
ubiquity-framework Documentation

phpmv

Jan 25, 2021

1	Quick start with console	1
2	Quick start with web tools	9
3	Ubiquity-devtools installation	21
4	Project creation	23
5	Project configuration	25
6	Devtools usage	29
7	URLs	33
8	Router	37
9	Controllers	51
10	Events	59
11	Dependency injection	63
12	CRUD Controllers	69
13	Auth Controllers	81
14	Database	89
15	Models generation	93
16	ORM	95
17	DAO	103
18	Request	113
19	Response	117
20	Session	121

21	Cookie	125
22	Views	127
23	Assets	131
24	Themes	133
25	jQuery Semantic-UI	141
26	Normalizers	151
27	Validators	153
28	Transformers	159
29	Translation module	165
30	Security	169
31	Security module	173
32	ACL management	181
33	Rest	189
34	Webtools	217
35	Contributing	223
36	Coding guide	225
37	Documenting guide	231
38	Ubiquity commands	233
39	Composer management	241
40	Ubiquity Caching	245
41	Ubiquity dependencies	247
42	OAuth2 client module	249
43	Async platforms	255
44	Indices and tables	259

CHAPTER 1

Quick start with console

Note: If you do not like console mode, you can switch to quick-start with *web tools (UbiquityMyAdmin)*.

1.1 Install Composer

ubiquity utilizes Composer to manage its dependencies. So, before using, you will need to make sure you have [Composer](#) installed on your machine.

1.2 Install Ubiquity-devtools

Download the Ubiquity-devtools installer using Composer.

```
composer global require phpmv/ubiquity-devtools
```

Test your recent installation by doing:

```
Ubiquity version
```

```
• PHP 7.2.15-0ubuntu0.18.04.1
• Ubiquity devtools (1.1.3)
```

You can get at all times help with a command by typing: `Ubiquity help` followed by what you are looking for.

Example :

```
Ubiquity help project
```

1.3 Project creation

Create the **quick-start** projet

```
Ubiquity new quick-start
```

1.4 Directory structure

The project created in the **quick-start** folder has a simple and readable structure:

the **app** folder contains the code of your future application:

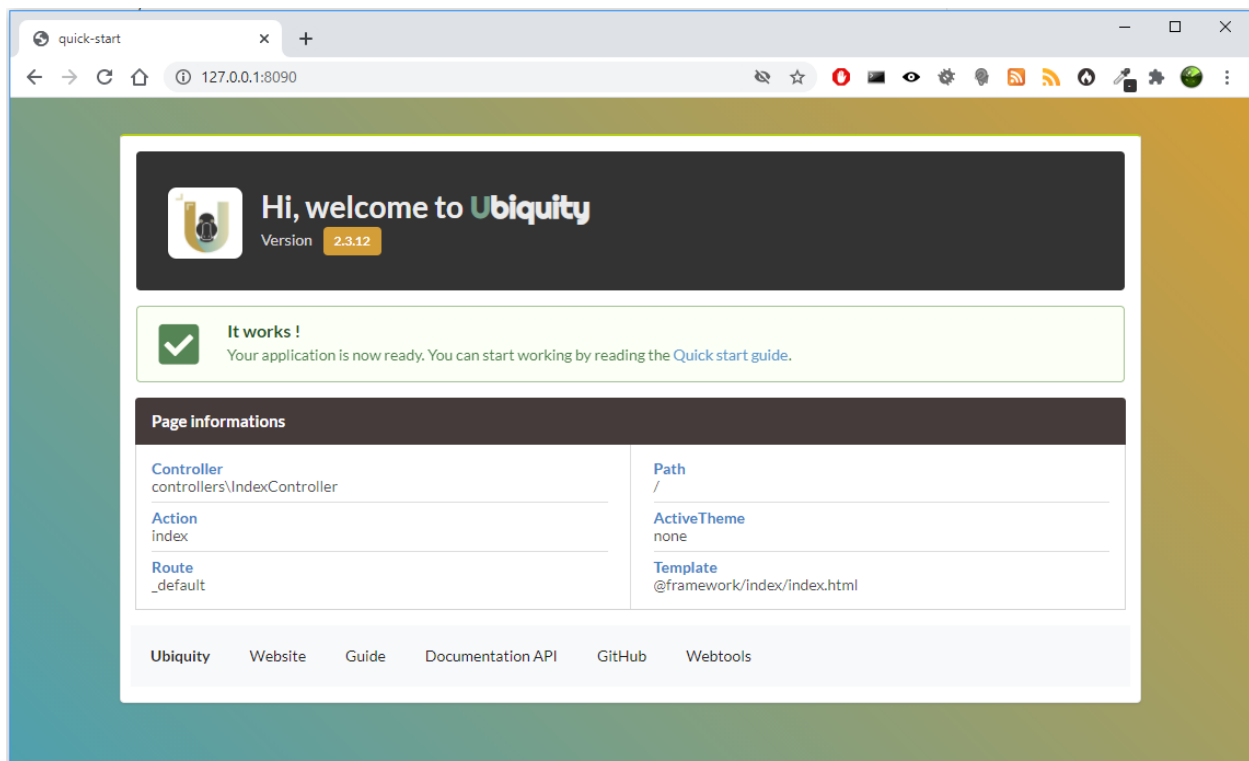
```
app
├── cache
├── config
├── controllers
├── models
└── views
```

1.5 Start-up

Go to the newly created folder **quick-start** and start the build-in php server:

```
Ubiquity serve
```

Check the correct operation at the address **http://127.0.0.1:8090**:



Note: If port 8090 is busy, you can start the server on another port using -p option.

```
Ubiquity serve -p=8095
```

1.6 Controller

The console application **dev-tools** saves time in repetitive operations. We go through it to create a controller.

```
Ubiquity controller DefaultController
```

```

· The project folder is /var/www/html/quick-start
■ success : Controller creation
· Creation of the Controller DefaultController at the location app/controllers/DefaultController.php
```

We can then edit `app/controllers/DefaultController` file in our favorite IDE:

Listing 1: `app/controllers/DefaultController.php`

```

1 namespace controllers;
2 /**
3  * Controller DefaultController
4  */
5 class DefaultController extends ControllerBase{
6     public function index() {}
7 }
```

Add the traditional message, and test your page at `http://127.0.0.1:8090/DefaultController`

Listing 2: `app/controllers/DefaultController.php`

```

class DefaultController extends ControllerBase{

    public function index() {
        echo 'Hello world!';
    }

}
```

For now, we have not defined routes, Access to the application is thus made according to the following scheme: `controllerName/actionName/param`

The default action is the **index** method, we do not need to specify it in the url.

1.7 Route

Important: The routing is defined with the attribute `Route` (with `php>8`) or the annotation `@route` and is not done in a configuration file: it's a design choice.

The **automated** parameter set to **true** allows the methods of our class to be defined as sub routes of the main route /hello.

With annotations:

Listing 3: app/controllers/DefaultController.php

```
1 namespace controllers;
2 /**
3  * Controller DefaultController
4  * @route("/hello","automated"=>true)
5  */
6 class DefaultController extends ControllerBase{
7
8     public function index(){
9         echo 'Hello world!';
10    }
11
12 }
```

With attributes (php>8):

Listing 4: app/controllers/DefaultController.php

```
1 namespace controllers;
2 use Ubiquity\attributes\items\router\Route;
3
4 #[Route('/hello', automated: true)]
5 class DefaultController extends ControllerBase{
6
7     public function index(){
8         echo 'Hello world!';
9     }
10
11 }
```

1.7.1 Router cache

Important: No changes on the routes are effective without initializing the cache. Annotations are never read at runtime. This is also a design choice.

We can use the console for the cache re-initialization:

```
Ubiquity init-cache
```

```
■ success : init-cache:all
  · cache directory is /var/www/html/quick-start/app/cache/
  · Models directory is /var/www/html/quick-start/app/models
  · Models cache reset
  · Controllers directory is /var/www/html/quick-start/app/controllers
  · Router cache reset
  · Controllers directory is /var/www/html/quick-start/app/controllers
  · Rest cache reset
```


Let's check that the route exists:

```
Ubiquity info:routes
```

path	controller	action	parameters
/hello/(index/)?	controllers\DefaultController	index	[]

· 1 routes (routes)

We can now test the page at <http://127.0.0.1:8090/hello>

1.8 Action & route with parameters

We will now create an action (sayHello) with a parameter (name), and the associated route (to): The route will use the parameter **name** of the action:

```
Ubiquity action DefaultController.sayHello -p=name -r-to/{name}/
```

```
■ info
  · You need to re-init Router cache to apply this update with init-cache command

■ info : Creation
  · The action sayHello is created in controller controllers\DefaultController
```

After re-initializing the cache (**init-cache** command), the **info:routes** command should display:

path	controller	action	parameters
/hello/(index/)?	controllers\DefaultController	index	[]
/hello/to/(.+?)/		sayHello	[name*]

· 2 routes (routes)

Change the code in your IDE: the action must say Hello to somebody...

Listing 5: app/controllers/DefaultController.php

```
/**
 * @route("to/{name}/")
 */
public function sayHello($name) {
    echo 'Hello ' . $name . '!!';
}
```

and test the page at [http://127.0.0.1:8090/hello/to/Mr SMITH](http://127.0.0.1:8090/hello/to/Mr%20SMITH)

1.9 Action, route parameters & view

We will now create an action (information) with two parameters (title and message), the associated route (info), and a view to display the message: The route will use the two parameters of the action.

```
Ubiquity action DefaultController.information -p=title,message='nothing' -r=info/
↪ {title}/{message} -v
```

Note: The -v (-view) parameter is used to create the view associated with the action.

After re-initializing the cache, we now have 3 routes:

path	controller	action	parameters
/hello/(index/)?	controllers\DefaultController	index	[]
/hello/to/(.+?)/		sayHello	[name*]
/hello/info/(.+?)/(.*)?		information	[title*,message]

• 3 routes (routes)

Let's go back to our development environment and see the generated code:

Listing 6: app/controllers/DefaultController.php

```
/**
 * @route("info/{title}/{message}")
 */
public function information($title,$message='nothing') {
    $this->loadView('DefaultController/information.html');
}
```

We need to pass the 2 variables to the view:

```
/**
 * @route("info/{title}/{message}")
 */
public function information($title,$message='nothing') {
    $this->loadView('DefaultController/information.html',compact('title','message'
↪ ));
}
```

And we use our 2 variables in the associated twig view:

Listing 7: app/views/DefaultController/information.html

```
<h1>{{title}}</h1>
<div>{{message | raw}}</div>
```

We can test your page at [http://127.0.0.1:8090/hello/info/Quick start/Ubiquity is quiet simple](http://127.0.0.1:8090/hello/info/Quick%20start/Ubiquity%20is%20quiet%20simple) It's obvious



Quick start with web tools

2.1 Install Composer

ubiquity utilizes Composer to manage its dependencies. So, before using, you will need to make sure you have [Composer](#) installed on your machine.

2.2 Install Ubiquity-devtools

Download the Ubiquity-devtools installer using Composer.

```
composer global require phpmv/ubiquity-devtools
```

Test your recent installation by doing:

```
Ubiquity version
```

```
• PHP 7.2.15-0ubuntu0.18.04.1
• Ubiquity devtools (1.1.3)
```

You can get at all times help with a command by typing: `Ubiquity help` followed by what you are looking for.

Example :

```
Ubiquity help project
```

2.3 Project creation

Create the **quick-start** projet with **Webtools** interface (the **-a** option)

```
Ubiquity new quick-start -a
```

2.4 Directory structure

The project created in the **quick-start** folder has a simple and readable structure:
the **app** folder contains the code of your future application:

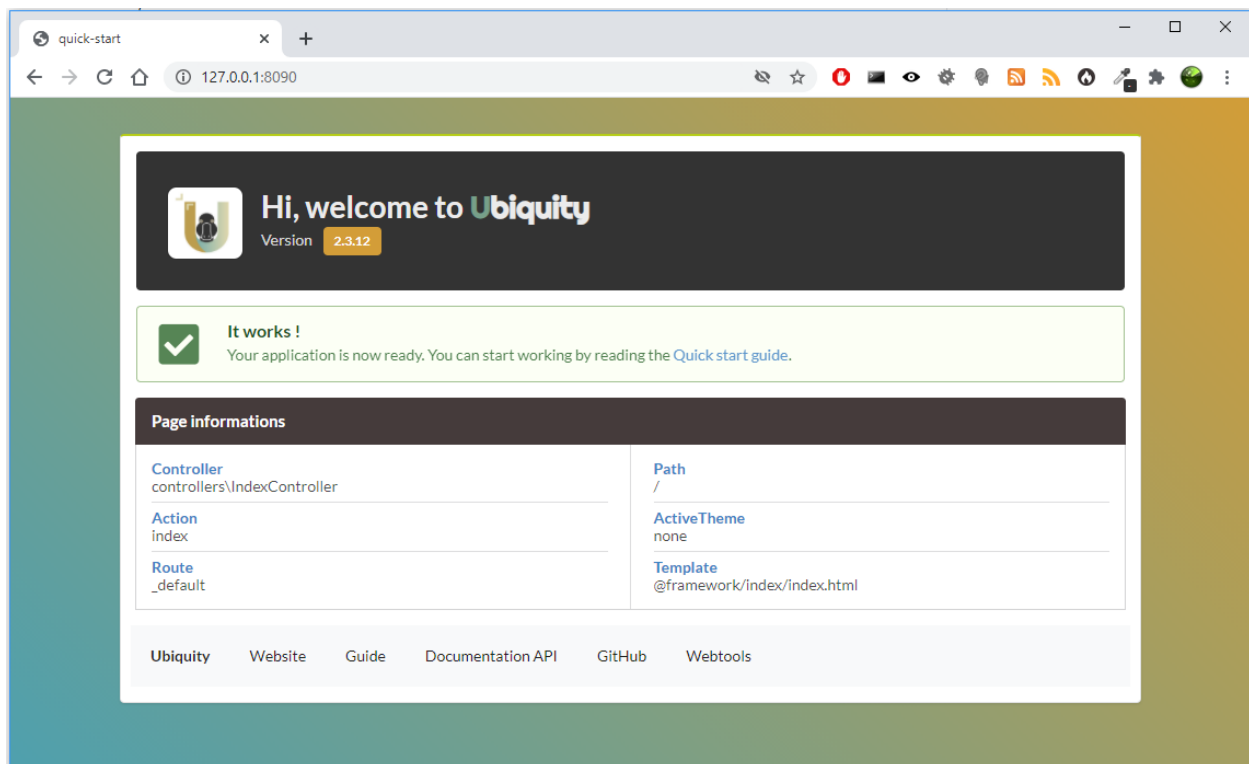
```
app
├── cache
├── config
├── controllers
├── models
└── views
```

2.5 Start-up

Go to the newly created folder **quick-start** and start the build-in php server:

```
Ubiquity serve
```

Check the correct operation at the address **http://127.0.0.1:8090**:



Note: If port 8090 is busy, you can start the server on another port using -p option.


```
Ubiquity serve -p=8095
```

2.6 Controller

Goto admin interface by clicking on the button **Webtools**:

The screenshot displays the Webtools administration interface. At the top, a navigation bar includes a home icon and links for Webtools, models, routes, controllers, cache, rest, config, seo, translate, themes, maintenance, composer, and security. Below this is a light blue notification box titled 'Tools displaying' with an information icon and a close button, containing the text: 'This is your first use of devtools. You can select the tools you want to display.' The main section has a dark header with a gear icon, the title 'Webtools', the subtitle 'Ubiquity framework administration', and a 'Customize...' button. The content area is titled 'Ordering and selecting tools' and features two columns: 'Left side' and 'Right side'. The 'Left side' column contains buttons for 'models', 'routes', 'controllers', 'cache', 'rest', and 'config', each with a close icon and a dropdown arrow. The 'Right side' column contains buttons for 'seo', 'translate', 'themes', 'maintenance', and 'composer', also with close icons and dropdown arrows. A red-bordered button labeled 'Reset configuration parameters' is positioned below the tool lists. At the bottom, there are two buttons: a green 'Validate' button with a checkmark icon and a grey 'Cancel' button with a circle icon.

The web application **Webtools** saves time in repetitive operations.

 Webtools

models

routes

controllers

cache


config

translate

composer


security

commands


 Webtools

Ubiquity framework administration


Customize...

 Models


Used to perform CRUD operations on data.

 Routes


Displays defined routes with annotations

 Controllers


Displays controllers and actions

 Cache


Annotations, models, router and controller cache

 Config


Configuration variables

 Translate


Translation module

 Composer

Manages composer dependencies

 Security

Manages security

 Commands

Devtools commands


We go through it to create a controller.

Go to the **controllers** part, enter **DefaultController** in the **controllerName** field and create the controller:




View


DefaultController


 Create controller

The controller **DefaultController** is created:




The **DefaultController** controller has been created in C:\xampp\htdocs\quick-start-2\ubiquity\app\controllers\DefaultController.php.















View

Controller name

 Create controller

 Create special controller

 Filter controllers

Controller	Action [routes]	Default values
 controllers\DefaultController 	 index ()	
 controllers\IndexController 	 index ()  /_default/  @framework/index/semantic.html	

We can then edit `app/controllers/DefaultController` file in our favorite IDE:

Listing 1: `app/controllers/DefaultController.php`

```
1 namespace controllers;
2 /**
3  * Controller DefaultController
4  */
5 class DefaultController extends ControllerBase{
6     public function index() {}
7 }
```


Add the traditional message, and test your page at `http://127.0.0.1:8090/DefaultController`

Listing 2: `app/controllers/DefaultController.php`

```
class DefaultController extends ControllerBase{

    public function index(){
        echo 'Hello world!';
    }

}
```

For now, we have not defined routes, Access to the application is thus made according to the following scheme: `controllerName/actionName/param`

The default action is the **index** method, we do not need to specify it in the url.

2.7 Route

Important: The routing is defined with the annotation `@route` and is not done in a configuration file: it's a design choice.

The **automated** parameter set to **true** allows the methods of our class to be defined as sub routes of the main route `/hello`.

Listing 3: `app/controllers/DefaultController.php`

```
1 namespace controllers;
2 /**
3  * Controller DefaultController
4  * @route("/hello","automated"=>true)
5  */
6 class DefaultController extends ControllerBase{
7
8     public function index(){
9         echo 'Hello world!';
10    }
11
12 }
```

2.7.1 Router cache

Important: No changes on the routes are effective without initializing the cache. Annotations are never read at runtime. This is also a design choice.

We can use the **web tools** for the cache re-initialization:

Go to the **Routes** section and click on the **re-init cache** button



The route now appears in the interface:

Routes
Displays defined routes with annotations

Router cache entry is /var/www/html/quick-start/ubiquity/./app/cache/controllers/routes.default.cache.php

(Re-)Init router cache

Path	Methods	Action & parameters	Cache	Expired	Name
<div> controllers\DefaultController::class </div>					
<div> /hello/{index}/? </div>		<div> index () </div>	<input type="checkbox"/>		DefaultController-index

We can now test the page by clicking on the **GET** button or by going to the address `http://127.0.0.1:8090/hello`

2.8 Action & route with parameters

We will now create an action (sayHello) with a parameter (name), and the associated route (to): The route will use the parameter **name** of the action:

Go to the **Controllers** section:

- click on the + button associated with DefaultController,
- then select **Add new action in..** item.

Controller

controllers\DefaultController

+

Add new action in controllers\DefaultController...

Enter the action information in the following form:

Creating a new action in controller

Controller

controllers\DefaultController

Action & parameters

sayHello

name

Implementation

```
echo 'Hello '.$name.'!';
```

☐ Create associated view
 ☒ Add route...

to/{name}/

☐ Duration

Validate

Close

After re-initializing the cache with the orange button, we can see the new route **hello/to/{name}**:

Controller	Action [routes]	Default values
	⚡ index () 🚗 /hello/{index}?	
<div> <div>♥</div> <div>controllers\DefaultController</div> <div>+</div> </div>	⚡ sayHello (name) 🚗 /hello/to/{.+?}/	

Check the route creation by going to the Routes section:

Path	Methods	Action & parameters	Cache	Expired	Name
♥ controllers\DefaultController::class					
🚗 /hello/{index}?		index ()	<input type="checkbox"/>		DefaultController-index
🚗 /hello/to/{.+?}/		sayHello (name*)	<input type="checkbox"/>		DefaultController-sayHello

GET

POST

▼

We can now test the page by clicking on the **GET** button:

GET:/hello/to/(.+?)/



Required URL parameters

You must complete the following parameters before continuing navigation testing

Name *

Mr SMITH

Validate

Close

We can see the result:

GET:/hello/to/(.+?)/

Hello Mr SMITH!

Close

We could directly go to `http://127.0.0.1:8090/hello/to/Mr SMITH` address to test

2.9 Action, route parameters & view

We will now create an action (information) with tow parameters (title and message), the associated route (info), and a view to display the message: The route will use the two parameters of the action.

In the **Controllers** section, create another action on **DefaultController**:

Controller



controllers\DefaultController



Add new action in controllers\DefaultController...

Enter the action information in the following form:

Creating a new action in controller

Controller

controllers\DefaultController

Action & parameters

information

title,message='nothing'

Implementation

Implementation

☒ Create associated view

☒ Add route...

info/{title}/{message}/

☐ Duration

Validate

Close

Note: The view checkbox is used to create the view associated with the action.

After re-initializing the cache, we now have 3 routes:

Controller	Action [routes]	Default values
	index () /hello/{index}/?	
controllers\DefaultController	sayHello (name) /hello/to/{.+?}/	
	information (title, message) /hello/info/{.+?}/{/*?}	message="nothing"
	DefaultController/information.html	

Let's go back to our development environment and see the generated code:

Listing 4: app/controllers/DefaultController.php

```

/**
 * @route("info/{title}/{message}")
 */
public function information($title, $message='nothing') {
    $this->loadView('DefaultController/information.html');
}

```

We need to pass the 2 variables to the view:

```
/**
 *@route("info/{title}/{message}")
 */
public function information($title,$message='nothing'){
    $this->loadView('DefaultController/information.html',compact('title','message
    ↵')));
}
```

And we use our 2 variables in the associated twig view:

Listing 5: app/views/DefaultController/information.html

```
<h1>{{title}}</h1>
<div>{{message | raw}}</div>
```

We can test our page at <http://127.0.0.1:8090/hello/info/Quick start/Ubiquity is quiet simple> It's obvious



New in documentation

- *Security*
- *Async platforms*
- *Commands module*
- *Composer module*
- *OAuth client module*
- *Mailer module*
- *Servers configuration*
- *Database connection*
- *Optimization*
- *Rich client*
- *REST module*
- *Data transformers*
- *Dependency injection*
- *Events*
- *Views and themes*

- *Contributing*
- *Quick start with webtools (UbiquityMyAdmin)*
- Generating models:
 - with webtools (UbiquityMyAdmin)
 - with console (devtools)

Ubiquity-devtools installation

3.1 Install Composer

ubiquity utilizes Composer to manage its dependencies. So, before using, you will need to make sure you have [Composer](#) installed on your machine.

3.2 Install Ubiquity-devtools

Download the Ubiquity-devtools installer using Composer.

```
composer global require phpmv/ubiquity-devtools
```

Make sure to place the `~/composer/vendor/bin` directory in your PATH so the **Ubiquity** executable can be located by your system.

Once installed, the simple `Ubiquity new` command will create a fresh Ubiquity installation in the directory you specify. For instance, `Ubiquity new blog` would create a directory named **blog** containing an Ubiquity project:

```
Ubiquity new blog
```

The semantic option adds Semantic-UI for the front end.

You can see more options about installation by reading the [Project creation](#) section.

CHAPTER 4

Project creation

After installing *Ubiquity-devtools installation*, in your terminal, call the *new* command in the root folder of your web server :

4.1 Samples

A simple project

```
Ubiquity new projectName
```

A project with UbiquityMyAdmin interface

```
Ubiquity new projectName -a
```

A project with bootstrap and semantic-ui themes installed

```
Ubiquity new projectName --themes-bootstrap,semantic
```

4.2 Installer arguments

short name	name	role	default	Allowed values	Since dev-tools
b	dbName	Sets the database name.			
s	server-Name	Defines the db server address.	<i>127.0.0.1</i>		
p	port	Defines the db server port.	<i>3306</i>		
u	user	Defines the db server user.	<i>root</i>		
w	pass-word	Defines the db server password.	<i>'</i>		
h	themes	Install themes.		semantic,bootstrap,foundation	
m	all-models	Creates all models from db.	<i>false</i>		
a	admin	Adds UbiquityMyAdmin interface.	<i>false</i>		
i	siteUrl	Defines the site URL.	<i>http://127.0.0.1/{projectname}</i>		1.2.6
e	rewrite-Base	Sets the base for rewriting.	<i>/{projectname}/</i>		1.2.6

4.3 Arguments usage

4.3.1 short names

Example of creation of the **blog** project, connected to the **blogDb** database, with generation of all models

```
Ubiquity new blog -b=blogDb -m=true
```

4.3.2 long names

Example of creation of the **blog** project, connected to the **blogDb** database, with generation of all models and integration of semantic theme

```
Ubiquity new blog --dbName=blogDb --all-models=true --themes=semantic
```

4.4 Running

To start the embedded web server and test your pages, run from the application root folder:

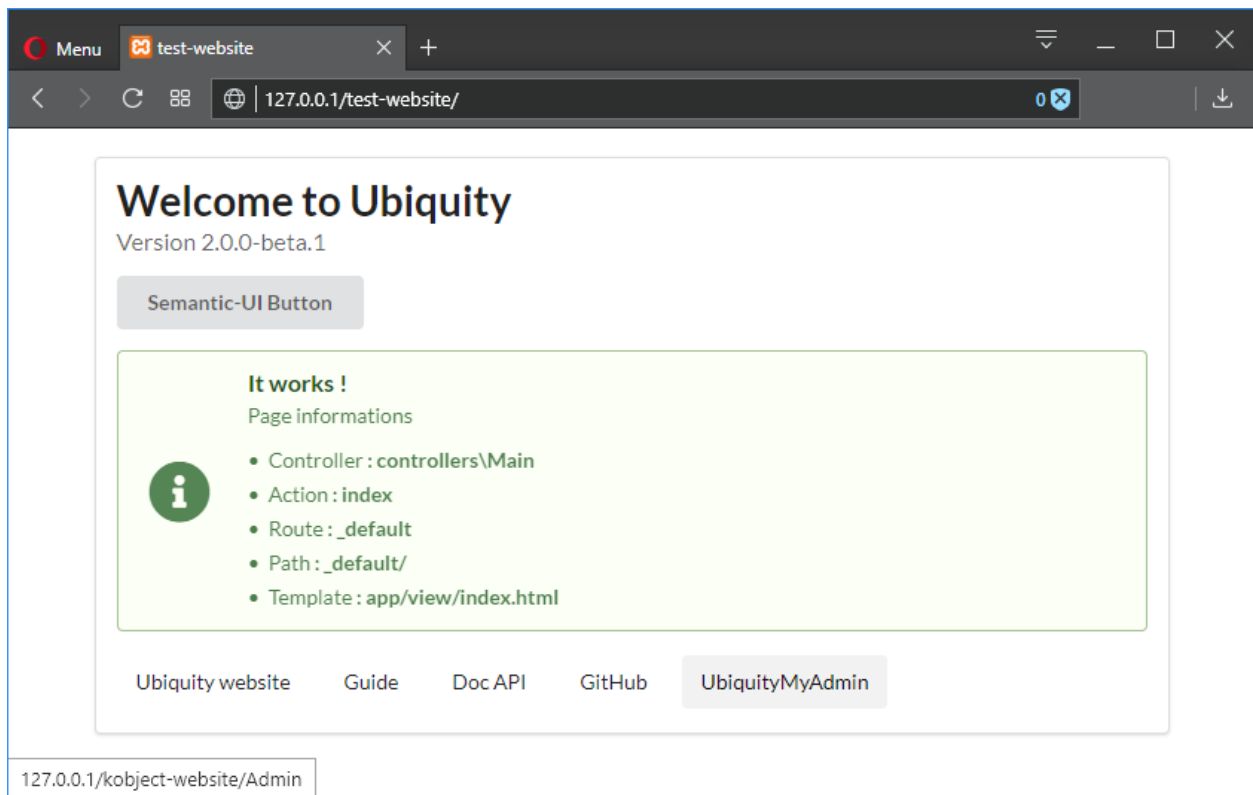
```
Ubiquity serve
```

The web server is started at `127.0.0.1:8090`

CHAPTER 5

Project configuration

Normally, the installer limits the modifications to be performed in the configuration files and your application is operational after installation



5.1 Main configuration

The main configuration of a project is localised in the `app/conf/config.php` file.

Listing 1: `app/conf/config.php`

```
1 return array(  
2     "siteUrl"=>"%siteUrl%",  
3     "database"=>[  
4         "dbName"=>"%dbName%",  
5         "serverName"=>"%serverName%",  
6         "port"=>"%port%",  
7         "user"=>"%user%",  
8         "password"=>"%password%"  
9     ],  
10    "namespaces"=>[],  
11    "templateEngine"=>'Ubiquity\views\engine\Twig',  
12    "templateEngineOptions"=>array("cache"=>false),  
13    "test"=>false,  
14    "debug"=>false,  
15    "di"=>[%injections%],  
16    "cacheDirectory"=>"cache/",  
17    "mvcNS"=>["models"=>"models", "controllers"=>"controllers"]  
18 );
```

5.2 Services configuration

Services loaded on startup are configured in the `app/conf/services.php` file.

Listing 2: `app/conf/services.php`

```
1 use Ubiquity\controllers\Router;  
2  
3 try{  
4     \Ubiquity\cache\CacheManager::startProd($config);  
5 } catch (Exception $e) {  
6     //Do something  
7 }  
8 \Ubiquity\orm\DAO::startDatabase($config);  
9 Router::start();  
10 Router::addRoute("_default", "controllers\\IndexController");
```

5.3 Pretty URLs

5.3.1 Apache

The framework ships with an `.htaccess` file that is used to allow URLs without `index.php`. If you use Apache to serve your Ubiquity application, be sure to enable the `mod_rewrite` module.

Listing 3: .htaccess

```
AddDefaultCharset UTF-8
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /blog/
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{HTTP_ACCEPT} !(*.images.*)
    RewriteRule ^(.*)$ index.php?c=$1 [L,QSA]
</IfModule>
```

See Apache configuration for more.

5.3.2 Nginx

On Nginx, the following directive in your site configuration will allow “pretty” URLs:

```
location /{
    rewrite ^/(.*)$ /index.php?c=$1 last;
}
```

See NginX configuration for more.

5.3.3 Laravel Valet Driver

Valet is a php development environment for Mac minimalists. No Vagrant, no /etc/hosts file. You can even share your sites publicly using local tunnels.

Laravel Valet configures your Mac to always run Nginx in the background when your machine starts. Then, using DnsMasq, Valet proxies all requests on the *.test domain to point to sites installed on your local machine.

Get more info about [Laravel Valet](#)

Create UbiquityValetDriver.php under ~/.config/valet/Drivers/ add below php code and save it.

```
<?php

class UbiquityValetDriver extends BasicValetDriver{

    /**
     * Determine if the driver serves the request.
     *
     * @param string $sitePath
     * @param string $siteName
     * @param string $uri
     * @return bool
     */
    public function serves($sitePath, $siteName, $uri){
        if(is_dir($sitePath . DIRECTORY_SEPARATOR . '.ubiquity')) {
            return true;
        }
        return false;
    }

    public function isStaticFile($sitePath, $siteName, $uri){
```

(continues on next page)

(continued from previous page)

```
        if(is_file($sitePath . $uri)) {
            return $sitePath . $uri;
        }
        return false;
    }

    /**
     * Get the fully resolved path to the application's front controller.
     *
     * @param string $sitePath
     * @param string $siteName
     * @param string $uri
     * @return string
     */
    public function frontControllerPath($sitePath, $siteName, $uri) {
        $_SERVER['DOCUMENT_ROOT'] = $sitePath;
        $_SERVER['SCRIPT_NAME'] = '/index.php';
        $_SERVER['SCRIPT_FILENAME'] = $sitePath . '/index.php';
        $_SERVER['DOCUMENT_URI'] = $sitePath . '/index.php';
        $_SERVER['PHP_SELF'] = '/index.php';

        $_GET['c'] = '';

        if($uri) {
            $_GET['c'] = ltrim($uri, '/');
            $_SERVER['PHP_SELF'] = $_SERVER['PHP_SELF'] . $uri;
            $_SERVER['PATH_INFO'] = $uri;
        }

        $indexPath = $sitePath . '/index.php';

        if(file_exists($indexPath)) {
            return $indexPath;
        }
    }
}
```


6.1 Project creation

See *Project creation* to create a project.

Tip: For all other commands, you must be in your project folder or one of its subfolders.

Important: The `.ubiquity` folder created automatically with the project allows the devtools to find the root folder of the project. If it has been deleted or is no longer present, you must recreate this empty folder.

6.2 Controller creation

6.2.1 Specifications

- `command : controller`
- `Argument : controller-name`
- `aliases : create-controller`

6.2.2 Parameters

short name	name	role	default	Allowed values
v	view	Creates the associated view index.	true	true, false

6.2.3 Samples:

Creates the controller `controllers\ClientController` class in `app/controllers/ClientController.php`:

```
Ubiquity controller ClientController
```

Creates the controller `controllers\ClientController` class in `app/controllers/ClientController.php` and the associated view in `app/views/ClientController/index.html`:

```
Ubiquity controller ClientController -v
```

6.3 Action creation

6.3.1 Specifications

- `command`: `action`
- `Argument`: `controller-name.action-name`
- `aliases`: `new-action`

6.3.2 Parameters

short name	name	role	default	Allowed values
p	params	The action parameters (or arguments).		a,b=5 or \$a,\$b,\$c
r	route	The associated route path.		/path/to/route
v	create-view	Creates the associated view.	false	true,false

6.3.3 Samples:

Adds the action `all` in controller `Users`:

```
Ubiquity action Users.all
```

code result:

Listing 1: `app/controllers/Users.php`

```
1 namespace controllers;
2 /**
3  * Controller Users
4  */
5 class Users extends ControllerBase{
6
7     public function index() {}
8
9     public function all() {
10
11     }
12
13 }
```

Adds the action display in controller Users with a parameter:

```
Ubiquity action Users.display -p=idUser
```

code result:

Listing 2: app/controllers/Users.php

```

1  class Users extends ControllerBase{
2
3      public function index() {}
4
5      public function display($idUser) {
6
7      }
8  }
```

Adds the action display with an associated route:

```
Ubiquity action Users.display -p=idUser -r=/users/display/{idUser}
```

code result:

Listing 3: app/controllers/Users.php

```

1  class Users extends ControllerBase{
2
3      public function index() {}
4
5      /**
6       *@route("/users/display/{idUser}")
7       */
8      public function display($idUser) {
9
10     }
11 }
```

Adds the action search with multiple parameters:

```
Ubiquity action Users.search -p=name,address=''
```

code result:

Listing 4: app/controllers/Users.php

```

1  class Users extends ControllerBase{
2
3      public function index() {}
4
5      /**
6       *@route("/users/display/{idUser}")
7       */
8      public function display($idUser) {
9
10     }
11
12     public function search($name,$address='') {
13 }
```

(continues on next page)

(continued from previous page)

14
15

```
}  
}
```

Adds the action `search` and creates the associated view:

```
Ubiquity action Users.search -p=name,address -v
```

6.4 Model creation

Note: Optionally check the database connection settings in the `app/config/config.php` file before running these commands.

To generate a model corresponding to the **user** table in database:

```
Ubiquity model user
```

6.5 All models creation

For generating all models from the database:

```
Ubiquity all-models
```

6.6 Cache initialization

To initialize the cache for routing (based on annotations in controllers) and orm (based on annotations in models) :

```
Ubiquity init-cache
```

CHAPTER 7

URLs

like many other frameworks, if you are using router with its default behavior, there is a one-to-one relationship between a URL string and its corresponding controller class/method. The segments in a URI normally follow this pattern:

```
example.com/controller/method/param  
example.com/controller/method/param1/param2
```

7.1 Default method

When the URL is composed of a single part, corresponding to the name of a controller, the index method of the controller is automatically called :

URL :

```
example.com/Products  
example.com/Products/index
```

Controller :

Listing 1: app/controllers/Products.php

```
1 class Products extends ControllerBase{  
2     public function index(){  
3         //Default action  
4     }  
5 }
```

7.2 Required parameters

If the requested method requires parameters, they must be passed in the URL:

Controller :

Listing 2: app/controllers/Products.php

```
1 class Products extends ControllerBase{
2     public function display($id) {}
3 }
```

Valid Urls :

```
example.com/Products/display/1
example.com/Products/display/10/
example.com/Products/display/ECS
```

7.3 Optional parameters

The called method can accept optional parameters.

If a parameter is not present in the URL, the default value of the parameter is used.

Controller :

Listing 3: app/controllers/Products.php

```
class Products extends ControllerBase{
    public function sort($field, $order="ASC") {}
}
```

Valid Urls :

```
example.com/Products/sort/name (uses "ASC" for the second parameter)
example.com/Products/sort/name/DESC
example.com/Products/sort/name/ASC
```

7.4 Case sensitivity

On Unix systems, the name of the controllers is case-sensitive.

Controller :

Listing 4: app/controllers/Products.php

```
class Products extends ControllerBase{
    public function caseInsensitive() {}
}
```

Urls :

```
example.com/Products/caseInsensitive (valid)
example.com/Products/caseinsensitive (valid because the method names are case_
↳ insensitive)
example.com/products/caseInsensitive (invalid since the products controller does not_
↳ exist)
```

7.5 Routing customization

The *Router* and annotations/attributes in controller classes allow you to customize URLs.

Routing can be used in addition to the default mechanism that associates `controller/action/{parameters}` with an url.

8.1 Dynamic routes

Dynamic routes are defined at runtime. It is possible to define these routes in the **app/config/services.php** file.

Important: Dynamic routes should only be used if the situation requires it:

- in the case of a micro-application
- if a route must be dynamically defined

In all other cases, it is advisable to declare the routes with annotations, to benefit from caching.

8.1.1 Callback routes

The most basic Ubiquity routes accept a Closure. In the context of micro-applications, this method avoids having to create a controller.

Listing 1: app/config/services.php

```
1 use Ubiquity\controllers\Router;
2
3 Router::get("foo", function(){
4     echo 'Hello world!';
5 });
```

Callback routes can be defined for all http methods with:

- Router::post

- Router::put
- Router::delete
- Router::patch
- Router::options

8.1.2 Controller routes

Routes can also be associated more conventionally with an action of a controller:

Listing 2: app/config/services.php

```
1 use Ubiquity\controllers\Router;
2
3 Router::addRoute('bar', \controllers\FooController::class, 'index');
```

The method `FooController::index()` will be accessible via the url `/bar`.

In this case, the **FooController** must be a class inheriting from **UbiquitycontrollersController** or one of its sub-classes, and must have an **index** method:

Listing 3: app/controllers/FooController.php

```
1 namespace controllers;
2
3 class FooController extends ControllerBase{
4
5     public function index(){
6         echo 'Hello from foo';
7     }
8 }
```

8.1.3 Default route

The default route matches the path `/`. It can be defined using the reserved path `_default`

Listing 4: app/config/services.php

```
1 use Ubiquity\controllers\Router;
2
3 Router::addRoute("_default", \controllers\FooController::class, 'bar');
```

8.2 Static routes

Static routes are defined using annotation or with php native attributes since Ubiquity 2.4.0.

Note: These annotations or attributes are never read at runtime. It is necessary to reset the router cache to take into account the changes made on the routes.

8.2.1 Creation

Attributes

Annotations

Listing 5: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 use Ubiquity\attributes\items\router\Route;
4
5 class ProductsController extends ControllerBase{
6
7     #[Route('products')]
8     public function index() {}
9
10 }
```

Listing 6: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 class ProductsController extends ControllerBase{
4
5     /**
6      * @route("products")
7      */
8     public function index() {}
9
10 }
```

The method `Products::index()` will be accessible via the url `/products`.

Note:

The initial or terminal slash is ignored in the path. The following routes are therefore equivalent:

- `#[Route('products')]`
- `#[Route('/products')]`
- `#[Route('/products/')]`

8.2.2 Route parameters

A route can have parameters:

Attributes

Annotations

Listing 7: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 use Ubiquity\attributes\items\router\Route;
```

(continues on next page)

(continued from previous page)

```

4
5 class ProductsController extends ControllerBase{
6
7     #[Route('products/{value}')]
8     public function search($value){
9         // $value will equal the dynamic part of the URL
10        // e.g. at /products/brocolis, then $value='brocolis'
11        // ...
12    }
13 }

```

Listing 8: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 class ProductsController extends ControllerBase{
4
5     /**
6     * @route("products/{value}")
7     */
8     public function search($value){
9         // $value will equal the dynamic part of the URL
10        // e.g. at /products/brocolis, then $value='brocolis'
11        // ...
12    }
13 }

```

8.2.3 Route optional parameters

A route can define optional parameters, if the associated method has optional arguments:

Attributes

Annotations

Listing 9: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 use Ubiquity\attributes\items\router\Route;
4
5 class ProductsController extends ControllerBase{
6
7     #[Route('products/all/{pageNum}/{countPerPage}')]
8     public function list($pageNum,$countPerPage=50){
9         // ...
10    }
11 }

```

Listing 10: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 class ProductsController extends ControllerBase{
4
5     ...

```

(continues on next page)

(continued from previous page)

```

5  /**
6  * @route("products/all/{pageNum}/{countPerPage}")
7  */
8  public function list($pageNum, $countPerPage=50) {
9      // ...
10 }
11 }

```

8.2.4 Route requirements

It is possible to add specifications on the variables passed in the url via the attribute **requirements**.

Attributes

Annotations

Listing 11: app/controllers/ProductsController.php

```

1  namespace controllers;
2
3  use Ubiquity\attributes\items\router\Route;
4
5  class ProductsController extends ControllerBase{
6      ...
7      #[Route('products/all/{pageNum}/{countPerPage}', requirements: ["pageNum"=>"\d+",
8      ↪ "countPerPage"=>"\d?"])]
9      public function list($pageNum, $countPerPage=50) {
10         // ...
11     }
12 }

```

Listing 12: app/controllers/ProductsController.php

```

1  namespace controllers;
2
3  class ProductsController extends ControllerBase{
4      ...
5      /**
6      * @route("products/all/{pageNum}/{countPerPage}", "requirements"=>["pageNum"=>"\d+
7      ↪ ", "countPerPage"=>"\d?"])
8      */
9      public function list($pageNum, $countPerPage=50) {
10         // ...
11     }
12 }

```

The defined route matches these urls:

- products/all/1/20
- products/all/5/

but not with that one:

- products/all/test

8.2.5 Route http methods

It is possible to specify the http method or methods associated with a route:

Attributes

Annotations

Listing 13: app/controllers/ProductsController.php

```
1 namespace controllers;
2
3 use Ubiquity\attributes\items\router\Route;
4
5 class ProductsController extends ControllerBase{
6
7     #[Route('products', methods: ['get', 'post'])]
8     public function index() {}
9
10 }
```

Listing 14: app/controllers/ProductsController.php

```
1 namespace controllers;
2
3 class ProductsController extends ControllerBase{
4
5     /**
6     * @route("products","methods"=>["get","post"])
7     */
8     public function index() {}
9
10 }
```

The **methods** attribute can accept several methods: `@route("testMethods", "methods"=>["get", "post", "delete"])` `#[Route('testMethods', methods: ['get', 'post', 'delete'])]`

The **@route** annotation or **Route** attribute defaults to all HTTP methods. There is a specific annotation for each of the existing HTTP methods:

- **@get => Get**
- **@post => Post**
- **@put => Put**
- **@patch => Patch**
- **@delete => Delete**
- **@head => Head**
- **@options => Options**

Attributes

Annotations

Listing 15: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 use Ubiquity\attributes\items\router\Get;
4
5 class ProductsController extends ControllerBase{
6
7     #[Get('products')]
8     public function index() {}
9
10 }
```

Listing 16: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 class ProductsController extends ControllerBase{
4
5     /**
6      * @get("products")
7      */
8     public function index() {}
9
10 }
```

8.2.6 Route name

It is possible to specify the **name** of a route, this name then facilitates access to the associated url. If the **name** attribute is not specified, each route has a default name, based on the pattern **controllerName_methodName**.

Attributes

Annotations

Listing 17: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 use Ubiquity\attributes\items\router\Route;
4
5 class ProductsController extends ControllerBase{
6
7     #[Route('products', name: 'products.index')]
8     public function index() {}
9
10 }
```

Listing 18: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 class ProductsController extends ControllerBase{
4
5     /**
6      * @route("products", "name"=>"products.index")
7      */
8 }
```

(continues on next page)

(continued from previous page)

```
7  */
8  public function index() {}
9
10 }
```

8.2.7 URL or path generation

Route names can be used to generate URLs or paths.

Linking to Pages in Twig

```
<a href="{{ path('products.index') }}">Products</a>
```

8.2.8 Global route

The **@route** annotation can be used on a controller class :

Attributes

Annotations

Listing 19: app/controllers/ProductsController.php

```
1 namespace controllers;
2
3 use Ubiquity\attributes\items\router\Route;
4
5 #[Route('products')]
6 class ProductsController extends ControllerBase{
7     ...
8     #[Route('/all')]
9     public function display() {}
10
11 }
```

Listing 20: app/controllers/ProductsController.php

```
1 namespace controllers;
2 /**
3  * @route("/product")
4  */
5 class ProductsController extends ControllerBase{
6     ...
7     /**
8      * @route("/all")
9      */
10     public function display() {}
11
12
13 }
```

In this case, the route defined on the controller is used as a prefix for all controller routes : The generated route for the action **display** is `/product/all`

automated routes

If a global route is defined, it is possible to add all controller actions as routes (using the global prefix), by setting the **automated** parameter :

Attributes

Annotations

Listing 21: app/controllers/ProductsController.php

```

1 namespace controllers;
2
3 use Ubiquity\attributes\items\router\Route;
4
5 #[Route('/products', automated: true)]
6 class ProductsController extends ControllerBase{
7
8     public function index() {}
9
10    public function generate() {}
11
12    public function display($id) {}
13
14 }
```

Listing 22: app/controllers/ProductsController.php

```

1 namespace controllers;
2 /**
3  * @route("/product","automated"=>true)
4  */
5 class ProductsController extends ControllerBase{
6
7     public function index() {}
8
9     public function generate() {}
10
11    public function display($id) {}
12
13 }
```

The automated attribute defines the 3 routes contained in ProductsController:

- */product/(index/)?*
- */product/generate*
- */product/display/{id}*

inherited routes

With the **inherited** attribute, it is also possible to generate the declared routes in the base classes, or to generate routes associated with base class actions if the **automated** attribute is set to true in the same time.

The base class:

Attributes

Annotations

Listing 23: app/controllers/ProductsBase.php

```
1 namespace controllers;
2
3 use Ubiquity\attributes\items\router\Route;
4
5 abstract class ProductsBase extends ControllerBase{
6
7     #[Route('(index/)?')]
8     public function index() {}
9
10    #[Route('sort/{name}')]
11    public function sortBy($name) {}
12
13 }
```

Listing 24: app/controllers/ProductsBase.php

```
1 namespace controllers;
2
3 abstract class ProductsBase extends ControllerBase{
4
5     /**
6      * @route("(index/)?")
7      */
8     public function index() {}
9
10    /**
11     * @route("sort/{name}")
12     */
13    public function sortBy($name) {}
14
15 }
```

The derived class using inherited members:

Attributes

Annotations

Listing 25: app/controllers/ProductsController.php

```
1 namespace controllers;
2
3 use Ubiquity\attributes\items\router\Route;
4
5 #[Route('/product', inherited: true)]
6 class ProductsController extends ProductsBase{
7
8     public function display() {}
9
10 }
```

Listing 26: app/controllers/ProductsController.php

```
1 namespace controllers;
2 /**
```

(continues on next page)

(continued from previous page)

```

3  * @route("/product","inherited"=>true)
4  */
5  class ProductsController extends ProductsBase{
6
7      public function display() {}
8
9  }

```

The **inherited** attribute defines the 2 routes defined in ProductsBase:

- /products/(index)?
- /products/sort/{name}

If the **automated** and **inherited** attributes are combined, the base class actions are also added to the routes.

8.2.9 Route priority

The **priority** parameter of a route allows this route to be resolved in a priority order.

The higher the priority parameter, the more the route will be defined at the beginning of the stack of routes in the cache.

In the example below, the **products/all** route will be defined before the **/products** route.

Attributes

Annotations

Listing 27: app/controllers/ProductsController.php

```

1  namespace controllers;
2
3  use Ubiquity\attributes\items\router\Route;
4
5  class ProductsController extends ControllerBase{
6
7      #[Route('products', priority: 1)]
8      public function index() {}
9
10     #[Route('products/all', priority: 10)]
11     public function all() {}
12
13 }

```

Listing 28: app/controllers/ProductsController.php

```

1  namespace controllers;
2
3  class ProductsController extends ControllerBase{
4
5      /**
6      * @route("products","priority"=>1)
7      */
8      public function index() {}
9
10     /**

```

(continues on next page)

(continued from previous page)

```
11 * @route("products/all", "priority"=>10)
12 */
13 public function all() {}
14
15 }
```

The default priority value is 0.

8.3 Routes response caching

It is possible to cache the response produced by a route:

In this case, the response is cached and is no longer dynamic.

Attributes

Annotations

```
#[Route('products/all', cache: true)]
public function all() {}
```

```
/**
 * @route("products/all", "cache"=>true)
 */
public function all() {}
```

8.3.1 Cache duration

The **duration** is expressed in seconds, if it is omitted, the duration of the cache is infinite.

Attributes

Annotations

```
#[Route('products/all', cache: true, duration: 3600)]
public function all() {}
```

```
/**
 * @route("products/all", "cache"=>true, "duration"=>3600)
 */
public function all() {}
```

8.3.2 Cache expiration

It is possible to force reloading of the response by deleting the associated cache.

```
Router::setExpired("products/all");
```

8.4 Dynamic routes caching

Dynamic routes can also be cached.

Important: This possibility is only useful if this caching is not done in production, but at the time of initialization of the cache.

```
Router::get("foo", function() {
    echo 'Hello world!';
});

Router::addRoute("string", \controllers\Main::class, "index");
CacheManager::storeDynamicRoutes(false);
```

Checking routes with devtools :

Ubiquity info:routes

• The project folder is C:\xampp7.4.4\htdocs\quick-start

path	controller	action	parameters
'/_default/'	'controllers\IndexController'	'index'	
'/string/'	≡	≡	≡
'/foo/'	(x)=>{}	' '	≡

• 3 routes (routes)

8.5 Error management (404 & 500 errors)

8.5.1 Default routing system

With the default routing system (the controller+action couple defining a route), the error handler can be redefined to customize the error management.

In the configuration file **app/config/config.php**, add the **onError** key, associated to a callback defining the error messages:

```
"onError"=>function ($code, $message = null,$controller=null) {
    switch($code) {
        case 404:
            $init=($controller==null);
            \Ubiquity\controllers\Startup::forward('IndexController/p404',$init,$init);
            break;
    }
}
```

Implement the requested action **p404** in the **IndexController**:

Listing 29: app/controllers/IndexController.php

```
...

public function p404() {
    echo "<div class='ui error message'><div class='header'>404</div>The page you are_
↳looking for doesn't exist!</div>";
}
```

8.5.2 Routage with annotations

It is enough in this case to add a last route disabling the default routing system, and corresponding to the management of the 404 error:

Attributes

Annotations

Listing 30: app/controllers/IndexController.php

```
...

#[Route('{url}', priority: -1000)]
public function p404($url) {
    echo "<div class='ui error message'><div class='header'>404</div>The page `{$url}`_
↳you are looking for doesn't exist!</div>";
}
```

Listing 31: app/controllers/IndexController.php

```
...

/**
 * @route("{url}", "priority"=>-1000)
 */
public function p404($url) {
    echo "<div class='ui error message'><div class='header'>404</div>The page `{$url}`_
↳you are looking for doesn't exist!</div>";
}
```

A controller is a PHP class inheriting from `Ubiquity\controllers\Controller`, providing an entry point in the application. Controllers and their methods define accessible URLs.

9.1 Controller creation

The easiest way to create a controller is to do it from the devtools.

From the command prompt, go to the project folder. To create the Products controller, use the command:

```
Ubiquity controller Products
```

The `Products.php` controller is created in the `app/controllers` folder of the project.

Listing 1: `app/controllers/Products.php`

```
1 namespace controllers;
2 /**
3  * Controller Products
4  */
5 class Products extends ControllerBase{
6
7     public function index() {}
8
9 }
```

It is now possible to access URLs (the `index` method is solicited by default):

```
example.com/Products
example.com/Products/index
```

Note: A controller can be created manually. In this case, he must respect the following rules:

- The class must be in the **app/controllers** folder
 - The name of the class must match the name of the php file
 - The class must inherit from **ControllerBase** and be defined in the namespace **controllers**
 - and must override the abstract **index** method
-

9.2 Methods

9.2.1 public

The second segment of the URI determines which public method in the controller gets called. The “index” method is always loaded by default if the second segment of the URI is empty.

Listing 2: app/controllers/First.php

```
1 namespace controllers;
2 class First extends ControllerBase{
3
4     public function hello(){
5         echo "Hello world!";
6     }
7
8 }
```

The hello method of the First controller makes the following URL available:

```
example.com/First/hello
```

9.2.2 method arguments

the arguments of a method must be passed in the url, except if they are optional.

Listing 3: app/controllers/First.php

```
namespace controllers;
class First extends ControllerBase{

    public function says($what,$who="world"){
        echo $what." ".$who;
    }

}
```

The hello method of the First controller makes the following URLs available:

```
example.com/First/says/hello (says hello world)
example.com/First/says/Hi/everyone (says Hi everyone)
```

9.2.3 private

Private or protected methods are not accessible from the URL.

9.3 Default controller

The default controller can be set with the Router, in the `services.php` file

Listing 4: `app/config/services.php`

```
Router::start();
Router::addRoute("_default", "controllers\First");
```

In this case, access to the `example.com/` URL loads the controller **First** and calls the default **index** method.

9.4 views loading

9.4.1 loading

Views are stored in the `app/views` folder. They are loaded from controller methods. By default, it is possible to create views in php, or with twig. **Twig** is the default template engine for html files.

php view loading

If the file extension is not specified, the **loadView** method loads a php file.

Listing 5: `app/controllers/First.php`

```
namespace controllers;
class First extends ControllerBase{
    public function displayPHP(){
        //loads the view app/views/index.php
        $this->loadView("index");
    }
}
```

twig view loading

If the file extension is html, the **loadView** method loads an html twig file.

Listing 6: `app/controllers/First.php`

```
namespace controllers;
class First extends ControllerBase{
    public function displayTwig(){
        //loads the view app/views/index.html
        $this->loadView("index.html");
    }
}
```

Default view loading

If you use the default view naming method : The default view associated to an action in a controller is located in `views/controller-name/action-name` folder:

```
views
├── Users
│   └── info.html
```

Listing 7: app/controllers/Users.php

```
1 namespace controllers;
2
3 class Users extends BaseController{
4     ...
5     public function info(){
6         $this->loadDefaultView();
7     }
8 }
9 }
```

9.4.2 view parameters

One of the missions of the controller is to pass variables to the view. This can be done at the loading of the view, with an associative array:

Listing 8: app/controllers/First.php

```
class First extends ControllerBase{
    public function displayTwigWithVar($name){
        $message="hello";
        //loads the view app/views/index.html
        $this->loadView("index.html", ["recipient"=>$name, "message"=>$message]);
    }
}
```

The keys of the associative array create variables of the same name in the view. Using of this variables in Twig:

Listing 9: app/views/index.html

```
<h1>{{message}} {{recipient}}</h1>
```

Variables can also be passed before the view is loaded:

```
//passing one variable
$this->view->setVar("title", "Message");
//passing an array of 2 variables
$this->view->setVars(["message"=>$message, "recipient"=>$name]);
//loading the view that now contains 3 variables
$this->loadView("First/index.html");
```

9.4.3 view result as string

It is possible to load a view, and to return the result in a string, assigning true to the 3rd parameter of the loadview method :

```
$viewResult=$this->loadView("First/index.html", [], true);
echo $viewResult;
```

9.4.4 multiple views loading

A controller can load multiple views:

Listing 10: app/controllers/Products.php

```
namespace controllers;
class Products extends ControllerBase{
    public function all() {
        $this->loadView("Main/header.html", ["title"=>"Products"]);
        $this->loadView("Products/index.html", ["products"=>$this->products]);
        $this->loadView("Main/footer.html");
    }
}
```

Important: A view is often partial. It is therefore important not to systematically integrate the **html** and **body** tags defining a complete html page.

9.4.5 views organization

It is advisable to organize the views into folders. The most recommended method is to create a folder per controller, and store the associated views there. To load the `index.html` view, stored in `app/views/First`:

```
$this->loadView("First/index.html");
```

9.5 initialize and finalize

The **initialize** method is automatically called before each requested action, the method **finalize** after each action.

Example of using the initialize and finalize methods with the base class automatically created with a new project:

Listing 11: app/controllers/ControllerBase.php

```
namespace controllers;

use Ubiquity\controllers\Controller;
use Ubiquity\utils\Http\Request;

/**
 * ControllerBase.
 */
abstract class ControllerBase extends Controller{
    protected $headerView = "@activeTheme/main/vHeader.html";
    protected $footerView = "@activeTheme/main/vFooter.html";

    public function initialize() {
        if (! $this->isAjax()) {
            $this->loadView($this->headerView);
        }
    }

    public function finalize() {
```

(continues on next page)

(continued from previous page)

```

        if (! URequest::isAjax ()) {
            $this->loadView ( $this->footerView );
        }
    }
}

```

9.6 Access control

Access control to a controller can be performed manually, using the *isValid* and *onInvalidControl* methods.

The *isValid* method must return a boolean which determine if access to the *action* passed as a parameter is possible:

In the following example, access to the actions of the **IndexController** controller is only possible if an **activeUser** session variable exists:

Listing 12: app/controllers/IndexController.php

```

class IndexController extends ControllerBase{
    ...
    public function isValid($action){
        return USession::exists('activeUser');
    }
}

```

If the **activeUser** variable does not exist, an **unauthorized 401** error is returned.

The *onInvalidControl* method allows you to customize the unauthorized access:

Listing 13: app/controllers/IndexController.php

```

class IndexController extends ControllerBase{
    ...
    public function isValid($action){
        return USession::exists('activeUser');
    }

    public function onInvalidControl(){
        $this->initialize();
        $this->loadView("unauthorized.html");
        $this->finalize();
    }
}

```

Listing 14: app/views/unauthorized.html

```

<div class="ui container">
    <div class="ui brown icon message">
        <i class="ui ban icon"></i>
        <div class="content">
            <div class="header">
                Error 401
            </div>
            <p>You are not authorized to access to <b>{{app.
↪getController() ~ "::~" ~ app.getAction() }}</b>.</p>
        </div>
    </div>
</div>

```

(continues on next page)

(continued from previous page)

```
</div>
</div>
```

It is also possible to automatically generate access control from *AuthControllers*

9.7 Forwarding

A redirection is not a simple call to an action of a controller. The redirection involves the *initialize* and *finalize* methods, as well as access control.

The **forward** method can be invoked without the use of the *initialize* and *finalize* methods:

It is possible to redirect to a route by its name:

9.8 Dependency injection

See *Dependency injection*

9.9 namespaces

The controller namespace is defined by default to *controllers* in the *app/config/config.php* file.

9.10 Super class

Inheritance can be used to factorize controller behavior. The *BaseController* class created with a new project is present for this purpose.

9.11 Specific controller base classes

Controller class	role
Controller	Base class for all controllers
SimpleViewController	Base class associated with a php template engine (for using with micro-services)
SimpleViewAsyncController	Base class associated with a php template engine for async servers

CHAPTER 10

Events

Note: The Events module uses the static class **EventManager** to manage events.

10.1 Framework core events

Ubiquity emits events during the different phases of submitting a request. These events are relatively few in number, to limit their impact on performance.

Part	Event name	Parameters	Occurs when
ViewEvents	BEFORE_RENDER	viewname, parameters	Before rendering a view
ViewEvents	AFTER_RENDER	viewname, parameters	After rendering a view
DAOEvents	GET_ALL	objects, classname	After loading multiple objects
DAOEvents	GET_ONE	object, classname	After loading one object
DAOEvents	UPDATE	instance, result	After updating an object
DAOEvents	INSERT	instance, result	After inserting an object

Note: There is no **BeforeAction** and **AfterAction** event, since the **initialize** and **finalize** methods of the controller class perform this operation.

10.2 Listening to an event

Example 1 :

Adding an **_updated** property on modified instances in the database :

Listing 1: app/config/services.php

```
1 use Ubiquity\events\EventsManager;
2 use Ubiquity\events\DAOEvents;
3
4 ...
5
6 EventsManager::addListener(DAOEvents::AFTER_UPDATE, function($instance,$result) {
7     if($result==1) {
8         $instance->_updated=true;
9     }
10 });
```

Note: The parameters passed to the callback function vary according to the event being listened to.

Example 2 :

Modification of the view rendering

Listing 2: app/config/services.php

```
1 use Ubiquity\events\EventsManager;
2 use Ubiquity\events\ViewEvents;
3
4 ...
5
6 EventsManager::addListener(ViewEvents::AFTER_RENDER, function(&$render,$viewname,
7     ↪$datas) {
8     $render='<h1>'.$viewname.'</h1>'.$render;
9
10 });
```

10.3 Creating your own events

Example :

Creating an event to count and store the number of displays per action :

Listing 3: app/eventListener/TracePageEventListener.php

```
1 namespace eventListener;
2
3 use Ubiquity\events\EventListenerInterface;
4 use Ubiquity\utils\base\UArray;
5
6 class TracePageEventListener implements EventListenerInterface {
7     const EVENT_NAME = 'tracePage';
8
9     public function on(&...$params) {
10         $filename = \ROOT . \DS . 'config\stats.php';
11         $stats = [ ];
12         if (file_exists ( $filename )) {
13             $stats = include $filename;
14         }
15     }
16 }
```

(continues on next page)

(continued from previous page)

```

15         $page = $params [0] . '::~' . $params [1];
16         $value = $stats [$page] ?? 0;
17         $value ++;
18         $stats [$page] = $value;
19         UArray::save ( $stats, $filename );
20     }
21 }

```

10.4 Registering events

Registering the **TracePageEventListener** event in `services.php` :

Listing 4: `app/config/services.php`

```

1  use Ubiquity\events\EventsManager;
2  use eventListener\TracePageEventListener;
3
4  ...
5
6  EventsManager::addListener(TracePageEventListener::EVENT_NAME,
↪TracePageEventListener::class);

```

10.5 Triggering events

An event can be triggered from anywhere, but it makes more sense to do it here in the **initialize** method of the base controller :

Listing 5: `app/controllers/ControllerBase.php`

```

1  namespace controllers;
2
3  use Ubiquity\controllers\Controller;
4  use Ubiquity\utils\http\URequest;
5  use Ubiquity\events\EventsManager;
6  use eventListener\TracePageEventListener;
7  use Ubiquity\controllers\Startup;
8
9  /**
10   * ControllerBase.
11   */
12  abstract class ControllerBase extends Controller{
13      protected $headerView = "@activeTheme/main/vHeader.html";
14      protected $footerView = "@activeTheme/main/vFooter.html";
15      public function initialize() {
16          $controller=Startup::getController();
17          $action=Startup::getAction();
18          EventsManager::trigger(TracePageEventListener::EVENT_NAME,
↪$controller,$action);
19          if (! URequest::isAjax ()) {
20              $this->loadView ( $this->headerView );
21          }

```

(continues on next page)

(continued from previous page)

```

22         }
23         public function finalize() {
24             if (! URequest::isAjax ()) {
25                 $this->loadView ( $this->footerView );
26             }
27         }
28     }

```

The result in app/config/stats.php :

Listing 6: app/config/stats.php

```

return array(
    "controllers\\IndexController::index"=>5,
    "controllers\\IndexController::ct"=>1,
    "controllers\\NewController::index"=>1,
    "controllers\\TestUCookieController::index"=>1
);

```

10.6 Events registering optimization

It is preferable to cache the registration of listeners, to optimize their loading time :

Create a client script, or a controller action (not accessible in production mode) :

```

use Ubiquity\\events\\EventManager;

public function initEvents() {
    EventsManager::start();
    EventsManager::addListener(DAOEvents::AFTER_UPDATE, function($instance,
    ↪ $result) {
        if($result==1) {
            $instance->_updated=true;
        }
    });
    EventsManager::addListener(TracePageEventListener::EVENT_NAME, ↪
    ↪ TracePageEventListener::class);
    EventsManager::store();
}

```

After running, cache file is generated in app/cache/events/events.cache.php.

Once the cache is created, the services.php file just needs to have the line :

```

\\Ubiquity\\events\\EventManager::start();

```

Dependency injection

Note: For performance reasons, dependency injection is not used in the core part of the framework.

Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on it.

Note: Ubiquity only supports property injection, so as not to require introspection at execution. Only controllers support dependency injection.

11.1 Service autowiring

11.1.1 Service creation

Create a service

Listing 1: app/services/Service.php

```
1 namespace services;
2
3 class Service{
4     public function __construct($ctrl){
5         echo 'Service instantiation in '.get_class($ctrl);
6     }
7
8     public function do($someThink=""){
9         echo 'do '.$someThink ."in service";
10    }
11 }
```

11.1.2 Autowiring in Controller

Create a controller that requires the service

Listing 2: app/services/Service.php

```
1 namespace controllers;
2
3 /**
4  * Controller Client
5  */
6 class ClientController extends ControllerBase{
7
8     /**
9      * @autowired
10     * @var services\Service
11     */
12     private $service;
13
14     public function index() {}
15
16     /**
17     * @param \services\Service $service
18     */
19     public function setService($service) {
20         $this->service = $service;
21     }
22 }
```

In the above example, Ubiquity looks for and injects **\$service** when **ClientController** is created.

The @autowired annotation requires that:

- the type to be instantiated is declared with the **@var** annotation
- **\$service** property has a setter, or whether declared public

As the annotations are never read at runtime, it is necessary to generate the cache of the controllers:

```
Ubiquity init-cache -t=controllers
```

It remains to check that the service is injected by going to the address `/ClientController`.

11.2 Service injection

11.2.1 Service

Let's now create a second service, requiring a special initialization.

Listing 3: app/services/ServiceWithInit.php

```
1 class ServiceWithInit{
2     private $init;
3
4     public function init(){
5         $this->init=true;
6     }
7 }
```

(continues on next page)

(continued from previous page)

```

6         }
7
8         public function do() {
9             if($this->init) {
10                 echo 'init well initialized!';
11             } else {
12                 echo 'Service not initialized';
13             }
14         }
15     }

```

11.2.2 Injection in controller

Listing 4: app/controllers/ClientController.php

```

1 namespace controllers;
2
3 /**
4  * Controller Client
5  */
6 class ClientController extends ControllerBase{
7
8     /**
9      * @autowired
10     * @var \services\Service
11     */
12     private $service;
13
14     /**
15     * @injected
16     */
17     private $serviceToInit;
18
19     public function index() {
20         $this->serviceToInit->do();
21     }
22
23     /**
24     * @param \services\Service $service
25     */
26     public function setService($service) {
27         $this->service = $service;
28     }
29
30     /**
31     * @param mixed $serviceToInit
32     */
33     public function setServiceToInit($serviceToInit) {
34         $this->serviceToInit = $serviceToInit;
35     }
36
37 }

```

11.2.3 Di declaration

In `app/config/config.php`, create a new key for **serviceToInit** property to inject in **di** part.

```
"di"=>[ "ClientController.serviceToInit"=>function() {  
    $service=new \services\ServiceWithInit();  
    $service->init();  
    return $service;  
}]
```

generate the cache of the controllers:

```
Ubiquity init-cache -t=controllers
```

Check that the service is injected by going to the address `/ClientController`.

Note: If the same service is to be used in several controllers, use the wildcard notation :

```
"di"=>[ "*.serviceToInit"=>function() {  
    $service=new \services\ServiceWithInit();  
    $service->init();  
    return $service;  
}]
```

11.2.4 Injection with a qualifier name

If the name of the service to be injected is different from the key of the **di** array, it is possible to use the name attribute of the **@injected** annotation

In `app/config/config.php`, create a new key for **serviceToInit** property to inject in **di** part.

```
"di"=>[ "*.service"=>function() {  
    $service=new \services\ServiceWithInit();  
    $service->init();  
    return $service;  
}]
```

```
/**  
 * @injected("service")  
 */  
private $serviceToInit;
```

11.3 Service injection at runtime

It is possible to inject services at runtime, without these having been previously declared in the controller classes.

Listing 5: app/services/RuntimeService.php

```

1 namespace services;
2
3 class RuntimeService{
4     public function __construct($ctrl){
5         echo 'Service instantiation in ' .get_class($ctrl);
6     }
7 }

```

In app/config/config.php, create the **@exec** key in **di** part.

```

"di"=>["@exec"=>"rService"=>function($ctrl){
    return new \services\RuntimeService($ctrl);
}
]

```

With this declaration, the **\$rService** member, instance of **RuntimeService**, is injected into all the controllers. It is then advisable to use the javadoc comments to declare **\$rService** in the controllers that use it (to get the code completion on **\$rService** in your IDE).

Listing 6: app/controllers/MyController.php

```

1 namespace controllers;
2
3 /**
4  * Controller Client
5  * property services\RuntimeService $rService
6  */
7 class MyController extends ControllerBase{
8
9     public function index(){
10         $this->rService->do();
11     }
12 }

```


The **CRUD** controllers allow you to perform basic operations on a **Model** class:

- Create
- Read
- Update
- Delete
- ...

12.1 Creation

In the admin interface (web-tools), activate the **Controllers** part, and choose create **Crud controller**:



+ Create special controller

Then fill in the form:

- Enter the controller name
- Select the associated model
- Then click on the validate button

Adding a CRUD controller

Name

controllers\ UsersController

Model

models\User

☐ Create override Datas class

☐ Create override Events class

☐ Add route...

☐ Create override ModelViewer class

☐ Create override CRUDFiles class (URLs and files)

12.2 Description of the features

The generated controller:

Listing 1: app/controllers/Products.php

```

1 <?php
2 namespace controllers;
3
4 /**
5  * CRUD Controller UsersController
6  */
7 class UsersController extends \Ubiquity\controllers\crud\CRUDController{
8
9     public function __construct() {
10         parent::__construct();
11         $this->model="models\\User";
12     }
13
14     public function _getBaseRoute() {
15         return 'UsersController';
16     }
17 }

```

Test the created controller by clicking on the get button in front of the **index** action:

⚡ index()

+ Create view UsersController/index.html

GET

POST

▼

12.2.1 Read (index action)

GET:UsersController/index

Add a new models\User...

Id	Name	Email	Password	
1	Henry Zhu	henry.zhu@gmail.com	****	<div><div></div><div></div></div>
2	Evan YOU	evan.you@vuejs.org	****	<div><div></div><div></div></div>
3	Fabien POTENCIER	fab.potencier@symfony.fr	***	<div><div></div><div></div></div>







Search...

Close

Clicking on a row of the dataTable (instance) displays the objects associated to the instance (**details** action):

GET:UsersController/index

+ Add a new models\User...

Id	Name	Email	Password	
1	Henry Zhu	henry.zhu@gmail.com	****	 
2	Evan YOU	evan.you@vuejs.org	****	 
3	Fabien POTENCIER	fab.potencier@symfony.fr	***	 

Search...

estimations (0)

projects (1)

VueJS

participations (3)

Paris-h2



VueJS

Sudoku

Close

Using the search area:

+ Add a new models\User...

Id	Name	Email	Password	
3	Fabien POTENCIER	fab.potencier@symfony.fr	***	 

fab ✕


Search...


12.2.2 Create (newModel action)

It is possible to create an instance by clicking on the add button

+ Add a new models\User...

The default form for adding an instance of User:

 Add a new models\User...

 models\User
+ New object creation

Id

Name

Email

Password


ParticipationsIds

12.2.3 Update (update action)

The edit button on each row allows you to edit an instance



The default form for adding an instance of User:

 models\User
Editing an existing object

Id

2

Name

Evan YOU

Email

evan.you@vuejs.org

Password

••••

ParticipationsIds

Paris-h2 ✕ VueJS ✕ Sudoku ✕

12.2.4 Delete (delete action)

The delete button on each row allows you to edit an instance



Display of the confirmation message before deletion:

+ Add a new models\User...

Id	Name	Email	Password	
1	Henry Zhu	henry.zhu@gmail.com	****	<div style="display: inline-block; border: 1px solid #ccc; border-radius: 50%; width: 25px; height: 25px; text-align: center; line-height: 25px;">✎</div> <div style="display: inline-block; border: 1px solid #ccc; border-radius: 50%; width: 25px; height: 25px; text-align: center; line-height: 25px; margin-left: 5px;">✖</div>
2	Evan YOU	evan.you@vuejs.org	****	<div style="display: inline-block; border: 1px solid #ccc; border-radius: 50%; width: 25px; height: 25px; text-align: center; line-height: 25px;">✎</div> <div style="display: inline-block; border: 1px solid #ccc; border-radius: 50%; width: 25px; height: 25px; text-align: center; line-height: 25px; margin-left: 5px;">✖</div>
3	Fabien POTENCIER	fab.potencier@symfony.fr	***	<div style="display: inline-block; border: 1px solid #ccc; border-radius: 50%; width: 25px; height: 25px; text-align: center; line-height: 25px;">✎</div> <div style="display: inline-block; border: 1px solid #ccc; border-radius: 50%; width: 25px; height: 25px; text-align: center; line-height: 25px; margin-left: 5px;">✖</div>

?

Remove confirmation

Do you confirm the deletion of ``evan.you@vuejs.org``?

✕

○ Cancel

✔ Confirm

12.3 Customization

Create again a CrudController from the admin interface:

Adding a CRUD controller

Name

controllers\ UsersController

☐ Create override Datas class

☐ Create override Events class

Model

models\User

☐ Create override ModelViewer class

☐ Create override CRUDFiles class (URLs and files)

@framework/crud/index.html ✕
@framework/crud/form.html ✕

@framework/crud/display.html ✕

☒ Add route...

Path

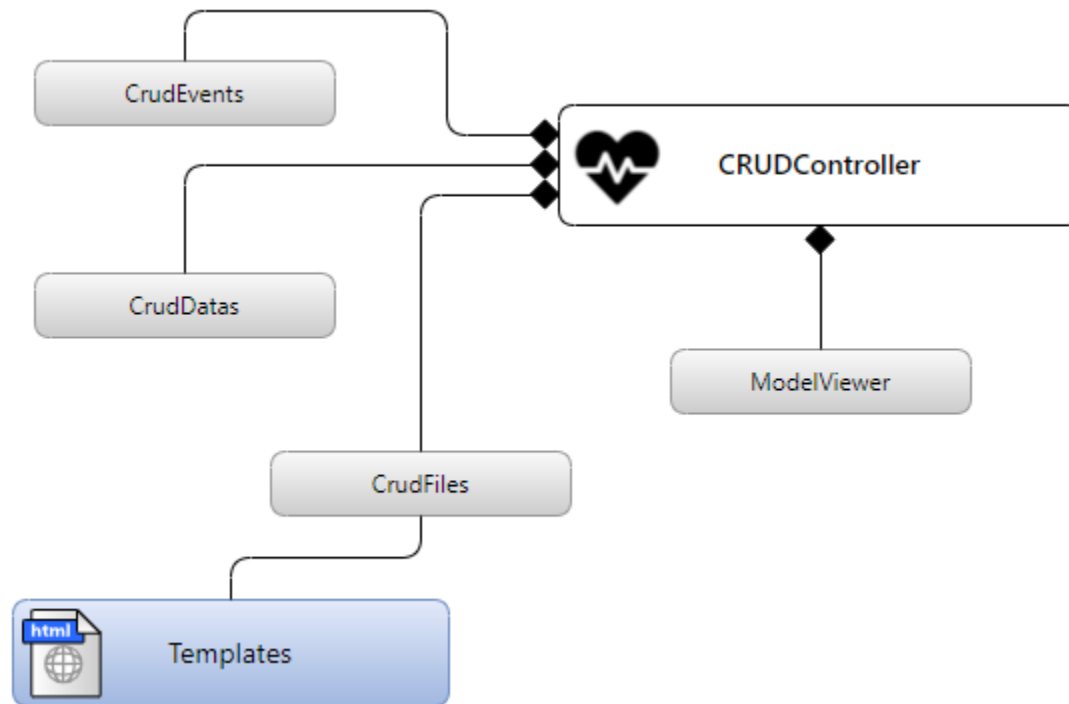
users

✔ Validate

○ Cancel

It is now possible to customize the module using overriding.

12.3.1 Overview



12.3.2 Classes overriding

CRUDController methods to override

Method	Signification	Default return
routes		
index()	Default page : list all objects	
edit(\$modal="no", \$ids="")	Edits an instance	
newModel(\$modal="no")	Creates a new instance	
display(\$modal="no", \$ids="")	Displays an instance	
delete(\$ids)	Deletes an instance	
update()	Displays the result of an instance updating	
showDetail(\$ids)	Displays associated members with foreign keys	
refresh_()	Refreshes the area corresponding to the DataTable (#lv)	
refreshTable(\$id=null)	//TO COMMENT	

ModelViewer methods to override

Method	Signification	Default return
index route		
getModelDataTable(\$instances, \$model, \$totalCount, \$page=1)	Creates the dataTable and Adds its behavior	DataTable
getDataTableInstance(\$instances, \$model, \$totalCount, \$page=1)	Creates the dataTable	DataTable
recordsPerPage(\$model, \$totalCount=0)	Returns the count of rows to display (if null there's no pagination)	null or 6
getGroupByFields()	Returns an array of members on which to perform a grouping	[]
getDataTableRowButtons()	Returns an array of buttons to display for each row ["edit","delete","display"]	["edit","delete"]
onDataTableRowButton(HtmlButton \$bt)	To override for modifying the dataTable row buttons	
getCaptions(\$captions, \$className)	Returns the captions of the column headers	all member names
detail route		
showDetailsOnDataTableClick()	To override to make sure that the detail of a clicked object is displayed or not	true
onDisplayFkElementListDetails(\$element, \$member, \$className, \$object)	To modify for displaying each element in a list component of foreign objects	
getFkHeaderElementDetails(\$member, \$className, \$object)	Returns the header for a single foreign object (issue from ManyToOne)	Html-Header
getFkElementDetails(\$member, \$className, \$object)	Returns a component for displaying a single foreign object (manyToOne relation)	HtmlLabel
getFkHeaderListDetails(\$member, \$className, \$list)	Returns the header for a list of foreign objects (oneToMany or ManyToMany)	Html-Header
getFkListDetails(\$member, \$className, \$list)	Returns a list component for displaying a collection of foreign objects (many)	HtmlList
edit and newModel routes		
getForm(\$identifier, \$instance)	Returns the form for adding or modifying an object	Html-Form
getFormTitle(\$form, \$instance)	Returns an associative array defining form message title with keys "icon","message","subMessage"	Html-Form
setFormFieldsComponent(DataForm \$form, \$fieldTypes)	Sets the components for each field	
onGenerateFormField(\$field)	For doing something when \$field is generated in form	
isModal(\$objects, \$model)	Condition to determine if the edit or add form is modal for \$model objects	count(\$objects)>5
getFormCaptions(\$captions, \$className, \$instance)	Returns the captions for form fields	all member names
display route		
getModelDataElement(\$instance, \$model, \$modal)	Returns a DataElement object for displaying the instance	DataElement
getElementCaptions(\$captions, \$className, \$instance)	Returns the captions for DataElement fields	all member names
delete route		
onConfirmButtons(HtmlButton \$confirmBtn, HtmlButton \$cancelBtn)	To override for modifying delete confirmation buttons	

CRUDDatas methods to override

Method	Signification	Default return
index route		
<code>_getInstancesFilter(\$model)</code>	Adds a condition for filtering the instances displayed in <code>dataTable</code>	1=1
<code>getFieldNames(\$model)</code>	Returns the fields to display in the index action for <code>\$model</code>	all member names
<code>getSearchFieldNames(\$model)</code>	Returns the fields to use in search queries	all member names
edit and newModel routes		
<code>getFormFieldNames(\$model,\$instance)</code>	Returns the fields to update in the edit and newModel actions for <code>\$model</code>	all member names
<code>getManyToOneDatas(\$fkClass,\$instance,\$member)</code>	Returns a list (filtered) of <code>\$fkClass</code> objects to display in an html list	all <code>\$fkClass</code> instances
<code>getOneToManyDatas(\$fkClass,\$instance,\$member)</code>	Returns a list (filtered) of <code>\$fkClass</code> objects to display in an html list	all <code>\$fkClass</code> instances
<code>getManyToManyDatas(\$fkClass,\$instance,\$member)</code>	Returns a list (filtered) of <code>\$fkClass</code> objects to display in an html list	all <code>\$fkClass</code> instances
display route		
<code>getElementFieldNames(\$model)</code>	Returns the fields to display in the display action for <code>\$model</code>	all member names

CRUDEvents methods to override

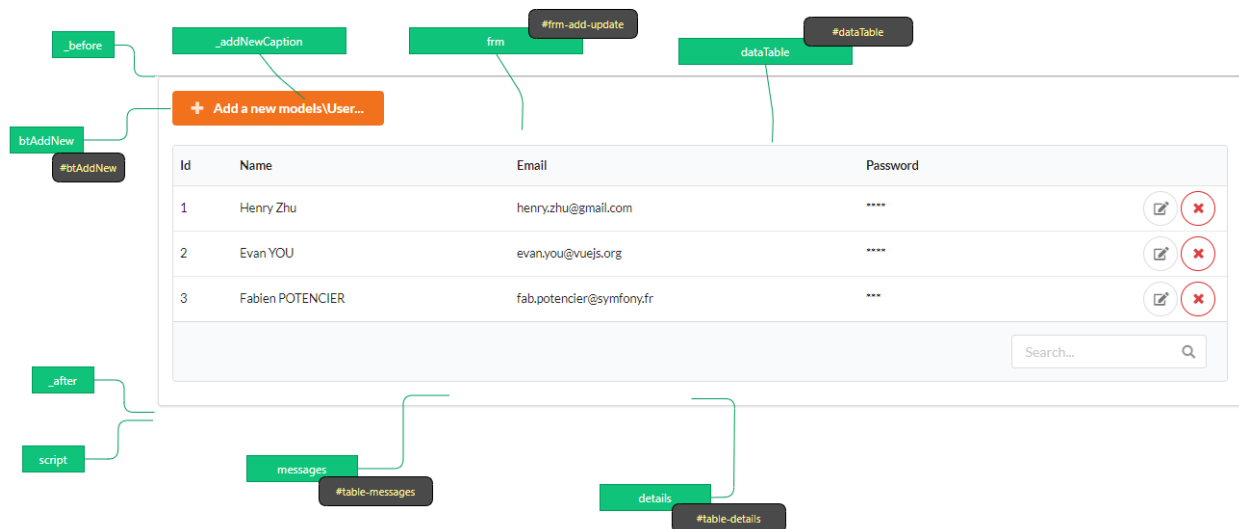
Method	Signification	Default return
index route		
<code>onConfDeleteMessage(CRUDMessage \$message,\$instance)</code>	Returns the confirmation message displayed before deleting an instance	CRUDMessage
<code>onSuccessDeleteMessage(CRUDMessage \$message,\$instance)</code>	RReturns the message displayed after a deletion	CRUDMessage
<code>onErrorDeleteMessage(CRUDMessage \$message,\$instance)</code>	Returns the message displayed when an error occurred when deleting	CRUDMessage
edit and newModel routes		
<code>onSuccessUpdateMessage(CRUDMessage \$message)</code>	Returns the message displayed when an instance is added or inserted	CRUDMessage
<code>onErrorUpdateMessage(CRUDMessage \$message)</code>	Returns the message displayed when an error occurred when updating or inserting	CRUDMessage
all routes		
<code>onNotFoundMessage(CRUDMessage \$message,\$ids)</code>	Returns the message displayed when an instance does not exists	
<code>onDisplayElements(\$dataTable,\$objects,\$refresh)</code>	Triggered after displaying objects in <code>dataTable</code>	

CRUDFiles methods to override

Method	Signification	Default return
template files		
getViewBaseTemplate()	Returns the base template for all Crud actions if getBaseTemplate return a base template filename	@framework/crud/baseTemplate.html
getViewIndex()	Returns the template for the index route	@framework/crud/index.html
getViewForm()	Returns the template for the edit and newInstance routes	@framework/crud/form.html
getViewDisplay()	Returns the template for the display route	@framework/crud/display.html
Urls		
getRouteRefresh()	Returns the route for refreshing the index route	/refresh_
getRouteDetails()	Returns the route for the detail route, when the user click on a dataTable row	/showDetail
getRouteDelete()	Returns the route for deleting an instance	/delete
getRouteEdit()	Returns the route for editing an instance	/edit
getRouteDisplay()	Returns the route for displaying an instance	/display
getRouteRefreshTable()	Returns the route for refreshing the dataTable	/refreshTable
getDetailClickURL(\$model)	Returns the route associated with a foreign key instance in list	""

12.3.3 Twig Templates structure

index.html



form.html

Displayed in **frm** block

models\User
Editing an existing object

Id
1

Name
Henry Zhu

Email
henry.zhu@gmail.com

Password
....

ParticipationsIds
Paris-h2 x VueJS x ScrumPoker x Sudoku x

Buttons:

Labels: #action-modal-frmEdit-0, #bt-cancel

display.html

Displayed in **frm** block

Buttons: + Add a new models\User..., x Close, x Delete evan.you@vuejs.org..., Edit evan.you@vuejs.org...

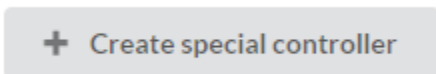
Id	2
Name	Evan YOU
Email	evan.you@vuejs.org
Password	evan
Estimations	
Participations	Paris-h2 VueJS Sudoku
Projects	VueJS

The Auth controllers allow you to perform basic authentication with:

- login with an account
- account creation
- logout
- controllers with required authentication

13.1 Creation

In the admin interface (web-tools), activate the **Controllers** part, and choose create **Auth controller**:



Then fill in the form:

- Enter the controller name (BaseAuthController in this case)

Adding an Auth controller

Name

controllers\ BaseAuthController

Base class

Ubiquity\controllers\auth\AuthController ▼

☐ Create override AuthFiles class

☐ Add route...

The generated controller:

Listing 1: app/controllers/BaseAuthController.php

```

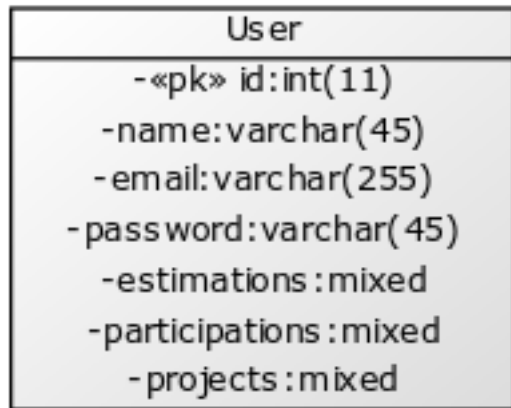
1  /**
2   * Auth Controller BaseAuthController
3   */
4  class BaseAuthController extends \Ubiquity\controllers\auth\AuthController{
5
6      protected function onConnect($connected) {
7          $urlParts=$this->getOriginalURL();
8          USession::set($this->_getUserSessionKey(), $connected);
9          if(isset($urlParts)){
10             Startup::forward(implode("/", $urlParts));
11          }else{
12             //TODO
13             //Forwarding to the default controller/action
14          }
15      }
16
17      protected function _connect() {
18          if(URequest::isPost()){
19              $email=URequest::post($this->_getLoginInputName());
20              $password=URequest::post($this->_getPasswordInputName());
21              //TODO
22              //Loading from the database the user corresponding to the
23              ↪parameters
24              //Checking user credentials
25              //Returning the user
26          }
27          return;
28      }
29
30      /**
31       * {@inheritdoc}
32       * @see \Ubiquity\controllers\auth\AuthController::isValidUser()
33       */
34      public function _isValidUser($action=null) {
35          return USession::exists($this->_getUserSessionKey());
36      }
37
38      public function _getBaseRoute() {
39          return 'BaseAuthController';
40      }
41  }

```

13.2 Implementation of the authentication

Example of implementation with the administration interface : We will add an authentication check on the admin interface.

Authentication is based on verification of the email/password pair of a model **User**:



13.2.1 BaseAuthController modification

Listing 2: app/controllers/BaseAuthController.php

```

1  /**
2   * Auth Controller BaseAuthController
3   */
4  class BaseAuthController extends \Ubiquity\controllers\auth\AuthController{
5
6      protected function onConnect($connected) {
7          $urlParts=$this->getOriginalURL();
8          USession::set($this->_getUserSessionKey(), $connected);
9          if(isset($urlParts)){
10              Startup::forward(implode("/", $urlParts));
11          }else{
12              Startup::forward("admin");
13          }
14      }
15
16      protected function _connect() {
17          if(URequest::isPost()){
18              $email=URequest::post($this->_getLoginInputName());
19              $password=URequest::post($this->_getPasswordInputName());
20              return DAO::uGetOne(User::class, "email=? and password= ?",false,
21                  ↪[$email,$password]);
22          }
23          return;
24      }
25
26      /**
27       * {@inheritdoc}
28       * @see \Ubiquity\controllers\auth\AuthController::isValidUser()
29       */
30      public function _isValidUser($action=null) {
31          return USession::exists($this->_getUserSessionKey());
32      }
33
34      public function _getBaseRoute() {
35          return 'BaseAuthController';
36      }
37  }

```

(continues on next page)

(continued from previous page)

```

37      * {@inheritdoc}
38      * @see \Ubiquity\controllers\auth\AuthController::_getLoginInputName()
39      */
40      public function _getLoginInputName() {
41          return "email";
42      }
43  }

```

13.2.2 Admin controller modification

Modify the Admin Controller to use BaseAuthController:


Listing 3: app/controllers/Admin.php

```

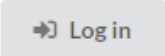
1  class Admin extends UbiquityMyAdminBaseController{
2      use WithAuthTrait;
3      protected function getAuthController(): AuthController {
4          return new BaseAuthController();
5      }
6  }

```

Test the administration interface at **/admin**:



Forbidden access
You are not authorized to access the page Admin !



After clicking on **login**:


Connection

Email *


Password *

☐ Remember me

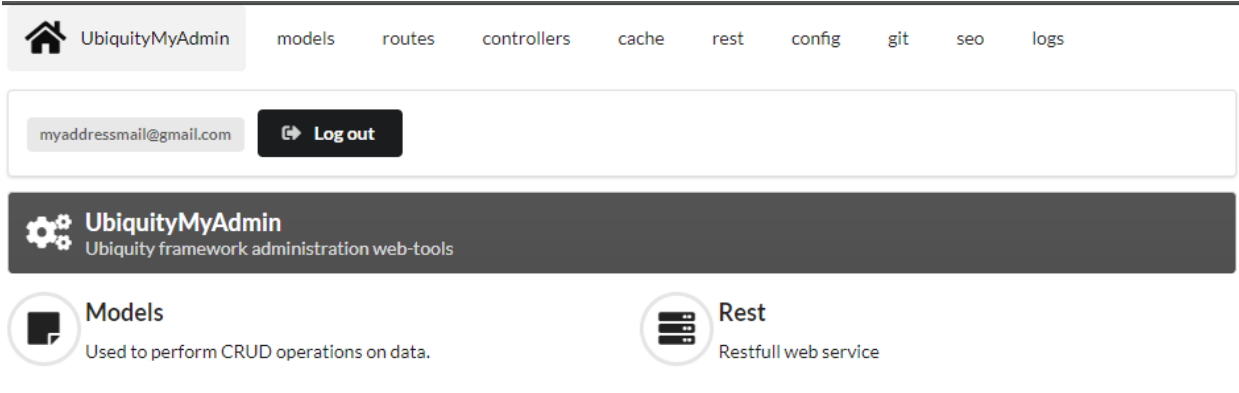
If the authentication data entered is invalid:



Connection problem
Invalid credentials!



If the authentication data entered is valid:



13.2.3 Attaching the zone info-user

Modify the **BaseAuthController** controller:

Listing 4: app/controllers/BaseAuthController.php

```

1  /**
2   * Auth Controller BaseAuthController
3   */
4  class BaseAuthController extends \Ubiquity\controllers\auth\AuthController{
5      ...
6      public function _displayInfoAsString() {
7          return true;
8      }
9  }
```

The **_userInfo** area is now present on every page of the administration:



It can be displayed in any twig template:

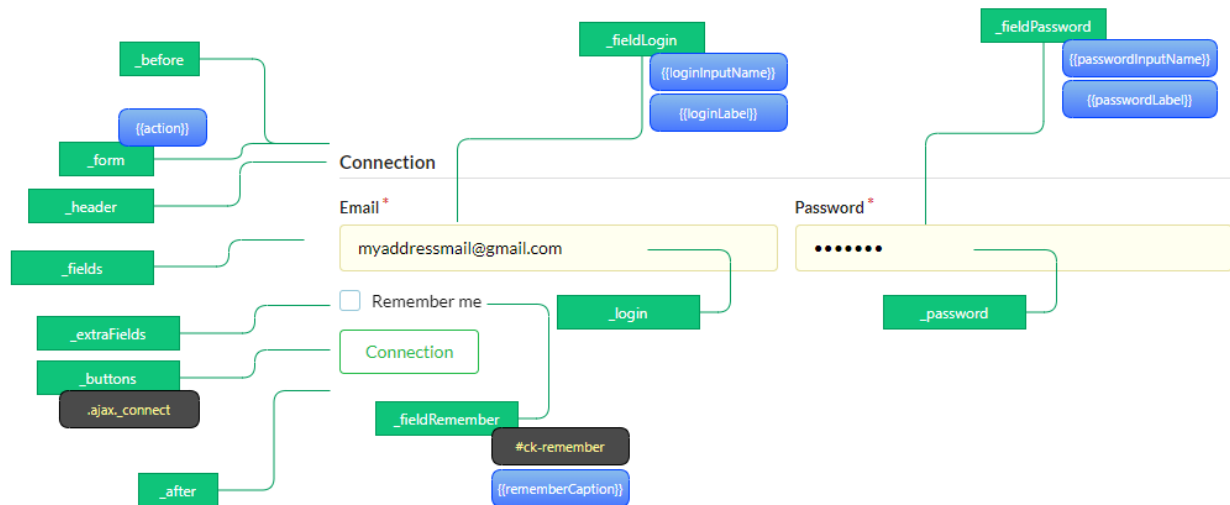
```
{{ _userInfo | raw }}
```

13.3 Description of the features

13.3.1 Customizing templates

index.html template

The index.html template manages the connection:



Example with the `_userInfo` aera:

Create a new AuthController named **PersoAuthController**:

Adding an Auth controller

Name

controllers\
 PersoAuthController

Base class

controllers\BaseAuthController

☐ Create override AuthFiles class

@framework/auth/info.html

☐ Add route...

Edit the template `app/views/PersoAuthController/info.html`

Listing 5: `app/views/PersoAuthController/info.html`

```

1  {% extends "@framework/auth/info.html" %}
2  {% block _before %}
3      <div class="ui tertiary inverted red segment">
4  {% endblock %}
5  {% block _userInfo %}
6      {{ parent() }}
7  {% endblock %}
8  {% block _logoutButton %}
9      {{ parent() }}
10 {% endblock %}
11 {% block _logoutCaption %}
12     {{ parent() }}
13 {% endblock %}
14 {% block _loginButton %}
15     {{ parent() }}
16 {% endblock %}
17 {% block _loginCaption %}
18     {{ parent() }}

```

(continues on next page)

(continued from previous page)

```

19 {% endblock %}
20 {% block _after %}
21     </div>
22 {% endblock %}

```

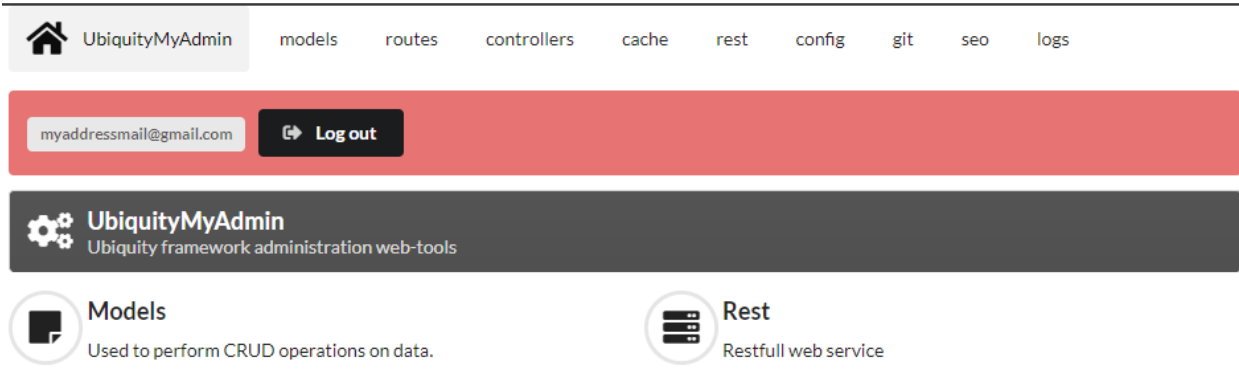
Change the AuthController **Admin** controller:

Listing 6: app/controllers/Admin.php

```

1 class Admin extends UbiquityMyAdminController{
2     use WithAuthTrait;
3     protected function getAuthController(): AuthController {
4         return new PersoAuthController();
5     }
6 }

```



13.3.2 Customizing messages

Listing 7: app/controllers/PersoAuthController.php

```

1 class PersoAuthController extends \controllers\BaseAuth{
2     ...
3     /**
4      * {@inheritdoc}
5      * @see \Ubiquity\controllers\auth\AuthController::badLoginMessage()
6      */
7     protected function badLoginMessage(\Ubiquity\utils\flash\FlashMessage $fMessage) {
8         $fMessage->setTitle("Erreur d'authentification");
9         $fMessage->setContent("Login ou mot de passe incorrects !");
10        $this->_setLoginCaption("Essayer à nouveau");
11    }
12 }
13 ...
14 }

```

13.3.3 Self-check connection

Listing 8: app/controllers/PersoAuthController.php

```
1 class PersoAuthController extends \controllers\BaseAuth{
2     ...
3     /**
4      * {@inheritdoc}
5      * @see \Ubiquity\controllers\auth\AuthController::_checkConnectionTimeout()
6      */
7     public function _checkConnectionTimeout() {
8         return 10000;
9     }
10    ...
11 }
```

13.3.4 Limitation of connection attempts

Listing 9: app/controllers/PersoAuthController.php

```
1 class PersoAuthController extends \controllers\BaseAuth{
2     ...
3     /**
4      * {@inheritdoc}
5      * @see \Ubiquity\controllers\auth\AuthController::attemptsNumber()
6      */
7     protected function attemptsNumber() {
8         return 3;
9     }
10    ...
11 }
```

The **DAO** class is responsible for loading and persistence operations on models :

14.1 Connecting to the database

Check that the database connection parameters are correctly entered in the configuration file:

```
Ubiquity config -f=database
```

• Displaying config variables from `app/config/config.php` file

field	value
database	<ul style="list-style-type: none">• type : 'mysql'• dbName : 'messengerie'• serverName : '127.0.0.1'• port : 3306• user : 'root'• password : ''• options : []• cache : false• wrapper : 'Ubiquity\\db\\providers\\pdo\\PDOWrapper'

14.1.1 Transparent connection

Since Ubiquity 2.3.0, The connection to the database is done automatically the first time you request it:

```
use Ubiquity\orm\DAO;  
  
$firstUser=DAO::getById(User::class,1);//Automatically start the database
```

This is the case for all methods in the **DAO** class used to perform CRUD operations.

14.1.2 Explicit connection

In some cases, however, it may be useful to make an explicit connection to the database, especially to check the connection.

```
use Ubiquity\orm\DAO;
use Ubiquity\controllers\Startup;
...
try{
    $config=\Ubiquity\controllers\Startup::getConfig();
    DAO::startDatabase($config);
    $users=DAO::getAll(User::class, '');
}catch(Exception $e){
    echo $e->getMessage();
}
```

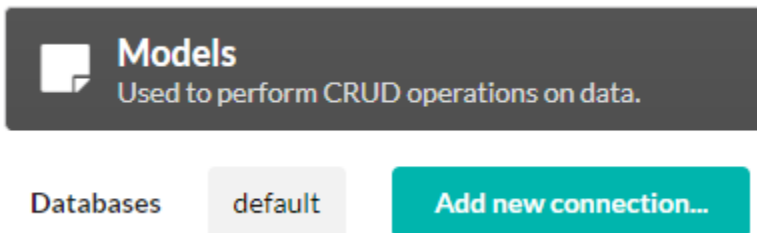
14.2 Multiple connections

14.2.1 Adding a new connection

Ubiquity allows you to manage several connections to databases.

With Webtools

In the **Models** part, choose **Add new connection** button:



Define the connection configuration parameters:

Adding a new DB connection

Connection name

Provider	<input type="text" value="pdo"/>	
Type	<input type="text" value="mysql"/>	
dbName	<input checked="" type="checkbox"/> utest	<input type="button" value="Test"/>
serverName	<input type="text" value="127.0.0.1"/> <input type="text" value="127.0.0.1"/>	
port	<input type="text" value="3306"/>	
user	<input type="text" value="root"/> <input type="text" value="root"/>	
password	<input type="text" value="Database-password"/>	
options	<input type="text" value="array()"/> <input type="text" value="array()"/>	
cache	<input type="checkbox"/> Database-cache	

Generate models for the new connection: The generated models include the @database annotation or the Database attribute mentioning their link to the connection.

Attributes

Annotations

```
<?php
namespace models\tests;
use Ubiquity\attributes\items\Database;
use Ubiquity\attributes\items\Table;

#[Database('tests')]
#[Table('groupe')]
class Groupe{
    * * *
}
```

```
<?php
namespace models\tests;
/**
 * @database('tests')
 * @table('groupe')
 */
class Groupe{
    * * *
}
```

Models are generated in a sub-folder of models.

With several connections, do not forget to add the following line to the `services.php` file:

```
\Ubiquity\orm\DAO::start();
```

The `start` method performs the match between each model and its associated connection.

15.1 From existing database

- with console
- with web-tools

Note: if you want to automatically generate the models, consult the *generating models* part.

A model class is just a plain old php object without inheritance. Models are located by default in the **app\models** folder. Object Relational Mapping (ORM) relies on member annotations in the model class.

16.1 Models definition

16.1.1 A basic model

- A model must define its primary key using the **@id** annotation on the members concerned
- Serialized members must have getters and setters
- Without any other annotation, a class corresponds to a table with the same name in the database, each member corresponds to a field of this table

Listing 1: app/models/User.php

```
1 namespace models;
2 class User{
3     /**
4      * @id
5      */
6     private $id;
7
8     private $firstname;
9
10    public function getFirstname(){
11        return $this->firstname;
12    }
```

(continues on next page)

(continued from previous page)

```
13     public function setFirstname($firstname) {
14         $this->firstname=$firstname;
15     }
16 }
```

16.1.2 Mapping

Table->Class

If the name of the table is different from the name of the class, the annotation **@table** allows to specify the name of the table.

Listing 2: app/models/User.php

```
1  namespace models;
2
3  /**
4   * @table("name"=>"user")
5   */
6  class User{
7      /**
8       * @id
9       */
10     private $id;
11
12     private $firstname;
13
14     public function getFirstname() {
15         return $this->firstname;
16     }
17     public function setFirstname($firstname) {
18         $this->firstname=$firstname;
19     }
20 }
```

Field->Member

If the name of a field is different from the name of a member in the class, the annotation **@column** allows to specify a different field name.

Listing 3: app/models/User.php

```
1  namespace models;
2
3  /**
4   * @table("user")
5   */
6  class User{
7      /**
8       * @id
9       */
10     private $id;
11 }
```

(continues on next page)

(continued from previous page)

```

12  /**
13  * column("user_name")
14  **/
15  private $firstname;
16
17  public function getFirstname() {
18      return $this->firstname;
19  }
20  public function setFirstname($firstname) {
21      $this->firstname=$firstname;
22  }
23  }

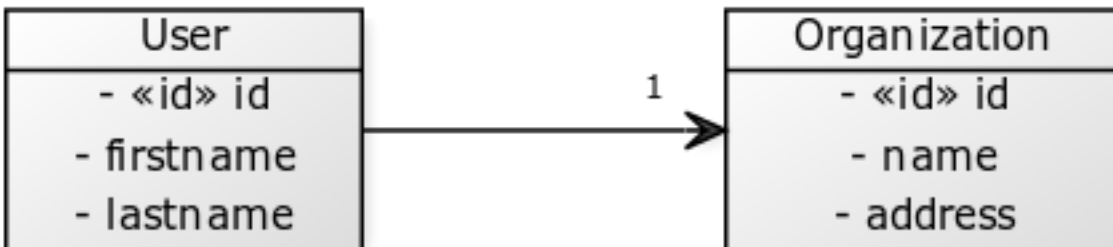
```

16.1.3 Associations

Note: Naming convention Foreign key field names consist of the primary key name of the referenced table followed by the name of the referenced table whose first letter is capitalized. **Example** idUser for the table user whose primary key is id

ManyToOne

A **user** belongs to an **organization**:



Listing 4: app/models/User.php

```

1  namespace models;
2
3  class User{
4      /**
5       * @id
6       **/
7      private $id;
8
9      private $firstname;
10
11      /**
12       * @ManyToOne
13       * @joinColumn("className"=>"models\\Organization", "name"=>
14       ↪ "idOrganization", "nullable"=>false)
15       **/
16      private $organization;

```

(continues on next page)

(continued from previous page)

```

16
17     public function getOrganization(){
18         return $this->organization;
19     }
20
21     public function setOrganization($organization){
22         $this->organization=$organization;
23     }
24 }

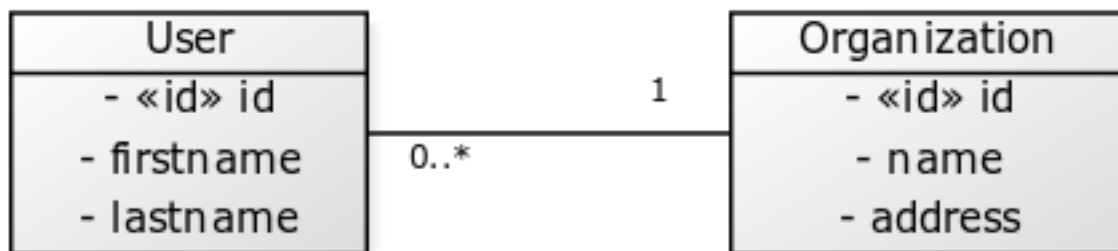
```

The **@joinColumn** annotation specifies that:

- The member **\$organization** is an instance of **modelsOrganization**
- The table **user** has a foreign key **idOrganization** referring to organization primary key
- This foreign key is not null => a user will always have an organization

OneToMany

An **organization** has many **users**:



Listing 5: app/models/Organization.php

```

1     namespace models;
2
3     class Organization{
4         /**
5          * @id
6          */
7         private $id;
8
9         private $name;
10
11         /**
12          * @OneToMany("mappedBy"=>"organization", "className"=>"models\\User")
13          */
14         private $users;
15     }

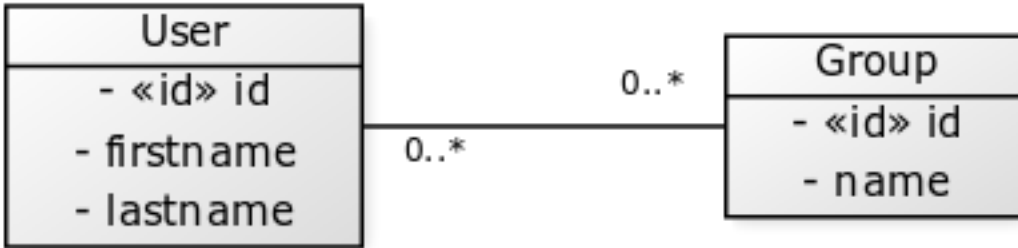
```

In this case, the association is bi-directional. The **@oneToMany** annotation must just specify:

- The class of each user in users array : **modelsUser**
- the value of **@mappedBy** is the name of the association-mapping attribute on the owning side : **\$organization** in **User** class

ManyToMany

- A **user** can belong to **groups**.
- A **group** consists of multiple **users**.



Listing 6: app/models/User.php

```

1 namespace models;
2
3 class User{
4     /**
5      * @id
6      */
7     private $id;
8
9     private $firstname;
10
11     /**
12      * @manyToMany("targetEntity"=>"models\\Group", "inversedBy"=>"users")
13      * @joinTable("name"=>"groupusers")
14      */
15     private $groups;
16
17 }
  
```

Listing 7: app/models/Group.php

```

1 namespace models;
2
3 class Group{
4     /**
5      * @id
6      */
7     private $id;
8
9     private $name;
10
11     /**
12      * @manyToMany("targetEntity"=>"models\\User", "inversedBy"=>"groups")
13      * @joinTable("name"=>"groupusers")
14      */
15     private $users;
16
17 }
  
```

If the naming conventions are not respected for foreign keys, it is possible to specify the related fields.

Listing 8: app/models/Group.php

```

1  namespace models;
2
3  class Group{
4      /**
5       * @id
6       */
7      private $id;
8
9      private $name;
10
11     /**
12      * @manyToMany("targetEntity"=>"models\\User", "inversedBy"=>"groupes")
13      * @joinTable("name"=>"groupeusers",
14      * "joinColumns"=>["name"=>"id_groupe", "referencedColumnName"=>"id"],
15      * "inverseJoinColumns"=>["name"=>"id_user", "referencedColumnName"=>"id"])
16     */
17     private $users;
18
19 }

```

16.2 ORM Annotations

16.2.1 Annotations for classes

@annotation	role	properties	role
@table	Defines the associated table name.		

16.2.2 Annotations for members

@annotation	role	properties	role
@id	Defines the primary key(s).		
@column	Specify the associated field characteristics.	name	Name of the associated field
		nullable	true if value can be null
		dbType	Type of the field in database
@transient	Specify that the field is not persistent.		

16.2.3 Associations

@annotation (extends)	role	properties [optional]	role
@manyToOne	Defines a single-valued association to another entity class.		
@joinColumn (@column)	Indicates the foreign key in many-ToOne asso.	className	Class of the member
		[referenced-Column-Name]	Name of the associated column
@oneToMany	Defines a multi-valued association to another entity class.	className	Class of the objects in member
		[mappedBy]	Name of the association-mapping attribute on the owning side
@manyToMany	Defines a many-valued association with many-to-many multiplicity	targetEntity	Class of the objects in member
		[inversedBy]	Name of the association-member on the inverse-side
		[mappedBy]	Name of the association-member on the owning side
@joinTable	Defines the association table for many-to-many multiplicity	name	The name of the association table
		[joinColumns]	@column => name and referenced-ColumnName for this side
		[inverseJoin-Columns]	@column => name and referenced-ColumnName for the other side

The **DAO** class is responsible for loading and persistence operations on models :

17.1 Connecting to the database

Check that the database connection parameters are correctly entered in the configuration file:

```
Ubiquity config -f=database
```

Since 2.3.0 release

Database startup with `DAO::startDatabase($config)` in `services.php` file is useless, no need to start the database, the connection is made automatically at the first request. Use `DAO::start()` in `app/config/services.php` file when using several databases (with multi db feature)

17.2 Loading data

17.2.1 Loading an instance

Loading an instance of the `models\User` class with id 5

```
use Ubiquity\orm\DAO;
use models\User;

$user=DAO::getById(User::class, 5);
```

Loading an instance using a condition:

```
use Ubiquity\orm\DAO;
use models\User;
```

(continues on next page)

(continued from previous page)

```
DAO::getOne(User::class, 'name= ?', false, ['DOE']);
```

BelongsTo loading

By default, members defined by a **belongsTo** relationship are automatically loaded

Each user belongs to only one category:

```
$user=DAO::getById(User::class,5);  
echo $user->getCategory()->getName();
```

It is possible to prevent this default loading ; the third parameter allows the loading or not of belongsTo members:

```
$user=DAO::getOne(User::class,5, false);  
echo $user->getCategory(); // NULL
```

HasMany loading

Loading **hasMany** members must always be explicit ; the third parameter allows the explicit loading of members.

Each user has many groups:

```
$user=DAO::getOne(User::class,5, ['groupes']);  
foreach($user->getGroupes() as $groupe) {  
    echo $groupe->getName(). '<br>';  
}
```

Composite primary key

Either the *ProductDetail* model corresponding to a product ordered on a command and whose primary key is composite:

Listing 1: app/models/ProductDetail.php

```
1  namespace models;  
2  class ProductDetail{  
3      /**  
4       * @id  
5       */  
6      private $idProduct;  
7  
8      /**  
9       * @id  
10     */  
11     private $idCommand;  
12  
13     ...  
14 }
```

The second parameter *\$keyValues* can be an array if the primary key is composite:

```
$productDetail=DAO::getOne(ProductDetail::class,[18,'BF327']);
echo 'Command:'. $productDetail->getCommande(). '<br>';
echo 'Product:'. $productDetail->getProduct(). '<br>';
```

17.2.2 Loading multiple objects

Loading instances of the *User* class:

```
$users=DAO::getAll(User::class);
foreach($users as $user){
    echo $user->getName(). "<br>";
}
```

loading of related members

Loading instances of the *User* class with its category and its groups :

```
$users=DAO::getAll(User::class,['groupes','category']);
foreach($users as $user){
    echo "<h2>". $user->getName(). "</h2>";
    echo $user->getCategory(). "<br>";
    echo "<h3>Groups</h3>";
    echo "<ul>";
    foreach($user->getGroupes() as $groupe){
        echo "<li>". $groupe->getName(). "</li>";
    }
    echo "</ul>";
}
```

Descending in the hierarchy of related objects: Loading instances of the *User* class with its category, its groups and the organization of each group :

```
$users=DAO::getAll(User::class,['groupes.organization','category']);
foreach($users as $user){
    echo "<h2>". $user->getName(). "</h2>";
    echo $user->getCategory(). "<br>";
    echo "<h3>Groups</h3>";
    echo "<ul>";
    foreach($user->getGroupes() as $groupe){
        echo "<li>". $groupe->getName(). "<br>";
        echo "<li>". $groupe->getOrganization()->getName(). "</li>";
    }
    echo "</ul>";
}
```

Using wildcards:

Loading instances of the *User* class with its category, its groups and all related members of each group:

```
$users=DAO::getAll(User::class,['groupes.*','category']);
```

17.2.3 Querying using conditions

Simple queries

The *condition* parameter is equivalent to the WHERE part of an SQL statement:

```
$users=DAO::getAll(User::class, 'firstName like "bren%" and not suspended', false);
```

To avoid SQL injections and benefit from the preparation of statements, it is preferable to perform a parameterized query:

```
$users=DAO::getAll(User::class, 'firstName like ? and suspended= ?', false, ['bren%',  
↪ false]);
```


UQueries

The use of **U-queries** allows to set conditions on associate members:

Selection of users whose organization has the domain **lecnam.net**:

```
$users=DAO::uGetAll(User::class, 'organization.domain= ?', false, ['lecnam.net']);
```



It is possible to view the generated request in the logs (if logging is enabled):

Database			
<input type="checkbox"/>		prepareAndFetchAll	SELECT `User`.`id`,`User`.`firstname`,`User`.`lastname`,`User`.`email`,`User`.`password`,`User`.`suspended`,`User`.`idOrganization` FROM `User` INNER JOIN `Organization` `Organization_U5cc496dd67c4a` ON `User`.`idOrganization`=`Organization_U5cc496dd67c4a`.`id` WHERE Organization_U5cc496dd67c4a.domain=? 1

The result can be verified by selecting all users in this organization:

```
$organization=DAO::getOne(Organization::class, 'domain= ?', ['users'], ['lecnam.net']);  
$users=$organization->getUsers();
```

The corresponding logs:

Database			
<input type="checkbox"/>		prepareAndFetchAll	SELECT `User`.`id`,`User`.`firstname`,`User`.`lastname`,`User`.`email`,`User`.`password`,`User`.`suspended`,`User`.`idOrganization` FROM `User` WHERE idOrganization=? 1
<input type="checkbox"/>		prepareAndFetchAll	SELECT `Organization`.`id`,`Organization`.`name`,`Organization`.`domain`,`Organization`.`aliases` FROM `Organization` WHERE domain=? limit 1 1

17.2.4 Counting

Existence testing

```
if(DAO::exists(User::class, 'lastname like ?', ['SMITH'])) {  
    //there's a Mr SMITH  
}
```

Counting

To count the instances, what not to do, if users are not already loaded:

```
$users=DAO::getAll(User::class);  
echo "there are ". \count($users) . " users";
```

What needs to be done:

```
$count=DAO::count (User::class);
echo "there are $count users";
```

With a condition:

```
$notSuspendedCount=DAO::count (User::class, 'suspended = ?', [false]);
```

with a condition on associated objects:

Number of users belonging to the **OTAN** named organization.

```
$count=DAO::uCount (User::class, 'organization.name= ?', ['OTAN']);
```

17.3 Modifying data

17.3.1 Adding an instance

Adding an organization:

```
$orga=new Organization();
$orga->setName('Foo');
$orga->setDomain('foo.net');
if(DAO::save($orga)) {
    echo $orga.' added in database';
}
```

Adding an instance of User, in an organization:

```
$orga=DAO::getById (Organization::class, 1);
$user=new User();
$user->setFirstname('DOE');
$user->setLastname('John');
$user->setEmail('doe@bar.net');
$user->setOrganization($orga);
if(DAO::save($user)) {
    echo $user.' added in database in '.$orga;
}
```

17.3.2 Updating an instance

First, the instance must be loaded:

```
$orga=DAO::getOne (Organization::class, 'domain= ?', false, ['foo.net']);
$orga->setAliases('foo.org');
if(DAO::save($orga)) {
    echo $orga.' updated in database';
}
```

17.3.3 Deleting an instance

If the instance is loaded from database:

```
$orga=DAO::getById(Organization::class,5,false);
if(DAO::remove($orga)){
    echo $orga.' deleted from database';
}
```

If the instance is not loaded, it is more appropriate to use the *delete* method:

```
if(DAO::delete(Organization::class,5)){
    echo 'Organization deleted from database';
}
```

17.4 Deleting multiple instances

Deletion of multiple instances without prior loading:

```
if($res=DAO::deleteAll(models\User::class, 'id in (?, ?, ?)', [1,2,3])){
    echo "$res elements deleted";
}
```

17.5 Bulk queries

Bulk queries allow several operations (insertion, modification or deletion) to be performed in a single query, which contributes to improved performance.

17.5.1 Bulk inserts

Insertions example:

```
$u = new User();
$u->setName('Martin1');
DAO::toInsert($u);
$u = new User();
$u->setName('Martin2');
DAO::toInsert($u);
//Perform inserts
DAO::flushInserts();
```

17.5.2 Bulk updates

Updates example:

```
$users = DAO::getAll(User::class, 'name like ?', false, [
    'Martin%'
]);
foreach ($users as $user) {
    $user->setName(\strtoupper($user->getName()));
    DAO::toUpdate($user);
}
DAO::flushUpdates();
```


17.5.3 Bulk deletes

Deletions example

```
$users = DAO::getAll(User::class, 'name like ?', false, [
    'BULK%'
]);
DAO::toDeletes($users);
DAO::flushDeletes();
```

The `DAO::flush()` method can be called if insertions, updates or deletions are pending.

17.6 Transactions

17.6.1 Explicit transactions

All DAO operations can be inserted into a transaction, so that a series of changes can be atomized:

```
try{
    DAO::beginTransaction();
    $orga=new Organization();
    $orga->setName('Foo');
    DAO::save($orga);

    $user=new User();
    $user->setFirstname('DOE');
    $user->setOrganization($orga);
    DAO::save($user);
    DAO::commit();
}catch (\Exception $e){
    DAO::rollBack();
}
```

In case of multiple databases defined in the configuration, transaction-related methods can take the database offset defined in parameter.

```
DAO::beginTransaction('db-messagerie');
//some DAO operations on messagerie models
DAO::commit('db-messagerie');
```

17.6.2 Implicit transactions

Some DAO methods implicitly use transactions to group together insert, update or delete operations.

```
$users=DAO::getAll(User::class);
foreach ($users as $user){
    $user->setSuspended(true);
    DAO::toUpdate($user);
}
DAO::updateGroups();//Perform updates in a transaction
```

17.7 SDAO class

The **SDAO** class accelerates CRUD operations for the business classes without relationships.

Models must in this case declare public members only, and not respect the usual encapsulation.

Listing 2: app/models/Product.php

```
1 namespace models;
2 class Product{
3     /**
4      * @id
5      */
6     public $id;
7
8     public $name;
9
10    ...
11 }
```

The **SDAO** class inherits from **DAO** and has the same methods for performing CRUD operations.

```
use Ubiquity\orm\DAO;

$product=DAO::getById(Product::class, 5);
```

17.8 Prepared DAO queries

Preparing certain requests can improve performance with Swoole, Workerman or Roadrunner servers. This preparation initializes the objects that will then be used to execute the query. This initialization is done at server startup, or at the startup of each worker, if such an event exists.

17.8.1 Swoole sample

Preparation

Listing 3: app/config/swooleServices.php

```
$swooleServer->on('workerStart', function ($srv) use (&$config) {
    \Ubiquity\orm\DAO::startDatabase($config);
    \Ubiquity\orm\DAO::prepareGetById('user', User::class);
    \Ubiquity\orm\DAO::prepareGetAll('productsByName', Product::class, 'name like ?');
});
```

Usage

Listing 4: app/controllers/UsersController.php

```
public function displayUser($idUser) {
    $user=DAO::executePrepared('user', [1]);
    echo $user->getName();
}
```

(continues on next page)

(continued from previous page)

```
}  
  
public function displayProducts($name) {  
    $products=DAO::executePrepared('productsByName', [$name]);  
    ...  
}
```

Note: For all Http features, Ubiquity uses technical classes containing static methods. This is a design choice to avoid dependency injection that would degrade performances.

The **URequest** class provides additional functionality to more easily manipulate native **\$_POST** and **\$_GET** php arrays.

18.1 Retrieving data

18.1.1 From the get method

The **get** method returns the *null* value if the key **name** does not exist in the get variables.

```
use Ubiquity\utils\http\URequest;  
  
$name=URequest::get("name");
```

The **get** method can be called with the optional second parameter returning a value if the key does not exist in the get variables.

```
$name=URequest::get("name",1);
```

18.1.2 From the post method

The **post** method returns the *null* value if the key **name** does not exist in the post variables.

```
use Ubiquity\utils\http\URequest;  
  
$name=URequest::post("name");
```

The **post** method can be called with the optional second parameter returning a value if the key does not exist in the post variables.

```
$name=URequest::post("name",1);
```

The **getPost** method applies a callback to the elements of the `$_POST` array and return them (default callback : **htmlEntities**) :

```
$protectedValues=URequest::getPost();
```

18.2 Retrieving and assigning multiple data

It is common to assign the values of an associative array to the members of an object. This is the case for example when validating an object modification form.

The **setValuesToObject** method performs this operation :

Consider a **User** class:

```
class User {
    private $id;
    private $firstname;
    private $lastname;

    public function setId($id) {
        $this->id=$id;
    }
    public function getId() {
        return $this->id;
    }

    public function setFirstname($firstname) {
        $this->firstname=$firstname;
    }
    public function getFirstname() {
        return $this->firstname;
    }

    public function setLastname($lastname) {
        $this->lastname=$lastname;
    }
    public function getLastname() {
        return $this->lastname;
    }
}
```

Consider a form to modify a user:

```
<form method="post" action="Users/update">
  <input type="hidden" name="id" value="{{user.id}}">
  <label for="firstname">Firstname:</label>
  <input type="text" id="firstname" name="firstname" value="{{user.firstname}}">
  <label for="lastname">Lastname:</label>
  <input type="text" id="lastname" name="lastname" value="{{user.lastname}}">
  <input type="submit" value="validate modifications">
</form>
```

The **update** action of the **Users** controller must update the user instance from POST values. Using the **setPostValuesToObject** method avoids the assignment of variables posted one by one to the members of the object. It is also possible to use **setGetValuesToObject** for the **get** method, or **setValuesToObject** to assign the values of any associative array to an object.

Listing 1: app/controllers/Users.php

```

1  namespace controllers;
2
3  use Ubiquity\orm\DAO;
4  use Ubiquity\utils\http\URequest;
5
6  class Users extends BaseController{
7      ...
8      public function update() {
9          $user=DAO::getOne("models\User",URequest::post("id"));
10         URequest::setPostValuesToObject($user);
11         DAO::update($user);
12     }
13 }

```

Note: **SetValuesToObject** methods use setters to modify the members of an object. The class concerned must therefore implement setters for all modifiable members.

18.3 Testing the request

18.3.1 isPost

The **isPost** method returns *true* if the request was submitted via the POST method: In the case below, the *initialize* method only loads the *vHeader.html* view if the request is not an Ajax request.

Listing 2: app/controllers/Users.php

```

1  namespace controllers;
2
3  use Ubiquity\orm\DAO;
4  use Ubiquity\utils\http\URequest;
5
6  class Users extends BaseController{
7      ...
8      public function update() {
9          if(URequest::isPost()){
10             $user=DAO::getOne("models\User",URequest::post("id"));
11             URequest::setPostValuesToObject($user);
12             DAO::update($user);
13         }
14     }
15 }

```

18.3.2 isAjax

The **isAjax** method returns *true* if the query is an Ajax query:

Listing 3: app/controllers/Users.php

```
1  ...
2  public function initialize() {
3      if(!URequest::isAjax()) {
4          $this->loadView("main/vHeader.html");
5      }
6  }
7  ...
```

18.3.3 isCrossSite

The **isCrossSite** method verifies that the query is not cross-site.

Response

Note: For all Http features, Ubiquity uses technical classes containing static methods. This is a design choice to avoid dependency injection that would degrade performances.

The **UResponse** class handles only the headers, not the response body, which is conventionally provided by the content displayed by the calls used to output data (echo, print ...).

The **UResponse** class provides additional functionality to more easily manipulate response headers.

19.1 Adding or modifying headers

```
use Ubiquity\utils\http\UResponse;  
$animal='camel';  
UResponse::header('Animal', $animal);
```

Forcing multiple header of the same type:

```
UResponse::header('Animal', 'monkey', false);
```

Forces the HTTP response code to the specified value:

```
UResponse::header('Messages', $message, false, 500);
```

19.2 Defining specific headers

19.2.1 content-type

Setting the response content-type to **application/json**:

```
UResponse::asJSON();
```

Setting the response content-type to **text/html**:

```
UResponse::asHtml();
```

Setting the response content-type to **plain/text**:

```
UResponse::asText();
```

Setting the response content-type to **application/xml**:

```
UResponse::asXml();
```

Defining specific encoding (default value is always **utf-8**):

```
UResponse::asHtml('iso-8859-1');
```

19.3 Cache

Forcing the disabling of the browser cache:

```
UResponse::noCache();
```

19.4 Accept

Define which content types, expressed as MIME types, the client is able to understand. See [Accept default values](#)

```
UResponse::setAccept('text/html');
```

19.5 CORS responses headers

Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell a browser to let your web application running at one origin (domain) have permission to access selected resources from a server at a different origin.

19.5.1 Access-Control-Allow-Origin

Setting allowed origin:

```
UResponse::setAccessControlOrigin('http://myDomain/');
```

19.5.2 Access-Control-Allow-methods

Defining allowed methods:

```
UResponse::setAccessControlMethods('GET, POST, PUT, DELETE, PATCH, OPTIONS');
```

19.5.3 Access-Control-Allow-headers

Defining allowed headers:

```
UResponse::setAccessControlHeaders('X-Requested-With, Content-Type, Accept, Origin, ↵
↵Authorization');
```

19.5.4 Global CORS activation

enabling CORS for a domain with default values:

- allowed methods: GET, POST, PUT, DELETE, PATCH, OPTIONS
- allowed headers: X-Requested-With, Content-Type, Accept, Origin, Authorization

```
UResponse::enableCors('http://myDomain/');
```

19.6 Testing response headers

Checking if headers have been sent:

```
if(!UResponse::isSent()){
    //do something if headers are not send
}
```

Testing if response content-type is **application/json**:

Important: This method only works if you used the UResponse class to set the headers.

```
if(UResponse::isJSON()){
    //do something if response is a JSON response
}
```

Note: For all Http features, Ubiquity uses technical classes containing static methods. This is a design choice to avoid dependency injection that would degrade performances.

The **USession** class provides additional functionality to more easily manipulate native **\$_SESSION** php array.

20.1 Starting the session

The Http session is started automatically if the **sessionName** key is populated in the **app/config.php** configuration file:

```
<?php
return array(
    ...
    "sessionName"=>"key-for-app",
    ...
);
```

If the **sessionName** key is not populated, it is necessary to start the session explicitly to use it:

```
use Ubiquity\utils\http\USession;
...
USession::start("key-for-app");
```

Note: The **name** parameter is optional but recommended to avoid conflicting variables.

20.2 Creating or editing a session variable

```
use Ubiquity\utils\http\USession;  
  
USession::set("name", "SMITH");  
USession::set("activeUser", $user);
```

20.3 Retrieving data

The **get** method returns the *null* value if the key **name** does not exist in the session variables.

```
use Ubiquity\utils\http\USession;  
  
$name=USession::get("name");
```

The **get** method can be called with the optional second parameter returning a value if the key does not exist in the session variables.

```
$name=USession::get("page", 1);
```

Note: The **session** method is an alias of the **get** method.

The **getAll** method returns all session vars:

```
$sessionVars=USession::getAll();
```

20.4 Testing

The **exists** method tests the existence of a variable in session.

```
if(USession::exists("name")){  
    //do something when name key exists in session  
}
```

The **isStarted** method checks the session start

```
if(USession::isStarted()){  
    //do something if the session is started  
}
```

20.5 Deleting variables

The **delete** method remove a session variable:

```
USession::delete("name");
```

20.6 Explicit closing of the session

The **terminate** method closes the session correctly and deletes all session variables created:

```
USession::terminate();
```

Note: For all Http features, Ubiquity uses technical classes containing static methods. This is a design choice to avoid dependency injection that would degrade performances.

The **UCookie** class provides additional functionality to more easily manipulate native **\$_COOKIES** php array.

21.1 Cookie creation or modification

```
use Ubiquity\utils\http\UCookie;

$cookie_name = 'user';
$cookie_value = 'John Doe';
UCookie::set($cookie_name, $cookie_value); //duration : 1 day
```

Creating a cookie that lasts 5 days:

```
UCookie::set($cookie_name, $cookie_value, 5*60*60*24);
```

On a particular domain:

```
UCookie::set($cookie_name, $cookie_value, 5*60*60*24, '/admin');
```

Sending a cookie without urlencoding the cookie value:

```
UCookie::setRaw($cookie_name, $cookie_value);
```

Testing the cookie creation:

```
if(UCookie::setRaw($cookie_name, $cookie_value)){
    //cookie created
}
```

21.2 Retrieving a Cookie

```
$userName=UCookie::get('user');
```

21.2.1 Testing the existence

```
if(UCookie::exists('user')){  
    //do something if cookie user exists  
}
```

21.2.2 Using a default value

If the page cookie does not exist, the default value of 1 is returned:

```
$page=UCookie::get('page',1);
```

21.3 Deleting a cookie

Deleting the cookie with the name **page**:

```
UCookie::delete('page');
```

21.4 Deleting all cookies

Deleting all cookies from the entire domain:

```
UCookie::deleteAll();
```

Deleting all cookies from the domain **admin**:

```
UCookie::deleteAll('/admin');
```

Ubiquity uses Twig as the default template engine (see [Twig documentation](#)). The views are located in the **app/views** folder. They must have the **.html** extension for being interpreted by Twig.

Ubiquity can also be used with a PHP view system, to get better performance, or simply to allow the use of php in the views.

22.1 Loading

Views are loaded from controllers:

Listing 1: app/controllers/Users.php

```
1 namespace controllers;
2
3 class Users extends BaseController{
4     ...
5     public function index(){
6         $this->loadView("index.html");
7     }
8 }
9 }
```

22.1.1 Default view loading

If you use the default view naming method : The default view associated to an action in a controller is located in `views/controller-name/action-name` folder:

```
views
├── Users
│   └── info.html
```

Listing 2: app/controllers/Users.php

```
1 namespace controllers;
2
3 class Users extends BaseController{
4     ...
5     public function info(){
6         $this->loadDefaultView();
7     }
8 }
9 }
```

22.2 Loading and passing variables

Variables are passed to the view with an associative array. Each key creates a variable of the same name in the view.

Listing 3: app/controllers/Users.php

```
1 namespace controllers;
2
3 class Users extends BaseController{
4     ...
5     public function display($message, $type){
6         $this->loadView("users/display.html", ["message"=>$message, "type
7         =>$type]);
8     }
9 }
```

In this case, it is useful to call `compact` for creating an array containing variables and their values :

Listing 4: app/controllers/Users.php

```
1 namespace controllers;
2
3 class Users extends BaseController{
4     ...
5     public function display($message, $type){
6         $this->loadView("users/display.html", compact("message", "type"));
7     }
8 }
9 }
```

22.3 Displaying in view

The view can then display the variables:

Listing 5: users/display.html

```
<h2>{{type}}</h2>
<div>{{message}}</div>
```

Variables may have attributes or elements you can access, too.

You can use a dot (.) to access attributes of a variable (methods or properties of a PHP object, or items of a PHP array), or the so-called “subscript” syntax ([]):

```
{{ foo.bar }}
{{ foo['bar'] }}
```

22.4 Ubiquity extra functions

Global app variable provides access to predefined Ubiquity Twig features:

- app is an instance of Framework and provides access to public methods of this class.

Get framework installed version:

```
{{ app.version() }}
```

Return the active controller and action names:

```
{{ app.getController() }}
{{ app.getAction() }}
```

Return global wrapper classes :

For request:

```
{{ app.getRequest().isAjax() }}
```

For session :

```
{{ app.getSession().get('homePage', 'index') }}
```

see [Framework class in API](#) for more.

22.5 PHP view loading

Disable if necessary Twig in the configuration file by deleting the **templateEngine** key.

Then create a controller that inherits from SimpleViewController, or SimpleViewAsyncController if you use **Swoole** or **Workerman**:

Listing 6: app/controllers/Users.php

```
1 namespace controllers;
2
3 use Ubiquity\controllers\SimpleViewController;
4
5 class Users extends SimpleViewController{
```

(continues on next page)

(continued from previous page)

```
6      ***
7      public function display($message, $type) {
8          $this->loadView("users/display.php", compact("message", "type"));
9      }
10 }
11 }
```

Note: In this case, the functions for loading assets and themes are not supported.

CHAPTER 23

Assets

Assets correspond to javascript files, style sheets, fonts, images to include in your application. They are located from the **public/assets** folder. It is preferable to separate resources into sub-folders by type.

```
public/assets
├── css
│   ├── style.css
│   └── semantic.min.css
└── js
    └── jquery.min.js
```

Integration of css or js files :

```
{{ css('css/style.css') }}
{{ css('css/semantic.min.css') }}

{{ js('js/jquery.min.js') }}
```

```
{{ css('https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.4.1/semantic.min.css') }}
{{ js('https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.4.1/semantic.min.js') }}
```

CDN with extra parameters:

```
{{ css('https://cdn.jsdelivr.net/npm/foundation-sites@6.5.3/dist/css/foundation.min.
→css',{crossorigin: 'anonymous',integrity: 'sha256-/PFxCnsMh+...'}) }}
```

Note: The themes are totally useless if you only have one presentation to apply.

Ubiquity support themes wich can have it's own assets and views according to theme template to be rendered by controller. Each controller action can render a specific theme, or they can use the default theme configured at *config.php* file in `templateEngineOptions => array("activeTheme" => "semantic")`.

Ubiquity is shipped with 3 default themes : **Bootstrap**, **Foundation** and **Semantic-UI**.


24.1 Installing a theme

With devtools, run :

```
Ubiquity install-theme bootstrap
```

The installed theme is one of **bootstrap**, **foundation** or **semantic**.

With **webtools**, you can do the same, provided that the **devtools** are installed and accessible (Ubiquity folder added in the system path) :



Themes
Themes module

Check devtools command

Ubiquity

☒

Save



Ubiquity devtools

- The project folder is C:\xampp7.3\htdocs\verif
- PHP 7.3.2
- Ubiquity devtools (1.1.7+)
- Ubiquity 2.0.11+


Active theme : **semantic**

Installed themes


semantic


Install an existing theme


bootstrap



foundation



Click to install one theme


Create a new theme

Theme name

extends...

Create theme

24.2 Creating a new theme

With devtools, run :


```
Ubiquity create-theme myTheme
```

Creating a new theme from Bootstrap, Semantic...


With devtools, run :

```
Ubiquity create-theme myBootstrap -x=bootstrap
```

With webtools :



Themes
 Themes module

Check devtools command
 Ubiquity
 ☒
 Save



Ubiquity devtools
 • The project folder is C:\xampp7.3\htdocs\verif
 • PHP 7.3.2
 • Ubiquity devtools (1.1.7+)
 • Ubiquity 2.0.11+

Active theme: **semantic**

Installed themes
 semantic


 Install an existing theme

bootstrap
 foundation


 Create a new theme

myBootstrap
 bootstrap
 Create theme

Click to create a new theme

24.3 Theme functioning and structure

24.3.1 Structure

Theme view folder

The views of a theme are located from the **app/views/themes/theme-name** folder

```

app/views
├── themes
│   ├── bootstrap
│   │   └── main
│   │       ├── vHeader.html
│   │       └── vFooter.html
│   └── semantic
│       └── main
│           ├── vHeader.html
│           └── vFooter.html

```

The controller base class is responsible for loading views to define the header and footer of each page :

Listing 1: app/controllers/ControllerBase.php

```

1  <?php
2  namespace controllers;
3
4  use Ubiquity\controllers\Controller;
5  use Ubiquity\utils\http\URequest;
6
7  /**

```

(continues on next page)

(continued from previous page)

```

8      * ControllerBase.
9      **/
10     abstract class ControllerBase extends Controller{
11         protected $headerView = "@activeTheme/main/vHeader.html";
12         protected $footerView = "@activeTheme/main/vFooter.html";
13
14         public function initialize() {
15             if (! URequest::isAjax ()) {
16                 $this->loadView ( $this->headerView );
17             }
18         }
19         public function finalize() {
20             if (! URequest::isAjax ()) {
21                 $this->loadView ( $this->footerView );
22             }
23         }
24     }

```

Theme assets folder

The assets of a theme are created inside `public/assets/theme-name` folder.

The structure of the assets folder is often as follows :

```

public/assets/bootstrap
├── css
│   ├── style.css
│   └── all.min.css
├── scss
│   ├── myVariables.scss
│   └── app.scss
├── webfonts
└── img

```

24.4 Change of the active theme

24.4.1 Persistent change

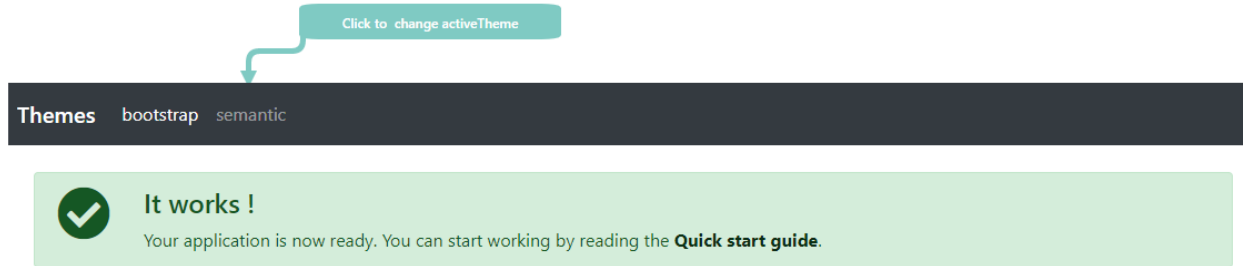
`activeTheme` is defined in `app/config/config.php` with `templateEngineOptions => array("activeTheme" => "semantic")`

The active theme can be changed with **devtools** :

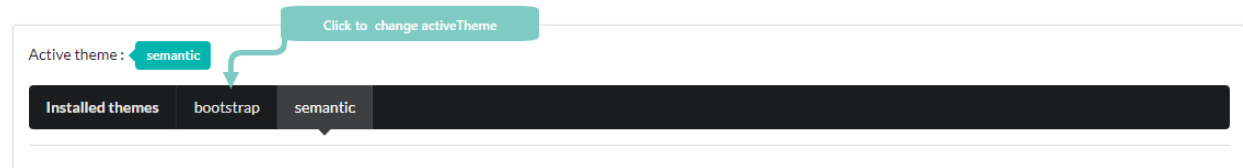
```
Ubiquity config:set --templateEngineOptions.activeTheme=bootstrap
```

It can also be done from the home page, or with **webtools** :

From the home page :



From the webtools :



This change can also be made at runtime :

From a controller :

```
ThemeManager::saveActiveTheme('bootstrap');
```

24.4.2 Non-persistent local change

To set a specific theme for all actions within a controller, the simplest method is to override the controller's **initialize** method :

Listing 2: app/controllers/Users.php

```

1 namespace controllers;
2
3 use \Ubiquity\themes\ThemesManager;
4
5 class Users extends BaseController{
6
7     public function initialize(){
8         parent::initialize();
9         ThemesManager::setActiveTheme('bootstrap');
10    }
11 }
```

Or if the change should only concern one action :

Listing 3: app/controllers/Users.php

```

1 namespace controllers;
2
3 use \Ubiquity\themes\ThemesManager;
4
5 class Users extends BaseController{
6
7     public function doStuff(){
8         ThemesManager::setActiveTheme('bootstrap');
```

(continues on next page)

(continued from previous page)

```
9      ...
10    }
11  }
```

Conditional theme change, regardless of the controller :

Example with a modification of the theme according to a variable passed in the URL

Listing 4: app/config/services.php

```
1 use Ubiquity\themes\ThemesManager;
2 use Ubiquity\utils\http\URequest;
3
4 ...
5
6 ThemesManager::onBeforeRender(function() {
7     if(URequest::get("th")== 'bootstrap') {
8         ThemesManager::setActiveTheme("bootstrap");
9     }
10 });
```

24.4.3 Mobile device support

Add a mobile device detection tool. Installing MobileDetect:

```
composer require mobiledetect/mobiledetectlib
```

It is generally easier to create different views per device.

Create a specific theme for the mobile part (by creating a folder `views/themes/mobile` and putting the views specific to mobile devices in it). It is important in this case to use the same file names for the mobile and non-mobile part.

It is also advisable in this case that all view loadings use the **@activeTheme** namespace:

```
$this->loadView("@activeTheme/index.html");
```

index.html must be available in this case in the folders `views` and `views/themes/mobile`.

Global mobile detection (from services.php)

Listing 5: app/config/services.php

```
1 use Ubiquity\themes\ThemesManager;
2
3 ...
4
5 ThemesManager::onBeforeRender(function() {
6     $mb = new \Mobile_Detect();
7     if ($mb->isMobile()) {
8         ThemesManager::setActiveTheme('mobile');
9     }
10 });
```

Locale detection (from a controller)

Listing 6: app/controllers/FooController.php

```

1 use Ubiquity\themes\ThemesManager;
2
3 ...
4
5 public function initialize() {
6     $mb = new \Mobile_Detect();
7     if ($mb->isMobile()) {
8         ThemesManager::setActiveTheme('mobile');
9     }
10    parent::initialize();
11 }
```

24.5 View and assets loading

24.5.1 Views

For loading a view from the **activeTheme** folder, you can use the **@activeTheme** namespace :

Listing 7: app/controllers/Users.php

```

1 namespace controllers;
2
3 class Users extends BaseController{
4
5     public function action(){
6         $this->loadView('@activeTheme/action.html');
7         ...
8     }
9 }
```

If the **activeTheme** is **bootstrap**, the loaded view is `app/views/themes/bootstrap/action.html`.

24.5.2 DefaultView

If you follow the Ubiquity view naming model, the default view loaded for an action in a controller when a theme is active is : `app/views/themes/theme-name/controller-name/action-name.html`.

For example, if the **activeTheme** is **bootstrap**, the default view for the action `display` in the `Users` controller must be located in `app/views/themes/bootstrap/Users/display.html`.

Listing 8: app/controllers/Users.php

```

1 namespace controllers;
2
3 class Users extends BaseController{
4
5     public function display(){
6         $this->loadDefaultView();
7         ...
8     }
9 }
```

(continues on next page)

(continued from previous page)

```
8     }  
9 }
```

Note: The devtools commands to create a controller or an action and their associated view use the **@activeTheme** folder if a theme is active.

```
Ubiquity controller Users -v  
Ubiquity action Users.display -v
```

24.6 Assets loading

The mechanism is the same as for the views : @activeTheme namespace refers to the public/assets/theme-name/ folder

```
{{ css('@activeTheme/css/style.css') }}  
{{ js('@activeTheme/js/scripts.js') }}
```

If the **bootstrap** theme is active, the assets folder is public/assets/bootstrap/.

24.7 Css compilation

For Bootstrap or foundation, install sass:

```
npm install -g sass
```

Then run from the project root folder:

For bootstrap:

```
ssass public/assets/bootstrap/scss/app.scss public/assets/bootstrap/css/style.css --  
↪load-path=vendor
```

For foundation:

```
ssass public/assets/foundation/scss/app.scss public/assets/foundation/css/style.css --  
↪load-path=vendor
```


By default, Ubiquity uses the [phpMv-UI](#) library for the client-rich part. **PhpMv-UI** allows to create components based on Semantic-UI or Bootstrap and to generate jQuery scripts in PHP.

This library is used for the **webtools** administration interface.

25.1 Integration

By default, a **\$jquery** variable is injected in controllers at runtime.

This operation is done using dependency injection, in `app/config.php`:

Listing 1: `app/config.php`

```
...
"di"=>array(
    "@exec"=>array(
        "jquery"=>function ($controller) {
            return \Ubiquity\core\Framework::diSemantic(
                $controller);
        }
    )
)
...
```

So there's nothing to do, but to facilitate its use and allow code completion in a controller, it is recommended to add the following code documentation:

Listing 2: `app/controllers/FooController.php`

```
/**
 * Controller FooController
 * @property \Ajax\php\ubiquity\JsUtils $jquery
 */
```

(continues on next page)

(continued from previous page)

```
class FooController extends ControllerBase{  
    public function index() {}  
}
```

25.2 jQuery

25.2.1 Href to ajax requests

Create a new Controller and its associated view, then define the following routes:

Listing 3: app/controllers/FooController.php

```
1 namespace controllers;  
2  
3 class FooController extends ControllerBase {  
4  
5     public function index() {  
6         $this->loadview("FooController/index.html");  
7     }  
8  
9     /**  
10      *  
11      *@get("a", "name"=>"action.a")  
12      */  
13     public function aAction() {  
14         echo "a";  
15     }  
16  
17     /**  
18      *  
19      *@get("b", "name"=>"action.b")  
20      */  
21     public function bAction() {  
22         echo "b";  
23     }  
24 }
```

The associated view:

Listing 4: app/views/FooController/index.html

```
<a href="{{path('action.a')}}">Action a</a>  
<a href="{{path('action.b')}}">Action b</a>
```

Initialize router cache:

```
Ubiquity init:cache -t=controllers
```

Test this page in your browser at <http://127.0.0.1:8090/FooController>.

Transformation of requests into Ajax requests

The result of each ajax request should be displayed in an area of the page defined by its jQuery selector (`.result span`)

Listing 5: app/controllers/FooController.php

```
namespace controllers;

/**
 * @property \Ajax\php\ubiquity\JsUtils $jquery
 */
class FooController extends ControllerBase {

    public function index() {
        $this->jquery->getHref('a', '.result span');
        $this->jquery->renderView("FooController/index.html");
    }
    ...
}
```

Listing 6: app/views/FooController/index.html

```
<a href="{{path('action.a')}}">Action a</a>
<a href="{{path('action.b')}}">Action b</a>
<div class='result'>
    Selected action:
    <span>No One</span>
</div>
{{ script_foot | raw }}
```

Note: The `script_foot` variable contains the generated jquery script produced by the **renderView** method. The **raw** filter marks the value as being “safe”, which means that in an environment with automatic escaping enabled this variable will not be escaped.

Let’s add a little css to make it more professional:

Listing 7: app/views/FooController/index.html

```
<div class="ui buttons">
    <a class="ui button" href="{{path('action.a')}}">Action a</a>
    <a class="ui button" href="{{path('action.b')}}">Action b</a>
</div>
<div class='ui segment result'>
    Selected action:
    <span class="ui label">No One</span>
</div>
{{ script_foot | raw }}
```

If we want to add a new link whose result should be displayed in another area, it is possible to specify it via the **data-target** attribute

The new action:

Listing 8: app/controllers/FooController.php

```
namespace controllers;

class FooController extends ControllerBase {
    /**
     * @get("c", "name"=>"action.c")
     */
    public function cAction() {
        echo \rand(0, 1000);
    }
}
```

The associated view:

Listing 9: app/views/FooController/index.html

```
<div class="ui buttons">
  <a class="ui button" href="{{path('action.a')}}">Action a</a>
  <a class="ui button" href="{{path('action.b')}}">Action b</a>
  <a class="ui button" href="{{path('action.c')}}" data-target=".result p">Action c
</a>
</div>
<div class='ui segment result'>
  Selected action:
  <span class="ui label">No One</span>
  <p></p>
</div>
{{ script_foot | raw }}
```

GET:FooController/index

Action aAction bAction c

Action choisie : b86

Close

Definition of the ajax request attributes:

In the following example, the parameters passed to the **attributes** variable of the `getHref` method:

- remove the history of the navigation,
- make the ajax loader internal to the clicked button.

Listing 10: app/controllers/FooController.php

```

1 namespace controllers;
2
3 /**
4  * @property \Ajax\php\ubiquity\JsUtils $jquery
5  */
6 class FooController extends ControllerBase {
7
8     public function index() {
9         $this->jquery->getHref('a', '.result span', [
10             'hasLoader' => 'internal',
11             'historize' => false
12         ]);
13         $this->jquery->renderView("FooController/index.html");
14     }
15     ...
16 }

```

Note: It is possible to use the `postHref` method to use the **POST** http method.

25.2.2 Classical ajax requests

For this example, create the following database:

```

CREATE DATABASE `uguide` DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
USE `uguide`;

CREATE TABLE `user` (
  `id` int(11) NOT NULL,
  `firstname` varchar(30) NOT NULL,
  `lastname` varchar(30) NOT NULL,
  `password` varchar(30) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `user` (`id`, `firstname`, `lastname`) VALUES
(1, 'You', 'Evan'),
(2, 'Potencier', 'Fabien'),
(3, 'Otwell', 'Taylor');

ALTER TABLE `user` ADD PRIMARY KEY (`id`);
ALTER TABLE `user`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=1;

```

Connect the application to the database, and generate the *User* class:

With devtools:

```

Ubiquity config:set --database.dbName=uguide
Ubiquity all-models

```

Create a new Controller *UsersJqueryController*

```

Ubiquity controller UsersJqueryController -v

```

Create the following actions in *UsersJqueryController*:

Controller	Action [routes]	Default values
<div> <div>♥</div> <div>controllers\UsersJqueryController</div> <div>+</div> </div>	<div>⚡</div> <div>index ()</div> <div>/users/(index)?</div> <div>⚙️</div>	
	<div>⚡</div> <div>displayUsers ()</div> <div>/users/all/</div> <div>⚙️</div>	
	<div>⚡</div> <div>displayOneUser (userId)</div> <div>/users/(.+?)/</div> <div>⚙️</div>	

Index action

The *index* action must display a button to obtain the list of users, loaded via an ajax request:

Listing 11: app/controllers/UsersJqueryController.php

```

1 namespace controllers;
2
3 /**
4  * Controller UsersJqueryController
5  *
6  * @property \Ajax\php\ubiquity\JsUtils $jquery
7  * @route("users")
8  */
9 class UsersJqueryController extends ControllerBase {
10
11     /**
12      *
13      * {@inheritdoc}
14      * @see \Ubiquity\controllers\Controller::index()
15      * @get
16      */
17     public function index() {
18         $this->jquery->getOnClick('#users-bt', Router::path('display.users'), '
↪ #users', [
19             'hasLoader' => 'internal'
20         ]);
21         $this->jquery->renderDefaultView();
22     }
23 }
```

The default view associated to *index* action:

Listing 12: app/views/UsersJqueryController/index.html

```

<div class="ui container">
  <div id="users-bt" class="ui button">
    <i class="ui users icon"></i>
    Display <b>users</b>
  </div>
  <p></p>
  <div id="users">
  </div>
</div>
{{ script_foot | raw }}
```

displayUsers action

All users are displayed, and a click on a user must display the user details via a posted ajax request:

Listing 13: app/controllers/UsersJqueryController.php

```

1 namespace controllers;
2
3 /**
4  * Controller UsersJqueryController
5  *
6  * @property \Ajax\php\ubiquity\JsUtils $jquery
7  * @route("users")
8  */
9 class UsersJqueryController extends ControllerBase {
10     ...
11     /**
12      *
13      * @get("all","name"=>"display.users","cache"=>true)
14      */
15     public function displayUsers() {
16         $users = DAO::getAll(User::class);
17         $this->jquery->click('#close-bt', '$("#users").html("");');
18         $this->jquery->postOnClick('li[data-ajax]', Router::path('display.one.
19         ↪user', [
20             ""
21             ]), '{}', '#user-detail', [
22                 'attr' => 'data-ajax',
23                 'hasLoader' => false
24             ]);
25         $this->jquery->renderDefaultView([
26             'users' => $users
27         ]);
28     }

```

The view associated to *displayUsers* action:

Listing 14: app/views/UsersJqueryController/displayUsers.html

```

<div class="ui top attached header">
  <i class="users circular icon"></i>
  <div class="content">Users</div>
</div>
<div class="ui attached segment">
  <ul id='users-content'>
    {% for user in users %}
      <li data-ajax="{{user.id}}">{{user.firstname }} {{user.lastname}}</li>
    {% endfor %}
  </ul>
  <div id='user-detail'></div>
</div>
<div class="ui bottom attached inverted segment">
  <div id="close-bt" class="ui inverted button">Close</div>
</div>
{{ script_foot | raw }}

```

displayOneUser action

Listing 15: app/controllers/UsersJqueryController.php

```

1 namespace controllers;
2
3 /**
4  * Controller UsersJqueryController
5  *
6  * @property \Ajax\php\ubiquity\JsUtils $jquery
7  * @route("users")
8  */
9 class UsersJqueryController extends ControllerBase {
10     ...
11     /**
12      *
13      * @post("{userId}", "name"=>"display.one.user", "cache"=>true, "duration"=>3600)
14      */
15     public function displayOneUser($userId) {
16         $user = DAO::getById(User::class, $userId);
17         $this->jquery->hide('#users-content', '', '', true);
18         $this->jquery->click('#close-user-bt', '$("#user-detail").html("");$("
19         ↪#users-content").show();');
20         $this->jquery->renderDefaultView([
21             'user' => $user
22         ]);
23     }
24 }

```

The view associated to *displayOneUser* action:

Listing 16: app/views/UsersJqueryController/displayUsers.html

```

<div class="ui label">
    <i class="ui user icon"></i>
    Id
    <div class="detail">{{user.id}}</div>
</div>
<div class="ui label">
    Firstname
    <div class="detail">{{user.firstname}}</div>
</div>
<div class="ui label">
    Lastname
    <div class="detail">{{user.lastname}}</div>
</div>
<p></p>
<div id="close-user-bt" class="ui black button">
    <i class="ui users icon"></i>
    Return to users
</div>
{{ script_foot | raw }}

```

25.3 Semantic components

Next, we are going to make a controller implementing the same functionalities as before, but using **PhpMv-UI** components (Semantic part).

25.3.1 HtmlButton sample

Create a new Controller *UsersJqueryController*

```
Ubiquity controller UsersCompoController -v
```

Listing 17: app/controllers/UsersJqueryController.php

```

1 namespace controllers;
2
3 use Ubiquity\controllers\Router;
4
5 /**
6  * Controller UsersCompoController
7  *
8  * @property \Ajax\php\ubiquity\JsUtils $jquery
9  * @route("users-compo")
10  */
11 class UsersCompoController extends ControllerBase {
12
13     private function semantic() {
14         return $this->jquery->semantic();
15     }
16
17     /**
18      *
19      * @get
20      */
21     public function index() {
22         $bt = $this->semantic()->htmlButton('users-bt', 'Display users');
23         $bt->addIcon('users');
24         $bt->getOnClick(Router::path('display.compo.users'), '#users', [
25             'hasLoader' => 'internal'
26         ]);
27         $this->jquery->renderDefaultView();
28     }

```

Note: Calling `renderView` or `renderDefaultView` on the JQuery object performs the compilation of the component, and generates the corresponding HTML and JS.

The associated view integrates the button component with the *q* array available in the view :

Listing 18: app/views/UsersCompoController/index.html

```

<div class="ui container">
  {{ q['users-bt'] | raw }}
  <p></p>
  <div id="users">
    </div>
  </div>
  {{ script_foot | raw }}

```

//todo DataTable sample ++++++

CHAPTER 26

Normalizers

Note: The Normalizer module uses the static class **NormalizersManager** to manage normalization.

Note: The Validators module uses the static class **ValidatorsManager** to manage validation.

Validators are used to check that the member datas of an object complies with certain constraints.

27.1 Adding validators

Either the **Author** class that we want to use in our application :

Listing 1: app/models/Author.php

```
1 namespace models;
2
3 class Author {
4     /**
5      * @var string
6      * @validator("notEmpty")
7      */
8     private $name;
9
10    public function getName() {
11        return $this->name;
12    }
13
14    public function setName($name) {
15        $this->name=$name;
16    }
17 }
```

We added a validation constraint on the **name** member with the **@validator** annotation, so that it is not empty.

27.2 Generating cache

Run this command in console mode to create the cache data of the **Author** class :

```
Ubiquity init-cache -t=models
```

Validator cache is generated in `app/cache/contents/validators/models/Author.cache.php`.

27.3 Validating instances

27.3.1 an instance

```
public function testValidateAuthor() {
    $author=new Author();
    //Do something with $author
    $violations=ValidatorsManager::validate($author);
    if(sizeof($violations)>0){
        echo implode('<br>', ValidatorsManager::validate($author));
    }else{
        echo 'The author is valid!';
    }
}
```

if the **name** of the author is empty, this action should display:

```
name : This value should not be empty
```

The **validate** method returns an array of **ConstraintViolation** instances.

27.3.2 multiple instances

```
public function testValidateAuthors() {
    $authors=DAO::getAll(Author::class);
    $violations=ValidatorsManager::validateInstances($author);
    foreach($violations as $violation){
        echo $violation.'<br>';
    }
}
```

27.4 Models generation with default validators

When classes are automatically generated from the database, default validators are associated with members, based on the fields' metadatas.

```
Ubiquity create-model User
```

Listing 2: app/models/Author.php

```

1  namespace models;
2  class User{
3      /**
4       * @id
5       * @column("name"=>"id", "nullable"=>false, "dbType"=>"int(11)")
6       * @validator("id", "constraints"=>array("autoinc"=>true))
7       */
8      private $id;
9
10     /**
11      * @column("name"=>"firstname", "nullable"=>false, "dbType"=>"varchar(65)")
12      * @validator("length", "constraints"=>array("max"=>65, "notNull"=>true))
13      */
14     private $firstname;
15
16     /**
17      * @column("name"=>"lastname", "nullable"=>false, "dbType"=>"varchar(65)")
18      * @validator("length", "constraints"=>array("max"=>65, "notNull"=>true))
19      */
20     private $lastname;
21
22     /**
23      * @column("name"=>"email", "nullable"=>false, "dbType"=>"varchar(255)")
24      * @validator("email", "constraints"=>array("notNull"=>true))
25      * @validator("length", "constraints"=>array("max"=>255))
26      */
27     private $email;
28
29     /**
30      * @column("name"=>"password", "nullable"=>true, "dbType"=>"varchar(255)")
31      * @validator("length", "constraints"=>array("max"=>255))
32      */
33     private $password;
34
35     /**
36      * @column("name"=>"suspended", "nullable"=>true, "dbType"=>"tinyint(1)")
37      * @validator("isBool")
38      */
39     private $suspended;
40 }

```

These validators can then be modified. Modifications must always be followed by a re-initialization of the model cache.

```
Ubiquity init-cache -t=models
```

Models validation informations can be displayed with devtools :

```
Ubiquity info:validation -m=User
```

• The project folder is C:\xampp7.3\htdocs\verif

field	value
id	• [type: 'id',constraints: [autoinc: true]]
firstname	• [type: 'length',constraints: [max: 65,notNull: true]]
lastname	≡
email	• [type: 'email',constraints: [notNull: true]] • [type: 'length',constraints: [max: 255]]
password	• [type: 'length',constraints: [max: 255]]
suspended	• [type: 'isBool',constraints: []]

Gets validators on email field:

```
Ubiquity info:validation email -m=User
```

```
• email
  • type : 'email'
  • constraints : [notNull: true]
  • type : 'length'
  • constraints : [max: 255]
```

Validation informations are also accessible from the **models** part of the webtools:

models\User

Data administration

Datas

Structure

Validation

id	[[type: 'id', constraints: [autoinc: true]]]
firstname	[[type: 'length', constraints: [max: 65, notNull: true]]]
lastname	[[type: 'length', constraints: [max: 65, notNull: true]]]
email	[[type: 'email', constraints: [notNull: true]], [type: 'length', constraints: [max: 255]]]
password	[[type: 'length', constraints: [max: 255]]]
suspended	[[type: 'isBool', constraints: []]]

✓

Validate instances

27.5 Validator types

27.5.1 Basic

Validator	Roles	Constraints	Accepted values
isBool	Check if value is a boolean		true,false,0,1
isEmpty	Check if value is empty		“,null
isFalse	Check if value is false		false,'false',0,'0'
isNull	Check if value is null		null
isTrue	Check if value is true		true,'true',1,'1'
notEmpty	Check if value is not empty		!null && !"
notNull	Check if value is not null		!null
type	Check if value is of type {type}	{type}	

27.5.2 Comparison

27.5.3 Dates

27.5.4 Multiples

27.5.5 Strings

Note: The Transformers module uses the static class **TransformersManager** to manage data transformations.

Transformers are used to transform datas after loading from the database, or before displaying in a view.

28.1 Adding transformers

Either the **Author** class that we want to use in our application :

Listing 1: app/models/Author.php

```
1 namespace models;
2
3 class Author {
4     /**
5      * @var string
6      * @transformer("upper")
7      */
8     private $name;
9
10    public function getName() {
11        return $this->name;
12    }
13
14    public function setName($name) {
15        $this->name=$name;
16    }
17 }
```

We added a transformer on the **name** member with the **@transformer** annotation, in order to capitalize the name in the views.

28.2 Generating cache

Run this command in console mode to create the cache data of the **Author** class :

```
Ubiquity init-cache -t=models
```

transformer cache is generated with model metadatas in `app/cache/models/Author.cache.php`.

Transformers informations can be displayed with devtools :

```
Ubiquity info:model -m=Author -f=#transformers
```

field	value
#transformers	· toView : [name: 'Ubiquity\\contents\\transformation\\transformers\\UpperCase']

28.3 Using transformers

Start the **TransformersManager** in the file `app/config/services.php`:

Listing 2: `app/config/services.php`

```
\Ubiquity\contents\transformation\TransformersManager::startProd();
```

You can test the result in the administration interface:

Id	Name
1	JOHN GRISHAM
2	JOANNE ROWLING
3	STEPHEN EDWIN KING

or by creating a controller:

Listing 3: `app/controllers/Authors.php`

```

1 namespace controllers;
2
3 class Authors {
4
5     public function index() {
6         DAO::transformersOp='toView';
7         $authors=DAO::getAll(Author::class);
8         $this->loadDefaultView(['authors'=>$authors]);
9     }
10
11 }
```

Listing 4: app/views/Authors/index.html

```

<ul>
  {% for author in authors %}
    <li>{{ author.name }}</li>
  {% endfor %}
</ul>

```

28.4 Transformer types

28.4.1 transform

The **transform** type is based on the **TransformerInterface** interface. It is used when the transformed data must be converted into an object. The **DateTime** transformer is a good example of such a transformer:

- When loading the data, the Transformer converts the date from the database into an instance of php DateTime.
- Its **reverse** method performs the reverse operation (php date to database compatible date).

28.4.2 toView

The **toView** type is based on the **TransformerViewInterface** interface. It is used when the transformed data must be displayed in a view.

28.4.3 toForm

The **toForm** type is based on the **TransformerFormInterface** interface. It is used when the transformed data must be used in a form.

28.5 Transformers usage

28.5.1 Transform on data loading

If omitted, default **transformerOp** is **transform**

```
$authors=DAO::getAll (Author::class);
```

Set **transformerOp** to **toView**

```

DAO::transformersOp='toView';
$authors=DAO::getAll (Author::class);

```

28.5.2 Transform after loading

Return the transformed member value:

```
TransformersManager::transform($author, 'name', 'toView');
```

Return a transformed value:

```
TransformersManager::applyTransformer($author, 'name', 'john doe', 'toView');
```

Transform an instance by applying all defined transformers:

```
TransformersManager::transformInstance($author, 'toView');
```

28.6 Existing transformers

Transformer	Type(s)	Description
datetime	transform, toView, toForm	Transform a database datetime to a php DateTime object
upper	toView	Make the member value uppercase
lower	toView	Make the member value lowercase
firstUpper	toView	Make the member value first character uppercase
password	toView	Mask the member characters
md5	toView	Hash the value with md5

28.7 Create your own

28.7.1 Creation

Create a transformer to display a user name as a local email address:

Listing 5: app/transformers/toLocalEmail.php

```
1 namespace transformers;
2 use Ubiquity\contents\transformation\TransformerViewInterface;
3
4 class ToLocalEmail implements TransformerViewInterface{
5
6     public static function toView($value) {
7         if($value!=null)
8             return sprintf('%s@mydomain.local', strtolower($value));
9     }
10
11 }
```

28.7.2 Registration

Register the transformer by executing the following script:

```
TransformersManager::registerClassAndSave('localEmail',
↪\transformers\ToLocalEmail::class);
```

28.7.3 Usage

Listing 6: app/models/User.php

```
1 namespace models;
2
3 class User {
4     /**
5      * @var string
6      * @transformer("localEmail")
7      */
8     private $name;
9
10    public function getName() {
11        return $this->name;
12    }
13
14    public function setName($name) {
15        $this->name=$name;
16    }
17 }

```

```
DAO::transformersOp='toView';
$user=DAO::getOne(User::class, "name='Smith'");
echo $user->getName();

```

Smith user name will be displayed as **smith@mydomain.local**.

Note: The Translation module uses the static class **TranslatorManager** to manage translations.

29.1 Module structure

Translations are grouped by **domain**, within a **locale** :

In the translation root directory (default **app/translations**):

- Each locale corresponds to a subfolder.
- For each locale, in a subfolder, a domain corresponds to a php file.

```
translations
├── en_EN
│   ├── messages.php
│   └── blog.php
└── fr_FR
    ├── messages.php
    └── blog.php
```

- each domain file contains an associative array of translations **key-> translation value**
- **Each key can be associated with**
 - a translation
 - a translation containing variables (between % and %)
 - an array of translations for handle pluralization

Listing 1: app/translations/en_EN/messages.php

```
return [
    'okayBtn'=>'Okay',
    'cancelBtn'=>'Cancel',
    'deleteMessage'=>['No message to delete!', '1 message to delete.', '%count%_
↵messages to delete.'],
];
```

29.2 Starting the module

Module startup is logically done in the **services.php** file.

Listing 2: app/config/services.php

```
1 Ubiquity\cache\CacheManager::startProd($config);
2 Ubiquity\translation\TranslatorManager::start();
```

With no parameters, the call of the **start** method uses the locale **en_EN**, without fallbacklocale.

Important: The translations module must be started after the cache has started.

29.2.1 Setting the locale

Changing the locale when the manager starts:

Listing 3: app/config/services.php

```
1 Ubiquity\cache\CacheManager::startProd($config);
2 Ubiquity\translation\TranslatorManager::start('fr_FR');
```

Changing the locale after loading the manager:

```
TranslatorManager::setLocale('fr_FR');
```

29.2.2 Setting the fallbackLocale

The **en_EN** locale will be used if **fr_FR** is not found:

Listing 4: app/config/services.php

```
1 Ubiquity\cache\CacheManager::startProd($config);
2 Ubiquity\translation\TranslatorManager::start('fr_FR', 'en_EN');
```

29.3 Defining the root translations dir

If the **rootDir** parameter is missing, the default directory used is `app/translations`.

Listing 5: app/config/services.php

```

1 Ubiquity\cache\CacheManager::startProd($config);
2 Ubiquity\translation\TranslatorManager::start('fr_FR', 'en_EN', 'myTranslations');

```

29.4 Make a translation

29.4.1 With php

Translation of the **okayBtn** key into the default locale (specified when starting the manager):

```
$okBtnCaption=TranslatorManager::trans('okayBtn');
```

With no parameters, the call of the **trans** method uses the default locale, the domain **messages**.

Translation of the **message** key using a variable:

```
$okBtnCaption=TranslatorManager::trans('message', ['user'=>$user]);
```

In this case, the translation file must contain a reference to the **user** variable for the key **message**:

Listing 6: app/translations/en_EN/messages.php

```
['message'=>'Hello %user%!', ...];
```

29.4.2 In twig views:

Translation of the **okayBtn** key into the default locale (specified when starting the manager):

```
{{ t('okayBtn') }}
```

Translation of the **message** key using a variable:

```
{{ t('message', parameters) }}
```


30.1 Guiding principles

30.1.1 Forms validation

Client-side validation

It is preferable to perform an initial client-side validation to avoid submitting invalid data to the server.

Example of the creation of a form in the action of a controller (this part could be located in a dedicated service for a better separation of layers):

Listing 1: app/controllers/UsersManagement.php

```
1 public function index() {
2     $frm=$this->jquery->semantic()->dataForm('frm-user',new User());
3     $frm->setFields(['login','password','connection']);
4     $frm->fieldAsInput('login',
5         ['rules'=>'empty']
6     );
7     $frm->fieldAsInput('password',
8         [
9             'inputType'=>'password',
10            'rules'=>['empty','minLength[6]']
11        ]
12    );
13    $frm->setValidationParams(['on'=>'blur','inline'=>true]);
14    $frm->fieldAsSubmit('connection','fluid green','/submit','#response');
15    $this->jquery->renderDefaultView();
16 }
```

The Associated View:

Listing 2: app/views/UsersManagement/index.html

```
{{ q['frm-user'] | raw }}
{{ script_foot | raw }}
<div id="response"></div>
```

login *

password *

password must be at least 6 characters

Note: The CRUD controllers automatically integrate this client-side validation using the Validators attached to the members of the models.

```
#[Column(name: "password", nullable: true, dbType: "varchar(255)")]
#[Validator(type: "length", constraints: ["max">20, "min">6])]
#[Transformer(name: "password")]
private $password;
```

Server-side validation

It is preferable to restrict the URLs allowed to modify data. Beforehand, by specifying the Http method in the routes, and by testing the request :

```
#[Post(path: "/submit")]
public function submitUser() {
    if(!URequest::isCrossSite() && URequest::isAjax()){
        $datas=URequest::getPost();//post with htmlentities
        //Do something with $datas
    }
}
```

Note: The **Ubiquity-security** module offers additional control to avoid cross-site requests.

After modifying an object, it is possible to check its validity, given the validators attached to the members of the associated Model:

```
#[Post(path: "/submit")]
public function submitUser() {
    if(!URequest::isCrossSite()){
        $datas=URequest::getPost();//post with htmlentities
        $user=new User();
        URequest::setValuesToObject($user, $datas);
    }
}
```

(continues on next page)

(continued from previous page)

```

    $violations=ValidatorsManager::validate($user);
    if(\count($violations)==0){
        //do something with this valid user
    } else {
        //Display violations...
    }
}
}
}

```

30.1.2 DAO operations

It is always recommended to use parameterized queries, regardless of the operations performed on the data:

- To avoid SQL injections.
- To allow the use of prepared queries, speeding up processing.

```
$googleUsers=DAO::getAll(User::class, 'email like ?', false, ['%@gmail.com']);
```

```
$countActiveUsers=DAO::count(User::class, 'active= ?', [true]);
```

Note: DAO operations that take objects as parameters use this mechanism by default.

```
DAO::save($user);
```

30.1.3 Passwords management

The Password Transformer allows a field to be of the password type when displayed in an automatically generated CRUD form.

```
#[Transformer(name: "password")]
private $password;
```

After submission from a form, it is possible to encrypt a password from the URequest class:

```

$encryptedPassword=URequest::password_hash('password');
$user->setPassword($encryptedPassword);
DAO::save($user);

```

The algorithm used in this case is defined by the php PASSWORD_DEFAULT.

It is also possible to check a password entered by a user in the same way, to compare it to a hash:

```

if(URequest::password_verify('password', $existingPasswordHash)){
    //password is ok
}

```

Important: Set up Https to avoid sending passwords in clear text.

30.2 Security module/ ACL management

In addition to these few rules, you can install if necessary:

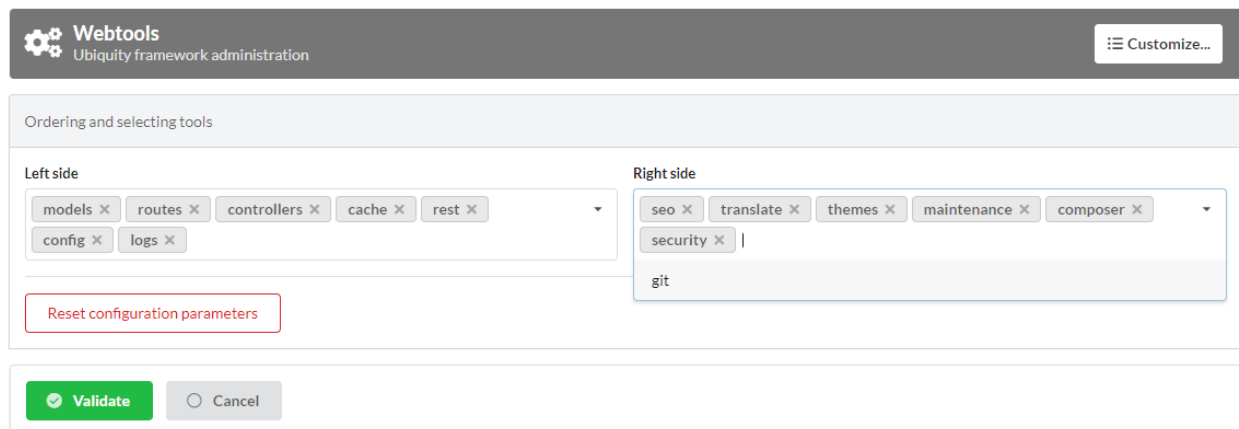
- *Ubiquity-acl*
- *Ubiquity-security*

31.1 Installation

Install the Ubiquity-security module from the command prompt or from the **Webtools** (Composer part).


```
composer require phpmv/ubiquity-security
```

Then activate the display of the Security part in the **Webtools**:






31.2 Session CSRF


The session is by default protected against CSRF attacks via the `VerifyCsrfToken` class (even without the **Ubiquity-security** module). A token instance (`CSRFToken`) is generated at the session startup. The validity of the token is then checked via a cookie at each request.


Security
 Manages security


Components

ubiquity-security	<div>  Installed ^0.0.1 </div>
ubiquity-acl	<div>  Install with composer </div>
Shieldon	<div>  Install with composer </div>

Services

Encryption manager	<div>  Start </div>
--------------------	--

Session

Started?	<div> <input type="checkbox"/> </div>
Instance class	<div> Ubiquity\utils\http\session\PhpSession </div>
Csrf protection	<div>  Ubiquity\utils\http\session\protection\VerifyCsrfToken </div>
Session count	<div> 2 </div>

Cookies

Transformer	<div> Nothing </div>
-------------	--

This protection can be customized by creating a class implementing the `VerifySessionCsrfInterface`.

Listing 1: `app/session/MyCsrfProtection.php`

```

class MyCsrfProtection implements VerifySessionCsrfInterface {
    private AbstractSession $sessionInstance;

    public function __construct(AbstractSession $sessionInstance) {
        $this->sessionInstance = $sessionInstance;
    }

    public function init() {
        //TODO when the session starts
    }

    public function clear() {
        //TODO when the session ends
    }

    public function start() {
        //TODO When the session starts or is resumed
    }

    public static function getLevel() {
        return 1; //An integer to appreciate the level of security
    }
}

```

(continues on next page)

(continued from previous page)

```

    }
}

```

Starting the custom protection in services:

Listing 2: app/config/services.php

```

use Ubiquity\utils\http\session\PhpSession;
use Ubiquity\controllers\Startup;
use app\session\MyCsrfProtection;

Startup::setSessionInstance(new PhpSession(new MyCsrfProtection()));

```

31.2.1 Deactivating the protection

If you do not need to protect your session against Csrf attacks, start the session with the `NoCsrfProtection` class.

Listing 3: app/config/services.php

```

use Ubiquity\utils\http\session\PhpSession;
use Ubiquity\controllers\Startup;
use Ubiquity\utils\http\session\protection\NoCsrfProtection;

Startup::setSessionInstance(new PhpSession(new NoCsrfProtection()));

```

31.3 CSRF manager

The **CsrfManager** service can be started directly from the **webtools** interface. Its role is to provide tools to protect sensitive routes from Csrf attacks (the ones that allow the validation of forms for example).

✔ Form Csrf	
Selector	Ubiquity\security\csrf\generators\Md5Selector
Validator	Ubiquity\security\csrf\generators\RandomValidator
Storage	Ubiquity\security\csrf\storages\SessionTokenStorage

- The service is started in the `services.php` file.

Listing 4: app/config/services.php

```
\Ubiquity\security\csrf\CsrfManager::start();
```

31.3.1 Example of form protection:

The form view:

```
<form id="frm-bar" action="/submit" method="post">
    {{ csrf('frm-bar') }}
    <input type="text" id="sensitiveData" name="sensitiveData">
</form>
```

The `csrf` method generates a token for the form (By adding a hidden field in the form corresponding to the token.).

The form submitting in a controller:

```
use Ubiquity\security\csrf\UCsrfHttp;

#[Post('/submit')]
public function submit(){
    if(UCsrfHttp::isValidPost('frm-bar')){
        //Token is valid! => do something with post datas
    }
}
```

Note: It is also possible to manage this protection via cookie.

31.3.2 Example of protection with ajax:

The meta field `csrf-token` is generated on all pages.

Listing 5: app/controllers/BaseController.php

```
abstract class ControllerBase extends Controller{
    protected $headerView = "@activeTheme/main/vHeader.html";
    protected $footerView = "@activeTheme/main/vFooter.html";

    public function initialize() {
        if (! URequest::isAjax ()) {
            $meta=UCsrfHttp::getTokenMeta('postAjax');
            $this->loadView ( $this->headerView, ['meta'=>$meta] );
        }
    }
}
```

This field is added in the headerView:

Listing 6: app/views/main/vHeader.html

```
{% block header %}
<base href="{{config['siteUrl'] }}">
```

(continues on next page)

(continued from previous page)

```
<meta charset="UTF-8">
<link rel="icon" href="data:;base64,iVBORw0KGgo=">
{{meta | raw}}
<title>Tests</title>
{% endblock %}
```

Example with a button posting data via ajax. The parameter `csrf` is set to `true`. So when the request is posted, the `csrf-token` is sent in the request headers.

```
#[Get(path: "/ajax")]
public function ajax() {
    $this->jquery->postOnClick('#bt', '/postAjax', '{id:55}', '#myResponse', ['csrf'=>
    ↪true]);
    $this->jquery->renderDefaultView();
}
```

The submitting route can check the presence and validity of the token:

```
#[Post(path: "postAjax")]
public function postAjax() {
    if(UCSrfHttp::isValidMeta('postAjax')) {
        var_dump($_POST);
    } else {
        echo 'invalid or absent meta csrf-token';
    }
}
```

31.4 Encryption manager

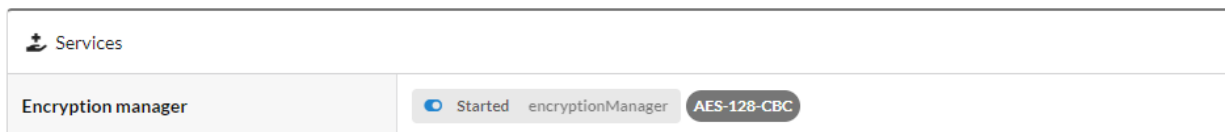
The **EncryptionManager** service can be started directly from the **webtools** interface.

- In this case, a key is generated in the configuration file `app/config/config.php`.
- The service is started in the `services.php` file.

Listing 7: `app/config/services.php`

```
\Ubiquity\security\data\EncryptionManager::start($config);
```

Note: By default, encryption is performed in AES-128.



31.4.1 Changing the cipher:

Upgrade to AES-256:

Listing 8: app/config/services.php

```
\Ubiquity\security\data\EncryptionManager::startProd($config, Encryption::AES256);
```

Generate a new key:

```
Ubiquity new:key 256
```

The new key is generated in the app/config/config.php file.

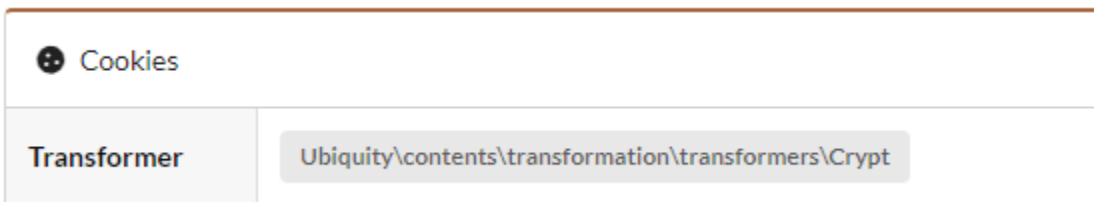
Cookie encryption

Cookies can be encrypted by default, by adding this in services.php:

Listing 9: app/config/services.php

```
use Ubiquity\utils\http\UCookie;
use Ubiquity\contents\transformation\transformers\Crypt;

UCookie::setTransformer(new Crypt());
```



Model data encryption

The Crypt transformer can also be used on the members of a model:

Listing 10: app/models/User.php

```
class Foo{
    #[Transformer(name: "crypt")]
    private $secret;
    ...
}
```

Usage:

```
$o=new Foo();
$o->setSecret('bar');
TransformersManager::transformInstance($o); // secret member is encrypted
```

Generic Data encryption

Strings encryption:

```
$encryptedBar=EncryptionManager::encryptString('bar');
```

To then decrypt it:

```
echo EncryptionManager::decryptString($encryptedBar);
```

It is possible to encrypt any type of data:

```
$encryptedUser=EncryptionManager::encrypt($user);
```

To then decrypt it, with possible serialisation/deserialisation if it is an object:

```
$user=EncryptionManager::decrypt($encryptedUser);
```

31.5 Password management

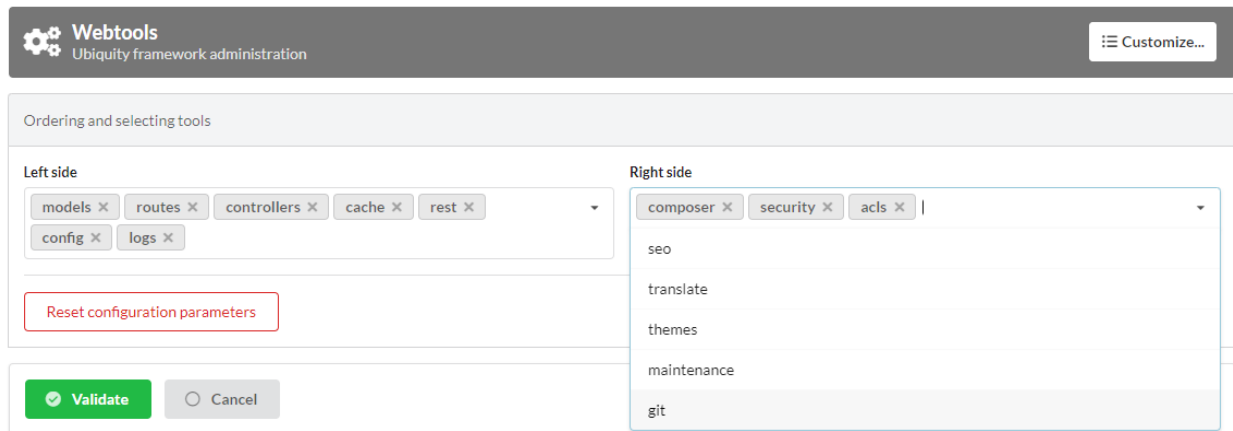
31.6 Users token

32.1 Installation

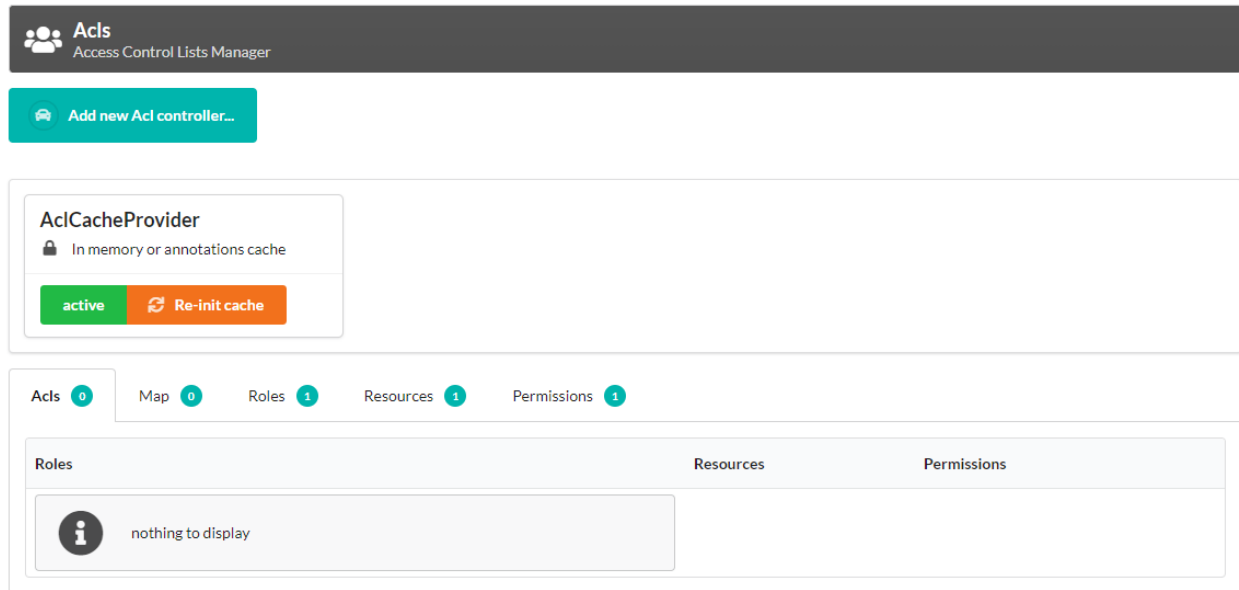
Install the **Ubiquity-acl** module from the command prompt or from the **Webtools** (Composer part).

```
composer require phpmv/ubiquity-acl
```

Then activate the display of the Acl part in the **Webtools**:



ACL interface in **webtools**:



32.2 Acl Rules

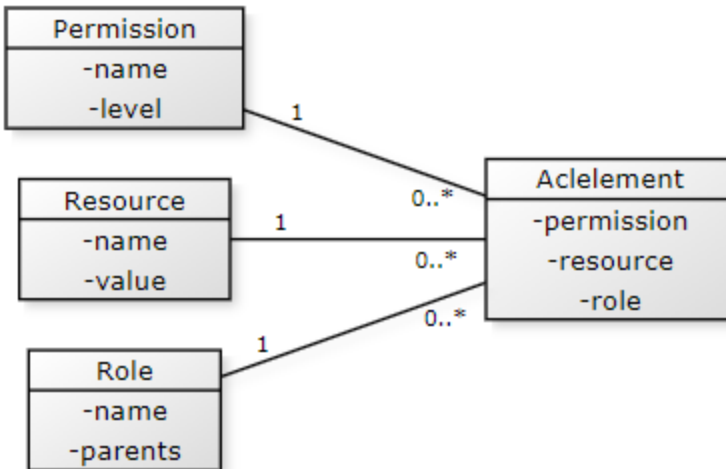
ACLs are used to define access to an Ubiquity application. They are defined according to the following principles:

An Ubiquity application is composed of :

- **Resources** (possibly controllers, or actions of these controllers)
- **Roles**, possibly assigned to users. Each **Role** can inherit parent roles.
- **Permissions**, which correspond to a right to do. Each permission has a level (represented by an integer value).

Additional rules:

- An AclElement (**Allow**) grants Permission to a Role on a Resource.
- Each role inherits authorisations from its parents, in addition to its own.
- If a role has a certain level of access permission on a resource, it will also have all the permissions of a lower level on that resource.
- The association of a resource and a permission to a controller or a controller action defines a **map** element.

**Naming tips:**

- Role, in capital letters, beginning with an arobase (@USER, @ADMIN, @ALL...).
- Permissions, in upper case, named using a verb (READ, WRITE, OPEN...).
- Resource, capitalized on the first letter (Products, Customers...)

32.3 ACL Starting

The **AclManager** service can be started directly from the **webtools** interface, in the **Security** part.

- The service is started in the `services.php` file.

Listing 1: `app/config/services.php`

```
\Ubiquity\security\acl\AclManager::startWithCacheProvider();
```

32.3.1 ACLCacheProvider

This default provider allows you to manage ACLs defined through attributes or annotations.

AclController

An **AclController** enables automatic access management based on ACLs to its own resources. It is possible to create them automatically from **webtools**.

Creating a new Acl controller

controllers\

BaseAclController

☒ Add default view
 ☐ Add route...

Validate

Close

But it is just a basic controller, using the AclControllerTrait feature.

This controller just goes to redefine the `_getRole` method, so that it returns the role of the active user, for example.

Listing 2: app/controllers/BaseAclController.php

```
<?php
namespace controllers;

use Ubiquity\controllers\Controller;
use Ubiquity\security\acl\controllers\AclControllerTrait;
use Ubiquity\attributes\items\acl\Allow;

class BaseAclController extends Controller {
    use AclControllerTrait;

    #[Allow('@ME')]
    public function index() {
        $this->loadView("BaseAclController/index.html");
    }

    public function _getRole() {
        $_GET['role']??'@ME';//Just for testing: logically, this is the active user's_
        ↪role
    }

    /**
     * {@inheritdoc}
     * @see \Ubiquity\controllers\Controller::onInvalidControl()
     */
    public function onInvalidControl() {
        echo $this->_getRole() . ' is not allowed!';
    }
}
```

Authorisation has been granted for the resource:

- Without specifying the resource, the controller's actions are defined as a resource.
- Without specifying the permission, the ALL permission is used.

AcIs 1	Map 1	Roles 2	Resources 2	Permissions 1
Roles	Resources	Permissions		
@ME	BaseAcIController	ALL		

And this association is present in the AcIs map:

AcIs 1	Map 1	Roles 2	Resources 2	Permissions 1
Controller.action	Resource	Permission	Roles	
controllers\BaseAcIController.*	BaseAcIController	ALL	@ME	

Allow with Role, resource and permission

Allow without prior creation:

@USER is allowed to access to Foo resource with READ permission.

Listing 3: app/controllers/BaseAcIController.php

```
use Ubiquity\attributes\items\acl\Allow;

class BaseAcIController extends Controller {
use AcIControllerTrait;
...

#[Allow('@USER', 'Foo', 'READ')]
public function foo(){
    echo 'foo page allowed for @USER and @ME';
}
}
```

Note: The role, resource and permission are automatically created as soon as they are invoked with Allow.

Allow with explicit creation:

Listing 4: app/controllers/BaseAcIController.php

```
use Ubiquity\attributes\items\acl\Allow;
use Ubiquity\attributes\items\acl\Permission;

class BaseAcIController extends Controller {
use AcIControllerTrait;
...

#[Permission('READ', 500)]
#[Allow('@USER', 'Foo', 'READ')]
public function foo(){
    echo 'foo page allowed for @USER and @ME';
}
}
```

Adding ACL at runtime

Whether in a controller or in a service, it is possible to add Roles, Resources, Permissions and Authorizations at runtime:

For example : \ Adding a Role @USER inheriting from @GUEST.

```
use Ubiquity\security\acl\AclManager;

AclManager::addRole('@GUEST');
AclManager::addRole('@USER', ['@GUEST']);
```

32.4 Strategies for defining ACLs

32.4.1 With few resources:

Defining authorisations for each controller's action or action group:

Resources logically correspond to controllers, and permissions to actions. But this rule may not be respected, and an action may be defined as a resource, as required.

The only mandatory rule is that a Controller/action pair can only correspond to one Resource/permission pair (not necessarily unique).

Listing 5: app/controllers/BaseAclController.php

```
namespace controllers;

use Ubiquity\controllers\Controller;
use Ubiquity\security\acl\controllers\AclControllerTrait;
use Ubiquity\attributes\items\acl\Permission;
use Ubiquity\attributes\items\acl\Resource;

#[Resource('Foo')]
#[Allow('@ADMIN')]
class FooController extends Controller {
    use AclControllerTrait;

    #[Allow('@NONE')]
    public function index() {
        echo 'index';
    }

    #[Allow('@USER')]
    public function read() {
        echo 'read';
    }

    #[Allow('@USER')]
    public function write() {
        echo 'write';
    }

    public function admin() {
        echo 'admin';
    }
}
```

(continues on next page)

(continued from previous page)

```

public function _getRole() {
    return $_GET['role']??'@NONE';
}

/**
 * {@inheritdoc}
 * @see \Ubiquity\controllers\Controller::onInvalidControl()
 */
public function onInvalidControl() {
    echo $this->_getRole() . ' is not allowed!';
}
}

```

32.4.2 With more resources:

Listing 6: app/controllers/BaseAclController.php

```

namespace controllers;

use Ubiquity\controllers\Controller;
use Ubiquity\security\acl\controllers\AclControllerTrait;
use Ubiquity\attributes\items\acl\Permission;
use Ubiquity\attributes\items\acl\Resource;

#[Resource('Foo')]
class FooController extends Controller {
    use AclControllerTrait;

    #[Permission('INDEX',1)]
    public function index() {
        echo 'index';
    }

    #[Permission('READ',2)]
    public function read() {
        echo 'read';
    }

    #[Permission('WRITE',3)]
    public function write() {
        echo 'write';
    }

    #[Permission('ADMIN',10)]
    public function admin() {
        echo 'admin';
    }

    public function _getRole() {
        return $_GET['role']??'@NONE';
    }

    /**

```

(continues on next page)

(continued from previous page)

```
* {@inheritdoc}
* @see \Ubiquity\controllers\Controller::onInvalidControl()
*/
public function onInvalidControl() {
    echo $this->_getRole() . ' is not allowed!';
}
}
```


The REST module implements a basic CRUD, with an authentication system, directly testable in the administration part.

33.1 REST and routing

The router is essential to the REST module, since REST (Representation State Transfer) is based on URLs and HTTP methods.

Note: For performance reasons, REST routes are cached independently of other routes. It is therefore necessary to start the router in a particular way to activate the REST routes and not to obtain a recurring 404 error.

The router is started in `services.php`.

Without activation of REST routes:

Listing 1: `app/config/services.php`

```
...
Router::start();
```

To enable REST routes in an application that also has a non-REST part:

Listing 2: `app/config/services.php`

```
...
Router::startAll();
```

To activate only Rest routes:

```
Router::startRest();
```

It is possible to start routing conditionally (this method will only be more efficient if the number of routes is large in either part):

Listing 3: app/config/services.php

```
...
    if($config['isRest']()){
        Router::startRest();
    }else{
        Router::start();
    }
}
```

33.2 Resource REST

A REST controller can be directly associated with a model.

Note: If you do not have a mysql database on hand, you can download this one: `messengerie.sql`

33.2.1 Creation

With devtools:

```
Ubiquity rest RestUsersController -r=User -p=/rest/users
```

Or with webtools:


Go to the **REST** section and choose **Add a new resource**:

(Re-)Init Rest cache

+ Add a new resource

Access token

access token



New resource

Creating a new REST controller...

Controller name *

controllers\ RestUsersController

Base class

Ubiquity\controllers\rest\RestController

Main route path *

/rest/users

Resource

models\User

☒ Re-init Rest cache (recommended)

+ Create new controller

Cancel

The created controller :

Listing 4: app/controllers/RestUsersController.php

```
1 namespace controllers;
2
3 /**
4  * Rest Controller RestUsersController
5  * @route("/rest/users","inherited"=>true,"automated"=>true)
```

(continues on next page)

(continued from previous page)

```

6      * @rest ("resource"=>"models\\User")
7      */
8      class RestUsersController extends \Ubiquity\controllers\rest\RestController {
9
10     }

```

Since the attributes **automated** and **inherited** of the route are set to true, the controller has the default routes of the parent class.

33.2.2 Test interface

Webtools provide an interface for querying datas:

(Re-)Init Rest cache
+ Add a new resource
Access token

models\User

? Rest Controller RestUsers

Controller

controllers\RestUsers

Route

/rest/users

Path	Methods	Action & Parameters	Cache	Exp?
/rest/users/delete/{*}	delete	delete (...keyValues) 🔒	<input type="checkbox"/>	
/rest/users/get/{*}		get (condition, included, useCache)	<input type="checkbox"/>	
/rest/users/getOne/{.+?}/{*}		getOne (keyValues*, included, useCache)	<input type="checkbox"/>	
/rest/users/update/{*}	patch	update (...keyValues) 🔒	<input type="checkbox"/>	
/rest/users/add/	post	add () 🔒	<input type="checkbox"/>	
/rest/users/{index}/?		index ()	<input type="checkbox"/>	
/rest/users/connect/		connect ()	<input type="checkbox"/>	

Getting an instance

A user instance can be accessed by its primary key (**id**):

/rest/users/getOne/(.+?)/(*?)
getOne (keyValues*, included, useCache)
Test

?

Method getOne
Get the first object corresponding to the **\$keyValues**

- **string \$keyValues** primary key(s) value(s) or condition
- **boolean|string \$included** if true, loads associate members with associations, if string, example : client, commands
- **boolean \$useCache** if true then response is cached

/rest/users/getOne/1
GET
Headers...
Parameters...
Send

☐ Use payload

Request headers

Request parameters

Response headers

```

{
  "pragma": " no-cache",
  "date": " Sat, 13 Apr 2019 11",
  "server": " Apache/2.4.38 (Win64) OpenSSL/1.1.1a PHP/7.3.2",
  "x-powered-by": " PHP/7.3.2",
  "x-xdebug-profile-filename": " C",
  "vary": " Accept",
  "content-type": " application/json; charset=utf8",
  "access-control-allow-origin": " http",
  "access-control-max-age": " 86400",
  "cache-control": " no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": " true",
  "connection": " Keep-Alive",
  "keep-alive": " timeout=5, max=99",
  "content-length": " 163",
  "expires": " Thu, 19 Nov 1981 08"
}

```

Response status : OK 200

```

{
  "data": {
    "id": "1",
    "firstname": "Benjamin",
    "lastname": "Shermans",
    "email": "benjamin.sherman@gmail.com",
    "password": "OWC09RSW6AE",
    "suspended": "1",
    "idOrganization": "2"
  }
}

```

Inclusion of associated members: the organization of the user

The screenshot shows a REST client interface with the following components:

- URL Bar:** `/rest/users/getOne/1/organization`
- Method:** `GET`
- Buttons:** Headers..., Parameters..., Send
- Request Headers:** (Empty)
- Request Parameters:** (Empty)
- Response Headers:**

```
{
  "pragma": "no-cache",
  "date": "Sun, 14 Apr 2019 01",
  "server": "Apache/2.4.38 (win64) OpenSSL/1.1.1a PHP/7.3.2",
  "x-powered-by": "PHP/7.3.2",
  "x-debug-profile-filename": "C",
  "vary": "Accept",
  "content-type": "application/json; charset=utf8",
  "access-control-allow-origin": "http",
  "access-control-max-age": "86400",
  "cache-control": "no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": "true",
  "connection": "Keep-Alive",
  "keep-alive": "timeout=5, max=99",
  "content-length": "269",
  "expires": "Thu, 19 Nov 1981 08"
}
```
- Response Status:** OK 200
- Response Body (JSON):**

```
{
  "data": {
    "id": "1",
    "firstname": "Benjamin",
    "lastname": "Shermans",
    "email": "benjamin.sherman@gmail.com",
    "password": "*****",
    "suspended": "1",
    "idOrganization": "2",
    "organization": {
      "id": "2",
      "name": "UNIVERSITÉ DE CAEN-NORMANDIE",
      "domain": "unicaen.fr",
      "aliases": null
    }
  }
}
```

Inclusion of associated members: organization, connections and groups of the user

The screenshot shows a REST client interface with the following components:

- URL Bar:** `/rest/users/getOne/1/true`
- Method:** `GET`
- Buttons:** Headers..., Parameters..., Send
- Request Headers:** (Empty)
- Request Parameters:** (Empty)
- Response Status:** OK 200
- Response Headers:**

```
{
  "pragma": "no-cache",
  "date": "Sun, 14 Apr 2019 01",
  "server": "Apache/2.4.38 (Ubuntu) OpenSSL/1.1.1a PHP/7.3.2",
  "x-powered-by": "PHP/7.3.2",
  "x-xdebug-profile-filename": "c",
  "vary": "Accept",
  "content-type": "application/json; charset=utf8",
  "access-control-allow-origin": "http",
  "access-control-max-age": "86400",
  "cache-control": "no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": "true",
  "connection": "Keep-Alive",
  "keep-alive": "timeout=5, max=99",
  "content-length": "739",
  "expires": "Thu, 19 Nov 1981 08"
}
```
- Response Body (JSON):**

```
{
  "data": {
    "id": "1",
    "firstname": "Benjamin",
    "lastname": "Shermans",
    "email": "benjamin.sherman@gmail.com",
    "password": "*****",
    "suspended": "1",
    "idOrganization": "2",
    "organization": {
      "id": "2",
      "name": "UNIVERSITÉ DE CAEN-NORMANDIE",
      "domain": "unicaen.fr",
      "aliases": null
    }
  },
  "connections": [
    {
      "id": "3",
      "dateCo": "2018-06-04 02:52:12",
      "url": "groupes/2p",
      "idUser": "1"
    },
    {
      "id": "7",
      "dateCo": "2018-06-04 04:25:13",
      "url": "organizations/display/2",
      "idUser": "1"
    },
    {
      "id": "8",
      "dateCo": "2018-06-05 17:00:23",
      "url": "organizations/display/2",
      "idUser": "1"
    },
    {
      "id": "52",
      "dateCo": "2018-06-23 03:24:29",
      "url": "organizations/display/2",
      "idUser": "1"
    }
  ],
  "groupes": [
    {
      "id": "2",
      "name": "Auditeurs",
      "email": "autiteurs",
      "aliases": "ETU;STAGIAIRES;",
      "idOrganization": "1"
    }
  ]
}
```

Getting multiple instances

Getting all instances:

The screenshot shows a REST client interface with the following components:

- URL Bar:** Contains the path `/rest/orgas/get/(.*)`, a `get` method dropdown, a `get (condition, included, useCache)` query string, and a `Test` button.
- Request Section:** Includes a `/rest/orgas/get/` input field, a `GET` method dropdown, `Headers...` and `Parameters...` buttons, and a `Send` button.
- Response Section:** Shows a `Response status: OK 200` message.
- Request Headers:** A section for adding request headers.
- Request Parameters:** A section for adding request parameters.
- Response Headers:** A section displaying the response headers:


```
{
  "pragma": "no-cache",
  "date": "Mon, 15 Apr 2019 01",
  "server": "Apache/2.4.38 (Win64) OpenSSL/1.1.1a PHP/7.3.2",
  "x-powered-by": "PHP/7.3.2",
  "x-debug-profile-filename": "C",
  "vary": "Accept",
  "content-type": "application/json; charset=utf8",
  "access-control-allow-origin": "http",
  "access-control-max-age": "86400",
  "cache-control": "no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": "true",
  "connection": "Keep-Alive",
  "keep-alive": "timeout=5, max=99",
  "content-length": "555",
  "expires": "Thu, 19 Nov 1981 08"
}
```
- Response Body:** A dark-themed area displaying the JSON response:


```
{
  "datas": [
    {
      "id": "1",
      "name": "CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS",
      "domain": "lecnam.net",
      "aliases": "cnam-basse-normandie.fr;cnam.fr"
    },
    {
      "id": "2",
      "name": "UNIVERSITÉ DE CAEN-NORMANDIE",
      "domain": "unicaen.fr",
      "aliases": null
    },
    {
      "id": "3",
      "name": "IUT CAMPUS III",
      "domain": "iut3.unicaen.fr",
      "aliases": "unicaen.fr"
    },
    {
      "id": "4",
      "name": "IUT LISIEUX",
      "domain": "iut.lisieux.unicaen.fr",
      "aliases": "unicaen.fr"
    },
    {
      "id": "30",
      "name": "CNAM",
      "domain": "lecnam.org",
      "aliases": "cnam.org"
    },
    {
      "id": "66",
      "name": "GOOGLE",
      "domain": "google.com",
      "aliases": null
    }
  ]
}
```

Setting a condition:

The screenshot shows a REST client interface with the following components:

- URL bar:** `/rest/orgas/get/name like 'c'`
- Method:** `GET`
- Buttons:** Headers..., Parameters..., Send
- Request section:**
 - ☐ Use payload
 - Request headers (empty)
 - Request parameters (empty)
- Response section:**
 - Response status: OK 200
 - Response headers (JSON):

```
{  "pragma": "no-cache",  "date": "Mon, 15 Apr 2019 01",  "server": "Apache/2.4.38 (Win64) OpenSSL/1.1.1a PHP/7.3.2",  "x-powered-by": "PHP/7.3.2",  "x-debug-profile-filename": "C",  "vary": "Accept",  "content-type": "application/json; charset=utf8",  "access-control-allow-origin": "http",  "access-control-max-age": "86400",  "cache-control": "no-store, no-cache, must-revalidate",  "access-control-allow-credentials": "true",  "connection": "Keep-Alive",  "keep-alive": "timeout=5, max=99",  "content-length": "224",  "expires": "Thu, 19 Nov 1981 08"}

```
 - Response body (JSON):

```
{  "datas": [    {      "id": "30",      "name": "CNAM",      "domain": "lecnam.org",      "aliases": "cnam.org"    },    {      "id": "1",      "name": "CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS",      "domain": "lecnam.net",      "aliases": "cnam-basse-normandie.fr;cnam.fr"    }  ],  "count": 2
}

```

Including associated members:

REST client interface showing a GET request to `/rest/orgas/get/name like 'c'*/groupes`. The response status is OK 200. The response body is a JSON array of objects representing organizational data.

Request:

- Method: GET
- URL: `/rest/orgas/get/name like 'c'*/groupes`
- Headers: (empty)
- Parameters: (empty)

Response:


```
{
  "datas": [
    {
      "id": "30",
      "name": "CNAM",
      "domain": "lecnam.org",
      "aliases": "cnam.org",
      "groupes": []
    },
    {
      "id": "1",
      "name": "CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS",
      "domain": "lecnam.net",
      "aliases": "cnam-basse-normandie.fr;cnam.fr",
      "groupes": [
        {
          "id": "1",
          "name": "Personnels",
          "email": "personnels",
          "aliases": "ALL;",
          "idOrganization": "1"
        },
        {
          "id": "2",
          "name": "Auditeurs",
          "email": "autiteurs",
          "aliases": "ETU;STAGIAIRES;",
          "idOrganization": "1"
        }
      ]
    }
  ],
  "count": 2
}
```

Adding an instance

The datas are sent by the **POST** method, with a content type defined at `application/x-www-form-urlencoded`:

Add name and domain parameters by clicking on the **parameters** button:

Parameters for the GET:/rest/orgas/add/


Get parameters
Enter your parameters.


Parameter name	Parameter value
name	Google
Parameter name	Parameter value
domain	google.com

Add parameter
Add parameters from models\Organization

Validate
Close

The addition requires an authentication, so an error is generated, with the status 401:

/rest/orgas/add/
post
add ()
Test


Method add
Insert a new instance of **\$model**
Require members values in **\$.POST** array
Requires an authorization with access token

/rest/orgas/add/
POST
Headers...
Parameters...
Send

☐ Use payload

Response status : Unauthorized 401

Request headers

Request parameters

Name	Value
name	Google
domain	google.com

Response headers

```
{
  "pragma": "no-cache",
  "date": "Mon, 15 Apr 2019 00",
  "server": "Apache/2.4.38 (Win64) OpenSSL/1.1.1a PHP/7.3.2",
  "x-powered-by": "PHP/7.3.2",
  "x-xdebug-profile-filename": "C",
  "vary": "Accept",
  "content-type": "application/json; charset=utf8",
  "access-control-allow-origin": "http",
  "access-control-max-age": "86400",
  "cache-control": "no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": "true",
  "connection": "Keep-Alive",
  "keep-alive": "timeout=5, max=99",
  "content-length": "1207",
  "code": 401,
  "status": 500,
  "source": {
    "pointer": "C:\\xampp7.3\\htdocs\\verif3\\vendor\\phpmv\\ubiquity\\src\\Ubiquity\\controllers\\rest\\RestBaseController.php",
    "title": "HTTP/1.1 401 Unauthorized, you need an access token for this request",
    "detail": "#0
C:\\xampp7.3\\htdocs\\verif3\\vendor\\phpmv\\ubiquity\\src\\Ubiquity\\controllers\\rest\\RestBaseController.php(52): Ubiquity\\controllers\\rest\\RestBaseController->onInvalidControl()\\n#1
C:\\xampp7.3\\htdocs\\verif3\\vendor\\phpmv\\ubiquity\\src\\Ubiquity\\controllers\\Startup.php(132): Ubiquity\\controllers\\rest\\RestBaseController->_construct()\\n#2
C:\\xampp7.3\\htdocs\\verif3\\vendor\\phpmv\\ubiquity\\src\\Ubiquity\\controllers\\Startup.php(38): Ubiquity\\controllers\\Startup::runAction(Array, true, true)\\n#3
C:\\xampp7.3\\htdocs\\verif3\\vendor\\phpmv\\ubiquity\\src\\Ubiquity\\controllers\\Startup.php(98): Ubiquity\\controllers\\Startup::_preRunAction(Array, true, true)\\n#4
C:\\xampp7.3\\htdocs\\verif3\\vendor\\phpmv\\ubiquity\\src\\Ubiquity\\controllers\\Startup.php(73): Ubiquity\\controllers\\Startup::forward('rest/orgas/add')\\n#5
C:\\xampp7.3\\htdocs\\verif3\\index.php(9): Ubiquity\\controllers\\Startup::run(Array)\\n#6 {main}"
  }
}
```

The administration interface allows you to simulate the default authentication and obtain a token, by requesting the

connect method:

The screenshot shows a REST client interface with the following components:

- URL Bar:** Contains the path `/rest/orgas/connect/` and the method `connect ()`. A `Test` button is on the right.
- Method connect:** A tooltip explains: "Realize the connection to the server. To override in derived classes to define your own authentication."
- Request Bar:** Shows the URL `/rest/orgas/connect/`, the method `GET`, and buttons for `Headers...`, `Parameters...`, and `Send`.
- Use payload:** A checkbox that is currently unchecked.
- Response status:** A green box indicates `OK 200`.
- Request headers:** A section with a header icon and a text input field.
- Request parameters:** A section with a gear icon and a text input field.
- Response headers:** A section displaying the following headers:


```
{
  "date": " Mon, 15 Apr 2019 00",
  "x-powered-by": " PHP/7.3.2",
  "authorization": " Bearer f694f868e96a47181b00",
  "access-control-max-age": " 86400",
  "connection": " Keep-Alive",
  "content-length": " 79",
  "pragma": " no-cache",
  "server": " Apache/2.4.38 (Min64) OpenSSL/1.1.1a PHP/7.3.2",
  "x-xdebug-profile-filename": " C",
  "vary": " Accept",
  "content-type": " application/json; charset=utf8",
  "access-control-allow-origin": " http",
  "cache-control": " no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": " true",
  "keep-alive": " timeout=5, max=98",
}
```
- Response body:** A dark box displaying the following JSON response:


```
{
  "access_token": "f694f868e96a47181b00",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

The token is then automatically sent in the following requests. The record can then be inserted.

/rest/orgas/add/

POST

Headers...

Parameters...

Send

☐ Use payload

Request headers

Request parameters

Name	Value	
name	Google	✕
domain	google.com	✕

Response headers

```
{
  "date": " Mon, 15 Apr 2019 00",
  "x-powered-by": " PHP/7.3.2",
  "authorization": " Bearer f694f868e96a47181b00",
  "access-control-max-age": " 86400",
  "connection": " Keep-Alive",
  "content-length": " 78",
  "pragma": " no-cache",
  "server": " Apache/2.4.38 (Win64) OpenSSL/1.1.1a PHP/7.3.2",
  "x-xdebug-profile-filename": " c",
  "vary": " Accept",
  "content-type": " application/json; charset=utf8",
  "access-control-allow-origin": " http",
  "cache-control": " no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": " true",
  "keep-alive": " timeout=5, max=99",
  "expires": " Thu, 19 Nov 1981 08"
}
```

Response status: OK 200

```
{
  "status": "inserted",
  "data": {
    "name": "Google",
    "domain": "google.com",
    "id": "66"
  }
}
```

Updating an instance

The update follows the same scheme as the insertion.

Deleting an instance

The screenshot shows a REST client interface with the following details:

- URL:** `/rest/orgas/delete/66`
- Method:** `DELETE`
- Response status:** `OK 200`
- Response body (JSON):**

```
{
  "status": "deleted",
  "data": {
    "id": "66",
    "name": "GOOGLE",
    "domain": "google.com",
    "aliases": null,
    "links": {
      "self": "/rest/orgas/get/66"
    }
  }
}
```
- Request headers:** (Empty)
- Request parameters:** (Empty)
- Response headers:**

```
{
  "date": "Tue, 16 Apr 2019 02:",
  "x-powered-by": "PHP/7.3.2",
  "authorization": "Bearer 6ae96e8811bd8b62dc5fce5dcf2177f74a261a9ce24d",
  "access-control-max-age": "86400",
  "connection": "Keep-Alive",
  "content-length": "134",
  "pragma": "no-cache",
  "active-user": "root",
  "server": "Apache/2.4.38 (Win64) OpenSSL/1.1.1a PHP/7.3.2",
  "x-debug-profile-filename": "C",
  "vary": "Accept",
  "content-type": "text/html; charset=UTF-8",
  "access-control-allow-origin": "null",
  "cache-control": "no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": "true",
  "keep-alive": "timeout=5, max=95",
  "expires": "Thu, 19 Nov 1981 08"
}
```

33.2.3 Customizing

Routes

It is of course possible to customize and simplify the routes. In this case, it is preferable to use inheritance from the **RestController** class, and not to enable automatic routes.

Listing 5: `app/controllers/RestOrgas.php`

```

1 namespace controllers;
2
3 use models\Organization;
4
5 /**
6  * Rest Controller for organizations
7  *
8  * @route("/orgas")
9  * @rest
10  */
11 class RestOrgas extends \Ubiquity\controllers\rest\RestController {
12
```

(continues on next page)

(continued from previous page)

```
13 public function initialize() {
14     $this->model = Organization::class;
15     parent::initialize();
16 }
17
18 /**
19  *
20  * @get
21  */
22 public function index() {
23     $this->_get();
24 }
25
26 /**
27  *
28  * @get("{keyValues}")
29  */
30 public function get($keyValues) {
31     $this->_getOne($keyValues);
32 }
33
34 /**
35  *
36  * @post("/")
37  */
38 public function add() {
39     $this->_add();
40 }
41
42 /**
43  *
44  * @patch("{keyValues}")
45  */
46 public function update(...$keyValues) {
47     $this->_update(...$keyValues);
48 }
49
50 /**
51  *
52  * @delete("{keyValues}")
53  */
54 public function delete(...$keyValues) {
55     $this->_delete(...$keyValues);
56 }
57 }
```

After re-initializing the cache, the test interface shows the accessible routes:

<div> <div>?</div> <div>Controller controllers\RestOrgas</div> <div>Route /orgas</div> </div>				
Path	Methods	Action & Parameters	Cache	Exp?
/orgas/	post	add ()	<input type="checkbox"/>	
/orgas/(.*)	delete	delete (...keyValues)	<input type="checkbox"/>	
/orgas/(.+?)/	get	get (keyValues*)	<input type="checkbox"/>	
/orgas/[index]/?	get	index ()	<input type="checkbox"/>	
/orgas/(.*)	patch	update (...keyValues)	<input type="checkbox"/>	

Modification of sent data

33.2.4 By overriding

It is possible to modify the data sent to the update and add methods, in order to add, modify or delete the value of fields before sending. Either by overdefining the method `getDatas`:

Listing 6: app/controllers/RestOrgas.php

```

...

protected function getDatas() {
    $datas = parent::getDatas();
    unset($datas['aliases']); // Remove aliases field
    return $datas;
}

```

33.2.5 With events

Either in a more global way by acting on the rest events:

Listing 7: app/config/services.php

```

use Ubiquity\events\EventsManager;
use Ubiquity\events\RestEvents;
use Ubiquity\controllers\rest\RestBaseController;

...

EventsManager::addListener(RestEvents::BEFORE_INSERT, function ($o, array &$datas,
↳ RestBaseController $resource) {
    unset($datas['aliases']); // Remove aliases field
});

```

33.3 Authentication

Ubiquity REST implements an Oauth2 authentication with Bearer tokens. Only methods with @authorization annotation require the authentication, these are the modification methods (add, update & delete).

```

/**
 * Update an instance of $model selected by the primary key $keyValues
 * Require members values in $_POST array
 * Requires an authorization with access token
 *
 * @param array $keyValues
 * @authorization
 * @route("methods"=>["patch"])
 */
public function update(...$keyValues) {
    $this->_update ( ...$keyValues );
}

```

The **connect** method of a REST controller establishes the connection and returns a new token. It is up to the developer to override this method to manage a possible authentication with login and password.

```

{
  "access_token": "b641bf027617428c6eb6",
  "token_type": "Bearer",
  "expires_in": 3600
}

```

33.3.1 Simulation of a connection with login

In this example, the connection consists simply in sending a user variable by the post method. If the user is provided, the **connect** method of \$server instance returns a valid token that is stored in session (the session acts as a database here).

Listing 8: app/controllers/RestOrgas.php

```

1  namespace controllers;
2
3  use Ubiquity\utils\http\URequest;
4  use Ubiquity\utils\http\USession;
5
6  /**
7   * Rest Controller RestOrgas
8   * @route("/rest/orgas","inherited"=>true,"automated"=>true)
9   * @rest("resource"=>"models\\Organization")
10  */
11  class RestOrgas extends \Ubiquity\controllers\rest\RestController {
12
13      /**
14       * This method simulate a connection.
15       * Send a <b>user</b> variable with <b>POST</b> method to retrieve a_
16  ↪ valid access token
17       * @route("methods"=>["post"])
18       */
19      public function connect() {
20          if(!URequest::isCrossSite()) {
21              if(URequest::isPost()) {
22                  $user=URequest::post("user");
23                  if(isset($user)) {
24                      $tokenInfos=$this->server->connect ();
25                      USession::set($tokenInfos['access_token
26  ↪ '], $user);
27
28                      $tokenInfos['user']=$user;
29                      echo $this->_format($tokenInfos);
30                      return;
31                  }
32              }
33          }
34          throw new \Exception('Unauthorized',401);
35      }
36  }

```

For each request with authentication, it is possible to retrieve the connected user (it is added here in the response headers) :

Listing 9: app/controllers/RestOrgas.php

```

1  namespace controllers;
2
3  use Ubiquity\utils\http\URequest;
4  use Ubiquity\utils\http\USession;
5
6  /**
7   * Rest Controller RestOrgas
8   * @route("/rest/orgas","inherited"=>true,"automated"=>true)
9   * @rest("resource"=>"models\\Organization")
10  */
11  class RestOrgas extends \Ubiquity\controllers\rest\RestController {
12
13      ...
14  }

```

(continues on next page)

(continued from previous page)

```

15     public function isValid($action){
16         $result=parent::isValid($action);
17         if($this->requireAuth($action)){
18             $key=$this->server->_getHeaderToken();
19             $user=USession::get($key);
20             $this->server->_header('active-user',$user,true);
21         }
22         return $result;
23     }
24 }

```

Use the webtools interface to test the connection:

The screenshot displays the webtools interface for testing a REST API. At the top, a method description for 'connect' is shown, indicating it simulates a connection and retrieves a valid access token via POST. The URL bar shows '/rest/orgas/connect/' and the method is set to 'POST'. A 'Send' button is visible. Below the URL bar, there are sections for 'Request headers', 'Request parameters', and 'Response headers'. The 'Request parameters' section shows a single parameter: 'user' with the value 'Snow'. The 'Response status' is 'OK 200'. The 'Response body' is displayed in a dark box, showing a JSON object with the following fields: 'access_token', 'token_type', 'expires_in', and 'user'. The 'Response headers' section shows a list of headers including 'pragma', 'date', 'server', 'x-powered-by', 'x-debug-profile-filename', 'vary', 'content-type', 'access-control-max-age', 'cache-control', 'access-control-allow-credentials', 'authorization', 'connection', 'keep-alive', 'content-length', and 'expires'.

33.4 Customizing

33.4.1 Api tokens

It is possible to customize the token generation, by overriding the `getRestServer` method:

Listing 10: app/controllers/RestOrgas.php

```

1 namespace controllers;
2
3 use Ubiquity\controllers\rest\RestServer;
4 class RestOrgas extends \Ubiquity\controllers\rest\RestController {
5
6     ...
7
8     protected function getRestServer(): RestServer {
9         $srv= new RestServer($this->config);
10        $srv->setTokenLength(32);
11        $srv->setTokenDuration(4800);
12        return $srv;
13    }
14 }

```

33.4.2 Allowed origins and CORS

Cross-Origin Resource Sharing (CORS)

If you access your api from another site, it is necessary to set up **CORS**.

In this case, for requests of type PATCH, PUT, DELETE, your api must define a route allowing CORS to carry out its control pre-request using the OPTIONS method.

Listing 11: app/controllers/RestOrgas.php

```

1 class RestOrgas extends \Ubiquity\controllers\rest\RestController {
2
3     ...
4
5     /**
6      * @options('{url}')
7      */
8     public function options($url='') {}
9 }

```

Allowed origins

Allowed origins allow to define the clients that can access the resource in case of a cross domain request by defining The **Access-Control-Allow-Origin** response header. This header field is returned by the OPTIONS method.

Listing 12: app/controllers/RestOrgas.php

```

1 class RestOrgas extends \Ubiquity\controllers\rest\RestController {
2
3     ...
4
5     protected function getRestServer(): RestServer {
6         $srv= new RestServer($this->config);
7         $srv->setAllowOrigin('http://mydomain/');
8         return $srv;
9 }

```

(continues on next page)

(continued from previous page)

```

9         }
10    }

```

It is possible to authorize several origins:

Listing 13: app/controllers/RestOrgas.php

```

1    class RestOrgas extends \Ubiquity\controllers\rest\RestController {
2
3        ...
4
5        protected function getRestServer(): RestServer {
6            $srv= new RestServer($this->config);
7            $srv->setAllowOrigins(['http://mydomain1/', 'http://mydomain2/']);
8            return $srv;
9        }
10    }

```

33.4.3 Response

To change the response format, it is necessary to create a class inheriting from ResponseFormatter. We will take inspiration from **HAL**, and change the format of the responses by:

- adding a link to self for each resource
- adding an `_embedded` attribute for collections
- removing the data attribute for unique resources

Listing 14: app/controllers/RestOrgas.php

```

1    namespace controllers\rest;
2
3    use Ubiquity\controllers\rest\ResponseFormatter;
4    use Ubiquity\orm\OrmUtils;
5
6    class MyResponseFormatter extends ResponseFormatter {
7
8        public function cleanRestObject($o, &$classname = null) {
9            $pk = OrmUtils::getFirstKeyValue ( $o );
10           $r=parent::cleanRestObject ($o);
11           $r["links"]=[ "self"=>"/rest/orgas/get/".$pk];
12           return $r;
13       }
14
15       public function getOne($datas) {
16           return $this->format ( $this->cleanRestObject ( $datas ) );
17       }
18
19       public function get($datas, $pages = null) {
20           $datas = $this->getDatas ( $datas );
21           return $this->format ( [ "_embedded" => $datas, "count" =>_
22               =>\sizeof ( $datas ) ] );
23       }
24   }

```

Then assign `MyResponseFormatter` to the REST controller by overriding the `getResponseFormatter` method:

Listing 15: `app/controllers/RestOrgas.php`

```
1 class RestOrgas extends \Ubiquity\controllers\rest\RestController {
2
3     ...
4
5     protected function getResponseFormatter(): ResponseFormatter {
6         return new MyResponseFormatter();
7     }
8 }
```

Test the results with the `getOne` and `get` methods:

```
{
  "id": "1",
  "name": "CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS",
  "domain": "lecnam.net",
  "aliases": "cnam-basse-normandie.fr;cnam.fr",
  "links": {
    "self": "/rest/orgas/get/1"
  }
}
```

```
{
  "_embedded": [
    {
      "id": "30",
      "name": "CNAM",
      "domain": "lecnam.org",
      "aliases": "cnam.org",
      "links": {
        "self": "/rest/orgas/get/30"
      }
    },
    {
      "id": "1",
      "name": "CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS",
      "domain": "lecnam.net",
      "aliases": "cnam-basse-normandie.fr;cnam.fr",
      "links": {
        "self": "/rest/orgas/get/1"
      }
    }
  ],
  "count": 2
}
```

33.5 APIs

Unlike REST resources, APIs controllers are multi-resources.

33.5.1 SimpleRestAPI

33.5.2 JsonApi

Ubiquity implements the jsonApi specification with the class `JsonApiRestController`. JsonApi is used by [EmberJS](https://emberjs.com/) and others. see <https://jsonapi.org/> for more.

Creation


With devtools:

```
Ubiquity restapi JsonApiTest -p=/jsonapi
```

Or with webtools:

Go to the **REST** section and choose **Add a new resource**:

(Re-)Init Rest cache
+ Add a new resource
Access token
8a42d6733b126b0fb65i


New resource
 Creating a new REST controller...

Controller name *
 Base class

Main route path *


☒ Re-init Rest cache (recommended)


+ Create new controller
Cancel










Test the api in webtools:

(Re-)Init Rest cache
+ Add a new resource
Access token

JsonAPI 1.0

 Controller
controllers\JsonApiTest

 Route
/jsonapi

Path	Methods	Action & Parameters	Cache	Exp?
 /jsonapi/{*?}	options	options (...resource)	<input type="checkbox"/>	
 /jsonapi/links/	get	index ()	<input type="checkbox"/>	
 /jsonapi/{.+?}/{.+?}/relationships/{.+?}/	get	getRelationShip_ (resource*, id*, member*)	<input type="checkbox"/>	
 /jsonapi/{.+?}/{.+?}/	get	getOne_ (resource*, id*)	<input type="checkbox"/>	
 /jsonapi/{.+?}/	get	getAll_ (resource*)	<input type="checkbox"/>	
 /jsonapi/{.+?}/	post	add_ (resource*)	<input type="checkbox"/>	
 /jsonapi/{.+?}/{.*?}	patch	update_ (resource*, ...id)	<input type="checkbox"/>	
 /jsonapi/{.+?}/{.*?}/	delete	delete_ (resource*, ...id)	<input type="checkbox"/>	
 /jsonapi/connect/		connect ()	<input type="checkbox"/>	

Links

The **links** route (index method) returns the list of available urls:

The screenshot shows a REST client interface with the following components:

- URL bar:** `/jsonapi/links/`
- Method:** `GET`
- Buttons:** Headers..., Parameters..., Send
- Use payload:** ☐
- Request headers:** (empty)
- Request parameters:** (empty)
- Response status:** OK 200
- Response headers:**

```
{
  "pragma": "no-cache",
  "date": "Tue, 16 Apr 2019 01",
  "server": "Apache/2.4.38 (Win64) OpenSSL/1.1.1a PHP/7.3.2",
  "x-powered-by": "PHP/7.3.2",
  "x-xdebug-profile-filename": "C",
  "vary": "Accept",
  "content-type": "application/vnd.api+json; charset=utf8",
  "access-control-allow-origin": "*",
  "access-control-max-age": "86400",
  "cache-control": "no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": "true",
  "connection": "Keep-Alive",
  "keep-alive": "timeout=5, max=99",
  "content-length": "497",
  "expires": "Thu, 19 Nov 1981 08"
}
```
- Response body:**

```
{
  "links": [
    {
      "method": "options",
      "url": "/jsonapi/{resource}"
    },
    {
      "method": "get",
      "url": "/jsonapi/links/"
    },
    {
      "method": "get",
      "url": "/jsonapi/{resource}/{id}/relationships/{member}/"
    },
    {
      "method": "get",
      "url": "/jsonapi/{resource}/{id}/"
    },
    {
      "method": "get",
      "url": "/jsonapi/{resource}/"
    },
    {
      "method": "post",
      "url": "/jsonapi/{resource}/"
    },
    {
      "method": "patch",
      "url": "/jsonapi/{resource}/{id}"
    }
  ]
}
```

Getting an array of objects

By default, all associated members are included:

The screenshot shows a REST client interface with the following details:

- URL:** `/jsonapi/users`
- Method:** GET
- Response status:** OK 200
- Request headers:** (empty)
- Request parameters:** (empty)
- Response headers:**

```
{
  "date": "Wed, 17 Apr 2019 00:",
  "x-powered-by": "PHP/7.3.2",
  "transfer-encoding": "chunked",
  "access-control-max-age": "86400",
  "connection": "Keep-Alive",
  "pragma": "no-cache",
  "server": "Apache/2.4.38 (Ubuntu) OpenSSL/1.1.1a PHP/7.3.2",
  "x-debug-profile-filename": "C",
  "vary": "Accept",
  "access-control-allow-methods": "GET, POST, OPTIONS, PUT, DELETE",
  "content-type": "application/vnd.api+json; charset=utf-8",
  "access-control-allow-origin": "*",
  "cache-control": "no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": "true",
  "keep-alive": "timeout=5, max=99",
  "expires": "Thu, 19 Nov 1981 08:"
}
```
- Response body (JSON):**

```
{
  "data": [
    {
      "id": "1",
      "type": "user",
      "attributes": {
        "firstname": "Benjamin",
        "lastname": "Shermans",
        "email": "benjamin.sherman@gmail.com",
        "password": "*****",
        "suspended": "1"
      },
      "links": {
        "self": "/jsonapi/user/1/"
      },
      "relationships": {
        "organization": {
          "data": {
            "id": "2",
            "type": "organization"
          },
          "links": [
            "/jsonapi/user/1/organization/",
            "/jsonapi/organization/2/"
          ]
        }
      },
      "included": {
        "organization": {
          "id": "2",

```

Including associated members

you need to use the **include** parameter of the request:

URL	Description
<code>/jsonapi/user?include=false</code>	No associated members are included
<code>/jsonapi/user?include=organization</code>	Include the organization
<code>/jsonapi/user?include=organization,connections</code>	Include the organization and the connections
<code>/jsonapi/user?include=groupes.organization</code>	Include the groups and their organization

Filtering instances

you need to use the **filter** parameter of the request, **filter** parameter corresponds to the **where** part of an SQL statement:

URL	Description
<code>/jsonapi/user?l=1</code>	No filtering
<code>/jsonapi/user?firstname='Benjamin'</code>	Returns all users named Benjamin
<code>/jsonapi/user?filter=firstname like 'B*'</code>	Returns all users whose first name begins with a B
<code>/jsonapi/user?filter=suspended=0 and lastname like 'ca*'</code>	Returns all suspended users whose lastname begins with ca

Pagination

you need to use the **page[number]** and **page[size]** parameters of the request:


URL	Description
/jsonapi/user	No pagination
/jsonapi/user?page[number]=1	Display the first page (page size is 1)
/jsonapi/user?page[number]=1&&page[size]=10	Display the first page (page size is 10)

Adding an instance

The datas, contained in `data[attributes]`, are sent by the **POST** method, with a content type defined at `application/json; charset=utf-8`.

Add your parameters by clicking on the **parameters** button:

Parameters for the POST:/jsonapi/organization

**Post parameters**
Enter your parameters.

Parameter name

data

Parameter value

`{'attributes':{'name':'phpMv','domain':'kobject.net'}}`

✕

Add parameter

Validate

Close

The addition requires an authentication, so an error is generated, with the status 401 if the token is absent or expired.

The screenshot shows a REST client interface with the following details:

- URL:** `/jsonapi/organizations/`
- Method:** `POST`
- Buttons:** Headers..., Parameters..., Send
- Request:**
 - ☒ Use payload
 - Request headers:** (empty)
 - Request parameters:**

Name	Value
data	{attributes:{name:phpl
 - Response headers:**

```
{
  "date": " Wed, 17 Apr 2019 00",
  "x-powered-by": " PHP/7.3.2",
  "authorization": " Bearer 08291c344c534473b05b",
  "access-control-max-age": " 86400",
  "connection": " Keep-Alive",
  "content-length": " 164",
  "pragma": " no-cache",
  "server": " Apache/2.4.38 (win64) OpenSSL/1.1.1a PHP/7.3.2",
  "x-xdebug-profile-filename": " C",
  "vary": " Accept",
  "access-control-allow-methods": " GET, POST, OPTIONS, PUT, DELE",
  "content-type": " application/vnd.api+json; charset=utf8",
  "access-control-allow-origin": " *",
  "cache-control": " no-store, no-cache, must-revalidate",
  "access-control-allow-credentials": " true",
  "keep-alive": " timeout=5, max=99",
  "expires": " Thu, 19 Nov 1981 08"
}
```
- Response:**
 - Status:** OK 200
 - Body:**

```
{
  "status": "inserted",
  "data": {
    "id": "32",
    "type": "organization",
    "attributes": {
      "name": "phpMv",
      "domain": "kobject.net"
    },
    "links": {
      "self": "/jsonapi/organization/32/"
    }
  }
}
```

Deleting an instance

Deletion requires the **DELETE** method, and the use of the **id** of the object to be deleted:

The screenshot shows a REST client interface with the following components:

- URL Bar:** Contains the URL `/jsonapi/(.+?)/(*?)/` and a `delete` button.
- Method and Path:** The method is `DELETE` and the path is `/jsonapi/organizations/32/`.
- Buttons:** Includes `Headers...`, `Parameters...`, and a `Send` button.
- Response Status:** A green box indicates `Response status: OK 200`.
- Request Headers:** A section for adding request headers.
- Request Parameters:** A section for adding request parameters.
- Response Headers:** A section displaying the response headers, including:
 - `date: "Wed, 17 Apr 2019 00"`
 - `"x-powered-by": " PHP/7.3.2"`
 - `authorization: " Bearer 08291c344c534473b05b"`
 - `access-control-max-age: " 86400"`
 - `connection: " Keep-Alive"`
 - `content-length: " 178"`
 - `pragma: " no-cache"`
 - `server: " Apache/2.4.38 (Win64) OpenSSL/1.1.1a PHP/7.3.2"`
 - `"x-xdebug-profile-filename": " C"`
 - `vary: " Accept"`
 - `access-control-allow-methods: " GET, POST, OPTIONS, PUT, DELETE"`
 - `content-type: " application/vnd.api+json; charset=utf8"`
 - `access-control-allow-origin: " "`
 - `cache-control: " no-store, no-cache, must-revalidate"`
 - `access-control-allow-credentials: " true"`
 - `keep-alive: " timeout=5, max=99"`
 - `expires: " Thu, 19 Nov 1981 08"`
- Response Body:** A dark box displaying the JSON response:

```
{
  "status": "deleted",
  "data": {
    "id": "32",
    "type": "organization",
    "attributes": {
      "name": "PHPMV",
      "domain": "kobject.net",
      "aliases": null
    },
    "links": {
      "self": "/jsonapi/organization/32/"
    }
  }
}
```

Note: Webtools allow you to manage an Ubiquity application via a web interface. Since **Ubiquity 2.2.0**, webtools are in a separate [repository](#).

34.1 Installation

Update the devtools if necessary to get started:

```
composer global update
```

34.1.1 At the project creation

Create a projet with **webtools** (`-a` option)

```
Ubiquity new quick-start -a
```

34.1.2 In an existing project

In a console, go to the project folder and execute:


```
Ubiquity admin
```

34.2 Starting


Start the embedded web server, from the project folder:

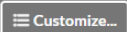
Ubiquity serve


go to the address: `http://127.0.0.1:8090/Admin`


 UbiquityMyadmin


models routes controllers cache rest config git seo logs translate themes maintenance


 **Web-tools**
Ubiquity framework administration


 Customize...


 **Models**
Used to perform CRUD operations on data.


 **Routes**
Displays defined routes with annotations


 **Controllers**
Displays controllers and actions


 **Cache**
Annotations, models, router and controller cache


 **Rest**
Restfull web service


 **Config**
Configuration variables

 **Git**
Git versioning

 **Seo**
Search Engine Optimization


 **Logs**
Log files

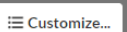
 **Translate**
Translation module

 **Themes**
Themes module


34.3 Customizing

Click on **customize** to display only the tools you use:

 **Web-tools**
Ubiquity framework administration

 Customize...

Ordering and selecting tools

 **Customizing**
You can select and re-order your tools.
To re-order or move a tool to another side, the tools must be de-selected and then selected in the desired order.


Left side

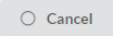
models × routes × controllers ×


Right side


config × cache ×


Reset configuration parameters


 Validate


 Cancel


 UbiquityMyadmin
 models routes controllers config cache


 **Web-tools**
Ubiquity framework administration
 Customize...

 **Models**
Used to perform CRUD operations on data.

 **Config**
Configuration variables

 **Routes**
Displays defined routes with annotations

 **Cache**
Annotations, models, router and controller cache

 **Controllers**
Displays controllers and actions

34.4 Webtools modules

34.4.1 Routes

 **Routes**
Displays defined routes with annotations

Displays default (non REST) routes.

Operations:

- Filter routes
- Test routes (GET, POST...)
- Initialize router cache

34.4.2 Controllers

 **Controllers**
Displays controllers and actions

Displays non REST controllers.

Operations:

- Create a controller (and optionally the view associated to the default **index** action)
- Create an action in a controller (optionally the associated view, the associated route)
- Create a special controller (CRUD or Auth)
- Test an action (GET, POST...)

34.4.3 Models



Models

Used to perform CRUD operations on data.

Displays the metadatas of the models, allows to browse the entities.

Operations:

- Create models from database
- Generate models cache
- Generate database script from existing models
- Performs CRUD operations on models

34.4.4 Rest



Rest

Restfull web service

Displays an manage REST services.

Operations:

- Re-initialize Rest cache and routes
- Create a new Service (using an api)
- Create a new resource (associated to a model)
- Test and query a web service using http methods
- Performs CRUD operations on models

34.4.5 Cache



Cache

Annotations, models, router and controller cache

Displays cache files.

Operations:

- Delete or re-initialize models cache
- Delete or re-initialize controllers cache
- Delete other cache files

34.4.6 Maintenance



Maintenance

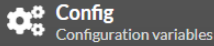
Manages maintenance modes

Allows to manage maintenance modes.

Operations:

- Create or update a maintenance mode
- De/Activate a maintenance mode
- Delete a maintenance mode

34.4.7 Config



Allows the display and modification of the app configuration.

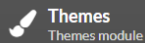
34.4.8 Git



Synchronizes the project using git.

Operations:

- Configuration with external repositories
- Commit
- Push
- Pull



Manages Css themes.

Operations:

- Install an existing theme
- Activate a theme
- Create a new theme (eventually base on an existing theme)

35.1 System requirements

Before working on Ubiquity, setup your environment with the following software:

- Git
- PHP version 7.1 or above.

35.2 Get Ubiquity source code

On [Ubiquity github repository](#) :

- Click *Fork* Ubiquity project
- Clone your fork locally:

```
git clone git@github.com:USERNAME/ubiquity.git
```

35.3 Work on your Patch

Note: Before you start, you must know that all the patches you are going to submit must be released under the Apache 2.0 license, unless explicitly specified in your commits.

35.3.1 Create a Topic Branch

Note: Use a descriptive name for your branch:

- `issue_xxx` where `xxx` is the issue number is a good convention for bug fixes
- `feature_name` is a good convention for new features

```
git checkout -b NEW_BRANCH_NAME master
```

35.3.2 Work on your Patch

Work on your code and commit as much as you want, and keep in mind the following:

- Read about the *Ubiquity coding standards*;
- Add unit, functional or acceptance tests to prove that the bug is fixed or that the new feature actually works;
- Do atomic and logically separate commits (use *git rebase* to have a clean and logical history);
- Write good commit messages (see the tip below).
- Increase the version numbers in any modified files, respecting *semver* rules:

Given a version number `MAJOR.MINOR.PATCH`, increment the:

- `MAJOR` version when you make incompatible API changes,
- `MINOR` version when you add functionality in a backwards-compatible manner, and
- `PATCH` version when you make backwards-compatible bug fixes.

35.4 Submit your Patch

Update the [Unrelease] part of the `CHANGELOG.md` file by integrating your changes into the appropriate parts:

- Added
- Changed
- Removed
- Fixed

Eventually rebase your Patch Before submitting, update your branch (needed if it takes you a while to finish your changes):

```
git checkout master
git fetch upstream
git merge upstream/master
git checkout NEW_BRANCH_NAME
git rebase master
```

35.5 Make a Pull Request

You can now make a pull request on [Ubiquity github repository](#) .

Note: Although the framework is very recent, please note some early Ubiquity classes do not fully follow this guide and have not been modified for backward compatibility reasons. However all new codes must follow this guide.

36.1 Design choices

36.1.1 Fetching and using Services

Dependency injections

Avoid using dependency injection for all parts of the framework, internally. Dependency injection is a resource-intensive mechanism:

- it needs to identify the element to instantiate ;
- then to proceed to its instantiation ;
- to finally assign it to a variable.

Getting services from a container

Also avoid public access to services registered in a service container. This type of access involves manipulating objects whose return type is unknown, not easy to handle for the developer.

For example, It's hard to manipulate the untyped return of `$this->serviceContainer->get('translator')`, as some frameworks allow, and know which methods to call on it.

When possible, and when it does not reduce flexibility too much, the use of static classes is suggested:

For a developer, the `TranslatorManager` class is accessible from an entire project without any object instantiation. It exposes its public interface and allows code completion:

- The translator does not need to be injected to be used;
- It does not need to be retrieved from a service container.

The use of static classes inevitably creates a strong dependency and affects flexibility. But to come back to the Translator example, there is no reason to change it if it is satisfying. It is not desirable to want to provide flexibility at all costs when it is not necessary, and then for the user to see that its application is a little slow.

36.2 Optimization

Execution of each line of code can have significant performance implications. Compare and benchmark implementation solutions, especially if the code is repeatedly called:

- Identify these repetitive and expensive calls with php profiling tools ([Blackfire profiler](#) , [Tideways](#) ...)
- Benchmark your different implementation solutions with [phpMyBenchmarks](#)

36.3 Code quality

Ubiquity use [Scrutinizer-CI](#) for code quality.

- For classes and methods :
 - A or B evaluations are good
 - C is acceptable, but to avoid if possible
 - The lower notes are to be prohibited

36.3.1 Code complexity

- Complex methods must be split into several, to facilitate maintenance and allow reuse;
- For complex classes , do an extract-class or extract-subclass refactoring and split them using Traits;

36.3.2 Code duplications

Absolutely avoid duplication of code, except if duplication is minimal, and is justified by performance.

36.3.3 Bugs

Try to solve all the bugs reported as you go, without letting them accumulate.

36.4 Tests

Any bugfix that doesn't include a test proving the existence of the bug being fixed, may be suspect. Ditto for new features that can't prove they actually work.

It is also important to maintain an acceptable coverage, which may drop if a new feature is not tested.

36.5 Code Documentation

The current code is not yet fully documented, feel free to contribute in order to fill this gap.

36.6 Coding standards

Ubiquity coding standards are mainly based on the [PSR-1](#) , [PSR-2](#) and [PSR-4](#) standards, so you may already know most of them. The few intentional exceptions to the standards are normally reported in this guide.

36.6.1 Naming Conventions

- Use camelCase for PHP variables, members, function and method names, arguments (e.g. `$modelsCacheDirectory`, `isStarted()`);
- Use namespaces for all PHP classes and UpperCamelCase for their names (e.g. `CacheManager`);
- Prefix all abstract classes with `Abstract` except PHPUnit BaseTests;
- Suffix interfaces with `Interface`;
- Suffix traits with `Trait`;
- Suffix exceptions with `Exception`;
- Suffix core classes manager with `Manager` (e.g. `CacheManager`, `TranslatorManager`);
- Prefix Utility classes with `U` (e.g. `UString`, `URequest`);
- Use UpperCamelCase for naming PHP files (e.g. `CacheManager.php`);
- Use uppercase for constants (e.g. `const SESSION_NAME='Ubiquity'`).

36.6.2 Indentation, tabs, braces

- Use Tabs, not spaces; (!PSR-2)
- Use brace always on the same line; (!PSR-2)
- Use braces to indicate control structure body regardless of the number of statements it contains;

36.6.3 Classes

- Define one class per file;
- Declare the class inheritance and all the implemented interfaces on the same line as the class name;
- Declare class properties before methods;
- Declare private methods first, then protected ones and finally public ones;
- Declare all the arguments on the same line as the method/function name, no matter how many arguments there are;
- Use parentheses when instantiating classes regardless of the number of arguments the constructor has;
- Add a use statement for every class that is not part of the global namespace;

36.6.4 Operators

- Use identical comparison and equal when you need type juggling;

Example

```
<?php
namespace Ubiquity\namespace;

use Ubiquity\othernamespace\Foo;

/**
 * Class description.
 * Ubiquity\namespace$Example
 * This class is part of Ubiquity
 *
 * @author authorName <authorMail>
 * @version 1.0.0
 * @since Ubiquity x.x.x
 */
class Example {
    /**
     * @var int
     *
     */
    private $theInt = 1;

    /**
     * Does something from **a** and **b**
     *
     * @param int $a The a
     * @param int $b The b
     */
    function foo($a, $b) {
        switch ($a) {
            case 0 :
                $Other->doFoo ();
                break;
            default :
                $Other->doBaz ();
        }
    }

    /**
     * Adds some values
     *
     * @param param V $v The v object
     */
    function bar($v) {
        for($i = 0; $i < 10; $i++) {
            $v->add ( $i );
        }
    }
}
```

Important:

You can import this standardization files that integrates all these rules in your IDE:

- Eclipse
- PhpStorm

If your preferred IDE is not listed, you can submit the associated standardization file by creating a new PR.

Documenting guide

Ubiquity has two main sets of documentation:

- the guides, which help you learn about manipulations or concepts ;
- and the API, which serves as a reference for coding.

You can help improve the Ubiquity guides by making them more coherent, consistent, or readable, adding missing information, correcting factual errors, fixing typos, or bringing them up to date with the latest Ubiquity version.

To do so, make changes to Ubiquity guides source files (located here on GitHub). Then open a pull request to apply your changes to the master branch.

When working with documentation, please take into account the guidelines.

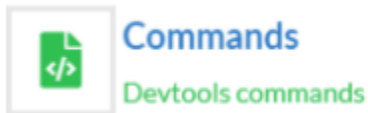
CHAPTER 38

Ubiquity commands

Note: This part is accessible from the **webtools**, so if you created your project with the **-a** option or with the **create-project** command..

38.1 Commands

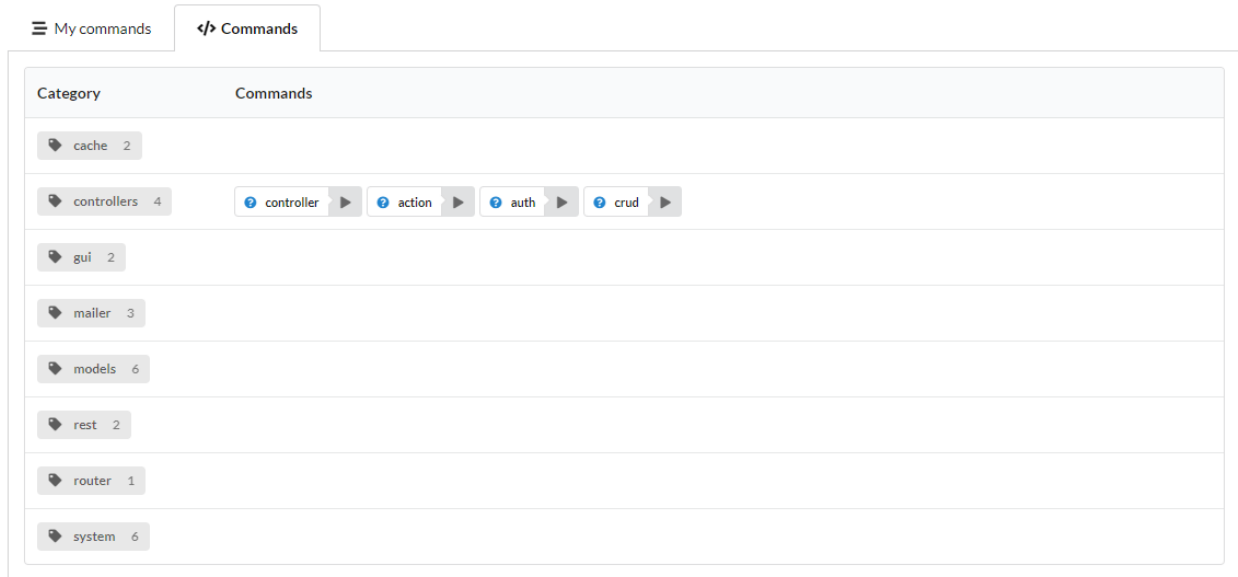
From the webtools, activate the **commands** part,



or go directly to <http://127.0.0.1:8090/Admin/commands>.

38.1.1 Commands list

Activate the **Commands** tab to get the list of existing devtools commands.



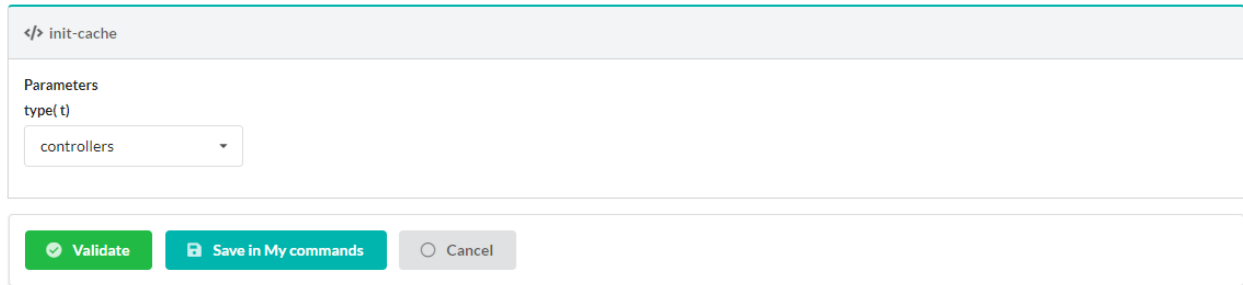
38.1.2 Command info

It is possible to get help on a command (which produces a result equivalent to `Ubiquity help cmdName`).

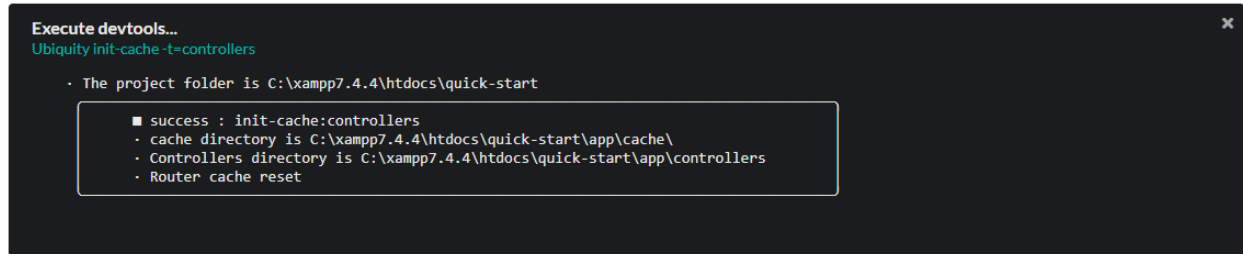


38.1.3 Command execution

Clicking on the run button of a command displays a form to enter the parameters (or executes it directly if it takes none).



After entering the parameters, the execution produces a result.

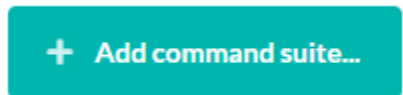


38.2 Commands suite

Return to **My commands tab**: It is possible to save a sequence of commands (with stored parameters), and then execute the same sequence:

38.2.1 Suite creation

Click on the **add command suite**



Add the desired commands and modify the parameters:

☰ suite #0

New command name

▼

+

Add command

Name

controller-cache-init

</> Ubiquity init-cache

Parameters

type(t)

controllers

</> Ubiquity info:routes

Parameters

type(t)

limit(l)

offset(o)

search(s)

method(m)

routes

|

0

s

✔ Validate

○ Cancel

The validation generates the suite:

☰ My commands

</> Commands

name	commandValues
controller-cache-init	<div><div></> Ubiquityinit-cache-t=controllers ▶</div><div></> Ubiquityinfo:routes-t=routes ▶</div><div><div>✎</div><div>✖</div><div>▶</div></div></div>

38.2.2 Commands suite execution

Clicking on the run button of the suite executes the list of commands it contains:

Execute devtools...

Ubiquity init-cache -t=controllers

The project folder is C:\xampp7.4.4\htdocs\quick-start

■ success : init-cache:controllers

· cache directory is C:\xampp7.4.4\htdocs\quick-start\app\cache\

· Controllers directory is C:\xampp7.4.4\htdocs\quick-start\app\controllers

· Router cache reset

Ubiquity info:routes -t=routes

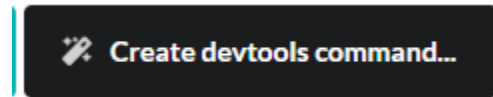
The project folder is C:\xampp7.4.4\htdocs\quick-start

Nothing to display

· 0 routes (routes)

38.3 Custom command creation

Click on the **Create devtools command** button.



Enter the characteristics of the new command:

- The command name
- The command value: name of the main argument
- The command parameters: In case of multiple parameters, use comma as separator
- The command description
- The command aliases: In case of multiple aliases, use comma as separator

create-command

commandName *

createArray

Parameters

value(v)

parameters(p)

description(d)

aliases(a)

jsonValue

f

Creates an array from JSON and save t

createarray,arrayFromJson

Validate

Cancel

Note: Custom commands are created in the **app/commands** folder of the project.

```

Execute devtools...
Ubiquity create-command createArray -v=jsonValue -p=f -d="Creates an array from JSON and save to file" -a=createarray,arrayFromJson

• The project folder is C:\xampp7.4\htdocs\quick-start

  ■ success : Command creation
  • Command createArray created in C:\xampp7.4\htdocs\quick-start\commands\CreateArray.cmd.php!

• Command createArray find by name

  ■ createArray [jsonValue] =>
  • Creates an array from JSON and save to file
  • Aliases : createarray,arrayFromJson
  • Parameters :
    -f          shortcut of --fLongName
                The f description.

  × Samples :
    Sample use of createArray
    • Ubiquity createArray jsonValue
  
```

The generated class:

Listing 1: app/commands/CreateArray.php

```

1 namespace commands;
2
3 use Ubiquity\devtools\cmd\commands\AbstractCustomCommand;
4 use Ubiquity\devtools\cmd\ConsoleFormatter;
5 use Ubiquity\devtools\cmd\Parameter;
6
7 class CreateArray extends AbstractCustomCommand {
8
9     protected function getValue(): string {
10         return 'jsonValue';
11     }
12
13     protected function getAliases(): array {
14         return array("createarray", "arrayFromJson");
15     }
16
17     protected function getName(): string {
18         return 'createArray';
19     }
20
21     protected function getParameters(): array {
22         return ['f' => Parameter::create('fLongName', 'The f description.', [])];
23     }
24
25     protected function getExamples(): array {
26         return ['Sample use of createArray'=>'Ubiquity createArray jsonValue'];
27     }
28
29     protected function getDescription(): string {
30         return 'Creates an array from JSON and save to file';
31     }
32
33     public function run($config, $options, $what, ...$otherArgs) {
34         //TODO implement command behavior
35         echo ConsoleFormatter::showInfo('Run createArray command');
36     }
37 }

```

The **CreateArray** command implemented:

Listing 2: app/commands/CreateArray.php

```

1 namespace commands;
2
3 use Ubiquity\devtools\cmd\commands\AbstractCustomCommand;
4 use Ubiquity\devtools\cmd\ConsoleFormatter;
5 use Ubiquity\devtools\cmd\Parameter;
6 use Ubiquity\utils\base\UFileSystem;
7
8 class CreateArray extends AbstractCustomCommand {
9
10     protected function getValue(): string {
11         return 'jsonValue';
12     }
13

```

(continues on next page)

(continued from previous page)

```

14     protected function getAliases(): array {
15         return array(
16             "createarray",
17             "arrayFromJson"
18         );
19     }
20
21     protected function getName(): string {
22         return 'createArray';
23     }
24
25     protected function getParameters(): array {
26         return [
27             'f' => Parameter::create('filename', 'The filename to create.',
↪[])
28         ];
29     }
30
31     protected function getExamples(): array {
32         return [
33             'Save an array in test.php' => "Ubiquity createArray \"{\\\\"
↪"created\\\\"":true}\\\" -f=test.php"
34         ];
35     }
36
37     protected function getDescription(): string {
38         return 'Creates an array from JSON and save to file';
39     }
40
41     public function run($config, $options, $what, ...$otherArgs) {
42         echo ConsoleFormatter::showInfo('Run createArray command');
43         $array = \json_decode($what, true);
44         $error = \json_last_error();
45         if ($error != 0) {
46             echo ConsoleFormatter::showMessage(\json_last_error_msg(), 'error
↪');
47         } else {
48             $filename = self::getOption($options, 'f', 'filename');
49             if ($filename != null) {
50                 UFileSystem::save($filename, "<?php\nreturn " . var_
↪export($array, true) . ";\n");
51                 echo ConsoleFormatter::showMessage("$filename succefully_
↪created!", 'success', 'CreateArray');
52             } else {
53                 echo ConsoleFormatter::showMessage("Filename must have a_
↪value!", 'error');
54             }
55         }
56     }
57 }

```

38.3.1 Custom command execution

The new command is accessible from the devtools, as long as it is in the project:

```
Ubiquity help createArray
```

```
C:\xampp7.4.4\htdocs\quick-start>Ubiquity help createArray

  • The project folder is C:\xampp7.4.4\htdocs\quick-start

  • Command createArray find by name

■ createArray [jsonValue] =>
  • Creates an array from JSON and save to file
  • Aliases : createarray,arrayFromJson
  • Parameters :
    -f          shortcut of --filename
                The filename to create.

  × Samples :
    Save an array in test.php
    • Ubiquity createArray "{\"created\":true}" -f=test.php
```

```
Ubiquity createArray "{\"b\":true,\"i\":5,\"s\":\"string\"}" -f=test.php
```

```
C:\xampp7.4.4\htdocs\quick-start>Ubiquity createArray "{\"b\":true,\"i\":5,\"s\":\"string\"}" -f=test.php

  • The project folder is C:\xampp7.4.4\htdocs\quick-start

  • Run createArray command

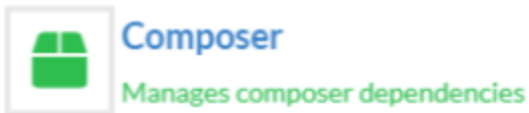
  ■ success : CreateArray
    • test.php succefully created!
```

Composer management

Note: This part is accessible from the **webtools**, so if you created your project with the **-a** option or with the **create-project** command..

39.1 Access


From the webtools, activate the **composer** part,


















or go directly to <http://127.0.0.1:8090/Admin/composer>.

39.2 Dependencies list

The interface displays the list of already installed dependencies, and those that are directly installable.


Composer
 Manages composer dependencies


+ Add dependency...
 Generate composer update...
 Optimize autoloader


Name	Version
 require	
 authentication	
phpmv/ubiquity-oauth	
 core	
php	^7.4
phpmv/ubiquity	^2.3
 database	
phpmv/ubiquity-tarantool	
phpmv/ubiquity-mysql	
 frontend	
phpmv/php-mv-ui	
 servers	
phpmv/ubiquity-workerman	
phpmv/ubiquity-swoole	
lapinskas/roadrunner-ubiquity	
phpmv/ubiquity-php-pm	
phpmv/ubiquity-reactphp	
 templates	
twig/twig	^3.0

39.3 Dependency installation

39.3.1 Among the listed dependencies:

Click on the **add** button of the dependencies you want to add.

 authentication

phpmv/ubiquity-oauth
 version...
 

Then click on the **Generate composer update** button:

Composer commands

Text

composer require phpmv/ubiquity-oauth

The validation generates the update.

39.3.2 For non listed dependencies:

Click on the **Add dependency** button :

Adding a new composer dependency

☐ dev

- Enter a vendor name (provider) ;
- Select a package in the list ;
- Select eventually a version (if none, the last stable version will be installed).

39.4 Dependency removal

Click on the **remove** button of the dependencies you want to add.

Name	Version	
require		
authentication		
phpmv/ubiquity-oauth	^0.0.2	<input checked="" type="button" value="To remove"/>

Then click on the **Generate composer update** button, and validate the update.

Note: It is possible to perform several addition or deletion operations and validate them simultaneously.

39.5 Composer optimization

Click on the **Optimize autoloader** button.

This optimize composer autoloading with an authoritative classmap.

CHAPTER 40

Ubiquity Caching

Ubiquity dependencies

- `^php 7.4`
- `phpmv/ubiquity => Ubiquity core`

41.1 In production

41.1.1 Templating

Twig is required if it is used as a template engine, which is not a requirement.

- `twig/twig => Template engine`

41.2 In development

41.2.1 Webtools

- `phpmv/ubiquity-dev => dev classes for webtools and devtools since v2.3.0`
- `phpmv/php-mv-ui => Front library`
- `mindplay/annotations => Annotations library, required for generating models, cache...`
- `monolog/monolog => Logging`
- `czproject/git-php => Git operations (+ require git console)`

41.2.2 Devtools

- `phpmv/ubiquity-devtools => Cli console`
- `phpmv/ubiquity-dev => dev classes for webtools and devtools since v2.3.0`

- `mindplay/annotations` => Annotations library, required for generating models, cache...

41.2.3 Testing

- `codeception/codeception` => Tests
- `codeception/c3` => C3 integration
- `phpmv/ubiquity-codeception` => Codeception for Ubiquity

OAuth2 client module

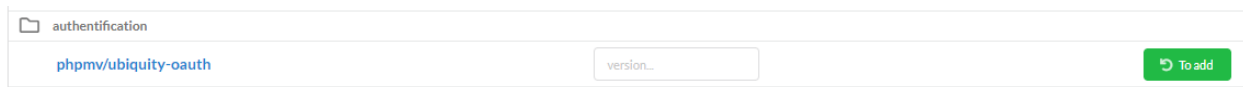
Note: This part is accessible from the **webtools**, so if you created your project with the **-a** option or with the **create-project** command. The OAuth module is not installed by default. It uses HybridAuth library.

42.1 Installation

In the root of your project:


```
composer require phpmv/ubiquity-oauth
```

Note: It is also possible to add the **ubiquity-oauth** dependency using the **Composer** part of the administration module.




42.2 OAuth configuration

42.2.1 Global configuration

 **OAuth**
Authentication OAuth or OpenID


[+ Add provider...](#) [⚙️ Global configuration](#) [+ Create OAuth controller](#)

▼ Callback URL

 **Callback**
Callback URL is missing in config file!


×


▼ Providers

Provider name	Enabled	Checked status	Actions
<div> nothing to display</div>			


Click on the **Global configuration** button, and modify the callback URL, which corresponds to the local callback url after a successful connection.

▼ Callback URL

 /oauth

 no route associated with callback

▼ Providers

Provider name	Enabled	Checked status	Actions
<div> nothing to display</div>			

42.2.2 OAuth controller

Click on the **Create OAuth controller** button and assign to the route the value previously given to the callback:

Adding an OAuth controller

Name

controllers\ OAuthTest

Base class

Ubiquity\controllers\auth\AbstractOAuthController

@route(path)


oauth|


✓ Validate

○ Cancel

Validate and reset the router cache:


▼ Callback URL

 /oauth

 controllers\OAuthTest::_oauth

✓

▼ Providers

Provider name	Enabled	Checked status	Actions
<div> nothing to display</div>			

42.2.3 Providers

Note: For an OAuth authentication, it is necessary to create an application at the provider beforehand, and to take note of the keys of the application (id and secret).

Click on the **Add provider** button and select **Google**:

Provider creation
 You need to create an application on your [Google](#) account and specify the id and secret credentials of the provider.

Configuration
 </> PHP source

Save configuration
 Cancel edition

Google

enabled	<input type="checkbox"/>						
force	<input type="checkbox"/>						
keys	<table> <tr> <td>id</td> <td>[empty]</td> <td><input type="text" value="610007010200-0nigm1-1041cmalic82u2in4fe11r04an-0000-google"/></td> </tr> <tr> <td>secret</td> <td>[empty]</td> <td><input type="text" value="NN00000000000000000000000000000000"/></td> </tr> </table>	id	[empty]	<input type="text" value="610007010200-0nigm1-1041cmalic82u2in4fe11r04an-0000-google"/>	secret	[empty]	<input type="text" value="NN00000000000000000000000000000000"/>
id	[empty]	<input type="text" value="610007010200-0nigm1-1041cmalic82u2in4fe11r04an-0000-google"/>					
secret	[empty]	<input type="text" value="NN00000000000000000000000000000000"/>					
scope	[empty] <input type="text" value="Scope"/>						
<div> Cancel deletions <div></div> </div>							

Check the connection by clicking on the **Check** button:

Callback URL

/oauth

controllers\OAuthTest::_oauth

Providers

Provider name	Enabled	Checked status	Actions
Google	<input checked="" type="checkbox"/>		<div> Check </div>

Post Login Information:

252

Chapter 42. OAuth2 client module

connection established to google

✕

identifier

[redacted]

photoURL

[redacted]

displayName

Jean-Christophe HERON

firstName

Jean-Christophe

lastName

HERON

language

fr

email

myaddressmail@gmail.com

emailVerified

myaddressmail@gmail.com

Close

42.3 OAuthController customization

The controller created is the following:

Listing 1: app/controllers/OAuthTest.php

```

namespace controllers;
use Hybridauth\Adapter\AdapterInterface;
/**
 * Controller OAuthTest
 */
class OAuthTest extends \Ubiquity\controllers\auth\AbstractOAuthController{

    public function index() {

    }

    /**
     * @get("oauth/{name}")
     */
    public function _oauth(string $name):void {
        parent::_oauth($name);
    }

    protected function onConnect(string $name,AdapterInterface $provider){
        //TODO
    }
}

```

- The **_oauth** method corresponds to the callback url
- The **onConnect** method is triggered on connection and can be overridden.

Example :

- Possible retrieval of an associated user in the database

- or creation of a new user
- Adding the logged-in user and redirection

Listing 2: app/controllers/OAuthTest.php

```
protected function onConnect(string $name, AdapterInterface $provider) {
    $userProfile = $provider->getUserProfile();
    $key = md5($name . $userProfile->identifier);
    $user = DAO::getOne(User::class, 'oauth= ?', false, [
        $key
    ]);
    if (! isset($user)) {
        $user = new User();
        $user->setOauth($key);
        $user->setLogin($userProfile->displayName);
        DAO::save($user);
    }
    USession::set('activeUser', $user);
    \header('location:./');
}
```

Note: Ubiquity supports multiple platforms : Swoole, Workerman, RoadRunner, PHP-PM, ngx_php.

43.1 Swoole

Install the Swoole extension on your system (linux) or in your Docker image :

```
#!/bin/bash
pecl install swoole
```

Run Ubiquity Swoole (for the first time, **ubiquity-swoole** package will be installed):

```
Ubiquity serve -t=swoole
```

43.1.1 Server configuration

Listing 1: .ubiquity/swoole-config.php

```
<?php
return array(
    "host" => "0.0.0.0",
    "port" => 8080,
    "options"=>[
        "worker_num" => \swoole_cpu_num() * 2,
        "reactor_num" => \swoole_cpu_num() * 2
    ]
);
```

The port can also be changed at server startup:

```
Ubiquity serve -t=swoole -p=8999
```

43.1.2 Services optimization

Startup of services will be done only once, at server startup.

Listing 2: app/config/services.php

```
\Ubiquity\cache\CacheManager::startProd($config);
\Ubiquity\orm\DAO::setModelsDatabases([
    'models\\Foo' => 'default',
    'models\\Bar' => 'default'
]);

\Ubiquity\cache\CacheManager::warmUpControllers([
    \controllers\IndexController::class,
    \controllers\FooController::class
]);

$swooleServer->on('workerStart', function ($srv) use (&$config) {
    \Ubiquity\orm\DAO::startDatabase($config, 'default');
    \controllers\IndexController::warmup();
    \controllers\FooController::warmup();
});
```

The warmUpControllers method:

- instantiates the controllers
- performs dependency injection
- prepares the call of the initialize and finalize methods (initialization of call constants)

At the start of each Worker, the **warmup** method of the controllers can for example initialize prepared DAO queries:

Listing 3: app/controllers/FooController.php

```
public static function warmup() {
    self::$oneFooDao = new DAOPreparedQueryById('models\\Foo');
    self::$allFooDao = new DAOPreparedQueryAll('models\\Foo');
}
```

43.2 Workerman

Workerman does not require any special installation (except for **libevent** to be used in production for performance reasons).

Run Ubiquity Workerman (for the first time, **ubiquity-workerman** package will be installed):

```
Ubiquity serve -t=workerman
```

43.2.1 Server configuration

Listing 4: .ubiquity/workerman-config.php

```
<?php
return array(
    "host" => "0.0.0.0",
    "port" => 8080,
    "socket"=>[
        "count" => 4,
        "reuseport" =>true
    ]
);
```

The port can also be changed at server startup:

```
Ubiquity serve -t=workerman -p=8999
```

43.2.2 Services optimization

Startup of services will be done only once, at server startup.

Listing 5: app/config/services.php

```
\Ubiquity\cache\CacheManager::startProd($config);
\Ubiquity\orm\DAO::setModelsDatabases([
    'models\Foo' => 'default',
    'models\Bar' => 'default'
]);

\Ubiquity\cache\CacheManager::warmUpControllers([
    \controllers\IndexController::class,
    \controllers\FooController::class
]);

$workerServer->onWorkerStart = function () use ($config) {
    \Ubiquity\orm\DAO::startDatabase($config, 'default');
    \controllers\IndexController::warmup();
    \controllers\FooController::warmup();
};
```

43.3 ngx_php

//TODO

43.4 Roadrunner

//TODO

CHAPTER 44

Indices and tables

- `genindex`
- `modindex`
- `search`