

Constants

integer **HTML32**

integer **HTML401_STRICT**

integer **HTML401_TRANSITIONAL**

integer **HTML401_FRAMESET**

integer **HTML5**

integer **XHTML10_STRICT**

integer **XHTML10_TRANSITIONAL**

integer **XHTML10_FRAMESET**

integer **XHTML11**

integer **XHTML20**

integer **XHTML5**

Methods

public static **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector container.

public static *Phalcon\DiInterface* **getDI** ()

Internally gets the request dispatcher

public static *Phalcon\Mvc\UrlInterface* **getUrlService** ()

Return a URL service from the default DI

public static *Phalcon\Mvc\DispatcherInterface* **getDispatcherService** ()

Returns a Dispatcher service from the default DI

public static *Phalcon\EscaperInterface* **getEscaperService** ()

Returns an Escaper service from the default DI

public static **setAutoescape** (*boolean* $autoescape)

Set autoescape mode in generated html

public static **setDefault** (*string* $id, *string* $value)

Assigns default values to generated tags by helpers

```
<?php

 //Assigning "peter" to "name" component
 Phalcon\Tag::setDefault("name", "peter");

 //Later in the view
 echo Phalcon\Tag::textField("name"); //Will have the value "peter" by default
```

public static **setDefaults** (*array* $values)

Assigns default values to generated tags by helpers

```
<?php

 //Assigning "peter" to "name" component
 Phalcon\Tag::setDefaults(array("name" => "peter"));

 //Later in the view
 echo Phalcon\Tag::textField("name"); //Will have the value "peter" by default
```

public static **displayTo** (*string* $id, *string* $value)

Alias of Phalcon\Tag::setDefault

public static *boolean* **hasValue** (*string* $name)

Check if a helper has a default value set using Phalcon\Tag::setDefault or value from $_POST

public static *mixed* **getValue** (*string* $name, [*array* $params])

Every helper calls this function to check whether a component has a predefined value using Phalcon\Tag::setDefault or value from $_POST

public static **resetInput** ()

Resets the request and internal values to avoid those fields will have any default value

public static *string* **linkTo** (*array|string* $parameters, [*string* $text])

Builds a HTML A tag using framework conventions

```
<?php

echo Phalcon\Tag::linkTo('signup/register', 'Register Here!');
```

protected static *string* **_inputField** ()

Builds generic INPUT tags

public static *string* **textField** (*array* $parameters)

Builds a HTML input[type="text"] tag

```
<?php

echo Phalcon\Tag::textField(array("name", "size" => 30))
```

public static *string* **passwordField** (*array* $parameters)

Builds a HTML input[type="password"] tag

```
<?php

 echo Phalcon\Tag::passwordField(array("name", "size" => 30))
```

public static *string* **hiddenField** (*array* $parameters)

Builds a HTML input[type="hidden"] tag

```
<?php

 echo Phalcon\Tag::hiddenField(array("name", "value" => "mike"))
```

public static *string* **fileField** (*array* $parameters)

Builds a HTML input[type="file"] tag

```
<?php

 echo Phalcon\Tag::fileField("file")
```

public static *string* **checkField** (*array* $parameters)

Builds a HTML input[type="check"] tag

```
<?php

 echo Phalcon\Tag::checkField(array("name", "size" => 30))
```

public static *string* **radioField** (*array* $parameters)

Builds a HTML input[type="radio"] tag

```
<?php

 echo Phalcon\Tag::radioField(array("name", "size" => 30))
```

public static *string* **imageInput** (*array* $parameters)

Builds a HTML input[type="image"] tag

```
<?php

 echo Phalcon\Tag::imageInput(array("src" => "/img/button.png"));
```

public static *string* **submitButton** (*array* $parameters)

Builds a HTML input[type="submit"] tag

```
<?php

 echo Phalcon\Tag::submitButton("Save")
```

public static *string* **selectStatic** (*array* $parameters, [*array* $data])

Builds a HTML SELECT tag using a PHP array for options

```
<?php

echo Phalcon\Tag::selectStatic("status", array("A" => "Active", "I" => "Inactive"))
```

public static *string* **select** (*array* $parameters, [*array* $data])

Builds a HTML SELECT tag using a Phalcon\Mvc\Model resultset as options

```
<?php

echo Phalcon\Tag::selectStatic(array(
    "robotId",
    Robots::find("type = 'mechanical'"),
    "using" => array("id", "name")
    ));
```

public static *string* **textArea** (*array* $parameters)

Builds a HTML TEXTAREA tag

```
<?php

 echo Phalcon\Tag::textArea(array("comments", "cols" => 10, "rows" => 4))
```

public static *string* **form** ([*array* $parameters])

Builds a HTML FORM tag

```
<?php

 echo Phalcon\Tag::form("posts/save");
 echo Phalcon\Tag::form(array("posts/save", "method" => "post"));
```

Volt syntax:

```
<?php

 {{ form("posts/save") }}
 {{ form("posts/save", "method": "post") }}
```

public static *string* **endForm** ()

Builds a HTML close FORM tag

public static **setTitle** (*string* $title)

Set the title of view content

```
<?php

 Phalcon\Tag::setTitle('Welcome to my Page');
```

public static **appendTitle** (*string* $title)

Appends a text to current document title

public static **prependTitle** (*string* $title)

Prepends a text to current document title

public static *string* **getTitle** ([*unknown* $tags])

Gets the current document title

```
<?php

    echo Phalcon\Tag::getTitle();
```

```
<?php

    {{ get_title() }}
```

public static *string* **stylesheetLink** ([*array* $parameters], [*boolean* $local])

Builds a LINK[rel="stylesheet"] tag

```
<?php

    echo Phalcon\Tag::stylesheetLink("http://fonts.googleapis.com/css?family=Rosario", false);
    echo Phalcon\Tag::stylesheetLink("css/style.css");
```

public static *string* **javascriptInclude** ([*array* $parameters], [*boolean* $local])

Builds a SCRIPT[type="javascript"] tag

```
<?php

    echo Phalcon\Tag::javascriptInclude("http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.mi
    echo Phalcon\Tag::javascriptInclude("javascript/jquery.js");
```

Volt syntax:

```
<?php

 {{ javascript_include("http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js", false) }}
 {{ javascript_include("javascript/jquery.js") }}
```

public static *string* **image** ([*array* $parameters])

Builds HTML IMG tags

public static *text* **friendlyTitle** (*string* $text, [*string* $separator], [*boolean* $lowercase])

Converts texts into URL-friendly titles

public static **setDocType** (*string* $doctype)

Set the document type of content

public static *string* **getDocType** ()

Get the document type declaration of content

2.48.190 Class Phalcon\Tag\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.191 Class Phalcon\Tag\Select

Generates a SELECT html tag using a static array of values or a Phalcon\Mvc\Model resultset

Methods

public static **selectField** (*array* $parameters, [*array* $data])

Generates a SELECT tag

protected static **_optionsFromResultset** ()

Generate the OPTION tags based on a resulset

protected static **_optionsFromArray** ()

Generate the OPTION tags based on an array

2.48.192 Class Phalcon\Text

Provides utilities when working with strings

Constants

integer **RANDOM_ALNUM**

integer **RANDOM_ALPHA**

integer **RANDOM_HEXDEC**

integer **RANDOM_NUMERIC**

integer **RANDOM_NOZERO**

Methods

public static *string* **camelize** (*string* $str)

Converts strings to camelize style

```
<?php

echo Phalcon\Text::camelize('coco_bongo'); //CocoBongo
```

public static *string* **uncamelize** (*string* $str)

Uncamelize strings which are camelized

```
<?php

echo Phalcon\Text::camelize('CocoBongo'); //coco_bongo
```

public static *string* **increment** (*string* $str, [*string* $separator])

Adds a number to a string or increment that number if it already is defined

```
<?php

echo Phalcon\Text::increment("a"); // "a_1"
  echo Phalcon\Text::increment("a_1"); // "a_2"
```

public static *string* **random** (*int* $type, [*int* $length])

Generates a random string based on the given type. Type is one of the RANDOM_* constants

```
<?php

echo Phalcon\Text::random(Phalcon\Text::RANDOM_ALNUM); //"aloiwkqz"
```

public static **startsWith** (*string* $str, *string* $start, [*boolean* $ignoreCase])

Check if a string starts with a given string

```
<?php

echo Phalcon\Text::startsWith("Hello", "He"); // true
  echo Phalcon\Text::startsWith("Hello", "he"); // false
  echo Phalcon\Text::startsWith("Hello", "he", false); // true
```

public static **endsWith** (*string* $str, *string* $end, [*boolean* $ignoreCase])

Check if a string ends with a given string

```
<?php

echo Phalcon\Text::endsWith("Hello", "llo"); // true
  echo Phalcon\Text::endsWith("Hello", "LLO"); // false
  echo Phalcon\Text::endsWith("Hello", "LLO", false); // true
```

public static *string* **lower** (*string* $str)

Lowecases a string, this function make use of the mbstring extension if available

public static *string* **upper** (*string* $str)

Uppercases a string, this function make use of the mbstring extension if available

2.48.193 Class Phalcon\Translate

2.48.194 Class Phalcon\Translate\Adapter

implements ArrayAccess

Base class for Phalcon\Translate adapters

Methods

public *string* _ (*string* $translateKey, [*array* $placeholders])

Returns the translation string of the given key

public **offsetSet** (*string* $offset, *string* $value)

Sets a translation value

public *boolean* **offsetExists** (*string* $translateKey)

Check whether a translation key exists

public **offsetUnset** (*string* $offset)

Elimina un indice del diccionario

public *string* **offsetGet** (*string* $traslateKey)

Returns the translation related to the given key

2.48.195 Class Phalcon\Translate\Adapter\NativeArray

extends Phalcon\Translate\Adapter

implements ArrayAccess, *Phalcon\Translate\AdapterInterface*

Allows to define translation lists using PHP arrays

Methods

public **__construct** (*array* $options)

Phalcon\Translate\Adapter\NativeArray constructor

public *string* **query** (*string* $index, [*array* $placeholders])

Returns the translation related to the given key

public *bool* **exists** (*string* $index)

Check whether is defined a translation key in the internal array

public *string* _ (*string* $translateKey, [*array* $placeholders]) inherited from Phalcon\Translate\Adapter

Returns the translation string of the given key

public **offsetSet** (*string* $offset, *string* $value) inherited from Phalcon\Translate\Adapter

Sets a translation value

public *boolean* **offsetExists** (*string* $translateKey) inherited from Phalcon\Translate\Adapter

Check whether a translation key exists

public **offsetUnset** (*string* $offset) inherited from Phalcon\Translate\Adapter

Elimina un indice del diccionario

public *string* **offsetGet** (*string* $traslateKey) inherited from Phalcon\Translate\Adapter

Returns the translation related to the given key

2.48.196 Class Phalcon\Translate\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.197 Class Phalcon\Validation

Methods

public **__construct** ([*array* $validators])

Phalcon\Validation constructor

public **validate** (*array|object* $data, [*object* $entity])

Validate a set of data according to a set of rules

public *Phalcon\Validator* **add** (*string* $attribute, *unknown* $validator)

Adds a validator to a field

public *array* **getValidators** ()

Returns the data that is currently validated

public *object* **getEntity** ()

Returns the bound entity

public *Phalcon\Validation\Message\Group* **getMessages** ()

Returns the registered validators

public **appendMessage** (*Phalcon\Validation\MessageInterface* $message)

Appends a message to the messages list

public *Phalcon\Validator* **bind** (*string* $entity, *string* $data)

Assigns the data to an entity The entity is used to obtain the validation values

public *mixed* **getValue** (*string* $attribute)

Gets the a value to validate in the array/object data source

2.48.198 Class Phalcon\Validation\Exception

extends Phalcon\Exception

Methods

final private *Exception* **__clone** () inherited from Exception

Clone the exception

public **__construct** ([*string* $message], [*int* $code], [*Exception* $previous]) inherited from Exception

Exception constructor

final public *string* **getMessage** () inherited from Exception

Gets the Exception message

final public *int* **getCode** () inherited from Exception

Gets the Exception code

final public *string* **getFile** () inherited from Exception

Gets the file in which the exception occurred

final public *int* **getLine** () inherited from Exception

Gets the line in which the exception occurred

final public *array* **getTrace** () inherited from Exception

Gets the stack trace

final public *Exception* **getPrevious** () inherited from Exception

Returns previous Exception

final public *Exception* **getTraceAsString** () inherited from Exception

Gets the stack trace as a string

public *string* **__toString** () inherited from Exception

String representation of the exception

2.48.199 Class Phalcon\Validation\Message

Encapsulates validation info generated in the validation process

Methods

public **__construct** (*string* $message, [*string* $field], [*string* $type])

Phalcon\Validation\Message constructor

public *Phalcon\Mvc\Model\Message* **setType** (*string* $type)

Sets message type

public *string* **getType** ()

Returns message type

public *Phalcon\Mvc\Model\Message* **setMessage** (*string* $message)

Sets verbose message

public *string* **getMessage** ()

Returns verbose message

public *Phalcon\Mvc\Model\Message* **setField** (*string* $field)

Sets field name related to message

public *string* **getField** ()

Returns field name related to message

public *string* **__toString** ()

Magic __toString method returns verbose message

public static *Phalcon\Mvc\Model\Message* **__set_state** (*array* $message)

Magic __set_state helps to recover messsages from serialization

2.48.200 Class Phalcon\Validation\Message\Group

implements Countable, ArrayAccess, Iterator, Traversable

Represents a group of validation messages

Methods

public **__construct** ([*array* $messages])

Phalcon\Validation\Message\Group constructor

public *Phalcon\Validation\Message* **offsetGet** (*string* $index)

Gets an attribute a message using the array syntax

```
<?php

print_r($messages[0]);
```

public **offsetSet** (*string* $index, *Phalcon\Validation\Message* $message)

Sets an attribute using the array-syntax

```
<?php

$messages[0] = new Phalcon\Validation\Message('This is a message');
```

public *boolean* **offsetExists** (*string* $index)

Checks if an index exists

```
<?php

var_dump(isset($message['database']));
```

public **offsetUnset** (*string* $index)

Removes a message from the list

```
<?php

unset($message['database']);
```

public **appendMessage** (*Phalcon\Validation\MessageInterface* $message)

Appends a message to the group

```
<?php

$messages->appendMessage(new Phalcon\Validation\Message('This is a message'));
```

public **appendMessages** (*Phalcon\Validation\MessageInterface[]* $messages)

Appends an array of messages to the group

```
<?php

$messages->appendMessages($messagesArray);
```

public *int* **count** ()

Returns the number of messages in the list

public **rewind** ()

Rewinds the internal iterator

public *Phalcon\Validation\Message* **current** ()

Returns the current message in the iterator

public *int* **key** ()

Returns the current position/key in the iterator

public **next** ()

Moves the internal iteration pointer to the next position

public *boolean* **valid** ()

Check if the current message the iterator is valid

public static *Phalcon\Mvc\Model\Message\Group* **__set_state** (*array* $group)

Magic __set_state helps to re-build messages variable exporting

2.48.201 Class Phalcon\Validation\Validator

This is a base class for validators

Methods

public **__construct** ([*array* $options])

Phalcon\Validation\Validator constructor

public *mixed* **isSetOption** (*string* $key)

Checks if an option is defined

public *mixed* **getOption** (*string* $key)

Returns an option in the validator's options Returns null if the option hasn't been set

2.48.202 Class Phalcon\Validation\Validator\Email

extends Phalcon\Validation\Validator

implements Phalcon\Validation\ValidatorInterface

Checks if a value has a correct e-mail format

```
<?php

use Phalcon\Validation\Validator\Email as EmailValidator;

$validator->add('email', new EmailValidator(array(
   'message' => 'The e-mail is not valid'
)));
```

Methods

public *boolean* **validate** (*Phalcon\Validation* $validator, *string* $attribute)

Executes the validation

public **__construct** ([*array* $options]) inherited from Phalcon\Validation\Validator

Phalcon\Validation\Validator constructor

public *mixed* **isSetOption** (*string* $key) inherited from Phalcon\Validation\Validator

Checks if an option is defined

public *mixed* **getOption** (*string* $key) inherited from Phalcon\Validation\Validator

Returns an option in the validator's options Returns null if the option hasn't been set

2.48.203 Class Phalcon\Validation\Validator\ExclusionIn

extends Phalcon\Validation\Validator

implements Phalcon\Validation\ValidatorInterface

Check if a value is not included into a list of values

```
<?php

use Phalcon\Validation\Validator\ExclusionIn;

$validator->add('status', new ExclusionIn(array(
   'message' => 'The status must not be A or B'
   'domain' => array('A', 'B')
)));
```

Methods

public *boolean* **validate** (*Phalcon\Validation* $validator, *string* $attribute)

Executes the validation

public **__construct** ([*array* $options]) inherited from Phalcon\Validation\Validator

Phalcon\Validation\Validator constructor

public *mixed* **isSetOption** (*string* $key) inherited from Phalcon\Validation\Validator

Checks if an option is defined

public *mixed* **getOption** (*string* $key) inherited from Phalcon\Validation\Validator

Returns an option in the validator's options Returns null if the option hasn't been set

2.48.204 Class Phalcon\Validation\Validator\Identical

extends Phalcon\Validation\Validator

implements Phalcon\Validation\ValidatorInterface

Checks if a value is identical to other

Methods

public *boolean* **validate** (*Phalcon\Validation* $validator, *string* $attribute)

Executes the validation

public **__construct** ([*array* $options]) inherited from Phalcon\Validation\Validator

Phalcon\Validation\Validator constructor

public *mixed* **isSetOption** (*string* $key) inherited from Phalcon\Validation\Validator

Checks if an option is defined

public *mixed* **getOption** (*string* $key) inherited from Phalcon\Validation\Validator

Returns an option in the validator's options Returns null if the option hasn't been set

2.48.205 Class Phalcon\Validation\Validator\InclusionIn

extends Phalcon\Validation\Validator

implements Phalcon\Validation\ValidatorInterface

Check if a value is included into a list of values

```
<?php

use Phalcon\Validation\Validator\InclusionIn;

$validator->add('status', new InclusionIn(array(
   'message' => 'The status must be A or B'
   'domain' => array('A', 'B')
)));
```

Methods

public *boolean* **validate** (*Phalcon\Validation* $validator, *string* $attribute)

Executes the validation

public **__construct** ([*array* $options]) inherited from Phalcon\Validation\Validator

Phalcon\Validation\Validator constructor

public *mixed* **isSetOption** (*string* $key) inherited from Phalcon\Validation\Validator

Checks if an option is defined

public *mixed* **getOption** (*string* $key) inherited from Phalcon\Validation\Validator

Returns an option in the validator's options Returns null if the option hasn't been set

2.48.206 Class Phalcon\Validation\Validator\PresenceOf

extends Phalcon\Validation\Validator

implements Phalcon\Validation\ValidatorInterface

Validates that a value is not null or empty string

```
<?php

use Phalcon\Validation\Validator\PresenceOf;

$validator->add('name', new PresenceOf(array(
   'message' => 'The name is required'
)));
```

Methods

public *boolean* **validate** (*Phalcon\Validation* $validator, *string* $attribute)

Executes the validation

public **__construct** ([*array* $options]) inherited from Phalcon\Validation\Validator

Phalcon\Validation\Validator constructor

public *mixed* **isSetOption** (*string* $key) inherited from Phalcon\Validation\Validator

Checks if an option is defined

public *mixed* **getOption** (*string* $key) inherited from Phalcon\Validation\Validator

Returns an option in the validator's options Returns null if the option hasn't been set

2.48.207 Class Phalcon\Validation\Validator\Regex

extends Phalcon\Validation\Validator

implements Phalcon\Validation\ValidatorInterface

Allows validate if the value of a field matches a regular expression

```
<?php

use Phalcon\Validation\Validator\Regex as RegexValidator;

$validator->add('created_at', new RegexValidator(array(
   'pattern' => '/^[0-9]{4}[-\/](0[1-9]|1[12])[-\/](0[1-9]|[12][0-9]|3[01])$/',
   'message' => 'The creation date is invalid'
)));
```

Methods

public *boolean* **validate** (*Phalcon\Validation* $validator, *string* $attribute)

Executes the validation

public **__construct** ([*array* $options]) inherited from Phalcon\Validation\Validator

Phalcon\Validation\Validator constructor

public *mixed* **isSetOption** (*string* $key) inherited from Phalcon\Validation\Validator

Checks if an option is defined

public *mixed* **getOption** (*string* $key) inherited from Phalcon\Validation\Validator

Returns an option in the validator's options Returns null if the option hasn't been set

2.48.208 Class Phalcon\Validation\Validator\StringLength

extends Phalcon\Validation\Validator

implements Phalcon\Validation\ValidatorInterface

Validates that a string has the specified maximum and minimum constraints

```
<?php

use Phalcon\Validation\Validator\StringLength as StringLength;

$validation->validate('name_last', new StringLength(array(
'max' => 50,
'min' => 2,
'messageMaximum' => 'We don't like really long names',
'messageMinimum' => 'We want more than just their initials'
)));
```

Methods

public *boolean* **validate** (*Phalcon\Validation* $validator, *string* $attribute)

Executes the validation

public **__construct** ([*array* $options]) inherited from Phalcon\Validation\Validator

Phalcon\Validation\Validator constructor

public *mixed* **isSetOption** (*string* $key) inherited from Phalcon\Validation\Validator

Checks if an option is defined

public *mixed* **getOption** (*string* $key) inherited from Phalcon\Validation\Validator

Returns an option in the validator's options Returns null if the option hasn't been set

2.48.209 Class Phalcon\Version

This class allows to get the installed version of the framework

Methods

protected static **_getVersion** ()

Area where the version number is set. The format is as follows: ABBCCDE A - Major version B - Med version (two digits) C - Min version (two digits) D - Special release: 1 = Alpha, 2 = Beta, 3 = RC, 4 = Stable E - Special release version i.e. RC1, Beta2 etc.

public static *string* **get** ()

Returns the active version (string)

```
<?php

 echo Phalcon\Version::get();
```

public static *int* **getId** ()

Returns the numeric active version

```
<?php

 echo Phalcon\Version::getId();
```

2.48.210 Interface Phalcon\Acl\AdapterInterface

Phalcon\Acl\AdapterInterface initializer

Methods

abstract public **setDefaultAction** (*int* $defaultAccess)

Sets the default access level (Phalcon\Acl::ALLOW or Phalcon\Acl::DENY)

abstract public *int* **getDefaultAction** ()

Returns the default ACL access level

abstract public *boolean* **addRole** (*Phalcon\Acl\RoleInterface* $role, [*array* $accessInherits])

Adds a role to the ACL list. Second parameter lets to inherit access data from other existing role

abstract public **addInherit** (*string* $roleName, *string* $roleToInherit)

Do a role inherit from another existing role

abstract public *boolean* **isRole** (*string* $roleName)

Check whether role exist in the roles list

abstract public *boolean* **isResource** (*string* $resourceName)

Check whether resource exist in the resources list

abstract public *boolean* **addResource** (*Phalcon\Acl\ResourceInterface* $resource, [*array* $accessList])

Adds a resource to the ACL list Access names can be a particular action, by example search, update, delete, etc or a list of them

abstract public **addResourceAccess** (*string* $resourceName, *mixed* $accessList)

Adds access to resources

abstract public **dropResourceAccess** (*string* $resourceName, *mixed* $accessList)

Removes an access from a resource

abstract public **allow** (*string* $roleName, *string* $resourceName, *mixed* $access)

Allow access to a role on a resource

abstract public *boolean* **deny** (*string* $roleName, *string* $resourceName, *mixed* $access)

Deny access to a role on a resource

abstract public *boolean* **isAllowed** (*string* $role, *string* $resource, *string* $access)

Check whether a role is allowed to access an action from a resource

abstract public *string* **getActiveRole** ()

Returns the role which the list is checking if it's allowed to certain resource/access

abstract public *string* **getActiveResource** ()

Returns the resource which the list is checking if some role can access it

abstract public *string* **getActiveAccess** ()

Returns the access which the list is checking if some role can access it

abstract public *Phalcon\Acl\RoleInterface* [] **getRoles** ()

Return an array with every role registered in the list

abstract public *Phalcon\Acl\ResourceInterface* [] **getResources** ()

Return an array with every resource registered in the list

2.48.211 Interface Phalcon\Acl\ResourceInterface

Phalcon\Acl\ResourceInterface initializer

Methods

abstract public **__construct** (*string* $name, [*string* $description])

Phalcon\Acl\ResourceInterface constructor

abstract public *string* **getName** ()

Returns the resource name

abstract public *string* **getDescription** ()

Returns resource description

abstract public *string* **__toString** ()

Magic method __toString

2.48.212 Interface Phalcon\Acl\RoleInterface

Phalcon\Acl\RoleInterface initializer

Methods

abstract public **__construct** (*string* $name, [*string* $description])

Phalcon\Acl\Role constructor

abstract public *string* **getName** ()

Returns the role name

abstract public *string* **getDescription** ()

Returns role description

abstract public *string* **__toString** ()

Magic method __toString

2.48.213 Interface Phalcon\Annotations\AdapterInterface

Phalcon\Annotations\AdapterInterface initializer

Methods

abstract public **setReader** (*Phalcon\Annotations\ReaderInterface* $reader)

Sets the annotations parser

abstract public *Phalcon\Annotations\ReaderInterface* **getReader** ()

Returns the annotation reader

abstract public *Phalcon\Annotations\Reflection* **get** (*string|object* $className)

Parses or retrieves all the annotations found in a class

abstract public *array* **getMethods** (*string* $className)

Returns the annotations found in all the class' methods

abstract public *Phalcon\Annotations\Collection* **getMethod** (*string* $className, *string* $methodName)

Returns the annotations found in a specific method

abstract public *array* **getProperties** (*string* $className)

Returns the annotations found in all the class' methods

abstract public *Phalcon\Annotations\Collection* **getProperty** (*string* $className, *string* $propertyName)

Returns the annotations found in a specific property

2.48.214 Interface Phalcon\Annotations\ReaderInterface

Phalcon\Annotations\ReaderInterface initializer

Methods

abstract public *array* **parse** (*string* $className)

Reads annotations from the class dockblocks, its methods and/or properties

abstract public static *array* **parseDocBlock** (*string* $docBlock, [*unknown* $file], [*unknown* $line])

Parses a raw doc block returning the annotations found

2.48.215 Interface Phalcon\Cache\BackendInterface

Phalcon\Cache\BackendInterface initializer

Methods

abstract public *mixed* **start** (*int|string* $keyName, [*long* $lifetime])

Starts a cache. The $keyname allows to identify the created fragment

abstract public **stop** ([*boolean* $stopBuffer])

Stops the frontend without store any cached content

abstract public *mixed* **getFrontend** ()

Returns front-end instance adapter related to the back-end

abstract public *array* **getOptions** ()

Returns the backend options

abstract public *boolean* **isFresh** ()

Checks whether the last cache is fresh or cached

abstract public *boolean* **isStarted** ()

Checks whether the cache has starting buffering or not

abstract public **setLastKey** (*string* $lastKey)

Sets the last key used in the cache

abstract public *string* **getLastKey** ()

Gets the last key stored by the cache

abstract public *mixed* **get** (*int|string* $keyName, [*long* $lifetime])

Returns a cached content

abstract public **save** ([*int|string* $keyName], [*string* $content], [*long* $lifetime], [*boolean* $stopBuffer])

Stores cached content into the file backend and stops the frontend

abstract public *boolean* **delete** (*int|string* $keyName)

Deletes a value from the cache by its key

abstract public *array* **queryKeys** ([*string* $prefix])

Query the existing cached keys

abstract public *boolean* **exists** ([*string* $keyName], [*long* $lifetime])

Checks if cache exists and it hasn't expired

2.48.216 Interface Phalcon\Cache\FrontendInterface

Phalcon\Cache\FrontendInterface initializer

Methods

abstract public *int* **getLifetime** ()

Returns the cache lifetime

abstract public *boolean* **isBuffering** ()

Check whether if frontend is buffering output

abstract public **start** ()

Starts the frontend

abstract public *string* **getContent** ()

Returns output cached content

abstract public **stop** ()

Stops the frontend

abstract public **beforeStore** (*mixed* $data)

Serializes data before storing it

abstract public **afterRetrieve** (*mixed* $data)

Unserializes data after retrieving it

2.48.217 Interface Phalcon\DI\InjectionAwareInterface

Phalcon\DI\InjectionAwareInterface initializer

Methods

abstract public **setDI** (*Phalcon\DiInterface* $dependencyInjector)

Sets the dependency injector

abstract public *Phalcon\DiInterface* **getDI** ()

Returns the internal dependency injector

2.48.218 Interface Phalcon\DI\ServiceInterface

Phalcon\DI\ServiceInterface initializer

Methods

abstract public **__construct** (*string* $name, *mixed* $definition, [*boolean* $shared])

abstract public **getName** ()

Returns the service's name

abstract public **setShared** (*boolean* $shared)

Sets if the service is shared or not

abstract public *boolean* **isShared** ()

Check whether the service is shared or not

abstract public **setDefinition** (*mixed* $definition)

Set the service definition

abstract public *mixed* **getDefinition** ()

Returns the service definition

abstract public *mixed* **resolve** ([*array* $parameters], [*Phalcon\DiInterface* $dependencyInjector])

Resolves the service

abstract public static *Phalcon\DI\ServiceInterface* **__set_state** (*array* $attributes)

Restore the interal state of a service

2.48.219 Interface Phalcon\Db\AdapterInterface

Phalcon\Db\AdapterInterface initializer

Methods

abstract public **__construct** (*array* $descriptor)

Constructor for Phalcon\Db\Adapter

abstract public *array* **fetchOne** (*string* $sqlQuery, [*int* $fetchMode], [*int* $placeholders])

Returns the first row in a SQL query result

abstract public *array* **fetchAll** (*string* $sqlQuery, [*int* $fetchMode], [*int* $placeholders])

Dumps the complete result of a query into an array

abstract public *boolean* **insert** (*string* $table, *array* $values, [*array* $fields], [*array* $dataTypes])

Inserts data into a table using custom RBDM SQL syntax

abstract public *boolean* **update** (*string* $table, *array* $fields, *array* $values, [*string* $whereCondition], [*array* $dataTypes])

Updates data on a table using custom RBDM SQL syntax

abstract public *boolean* **delete** (*string* $table, [*string* $whereCondition], [*array* $placeholders], [*array* $dataTypes])

Deletes data from a table using custom RBDM SQL syntax

abstract public *string* **getColumnList** (*array* $columnList)

Gets a list of columns

abstract public *string* **limit** (*string* $sqlQuery, *int* $number)

Appends a LIMIT clause to $sqlQuery argument

abstract public *string* **tableExists** (*string* $tableName, [*string* $schemaName])

Generates SQL checking for the existence of a schema.table

abstract public *string* **viewExists** (*string* $viewName, [*string* $schemaName])

Generates SQL checking for the existence of a schema.view

abstract public *string* **forUpdate** (*string* $sqlQuery)

Returns a SQL modified with a FOR UPDATE clause

abstract public *string* **sharedLock** (*string* $sqlQuery)

Returns a SQL modified with a LOCK IN SHARE MODE clause

abstract public *boolean* **createTable** (*string* $tableName, *string* $schemaName, *array* $definition)

Creates a table

abstract public *boolean* **dropTable** (*string* $tableName, *string* $schemaName, [*boolean* $ifExists])

Drops a table from a schema/database

abstract public *boolean* **addColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Adds a column to a table

abstract public *boolean* **modifyColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Modifies a table column based on a definition

abstract public *boolean* **dropColumn** (*string* $tableName, *string* $schemaName, *string* $columnName)

Drops a column from a table

abstract public *boolean* **addIndex** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index)

Adds an index to a table

abstract public *boolean* **dropIndex** (*string* $tableName, *string* $schemaName, *string* $indexName)

Drop an index from a table

abstract public *boolean* **addPrimaryKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index)

Adds a primary key to a table

abstract public *boolean* **dropPrimaryKey** (*string* $tableName, *string* $schemaName)

Drops primary key from a table

abstract public *boolean true* **addForeignKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ReferenceInterface* $reference)

Adds a foreign key to a table

abstract public *boolean true* **dropForeignKey** (*string* $tableName, *string* $schemaName, *string* $referenceName)

Drops a foreign key from a table

abstract public *string* **getColumnDefinition** (*Phalcon\Db\ColumnInterface* $column)

Returns the SQL column definition from a column

abstract public *array* **listTables** ([*string* $schemaName])

List all tables on a database <code> print_r($connection->listTables("blog") ?>

abstract public *array* **getDescriptor** ()

Return descriptor used to connect to the active database

abstract public *string* **getConnectionId** ()

Gets the active connection unique identifier

abstract public *string* **getSQLStatement** ()

Active SQL statement in the object

abstract public *string* **getRealSQLStatement** ()

Active SQL statement in the object without replace bound paramters

abstract public *array* **getSQLVariables** ()

Active SQL statement in the object

abstract public *array* **getSQLBindTypes** ()

Active SQL statement in the object

abstract public *string* **getType** ()

Returns type of database system the adapter is used for

abstract public *string* **getDialectType** ()

Returns the name of the dialect used

abstract public *Phalcon\Db\DialectInterface* **getDialect** ()

Returns internal dialect instance

abstract public *boolean* **connect** ([*array* $descriptor])

This method is automatically called in Phalcon\Db\Adapter\Pdo constructor. Call it when you need to restore a database connection

abstract public *Phalcon\Db\ResultInterface* **query** (*string* $sqlStatement, [*array* $placeholders], [*array* $dataTypes])

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server return rows

abstract public *boolean* **execute** (*string* $sqlStatement, [*array* $placeholders], [*array* $dataTypes])

Sends SQL statements to the database server returning the success state. Use this method only when the SQL statement sent to the server don't return any row

abstract public *int* **affectedRows** ()

Returns the number of affected rows by the last INSERT/UPDATE/DELETE reported by the database system

abstract public *boolean* **close** ()

Closes active connection returning success. Phalcon automatically closes and destroys active connections within Phalcon\Db\Pool

abstract public *string* **escapeIdentifier** (*string* $identifier)

Escapes a column/table/schema name

abstract public *string* **escapeString** (*string* $str)

Escapes a value to avoid SQL injections

abstract public **bindParams** (*string* $sqlStatement, *array* $params)

Bind params to a SQL statement

abstract public *array* **convertBoundParams** (*string* $sqlStatement, *array* $params)

Converts bound params like :name: or ?1 into ? bind params

abstract public *int* **lastInsertId** ([*string* $sequenceName])

Returns insert id for the auto_increment column inserted in the last SQL statement

abstract public *boolean* **begin** ()

Starts a transaction in the connection

abstract public *boolean* **rollback** ()

Rollbacks the active transaction in the connection

abstract public *boolean* **commit** ()

Commits the active transaction in the connection

abstract public *boolean* **isUnderTransaction** ()

Checks whether connection is under database transaction

abstract public *PDO* **getInternalHandler** ()

Return internal PDO handler

abstract public *Phalcon\Db\IndexInterface* [] **describeIndexes** (*string* $table, [*string* $schema])

Lists table indexes

abstract public *Phalcon\Db\ReferenceInterface* [] **describeReferences** (*string* $table, [*string* $schema])

Lists table references

abstract public *array* **tableOptions** (*string* $tableName, [*string* $schemaName])

Gets creation options from a table

abstract public *Phalcon\Db\RawValue* **getDefaultIdValue** ()

Return the default identity value to insert in an identity column

abstract public *boolean* **supportSequences** ()

Check whether the database system requires a sequence to produce auto-numeric values

abstract public *Phalcon\Db\ColumnInterface* [] **describeColumns** (*string* $table, [*string* $schema])

Returns an array of Phalcon\Db\Column objects describing a table

2.48.220 Interface Phalcon\Db\ColumnInterface

Phalcon\Db\ColumnInterface initializer

Methods

abstract public **__construct** (*string* $columnName, *array* $definition)

Phalcon\Db\ColumnInterface constructor

abstract public *string* **getSchemaName** ()

Returns schema's table related to column

abstract public *string* **getName** ()

Returns column name

abstract public *int* **getType** ()

Returns column type

abstract public *int* **getSize** ()

Returns column size

abstract public *int* **getScale** ()

Returns column scale

abstract public *boolean* **isUnsigned** ()

Returns true if number column is unsigned

abstract public *boolean* **isNotNull** ()

Not null

abstract public *boolean* **isPrimary** ()

Column is part of the primary key?

abstract public *boolean* **isAutoIncrement** ()

Auto-Increment

abstract public *boolean* **isNumeric** ()

Check whether column have an numeric type

abstract public *boolean* **isFirst** ()

Check whether column have first position in table

abstract public *string* **getAfterPosition** ()

Check whether field absolute to position in table

abstract public *int* **getBindType** ()

Returns the type of bind handling

abstract public static *PhalconDbColumnInterface* **__set_state** (*array* $data)

Restores the internal state of a Phalcon\Db\Column object

2.48.221 Interface Phalcon\Db\DialectInterface

Phalcon\Db\DialectInterface initializer

Methods

abstract public *string* **limit** (*string* $sqlQuery, *int* $number)

Generates the SQL for LIMIT clause

abstract public *string* **forUpdate** (*string* $sqlQuery)

Returns a SQL modified with a FOR UPDATE clause

abstract public *string* **sharedLock** (*string* $sqlQuery)

Returns a SQL modified with a LOCK IN SHARE MODE clause

abstract public *string* **select** (*array* $definition)

Builds a SELECT statement

abstract public *string* **getColumnList** (*array* $columnList)

Gets a list of columns

abstract public **getColumnDefinition** (*Phalcon\Db\ColumnInterface* $column)

Gets the column name in MySQL

abstract public *string* **addColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Generates SQL to add a column to a table

abstract public *string* **modifyColumn** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ColumnInterface* $column)

Generates SQL to modify a column in a table

abstract public *string* **dropColumn** (*string* $tableName, *string* $schemaName, *string* $columnName)

Generates SQL to delete a column from a table

abstract public *string* **addIndex** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index)

Generates SQL to add an index to a table

abstract public *string* **dropIndex** (*string* $tableName, *string* $schemaName, *string* $indexName)

Generates SQL to delete an index from a table

abstract public *string* **addPrimaryKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\IndexInterface* $index)

Generates SQL to add the primary key to a table

abstract public *string* **dropPrimaryKey** (*string* $tableName, *string* $schemaName)

Generates SQL to delete primary key from a table

abstract public *string* **addForeignKey** (*string* $tableName, *string* $schemaName, *Phalcon\Db\ReferenceInterface* $reference)

Generates SQL to add an index to a table

abstract public *string* **dropForeignKey** (*string* $tableName, *string* $schemaName, *string* $referenceName)

Generates SQL to delete a foreign key from a table

abstract public *string* **createTable** (*string* $tableName, *string* $schemaName, *array* $definition)

Generates SQL to create a table

abstract public *string* **dropTable** (*string* $tableName, *string* $schemaName)

Generates SQL to drop a table

abstract public *string* **tableExists** (*string* $tableName, [*string* $schemaName])

Generates SQL checking for the existence of a schema.table

abstract public *string* **describeColumns** (*string* $table, [*string* $schema])

Generates SQL to describe a table

abstract public *array* **listTables** ([*string* $schemaName])

List all tables on database

abstract public *string* **describeIndexes** (*string* $table, [*string* $schema])

Generates SQL to query indexes on a table

abstract public *string* **describeReferences** (*string* $table, [*string* $schema])

Generates SQL to query foreign keys on a table

abstract public *string* **tableOptions** (*string* $table, [*string* $schema])

Generates the SQL to describe the table creation options

2.48.222 Interface Phalcon\Db\IndexInterface

Phalcon\Db\IndexInterface initializer

Methods

abstract public **__construct** (*string* $indexName, *array* $columns)

Phalcon\Db\Index constructor

abstract public *string* **getName** ()

Gets the index name

abstract public *array* **getColumns** ()

Gets the columns that comprends the index

abstract public static *Phalcon\Db\IndexInterface* **__set_state** (*array* $data)

Restore a Phalcon\Db\Index object from export

2.48.223 Interface Phalcon\Db\ReferenceInterface

Phalcon\Db\ReferenceInterface initializer

Methods

abstract public **__construct** (*string* $referenceName, *array* $definition)

Phalcon\Db\ReferenceInterface constructor

abstract public *string* **getName** ()

Gets the index name

abstract public *string* **getSchemaName** ()

Gets the schema where referenced table is

abstract public *string* **getReferencedSchema** ()

Gets the schema where referenced table is

abstract public *array* **getColumns** ()

Gets local columns which reference is based

abstract public *string* **getReferencedTable** ()

Gets the referenced table

abstract public *array* **getReferencedColumns** ()

Gets referenced columns

abstract public static *Phalcon\Db\ReferenceInterface* **__set_state** (*array* $data)

Restore a Phalcon\Db\Reference object from export

2.48.224 Interface Phalcon\Db\ResultInterface

Phalcon\Db\ResultInterface initializer

Methods

abstract public **__construct** (*Phalcon\Db\AdapterInterface* $connection, *PDOStatement* $result, [*string* $sqlStatement], [*array* $bindParams], [*array* $bindTypes])

Phalcon\Db\Result\Pdo constructor

abstract public *boolean* **execute** ()

Allows to executes the statement again. Some database systems don't support scrollable cursors, So, as cursors are forward only, we need to execute the cursor again to fetch rows from the begining

abstract public *mixed* **fetch** ()

Fetches an array/object of strings that corresponds to the fetched row, or FALSE if there are no more rows. This method is affected by the active fetch flag set using Phalcon\Db\Result\Pdo::setFetchMode

abstract public *mixed* **fetchArray** ()

Returns an array of strings that corresponds to the fetched row, or FALSE if there are no more rows. This method is affected by the active fetch flag set using Phalcon\Db\Result\Pdo::setFetchMode

abstract public *array* **fetchAll** ()

Returns an array of arrays containing all the records in the result This method is affected by the active fetch flag set using Phalcon\Db\Result\Pdo::setFetchMode

abstract public *int* **numRows** ()

Gets number of rows returned by a resulset

abstract public **dataSeek** (*int* $number)

Moves internal resulset cursor to another position letting us to fetch a certain row

abstract public **setFetchMode** (*int* $fetchMode)

Changes the fetching mode affecting Phalcon\Db\Result\Pdo::fetch()

abstract public *PDOStatement* **getInternalResult** ()

Gets the internal PDO result object

2.48.225 Interface Phalcon\DiInterface

extends ArrayAccess

Phalcon\DiInterface initializer

Methods

abstract public *Phalcon\DI\ServiceInterface* **set** (*string* $alias, *mixed* $definition, [*boolean* $shared])

Registers a service in the services container

abstract public *Phalcon\DI\ServiceInterface* **setShared** (*string* $name, *mixed* $definition)

Registers an “always shared” service in the services container

abstract public **remove** (*string* $alias)

Removes a service in the services container

abstract public *Phalcon\DI\ServiceInterface* **attempt** (*string* $alias, *mixed* $definition, [*boolean* $shared])

Attempts to register a service in the services container Only is successful if a service hasn’t been registered previously with the same name

abstract public *mixed* **get** (*string* $alias, [*array* $parameters])

Resolves the service based on its configuration

abstract public *mixed* **getShared** (*string* $alias, [*array* $parameters])

Returns a shared service based on their configuration

abstract public *Phalcon\DI\ServiceInterface* **setRaw** (*string* $name, *Phalcon\DI\ServiceInterface* $rawDefinition)

Sets a service using a raw Phalcon\DI\Service definition

abstract public *mixed* **getRaw** (*string* $name)

Returns a service definition without resolving

abstract public *Phalcon\DI\ServiceInterface* **getService** (*string* $name)

Returns the corresponding Phalcon\Di\Service instance for a service

abstract public *boolean* **has** (*string* $alias)

Check whether the DI contains a service by a name

abstract public *boolean* **wasFreshInstance** ()

Check whether the last service obtained via getShared produced a fresh instance or an existing one

abstract public *array* **getServices** ()

Return the services registered in the DI

abstract public static **setDefault** (*Phalcon\DiInterface* $dependencyInjector)

Set a default dependency injection container to be obtained into static methods

abstract public static *Phalcon\DiInterface* **getDefault** ()

Return the last DI created

abstract public static **reset** ()

Resets the internal default DI

abstract public **offsetExists** (*unknown* $offset) inherited from ArrayAccess

...

abstract public **offsetGet** (*unknown* $offset) inherited from ArrayAccess

...

abstract public **offsetSet** (*unknown* $offset, *unknown* $value) inherited from ArrayAccess

...

abstract public **offsetUnset** (*unknown* $offset) inherited from ArrayAccess

...

2.48.226 Interface Phalcon\DispatcherInterface

Phalcon\DispatcherInterface initializer

Methods

abstract public **setActionSuffix** (*string* $actionSuffix)

Sets the default action suffix

abstract public **setDefaultNamespace** (*string* $namespace)

Sets the default namespace

abstract public **setDefaultAction** (*string* $actionName)

Sets the default action name

abstract public **setActionName** (*string* $actionName)

Sets the action name to be dispatched

abstract public *string* **getActionName** ()

Gets last dispatched action name

abstract public **setParams** (*array* $params)

Sets action params to be dispatched

abstract public *array* **getParams** ()

Gets action params

abstract public **setParam** (*mixed* $param, *mixed* $value)

Set a param by its name or numeric index

abstract public *mixed* **getParam** (*mixed* $param, [*string*|*array* $filters])

Gets a param by its name or numeric index

abstract public *boolean* **isFinished** ()

Checks if the dispatch loop is finished or has more pendent controllers/tasks to disptach

abstract public *mixed* **getReturnedValue** ()

Returns value returned by the lastest dispatched action

abstract public *object* **dispatch** ()

Dispatches a handle action taking into account the routing parameters

abstract public **forward** (*array* $forward)

Forwards the execution flow to another controller/action

2.48.227 Interface Phalcon\EscaperInterface

Phalcon\EscaperInterface initializer

Methods

abstract public **setEncoding** (*string* $encoding)

Sets the encoding to be used by the escaper

abstract public *string* **getEncoding** ()

Returns the internal encoding used by the escaper

abstract public **setHtmlQuoteType** (*int* $quoteType)

Sets the HTML quoting type for htmlspecialchars

abstract public *string* **escapeHtml** (*string* $text)

Escapes a HTML string

abstract public *string* **escapeHtmlAttr** (*string* $text)

Escapes a HTML attribute string

abstract public *string* **escapeCss** (*string* $css)

Escape CSS strings by replacing non-alphanumeric chars by their hexadecimal representation

abstract public *string* **escapeJs** (*string* $js)

Escape Javascript strings by replacing non-alphanumeric chars by their hexadecimal representation

abstract public *string* **escapeUrl** (*string* $url)

Escapes a URL. Internally uses rawurlencode

2.48.228 Interface Phalcon\Events\EventsAwareInterface

Phalcon\Events\EventsAwareInterface initializer

Methods

abstract public **setEventsManager** (*Phalcon\Events\ManagerInterface* $eventsManager)

Sets the events manager

abstract public *Phalcon\Events\ManagerInterface* **getEventsManager** ()

Returns the internal event manager

2.48.229 Interface Phalcon\Events\ManagerInterface

Phalcon\Events\ManagerInterface initializer

Methods

abstract public **attach** (*string* $eventType, *object* $handler)

Attach a listener to the events manager

abstract public **dettachAll** ([*string* $type])

Removes all events from the EventsManager

abstract public *mixed* **fire** (*string* $eventType, *object* $source, [*mixed* $data])

Fires a event in the events manager causing that the acive listeners will be notified about it

abstract public *array* **getListeners** (*string* $type)

Returns all the attached listeners of a certain type

2.48.230 Interface Phalcon\FilterInterface

Phalcon\FilterInterface initializer

Methods

abstract public *Phalcon\FilterInterface* **add** (*string* $name, *callable* $handler)

Adds a user-defined filter

abstract public *mixed* **sanitize** (*mixed* $value, *mixed* $filters)

Sanizites a value with a specified single or set of filters

abstract public *object[]* **getFilters** ()

Return the user-defined filters in the instance

2.48.231 Interface Phalcon\FlashInterface

Phalcon\FlashInterface initializer

Methods

abstract public *string* **error** (*string* $message)

Shows a HTML error message

abstract public *string* **notice** (*string* $message)

Shows a HTML notice/information message

abstract public *string* **success** (*string* $message)

Shows a HTML success message

abstract public *string* **warning** (*string* $message)

Shows a HTML warning message

abstract public *string* **message** (*string* $type, *string* $message)

Outputs a message

2.48.232 Interface Phalcon\Http\RequestInterface

Phalcon\Http\RequestInterface initializer

Methods

abstract public *mixed* **get** (*string* $name, [*string|array* $filters], [*mixed* $defaultValue])

Gets a variable from the $_REQUEST superglobal applying filters if needed

abstract public *mixed* **getPost** (*string* $name, [*string|array* $filters], [*mixed* $defaultValue])

Gets a variable from the $_POST superglobal applying filters if needed

abstract public *mixed* **getQuery** (*string* $name, [*string|array* $filters], [*mixed* $defaultValue])

Gets variable from $_GET superglobal applying filters if needed

abstract public *mixed* **getServer** (*string* $name)

Gets variable from $_SERVER superglobal

abstract public *boolean* **has** (*string* $name)

Checks whether $_SERVER superglobal has certain index

abstract public *boolean* **hasPost** (*string* $name)

Checks whether $_POST superglobal has certain index

abstract public *boolean* **hasQuery** (*string* $name)

Checks whether $_SERVER superglobal has certain index

abstract public *mixed* **hasServer** (*string* $name)

Checks whether $_SERVER superglobal has certain index

abstract public *string* **getHeader** (*string* $header)

Gets HTTP header from request data

abstract public *string* **getScheme** ()

Gets HTTP schema (http/https)

abstract public *boolean* **isAjax** ()

Checks whether request has been made using ajax. Checks if $_SERVER['HTTP_X_REQUESTED_WITH']=='XMLHttpRequest'

abstract public *boolean* **isSoapRequested** ()

Checks whether request has been made using SOAP

abstract public *boolean* **isSecureRequest** ()

Checks whether request has been made using any secure layer

abstract public *string* **getRawBody** ()

Gets HTTP raws request body

abstract public *string* **getServerAddress** ()

Gets active server address IP

abstract public *string* **getServerName** ()

Gets active server name

abstract public *string* **getHttpHost** ()

Gets information about schema, host and port used by the request

abstract public *string* **getClientAddress** ([*boolean* $trustForwardedHeader])

Gets most possibly client IPv4 Address. This methods search in $_SERVER['REMOTE_ADDR'] and optionally in $_SERVER['HTTP_X_FORWARDED_FOR']

abstract public *string* **getMethod** ()

Gets HTTP method which request has been made

abstract public *string* **getUserAgent** ()

Gets HTTP user agent used to made the request

abstract public *boolean* **isMethod** (*string*|*array* $methods)

Check if HTTP method match any of the passed methods

abstract public *boolean* **isPost** ()

Checks whether HTTP method is POST. if $_SERVER['REQUEST_METHOD']=='POST'

abstract public *boolean* **isGet** ()

Checks whether HTTP method is GET. if $_SERVER['REQUEST_METHOD']=='GET'

abstract public *boolean* **isPut** ()

Checks whether HTTP method is PUT. if $_SERVER['REQUEST_METHOD']=='PUT'

abstract public *boolean* **isHead** ()

Checks whether HTTP method is HEAD. if $_SERVER['REQUEST_METHOD']=='HEAD'

abstract public *boolean* **isDelete** ()

Checks whether HTTP method is DELETE. if $_SERVER['REQUEST_METHOD']=='DELETE'

abstract public *boolean* **isOptions** ()

Checks whether HTTP method is OPTIONS. if $_SERVER['REQUEST_METHOD']=='OPTIONS'

abstract public *boolean* **hasFiles** ()

Checks whether request include attached files

abstract public *Phalcon\Http\Request\FileInterface* [] **getUploadedFiles** ()

Gets attached files as Phalcon\Http\Request\FileInterface compatible instances

abstract public *string* **getHTTPReferer** ()

Gets web page that refers active request. ie: http://www.google.com

abstract public *array* **getAcceptableContent** ()

Gets array with mime/types and their quality accepted by the browser/client from $_SERVER['HTTP_ACCEPT']

abstract public *array* **getBestAccept** ()

Gets best mime/type accepted by the browser/client from $_SERVER['HTTP_ACCEPT']

abstract public *array* **getClientCharsets** ()

Gets charsets array and their quality accepted by the browser/client from $_SERVER['HTTP_ACCEPT_CHARSET']

abstract public *string* **getBestCharset** ()

Gets best charset accepted by the browser/client from $_SERVER['HTTP_ACCEPT_CHARSET']

abstract public *array* **getLanguages** ()

Gets languages array and their quality accepted by the browser/client from $_SERVER['HTTP_ACCEPT_LANGUAGE']

abstract public *string* **getBestLanguage** ()

Gets best language accepted by the browser/client from $_SERVER['HTTP_ACCEPT_LANGUAGE']

2.48.233 Interface Phalcon\Http\Request\FileInterface

Phalcon\Http\Request\FileInterface initializer

Methods

abstract public **__construct** (*array* $file)

Phalcon\Http\Request\FileInterface constructor

abstract public *int* **getSize** ()

Returns the file size of the uploaded file

abstract public *string* **getName** ()

Returns the real name of the uploaded file

abstract public *string* **getTempName** ()

Returns the temporal name of the uploaded file

abstract public *boolean* **moveTo** (*string* $destination)

Move the temporary file to a destination

2.48.234 Interface Phalcon\Http\ResponseInterface

Phalcon\Http\ResponseInterface initializer

Methods

abstract public *Phalcon\Http\ResponseInterface* **setStatusCode** (*int* $code, *string* $message)

Sets the HTTP response code

abstract public *Phalcon\Http\Response\Headers* **getHeaders** ()

Returns headers set by the user

abstract public *Phalcon\Http\ResponseInterface* **setHeader** (*string* $name, *string* $value)

Overwrites a header in the response

abstract public *Phalcon\Http\ResponseInterface* **setRawHeader** (*string* $header)

Send a raw header to the response

abstract public *Phalcon\Http\ResponseInterface* **resetHeaders** ()

Resets all the stablished headers

abstract public *Phalcon\Http\ResponseInterface* **setExpires** (*DateTime* $datetime)

Sets output expire time header

abstract public *Phalcon\Http\ResponseInterface* **setNotModified** ()

Sends a Not-Modified response

abstract public *Phalcon\Http\ResponseInterface* **setContentType** (*string* $contentType, [*string* $charset])

Sets the response content-type mime, optionally the charset

abstract public *Phalcon\Http\ResponseInterface* **redirect** ([*string* $location], [*boolean* $externalRedirect], [*int* $statusCode])

Redirect by HTTP to another action or URL

abstract public *Phalcon\Http\ResponseInterface* **setContent** (*string* $content)

Sets HTTP response body

abstract public *Phalcon\Http\ResponseInterface* **appendContent** (*string* $content)

Appends a string to the HTTP response body

abstract public *string* **getContent** ()

Gets the HTTP response body

abstract public *Phalcon\Http\ResponseInterface* **sendHeaders** ()

Sends headers to the client

abstract public *Phalcon\Http\ResponseInterface* **send** ()

Prints out HTTP response to the client

2.48.235 Interface Phalcon\Http\Response\HeadersInterface

Phalcon\Http\Response\HeadersInterface initializer

Methods

abstract public **set** (*string* $name, *string* $value)

Sets a header to be sent at the end of the request

abstract public *string* **get** (*string* $name)

Gets a header value from the internal bag

abstract public **setRaw** (*string* $header)

Sets a raw header to be sent at the end of the request

abstract public *boolean* **send** ()

Sends the headers to the client

abstract public **reset** ()

Reset set headers

abstract public static *Phalcon\Http\Response\HeadersInterface* **__set_state** (*array* $data)

Restore a Phalcon\Http\Response\Headers object

2.48.236 Interface Phalcon\Logger\AdapterInterface

Phalcon\Logger\AdapterInterface initializer

Methods

abstract public **setFormatter** (*Phalcon\Logger\FormatterInterface* $formatter)

Sets the message formatter

abstract public *Phalcon\Logger\FormatterInterface* **getFormatter** ()

Returns the internal formatter

abstract public **setLogLevel** (*int* $level)

Filters the logs sent to the handlers to be greater or equals than a specific level

abstract public *int* **getLogLevel** ()

Returns the current log level

abstract public **log** (*string* $message, [*int* $type])

Sends/Writes messages to the file log

abstract public **begin** ()

Starts a transaction

abstract public **commit** ()

Commits the internal transaction

abstract public **rollback** ()

Rollbacks the internal transaction

abstract public *boolean* **close** ()

Closes the logger

abstract public **debug** (*string* $message)

Sends/Writes a debug message to the log

abstract public **error** (*string* $message)

Sends/Writes an error message to the log

abstract public **info** (*string* $message)

Sends/Writes an info message to the log

abstract public **notice** (*string* $message)

Sends/Writes a notice message to the log

abstract public **warning** (*string* $message)

Sends/Writes a warning message to the log

abstract public **alert** (*string* $message)

Sends/Writes an alert message to the log

2.48.237 Interface Phalcon\Logger\FormatterInterface

Phalcon\Logger\FormatterInterface initializer

Methods

abstract public **format** (*string* $message, *int* $type, *int* $timestamp)

Applies a format to a message before sent it to the internal log

2.48.238 Interface Phalcon\Mvc\CollectionInterface

Phalcon\Mvc\CollectionInterface initializer

Methods

abstract public **__construct** ([*Phalcon\DiInterface* $dependencyInjector])

Phalcon\Mvc\Collection

abstract public **setId** (*mixed* $id)

Sets a value for the _id propery, creates a MongoId object if needed

abstract public *MongoId* **getId** ()

Returns the value of the _id property

abstract public *array* **getReservedAttributes** ()

Returns an array with reserved properties that cannot be part of the insert/update

abstract public *string* **getSource** ()

Returns collection name mapped in the model

abstract public **setConnectionService** (*string* $connectionService)

Sets a service in the services container that returns the Mongo database

abstract public *MongoDb* **getConnection** ()

Retrieves a database connection

abstract public *mixed* **readAttribute** (*string* $attribute)

Reads an attribute value by its name

```
<?php

echo $robot->readAttribute('name');
```

abstract public **writeAttribute** (*string* $attribute, *mixed* $value)

Writes an attribute value by its name

```
<?php

$robot->writeAttribute('name', 'Rosey');
```

abstract public static *Phalcon\Mvc\Collection* **cloneResult** (*Phalcon\Mvc\Collection* $collection, *array* $document)

Returns a cloned collection

abstract public *boolean* **fireEvent** (*string* $eventName)

Fires an event, implicitly calls behaviors and listeners in the events manager are notified

abstract public *boolean* **fireEventCancel** (*string* $eventName)

Fires an event, implicitly listeners in the events manager are notified This method stops if one of the callbacks/listeners returns boolean false

abstract public *boolean* **validationHasFailed** ()

Check whether validation process has generated any messages

```
<?php

use Phalcon\Mvc\Model\Validator\ExclusionIn as ExclusionIn;

class Subscriptors extends Phalcon\Mvc\Model
{

public function validation()
{
    $this->validate(new ExclusionIn(array(
            'field' => 'status',
            'domain' => array('A', 'I')
    )));
```

```
    if ($this->validationHasFailed() == true) {
            return false;
    }
}

}
```

abstract public *Phalcon\Mvc\Model\MessageInterface* [] **getMessages** ()

Returns all the validation messages

```
<?php

$robot = new Robots();
$robot->type = 'mechanical';
$robot->name = 'Astro Boy';
$robot->year = 1952;
if ($robot->save() == false) {
echo "Umh, We can't store robots right now ";
foreach ($robot->getMessages() as $message) {
    echo $message;
}
} else {
echo "Great, a new robot was saved successfully!";
}
```

abstract public **appendMessage** (*Phalcon\Mvc\Model\MessageInterface* $message)

Appends a customized message on the validation process

```
<?php

use \Phalcon\Mvc\Model\Message as Message;

class Robots extends Phalcon\Mvc\Model
{

    public function beforeSave()
    {
            if (this->name == 'Peter') {
                    $message = new Message("Sorry, but a robot cannot be named Peter");
                    $this->appendMessage($message);
            }
    }
}
```

abstract public *boolean* **save** ()

Creates/Updates a collection based on the values in the atributes

abstract public static *Phalcon\Mvc\Collection* **findById** (*string* $id)

Find a document by its id

abstract public static *array* **findFirst** ([*array* $parameters])

Allows to query the first record that match the specified conditions

```
<?php

 //What's the first robot in robots table?
 $robot = Robots::findFirst();
```

```
echo "The robot name is ", $robot->name;

//What's the first mechanical robot in robots table?
$robot = Robots::findFirst(array(
    array("type" => "mechanical")
));
echo "The first mechanical robot name is ", $robot->name;

//Get first virtual robot ordered by name
$robot = Robots::findFirst(array(
    array("type" => "mechanical"),
    "order" => array("name" => 1)
));
echo "The first virtual robot name is ", $robot->name;
```

abstract public static *array* **find** ([*array* $parameters])

Allows to query a set of records that match the specified conditions

```
<?php

//How many robots are there?
$robots = Robots::find();
echo "There are ", count($robots);

//How many mechanical robots are there?
$robots = Robots::find(array(
    array("type" => "mechanical")
));
echo "There are ", count($robots);

//Get and print virtual robots ordered by name
$robots = Robots::findFirst(array(
    array("type" => "virtual"),
    "order" => array("name" => 1)
));
foreach ($robots as $robot) {
  echo $robot->name, "\n";
}

//Get first 100 virtual robots ordered by name
$robots = Robots::find(array(
    array("type" => "virtual"),
    "order" => array("name" => 1),
    "limit" => 100
));
foreach ($robots as $robot) {
  echo $robot->name, "\n";
}
```

abstract public static *array* **count** ([*array* $parameters])

Perform a count over a collection

abstract public *boolean* **delete** ()

Deletes a model instance. Returning true on success or false otherwise.

```
<?php
```

```
$robot = Robots::findFirst();
$robot->delete();

foreach(Robots::find() as $robot){
   $robot->delete();
}
```

2.48.239 Interface Phalcon\Mvc\Collection\ManagerInterface

Phalcon\Mvc\Collection\ManagerInterface initializer

Methods

abstract public **setCustomEventsManager** (*Phalcon\Mvc\CollectionInterface* $model, *Phalcon\Events\ManagerInterface* $eventsManager)

Sets a custom events manager for a specific model

abstract public *Phalcon\Events\ManagerInterface* **getCustomEventsManager** (*Phalcon\Mvc\CollectionInterface* $model)

Returns a custom events manager related to a model

abstract public **initialize** (*Phalcon\Mvc\CollectionInterface* $model)

Initializes a model in the models manager

abstract public *bool* **isInitialized** (*string* $modelName)

Check whether a model is already initialized

abstract public *Phalcon\Mvc\CollectionInterface* **getLastInitialized** ()

Get the latest initialized model

abstract public **setConnectionService** (*Phalcon\Mvc\CollectionInterface* $model, *string* $connectionService)

Sets a connection service for a specific model

abstract public **useImplicitObjectIds** (*Phalcon\Mvc\CollectionInterface* $model, *boolean* $useImplicitObjectIds)

Sets if a model must use implicit objects ids

abstract public *boolean* **isUsingImplicitObjectIds** (*Phalcon\Mvc\CollectionInterface* $model)

Checks if a model is using implicit object ids

abstract public *Phalcon\Db\AdapterInterface* **getConnection** (*Phalcon\Mvc\CollectionInterface* $model)

Returns the connection related to a model

abstract public **notifyEvent** (*string* $eventName, *Phalcon\Mvc\CollectionInterface* $model)

Receives events generated in the models and dispatches them to a events-manager if available Notify the behaviors that are listening in the model

2.48.240 Interface Phalcon\Mvc\ControllerInterface

2.48.241 Interface Phalcon\Mvc\DispatcherInterface

extends PhalconDispatcherInterface

Phalcon\Mvc\DispatcherInterface initializer

Methods

abstract public **setControllerSuffix** (*string* $controllerSuffix)

Sets the default controller suffix

abstract public **setDefaultController** (*string* $controllerName)

Sets the default controller name

abstract public **setControllerName** (*string* $controllerName)

Sets the controller name to be dispatched

abstract public *string* **getControllerName** ()

Gets last dispatched controller name

abstract public *Phalcon\Mvc\ControllerInterface* **getLastController** ()

Returns the lastest dispatched controller

abstract public *Phalcon\Mvc\ControllerInterface* **getActiveController** ()

Returns the active controller in the dispatcher

abstract public **setActionSuffix** (*string* $actionSuffix) inherited from Phalcon\DispatcherInterface

Sets the default action suffix

abstract public **setDefaultNamespace** (*string* $namespace) inherited from Phalcon\DispatcherInterface

Sets the default namespace

abstract public **setDefaultAction** (*string* $actionName) inherited from Phalcon\DispatcherInterface

Sets the default action name

abstract public **setActionName** (*string* $actionName) inherited from Phalcon\DispatcherInterface

Sets the action name to be dispatched

abstract public *string* **getActionName** () inherited from Phalcon\DispatcherInterface

Gets last dispatched action name

abstract public **setParams** (*array* $params) inherited from Phalcon\DispatcherInterface

Sets action params to be dispatched

abstract public *array* **getParams** () inherited from Phalcon\DispatcherInterface

Gets action params

abstract public **setParam** (*mixed* $param, *mixed* $value) inherited from Phalcon\DispatcherInterface

Set a param by its name or numeric index

abstract public *mixed* **getParam** (*mixed* $param, [*string*|*array* $filters]) inherited from Phalcon\DispatcherInterface

Gets a param by its name or numeric index

abstract public *boolean* **isFinished** () inherited from Phalcon\DispatcherInterface

Checks if the dispatch loop is finished or has more pendent controllers/tasks to disptach

abstract public *mixed* **getReturnedValue** () inherited from Phalcon\DispatcherInterface

Returns value returned by the lastest dispatched action

abstract public *object* **dispatch** () inherited from Phalcon\DispatcherInterface

Dispatches a handle action taking into account the routing parameters

abstract public **forward** (*array* $forward) inherited from Phalcon\DispatcherInterface

Forwards the execution flow to another controller/action

2.48.242 Interface Phalcon\Mvc\ModelInterface

Phalcon\Mvc\ModelInterface initializer

Methods

abstract public **__construct** ([*Phalcon\DiInterface* $dependencyInjector], [*Phalcon\Mvc\Model\ManagerInterface* $modelsManager])

Phalcon\Mvc\Model constructor

abstract public *Phalcon\Mvc\ModelInterface* **setTransaction** (*Phalcon\Mvc\Model\TransactionInterface* $transaction)

Sets a transaction related to the Model instance

abstract public *string* **getSource** ()

Returns table name mapped in the model

abstract public *string* **getSchema** ()

Returns schema name where table mapped is located

abstract public **setConnectionService** (*string* $connectionService)

Sets both read/write connection services

abstract public **setWriteConnectionService** (*string* $connectionService)

Sets the DependencyInjection connection service used to write data

abstract public **setReadConnectionService** (*string* $connectionService)

Sets the DependencyInjection connection service used to read data

abstract public *string* **getReadConnectionService** ()

Returns DependencyInjection connection service used to read data

abstract public *string* **getWriteConnectionService** ()

Returns DependencyInjection connection service used to write data

abstract public *Phalcon\Db\AdapterInterface* **getReadConnection** ()

Gets internal database connection

abstract public *Phalcon\Db\AdapterInterface* **getWriteConnection** ()

Gets internal database connection

abstract public *Phalcon\Mvc\Model* **assign** (*array* $data, [*array* $columnMap])

Assigns values to a model from an array

abstract public static *Phalcon\Mvc\Model* $result **cloneResultMap** (*Phalcon\Mvc\Model* $base, *array* $data, *array* $columnMap, [*int* $dirtyState])

Assigns values to a model from an array returning a new model

abstract public static *Phalcon\Mvc\ModelInterface* $result **cloneResult** (*Phalcon\Mvc\ModelInterface* $base, *array* $result)

Assigns values to a model from an array returning a new model

abstract public static **cloneResultMapHydrate** (*array* $data, *array* $columnMap, *int* $hydrationMode)

Returns an hydrated result based on the data and the column map

abstract public static *Phalcon\Mvc\Model\ResultsetInterface* **find** ([*array* $parameters])

Allows to query a set of records that match the specified conditions

abstract public static *Phalcon\Mvc\ModelInterface* **findFirst** ([*array* $parameters])

Allows to query the first record that match the specified conditions

abstract public static *Phalcon\Mvc\Model\CriteriaInterface* **query** ([*Phalcon\DiInterface* $dependencyInjector])

Create a criteria for a especific model

abstract public static *int* **count** ([*array* $parameters])

Allows to count how many records match the specified conditions

abstract public static *double* **sum** ([*array* $parameters])

Allows to calculate a summatory on a column that match the specified conditions

abstract public static *mixed* **maximum** ([*array* $parameters])

Allows to get the maximum value of a column that match the specified conditions

abstract public static *mixed* **minimum** ([*array* $parameters])

Allows to get the minimum value of a column that match the specified conditions

abstract public static *double* **average** ([*array* $parameters])

Allows to calculate the average value on a column matching the specified conditions

abstract public *boolean* **fireEvent** (*string* $eventName)

Fires an event, implicitly calls behaviors and listeners in the events manager are notified

abstract public *boolean* **fireEventCancel** (*string* $eventName)

Fires an event, implicitly calls behaviors and listeners in the events manager are notified This method stops if one of the callbacks/listeners returns boolean false

abstract public **appendMessage** (*Phalcon\Mvc\Model\MessageInterface* $message)

Appends a customized message on the validation process

abstract public *boolean* **validationHasFailed** ()

Check whether validation process has generated any messages

abstract public *Phalcon\Mvc\Model\MessageInterface* [] **getMessages** ()

Returns all the validation messages

abstract public *boolean* **save** ([*array* $data])

Inserts or updates a model instance. Returning true on success or false otherwise.

abstract public *boolean* **create** ([*array* $data])

Inserts a model instance. If the instance already exists in the persistance it will throw an exception Returning true on success or false otherwise.

abstract public *boolean* **update** ([*array* $data])

Updates a model instance. If the instance doesn't exist in the persistance it will throw an exception Returning true on success or false otherwise.

abstract public *boolean* **delete** ()

Deletes a model instance. Returning true on success or false otherwise.

abstract public *int* **getOperationMade** ()

Returns the type of the latest operation performed by the ORM Returns one of the OP_* class constants

abstract public **refresh** ()

Refreshes the model attributes re-querying the record from the database

abstract public *mixed* **readAttribute** (*string* $attribute)

Reads an attribute value by its name

abstract public **writeAttribute** (*string* $attribute, *mixed* $value)

Writes an attribute value by its name

abstract public *Phalcon\Mvc\Model\ResultsetInterface* **getRelated** (*string* $modelName, [*array* $arguments])

Returns related records based on defined relations

2.48.243 Interface Phalcon\Mvc\Model\BehaviorInterface

Phalcon\Mvc\Model\BehaviorInterface initializer

Methods

abstract public **__construct** ([*array* $options])

Phalcon\Mvc\Model\Behavior

abstract public **notify** (*string* $type, *Phalcon\Mvc\ModelInterface* $model)

This method receives the notifications from the EventsManager

abstract public **missingMethod** (*Phalcon\Mvc\ModelInterface* $model, *string* $method, [*array* $arguments])

Calls a method when it's missing in the model

2.48.244 Interface Phalcon\Mvc\Model\CriteriaInterface

Phalcon\Mvc\Model\CriteriaInterface initializer

Methods

abstract public *Phalcon\Mvc\Model\CriteriaInterface* **setModelName** (*string* $modelName)

Set a model on which the query will be executed

abstract public *string* **getModelName** ()

Returns an internal model name on which the criteria will be applied

abstract public *Phalcon\Mvc\Model\CriteriaInterface* **bind** (*string* $bindParams)

Adds the bind parameter to the criteria

abstract public *Phalcon\Mvc\Model\CriteriaInterface* **where** (*string* $conditions)

Adds the conditions parameter to the criteria

abstract public *Phalcon\Mvc\Model\CriteriaInterface* **conditions** (*string* $conditions)

Adds the conditions parameter to the criteria

abstract public *Phalcon\Mvc\Model\CriteriaInterface* **order** (*string* $orderColumns)

Adds the order-by parameter to the criteria

abstract public *Phalcon\Mvc\Model\CriteriaInterface* **limit** (*int* $limit, [*int* $offset])

Adds the limit parameter to the criteria

abstract public *Phalcon\Mvc\Model\CriteriaInterface* **forUpdate** ([*boolean* $forUpdate])

Adds the “for_update” parameter to the criteria

abstract public *Phalcon\Mvc\Model\Criteria* **sharedLock** ([*boolean* $sharedLock])

Adds the “shared_lock” parameter to the criteria

abstract public *Phalcon\Mvc\Model\Criteria* **andWhere** (*string* $conditions)

Appends a condition to the current conditions using an AND operator

abstract public *Phalcon\Mvc\Model\Criteria* **orWhere** (*string* $conditions)

Appends a condition to the current conditions using an OR operator

abstract public *string* **getWhere** ()

Returns the conditions parameter in the criteria

abstract public *string* **getConditions** ()

Returns the conditions parameter in the criteria

abstract public *string* **getLimit** ()

Returns the limit parameter in the criteria

abstract public *string* **getOrder** ()

Returns the order parameter in the criteria

abstract public *string* **getParams** ()

Returns all the parameters defined in the criteria

abstract public static *static* **fromInput** (*Phalcon\DiInterface* $dependencyInjector, *string* $modelName, *array* $data)

Builds a Phalcon\Mvc\Model\Criteria based on an input array like $_POST

abstract public *Phalcon\Mvc\Model\ResultsetInterface* **execute** ()

Executes a find using the parameters built with the criteria

2.48.245 Interface Phalcon\Mvc\Model\ManagerInterface

Phalcon\Mvc\Model\ManagerInterface initializer

Methods

abstract public **initialize** (*Phalcon\Mvc\ModelInterface* $model)

Initializes a model in the model manager

abstract public *boolean* **isInitialized** (*string* $modelName)

Check of a model is already initialized

abstract public *Phalcon\Mvc\ModelInterface* **getLastInitialized** ()

Get last initialized model

abstract public *Phalcon\Mvc\ModelInterface* **load** (*string* $modelName)

Loads a model throwing an exception if it doesn't exist

abstract public *Phalcon\Mvc\Model\RelationInterface* **addHasOne** (*Phalcon\Mvc\ModelInterface* $model, *mixed* $fields, *string* $referenceModel, *mixed* $referencedFields, [*array* $options])

Setup a 1-1 relation between two models

abstract public *Phalcon\Mvc\Model\RelationInterface* **addBelongsTo** (*Phalcon\Mvc\ModelInterface* $model, *mixed* $fields, *string* $referenceModel, *mixed* $referencedFields, [*array* $options])

Setup a relation reverse 1-1 between two models

abstract public *Phalcon\Mvc\Model\RelationInterface* **addHasMany** (*Phalcon\Mvc\ModelInterface* $model, *mixed* $fields, *string* $referenceModel, *mixed* $referencedFields, [*array* $options])

Setup a relation 1-n between two models

abstract public *boolean* **existsBelongsTo** (*string* $modelName, *string* $modelRelation)

Checks whether a model has a belongsTo relation with another model

abstract public *boolean* **existsHasMany** (*string* $modelName, *string* $modelRelation)

Checks whether a model has a hasMany relation with another model

abstract public *boolean* **existsHasOne** (*string* $modelName, *string* $modelRelation)

Checks whether a model has a hasOne relation with another model

abstract public *Phalcon\Mvc\Model\ResultsetInterface* **getBelongsToRecords** (*string* $method, *string* $modelName, *string* $modelRelation, *Phalcon\Mvc\Model* $record, [*array* $parameters])

Gets belongsTo related records from a model

abstract public *Phalcon\Mvc\Model\ResultsetInterface* **getHasManyRecords** (*string* $method, *string* $modelName, *string* $modelRelation, *Phalcon\Mvc\Model* $record, [*array* $parameters])

Gets hasMany related records from a model

abstract public *Phalcon\Mvc\Model\ResultsetInterface* **getHasOneRecords** (*string* $method, *string* $modelName, *string* $modelRelation, *Phalcon\Mvc\Model* $record, [*array* $parameters])

Gets belongsTo related records from a model

abstract public *array* **getBelongsTo** (*Phalcon\Mvc\ModelInterface* $model)

Gets belongsTo relations defined on a model

abstract public *array* **getHasMany** (*Phalcon\Mvc\ModelInterface* $model)

Gets hasMany relations defined on a model

abstract public *array* **getHasOne** (*Phalcon\Mvc\ModelInterface* $model)

Gets hasOne relations defined on a model

abstract public *array* **getHasOneAndHasMany** (*Phalcon\Mvc\ModelInterface* $model)

Gets hasOne relations defined on a model

abstract public *Phalcon\Mvc\Model\RelationInterface* [] **getRelations** (*string* $modelName)

Query all the relationships defined on a model

abstract public *array* **getRelationsBetween** (*string* $firstModel, *string* $secondModel)

Query the relations between two models

abstract public *Phalcon\Mvc\Model\QueryInterface* **createQuery** (*string* $phql)

Creates a Phalcon\Mvc\Model\Query without execute it

abstract public *Phalcon\Mvc\Model\QueryInterface* **executeQuery** (*string* $phql, [*array* $placeholders])

Creates a Phalcon\Mvc\Model\Query and execute it

abstract public *Phalcon\Mvc\Model\Query\BuilderInterface* **createBuilder** ([*string* $params])

Creates a Phalcon\Mvc\Model\Query\Builder

abstract public **addBehavior** (*Phalcon\Mvc\ModelInterface* $model, *Phalcon\Mvc\Model\BehaviorInterface* $behavior)

Binds a behavior to a model

abstract public **notifyEvent** (*string* $eventName, *Phalcon\Mvc\ModelInterface* $model)

Receives events generated in the models and dispatches them to a events-manager if available Notify the behaviors that are listening in the model

abstract public *boolean* **missingMethod** (*Phalcon\Mvc\ModelInterface* $model, *string* $eventName, *array* $data)

Dispatch a event to the listeners and behaviors This method expects that the endpoint listeners/behaviors returns true meaning that a least one is implemented

abstract public *Phalcon\Mvc\Model\QueryInterface* **getLastQuery** ()

Returns the last query created or executed in the

2.48.246 Interface Phalcon\Mvc\Model\MessageInterface

Phalcon\Mvc\Model\MessageInterface initializer

Methods

abstract public **__construct** (*string* $message, [*string* $field], [*string* $type])

Phalcon\Mvc\Model\Message constructor

abstract public **setType** (*string* $type)

Sets message type

abstract public *string* **getType** ()

Returns message type

abstract public **setMessage** (*string* $message)

Sets verbose message

abstract public *string* **getMessage** ()

Returns verbose message

abstract public **setField** (*string* $field)

Sets field name related to message

abstract public *string* **getField** ()

Returns field name related to message

abstract public *string* **__toString** ()

Magic __toString method returns verbose message

abstract public static *Phalcon\Mvc\Model\MessageInterface* **__set_state** (*array* $message)

Magic __set_state helps to recover messsages from serialization

2.48.247 Interface Phalcon\Mvc\Model\MetaDataInterface

Phalcon\Mvc\Model\MetaDataInterface initializer

Methods

abstract public **setStrategy** (*Phalcon\Mvc\Model\MetaData\Strategy\Introspection* $strategy)

Set the meta-data extraction strategy

abstract public *Phalcon\Mvc\Model\MetaData\Strategy\Introspection* **getStrategy** ()

Return the strategy to obtain the meta-data

abstract public *array* **readMetaData** (*Phalcon\Mvc\ModelInterface* $model)

Reads meta-data for certain model

abstract public *mixed* **readMetaDataIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index)

Reads meta-data for certain model using a MODEL_* constant

abstract public **writeMetaDataIndex** (*Phalcon\Mvc\Model* $model, *int* $index, *mixed* $data)

Writes meta-data for certain model using a MODEL_* constant

abstract public *array* **readColumnMap** (*Phalcon\Mvc\ModelInterface* $model)

Reads the ordered/reversed column map for certain model

abstract public **readColumnMapIndex** (*Phalcon\Mvc\ModelInterface* $model, *int* $index)

Reads column-map information for certain model using a MODEL_* constant

abstract public *array* **getAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns table attributes names (fields)

abstract public *array* **getPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns an array of fields which are part of the primary key

abstract public *array* **getNonPrimaryKeyAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns an arrau of fields which are not part of the primary key

abstract public *array* **getNotNullAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns an array of not null attributes

abstract public *array* **getDataTypes** (*Phalcon\Mvc\ModelInterface* $model)

Returns attributes and their data types

abstract public *array* **getDataTypesNumeric** (*Phalcon\Mvc\ModelInterface* $model)

Returns attributes which types are numerical

abstract public *string* **getIdentityField** (*Phalcon\Mvc\ModelInterface* $model)

Returns the name of identity field (if one is present)

abstract public *array* **getBindTypes** (*Phalcon\Mvc\ModelInterface* $model)

Returns attributes and their bind data types

abstract public *array* **getAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns attributes that must be ignored from the INSERT SQL generation

abstract public *array* **getAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model)

Returns attributes that must be ignored from the UPDATE SQL generation

abstract public **setAutomaticCreateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes)

Set the attributes that must be ignored from the INSERT SQL generation

abstract public **setAutomaticUpdateAttributes** (*Phalcon\Mvc\ModelInterface* $model, *array* $attributes)

Set the attributes that must be ignored from the UPDATE SQL generation

abstract public *array* **getColumnMap** (*Phalcon\Mvc\ModelInterface* $model)

Returns the column map if any

abstract public *array* **getReverseColumnMap** (*Phalcon\Mvc\ModelInterface* $model)

Returns the reverse column map if any

abstract public *boolean* **hasAttribute** (*Phalcon\Mvc\ModelInterface* $model, *string* $attribute)

Check if a model has certain attribute

abstract public *boolean* **isEmpty** ()

Checks if the internal meta-data container is empty

abstract public **reset** ()

Resets internal meta-data in order to regenerate it

abstract public *array* **read** (*string* $key)

Reads meta-data from the adapter

abstract public **write** (*string* $key, *array* $data)

Writes meta-data to the adapter

2.48.248 Interface Phalcon\Mvc\Model\QueryInterface

Phalcon\Mvc\Model\QueryInterface initializer

Methods

abstract public **__construct** (*string* $phql)

Phalcon\Mvc\Model\Query constructor

abstract public *array* **parse** ()

Parses the intermediate code produced by Phalcon\Mvc\Model\Query\Lang generating another intermediate representation that could be executed by Phalcon\Mvc\Model\Query

abstract public *mixed* **execute** ([*array* $bindParams], [*array* $bindTypes])

Executes a parsed PHQL statement

2.48.249 Interface Phalcon\Mvc\Model\Query\BuilderInterface

Phalcon\Mvc\Model\Query\BuilderInterface initializer

Methods

abstract public **__construct** ([*array* $params])

Phalcon\Mvc\Model\Query\Builder

abstract public *Phalcon\Mvc\Model\Query\BuilderInterface* **columns** (*string|array* $columns)

Sets the columns to be queried

abstract public *string|array* **getColumns** ()

Return the columns to be queried

abstract public *Phalcon\Mvc\Model\Query\BuilderInterface* **from** (*string|array* $models)

Sets the models who makes part of the query

abstract public *Phalcon\Mvc\Model\Query\BuilderInterface* **addFrom** (*string* $model, [*string* $alias])

Add a model to take part of the query

abstract public *string|array* **getFrom** ()

Return the models who makes part of the query

abstract public *Phalcon\Mvc\Model\Query\BuilderInterface* **join** (*string* $model, [*string* $conditions], [*string* $alias])

Adds a INNER join to the query

abstract public *Phalcon\Mvc\Model\Query\BuilderInterface* **where** (*string* $conditions)

Sets conditions for the query

abstract public *Phalcon\Mvc\Model\Query\Builder* **andWhere** (*string* $conditions)

Appends a condition to the current conditions using a AND operator

abstract public *Phalcon\Mvc\Model\Query\Builder* **orWhere** (*string* $conditions)

Appends a condition to the current conditions using a OR operator

abstract public *string|array* **getWhere** ()

Return the conditions for the query

abstract public *Phalcon\Mvc\Model\Query\BuilderInterface* **orderBy** (*string* $orderBy)

Sets a ORDER BY condition clause

abstract public *string|array* **getOrderBy** ()

Return the set ORDER BY clause

abstract public *Phalcon\Mvc\Model\Query\BuilderInterface* **having** (*string* $having)

Sets a HAVING condition clause

abstract public *string|array* **getHaving** ()

Returns the HAVING condition clause

abstract public *Phalcon\Mvc\Model\Query\BuilderInterface* **limit** (*int* $limit, [*int* $offset])

Sets a LIMIT clause

abstract public *string|array* **getLimit** ()

Returns the current LIMIT clause

abstract public *Phalcon\Mvc\Model\Query\BuilderInterface* **groupBy** (*string* $group)

Sets a LIMIT clause

abstract public *string* **getGroupBy** ()

Returns the GROUP BY clause

abstract public *string* **getPhql** ()

Returns a PHQL statement built based on the builder parameters

abstract public *Phalcon\Mvc\Model\QueryInterface* **getQuery** ()

Returns the query built

2.48.250 Interface Phalcon\Mvc\Model\Query\StatusInterface

Phalcon\Mvc\Model\Query\StatusInterface initializer

Methods

abstract public **__construct** (*boolean* $success, *Phalcon\Mvc\ModelInterface* $model)

Phalcon\Mvc\Model\Query\Status

abstract public *Phalcon\Mvc\ModelInterface* **getModel** ()

Returns the model which executed the action

abstract public *Phalcon\Mvc\Model\MessageInterface* [] **getMessages** ()

Returns the messages produced by a operation failed

abstract public *boolean* **success** ()

Allows to check if the executed operation was successful

2.48.251 Interface Phalcon\Mvc\Model\RelationInterface

Phalcon\Mvc\Model\RelationInterface initializer

Methods

abstract public **__construct** (*int* $type, *string* $referencedModel, *string|array* $fields, *string|array* $referencedFields, [*array* $options])

Phalcon\Mvc\Model\Relation constructor

abstract public *int* **getType** ()

Returns the relation's type

abstract public *string* **getReferencedModel** ()

Returns the referenced model

abstract public *string|array* **getFields** ()

Returns the fields

abstract public *string|array* **getReferencedFields** ()

Returns the referenced fields

abstract public *string|array* **getOptions** ()

Returns the options

abstract public *string|array* **isForeingKey** ()

Check whether the relation act as a foreign key

abstract public *string|array* **getForeignKey** ()

Returns the foreign key configuration

abstract public *boolean* **hasThrough** ()

Check whether the relation

abstract public *string* **getThrough** ()

Returns the 'through' relation if any

2.48.252 Interface Phalcon\Mvc\Model\ResultInterface

Phalcon\Mvc\Model\ResultInterface initializer

Methods

abstract public **setDirtyState** (*boolean* $dirtyState)

Sets the object's state

2.48.253 Interface Phalcon\Mvc\Model\ResultsetInterface

Phalcon\Mvc\Model\ResultsetInterface initializer

Methods

abstract public *int* **getType** ()

Returns the internal type of data retrieval that the resultset is using

abstract public *Phalcon\Mvc\ModelInterface* **getFirst** ()

Get first row in the resultset

abstract public *Phalcon\Mvc\ModelInterface* **getLast** ()

Get last row in the resultset

abstract public **setIsFresh** (*boolean* $isFresh)

Set if the resultset is fresh or an old one cached

abstract public *boolean* **isFresh** ()

Tell if the resultset if fresh or an old one cached

abstract public *Phalcon\Cache\BackendInterface* **getCache** ()

Returns the associated cache for the resultset

abstract public *array* **toArray** ()

Returns a complete resultset as an array, if the resultset has a big number of rows it could consume more memory than currently it does.

2.48.254 Interface Phalcon\Mvc\Model\TransactionInterface

Phalcon\Mvc\Model\TransactionInterface initializer

Methods

abstract public **__construct** (*Phalcon\DiInterface* $dependencyInjector, [*boolean* $autoBegin], [*string* $service])

Phalcon\Mvc\Model\Transaction constructor

abstract public **setTransactionManager** (*Phalcon\Mvc\Model\Transaction\ManagerInterface* $manager)

Sets transaction manager related to the transaction

abstract public *boolean* **begin** ()

Starts the transaction

abstract public *boolean* **commit** ()

Commits the transaction

abstract public *boolean* **rollback** ([*string* $rollbackMessage], [*Phalcon\Mvc\ModelInterface* $rollbackRecord])

Rollbacks the transaction

abstract public *string* **getConnection** ()

Returns connection related to transaction

abstract public **setIsNewTransaction** (*boolean* $isNew)

Sets if is a reused transaction or new once

abstract public **setRollbackOnAbort** (*boolean* $rollbackOnAbort)

Sets flag to rollback on abort the HTTP connection

abstract public *boolean* **isManaged** ()

Checks whether transaction is managed by a transaction manager

abstract public *array* **getMessages** ()

Returns validations messages from last save try

abstract public *boolean* **isValid** ()

Checks whether internal connection is under an active transaction

abstract public **setRollbackedRecord** (*Phalcon\Mvc\ModelInterface* $record)

Sets object which generates rollback action

2.48.255 Interface Phalcon\Mvc\Model\Transaction\ManagerInterface

Phalcon\Mvc\Model\Transaction\ManagerInterface initializer

Methods

abstract public **__construct** ([*Phalcon\DiInterface* $dependencyInjector])

Phalcon\Mvc\Model\Transaction\Manager

abstract public *boolean* **has** ()

Checks whether manager has an active transaction

abstract public *Phalcon\Mvc\Model\TransactionInterface* **get** ([*boolean* $autoBegin])

Returns a new Phalcon\Mvc\Model\Transaction or an already created once

abstract public **rollbackPendent** ()

Rollbacks active transactions within the manager

abstract public **commit** ()

Commmits active transactions within the manager

abstract public **rollback** ([*boolean* $collect])

Rollbacks active transactions within the manager Collect will remove transaction from the manager

abstract public **notifyRollback** (*Phalcon\Mvc\Model\TransactionInterface* $transaction)

Notifies the manager about a rollbacked transaction

abstract public **notifyCommit** (*Phalcon\Mvc\Model\TransactionInterface* $transaction)

Notifies the manager about a commited transaction

abstract public **collectTransactions** ()

Remove all the transactions from the manager

2.48.256 Interface Phalcon\Mvc\Model\ValidatorInterface

Phalcon\Mvc\Model\ValidatorInterface initializer

Methods

abstract public *array* **getMessages** ()

Returns messages generated by the validator

abstract public *boolean* **validate** (*Phalcon\Mvc\ModelInterface* $record)

Executes the validator

2.48.257 Interface Phalcon\Mvc\ModuleDefinitionInterface

Phalcon\Mvc\ModuleDefinitionInterface initializer

Methods

abstract public **registerAutoloaders** ()

Registers an autoloader related to the module

abstract public **registerServices** (*Phalcon\DiInterface* $dependencyInjector)

Registers an autoloader related to the module

2.48.258 Interface Phalcon\Mvc\RouterInterface

Phalcon\Mvc\RouterInterface initializer

Methods

abstract public **setDefaultModule** (*string* $moduleName)

Sets the name of the default module

abstract public **setDefaultController** (*string* $controllerName)

Sets the default controller name

abstract public **setDefaultAction** (*string* $actionName)

Sets the default action name

abstract public **setDefaults** (*array* $defaults)

Sets an array of default paths

abstract public **handle** ([*string* $uri])

Handles routing information received from the rewrite engine

abstract public *Phalcon\Mvc\Router\RouteInterface* **add** (*string* $pattern, [*string/array* $paths], [*string* $httpMethods])

Adds a route to the router on any HTTP method

abstract public *Phalcon\Mvc\Router\RouteInterface* **addGet** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is GET

abstract public *Phalcon\Mvc\Router\RouteInterface* **addPost** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is POST

abstract public *Phalcon\Mvc\Router\RouteInterface* **addPut** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is PUT

abstract public *Phalcon\Mvc\Router\RouteInterface* **addDelete** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is DELETE

abstract public *Phalcon\Mvc\Router\RouteInterface* **addOptions** (*string* $pattern, [*string/array* $paths])

Add a route to the router that only match if the HTTP method is OPTIONS

abstract public *Phalcon\Mvc\Router\RouteInterface* **addHead** (*string* $pattern, [*string/array* $paths])

Adds a route to the router that only match if the HTTP method is HEAD

abstract public **clear** ()

Removes all the defined routes

abstract public *string* **getModuleName** ()

Returns processed module name

abstract public *string* **getControllerName** ()

Returns processed controller name

abstract public *string* **getActionName** ()

Returns processed action name

abstract public *array* **getParams** ()

Returns processed extra params

abstract public *Phalcon\Mvc\Router\RouteInterface* **getMatchedRoute** ()

Returns the route that matchs the handled URI

abstract public *array* **getMatches** ()

Return the sub expressions in the regular expression matched

abstract public *bool* **wasMatched** ()

Check if the router macthes any of the defined routes

abstract public *Phalcon\Mvc\Router\RouteInterface* [] **getRoutes** ()

Return all the routes defined in the router

abstract public *Phalcon\Mvc\Router\RouteInterface* **getRouteById** (*string* $id)

Returns a route object by its id

abstract public *Phalcon\Mvc\Router\RouteInterface* **getRouteByName** (*string* $name)

Returns a route object by its name

2.48.259 Interface Phalcon\Mvc\Router\RouteInterface

Phalcon\Mvc\Router\RouteInterface initializer

Methods

abstract public **__construct** (*string* $pattern, [*array* $paths], [*array*|*string* $httpMethods])

Phalcon\Mvc\Router\Route constructor

abstract public *string* **compilePattern** (*string* $pattern)

Replaces placeholders from pattern returning a valid PCRE regular expression

abstract public **via** (*string*|*array* $httpMethods)

Set one or more HTTP methods that constraint the matching of the route

abstract public **reConfigure** (*string* $pattern, [*array* $paths])

Reconfigure the route adding a new pattern and a set of paths

abstract public *string* **getName** ()

Returns the route's name

abstract public **setName** (*string* $name)

Sets the route's name

abstract public **setHttpMethods** (*string*|*array* $httpMethods)

Sets a set of HTTP methods that constraint the matching of the route

abstract public *string* **getRouteId** ()

Returns the route's id

abstract public *string* **getPattern** ()

Returns the route's pattern

abstract public *string* **getCompiledPattern** ()

Returns the route's pattern

abstract public *array* **getPaths** ()

Returns the paths

abstract public *string*|*array* **getHttpMethods** ()

Returns the HTTP methods that constraint matching the route

abstract public static **reset** ()

Resets the internal route id generator

2.48.260 Interface Phalcon\Mvc\UrlInterface

Phalcon\Mvc\UrlInterface initializer

Methods

abstract public **setBaseUri** (*string* $baseUri)

Sets a prefix to all the urls generated

abstract public *string* **getBaseUri** ()

Returns the prefix for all the generated urls. By default /

abstract public **setBasePath** (*string* $basePath)

Sets a base paths for all the generated paths

abstract public *string* **getBasePath** ()

Returns a base path

abstract public *string* **get** ([*string*|*array* $uri])

Generates a URL

abstract public *string* **path** ([*string* $path])

Generates a local path

2.48.261 Interface Phalcon\Mvc\ViewInterface

Phalcon\Mvc\ViewInterface initializer

Methods

abstract public **setViewsDir** (*string* $viewsDir)

Sets views directory. Depending of your platform, always add a trailing slash or backslash

abstract public *string* **getViewsDir** ()

Gets views directory

abstract public **setLayoutsDir** (*string* $layoutsDir)

Sets the layouts sub-directory. Must be a directory under the views directory. Depending of your platform, always add a trailing slash or backslash

abstract public *string* **getLayoutsDir** ()

Gets the current layouts sub-directory

abstract public **setPartialsDir** (*string* $partialsDir)

Sets a partials sub-directory. Must be a directory under the views directory. Depending of your platform, always add a trailing slash or backslash

abstract public *string* **getPartialsDir** ()

Gets the current partials sub-directory

abstract public **setBasePath** (*string* $basePath)

Sets base path. Depending of your platform, always add a trailing slash or backslash

abstract public **setRenderLevel** (*string* $level)

Sets the render level for the view

abstract public **setMainView** (*string* $viewPath)

Sets default view name. Must be a file without extension in the views directory

abstract public *string* **getMainView** ()

Returns the name of the main view

abstract public **setLayout** (*string* $layout)

Change the layout to be used instead of using the name of the latest controller name

abstract public *string* **getLayout** ()

Returns the name of the main view

abstract public **setTemplateBefore** (*string|array* $templateBefore)

Appends template before controller layout

abstract public **cleanTemplateBefore** ()

Resets any template before layouts

abstract public **setTemplateAfter** (*string|array* $templateAfter)

Appends template after controller layout

abstract public **cleanTemplateAfter** ()

Resets any template before layouts

abstract public **setParamToView** (*string* $key, *mixed* $value)

Adds parameters to views (alias of setVar)

abstract public **setVar** (*string* $key, *mixed* $value)

Adds parameters to views

abstract public *array* **getParamsToView** ()

Returns parameters to views

abstract public *string* **getControllerName** ()

Gets the name of the controller rendered

abstract public *string* **getActionName** ()

Gets the name of the action rendered

abstract public *array* **getParams** ()

Gets extra parameters of the action rendered

abstract public **start** ()

Starts rendering process enabling the output buffering

abstract public **registerEngines** (*array* $engines)

Register templating engines

abstract public **render** (*string* $controllerName, *string* $actionName, [*array* $params])

Executes render process from dispatching data

abstract public **pick** (*string* $renderView)

Choose a view different to render than last-controller/last-action

abstract public *string* **partial** (*string* $partialPath)

Renders a partial view

abstract public **finish** ()

Finishes the render process by stopping the output buffering

abstract public *Phalcon\Cache\BackendInterface* **getCache** ()

Returns the cache instance used to cache

abstract public **cache** ([*boolean|array* $options])

Cache the actual view render to certain level

abstract public **setContent** (*string* $content)

Externally sets the view content

abstract public *string* **getContent** ()

Returns cached ouput from another view stage

abstract public *string* **getActiveRenderPath** ()

Returns the path of the view that is currently rendered

abstract public **disable** ()

Disables the auto-rendering process

abstract public **enable** ()

Enables the auto-rendering process

abstract public **reset** ()

Resets the view component to its factory default values

2.48.262 Interface Phalcon\Mvc\View\EngineInterface

Phalcon\Mvc\View\EngineInterface initializer

Methods

abstract public **__construct** (*Phalcon\Mvc\ViewInterface* $view, [*Phalcon\DiInterface* $dependencyInjector])

Phalcon\Mvc\View\Engine constructor

abstract public *array* **getContent** ()

Returns cached ouput on another view stage

abstract public *string* **partial** (*string* $partialPath)

Renders a partial inside another view

abstract public **render** (*string* $path, *array* $params, [*boolean* $mustClean])

Renders a view using the template engine

2.48.263 Interface Phalcon\Paginator\AdapterInterface

Phalcon\Paginator\AdapterInterface initializer

Methods

abstract public **__construct** (*array* $config)

Phalcon\Paginator\AdapterInterface constructor

abstract public **setCurrentPage** (*int* $page)

Set the current page number

abstract public *stdClass* **getPaginate** ()

Returns a slice of the resultset to show in the pagination

2.48.264 Interface Phalcon\Session\AdapterInterface

Phalcon\Session\AdapterInterface initializer

Methods

abstract public **__construct** ([*array* $options])

Phalcon\Session construtor

abstract public **start** ()

Starts session, optionally using an adapter

abstract public **setOptions** (*array* $options)

Sets session options

abstract public *array* **getOptions** ()

Get internal options

abstract public *mixed* **get** (*string* $index, [*mixed* $defaultValue])

Gets a session variable from an application context

abstract public **set** (*string* $index, *string* $value)

Sets a session variable in an application context

abstract public *boolean* **has** (*string* $index)

Check whether a session variable is set in an application context

abstract public **remove** (*string* $index)

Removes a session variable from an application context

abstract public *string* **getId** ()

Returns active session id

abstract public *boolean* **isStarted** ()

Check whether the session has been started

abstract public *boolean* **destroy** ()

Destroys the active session

2.48.265 Interface Phalcon\Session\BagInterface

Phalcon\Session\BagInterface initializer

Methods

abstract public **initialize** ()

Initializes the session bag. This method must not be called directly, the class calls it when its internal data is accesed

abstract public **destroy** ()

Destroyes the session bag

abstract public **set** (*string* $property, *string* $value)

Setter of values

abstract public *mixed* **get** (*string* $property, [*mixed* $defaultValue])

Getter of values

abstract public *boolean* **has** (*string* $property)

Isset property

abstract public **__set** (*string* $property, *string* $value)

Setter of values

abstract public *mixed* **__get** (*string* $property)

Getter of values

abstract public *boolean* **__isset** (*string* $property)

Isset property

2.48.266 Interface Phalcon\Translate\AdapterInterface

Phalcon\Translate\AdapterInterface initializer

Methods

abstract public **__construct** (*array* $options)

Phalcon\Translate\Adapter\NativeArray constructor

abstract public *string* **_** (*string* $translateKey, [*array* $placeholders])

Returns the translation string of the given key

abstract public *string* **query** (*string* $index, [*array* $placeholders])

Returns the translation related to the given key

abstract public *bool* **exists** (*string* $index)

Check whether is defined a translation key in the internal array

2.48.267 Interface Phalcon\Validation\ValidatorInterface

Phalcon\Validation\ValidatorInterface initializer

Methods

abstract public *mixed* **isSetOption** (*string* $key)

Checks if an option is defined

abstract public *mixed* **getOption** (*string* $key)

Returns an option in the validator's options Returns null if the option hasn't been set

abstract public **validate** (*Phalcon\Validator* $validator, *string* $attribute)

Executes the validation

2.49 License

Phalcon is brought to you by the Phalcon Team! [Twitter - Google Plus - Github]

If you love Phalcon please return something to the community! :)

Other formats

- PDF
- HTML in one Zip
- ePub