
PhalconEye Documentation

Release 0.4.0

Ivan Vorontsov (LanTiaN)

Jul 27, 2017

Contents

1	What is Phalcon?	3
2	Table of Contents	5
2.1	Benchmarks	5
2.1.1	WordPress 3.9.1	6
2.1.2	Joomla 3.3.0	8
2.1.3	PhalconEye 0.4.0	10
2.2	Installation	12
2.3	User's guide	12
2.3.1	Working With A Grid	12
2.3.2	User management	13
2.3.3	Role management	15
2.3.4	Dynamic pages	16
2.3.4.1	Adding and editing pages	16
2.3.4.2	Page management	17
2.3.5	Menus	21
2.3.5.1	Adding and editing items	22
2.3.6	Languages	23
2.3.6.1	Adding a language	23
2.3.6.2	Performance note	24
2.3.6.3	Export	25
2.3.6.4	Import	25
2.3.6.5	Manage Translations	25
2.3.6.6	Wizard	26
2.3.7	File management	27
2.3.8	System settings	29
2.3.9	Performance settings	29
2.3.10	Access Rights	31
2.4	Developer's guide	33
2.4.1	CMS Structure	33
2.4.1.1	App directory	35
2.4.1.2	Public directory	35
2.4.2	Configuration	36
2.4.2.1	Stages	36
2.4.2.2	Config files	36
2.4.2.3	Behaviour	37

2.4.3	Packages	38
2.4.3.1	Package Manager	40
2.4.3.2	Package types	45
2.4.4	Models	58
2.4.4.1	Annotations	59
2.4.4.2	Methods	61
2.4.5	Views	61
2.4.5.1	Widget views	61
2.4.5.2	Module views	61
2.4.5.3	Helpers	62
2.4.5.4	Extension	62
2.4.6	Forms	62
2.4.6.1	Structure	65
2.4.6.2	Elements	67
2.4.6.3	Fieldsets	72
2.4.6.4	Conditions	73
2.4.6.5	Form view	74
2.4.6.6	Entities support	74
2.4.6.7	Validation	76
2.4.6.8	Filter	76
2.4.6.9	Text Form	77
2.4.6.10	File Form	77
2.4.6.11	Entity Form (Trait)	78
2.4.7	Grid System	78
2.4.7.1	Source	80
2.4.7.2	Columns	81
2.4.7.3	Actions	82
2.4.7.4	Grid View	83
2.4.8	Helpers	84
2.4.8.1	Helper creation	84
2.4.8.2	Existing helpers	84
2.4.9	Navigation	86
2.4.9.1	Navigation Styling	87
2.4.10	Languages And Translations	88
2.4.11	Access Control List	89
2.4.11.1	ACL Usage	89
2.4.11.2	Model ACL	89
2.4.12	Console	90
2.4.12.1	Usage	90
2.4.12.2	Commands	92
2.4.12.3	Command Creation	92
2.4.13	Assets	94
2.4.14	Cache	97

PhalconEye CMS is based on Phalcon PHP Framework. It contains basic building blocks for rapid development: modules, widgets, plugins and themes.

Originally, it has been designed to help web developers kickstart their projects, and therefore it is more a development platform then general purpose Content Management Systems unlike others you might already be familiar with such as Joomla or Wordpress.

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 to 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!

CHAPTER 2

Table of Contents

Benchmarks

Note that benchmarks can be different... this depends on cache system/settings, PC power, software, PHP version, etc...

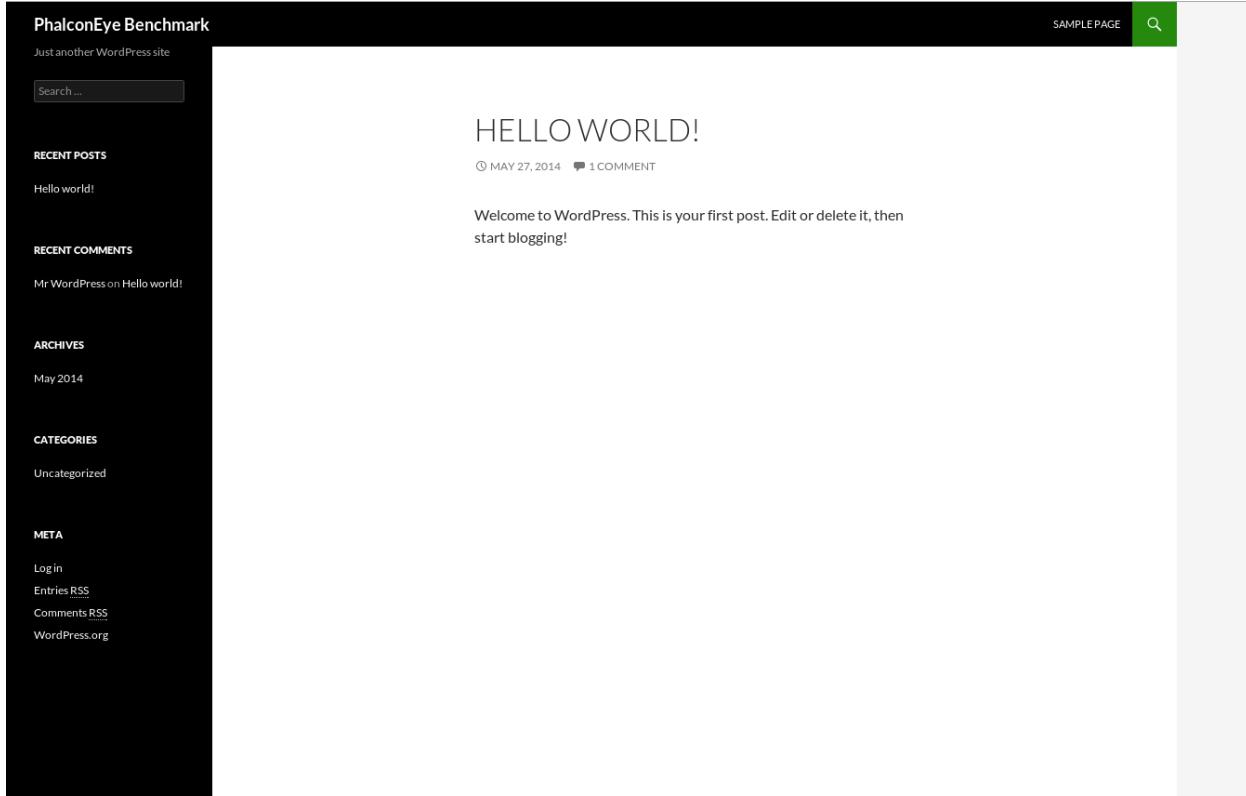
The testing hardware environment:

- Operating System: Linux 3.11.10-11-default openSUSE 13.1 (Bottle) (x86_64)
- Web Server: Apache httpd 2.4.6
- PHP: 5.4.20
- APC: 3.1.15-dev
- CPU: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
- Main Memory: 8GB 1867 MHz DDR3
- Hard Drive: SSD OCZ-AGILITY3 55,9 Gb

Software:

- WordPress 3.9.1
- Joomla 3.3.0
- PhalconEye 0.4.0

WordPress 3.9.1



ab2 test tool:

```

linux-ff8d:/www # ab2 -n 2000 -c 10 http://test.l/
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking test.l (be patient)
Completed 200 requests
Completed 400 requests
Completed 600 requests
Completed 800 requests
Completed 1000 requests
Completed 1200 requests
Completed 1400 requests
Completed 1600 requests
Completed 1800 requests
Completed 2000 requests
Finished 2000 requests

Server Software:      Apache/2.4.6
Server Hostname:     test.l
Server Port:          80

Document Path:        /
Document Length:     7308 bytes

Concurrency Level:   10
Time taken for tests: 17.754 seconds
Complete requests:   2000
Failed requests:     0
Write errors:         0
Total transferred:   15088000 bytes
HTML transferred:    14616000 bytes
Requests per second: 112.65 [#/sec] (mean)
Time per request:    88.772 [ms] (mean)
Time per request:    8.877 [ms] (mean, across all concurrent requests)
Transfer rate:       829.90 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0  0.1     0     3
Processing:    54   89 17.2    84   176
Waiting:       54   89 17.2    84   176
Total:         54   89 17.2    84   176

Percentage of the requests served within a certain time (ms)
 50%    84
 66%    92
 75%    98
 80%   101
 90%   114
 95%   124
 98%   136
 99%   141
100%   176 (longest request)
linux-ff8d:/www #

```

Joomla 3.3.0

PhalconEye Benchmark

Home About Author Login

Welcome to your blog

Details
Written by Joomla

This is a sample blog posting.
If you log in to the site (the Author Login link is on the very bottom of this page) you will be able to edit it and all of the other existing articles. You will also be able to create a new article and make other changes to the site.
As you add and modify articles you will see how your site changes and also how you can customise it in various ways.
Go ahead, you can't break it.

About your home page

Details
Written by Joomla

Your home page is set to display the four most recent articles from the blog category in a column. Then there are links to the 4 next oldest articles. You can change those numbers by editing the content options settings in the blog tab in your site administrator. There is a link to your site administrator in the top menu.
If you want to have your blog post broken into two parts, an introduction and then a full length separate page, use the Read More button to insert a break.

Read more: About your home page

Older Posts

- Welcome to your blog
- About your home page
- Your Template
- Your Modules

Blog Roll

- Joomla Community
- Joomla Leadership Blog

Most Read Posts

- Welcome to your blog
- About your home page
- Your Modules
- Your Template

My Blog

ab2 test tool:

```
Linux-ff8d:/www/ph.l/www # ab2 -n 2000 -c 10 http://test.l/
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking test.l (be patient)
Completed 200 requests
Completed 400 requests
Completed 600 requests
Completed 800 requests
Completed 1000 requests
Completed 1200 requests
Completed 1400 requests
Completed 1600 requests
Completed 1800 requests
Completed 2000 requests
Finished 2000 requests

Server Software:        Apache/2.4.6
Server Hostname:       test.l
Server Port:          80

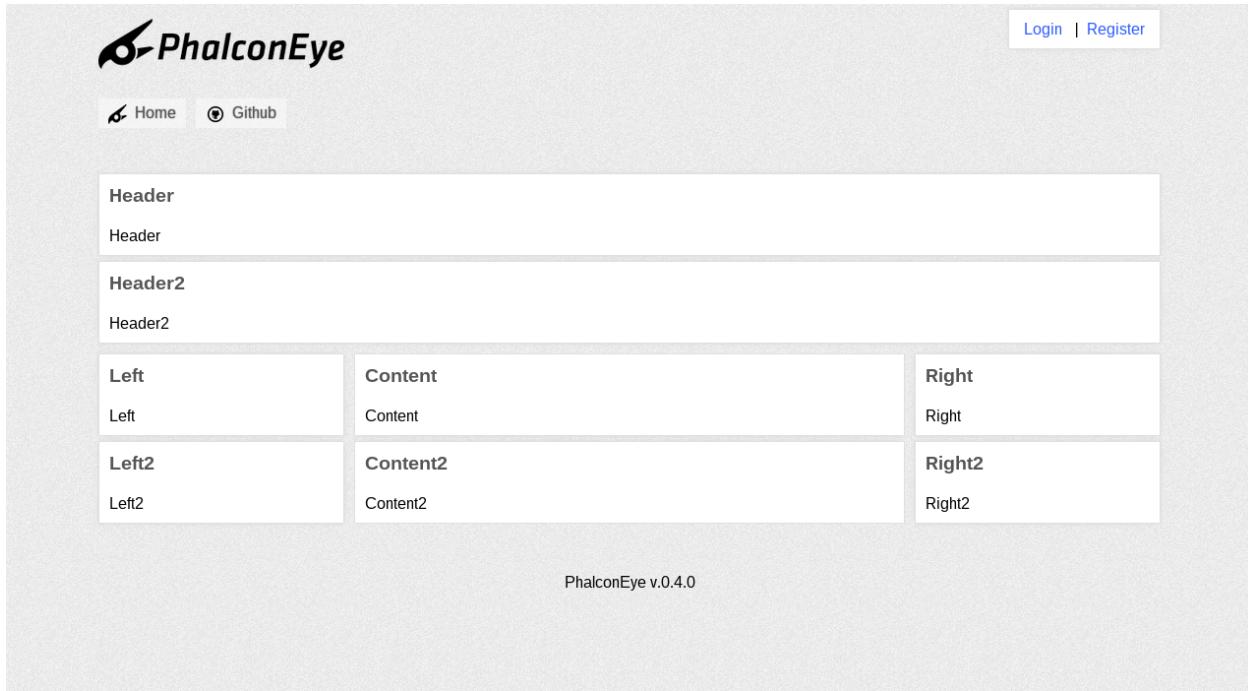
Document Path:         /
Document Length:      16140 bytes

Concurrency Level:    10
Time taken for tests: 30.341 seconds
Complete requests:   2000
Failed requests:     0
Write errors:         0
Total transferred:   33344000 bytes
HTML transferred:    32280000 bytes
Requests per second: 65.92 [#/sec] (mean)
Time per request:   151.705 [ms] (mean)
Time per request:   15.171 [ms] (mean, across all concurrent requests)
Transfer rate:       1073.22 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0    0.4     0    15
Processing:   110   151   26.5    146   265
Waiting:      107   147   25.8    141   257
Total:        110   151   26.6    146   265

Percentage of the requests served within a certain time (ms)
 50%    146
 66%    158
 75%    166
 80%    171
 90%    188
 95%    205
 98%    221
 99%    233
100%   265 (longest request)
linux-ff8d:/www/ph.l/www # █
```

PhalconEye 0.4.0



ab2 test tool:

```

Linux-ff8d:/www/ph.l/www # ab2 -n 2000 -c 10 http://ph.l/
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking ph.l (be patient)
Completed 200 requests
Completed 400 requests
Completed 600 requests
Completed 800 requests
Completed 1000 requests
Completed 1200 requests
Completed 1400 requests
Completed 1600 requests
Completed 1800 requests
Completed 2000 requests
Finished 2000 requests

Server Software:        Apache/2.4.6
Server Hostname:       ph.l
Server Port:          80

Document Path:         /
Document Length:      5852 bytes

Concurrency Level:    10
Time taken for tests: 6.359 seconds
Complete requests:   2000
Failed requests:     0
Write errors:         0
Total transferred:   12542000 bytes
HTML transferred:    11704000 bytes
Requests per second: 314.53 [#/sec] (mean)
Time per request:   31.794 [ms] (mean)
Time per request:   3.179 [ms] (mean, across all concurrent requests)
Transfer rate:       1926.16 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0  0.0     0      0
Processing:    19   32  7.7    30     78
Waiting:       19   32  7.6    30     78
Total:         19   32  7.7    30     78

Percentage of the requests served within a certain time (ms)
 50%    30
 66%    33
 75%    35
 80%    37
 90%    42
 95%    47
 98%    53
 99%    57
100%   78 (longest request)
linux-ff8d:/www/ph.l/www # █

```

Installation

1. Get Phalcon Framework up and running (1.3.1 version is required). See <http://docs.phalconphp.com/en/latest/reference/install.html>
2. If you have cloned PhalconEye from GitHub you must run ant task “ant dist” to get the package as zip. This task creates clean package with preinstalled assets.
3. Extract (or copy) PhalconEye’s code onto your webserver.
4. ‘public’ directory must be set as server’s web root. VirtualHost example for Apache:

```
<VirtualHost *:80>
    ServerAdmin admin@mail.com
    ServerName test.local

    DocumentRoot /www/phalconeye/www/public
    ErrorLog /www/phalconeye/logs/errors.log
    CustomLog /www/phalconeye/logs/access.log combined

    <Directory "/www/phalconeye/www/public">
        Options Indexes FollowSymLinks
        AllowOverride All
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

5. If you have installed the CMS into a subdirectory (eg. <http://youhost.com/phalconeye/>), you will also need to edit configuration in /app/config/development/application.php. Change ‘baseUrl’ to your subdirectory path (ie. ‘/phalconeye/’).
6. Restart Apache server and browse to <http://youhost.com/>
7. Visit the website follow the installation instructions.

User’s guide

Working With A Grid

Grid system allows fast and simple browsing of database rows.

Let’s take a look at main components of the grid:

The screenshot shows the PhalconEye CMS interface for managing users. On the left is a dark sidebar with navigation links like Dashboard, Users and Roles, Pages, Menus, Languages, Files, Packages, System, Performance, Access Rights, and a Blog module. The main area has a blue header with 'Users' and 'Roles' tabs, and buttons for 'Create new user' and 'Create new role'. Below is a grid titled 'Users (2)'.

ID	Username	Email	Role	Creation Date	Actions
2	lantian	lantian@mail.com	User	2014-05-23 21:21:53	Edit Delete
1	dad	ddwa@mail.com	Admin	2014-04-13 16:21:31	Edit Delete

Callouts numbered 1 through 7 point to specific elements:

- Grid title: 'Users (2)'.
- Column headers: 'ID', 'Username', 'Email', 'Role', 'Creation Date', 'Actions'.
- Row index: '2' in the first row.
- Role dropdown: 'User' in the first row.
- Action buttons: 'Filter' and 'Reset'.
- Action buttons: 'Edit' and 'Delete' for the first user.
- Action buttons: 'Edit' and 'Delete' for the second user.

At the bottom right of the grid, it says 'PhalconEye v.0.4.0 [26-05-2014 19:31:28]'

Components description:

1. Grid title.
2. Columns names. Some columns can be a link, some - just a text. When column is a link - you can sort all grid by clicking on this link.
3. Filter fields. Enter value that you want to find (or filter by it) and push button “Filter” or “Enter” on keyboard if field is still in focus.
4. Some fields can be select of different html input type.
5. Filter button will filter by typed data in filter fields. Reset - reset all filters and remove text from fields.
6. Action links, this links can be different, but all of them related to each row.
7. Grid also can have links to item profile.

User management

The CMS enables you to create, edit, delete and browse users in a very simple manner.

Browsing users is implemented via a grid system (built into the CMS):

ID	Username	Email	Role	Creation Date	Actions
2	lantian	lantian@mail.com	User	2014-05-23 21:21:53	Edit Delete
1	dad	ddwa@mail.com	Admin	2014-04-13 16:21:31	Edit Delete

PhalconEye v.0.4.0
[26-05-2014 19:31:28]

To add a new user simply go to “Create new user” from top navigation bar and fill in the form:

Fields description:

Username - user login (nickname)

Password - user password

Email - user email

Role - user role (or category)

To delete user - search for appropriate record via the grid and click “Delete” link located in “Actions” column. You

will be asked to confirm this action

Role management

PhalconEye enables Administrators to define privileged groups of users.

ID	Name	Description	Is default?	Actions
1	Admin	Administrator.	0	Edit
2	User	Default user role.	1	Edit
3	Guest	Guest role.	0	Edit

To add a new Role go to “Create new role” from top navigation bar and fill-in the form:

Role Creation
Create new role.

Name

Description

Is Default

Create **Cancel**

Fields description:

Name - name of the new Role

Description - short description of the Role

Is Default - whether the Role should be assigned to all new users

By default PhalconEye comes with 3 system Roles:

- **Admin** - administrators who can access backend of the CMS
- **User** - this is the default Role for users who register on your website
- **Guest** - all visitors

These system Roles can not be deleted as opposed to new Roles created by Administrators, which can be removed via grid system.

Dynamic pages

Dynamic pages allow Administrators to chose a layout from 12 pre-defined sets and add content without the need for knowledge of any programming language.

The screenshot shows the PhalconEye CMS dashboard with the 'Pages' module selected. The main content area displays a table titled 'Pages (3)' showing three entries: 'Header', 'Footer', and 'Home'. Each entry has columns for ID, Title, Url, Layout (represented by a small preview icon), Controller, and Actions (with 'Manage' and 'Edit' links). The sidebar on the left includes sections for Dashboard, Manage, Users and Roles, Pages, Menus, Languages, Files, Packages, Settings, System, Performance, Access Rights, Modules, and Blog. At the bottom are 'Back to site' and 'Logout' buttons, and a footer note: 'PhalconEye v.0.4.0 [26-05-2014 19:55:07]'.

ID	Title	Url	Layout	Controller	Actions
1	Header				Manage
2	Footer				Manage
3	Home	/			Manage Edit

You can see the type of layout for each page in “Layout” column.

Adding and editing pages

To create a new page - navigate to “Create new page”.

Fields description:

Title - will be used as HTML Title of the page as well as to identify page in grid.

Url - identify how your page will be accessible from the internet. It must be a relative Url so it can not start with ‘http’ or ‘/’. Example 1: If you set Url as “test” the page will be available at address: <http://yoursite.com/page/test> . Example 2: If Url is “some-long-page/with/id/1” a full link to the page will be: <http://yoursite.com/page/some-long-page/with/id/1> .

Description - used for metadata of the page <meta name="description" content="YOURCONTENT">

Keywords - used for metadata of the page <meta name="keywords" content="YOURCONTENT">

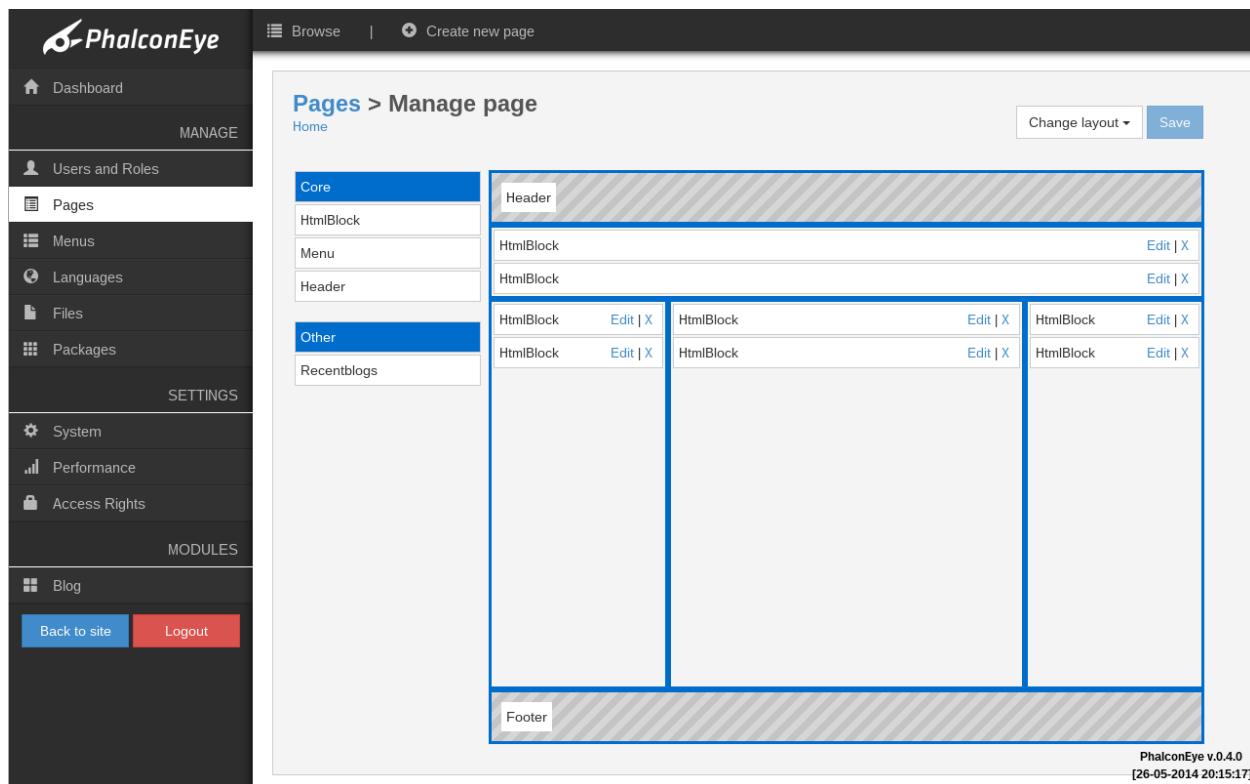
Controller - Developers can use this feature to forward request to specific MVC controller. Imaging that there is ShopPaypalController which performs Paypal checks before page dispatch and WebmoneyController that also performs some Webmoney checks before dispatch. And you have dynamic page with widgets, that have some resolved logic depending on controller. You can set PaypalController->indexAction to perform Paypal checks at this page. This action (and initialization method) will be performed before page rendering.

Roles - Enables you to set up restrictions for given Roles . By default everyone will be able to access the page, that is, if you select none or all of the Roles. Selecting a single Role will only allow access for users assigned to it.

Header and Footer are two specific “Page areas” in the CMS for whose the above settings can not be edited. These as well as the “Home page” are integrated into the CMS and cannot be removed.

Page management

Pages consist of widgets which are the basic build blocks in PhalconEye that perform a specific function such as displaying Menu or Social Icons. You will find widgets in WordPress, Joomla and Drupal call them modules. Widgets can be easily dragged and dropped into a specific widget area within the layout.



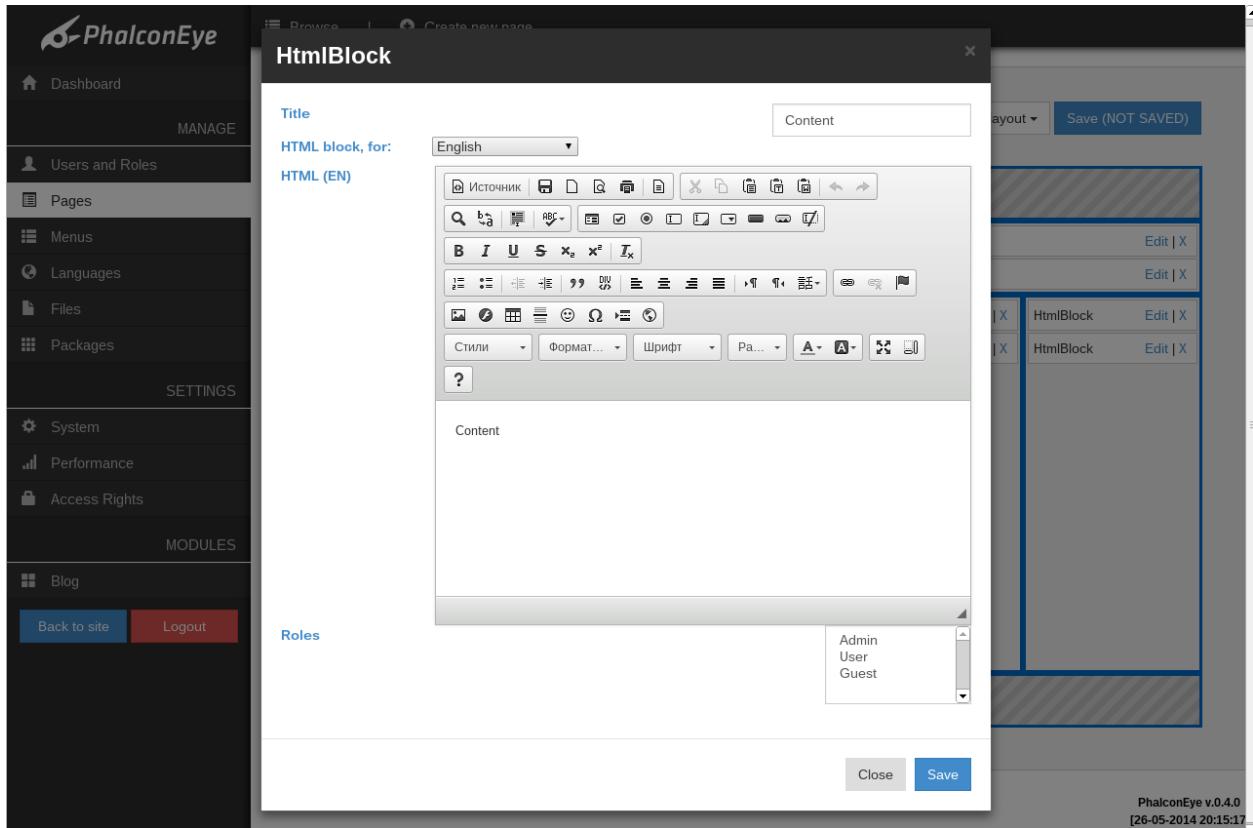
List of available widgets is located on the left of management page. Widgets can be part of modules and be displayed under appropriate module name (eg. Menu and Header widgets belong to Core module). They can also be installed as standalone packages and will be displayed under “Other” (see Recentblogs widget above).

Right next to list of available widget you can see the layout you have chosen for your page - this is the drop area for widgets.

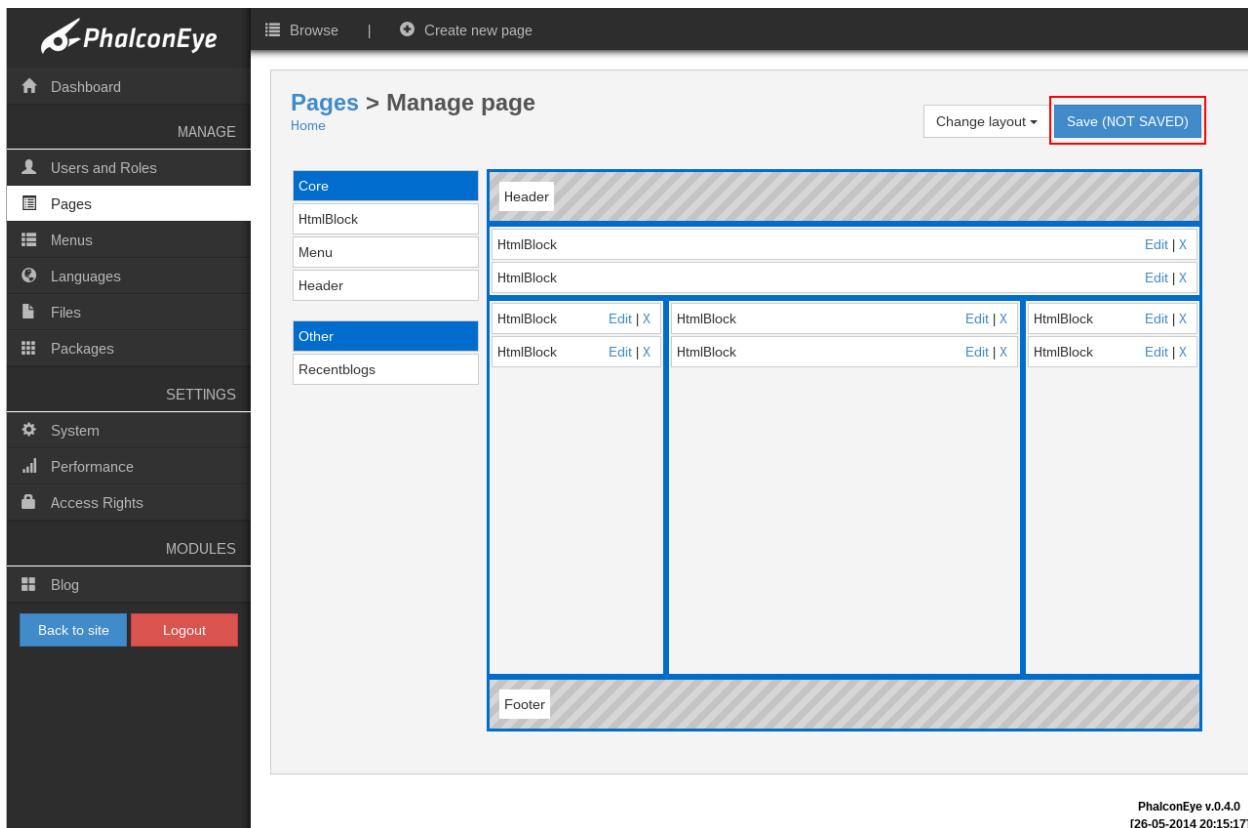
Adding new widget - Simply drag a widget from the left area and drop it onto any part of the layout. It is also possible to re-arrange widgets which have already been dropped by dragging and dropping them elsewhere.

Remove widget - Simply click ‘X’ link from within the widget.

Edit widget - Almost all widgets have their options. To edit them, click ‘Edit’ link within the widget and you will see a form with options:



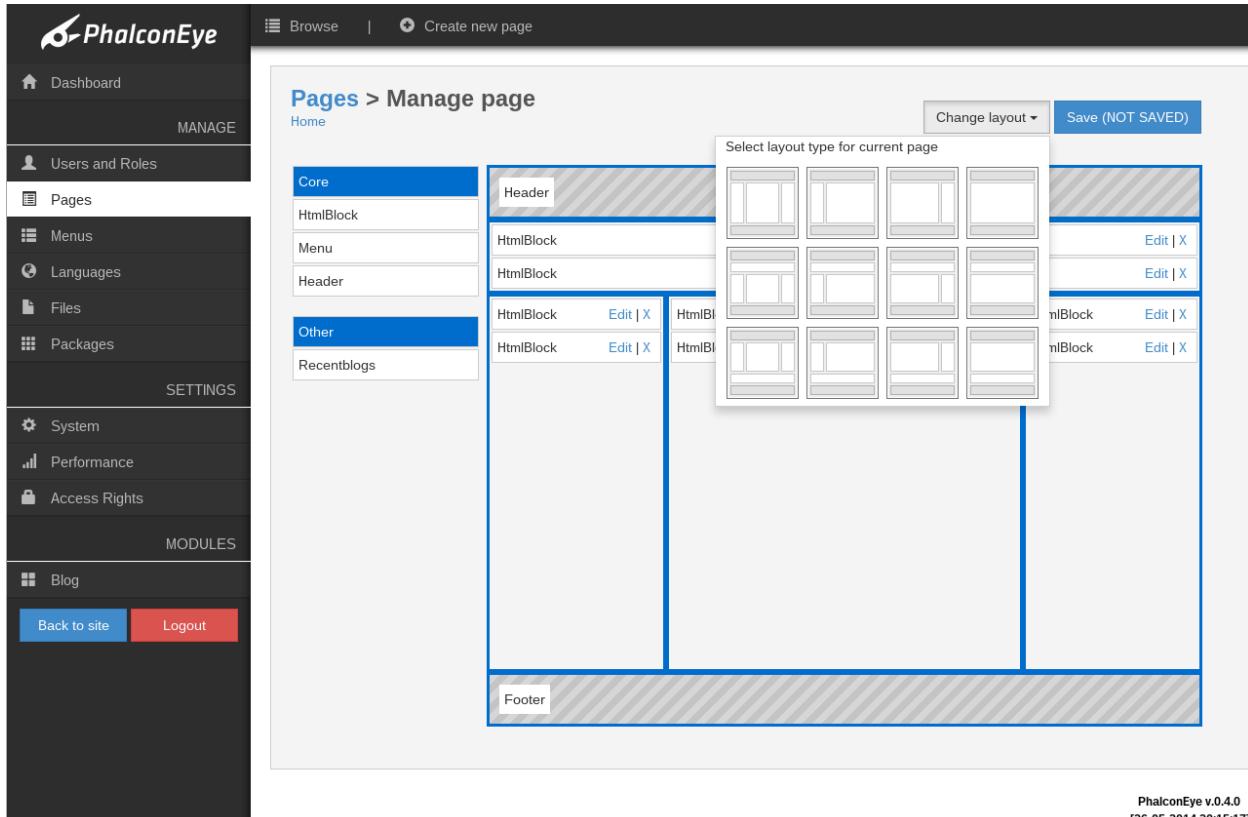
Once you have done editing widget's parameters, “Save” button becomes active. Note that you can configure multiple widgets at the same time, but all the changes will take effect only if you save the layout:



The screenshot shows the PhalconEye administration panel under the 'MANAGE' tab. On the left, a sidebar lists 'Dashboard', 'Users and Roles', 'Pages', 'Menus', 'Languages', 'Files', 'Packages', 'System', 'Performance', 'Access Rights', and 'Blog'. At the bottom are 'Back to site' and 'Logout' buttons. The main area is titled 'Pages > Manage page' and shows a 'Header' section with three 'HtmlBlock' components. Below the header is a 'Footer' section. A 'Core' sidebar on the left contains 'Header' and 'Other' sections. A top navigation bar includes 'Browse', 'Create new page', 'Change layout', and a red-bordered 'Save (NOT SAVED)' button.

PhalconEye v.0.4.0
[26-05-2014 20:15:17]

Page layout can be changed at any time:



This screenshot is identical to the one above, but a modal dialog is open over the 'Header' section. The dialog is titled 'Select layout type for current page' and displays a grid of 12 different layout templates, each showing a preview of a 3-column layout with varying content proportions. The 'Core' sidebar and top navigation bar are visible, along with the 'Save (NOT SAVED)' button.

PhalconEye v.0.4.0
[26-05-2014 20:15:17]

Be careful, though, when doing so! When a newly chosen layout has less columns than current, some of them might be lost permanently with all its widgets and their saved parameters.

To avoid that you can temporarily move elsewhere widgets from the column which is about to be removed.

@TODO: Describe precedence of which columns are removed.

Menus

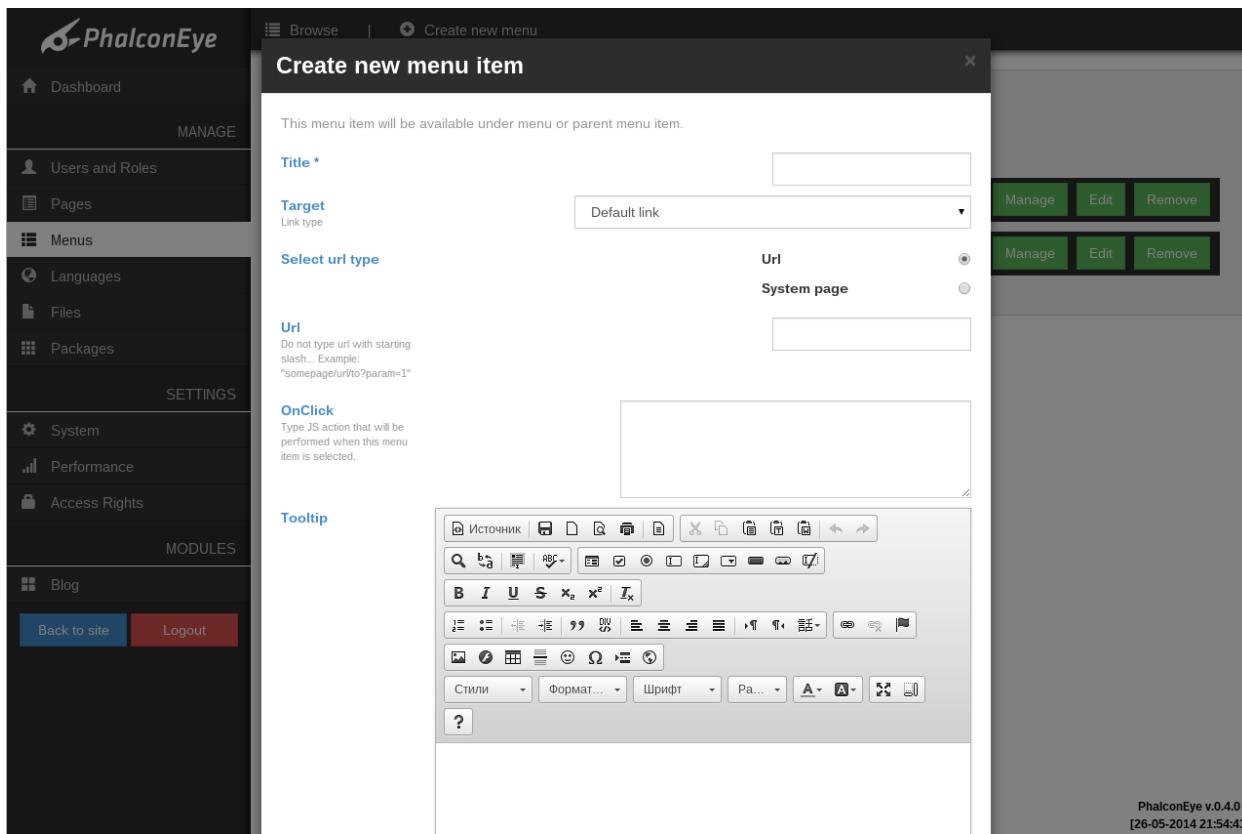
To add a menu click “Create new menu” from the top navigation and give it some unique name. Like in every CMS menus consist of menu items, which can be standalone items or other nested menus.

To manage menu items click “Manage” link from actions column:

ID	Menu title	Actions
1	Default menu	Manage Edit Delete
2	User menu	Manage Edit Delete

Use drag and drop if you need to re-arrange their order. Each menu item can have its own nested items creating a sub-menu tree structure. PhalconEye does not limit the depth of nesting! Feel free to create as complicated tree as you need.

Adding and editing items



Fields description:

Title - name of the menu item.

Target - this is HTML link attribute “target” which defines click behaviour.

Select url type - switch between an absolute (direct) url or link to one of CMS pages.

Url - absolute (direct) link (shown only if url type above is “Url”)

Page - start typing name of a page , a list of potential options will appear (shown only if url type above is “System page”)

OnClick - this is a html attribute - javascript code, which will be executed once the item is clicked

Tooltip - optional tooltip text for the item.

Tooltip position - defines position of the tooltip message: top, left, right, bottom.

Select icon - optionally you can select an image for the item (icon). Note: The icon will not be re-sized so be careful when choosing big images.

Icon position - defines alignment of the icon against item’s Title (either Left or Right)

Languages - defines the target languages for the item on which it will be visible.

Roles - select target Roles for the item. If any Role is chosen the item will be restricted only to users who belong to the Role

Is Enabled - whether or not the item should be visible

Languages

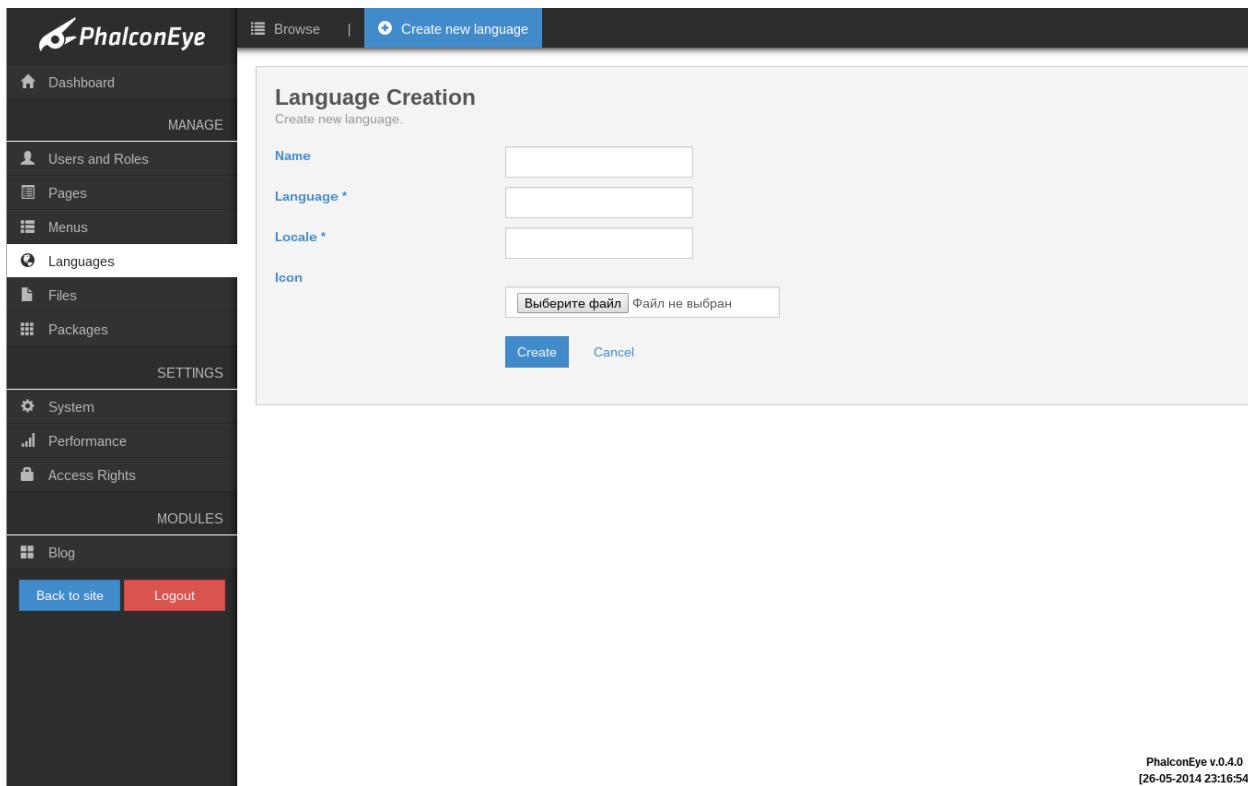
This area enables Administrators to create custom translations for elements on the website such as buttons, commands, alerts and so on...

ID	Name	Language	Locale	Icon	Actions
1	English	en	en_US	No icon	Manage Export
2	German	de	de_DE	No icon	Manage Export Wizard Edit Delete

PhalconEye v.0.4.0
[26-05-2014 23:03:37]

Adding a language

To add a new language - click “Create new language” from top navigation.



Fields description:

Name - system name of the language, whatever you prefer

Language - 2 character language code (eg. English - “en”, German - “de”, Russian - “ru”)

Locale - 5 character locale code (eg. en_US or en_EN). You can create two locales for the same languages.

Icon - not mandatory but might be used by some modules or widgets. Icon can represent a country flag.

Performance note

Custom translations are kept in database for simplicity of development. In production mode, however, translations are compiled into a php file to avoid database performance overhead. To re-compile available translations click “Compile languages” and wait until it completes.

Export

The screenshot shows the PhalconEye admin interface. On the left, there's a sidebar with 'MANAGE' and 'SETTINGS' sections, and a 'MODULES' section containing 'Blog'. The main area has a 'Browse' button and a 'Create new language' link. A modal window titled 'Export translations' is open, showing a 'Scope' dropdown with 'Blog', 'Core', and 'User' selected. Below the modal, a table lists translations for the German language ('German' column, 'de' column). The table includes columns for 'Icon' (showing 'No icon'), 'Actions' (with links for 'Manage Export Wizard', 'Edit', and 'Delete'), and a timestamp '26-05-2014 23:19:28'. At the bottom right of the interface, it says 'PhalconEye v0.4.0 [26-05-2014 23:19:28]'.

You can easily export a language you have created into a JSON file and import it into another instance of PhalconEye. Scope defines logical separation of the language to export only certain part of translations (e.g. Blog - only translations used in Blog module will be exported).

Import

Enables administrators to import translations for other languages. Simply, click “Import” and select a JSON file file with translations. After short period of time translations will be imported and you will see message about the result. Once finished, do not forget to compile them!

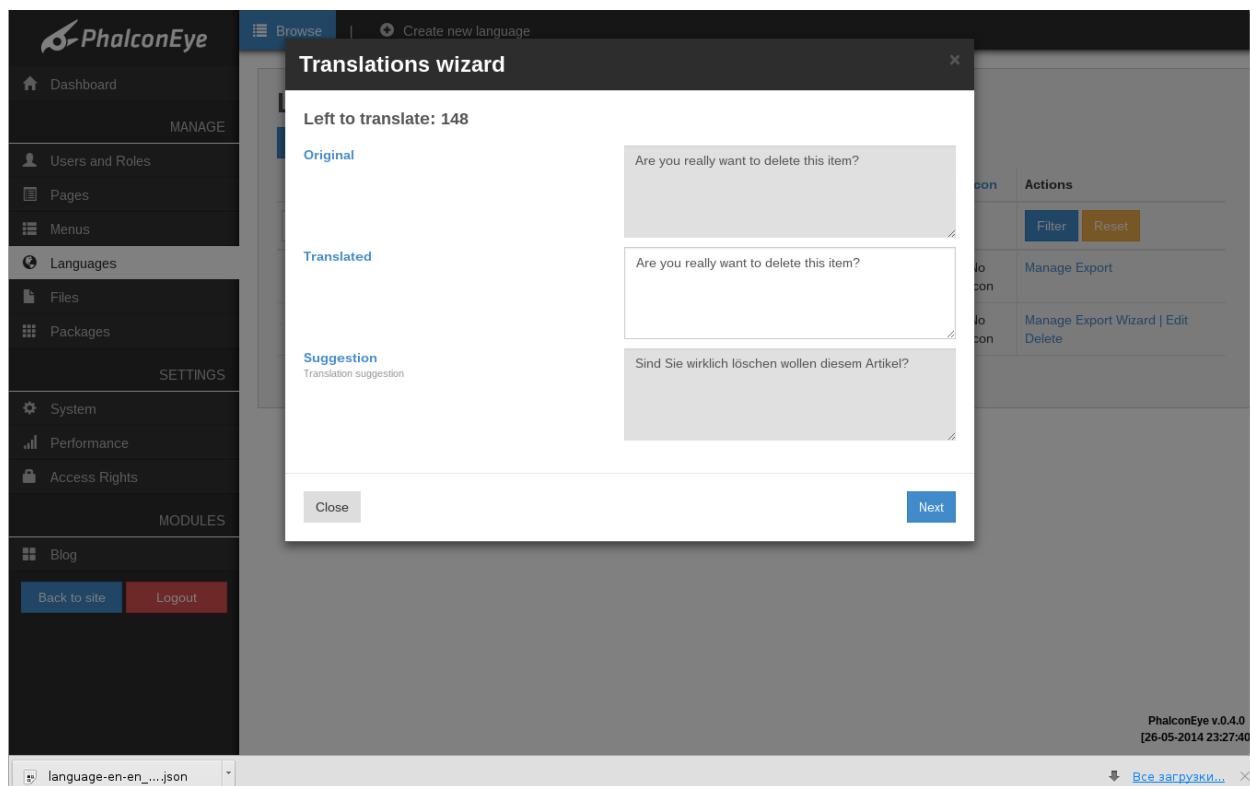
Manage Translations

You can manage translation by clicking “Manage” within a row.

1. Show only untranslated rows.
2. Synchronize current language with English. English is the default system language and all internal contents are in English. When system generates translations they are automatically added to English subset (it happens automatically in development mode). If, for instance, current subset of English language has got 100 translations and your German language only 80, use synchronization to copy untranslated records to the German subset.
3. Search box will search for any occurrences of the text (in both original and translated subsets).
4. Italic Text marked in red - is untranslated.

Wizard

Sped up translations with wizard! It will give you the original text, translation and a suggestion. The suggestion is automated on en -> current_language by Yandex translation API. By clicking “Next” button you will save the translation field to database.



File management

File management is implemented via Pydio software:

The screenshot displays two main interfaces side-by-side. On the left is the PhalconEye administrative dashboard, featuring a dark theme with white text and icons. It includes a sidebar with 'Dashboard' at the top, followed by 'MANAGE' and several sections: 'Users and Roles', 'Pages', 'Menus', 'Languages', 'Files', 'Packages', 'SETTINGS' (with 'System', 'Performance', and 'Access Rights' options), and 'MODULES' (with 'Blog'). At the bottom of the sidebar are 'Back to site' and 'Logout' buttons. On the right is a Pydio file manager interface titled 'Common Files'. The top bar of Pydio includes 'Workspaces', 'Logged as shared', a search bar, and a 'More' dropdown. The main area shows a tree view of 'Common Files' containing 'languages', 'Recycle Bin', and 'BC_4_1.zip'. A detailed view of 'BC_4_1.zip' is expanded, showing its contents: 'PE_logo.png', 'github.gif', and 'Recycle Bin'. The file 'BC_4_1.zip' itself is highlighted with a blue background. The right panel provides file metadata: Name (BC_4_1.zip), Size (2.45 Kb), Last Modif. (2014/03/25 15:44), Type (ZIP file), and Meta Data. The bottom right of the Pydio interface displays the text 'PhalconEye v.0.4.0 [26-05-2014 22:34:51]'.

You can read more about Pydio in .

System settings

PhalconEye

Dashboard

MANAGE

Users and Roles

Pages

Menus

Languages

Files

Packages

SETTINGS

System

Performance

Access Rights

MODULES

Blog

Back to site

Logout

System settings
All system settings here.

Site name: Phalcon Eye

Theme: Default

Default language: English

Save

PhalconEye v.0.4.0
[26-05-2014 22:45:02]

Fields description:

Site name - name of your website, which will be used as prefix in browser's title bar.

Theme - select front-end theme.

Default language - default front-end language. If "Auto-detect" is selected PhalconEye will try to figure out users' locale and set appropriate language.

Performance settings

Performance form allows to setup some improvements in speed.

Performance settings

Cache prefix
Example: "pe_"

Cache lifetime
This determines how long the system will keep cached data before reloading it from the database server. A shorter cache lifetime causes greater database server CPU usage, however the data will be more current.

Cache adapter
Cache type. Where cache will be stored.

Files location

Clear cache
All system cache will be cleaned.

Save

PhalconEye v.0.4.0
[26-05-2014 22:47:36]

Fields description:

Cache prefix - cache prefix in system, required if you use shared caching system with other websites.

Cache lifetime - cache TTL (time to live), maximum lifetime of cached data.

Cache adapter - you can select adapter for the cache: file, memcached, apc, mongo. Default is file and others require additional PHP extensions.

Clear cache - select to clear current cache.

If “File” adapter is selected:

Files location - path to a folder where cached data will be stored.

If “Memcached” is selected:

Memcached host - host address of the server.

Memcached port - server port.

Create a persistent connection to memcached - will create per-request persistent connection.

If “APC” adapter has no additional settings.

If “Mongo” adapter is selected:

A MongoDB connection string - mongo connection string (read about mongo).

Mongo database name - database name.

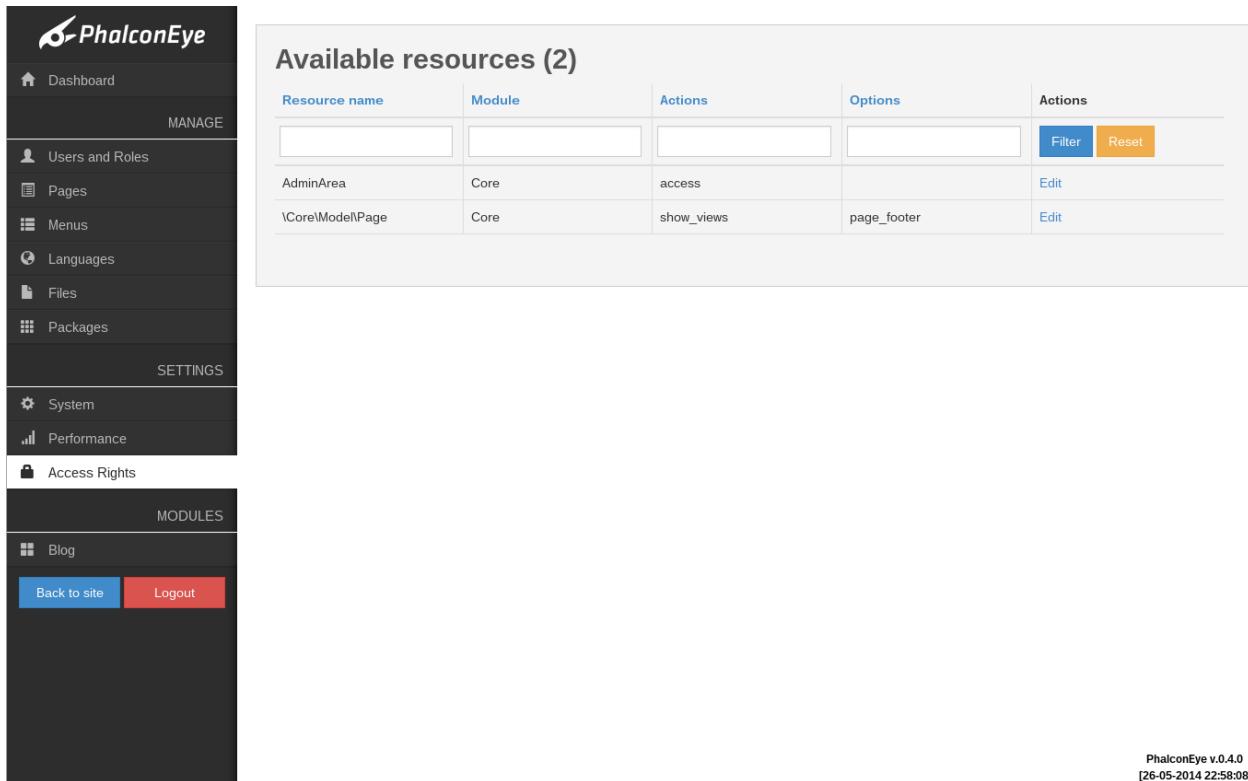
Mongo collection in the database - mongo collection.

Access Rights

Access Rights allows to control access restrictions for specific areas of the CMS. These Resources and their access policies may be defined by modules.

By default, the Core module comes with two Resources:

- “AdminArea”
- “CoreModelPage”



The screenshot shows the PhalconEye CMS dashboard. On the left, there's a sidebar with navigation links: Dashboard, MANAGE (Users and Roles, Pages, Menus, Languages, Files, Packages), SETTINGS (System, Performance), and MODULES (Blog). At the bottom of the sidebar are 'Back to site' and 'Logout' buttons. The main content area has a title 'Available resources (2)'. Below it is a table with the following data:

Resource name	Module	Actions	Options	Actions
AdminArea	Core	access		Edit
\Core\Model\Page	Core	show_views	page_footer	Edit

At the bottom right of the main area, it says 'PhalconEye v.0.4.0 [26-05-2014 22:58:08]'

CoreModelPage Resource has two configurable actions:

The screenshot shows the PhalconEye Admin Dashboard. On the left is a sidebar with the following navigation:

- Dashboard
- MANAGE**
- Users and Roles
- Pages
- Menus
- Languages
- Files
- Packages
- SETTINGS**
- System
- Performance
- Access Rights** (selected)
- MODULES**
- Blog

At the bottom of the sidebar are two buttons: "Back to site" (blue) and "Logout" (red).

The main content area is titled "Access Rights > Editing access rights of "\Core\Model\Page", for: Admin". It contains two sections:

- Actions**: A list of actions with checkboxes:
 - Show_views (ACTION_CORE_MODEL_PAGE_SHOW_VIEWS_DESCRIPTION)
- Options**: A section for "Page_footer" (OPTION_CORE_MODEL_PAGE_PAGE_FOOTER_DESCRIPTION) with a text input field.

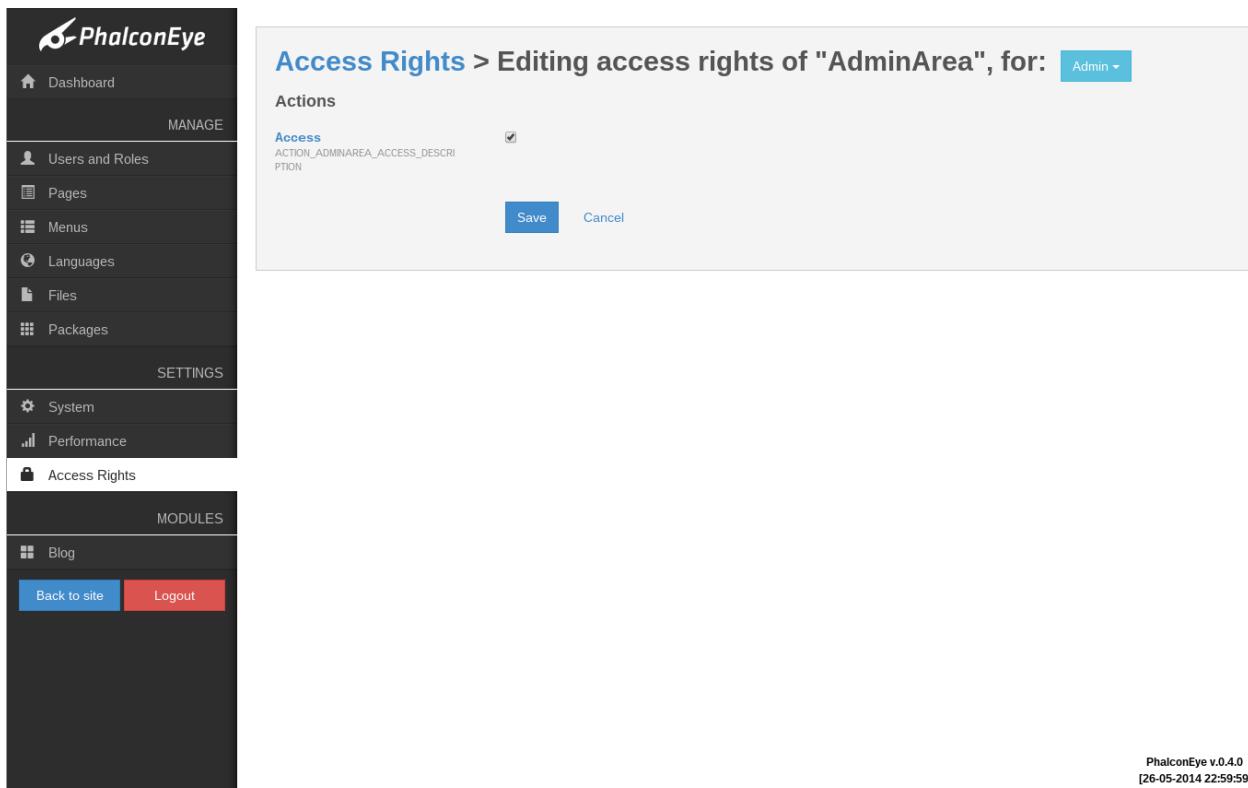
At the bottom right of the main content area are "Save" and "Cancel" buttons.

In the bottom right corner of the dashboard, there is a small text: "PhalconEye v.0.4.0 [26-05-2014 22:57:46]".

- “Show_views” - displays visits counter of on current page.
- “Page_footer” - stores additional HTML code.

These actions can be assigned individually to each Role, (“Admin” Role in this case) visible in the top right corner.

To make it clear look at another example:



Here, defined backend access rights for role “Admin” - it will let Admin users to access the Administration area.

Texts like “ACTION_ADMINAREA_ACCESS_DESCRIPTION” are translation placeholders and can be changed in language management system.

Developer’s guide

CMS Structure

Let’s look on project structure...

```

PhalconEye
- app                                     // Application source code.
|   - config                                // Main configuration
|   ↵directory.
|   |   - development
|   |   - production
|   - engine                                 // Engine library directory,
|   ↵ heart of PhalconEye.
|   |   - Api
|   |   - Asset
|   |   |   - Css
|   |   - Behaviour
|   |   - Cache
|   |   - Console
|   |   |   - Command
|   |   - Db
|   |   |   - Model
|   |   - Exception

```

```

|   |   - Form
|   |   |   - Behaviour
|   |   |   - Element
|   |   |   - Validator
|   |   - Grid
|   |   |   - Behaviour
|   |   |   - Source
|   |   - Helper
|   |   - Package
|   |   |   - Exception
|   |   |   - Model
|   |   |   - Structure
|   |   - Plugin
|   |   - Translation
|   |   - View
|   |   - Widget
|   - libraries                                // Directory for packages
↳with type "library", that can be installed.
|   - modules                                    // Directory for "module"
↳packages.
|   |   - Core                                     // Required module: Core.
|   |   |   - Api
|   |   |   - Assets
|   |   |   - Command
|   |   |   - Controller
|   |   |   - Form
|   |   |   - Helper
|   |   |   - Model
|   |   |   - View
|   |   |   - Widget
|   |   - User                                     // Required module: User.
|   |   |   - Controller
|   |   |   - Form
|   |   |   - Helper
|   |   |   - Model
|   |   |   - View
|   |   |   - Widget
|   - plugins                                    // Plugins directory
↳(packages).
|   - var                                         // This is work directory
↳contains: logs, cache, packages operation results, languages, etc.
|   |   - cache
|   |   |   - annotations
|   |   |   - languages
|   |   |   - metadata
|   |   |   - view
|   |   - logs
|   |   - temp
|   - widgets                                     // Widgets directory
↳(packages).
- public                                         // Public directory, this
↳directory can be accessed through internet, must be set as www root.
    - assets
    - themes

```

We can separate project on two logical parts:

App directory

Configuration

This directory contains different configuration stages. More info you can find in other *section* of documentation.

Engine

Engine directory is heart of CMS. Here under the hood main horsepower =).

Libraries

Libraries that can be installed via packages system. Read more information about *packages* in other section.

Modules

Modules are also part of packages. But here we can find default two modules: Core and User.

Core module contains code for general classes of CMS. Also it implements admin panel logic.
User module pointed on work with users.

Plugins

One more type of package. *Plugins* exist for defining additional logic by system events (events handlers).

Var

Var directory is for main garbage =). Contains: logs, temp files (that currently processing), system data (metadata about packages), cache.

Widgets

And again - package. *Widget* is main composition object of dynamic pages. Widgets used by end user for structuring his dynamic page. For example: html widget, when end user want to add some html to his page.

Public directory

As normal practice public directory must be defined as WWW_ROOT. Public directory allows to store static files like assets and user files. Here we can find some directories by default:

- assets
- external
- files
- themes

Assets

Assets files could be any files required by your application (css, js, images, etc). This directory is controlled by CMS. Don't try to copy there any of your files. Assets files installing to this directory directly from modules (each module has it's own directory with Assets files). Files under modules can't be accessed via http, so, they are installed here for farther usage.

External

This directory for external libs and programs. Here we can find jQuery files, Bootstrap css and other.

Files

This directory used for different public files. Also it's used by Ajaxplorer tool.

Themes

CMS themes files. Here can be directories with different files formats (less, css, images, etc). More detail information you can find in other section, which focused on [themes](#).

Configuration

Stages

Stages allow you to set different configuration per virtual host, server, etc. By default, PhalconEye comes with two stages: "development" and "production". CMS is bundled with "development" stage, this is defined in '/public/.htaccess' file:

```
SetEnv PHALCONEYE_STAGE development
```

If a stage isn't defined "production" will be used. You are allowed to add as many stages as you want, just create new directory inside 'config' folder and set default configuration.

Config files

Configuration contains two default files, which are necessary for the system:

1. application.php - contains main settings for application (cache, view, logging, etc). You can find most of these settings in .

```
<?php
return array(
    'debug' => true,                                // Use debug mode?
    'profiler' => true,                             // Show profiler for admins?
    'baseUrl' => '/',                               // Base site url.
    'cache' =>
        array(
            'lifetime' => '86400',
            'prefix' => 'pe_',
            'adapter' => 'File',
            'cacheDir' => ROOT_PATH . '/app/var/cache/data/',
        )
);
```

```

        ),
'logger' =>
    array(
        'enabled' => true,
        'path' => ROOT_PATH . '/app/var/logs/',
        'format' => '[%date%][%type%] %message%',
    ),
'view' =>
    array(
        'compiledPath' => ROOT_PATH . '/app/var/cache/view/',
        'compiledExtension' => '.php',
        'compiledSeparator' => '_',
        'compileAlways' => true,
    ),
'session' =>
    array(
        'adapter' => 'Files',
        'uniqueId' => 'PhalconEye_',
    ),
'assets' =>
    array(
        'local' => 'assets/', // Local assets location.
        'remote' => false, // This can be used for your CDN.
    ),
'metadata' =>
    array(
        'adapter' => 'Files',
        'metaDataDir' => ROOT_PATH . '/app/var/cache/metadata/',
    ),
'annotations' =>
    array(
        'adapter' => 'Files',
        'annotationsDir' => ROOT_PATH . '/app/var/cache/annotations/',
    )
);

```

2. database.php - contains info about database.

```

<?php
return array(
    'adapter' => 'Mysql',
    'host' => 'localhost',
    'port' => '3306',
    'username' => 'root',
    'password' => NULL,
    'dbname' => 'phalconeye',
);

```

Behaviour

All *.php files under stage directories are merged into one structure. For example we can have the following files:

```

.
-- development
- application.php
- yourconfig.php

```

```
- yourconfig2.php  
- database.php
```

It means that in “development” stage we would be able to get the values as follows:

```
<?php  
  
$config = $this->getDI()->getConfig();  
  
// Get debug mode.  
$isDebug = $config->application->debug;  
  
// Get database adapter.  
$adapter = $config->database->adapter;  
  
// Get custom config value.  
$someValue = $config->yourconfig->someValue;
```

If current stage isn’t “development” merged configuration will be cached in /app/var/cache/data/config.php file.

Packages

Package allows to create modular functionality and share it with community or just use in flexible projects that you can develop. Package can be exported from system as zip archive and will have manifest file and directory with source code, for example, module:

```
.  
- package  
| - Assets  
| - Command  
| - Controller  
| - Form  
| - Helper  
| - Model  
| - Plugin  
| - View  
| - Widget  
| - Bootstrap.php  
| - Installer.php  
- manifest.json
```

Manifest file is required for information about package:

```
{  
    "type": "module",  
    "name": "blog",  
    "title": "Blog",  
    "description": "PhalconEye Blog module.",  
    "version": "0.1.0",  
    "versioning": "x.x.x and x.x.x.x",  
    "author": "PhalconEye Team",  
    "of this package?  
    "website": "http://phalconeye.com/",  
    "dependencies": [  
        "set some relation to other packages, to make sure that your package will work as  
        follows.  
    {
```

```

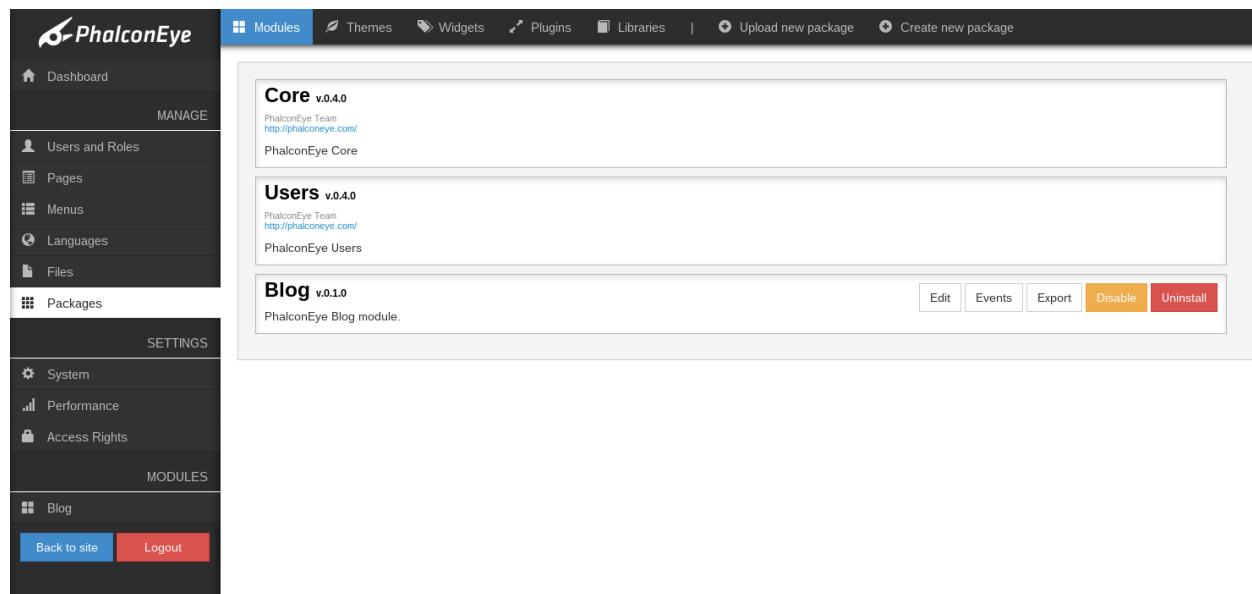
        "name": "core",
        "type": "module",
        "version": "0.4.0"
    },
    {
        "name": "user",
        "type": "module",
        "version": "0.4.0"
    }
],
"events": [
    /*
        Events, that must be attached
        Usage: Namespace\ClassName =
        Note: class
        Blog\Plugin\SomePluginDir\SomeName must be located in related directory and has
        correct name and namespace.
    */
    "Blog\\Plugin\\SomePluginDir\\SomeName=dispatch:beforeDispatchLoop",
    "Blog\\Plugin\\Core=init:afterEngine"
],
"widgets": [
    /*
        Widgets that related to this
        module.
    {
        "name": "recentblogs",
        "module": "blog",
        /*
            Widget unique name.
            Widget related to module, it's
        name.
        "description": "Recent blogs",
        "is_paginated": "1",
        /*
            Description of this widget.
            Flag: is there are any
        pagination?
        "is_acl_controlled": "1",
        /*
            Flag: must be controlled by
            ACL? Adds multiselect box for widget options, that allows to select allowed roles.
        "admin_form": null,
        /*
            Admin form that can control
            and display widget options, can be:
            null - no options or default,
            "action" - widget controller
            "Some/Namespace/Form/ClassName
        has adminAction method,
        /*
            - Form class that must be rendered.
        "enabled": true
    }
],
"i18n": [
    {
        /*
            Localization for this module.
            Separated by packages
        (different languages).
        "info": "PhalconEye Language Package",
        "version": "0.4.0",
        "date": "28-Apr-2014 21:10",
        "name": "English",
        "language": "en",
        "locale": "en_US",
        "content": {
            /*
                Language package short info.
                System version.
                Date of this package.
                Language name.
                Language unique identification.
                Language locale.
                Content of language (it's
            translations).
            "blog": {
                "Home": "SweetHome"
                /*
                    Key : Value (Original :
                    Translated)
            }
        }
    }
]

```

```
        }
    }
},
{
    "info": "PhalconEye Language Package",
    "version": "0.4.0",
    "date": "28-Apr-2014 21:10",
    "name": "German",
    "language": "de",
    "locale": "de_DE",
    "content": {
        "blog": {
            "Home": "Zuhause"
        }
    }
}
]
```

Package Manager

Package manager allows you to create, edit, delete, export packages that you are developing.
As you can see from image, there are several types of package:



Simple “HOW TO”:

Create new package

If you need to create new package, you can use PhalconEye tool - Package manager. It will allow you to create structured package.

Go to Admin panel -> Packages (left sidebar) -> Create new package (top menu). Fill form with your data:

Package Creation
Create new package.

Name
Name must be lowercase and contains only letters.

Package type

Title

Description
Blog module.

Version
Type package version. Ex.: 0.5.7

Author
Who create this package? Identify yourself!

Website
Where user will look for new version?

Header comments
This text will be placed in each file of package. Use comment block `/** */`.

Form data:

- **Name** - Package unique name. Used to form package directory and other identification stuff.
- **Package type** - Package type: module, widget, theme, plugin, library.
- **Title** - Package title, that will be represent your package.
- **Description** - Package description, short description of your package.
- **Version** - What version? Mask: x.x.x or x.x.x.x.
- **Author** - Who is author?
- **Website** - Your website.
- **Header comments** - This comments will be placed in all files, that will be generated, you can use it for your license.

Create “widget” package

If you will select package type “Widget” you will see that form has additional fields:

Header comments

This text will be placed in each file of package. Use comment block `/** */`.

```
/**  
Some License @ 2014  
Some text  
**/
```

Widget information

Is related to module?

Is Paginated?

If this enabled - widget will has additional control enabled for allowed per page items count selection in admin form

Is ACL controlled?

If this enabled - widget will has additional control enabled for allowed roles selection in admin form

Admin form

Does this widget have some controlling form?

Enabled?

Create
Cancel

Additional data:

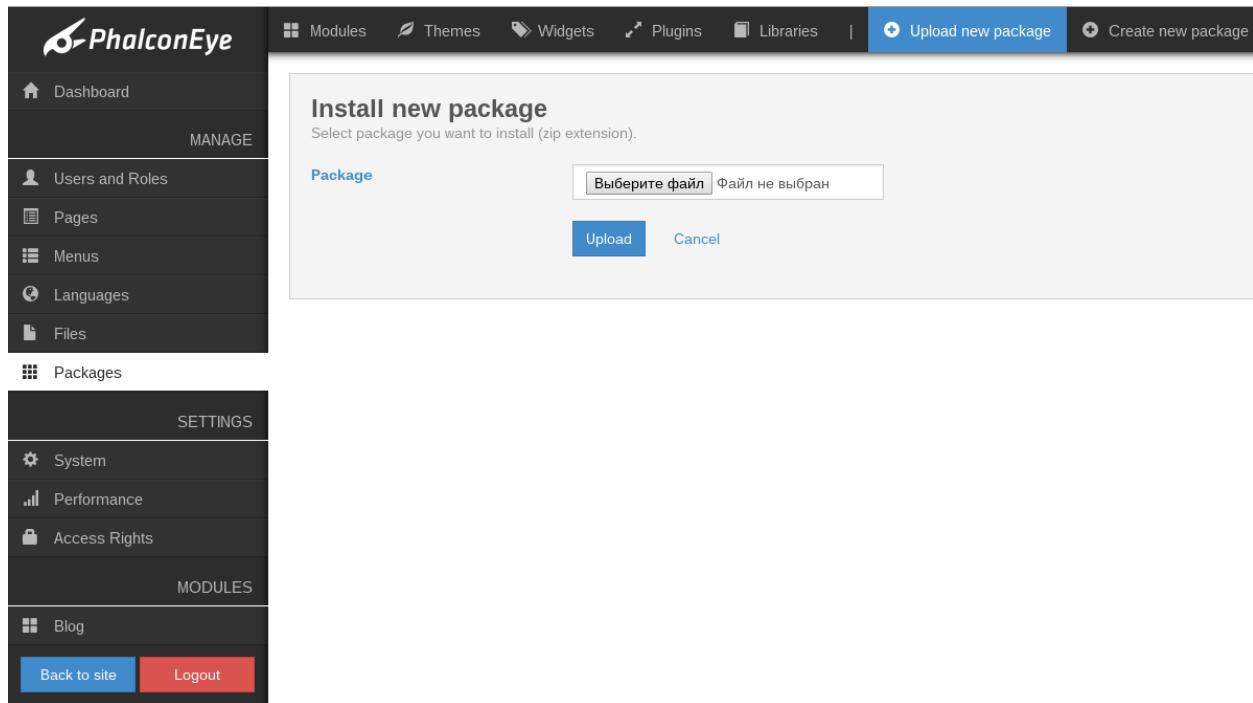
- **Is related to module?** - Here you can select related of your widget. Selectbox listed all current installed modules and has option to choose if this widget is related or not.
- If you will select option “NO” - you widget will be placed at /app/widget/<YOUWIDGETID> and marked as non-related.
- If you choose some module - you widget will be placed at /app/modules/<MODULE THAT YOU SELECTED>/Widget/<YOUWIDGETID>.
- **Is paginated?** - Mark widget as paginated. If you will check this - widget admin options form will have text box, where user will be able to enter amount of data that must be showed in frontend. At widget controller you will be able to check if this option and get this amount for your paginator.
- **Is ACL controlled?** - Same as “Is paginated” - adds multiselect box that will allow to select when this widget must be shown (which role can do this). Unlike “Is paginated” - ACL is controlled by system, and will not need to implement is in widget controller.
- **Admin form** - Here we have three options:
 - No (will be used default admin form with widget block title, paginator option and ACL if they enabled),
 - “Action” (each time when user will request widget admin form - adminAction will be used from widget controller, this action must return Form object).
 - “Form class” (type form class, that must be processed each time when user requests widget admin form).

options form, e.g.: “SomeNamespaceFormClass”.

- **Enabled?** - This widget is enabled, and enduser can use it?

Upload existing package

Upload (Installation) is very simple. Just select your package (*.zip file) and push “Upload” button. If package is broken or your system has no required dependencies error message will be shown.

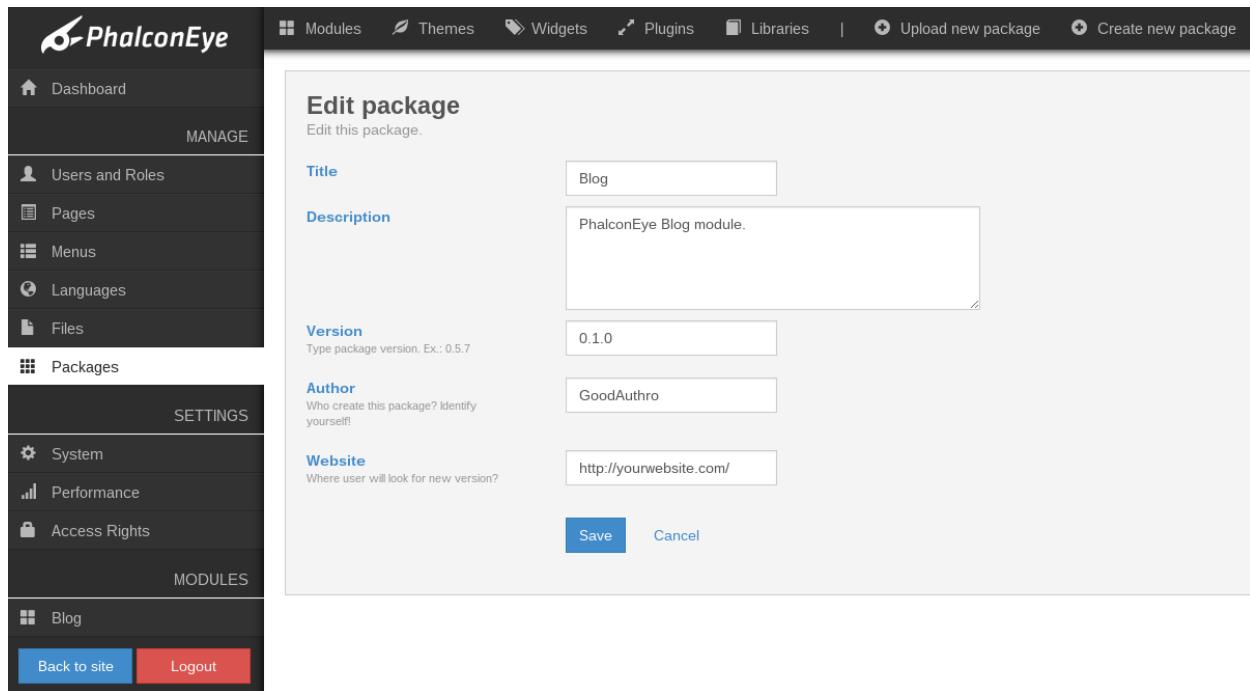


Edit, disable, uninstall package

Assume, that you have created new package and now you need to change some information (for e.g. version) or you need to create *.zip package from it.

Edit

You can edit information about your package. Edit form is limited, but you can change “meta” info about it.



Disable

You can disable package, it is possible from list. Go to Package manager -> <Package type> -> Find it in list and click “Disable”/“Enable” button. There are some limits for this functionality:

- Package which has dependencies can not be disabled or uninstalled.
- Package that is disabled - will not be loaded to PHP autoloader, this means that you will lose it's features.

Uninstall

To uninstall your package, please use Package manager or you risking to broke you system.

Go to Package manager -> <Package type> -> Find it in list and click “Uninstall” button. If package has dependencies system will not allow you to uninstall it.

Export package

Assume, that you has finished your package and want to distribute it.

Go to Package manager -> <Package type> -> Find it in list and click “Export” button.

The screenshot shows the PhalconEye administrative interface. On the left, there's a sidebar with 'Dashboard', 'MANAGE' (with 'Users and Roles', 'Pages', 'Menus', 'Languages', 'Files', and 'Packages'), 'SETTINGS' (with 'System', 'Performance', 'Access Rights'), and 'MODULES' (with 'Blog'). Below these are 'Back to site' and 'Logout' buttons. The main area is titled 'Edit package' and says 'Edit this package.' It has fields for 'Title' (set to 'Blog'), 'Description' (set to 'PhalconEye Blog module.'), 'Version' (set to '0.1.0'), 'Author' (set to 'GoodAuthro'), and 'Website' (set to 'http://yourwebsite.com/'). At the bottom are 'Save' and 'Cancel' buttons.

Export form has some fields:

- **Modules** - You can select from list modules on which your module depends.
- **Libraries** - You can select from list libraries on which your module depends.
- **Export with translations** - If you will check this option, package metadata.json file will contains translations from you package, that you was collected during development.

Package types

There are several types of packages, let's look on each of them:

Module Package

Modules - is main package type. It can be represented as standalone subsystem and can hold widgets, plugins, libraries, but not themes.

```
.
-
- package
|   - Api
|   - Assets
|   - Command
|   - Controller
|   - Form
|   - Helper
|   - Model
|   - Plugin
|   - View
|   - Widget
|   - Bootstrap.php
|   - Installer.php
- manifest.json
```

Let's look at all aspects of module structure:

Module Api

This API is just like a DI wrapper that allows to create API classes at module level. You can access to known API using module key in DI.

For example: Core module has class Core\Api\Acl that can be accessed in such way:

```
<?php

// Note: getObject is a method of class Core\Api\Acl

// In controller (or in any Phalcon\DI\Injectable):
$this->core->acl()->getObject($id);

// In any other place:
$this->getDI()->get("core")->acl()->getObject($id);

// Structure:
$this->getDI()->get("MODULE_NAME")->API_CLASS()->API_CLASS_METHOD
```

By the way... all objects that were accessed stay at DI with key of its namespace...

```
<?php

$this->core->acl()->getObject($id);

// Note that "Core\Api\Acl" key will be in DI only if you call it special API wrapper,
// by default it's not initialized.
// So this like a cache usage without wrapper, in such way you can trigger when your
// API was used.
$this->getDI()->get("Core\Api\Acl")->getObject($id);
```

Assets

Assets directory goal - all module must have its own frontend styles and visualization, so Assets must have (required) 3 directories: css, img, js. This directories are used as assets source for overall assets compilation.

```
/app/modules/Core/Assets
- css
|   - admin
|   - install.less
|   - profiler.less
- img
|   - admin
|   - grid
|   - loader
|   - misc
|   - pe_logo.png
|   - pe_logo_white.png
|   - profiler
- js
|   - admin
|   - core.js
```

```

|   - form
|   - form.js
|   - i18n.js
|   - pretty-exceptions
|   - profiler.js
|   - widgets
- sql
    - installation.sql

```

If you will look at public directory you will find “assets” directory there. This directory is accessible through web server. It contains all assets collected from installed modules, for example, if we have 3 modules core, user and blog we will have such structure:

```

public
- css
  - core
    |   - somestyle.css           // Originally this file is located at /app/modules/
  ↵Core/Assets/css/somestyle.css
  - user
    |   - somedir
      |   |   - someotherstyle.css // This file is located at /app/modules/User/Assets/
  ↵css/somedir/someotherstyle.css
    |   - somestyle.css
  - blog
    |   - somestyle.css
  - constants.css                // This files located at /public/themes/<current_
  ↵theme>/*.less(*.css)
  - theme.css

```

On this example (css only included) we can see structure after assets installation (collection). This files copied from modules Assets directory with directory tree structure, this work only for directories located in Assets and named as “css”, “img”, “js”. Other directories doesn’t affected by assets system, so if you will have directories like “sql” or “data” in Assets - they will not be copied to /public/assets directory.

Assets can be installed from console, using command: “php public/index.php assets install”. You can read about commands manager and console usage in this documentation. Also assets can be installed via “debug” switcher, when you enabling or disabling “debug” flag - system cleans cache and installs new assets.

Note: “css” directory can contains *.less files and *.css files. *.css files will be copied to public dir, but *.less will be compiled to *.css using constants.less from current theme. In that case constants.less can be used to archive main style of current theme (main colors, block sizes, etc).

Command

This directory contains commands classes, that can be used in console. You can read more about commands in commands manager section.

Controller

Controller directory contains all module controllers, request handlers. You can read how to use them at Phalcon documentation.

Form

Contains all forms classes, usually it can have such structure:

```
/app/modules/User/Form/
- Admin
|   - Create.php
|   - Edit.php
|   - RoleCreate.php
|   - RoleEdit.php
- Auth
    - Login.php
    - Register.php
```

How to create and use forms you can read in special section of this documentation.

Helper

Usually helpers oriented as View helpers, but you can use them everywhere in your code (where DI is available). Read about more helpers.

Model

Here is all module models (database entities). Read about models.

Plugin

Plugins used as event handlers. Each plugin can have attached event handlers to class. Read about plugins.

View

This directory contains views. There are directories with controller name inside which you can find views with extension (*.volt). Here some useful links about views:

- Views in PhalconEye - about internal system of views.
- [Views in Phalcon](#) - overall information about views in framework Phalcon.
- [Volt template engine](#) - about view template engine.

Widget

If you will look at Core module, you will find 3 widget there:

```
/app/modules/Core/Widget/
- Header
|   - Controller.php
|   - index.volt
- HtmlBlock
|   - Controller.php
|   - index.volt
- Menu
```

- Controller.php
- index.volt

Each directory inside “Widget” directory - is independent widget. Widget has it’s own controller and one or several views (depends on actions, that you need inside controller). Read more about widgets.

Installer

Installer is a script that allows to do some actions per installation/update or removal. Let’s look on example (Core installer):

```
<?php

use Engine\Installer as EngineInstaller;
use Phalcon\Acl as PhalconAcl;

class Installer extends EngineInstaller
{
    const
        /**
         * Current package version.
         */
        CURRENT_VERSION = '0.4.0';

    /**
     * Used to install specific database entities or other specific action.
     *
     * @return void
     */
    public function install()
    {
        $this->runSqlFile(__DIR__ . '/Assets/sql/installation.sql');
    }

    /**
     * Used before package will be removed from the system.
     *
     * @return void
     */
    public function remove()
    {

    }

    /**
     * Used to apply some updates.
     * Return 'string' (new version) if migration is not finished, 'null' if all
     ↪updates were applied.
     *
     * @param string $currentVersion Current module version.
     *
     * @return string/null
     */
    public function update($currentVersion)
    {
        return null;
    }
}
```

```
}
```

As you can see, installer has 3 mandatory methods: install, remove, update.

- **install** method executes after package was unpacked and moved to it's location (modules directory). Executes only after autoload setup.
- **remove** method executes before module removal from package manager.
- **update** method executes when user tries to install module of new version. In that case update will be executed several times until update process will reach current version.

For example, if current installed version is 1.0.0 and new package is 1.2.1. At 1.1.0 and 1.2.0 were database changes that you want to trigger correctly. Method "update" can look like this:

```
<?php
class Installer extends EngineInstaller
{
    const
        /**
         * Current package version.
         */
        CURRENT_VERSION = '1.2.1';

    public function update($currentVersion)
    {
        switch($currentVersion) {
            case "1.1.0"
                // Apply database changes from 1.0.0 to 1.1.0.
                ... CODE HERE...
                return "1.2.0";
            break;
            case "1.2.0"
                // Apply database changes from 1.2.0 to 1.2.1.
                ... CODE HERE...
                return CURRENT_VERSION;
            break;
        }

        return null;
    }
}
```

Bootstrap

Bootstrap initialize module systems and can adds some services to DI for other modules.

Note: Please, don't use huge initialization at bootstrap, coz if you will have more then 10 modules with huge initializations at bootstrap your system will be very slow!

```
<?php
namespace Core;

use Core\Model\Language;
use Core\Model\LanguageTranslation;
use Core\Model\Settings;
```

```

use Core\Model\Widget;
use Engine\Bootstrap as EngineBootstrap;
use Engine\Cache\System;
use Engine\Config;
use Engine\Translation\Db as TranslationDb;
use Phalcon\DI;
use Phalcon\DiInterface;
use Phalcon\Events\Manager;
use Phalcon\Mvc\View\Engine\Volt;
use Phalcon\Mvc\View;
use Phalcon\Translate\Adapter\NativeArray as TranslateArray;
use User\Model\User;

/**
 * Core Bootstrap.
 *
 * @category PhalconEye
 * @package Core
 * @author Ivan Vorontsov <ivan.vorontsov@phalconeye.com>
 * @copyright 2013-2014 PhalconEye Team
 * @license New BSD License
 * @link http://phalconeye.com/
 */
class Bootstrap extends EngineBootstrap
{
    /**
     * Current module name.
     *
     * @var string
     */
    protected $_moduleName = "Core";

    /**
     * Bootstrap construction.
     *
     * @param DiInterface $di Dependency injection.
     * @param Manager     $em Events manager object.
     */
    public function __construct($di, $em)
    {
        parent::__construct($di, $em);

        /**
         * Attach this bootstrap for all application initialization events.
         */
        $em->attach('init', $this);
    }

    /**
     * Init some subsystems after engine initialization.
     */
    public function afterEngine()
    {
        $di = $this->getDI();
        $config = $this->getConfig();

        $this->_initI18n($di, $config);
        if (!$config->installed) {
    
```

```
        return;
    }

    // Remove profiler for non-user.
    if (!User::getViewer()->id) {
        $di->remove('profiler');
    }

    // Init widgets system.
    $this->_initWidgets($di);

    /**
     * Listening to events in the dispatcher using the Acl.
     */
    if ($config->installed) {
        $this->getEventsManager()->attach('dispatch', $di->get('core')->acl());
    }

    // Install assets if required.
    if ($config->application->debug) {
        $di->get('assets')->installAssets(PUBLIC_PATH . '/themes/' ._
        Settings::getSetting('system_theme'));
    }
}

/**
 * Init locale.
 *
 * @param DI      $di      Dependency injection.
 * @param Config $config Dependency injection.
 *
 * @return void
 */
protected function _initI18n(DI $di, Config $config)
{
    $translate = ...
    // SOME CODE HERE

    $di->set('i18n', $translate);
}

/**
 * Prepare widgets metadata for Engine.
 *
 * @param DI $di Dependency injection.
 *
 * @return void
 */
protected function _initWidgets(DI $di)
{
    if ($di->get('app')->isConsole()) {
        return;
    }

    $widgets = ...
    // SOME CODE HERE

    $di->get('widgets')->addWidget($widgets);
}
```

```

    }
}
```

As you can see, bootstrap also can be attached to system events, to handle additional logic.

Plugins

Plugins is just like an event handlers. When you need some additional hooks - you can create plugins.

Plugins can be as part of module and as external package.

Let's look on example:

```
<?php

class DispatchErrorHandler
{
    /**
     * Before exception is happening.
     *
     * @param Event           $event      Event object.
     * @param Dispatcher      $dispatcher Dispatcher object.
     * @param PhalconException $exception Exception object.
     *
     * @throws \Phalcon\Exception
     * @return bool
     */
    public function beforeException($event, $dispatcher, $exception)
    {
        // Handle exceptions.
        // Some other code...
        return true;
    }
}
```

This example from core (EnginePluginDispatchErrorHandler) and attached to system as:

```
<?php

$eventsManager->attach("dispatch:beforeException", new DispatchErrorHandler());
```

That means, that you can attach your plugins manually.

Events Editor

If you want to automate you plugin (and make it editable), you can use Package Manager. It has events editor.

The screenshot shows the PhalconEye administrative interface. On the left is a sidebar with a dark theme containing the PhalconEye logo, a navigation menu with sections like 'Dashboard', 'MANAGE' (with 'Users and Roles', 'Pages', 'Menus', 'Languages', 'Files', and 'Packages'), 'SETTINGS' (with 'System', 'Performance', 'Access Rights'), and 'MODULES' (with 'Blog'). At the bottom of the sidebar are 'Back to site' and 'Logout' buttons. The main content area has a light background and a header 'Edit package events'. It contains instructions: 'In "Event" field write event name. Example: dispatch:beforeDispatchLoop' and 'In "Class" field write plugin class with namespace. Example: Plugin\SomePlugin\Some'. Below this are two input fields: 'Event: dispatch:before Class: Blog\Plugin\SomePluginDir\SomeName' and 'Event: init:afterEngine Class: Blog\Plugin\Core'. At the bottom are 'Save' and 'Cancel' buttons.

Events manager allowed for modules and plugins package types. In left field you enter event name, in right field - class with namespace that must handle this event. You can use both formats of event naming (dispatch or dispatch:eventName). Read more about events system at [Phalcon documentation](#).

Widgets

Widgets are main components of your content! Widget is component that has it's own controller and view. It can be placed on page and rendered by request.

```
./Widget/
- HtmlBlock
  - Controller.php
  - index.volt
```

Widgets can be part of some module or as external package. By default “index” action is used, but you can render widget with another action using widget wrapper - EngineWidgetElement->render(“update”).

Controller

Let's look on Controller example:

```
<?php

class Controller extends Engine\Widget\Controller
{
    /**
     * Index action.
     * This action is rendered by default.
     *
     * @return void
     */
    public function indexAction()
```

```

{
    // Get parameter provided by PhalconEye system.
    // This parameters handled by admin panel, when enduser can place his widget
    // and setup some params.
    $param = $this->getParam('count');

    // Get all params for this widget.
    $param = $this->getAllParams();

    // Engine\Widget\Controller is extended from Phalcon controller and it has DI
    // services.
    // So you can use anything from DI.
    // As for example - get param from HTTP request.
    $queryParam = $this->request->get("param");

    // Access to DI.
    $di = $this->getDI();

    // Set View params.
    $this->view->someParam = 1;

    // Set widget title, it's automatically takes from form params (added by
    // enduser), but you can override it.
    $this->view->title = "Some Title";

    // Set flag, that widget has no content and it's wrapper must not be rendered.
    return $this->setNoRender();
}

/**
 * This action is used, when user requests widgets options at page layout
 //management.
 *
 * @return CoreForm
 */
public function adminAction()
{
    return new YourWidgetFormClass();
}

/**
 * Cache this widget?
 * This method for wrapper, that will check, if you widget content must be cached.
 *
 * @return bool
 */
public function isCached()
{
    // If this method exists in widget controller
    // and returns "true" - rendered content of widget's view will be cached and
    // used at next time.
    return true;
}

/**
 * Get widget cache key.
 *
 * @return string|null
*/

```

```
 */
public function getCacheKey()
{
    // You can use this method to specify your widget cache.
    // By default system generates unique cache key automatically.
    // Note that this method will not be used if "isCached" method doesn't
    ↪ returns "true".
    return "some_you_unique_key";
}

/**
 * Get widget cache lifetime.
 *
 * @return int
 */
public function getCacheLifeTime()
{
    // Specify cache life time for your widget's cache.
    // 300 - is by default.
    return 300;
}
```

View

Let's look on index.volt example:

```
{% extends ".../Core/View/layouts/widget.volt" %}

{% block content %}
{{ html }}
{% endblock %}
```

Block “content” is main widget block. Here you can use usual volt template features.

Note: You can use Core layout ”.../Core/View/layouts/widget.volt” or create your own. But if you are using Core - specify full path to it (relative).

Admin Form

Widget params can be configured at widget options form at admin panel. Go to admin panel -> Pages -> Push on “Manage” link for some page -> find widget in layout (or place a new one) and push “Edit” link.

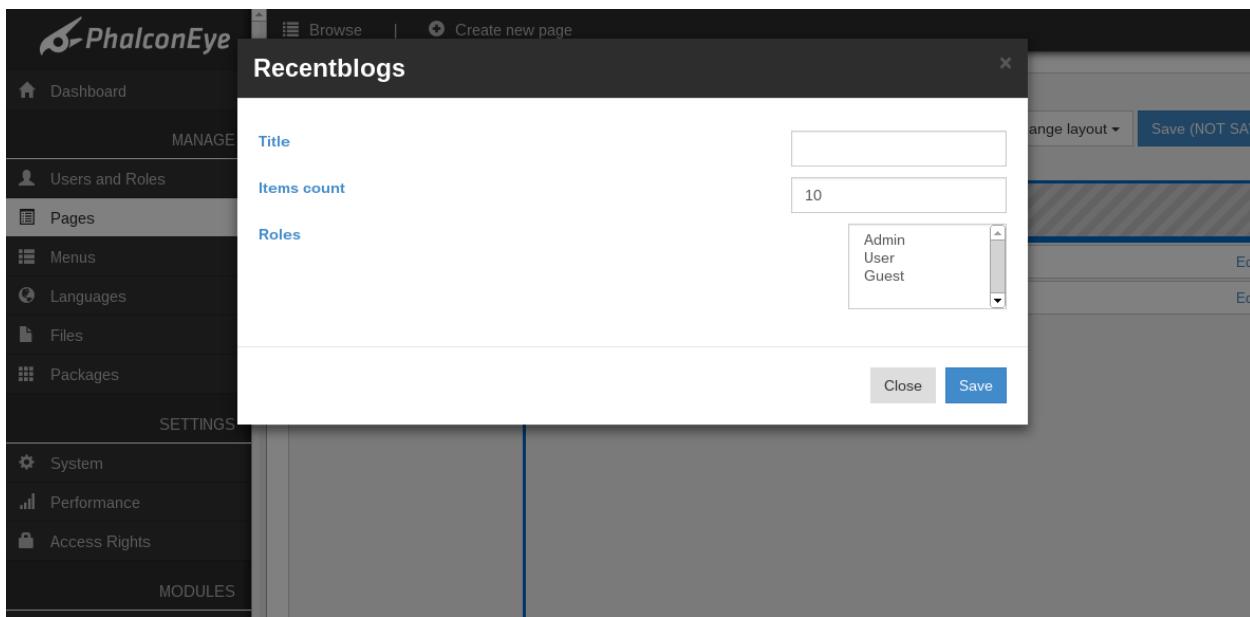
Note: Widget can have default params:

“title” - “Title” field at admin form.

“count” - “Is paginated” field at admin form.

“roles” - “Is ACL controller” field at admin form.

How default widget form looks (when all options allowed, title - is always present, by default):



Configure your own admin form you can in 2 ways:

- 1) At widget creation (or in database, field - "admin_form") set to "action". This will triggers "actionIndex" in widget controller. This action must return CoreForm instance, that will be rendered at admin panel (As example, look at HtmlBlock widget code).
- 2) At widget creation (or in database, field - "admin_form") set to "SomeNamespaceFormClass". This will create instance of "SomeNamespaceFormClass" when the user will request widget admin form.

Libraries

Libraries - your code (or vendor) for some specific tasks.

For example, you will need mail library. You can use existing mail libraries (and add your wrapper) or write your own. Let's say, you will use SwiftMailer (or other, doesn't matter). Library structure can be like this:

```
./libraries/
- Mail
  - vendor           // Here you can put your SwiftMailer.
  - Wrapper.php     // Wrapper written by you.
```

In that case, if you will create such library, your can access it via `Mail\Wrapper::factory()`. It means, that all classes inside subdirectories of libraries directory will be places in namespace of it's subdirectory name:

```
./libraries/
- Mail
|   - ..
|   - Wrapper.php    // Mail\Wrapper::factory()
- Some
  - New
    |   - Class      // Some\New\Class::factory()
  - Class.php        // Some\Class::factory()
```

Note: "factory" method is just for example, you can use any structure you want inside your library.

Themes

Theme package - archive with *.less, *.css and images. Respects folder structure.

Theme must include constants.less and theme.less files. Other *.less files you can separate them via LESS compiler logic. All *.less files inside themes compiles automatically. Constants files is required to handle style structure for modules, that can use this style manners for their html.

Constants from constants.less can be used in your modules and widgets to allow them use current style respecting theme changing by end user.

Models

Models is database entities. You can read more about them in [Phalcon documentation](#). But PhalconEye has little difference in models: annotations, changed methods (get, getFirst), etc.

Example of model:

```
<?php

namespace Core\Model;

use Engine\Db\AbstractModel;

/**
 * Access.
 *
 * @category PhalconEye
 * @package Core\Model
 * @author Ivan Vorontsov <ivan.vorontsov@phalconeye.com>
 * @copyright 2013-2014 PhalconEye Team
 * @license New BSD License
 * @link http://phalconeye.com/
 *
 * @Source("access")
 * @BelongsTo("role_id", "User\Model\Role", "id")
 */
class Access extends AbstractModel
{
    /**
     * @Primary
     * @Identity
     * @Column(type="string", nullable=false, column="object", size="55")
     */
    public $object;

    /**
     * @Primary
     * @Column(type="string", nullable=false, column="action", size="255")
     */
    public $action;

    /**
     * @Primary
     * @Column(type="integer", nullable=false, column="role_id", size="11")
     */
    public $role_id;
```

```
 /**
 * @Column(type="string", nullable=true, column="value", size="25")
 */
public $value;
}
```

Maybe you noticed that access to model fields possible through public modifier of this fields, but not through public methods. This was made coz of database fields naming conversion. In that case you can fast search through code and understand what field is responsible for. If you don't like this idea - you can change AbstractModel and add your own magic method (`__call`) or add methods to your model.

Annotations

Annotations are needed as metadata and for schema generation feature.

Name	Scope	Description	Parameters
Source	Class	This is table identity in database.	One parameter: table name.
BelongsTo	Class	Setup relation (n:1) to another model.	<ol style="list-style-type: none"> 1. Current model's field name. 2. Class name of model (with namespace). 3. Related model's field name. 4. Array of options.
HasMany	Class	Setup relation (1:n) to another model.	<ol style="list-style-type: none"> 1. Current model's field name. 2. Class name of model (with namespace). 3. Related model's field name. 4. Array of options.
Primary	Field	Set field as primary key.	—
Identity	Field	Set to field identity feature.	—
Column	Field	Specify column options.	<ul style="list-style-type: none"> • “type” - database type of this column (available: integer, string, text, boolean, date, datetime). • “nullable” (true/false) - field can be null or not. • “column” - column name in database. • “size” - size of column.
Acl	Class	This is not related to database... This allows to add some parameters to admin panel, that will be controlled for this model by enduser	<ul style="list-style-type: none"> • actions - Array named actions, that can be applied for this model. • options - Named options that can be associated with this model. <p>Example:</p> <pre><?php /** * @Acl(actions={ * "view", "edit", * "delete"},</pre>
60			Chapter 2, Table of Contents <pre>options={"count", "titlePrepend"}} */ class Blog extends AbstractModel</pre>

Methods

Here is some useful methods:

```
<?php

// Get table name in database (for queries and other stuff).
$tableName = Access::getTableName();

// Methods "get" and "getFirst" was modified with sprintf method inside.
// Parameters:
// 1. Condition with placeholders
// 2. Parameter for condition placeholders.
// 3. Order by field.
// 4. Limit.
$access1 = Access::get('action = "%s" AND value = "%s"', ["edit", "allowed"], "role_id",
    100)
$access2 = Access::getFirst('module = "%s" AND name = "%s"', ["edit", "allowed"],
    "role_id", 100)

// Create builder for access table, returns object of Phalcon\Mvc\Model\Query\Builder
// class. First param - table alias in query builder.
$builder = Access::getBuilder("tableAlias");

// Get row identity.
$id = $access1->getId();
```

Views

- Views in Phalcon - overall information about views in framework Phalcon.
- Volt template engine - about view template engine.

Volt engine is used as main rendering processor. Views can be found in modules and widgets.

Widget views

Widget can have one or several views, depends on it's controller actions. By default: index.volt. This view will be used for indexAction of widget controller.

Module views

Module views located under directory “View”. Dispatcher resolves view according to controller and it’s action name. For example, if you have “SomeController” and “newAction” method view for this action must be placed under /View/Some/new.volt (sensitive register).

Core module has default layouts for admin and main page layout, to use them, you must add “extends” modifier:

Note: Path to layout view must be relative.

```
{% extends "../Core/View/layouts/main.volt" %} {# Main layout, used for all
// frontend views. #}

{% extends "../Core/View/layouts/admin.volt" %} {# Admin layout, used in admin
// views. #}
```

```
{% extends ".../Core/View/layouts/widget.volt" %} {# Widget layout, used in widgets  
views. #}
```

Helpers

You can read more about them in other part about helpers.

You can use helpers inside your views, this allows to move server logic to php, outside from view part.

```
{} helper('setting', 'core') .get('system_title', '') {} {# Get setting 'system_title',  
with default value ''. #}
```

First parameter is helper class name (in that case, this will be Setting.php). Second parameter is namespace of this helper, by default this is ‘engine’, in that example - ‘core’. It means, that this class is accessible at CoreHelperSetting. After that call helper system returns your an object of CoreHelperSetting, this object created only once and by other calls it taken from cache (by singleton logic). Cache in that case is DI, so you also can check if helper is loaded by accessing it in DI:

```
<?php  
  
$di->has('Core\Helper\Setting');
```

Extension

View has some additional methods and filters.

```
{# call php code: #}  
{{ php('phpinfo()') }}  
{{ helper('formatter') .formatNumber(100, php('\NumberFormatter::DECIMAL')) }}
```



```
{# classof (get_class in php): #}  
{% if classof(element) is 'FieldSet' %}  
...  
{% endif %}
```



```
{# instanceof in php: #}  
{% if instanceof(element, 'Engine\Form\FieldSet') %}  
...  
{% endif %}
```



```
{# resolveView, allows to find relative path to view. First parameter is view name,  
second - module name: #}  
{{ partial(resolveView('partials/paginator', 'core')) }}
```



```
{# i18n filter, for translations: #}  
{{ "Some text" | i18n }}
```

Forms

Forms allows to handle user data. Forms contains such features:

- Fieldsets.

- Conditions.
- Elements and containers based (OOP objects).
- Fast methods for elements.
- Views as partial (separated logic for elements, containers, etc).
- Multiply entities support.
- Validation according to form validation and entities validation.
- Simple entity form.
- Text form.
- File form.

Form example:

```
<?php

namespace Core\Form\Admin\Setting;

use Core\Form\CoreForm;

/**
 * Performance settings.
 *
 * @category PhalconEye
 * @package Core\Form\Admin\Setting
 * @author Ivan Vorontsov <ivan.vorontsov@phalconeye.com>
 * @copyright 2013-2014 PhalconEye Team
 * @license New BSD License
 * @link http://phalconeye.com/
 */
class Performance extends CoreForm
{
    /**
     * Initialize form.
     *
     * @return void
     */
    public function initialize()
    {
        $this->setTitle('Performance settings');

        $this->addContentFieldSet(
            ->addText('prefix', 'Cache prefix', 'Example: "pe_"', 'pe_')
            ->addText(
                'lifetime',
                'Cache lifetime',
                'This determines how long the system will keep cached data before
                 reloading it from the database server.
                 A shorter cache lifetime causes greater database server CPU usage,
                 however the data will be more current.',
                86400
            )
            ->addSelect(
                'adapter',
                'Cache adapter',
                'Cache type. Where cache will be stored.'
            )
        );
    }
}
```

```
[  
    0 => 'File',  
    1 => 'Memcached',  
    2 => 'APC',  
    3 => 'Mongo'  
],  
0  
)  
  
/**  
 * File options  
 */  
->addText('cacheDir', 'Files location', null, ROOT_PATH . '/app/var/cache/  
↳data/')  
  
/**  
 * Memcached options.  
 */  
->addText('host', 'Memcached host', null, '127.0.0.1')  
->addText('port', 'Memcached port', null, '11211')  
->addCheckbox('persistent', 'Create a persistent connection to memcached?  
↳', null, 1, true, 0)  
  
/**  
 * Mongo options.  
 */  
->addText(  
    'server',  
    'A MongoDB connection string',  
    null,  
    'mongodb://[username:password@]host1[:port1][,host2[:port2],...[,  
↳hostN[:portN]]]'  
)  
->addText('db', 'Mongo database name', null, 'database')  
->addText('collection', 'Mongo collection in the database', null,  
↳'collection')  
  
/**  
 * Other.  
 */  
->addCheckbox('clear_cache', 'Clear cache', 'All system cache will be  
↳cleaned.', 1, false, 0);  
  
$this->addFooterFieldSet()->addButton('save');  
  
$this->addFilter('lifetime', self::FILTER_INT);  
$this->_setConditions();  
}  
  
/**  
 * Validates the form.  
 *  
 * @param array $data Data to validate.  
 * @param bool $skipEntityCreation Skip entity creation.  
 *  
 * @return boolean  
 */  
public function isValid($data = null, $skipEntityCreation = false)
```

```

{
    if (!\$data) {
        \$data = \$this->getDI()->getRequest()->getPost();
    }

    if (isset(\$data['adapter']) && \$data['adapter'] == '0') {
        if (empty(\$data['cacheDir']) || !is_dir(\$data['cacheDir'])) {
            \$this->addError('Files location isn\'t correct!');
        }

        return false;
    }
}

return parent::isValid(\$data, \$skipEntityCreation);
}

/**
 * Set form conditions.
 *
 * @return void
 */
protected function _setConditions()
{
    \$content = \$this->getFieldSet(self::FIELDSET_CONTENT);

    /**
     * Files conditions.
     */
    \$content->setCondition('cacheDir', 'adapter', 0);

    /**
     * Memcached conditions.
     */
    \$content->setCondition('host', 'adapter', 1);
    \$content->setCondition('port', 'adapter', 1);
    \$content->setCondition('persistent', 'adapter', 1);

    /**
     * Mongo conditions.
     */
    \$content->setCondition('server', 'adapter', 3);
    \$content->setCondition('db', 'adapter', 3);
    \$content->setCondition('collection', 'adapter', 3);
}
}

```

Structure

Root form class is abstract. So you can't create it directly. Also it has abstract methods, that identifies form rendering feature. That's why there is some simple form structure:

```

AbstractForm
|
CoreForm      EntityForm (trait)
|
|----- FileForm

```

```
|  
|----- TextForm
```

Core form implements all necessary methods:

```
<?php  
  
namespace Core\Form;  
  
use Engine\Form\AbstractForm;  
  
/**  
 * Main core form.  
 *  
 * @category PhalconEye  
 * @package Core\Form  
 * @author Ivan Vorontsov <ivan.vorontsov@phalconeye.com>  
 * @copyright 2013-2014 PhalconEye Team  
 * @license New BSD License  
 * @link http://phalconeye.com/  
 */  
class CoreForm extends AbstractForm  
{  
    const  
        /**  
         * Default layout path.  
         */  
        LAYOUT_DEFAULT_PATH = 'partials/form/default';  
  
    use EntityForm;  
  
    /**  
     * Get layout view path.  
     *  
     * @return string  
     */  
    public function getLayoutView()  
    {  
        return $this->_resolveView(self::LAYOUT_DEFAULT_PATH);  
    }  
  
    /**  
     * Get element view path.  
     *  
     * @return string  
     */  
    public function getElementView()  
    {  
        return $this->getLayoutView() . '/element';  
    }  
  
    /**  
     * Get errors view path.  
     *  
     * @return string  
     */  
    public function getErrorsView()  
    {
```

```

        return $this->getLayoutView() . '/errors';
    }

/**
 * Get notices view path.
 *
 * @return string
 */
public function getNoticesView()
{
    return $this->getLayoutView() . '/notices';
}

/**
 * Get fieldset view path.
 *
 * @return string
 */
public function getFieldSetView()
{
    return $this->getLayoutView() . '/fieldSet';
}

/**
 * Resolve view.
 *
 * @param string $view View path.
 * @param string $module Module name (capitalized).
 *
 * @return string
 */
protected function _resolveView($view, $module = 'Core')
{
    return '.../' . $module . '/View/' . $view;
}
}

```

Text and file form extended from it and used for text rendering and file uploading features respectively. Entity trait used for forms that must be created according to some entity. Read more about each form type below.

Elements

Elements are objects and form/fieldset is a container for these objects. So you can add element to form by creating it and adding:

```

<?php

// Create element.
$el = new Text("someName", [/*options*/], [/*attributes*/]);

// Add element with order 1001.
$this->add($el, 1001);

```

But this is a bit hard. So, there are exists some methods for element creation:

- addHtml
- addButton

- addButtonLink
- addText
- addTextArea
- addCkEditor
- addPassword
- addHidden
- addHeading
- addFile
- addRemoteFile
- addCheckbox
- addRadio
- addMultiCheckbox
- addSelect
- addMultiSelect

Default options of elements (not all allowed, and this is not a complete list, options can be added manually by element):

Name	Description
label	Label content for element
description	Description text for element
required	Mark element as required (you can't submit form without data for this element).
emptyNotAllowed	Mark element as required with non empty value (you can't submit form with empty string for this element).
ignore	Ignore element in validation and values, ignores it at backend, it will be skipped. Example: buttons.
htmlTemplate	Html template for element.
defaultValue	Default value of element. Example: checkbox, user set (un)checked state, but default value is '1'.

Non-default options:

El- e- mentName Name	Op- tion Name	Type	Description
But- ton	is- Sub- mit	Bool	Flag, that adds sub- mit fea- ture to the but- ton. If it 'false' - but- ton will not be able to sub- mit the form.
Check- box	check	Mixed	df some- thing is set to this op- tion (true, 'checked', etc) an ad- di- tional at- tribute checked="checked" will be added. If it is null,

List of all elements, their options and attributes:

Name	Description	Allowed Options	Default Options	Default Attributes
Button	Button element.	‘htmlTemplate’, ‘label’, ‘isSubmit’	‘isSubmit’ => true	<ul style="list-style-type: none"> • ‘id’ => \$this->getName() • ‘name’ => \$this->getName() • ‘required’ => true/false If ‘isSubmit’ == true <ul style="list-style-type: none"> • ‘type’ => ‘submit’ • ‘class’ => ‘btn btn-primary’ else <ul style="list-style-type: none"> • ‘class’ => ‘btn’
ButtonLink	Button as link, e.g.: back link.	‘htmlTemplate’, ‘label’	—	<ul style="list-style-type: none"> • ‘id’ => \$this->getName() • ‘name’ => \$this->getName() • ‘required’ => true/false • ‘class’ => ‘btn form_link_button’
Checkbox	Html input of type “checkbox”.	all default options, ‘checked’	—	<ul style="list-style-type: none"> • ‘id’ => \$this->getName() • ‘name’ => \$this->getName() • ‘required’ => true/false • ‘class’ => “ • ‘type’ => ‘checkbox’
CkEditor	CkEditor control.	all default options, ‘elementOptions’	—	<ul style="list-style-type: none"> • ‘id’ => \$this->getName() • ‘name’ => \$this->getName() • ‘required’ => true/false • ‘class’ => ‘form-control’ • ‘data-widget’ => ‘ckeditor’ • ‘data-name’ => \$this->getName()
2.4. Developer’s guide				<ul style="list-style-type: none"> • ‘data-options’ => ‘elementOptions’

Note: in most cases, when ‘htmlTemplate’ option is allowed element renders via it.

Fieldsets

Fieldset is a logical and/or visible separation. By default there are two fieldsets: content and footer. Content is for editable elements and footer is for buttons:

```
<?php

class Create extends CoreForm
{
    /**
     * Initialize form.
     *
     * @return void
     */
    public function initialize()
    {
        // Add elements to default content field set (field set key is 'form_content').
        $this->addContentFieldSet()
            ->addText('name', 'Name', 'Name must be in lowercase and contains only letters.')
            ->addSelect('type', 'Package type', null, Manager::$allowedTypes)
            ->addText('title');

        // Add buttons to footer (field set key is 'form_footer').
        $this->addFooterFieldSet()
            ->addButton('create')
            ->addButtonLink('cancel', 'Cancel', ['for' => 'admin-packages']);
    }
}
```

You can add your fieldsets or access them:

```
<?php

// Get content field set.
$contentFieldSet = $this->getFieldSet(self::FIELDSET_CONTENT); // self::FIELDSET_CONTENT = 'form_content'

// Add new field set.
$fieldSet = new FieldSet('fieldName', 'Some legend, if needed', ['class' => 'css-class'], /*... array of elements...*/);

// Add elements.
// Elements adding methods are the same as for form class.
$fieldSet->addElement(...);

// Set flag for rendering feature, this will remove html div separation between elements, by default used for buttons at footer.
$fieldSet->combineElements(true);

// Adds css attribute to all elements inside fieldset with key: id="fieldName_elementName".
$fieldSet->enableNamedElements(true);
```

```
// Changes all elements css name attribute according to fieldset name: name=
// "fieldSetName[elementName]".
$fieldSet->enableDataElements(true);

// Addd fieldset to form with order number 1001.
$this->addFieldSet($fieldSet, 1001);
```

Conditions

Conditions allows to set relation between fields.

For example we have 3 fields: select, text and text. Select and text must be visible always, but third text field must be visible only when select field has some specific value. Conditions allows you to setup such relation:

```
<?php

// Parameters:
// 1) Field that will be checked on value change. Our "select".
// 2) Our "third text field" that will be shown/hidden.
// 3) Value that must be in select to satisfy this condition and show "third text_
// field".
// 4) Comparison operator, you can find constants in Engine\Form\ConditionResolver._  

// Allowed: ==, !=, >, <, >=, <=.
// This operator defines how value of fieldA must be compared to value that you_
// entered in third parameter.
// 5) Summary operator. That operators also defined in Engine\Form\ConditionResolver._  

// Allowed: 'and', 'or'.
// In case when "third text field" also related to "second text field" you can add_
// new condition on that field,
// And in that case you will have two conditions, that's why you need to setup_
// result operator - logical AND or OR.
// "third text field" will be shown/hidden state depends on result of conditions_
// and their summary result.
$content->setCondition('fieldA', 'fieldB', 1, '==', 'and');

// Examples.
// Preconditions:
// select with values (1,2,3) - field1.
// text field - field2.
// text field - field3.

// Condition: field3 visible only when field1 has value '2' and field2 has value_
// greater than '15'.
$content->setCondition('field1', 'field3', 2); // Comparison by default is '==' and_
// result operator is 'and'.
$content->setCondition('field2', 'field3', 15, ConditionResolver::COND_CMP_GREATER);

// Condition: field3 visible when field1 has value '3' or field2 has lower or_
// equivalent to '0'.
$content->setCondition('field1', 'field3', 3, ConditionResolver::COND_CMP_EQUAL,_  

// ConditionResolver::COND_OP_OR);
$content->setCondition('field2', 'field3', 0, ConditionResolver::COND_CMP_LESS_OR_
// EQUAL, ConditionResolver::COND_OP_OR);

// Condition: fieldSet 'footer' visible only when field1 has value '3'.
$this->setFieldSetCondition(self::FIELDSET_FOOTER, 'field1', 3);
```

This conditions allows to show/hide fields (all logic based on js, already implemented). Also it's enables/disables validation for this fields and of course getValues method will return data without fields values if condition wasn't successful.

Form view

AbstractForm class has some abstract methods:

- getLayoutView - path to form layout view.
- getElementView - element view.
- getErrorsView - errors view.
- getNoticesView - notices view.
- getFieldSetView - view for fieldset.

This methods can be overridden, you can change one part of form view to your own. It means that you can simply change form style without problems to other forms.

Layout view example:

```
 {{ form.openTag() }}
<div>
    {%
        if form.getTitle() or form.getDescription() %
            <div class="form_header">
                <h3>{{ form.getTitle() }}</h3>

                <p>{{ form.getDescription() }}</p>
            </div>
    %endif %
    {{ partial(form.getErrorsView(), ['form': form]) }}
    {{ partial(form.getNoticesView(), ['form': form]) }}

    <div class="form_elements">
        {%
            for element in form.getAll() %
                {{ partial(form.getElementView(), ['element': element]) }}
        %endfor %
    </div>
    <div class="clear"></div>

    {%
        if form.useToken() %
            <input type="hidden" name="{{ security.getTokenKey() }}" value="{{ security.
            getToken() }}"/>
    %endif %
    </div>
    {{ form.closeTag() }}

```

Entities support

For example you have blog and you want to create form that will create blogs. You can associate form with entity and after validation you will have a new blog model.

To associate form with entity you must add it per initialization. In most cases form for creation can be extended for form that will edit this blogs. So this must be respected:

```
<?php

public function __construct(AbstractModel $entity = null)
{
    parent::__construct();

    if (!$entity) {
        $entity = new Blog();
    }

    $this->addEntity($entity);
}
```

Done! To get your complete blog entity, just get it after validation.

```
<?php

$this->view->form = $form = new CreateForm();
if (!$this->request->isPost() || !$form->isValid()) {
    return;
}

$blog = $form->getEntity();
```

Note that blog already saved to database. If you don't want to save it automatically, run validation with skip flag:

```
<?php

$this->view->form = $form = new CreateForm();
if (!$this->request->isPost() || !$form->isValid(null, true)) {
    return;
}

$blog = $form->getEntity(); // This entity isn't saved yet.
$blog->generateSlug();
$blog->save();
```

You can add multiple entities:

```
<?php

public function __construct(AbstractModel $entity1 = null, AbstractModel $entity2 = null)
{
    parent::__construct();

    if (!$entity1) {
        $entity1 = new Blog();
    }

    if (!$entity2) {
        $entity2 = new Tag();
    }

    $this->addEntity($entity1, 'blog');
    $this->addEntity($entity2, 'tag');
}
```

```
// In controller:  
  
$blog = $this->getEntity('blog');  
$tag = $this->getEntity('tag');
```

Validation

Validation is divided. Validation can be defined for form, fieldset, entity. But all this validation is checked independently. If you like [entity validation](#) you can use it. For form validation internal validation system can be used.

Example:

```
<?php  
  
$formOrFieldSet->getValidation()  
    ->add('language', new StringLength(['min' => 2, 'max' => 2]))  
    ->add('locale', new StringLength(['min' => 5, 'max' => 5]));  
    ->add('email', new Email())  
    ->add(  
        'controller',  
        new Regex(  
            [  
                'pattern' => '/$|(.*)Controller->(.*)Action/',  
                'message' => 'Wrong controller name. Example: NameController->  
                ↪someAction'  
            ]  
        )  
    );
```

Filter

Filter allows to filter entered values. There are some available filters (constants in AbstractForm class):

- FILTER_STRING
- FILTER_EMAIL
- FILTER_INT
- FILTER_FLOAT
- FILTER_ALPHANUM
- FILTER_STRIPTAGS
- FILTER_TRIM
- FILTER_LOWER
- FILTER_UPPER

About filter system read in [phalcon documentation](#).

```
<?php  
  
$form->addFilter('lifetime', AbstractForm::FILTER_INT);
```

Text Form

This form is same as CoreForm, but it has changed views. In normal form all elements renders as control, in text form all element doesn't renders, form takes only their values.

CoreForm element view:

```
<div class="form_element">
    {%
        if instanceof(element, 'Engine\Form\Element\File') and element.getOption(
            'isImage') and element.getValue() != '/'
    %}
        <div class="form_element_file_image">
            
        </div>
    {%
        endif %
    }
    {{ element.render() }}
</div>
```

TextForm element view:

```
<div class="form_element">
    {%
        if instanceof(element, 'Engine\Form\Element\File') and element.getOption(
            'isImage') %
    %}
        <div class="form_element_file_image">
            
        </div>
    {%
        endif %
    }
    {{ element.getValue() }}
</div>
```

File Form

File form extended from CoreForm and contains additional checks for files validation, image transformation, files management, etc. FileForm is marked as ‘multipart/form-data’ and has additional methods.

How to use it:

```
<?php

$form = new FileForm();

if (!$this->request->isPost() || !$form->isValid()) {
    return;
}

// Get all files from request.
$files = $form->getFiles();

// Get file of specific field.
$file = $form->getFiles('name');
```

Set file validation:

```
<?php

$form->getValidation()->add('file', new MimeType(['type' => 'application/json']));
```

Set image transformations on upload (performed after validation, if valid):

```
<?php

$form->setImageTransformation(
    'icon',
    [
        'adapter' => 'GD',
        'resize' => [32, 32]
    ]
);
```

‘adapter’ parameter is name of adapter that will be used (GD or Imagick). Other parameters are methods that will be called from adapter and value is parameters for this method (\$gd->resize(32,32));.

Entity Form (Trait)

Entity trait was designed as light and simple way of form creation according to model. It applied to CoreForm as trait and can be accessible through different form types, for example text:

```
<?php

$user = User::findFirst($id);
$this->view->form = $form = TextForm::factory($user, [], [['password']]);

$form
    ->setTitle('User details')
    ->addFooterFieldSet()
    ->addButtonLink('back', 'Back', ['for' => 'admin-users']);
```

EntityForm trait has one method “factory”:

```
<?php

/**
 * Create form according to entity specifications.
 *
 * @param AbstractModel[] $entities      Models.
 * @param array           $fieldTypes   Field types.
 * @param array           $excludeFields Exclude fields from form.
 *
 * @return AbstractForm
 */
public static function factory($entities, array $fieldTypes = [], array
    $excludeFields = []) {}
```

Field types parameter allows to change some fields html control (by default <input type=”text”/>). Exclude parameter allows to filter unnecessary fields.

Grid System

Grid is table of entities with actions, sorting and filtering.

Pages (3)

ID	Title	Url	Layout	Controller	Actions
					<button>Filter</button> <button>Reset</button>
1	Header				Manage
2	Footer				Manage
3	Home	/			Manage Edit

Same as in forms here is AbstractGrid and extended CoreGrid (abstract, too):

```
<?php

abstract class CoreGrid extends AbstractGrid
{
    /**
     * Get grid view name.
     *
     * @return string
     */
    public function getLayoutView()
    {
        return $this->_resolveView('partials/grid/layout');
    }

    /**
     * Get grid item view name.
     *
     * @return string
     */
    public function getItemView()
    {
        return $this->_resolveView('partials/grid/item');
    }

    /**
     * Get grid table body view name.
     *
     * @return string
     */
    public function getTableBodyView()
    {
        return $this->_resolveView('partials/grid/body');
    }

    /**
     * Resolve view.
     *
     * @param string $view View path.
     * @param string $module Module name (capitalized).
     *
     * @return string
     */
    protected function _resolveView($view, $module = null)
```

```
/*
protected function _resolveView($view, $module = 'Core')
{
    return '.../' . $module . '/View/' . $view;
}
}
```

Usage in controller:

```
<?php

public function indexAction()
{
    $grid = new UserGrid($this->view);
    if ($response = $grid->getResponse()) {
        return $response;
    }
}

// yep.. that's all ).
```

Source

Grid has source. Source can be QueryBuilder or Array. You can implement your own SourceResolver to handle different data.

Usual QueryBuilder:

```
<?php

// Method getSource is required.
public function getSource()
{
    $builder = new Builder();
    $builder
        ->columns(['u.*', 'r.name'])
        ->addFrom('User\Model\User', 'u')
        ->leftJoin('User\Model\Role', 'u.role_id = r.id', 'r')
        ->orderBy('u.id DESC');

    return $builder;
}
```

Array usage:

```
<?php

public function getSourceResolver()
{
    return new ArrayResolver($this);
}

public function getSource()
{
    $data = [['row1_column1' => 1, 'row1_column2' => 2], ['row2_column1' => 3, 'row2_
    ↪column2' => 4]];
    return $data;
}
```

```
}
```

Columns

Columns must be defined per required method `_initColumns()`. Columns definition contains:

- id (name of field in query builder or in array).
- label - Column label.
- sortable - flag that defines if column is sortable.
- type - column bind type parameter (see `PhalconDbColumn::BIND_*`).
- filter - flag that defines if this column can be filtered.
- use_having - flag that allows to build query using HAVING operator (in case query contains JOINS and joined has conditions).
- condition_like - flag that allows to use LIKE operator in condition.
- output_logic - this allows to change output behaviour, accepts function closure.

Example:

```
<?php

protected function _initColumns()
{
    $this
        // Add simple text column, this means, that in filtering will be available
        // text field.
        ->addTextColumn(
            'u.id',      // field name in query
            'ID',        // Label
            [
                self::COLUMN_PARAM_TYPE => Column::BIND_PARAM_INT,           // Bind
                // parameter, need to escape SQL injections.
                self::COLUMN_PARAM_OUTPUT_LOGIC =>                         // Special
                // output logic.
                function (GridItem $item, $di) {
                    $url = $di->get('url')->get(
                        ['for' => 'admin-users-view', 'id' => $item['u.id']]
                    );
                    return sprintf('<a href="%s">%s</a>', $url, $item['u.id']);
                }
            ]
        )
        ->addTextColumn('u.username', 'Username')
        ->addTextColumn('u.email', 'Email')
        ->addSelectColumn(
            'r.name',
            'Role',
            ['hasEmptyValue' => true, 'using' => ['name', 'name'], 'elementOptions' =>
            // Role::find()],
            [
                self::COLUMN_PARAM_USE_HAVING => false,                      // Don't use
                // HAVING
                self::COLUMN_PARAM_USE_LIKE => false,                         // And don't
                // use LIKE, '==' operator will be used ('=' IN SQL).
            ]
        )
}
```

```

        self::COLUMN_PARAM_OUTPUT_LOGIC =>
            function (GridItem $item) {
                return $item['name'];
            }
        ]
    )
->addTextColumn('u.creation_date', 'Creation Date');
}

```

Actions

Actions also can be defined:

```

<?php

public function getItemActions(GridItem $item)
{
    $actions = [
        'Manage' => ['href' => ['for' => 'admin-languages-manage', 'id' => $item['id']],
        'Export' => [
            'href' => ['for' => 'admin-languages-export', 'id' => $item['id']],
            'attr' => ['data-widget' => 'modal']
        ],
        'Wizard' => [
            'href' => ['for' => 'admin-languages-wizard', 'id' => $item['id']],
            'attr' => ['data-widget' => 'modal']
        ],
        '|' => [],
        'Edit' => ['href' => ['for' => 'admin-languages-edit', 'id' => $item['id']]],
        'Delete' => [
            'href' => [
                'for' => 'admin-languages-delete', 'id' => $item['id']
            ],
            'attr' => ['class' => 'grid-action-delete']
        ]
    ];

    if (
        $item->getObject()->language == Config::CONFIG_DEFAULT_LANGUAGE &&
        $item->getObject()->locale == Config::CONFIG_DEFAULT_LOCALE
    ) {
        unset($actions['|']);
        unset($actions['Edit']);
        unset($actions['Wizard']);
        unset($actions['Delete']);
    }

    return $actions;
}

```

getItemActions(GridItem \$item) must return array of actions with parameters. ‘href’ is required parameter, ‘attr’ is optional.

Grid View

Grid view divided on three parts: layout (main layout, starting from `<table>` tag), body (`<tbody>` tag), item (`<td>` tag with actions). Each view can be overridden in grid class.

Layout example:

```





```

Helpers

Mainly helpers existing for view extension. When in view must be performed some huge logic - this part of work can be moved to helper.

Helpers can be accessible in view in such way:

```
 {{ helper('setting', 'core') .get('system_title', '') }} # Get setting 'system_title',
 ↵ with default value ''. #}
```

First parameter is helper class name (in that case, this will be Setting.php. Second parameter is namespace of this helper, by default this is ‘engine’, in that example - ‘core’. It means, that this class is accessible at CoreHelperSetting. After that call helper system returns your an object of CoreHelperSetting, this object created only once and by other calls it taken from cache (by singleton logic). Cache in that case is DI, so you also can check if helper is loaded by accessing it in DI:

```
<?php
$di->has('Core\Helper\Setting');
```

Helper class can be used in other places:

```
<?php
// Using static method.
$settingsHelper = \Engine\Helper::getInstance('setting', 'core');
$systemTitle = $settingsHelper->get('system_title', '');

// Or directly from required class.
$settingsHelper = \Core\Helper\Setting::getInstance($this->getDI());
$systemTitle = $settingsHelper->get('system_title', ''');
```

Helper creation

To create helper you need extend it from Engine\Helper and write your methods:

```
<?php
class NewHelper extends \Engine\Helper
{
    public function someMethod1() {}

    public function someMethod2() {}
}
```

Existing helpers

Here is list of available helpers:

Name	Namespace	Methods
Assets	Engine\Helper\Assets	<ul style="list-style-type: none"> • addJs(\$file, \$collection = 'js') - Adds js file to assets collection. • addCss(\$file, \$collection = 'css') - Adds css file to assets collection.
Formatter	Engine\Helper\Formatter	<ul style="list-style-type: none"> • formatNumber(\$number, \$style = \NumberFormatter::DECIMAL) - Format number according to current locale. • formatCurrency(\$number) - Format currency as \NumberFormatter::CURRENCY (according to current locale).
Url	Engine\Helper\Url	<ul style="list-style-type: none"> • currentUrl() - Get current URL from request. • paginatorUrl(\$pageNumber = null) - Generate url for paginator.
I18n	Core\Helper\I18n	<ul style="list-style-type: none"> • add(\$translations, \$params = []) - Add translation to temporary storage, this is for js translations. • js(\$translations, \$params = []) - Render js translations, \$translations overrides current temporary storage. • clear() - clear temporary storage. • render(\$translations = null) - Render translations, without params concatenation. <p>All data after rendering placed in js variable "translatorData". Usage in php:</p> <pre><?php I18n::getInstance(\$this-> getDI()) ->add('Are you really want to delete this item?') ->add('Hello %item%', ['item' => 'World']) ->add('Close this window?');</pre> <p>Usage in volt:</p> <pre>{{ helper('i18n', 'core') }} ->add('Hello %item%', ['item' => 'World']) }} {{ helper('i18n', 'core') }} ->render() }}</pre>
2.4. Developer's guide		85

Navigation

Navigation allows to build menus and breadcrumbs.

Let's look on example:

```
<?php

$navigation = new Navigation();
$navigation
    ->setItems(
        [
            'index' => [
                'href' => 'admin/menus',
                'title' => 'Browse',
                'prepend' => '<i class="glyphicon glyphicon-list"></i>'
            ],
            1 => [
                'href' => 'javascript:',
                'title' => '|'
            ],
            'create' => [
                'href' => 'admin/menus/create',
                'title' => 'Create new menu',
                'prepend' => '<i class="glyphicon glyphicon-plus-sign"></i>'
            ]
        ]
    );
$this->view->navigation = $navigation;
```

setItems method defines items inside navigation. It accepts array of arrays. Index of each array can be used as active item setup.

Let's look on description of all attributes:

```
<?php

$navigation
    ->setItems(
        [
            'index' => [
                'href' => 'admin/menus',
                // Item link.
                'title' => 'Browse',
                // Item title.
                'target' => '_blank',
                // Link "target" attribute.
                'onclick' => 'alert("");',
                // Link "onclick" attribute.
                'tooltip' => 'Browse Description',
                // Item tooltip.
                'tooltip_position' => 'right',
                // Item tooltip position.
                'append' => '<i class="glyphicon glyphicon-list"></i>',
                // HTML/text that will be appended after title.
                'prepend' => '<i class="glyphicon glyphicon-list"></i>',
                // HTML/text that will be prepended before title.
            ],
        ]
    );
$this->view->navigation = $navigation;
```

```

    1 => [ // just an item,
        'href' => 'javascript:;',
    ↵ // Nothing special.
        'title' => '|'
    ↵ // Title again.
        ],
        'settings' => [ // type - dropdown
            'title' => 'Settings',
            'items' => [
    ↵ // Dropdown can be defined by present "items" key in item array.
            'admin/settings' => [
                'title' => 'System',
                'href' => 'admin/settings',
                'prepend' => '<i class="glyphicon glyphicon-cog"></i>'
            ],
            'admin/settings/performance' => [
                'title' => 'Performance',
                'href' => 'admin/performance',
                'prepend' => '<i class="glyphicon glyphicon-signal"></i>'
            ],
            2 => 'divider',
    ↵ // Styled divider.
            'admin/access' => [
                'title' => 'Access Rights',
                'href' => 'admin/access',
                'prepend' => '<i class="glyphicon glyphicon-lock"></i>'
            ]
        ]
    ],
    ]
);

```

To set some item as active use setActiveItem method, it checks items keys and their ‘href’ attribute if they are equal - navigation marks it as active.

Navigation Styling

For navigation customization there are some methods:

```

<?php

$navigation = new Navigation();

// Set overall list style class (By default applied to <ul> tag).
$navigation->setListClass('nav nav-categories');

// Set dropdown item class (Item of parent but with subitems, by default applied to
// <li> tag).
$navigation->setDropDownItemClass('dropdown');

// Set class list inside item (By default applied to <ul> tag).
$navigation->setDropDownItemMenuClass('dropdown-menu');

// Drop down item icon, <li>.
$navigation->setDropDownIcon('<b class="caret"></b>');

// By default: true. If active - parent of active item will be highlighted (imaging
// active item of dropdown).

```

```
$navigation->setEnabledDropDownHighlight (true) ;

// This content will be prepended to each item inside navigation.
$navigation->setItemPrependContent ('|');

// This content will be appended to each item inside navigation.
$navigation->setItemAppendContent ('|');
```

Languages And Translations

Multilanguages are supported out of the box. By default translation works from English to other languages (texts written in English).

Translation system works in two modes:

- **Development mode.** Translation stored in database. PhalconEye allows to add, edit, delete, translate such translations. In this mode all translations that wasn't found in database - will be added automatically (it collects them).
- **Production mode.** All translations stored in <LANGUAGE_UNIQ_KEY>.php file (e.g. “en.php”). There are no database queries or collection of not existing items. Just collection of translations.

Each language object contains name (title), unique key (e.g.”en”), locale (e.g. en_US). It means that you can have two equals English but with different locales.

Translation in controller:

```
<?php

// Get it from DI and translate.
$this->di->get('i18n')->_('Actions');

// Or like that.
$this->di->getI18n()->_('options :one: and :two', ['one' => 1, 'two' => 2]);

// Or:
$this->di->getI18n()->query('Options :one: and :two', ['one' => 1, 'two' => 2]);
```

Translation in view:

```
{ { 'Login'|i18n } }
```

Different tools:

```
{# Format number according to current locale. #}
{# Output: 100.0 #}
{{ helper('formatter') .formatNumber(100, php('\\NumberFormatter::DECIMAL')) }}
```



```
{# Format currency according to current locale. #}
{# Output: $100 #}
{{ helper('formatter') .formatCurrency(100) }}
```

Current language and it's locale stored in session, so to change language:

```
<?php

$language = preg_replace("/[^A-Za-z0-9?!]/", '', $this->request->get('lang', 'string
↔'));
```

```

if ($language && $languageObject = Language::findFirst("language = '" . $language . "'"))
{
    $this->di->get('session')->set('language', $languageObject->language);
    $this->di->get('session')->set('locale', $languageObject->locale);
}

```

Access Control List

ACL is based on [Phalcon ACL](#). Acl Roles are stored in database. Each user can have only one role. In production mode Acl compiles from database and is cached by the system.

There is only one default Acl key: Core\Api\Acl::ACL_ADMIN_AREA ('AdminArea'). This key is used for admin panel access. By default there are three roles: Admin, User and Guest. All not authenticated requests assigned to Guest Role. Logged-in sessions are assigned to User. Admin is the most privileged Role.

ACL Usage

Acl class is part of Core module API and can be accessed via api container (core container) from DI.

In controller:

```

<?php

// Check if current user has access to perform given action on the resource.
$this->core->acl()->isAllowed($viewer->getRole()->name, $resource, $action) ==_
↪Acl::ALLOW;

// Get allowed value on given resource for user.
$this->getDI()->get('core')->acl()->getAllowedValue($resource, $viewer->getRole(),
↪$valueName);

```

In view:

```

{# Check if user is allowed to view, and show something. #}
{%
    if helper('acl').isAllowed('\Core\Model\Page', 'show_views') %}
        <div class="page_views">{{ 'View count:'|i18n }}{{ page.view_count }}</div>
{%
    endif
}

{# Check if user is allowed to view, and show something. #}
{{ helper('acl').getAllowed('\Core\Model\Page', 'page_footer') }}

```

Model ACL

Let's take the Blog system as an example. We can allow or disallow access for some roles to perform actions such as: "create", "edit" and "delete". Also we have two values:

- "blog_footer" - Displays some HMTL content in footer on each blog (eg. for user: "Hello world" and for guest: "Bye").
- "blog_count" - Number of blogs per page on browse page.

We can also define required actions and their values in blog model via annotation @Acl:

```
<?php

/**
 * Blog model.
 *
 * @category PhalconEye
 * @package Blog\Model
 * @author Ivan Vorontsov <ivan.vorontsov@phalconeye.com>
 * @copyright 2013-2014 PhalconEye Team
 * @license New BSD License
 * @link http://phalconeye.com/
 *
 * @Source("blogs")
 * @Acl(actions={"create", "edit", "delete"}, options={"blog_footer", "blog_count"})
 */
class Blog extends AbstractModel
{

}
```

After defining required actions and values you can set their values in admin panel via Access Rights system. Note: In development mode PhalconEye will automatically pick up all changes made to models.

Console

Console allows to run some commands for remote purpose.

To run console manager, you will need to move into website root directory and execute:

```
php public/index.php command param1 param2
```

Usage

To use command simple type command and/or sub commands if required:

```
php public/index.php database update
```

Some commands have aliases:

```
php public/index.php db update
```

To show all available commands you can type:

```
php public/index.php

## or ##

php public/index.php help
```

Help can be used for command:

```
lantian@linux-ff8d:/www/ph.l/www> php public/index.php help db
=====
  /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \
   \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
    Commands Manager
=====

Help:
Database management.

  database update          Update database schema according to models metadata.
```

Or sub command:

```
lantian@linux-ff8d:/www/ph.l/www> php public/index.php help db update
=====
  /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \ /-\ \
   \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
    Commands Manager
=====

Help for "database update":
  Update database schema according to models metadata.

Available parameters:
  --model=NULL (string|null)      Model name to update. Example: \Test\Model\Class.
  --cleanup                         Cleanup database? Drop not related tables.
```

Commands

Name	Sub command	Parameters	Description
cache	cleanup	—	Removes all cache stored by PhalconEye. Also removes view cache, metadata, annotations, assets, etc.
database, db	update	<p>-- model=NULL (string null) - Model class name, that must be updated. Example: CoreModel-Some</p> <p>-- cleanup - Remove tables, that isn't related (not mentioned) to CMS</p>	Updates all tables related to CMS and their relations according to models metadata (annotations) defined in code.
application, app	sync	—	Synchronize application data with current copy of code. Imagine that you've created one module (that is related to other module) and external widget. And you have server for development and for testing. Changes were made on development server and you need to move them into testing server. You committed all changed data and fetched them on on testing server (module and widget code and *.json metadata). You runs this command and PhalconEye synchronize all data in database: installs new, removes old, etc... And adds new items to autoload system. So, this tool is for synchronization between stage databases.
assets	install	—	Installs all assets from modules and compiles theme less files.

Command Creation

Command can be created in module. Special directory “Command” must be placed in module root.

Command example:

```
<?php
```

```

/*
 * Assets command.
 *
 * @category PhalconEye
 * @package Core\Commands
 * @author Ivan Vorontsov <ivan.vorontsov@phalconeye.com>
 * @copyright 2013-2014 PhalconEye Team
 * @license New BSD License
 * @link http://phalconeye.com/
 *
 * @CommandName(['assets'])
 * @CommandDescription('Assets management.')
 */
class Assets extends AbstractCommand implements CommandInterface
{
    /**
     * Install assets from modules.
     *
     * @return void
     */
    public function installAction()
    {
        $assetsManager = new Manager($this->getDI(), false);
        $assetsManager->installAssets(PUBLIC_PATH . '/themes/' . Settings::getSetting(
            'system_theme'));

        print ConsoleUtil::success('Assets successfully installed.') . PHP_EOL;
    }
}

```

Each command must be extended from `AbstractCommand` and implements `CommandInterface`. Commands metadata defined via class annotations:

```

<?php

/**
 * @CommandName(['commandname', 'commandalias'])
 * @CommandDescription('Description of the command.')
 */
class SomeCommand extends AbstractCommand {}

/**
 * Command can gave initialization method, that will be performed before any action.
 *
 * @CommandName(['commandname', 'commandalias'])
 * @CommandDescription('Description of the command.')
 */
class SomeCommand extends AbstractCommand {
    public function initialize() {}
}

/**
 * To define sub command - add subcommandAction method. It will be automatically
 * added as sub command.
 *
 * @CommandName(['commandname', 'commandalias'])
 * @CommandDescription('Description of the command.')
 */

```

```

class SomeCommand extends AbstractCommand {
    public function subcommandAction() {}
}

/*
 * Parameters of sub command automatically takes as parameters of it.
 * NOTE: action with parameters must be commented well, coz this will be a
 →description of this commands!
 *
 * @CommandName(['commandname', 'commandalias'])
 * @CommandDescription('Description of the command.')
 */

class SomeCommand extends AbstractCommand {
    /*
 * Test action with params.
 *
 * @param string|null $param1 Param1 - string. Example: "string".
 * @param bool         $param2 Param2 is flag.
 *
 * @return void
 */
    public function testAction($param1 = null, $param2 = false) {}

    // Help for this command will looks like this:
//Help for "commandname test":
// Test action with params.
//
//Available parameters:
// --param1=NULL (string|null)           Param1 - string. Example: "string".
// --param2                           Param2 is flag.
}

```

Assets

Assets are public files which are bundled with Modules and the CMS itself. Before we start explaining how these are handled you should understand the difference between public and not-public files.

- “/app” - private folder whose files can not be accessed directly from requests.
- “/public” - public front-end folder which stores Assets and templates.

For example the Blog module has css, less, js files and images which can not be accessed directly because of their location on the web server (/app/modules/Blog).

Since these files must be available publicly the Assets system was implemented. It will take the files, merge, minify and copy them over to the public folder, that’s it! You will never need to worry about doing all this on your own.

There is a console command to install the Modules’ Assets and compile Theme’s less files:

```
php public/index.php assets install
```

Files will be available under public directory, public/assets:

```

public/assets
- css                                     // All css files from all
→modules.
|   - blog                                  // Module name.
|   - constants.css                         // Constants from theme.

```

```

|   - core
|   |   - admin
|   |   |   - main.css
|   |   - install.css
|   |   - profiler.css
|   - theme.css                                // Main theme file.
|   - user
- img                                         // All images from modules.
|   - blog
|   - core
|   |   - admin
|   |   |   - content
|   |   |   - middle_bottom.png
|   |   |   - middle_left_bottom.png
|   |   |   - middle_left.png
|   |   |   - middle.png
|   |   |   - placeholder.png
|   |   |   - right_middle_bottom.png
|   |   |   - right_middle_left_bottom.png
|   |   |   - right_middle_left.png
|   |   |   - right_middle.png
|   |   |   - top_middle_left.png
|   |   |   - top_middle.png
|   |   |   - top_right_middle_left.png
|   |   |   - top_right_middle.png
|   |   - grid
|   |   |   - filter-clear.png
|   |   - loader
|   |   |   - black.gif
|   |   |   - white.gif
|   - misc
|   |   |   - icon-chevron-down.png
|   |   |   - icon-chevron-up.png
|   |   - pe_logo.png
|   |   - pe_logo_white.png
|   |   - profiler
|   |   |   - bug.png
|   |   |   - close.png
|   |   |   - files.png
|   |   |   - memory.png
|   |   |   - sql.png
|   |   |   - time.png
|   |   - stripe.png
|   - user
- javascript.js                               // Merged js files. Will be used in production mode.
- js                                         // All js files, not merged.
|   - blog
|   - core
|   |   - admin
|   |   |   - dashboard.js
|   |   |   - languages.js
|   |   |   - menu.js
|   |   - core.js
|   |   - form
|   |   |   - remote-file.js
|   |   - form.js
|   |   - i18n.js

```

```

|   |   - pretty-exceptions
|   |   |   - js
|   |   |   |   - jquery.scrollTo-min.js
|   |   |   |   - pretty.js
|   |   |   - prettify
|   |   |   |   - lang-apollo.js
|   |   |   |   - lang-clj.js
|   |   |   |   - lang-css.js
|   |   |   |   - lang-go.js
|   |   |   |   - lang-hs.js
|   |   |   |   - lang-lisp.js
|   |   |   |   - lang-lua.js
|   |   |   |   - lang-ml.js
|   |   |   |   - lang-n.js
|   |   |   |   - lang-proto.js
|   |   |   |   - lang-scala.js
|   |   |   |   - lang-sql.js
|   |   |   |   - lang-tex.js
|   |   |   |   - lang-vb.js
|   |   |   |   - lang-vhdl.js
|   |   |   |   - lang-wiki.js
|   |   |   |   - lang-xq.js
|   |   |   |   - lang-yaml.js
|   |   |   |   - prettify.css
|   |   |   |   - prettify.js
|   |   |   - themes
|   |   |   |   - default.css
|   |   |   |   - minimalist.css
|   |   |   |   - night.css
|   |   - profiler.js
|   |   - widgets
|   |   |   - autocomplete.js
|   |   |   - ckeditor.js
|   |   |   - grid.js
|   |   |   - modal.js
|   - user
- style.css                                     // Merged css files. Used
                                                 ↵in production mode.

```

To install assets from the code:

```

<?php

$assetsManager = new Manager($this->getDI(), false);

// Install assets, using theme directory.
$assetsManager->installAssets(PUBLIC_PATH . '/themes/' . Settings::getSetting('system_
↪theme'));

// First parameter - refresh assets, this means that old will be removed, new - added.
// If first parameter is true - second is required (theme directory).
$assetsManager->clear(true, PUBLIC_PATH . '/themes/' . Settings::getSetting('system_
↪theme'));

// Get assets collections for JS or CSS:
$assetsManager->getEmptyJsCollection();
$assetsManager->getEmptyCssCollection();

```

```
// Add inline scripts (css or js) to <head>.
$assetsManager->addInline('test', '<link rel="stylesheet" href="../../_static/css/
˓→docs.css" type="text/css"/>');
$assetsManager->removeInline('test');
```

You can get more details about the Assets from .

Cache

Cache is required to improve performance. PhalconEye has 4 types of cache:

Name	Description
viewCache	Used by Phalcon to cache views. http://docs.phalconphp.com/en/latest/reference/views.html#caching-view-fragments
cacheOutput	Used by developer to cache output data (pieces of HTML or text).
cacheData	Used by developer to cache data (rows, arrays, etc).
modelsCache	Used by Phalcon to cache ORM's data. http://docs.phalconphp.com/en/latest/reference/models-cache.html

In development mode all cache data will be handled with non-persistent Dummy layer which does not store any data.

You can read more about cache system in .

CHAPTER 3

Other formats

- PDF
- HTML in one Zip
- ePub