micro-framework Documentation

Release 2.0.9

phpmv

Mar 31, 2019

Quick-starts

1 Quick start with console 1

2 Quick start with web tools 9

3 Ubiquity-devtools installation 19

4 Project creation 21

5 Project configuration 23

6 Devtools usage 27

7 URLs 31

8 Router 35

9 Controllers 45

10 Events 51

11 Dependency injection 55

12 CRUD Controllers 61

13 Auth Controllers 73

14 Models generation 81

15 ORM 83

16 DAO 91

17 Request 95

18 Response 99

19 Session 103

20 Cookie 107

21 Views 109

22 Assets 113

23 Themes 115

24 Normalizers 123

25 Validators 125

26 Translation module 131

27 Rest 135

28 Contributing 137

29 Coding guide 139

30 Documenting guide 143

31 External libraries 145

32 Ubiquity Caching 147

33 Ubiquity dependencies 149

34 Indices and tables 151

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:

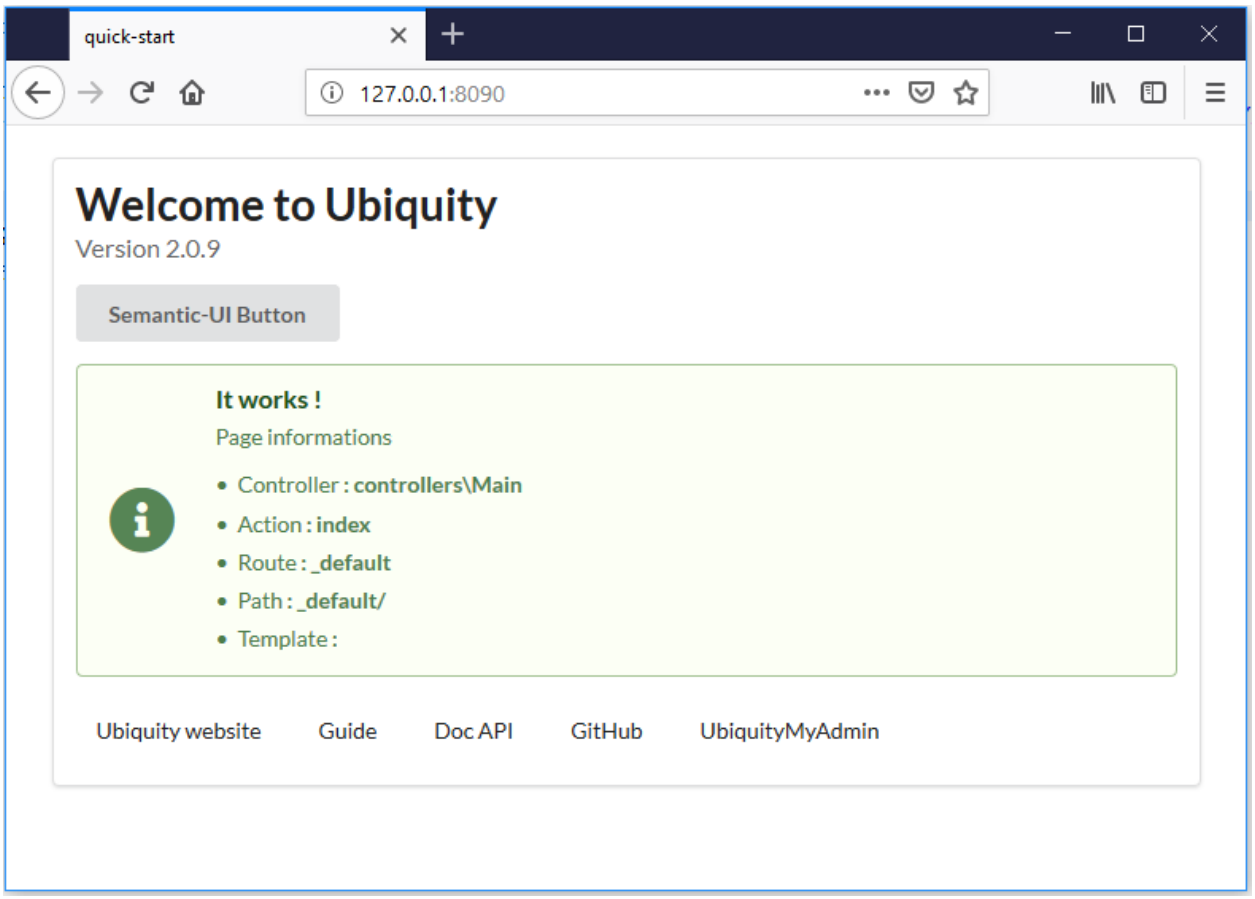
```
app
  ├ cache
  ├ config
  ├ controllers
  ├ models
  └ views
```

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
```

(continues on next page)

(continued from previous page)

```
 **/
class DefaultController extends ControllerBase{
     public function index(){}
}
```

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 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

```
    namespace controllers;
     /**
     * Controller DefaultController
     * @route("/hello","automated"=>true)
     **/
    class DefaultController extends ControllerBase{

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


    }
```

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 4: 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`

1.9 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.

```
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 5: 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'){
```

(continues on next page)

(continued from previous page)

```
        $this->loadView('DefaultController/information.html',compact('title','message
↪'));
}
```

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

Listing 6: 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` It's obvious



CHAPTER 2

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 **UbiquityMyAdmin** 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:

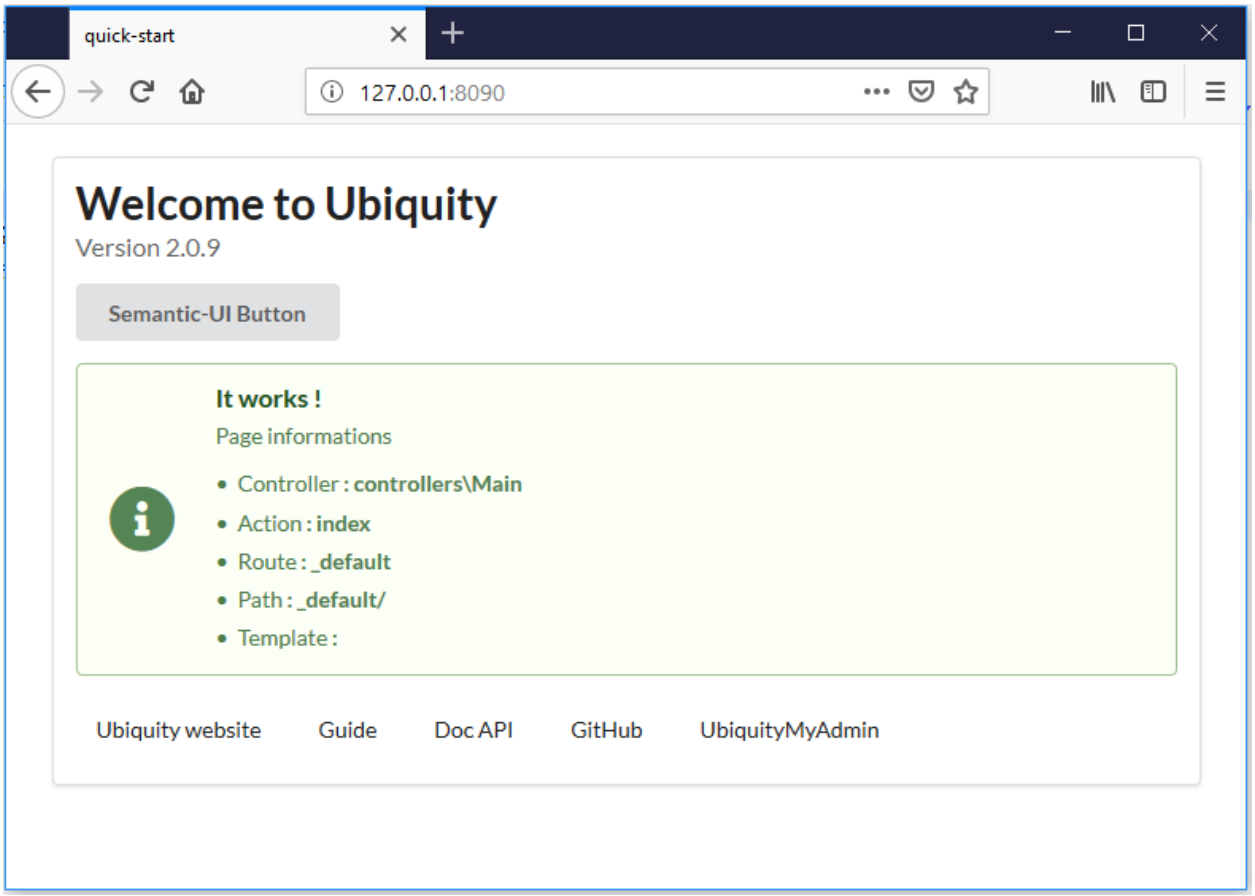
```
app
  ├ cache
  ├ config
  ├ controllers
  ├ models
  └ views
```

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**:

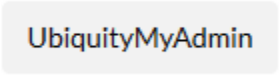


Note: If port 8090 is busy, you can start the server on another port using -p option.

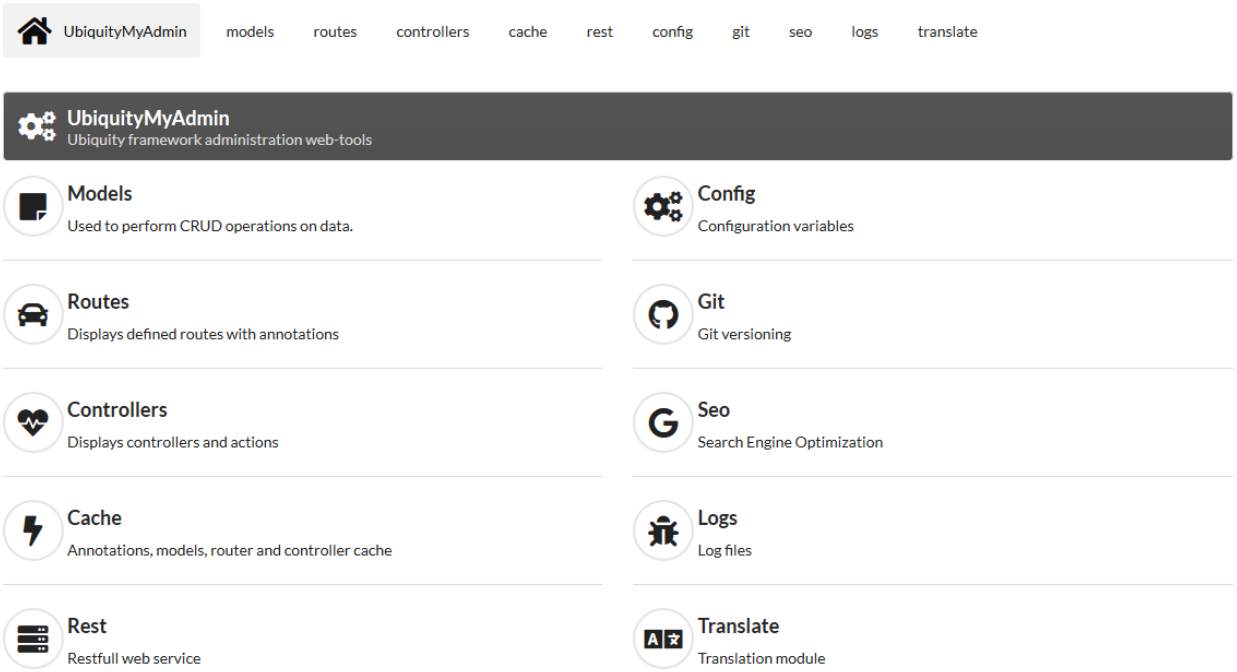
```
Ubiquity serve -p=8095
```

2.6 Controller

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



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

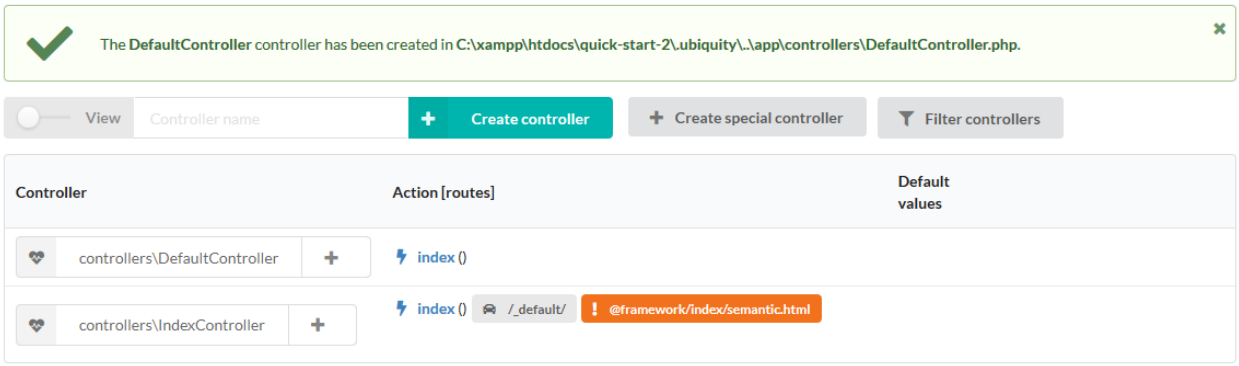


We go through it to create a controller.

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



The controller **DefaultController** is created:



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

Listing 1: app/controllers/DefaultController.php

```
namespace controllers;
 /**
 * Controller DefaultController
 **/
class DefaultController extends ControllerBase{
     public function index(){}
}
```

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

```
    namespace controllers;
     /**
```

(continues on next page)

(continued from previous page)

```
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

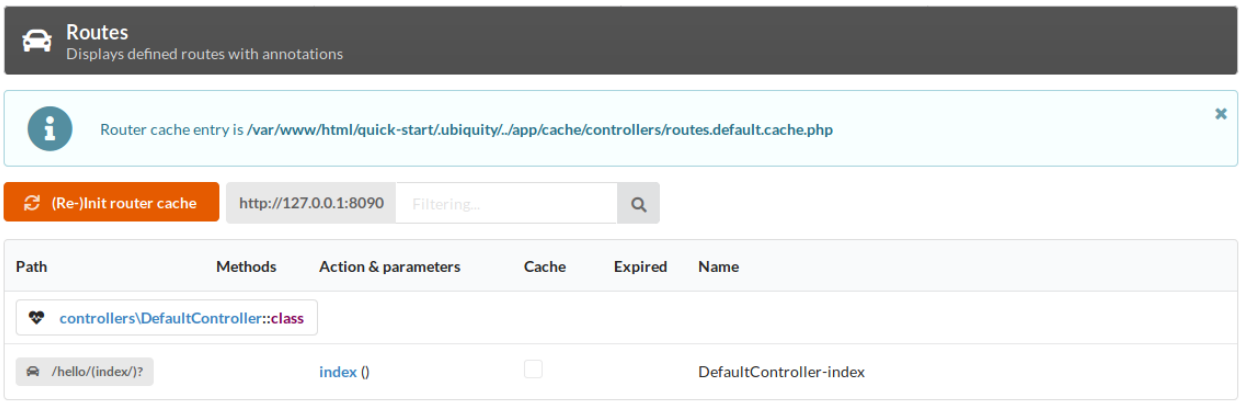
Important: No changes on the routes are effective without initializing the cache. Annotations are never read at runtime. This is also a design choice.

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:



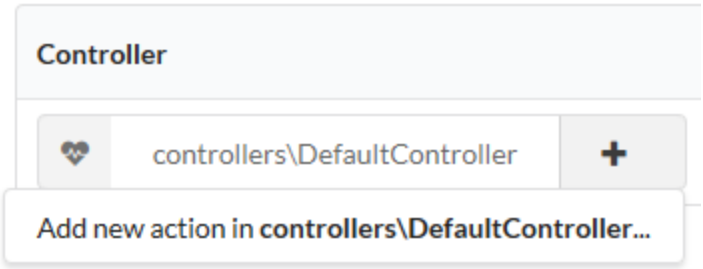
We can now test the page by clicking on the **GET** button or by going to the address `http://127.0.0.1:8090/hello`

2.8 Action & route with parameters

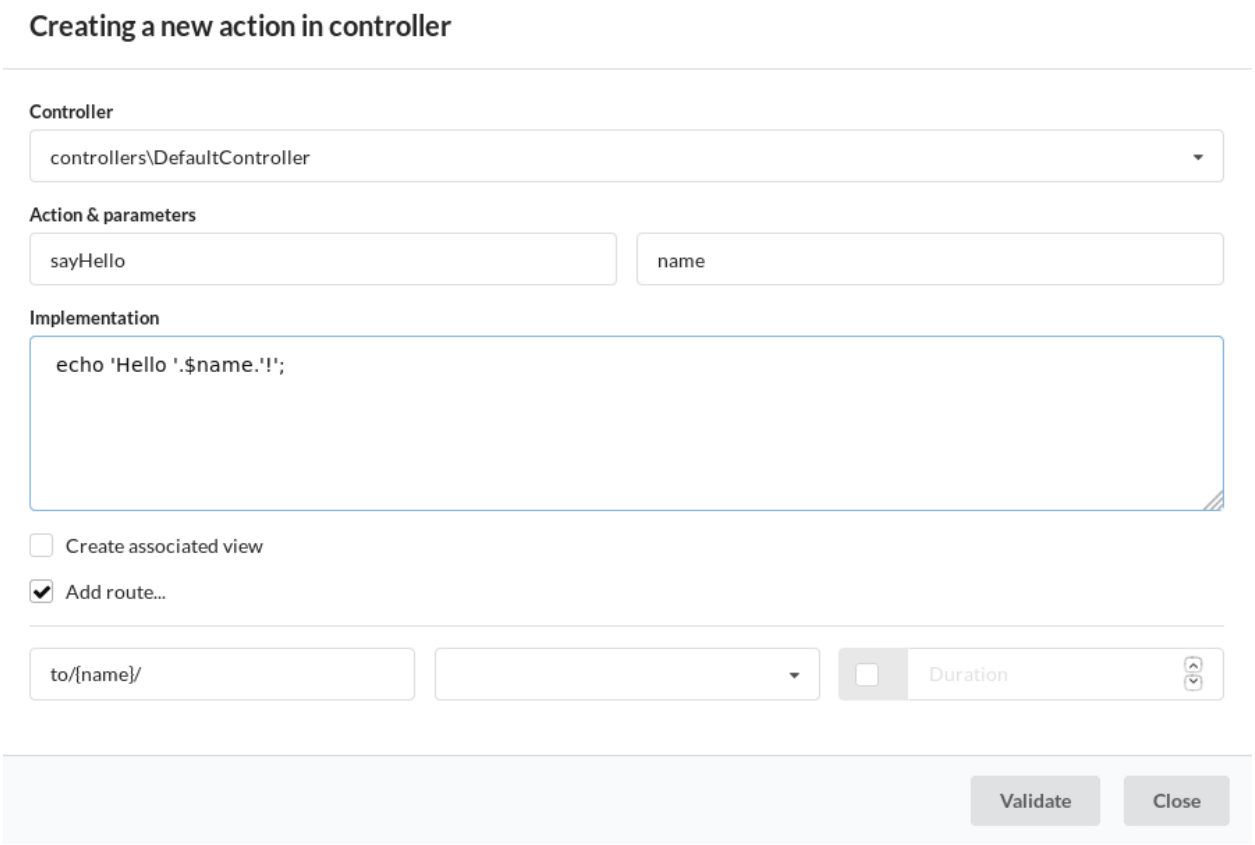
We will now create an action (sayHello) with a parameter (name), and the associated route (to): The route will use the parameter **name** of the action:

Go to the **Controllers** section:

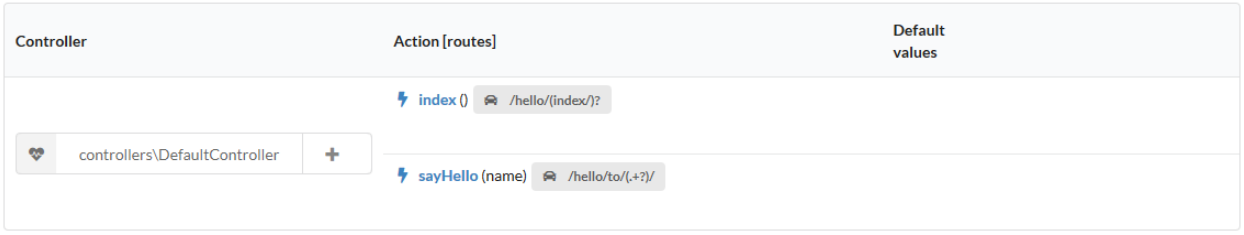
- click on the + button associated with DefaultController,
- then select **Add new action in..** item.



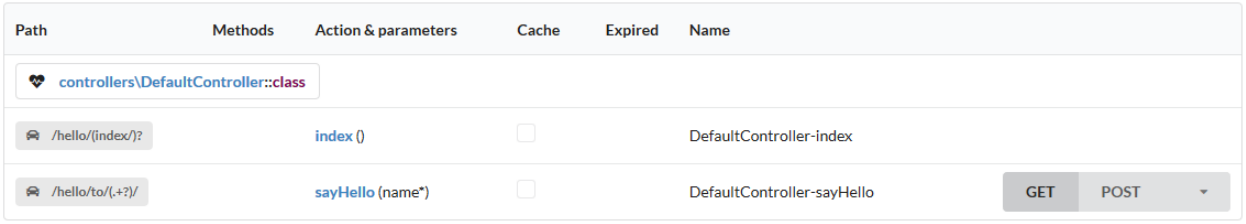
Enter the action information in the following form:



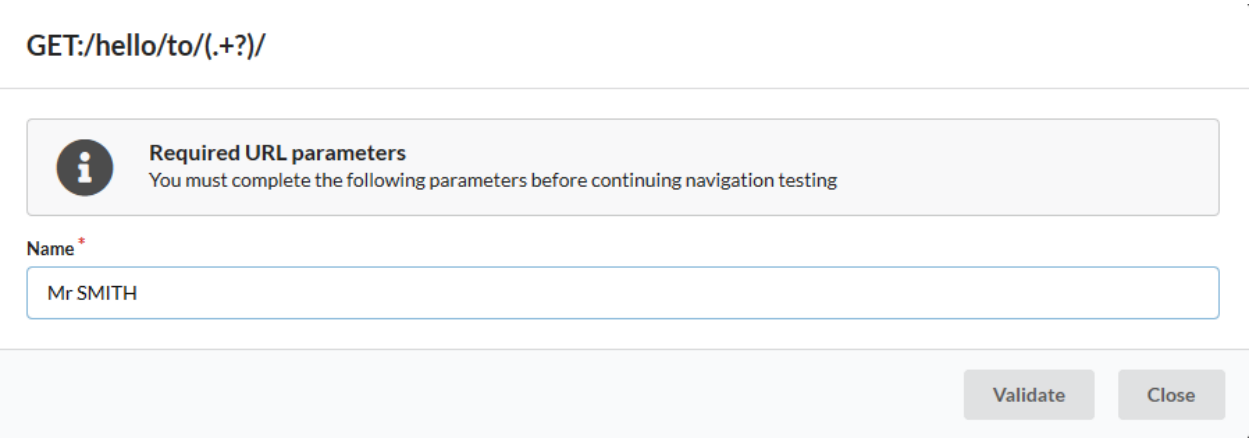
After re-initializing the cache with the orange button, we can see the new route **hello/to/{name}**:



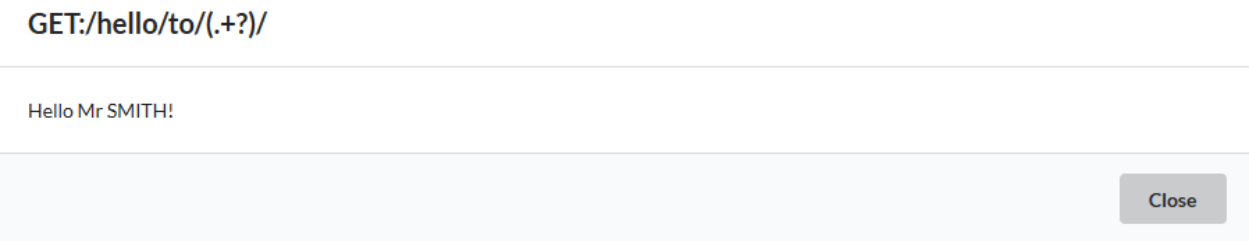
Check the route creation by going to the Routes section:



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



We can see the result:

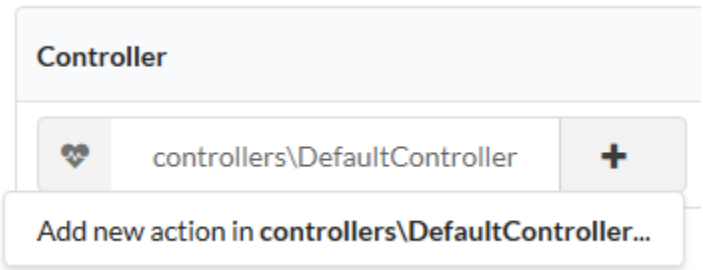


We could directly go to `http://127.0.0.1:8090/hello/to/Mr SMITH` address to test

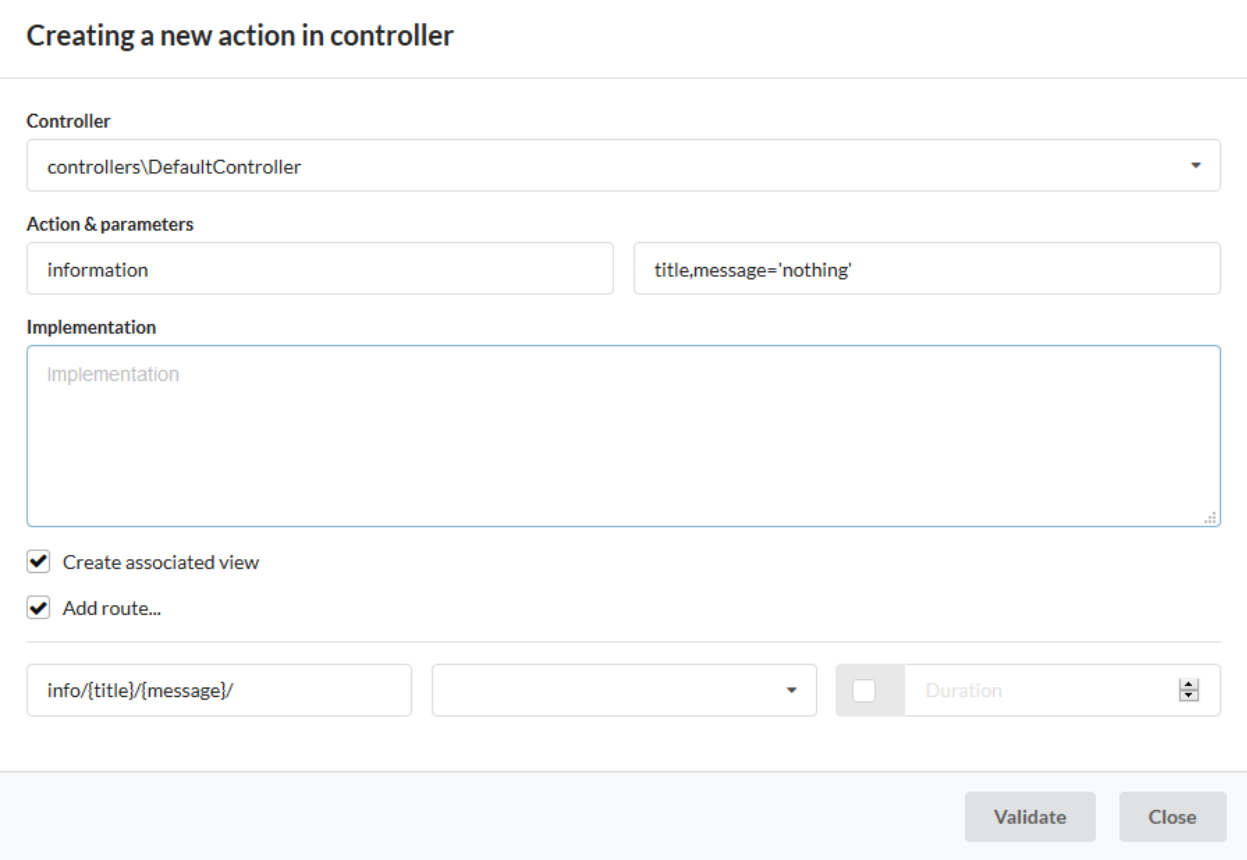
2.9 Action, route parameters & view

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**:

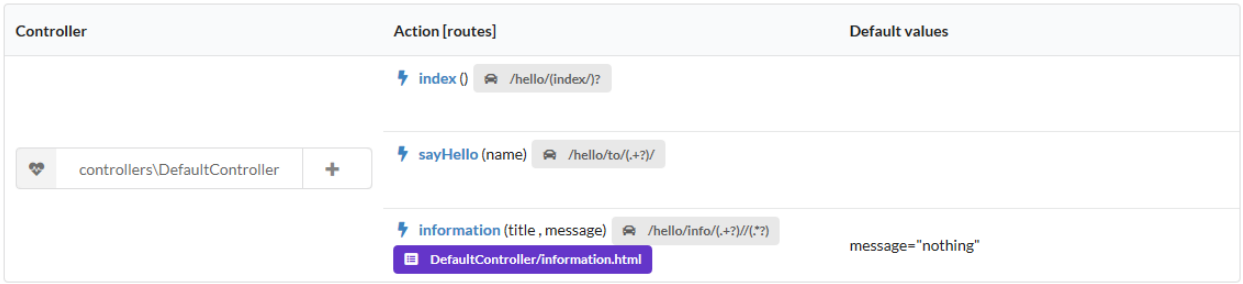


Enter the action information in the following form:



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

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



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

- *Dependency injection*
- *Events*
- *Views and themes*
- *Contributing*
- *Quick start with webtools (UbiquityMyAdmin)*
- Generating models:
  - with webtools (UbiquityMyAdmin)
  - with console (devtools)

CHAPTER 3

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 a bash console, 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
b	dbName	Sets the database name.		
s	serverName	Defines the db server address.	127.0.0.1	
p	port	Defines the db server port.	3306	
u	user	Defines the db server user.	root	
w	password	Defines the db server password.	''	
h	themes	Install themes.		semantic,bootstrap,foundation
m	all-models	Creates all models from db.	false	
a	admin	Adds UbiquityMyAdmin interface.	false	

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 **bogDb** database, with generation of all models and integration of semantic theme

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

4.4 Testing

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

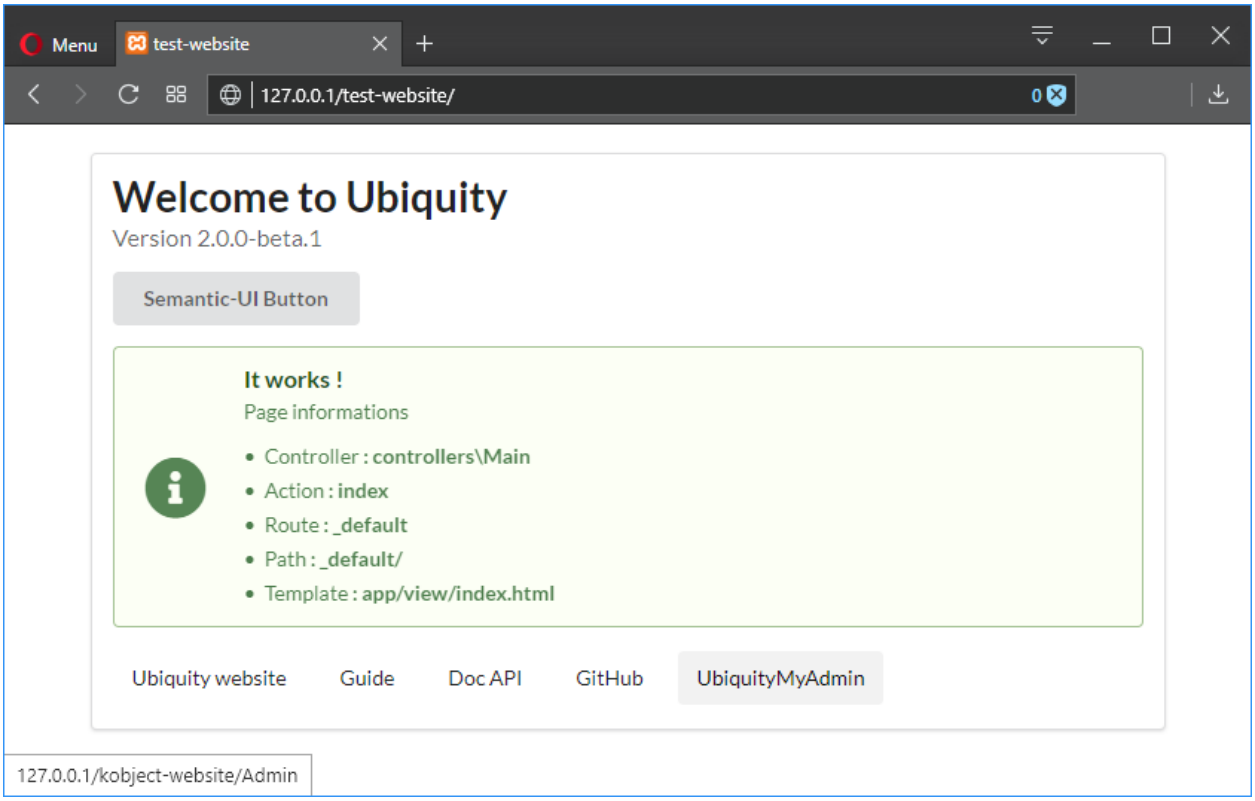
```
Ubiquity serve
```

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

```
return array(
                "siteUrl"=>"%siteUrl%",
                "database"=>[
                                "dbName"=>"%dbName%",
                                "serverName"=>"%serverName%",
                                "port"=>"%port%",
                                "user"=>"%user%",
                                "password"=>"%password%"
                ],
                "namespaces"=>[],
                "templateEngine"=>'Ubiquity\views\engine\Twig',
                "templateEngineOptions"=>array("cache"=>false),
                "test"=>false,
                "debug"=>false,
                "di"=>[%injections%],
                "cacheDirectory"=>"cache/",
                "mvcNS"=>["models"=>"models","controllers"=>"controllers"]
);
```

5.2 Services configuration

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

Listing 2: app/conf/services.php

```
    use Ubiquity\controllers\Router;

    try{
            \Ubiquity\cache\CacheManager::startProd($config);
    }catch(Exception $e){
            //Do something
    }
    \Ubiquity\orm\DAO::startDatabase($config);
    Router::start();
    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>
```

5.3.2 Nginx

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

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

CHAPTER 6

Devtools usage

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 it's 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

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

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 of controller classes allow you to customize URLs.

CHAPTER 8

Router

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

```
use Ubiquity\controllers\Router;

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 subclasses, and must have an **index** method:

Listing 3: app/controllers/FooController.php

```
namespace controllers;

class FooController extends ControllerBase{

        public function index(){
                echo 'Hello from foo';
        }
}
```

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

```
use Ubiquity\controllers\Router;

Router::addRoute("_default", \controllers\FooController::class,'bar');
```

8.2 Static routes

Static routes are defined using the **@route** annotation on controller methods.

Note: These annotations 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

Listing 5: app/controllers/ProductsController.php

```
namespace controllers;
 /**
 * Controller ProductsController
 **/
class ProductsController extends ControllerBase{

     /**
     * @route("products")
     */
     public function index(){}

}
```

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

8.2.2 Route parameters

A route can have parameters:

Listing 6: app/controllers/ProductsController.php

```
namespace controllers;
 /**
 * Controller ProductsController
 **/
class ProductsController extends ControllerBase{
     ...
     /**
     * Matches products/*
     *
     * @route("products/{value}")
     */
     public function search($value){
             // $value will equal the dynamic part of the URL
             // e.g. at /products/brocolis, then $value='brocolis'
             // ...
     }
}
```

8.2.3 Route optional parameters

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

Listing 7: app/controllers/ProductsController.php

```
namespace controllers;
 /**
 * Controller ProductsController
 **/
class ProductsController extends ControllerBase{
```

(continues on next page)

(continued from previous page)

```
    ...
    /**
    * Matches products/all/(.*?)/(.*?)
    *
    * @route("products/all/{pageNum}/{countPerPage}")
    */
    public function list($pageNum,$countPerPage=50){
            // ...
    }
}
```

8.2.4 Route requirements

php being an untyped language, it is possible to add specifications on the variables passed in the url via the attribute **requirements**.

Listing 8: app/controllers/ProductsController.php

```
namespace controllers;
 /**
 * Controller ProductsController
 **/
class ProductsController extends ControllerBase{
    ...
    /**
    * Matches products/all/(\d+)/(\d?)
    *
    * @route("products/all/{pageNum}/{countPerPage}","requirements"=>["pageNum"=>"\d+
↪","countPerPage"=>"\d?"])
    */
    public function list($pageNum,$countPerPage=50){
            // ...
    }
}
```

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:

Listing 9: app/controllers/ProductsController.php

```
namespace controllers;
 /**
 * Controller ProductsController
 **/
```

(continues on next page)

(continued from previous page)

```
class ProductsController extends ControllerBase{

     /**
 * @route("products","methods"=>["get"])
 */
     public function index(){}

}
```

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

It is also possible to use specific annotations **@get**, **@post**... `@get("products")`

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**.

Listing 10: app/controllers/ProductsController.php

```
namespace controllers;
 /**
 * Controller ProductsController
 **/
class ProductsController extends ControllerBase{

     /**
     * @route("products","name"=>"products_index")
     */
     public function index(){}

}
```

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 :

Listing 11: app/controllers/ProductsController.php

```
namespace controllers;
 /**
 * @route("/product")
 * Controller ProductsController
```

(continues on next page)

(continued from previous page)

```
 **/
class ProductsController extends ControllerBase{

 ...
     /**
 * @route("/all")
 **/
     public function display(){}

}
```

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 :

Listing 12: app/controllers/ProductsController.php

```
namespace controllers;
 /**
 * @route("/product","automated"=>true)
 * Controller ProductsController
 **/
class ProductsController extends ControllerBase{

     public function generate(){}

     public function display(){}

}
```

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:

Listing 13: app/controllers/ProductsBase.php

```
namespace controllers;
 /**
 * Controller ProductsBase
 **/
abstract class ProductsBase extends ControllerBase{

     /**
     *@route("(index/)?")
     **/
     public function index(){}

```

(continues on next page)

(continued from previous page)

```
12      /**
13      *@route("sort/{name}")
14      **/
15      public function sortBy($name){}
16
17  }
```

The derived class using inherited attribute:

Listing 14: app/controllers/ProductsController.php

```
1   namespace controllers;
2    /**
3    * @route("/product","inherited"=>true)
4    * Controller ProductsController
5    **/
6   class ProductsController extends ProductsBase{
7
8       public function display(){}
9
10  }
```

The inherited attribute defines the 2 routes contained 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 prority parameter of a route allows this route to be resolved more quickly.

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.

Listing 15: app/controllers/ProductsController.php

```
1   namespace controllers;
2    /**
3    * Controller ProductsController
4    **/
5   class ProductsController extends ControllerBase{
6
7       /**
8    * @route("products","priority"=>1)
9    */
10      public function index(){}
11
12      /**
13   * @route("products/all","priority"=>10)
14   */
15      public function all(){}
```

(continues on next page)

(continued from previous page)

```
16
17 }
```

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.

```
/**
* @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.

```
    /**
* @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 possiblity 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
```

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

CHAPTER 9

Controllers

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");
     }
}
```

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 7: 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 8: 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 9: 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

9.6 Access control

9.7 Forwarding

9.8 Dependency injection

9.9 namespaces

9.10 Super class

CHAPTER 10

Events

Note: The Events module uses the static class **EventsManager** 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	Occures 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

```
use Ubiquity\events\EventsManager;
use Ubiquity\events\DAOEvents;

...

	EventsManager::addListener(DAOEvents::AFTER_UPDATE, function($instance,$result){
		if($result==1){
			$instance->_updated=true;
		}
	});
```

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

```
use Ubiquity\events\EventsManager;
use Ubiquity\events\ViewEvents;

...

	EventsManager::addListener(ViewEvents::AFTER_RENDER,function(&$render,$viewname,
↪$datas){
		$render='<h1>'.$viewname.'</h1>'.$render;
	});
```

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

```
	namespace eventListener;

	use Ubiquity\events\EventListenerInterface;
	use Ubiquity\utils\base\UArray;

	class TracePageEventListener implements EventListenerInterface {
		const EVENT_NAME = 'tracePage';

		public function on(&...$params) {
			$filename = \ROOT . \DS . 'config\stats.php';
			$stats = [ ];
			if (file_exists ( $filename )) {
				$stats = include $filename;
			}
```

(continues on next page)

(continued from previous page)

```
                    $page = $params [0] . '::' . $params [1];
                    $value = $stats [$page] ?? 0;
                    $value ++;
                    $stats [$page] = $value;
                    UArray::save ( $stats, $filename );
            }
    }
```

10.4 Registering events

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

Listing 4: app/config/services.php

```
    use Ubiquity\events\EventsManager;
    use eventListener\TracePageEventListener;

    ...

    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

```
    namespace controllers;

    use Ubiquity\controllers\Controller;
    use Ubiquity\utils\http\URequest;
    use Ubiquity\events\EventsManager;
    use eventListener\TracePageEventListener;
    use Ubiquity\controllers\Startup;

    /**
     * ControllerBase.
     **/
    abstract class ControllerBase extends Controller{
            protected $headerView = "@activeTheme/main/vHeader.html";
            protected $footerView = "@activeTheme/main/vFooter.html";
            public function initialize() {
                    $controller=Startup::getController();
                    $action=Startup::getAction();
                    EventsManager::trigger(TracePageEventListener::EVENT_NAME,
↪$controller,$action);
                    if (! URequest::isAjax ()) {
                            $this->loadView ( $this->headerView );
                    }
```

(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\EventsManager;

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\EventsManager::start();
```

CHAPTER 11

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. **|br|** 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 2 types of injections, so as not to require introspection at execution:

- property injection
- setter injection

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($ctrl){
5              echo 'Service instanciation in '.get_class($ctrl);
```

(continues on next page)

(continued from previous page)

```
        }

        public function do($someThink=""){
            echo 'do '.$someThink ."in service";
        }
    }
```

11.1.2 Autowiring in Controller

Create a controller that requires the service

Listing 2: app/services/Service.php

```
    namespace controllers;

     /**
     * Controller Client
     **/
    class ClientController extends ControllerBase{

            /**
             * @autowired
             * @var services\Service
             */
            private $service;

            public function index(){}

            /**
             * @param \services\Service $service
             */
            public function setService($service) {
                    $this->service = $service;
            }
    }
```

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

```php
class ServiceWithInit{
        private $init;

        public function init(){
                $this->init=true;
        }

        public function do(){
                if($this->init){
                        echo 'init well initialized!';
                }else{
                        echo 'Service not initialized';
                }
        }
}
```

11.2.2 Injection in controller

Listing 4: app/controllers/ClientController.php

```php
namespace controllers;

 /**
  * Controller Client
  **/
class ClientController extends ControllerBase{

        /**
         * @autowired
         * @var \services\Service
         */
        private $service;

        /**
         * @injected
         */
        private $serviceToInit;

        public function index(){
                $this->serviceToInit->do();
        }

        /**
         * @param \services\Service $service
         */
        public function setService($service) {
                $this->service = $service;
        }

        /**
         * @param mixed $serviceToInit
         */
        public function setServiceToInit($serviceToInit) {
```

(continues on next page)

(continued from previous page)

```
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();
                }
        ]
```

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();
                }
        ]
```

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();
                }
        ]
```

```
/**
 * @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

```
namespace services;

     class RuntimeService{
         public function __construct($ctrl){
             echo 'Service instanciation in '.get_class($ctrl);
         }
     }
```

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

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

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

Listing 6: app/controllers/MyController.php

```
namespace controllers;

      /**
      * Controller Client
      * property services\RuntimeService $rService
      **/
     class MyController extends ControllerBase{

             public function index(){
                     $this->rService->do();
             }
     }

<br />
```

CHAPTER 12

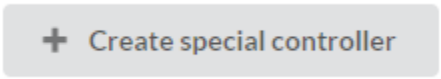
CRUD Controllers

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

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

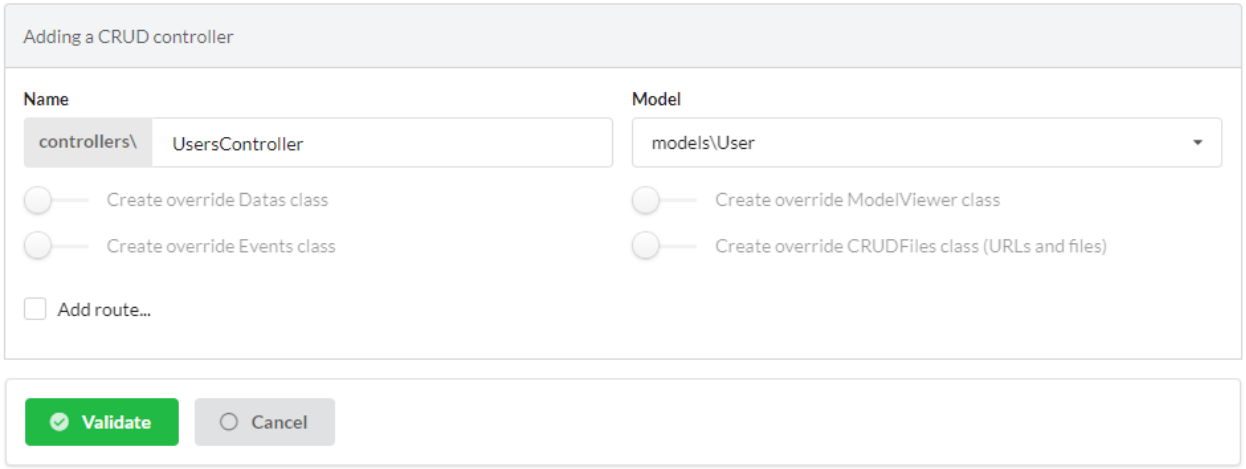
12.1 Creation

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



Then fill in the form:

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



12.2 Description of the features

The generated controller:

Listing 1: app/controllers/Products.php

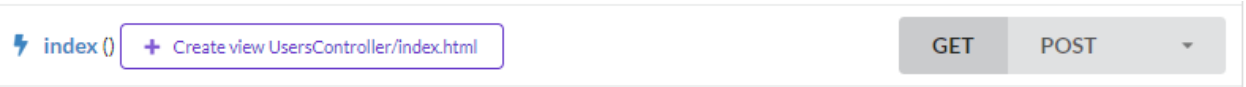
```php
<?php
namespace controllers;

 /**
 * CRUD Controller UsersController
 **/
class UsersController extends \Ubiquity\controllers\crud\CRUDController{

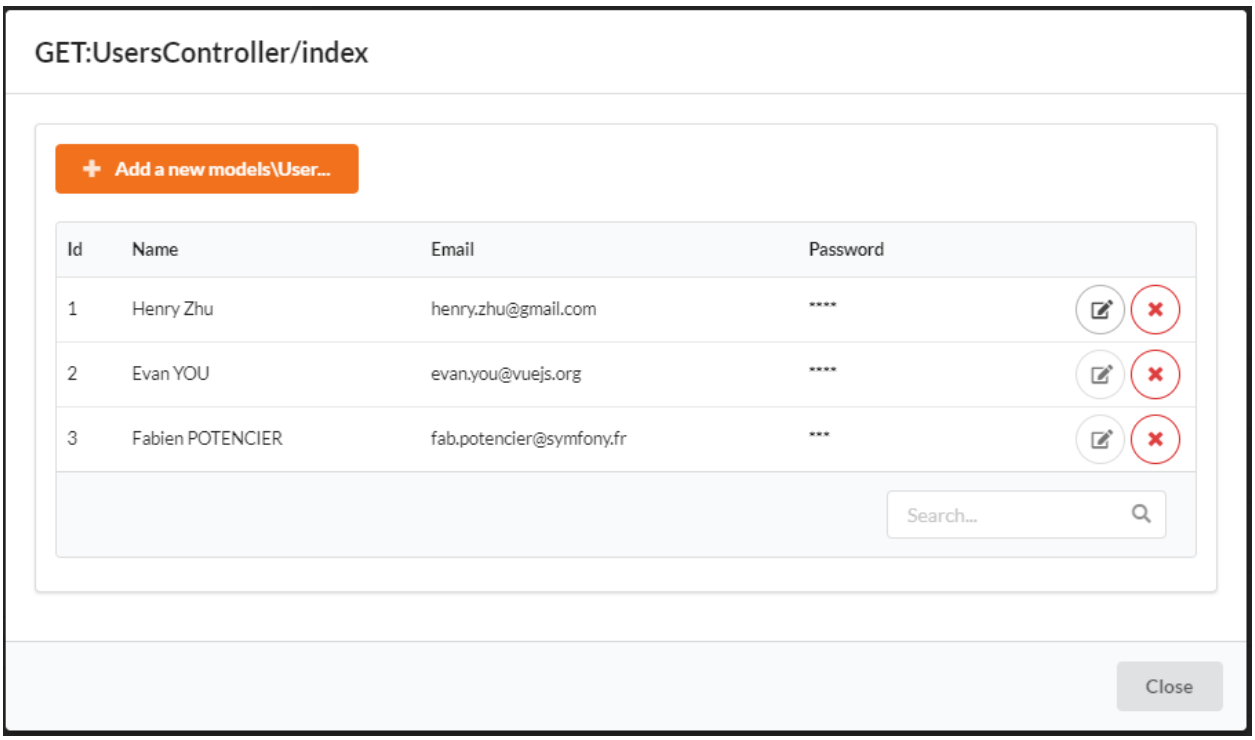
     public function __construct(){
             parent::__construct();
             $this->model="models\\User";
     }

     public function _getBaseRoute() {
             return 'UsersController';
     }
}
```

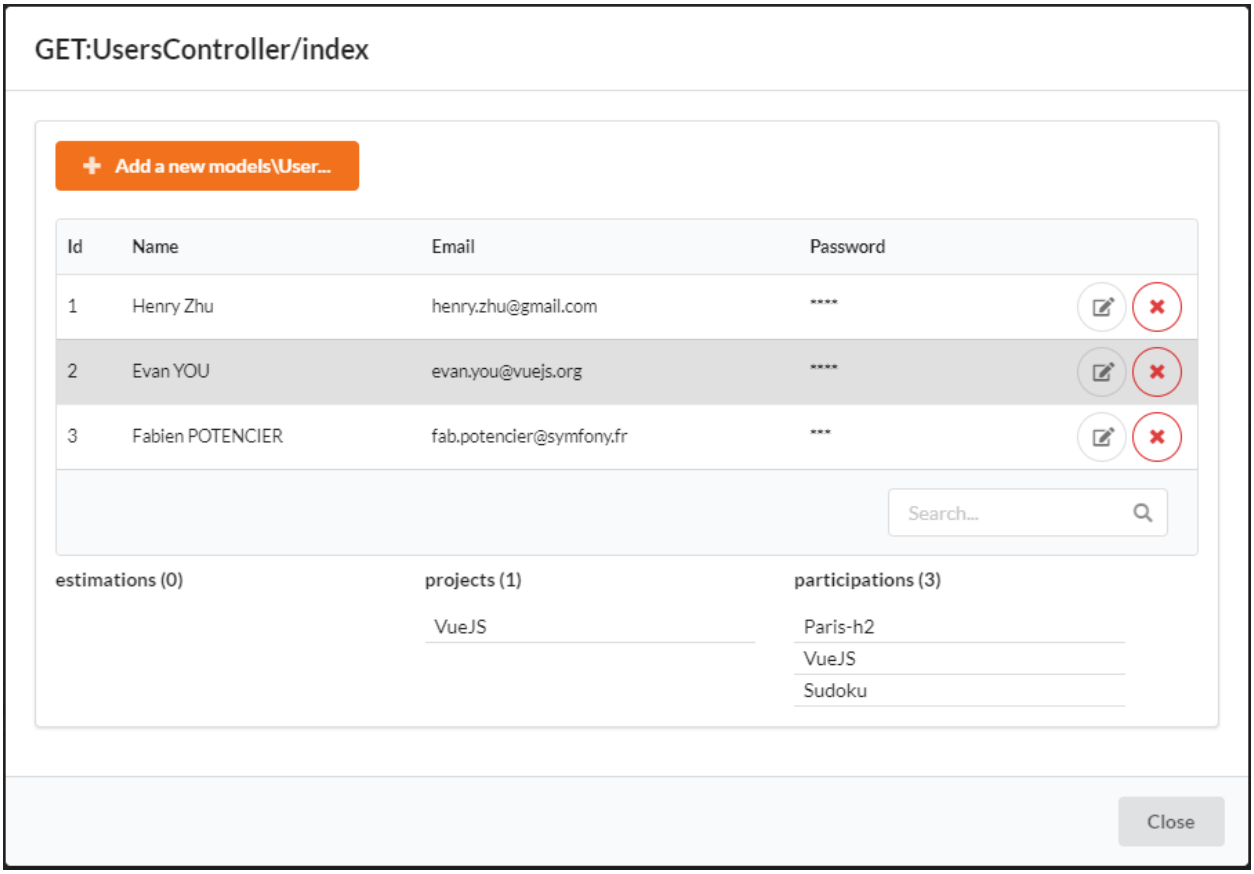
Test the created controller by clicking on the get button in front of the **index** action:



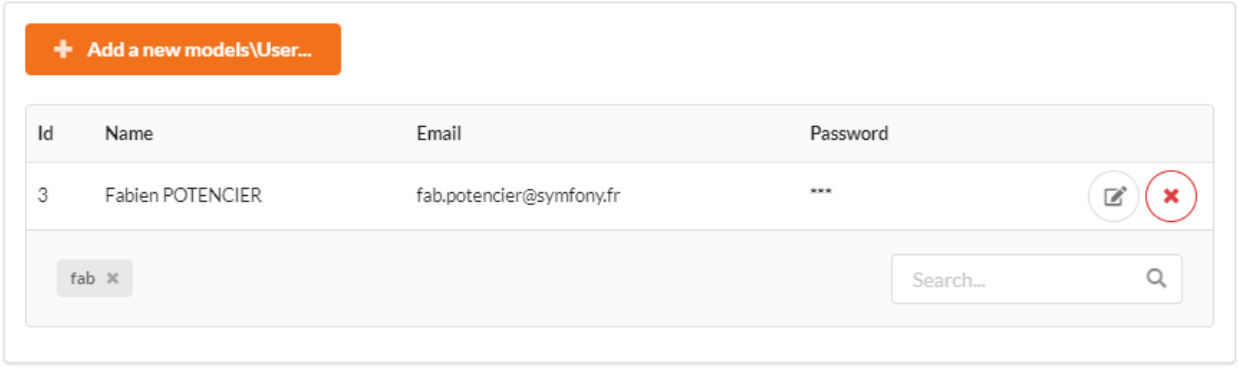
12.2.1 Read (index action)



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



Using the search area:

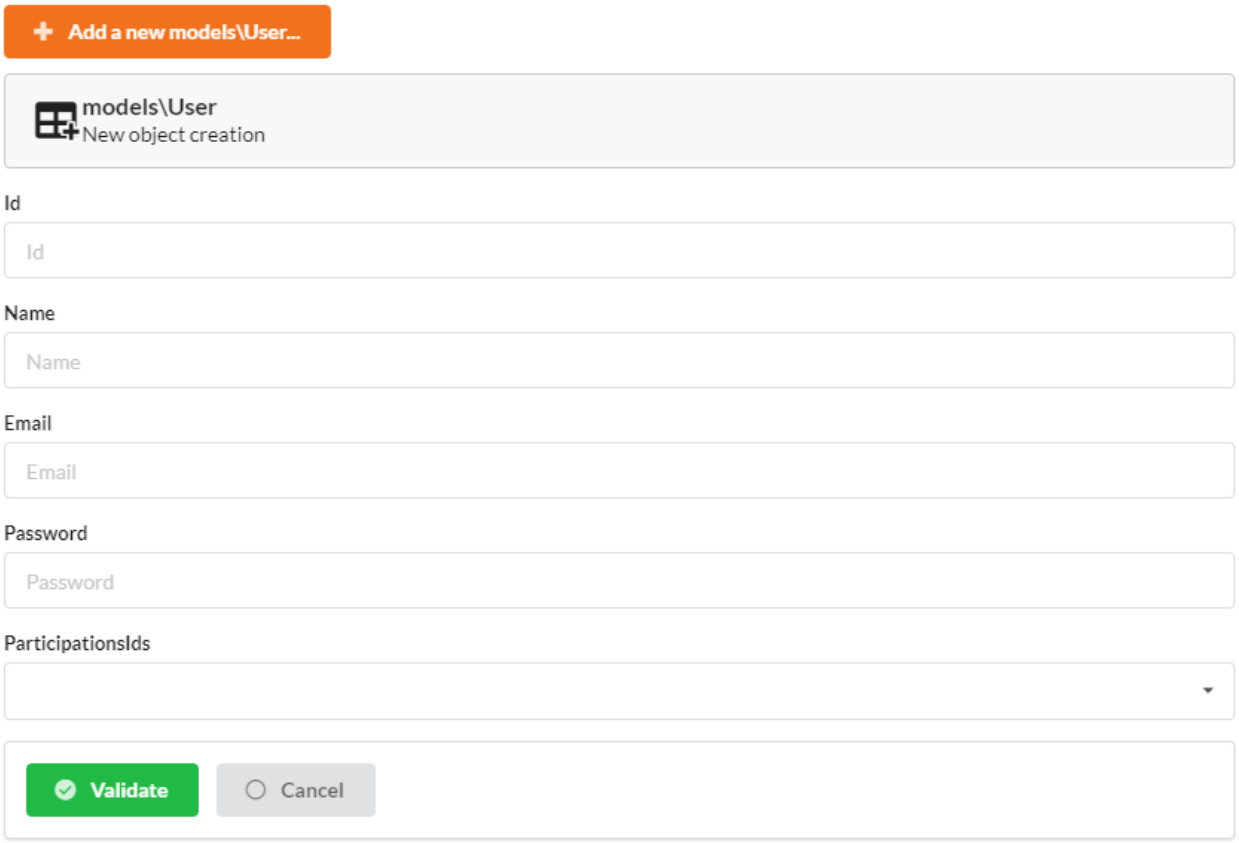


12.2.2 Create (newModel action)

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



The default form for adding an instance of User:

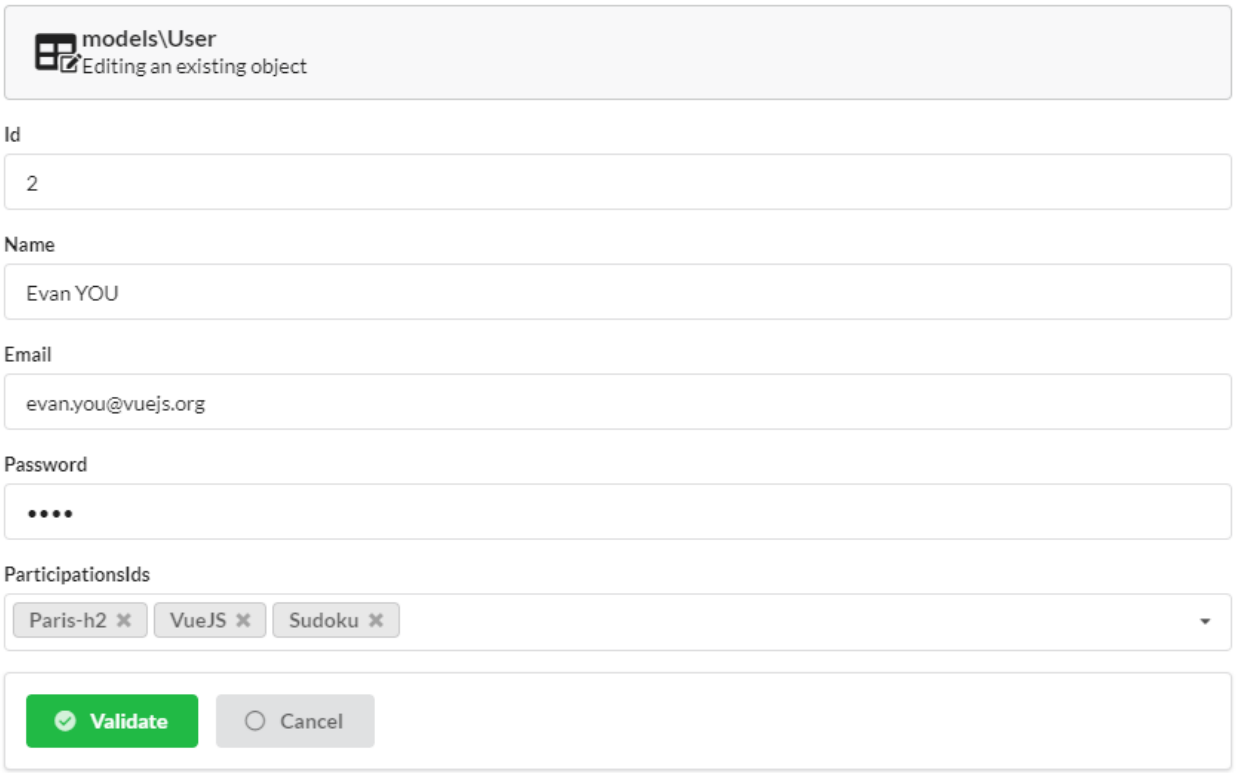


12.2.3 Update (update action)

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



The default form for adding an instance of User:

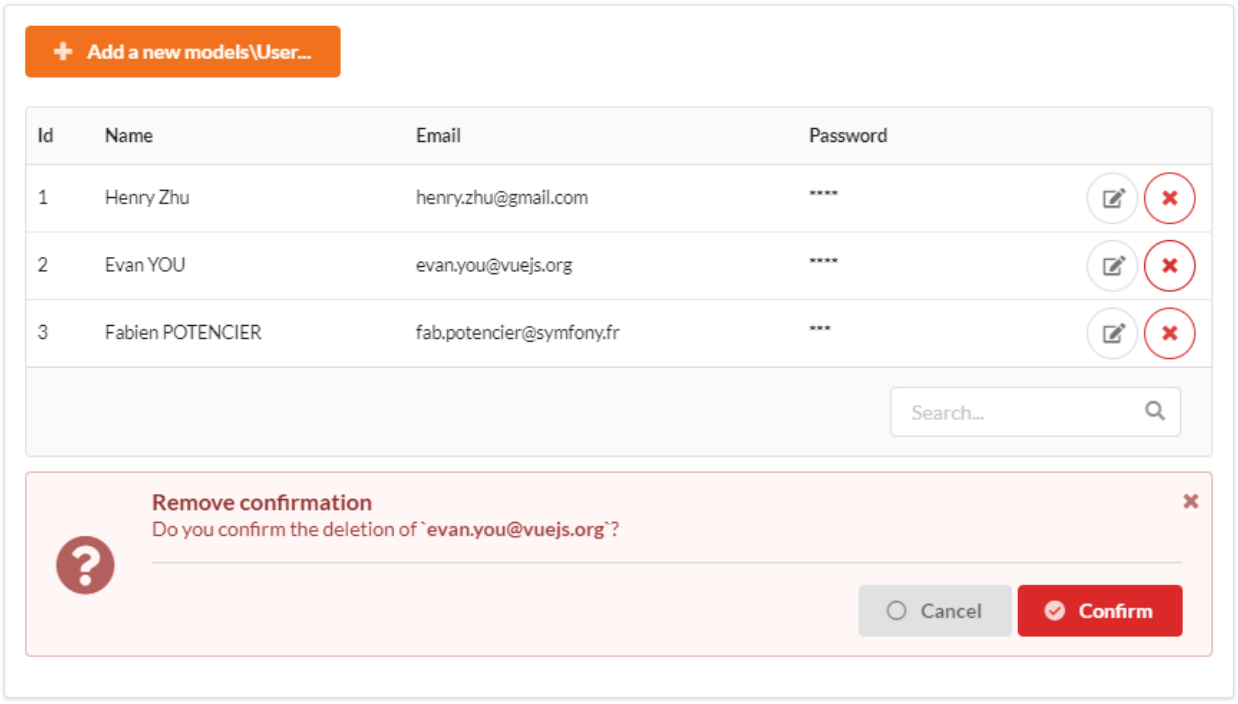


12.2.4 Delete (delete action)

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

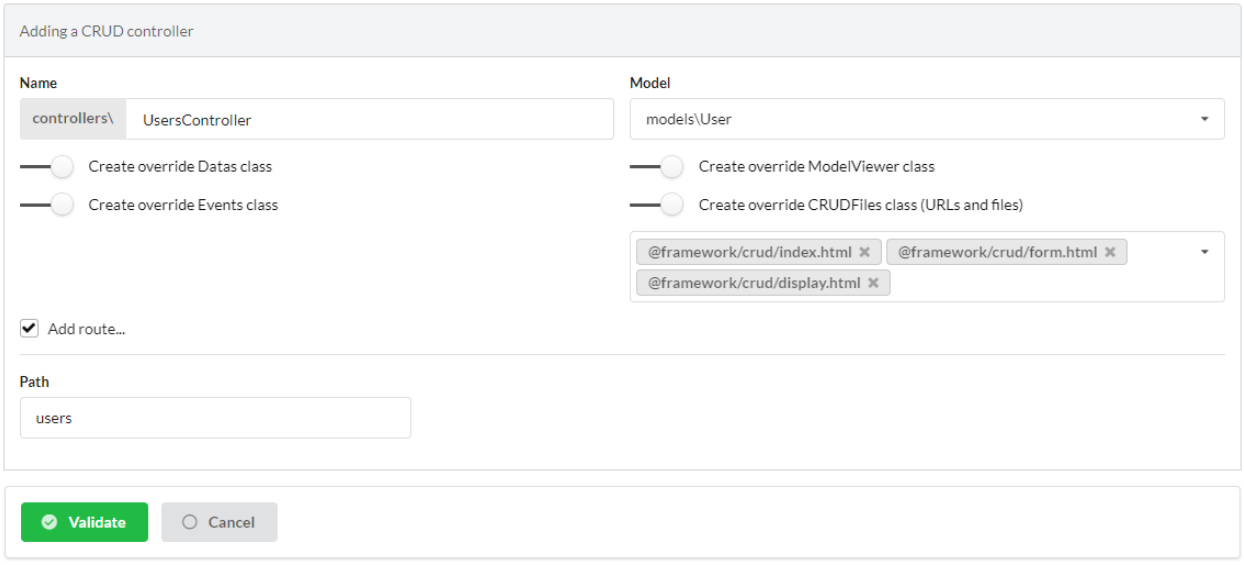


Display of the confirmation message before deletion:



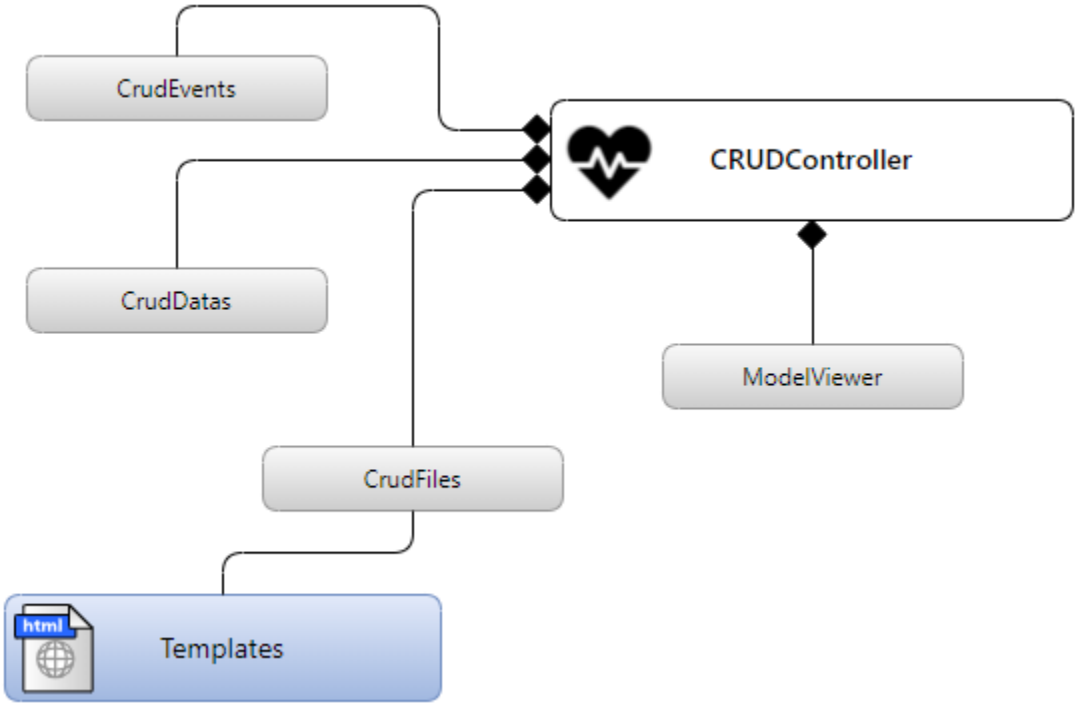
12.3 Customization

Create again a CrudController from the admin interface:



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, $object)	Returns the header for a single foreign object (issue from ManyToOne)	HtmlHeader
getFkElementDetails($member, $className, $object)	Returns a component for displaying a single foreign object (manyToOne relation)	HtmlLabel
getFkHeaderListDetails($member, $className, $list)	Returns the header for a list of foreign objects (oneToMany or ManyToMany)	HtmlHeader
getFkListDetails($member, $className, $list)	Returns a list component for displaying a collection of foreign objects (many)	HtmlList
edit and **newModel** routes		
getForm($identifier, $instance)	Returns the form for adding or modifying an object	HtmlForm
getFormTitle($form,$instance)	Returns an associative array defining form message title with keys "icon","message","subMessage"	HtmlForm
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		
_getInstancesFilter($model)	Adds a condition for filtering the instances displayed in dataTable	1=1
getFieldNames($model)	Returns the fields to display in the **index** action for $model	all member names
getSearchFieldNames($model)	Returns the fields to use in search queries	all member names
edit and **newModel** routes		
getFormFieldNames($model,$instance)	Returns the fields to update in the **edit** and **newModel** actions for $model	all member names
getManyToOneDatas($fkClass,$instance,$member)	Returns a list (filtered) of $fkClass objects to display in an html list	all $fkClass instances
getOneToManyDatas($fkClass,$instance,$member)	Returns a list (filtered) of $fkClass objects to display in an html list	all $fkClass instances
getManyToManyDatas($fkClass,$instance,$member)	Returns a list (filtered) of $fkClass objects to display in an html list	all $fkClass instances
display route		
getElementFieldNames($model)	Returns the fields to display in the **display** action for $model	all member names

CRUDEvents methods to override

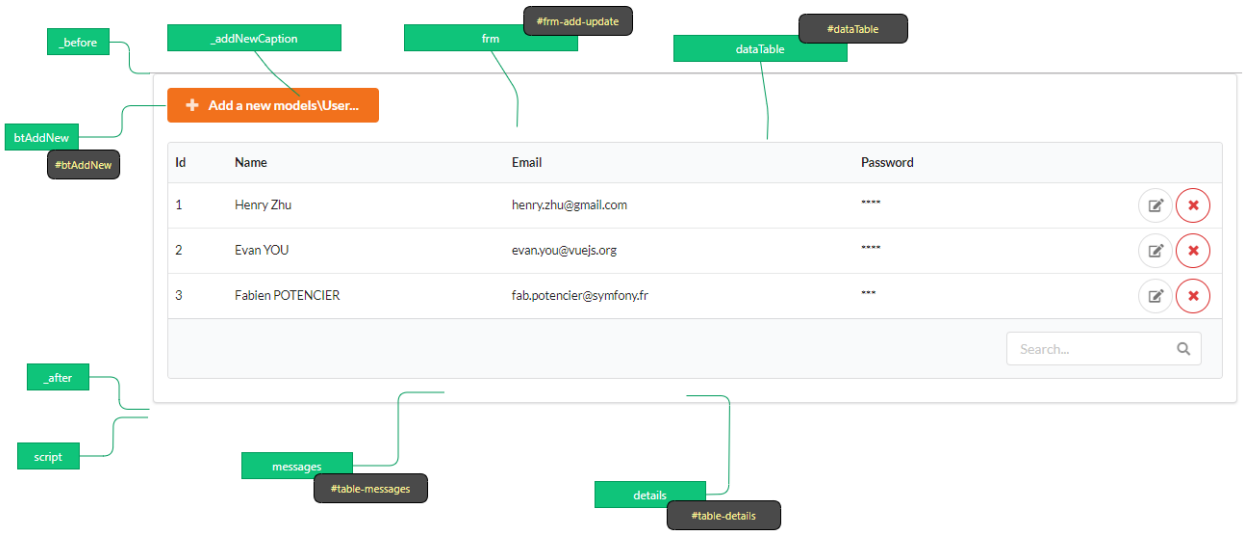
Method	Signification	Default return
index route		
onConfDeleteMessage(CRUDMessage $message,$instance)	Returns the confirmation message displayed before deleting an instance	CRUDMessage
onSuccessDeleteMessage(CRUDMessage $message,$instance)	RReturns the message displayed after a deletion	CRUDMessage
onErrorDeleteMessage(CRUDMessage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

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	""

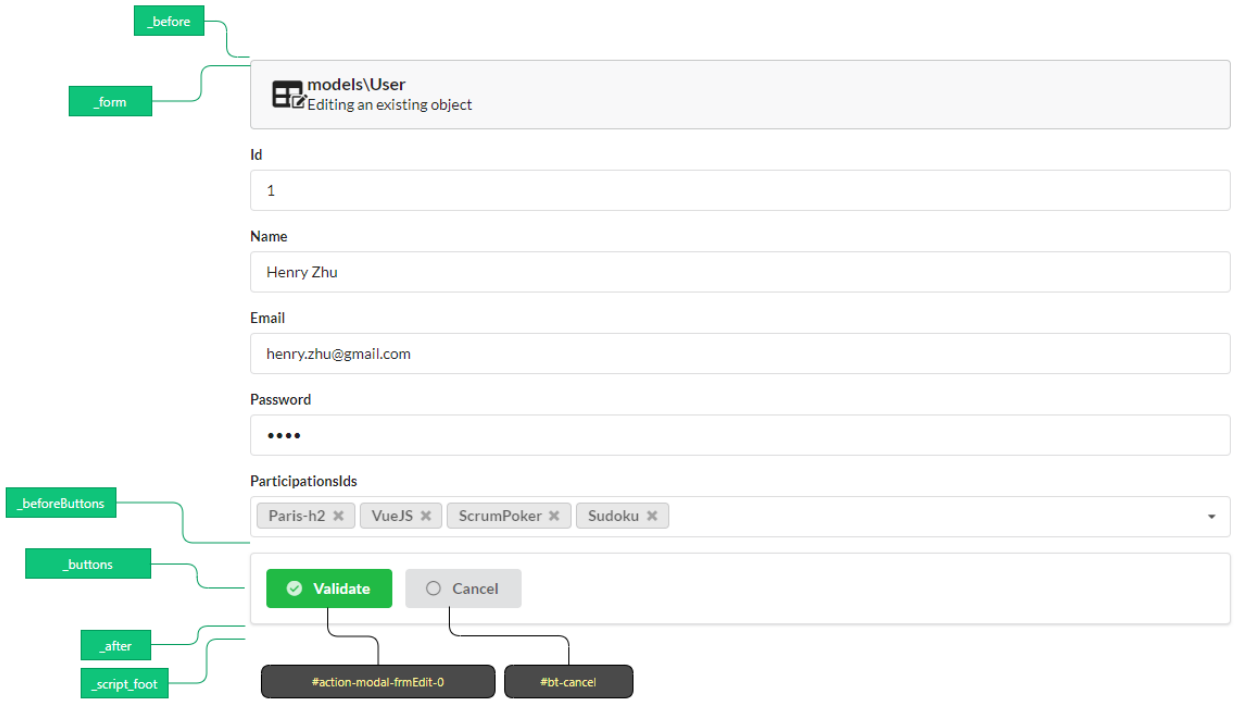
12.3.3 Twig Templates structure

index.html



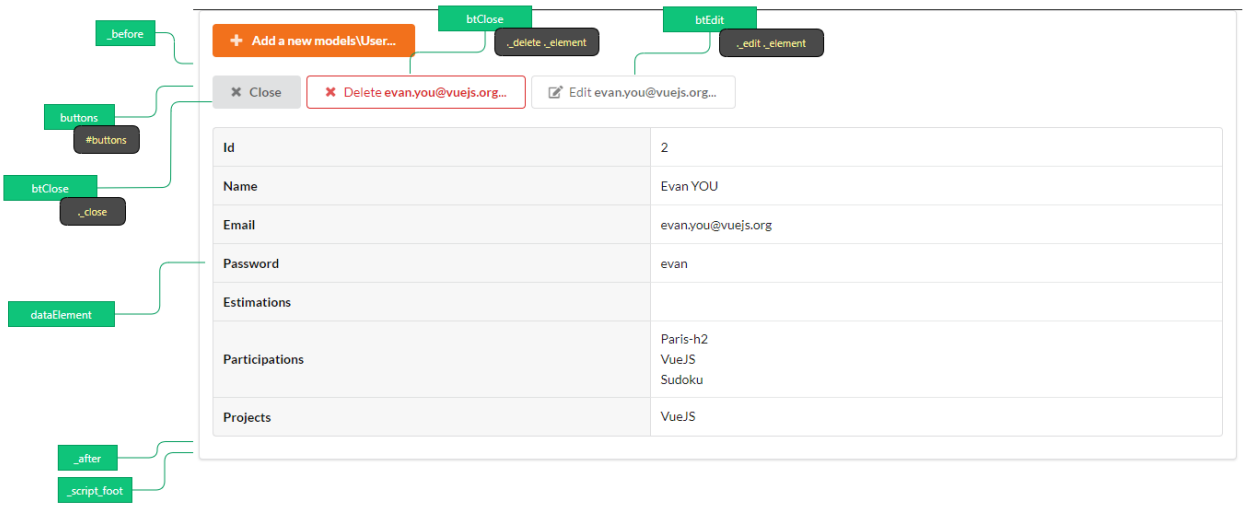
form.html

Displayed in **frm** block



display.html

Displayed in **frm** block



CHAPTER 13

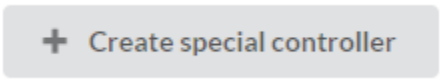
Auth Controllers

The Auth controllers allow you to perform basic authentification with:

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

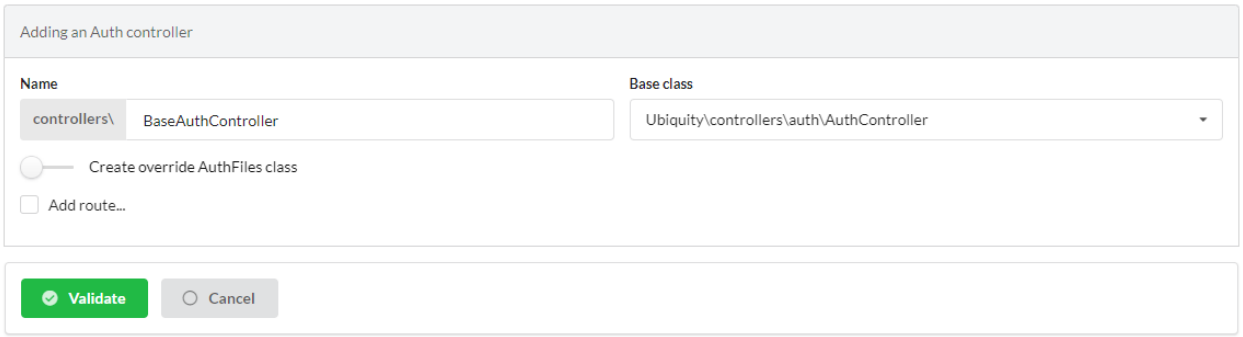
13.1 Creation

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)



The generated controller:

Listing 1: app/controllers/BaseAuthController.php

```php
/**
 * Auth Controller BaseAuthController
 **/
class BaseAuthController extends \Ubiquity\controllers\auth\AuthController{

	protected function onConnect($connected) {
		$urlParts=$this->getOriginalURL();
		USession::set($this->_getUserSessionKey(), $connected);
		if(isset($urlParts)){
			Startup::forward(implode("/",$urlParts));
		}else{
			//TODO
			//Forwarding to the default controller/action
		}
	}

	protected function _connect() {
		if(URequest::isPost()){
			$email=URequest::post($this->_getLoginInputName());
			$password=URequest::post($this->_getPasswordInputName());
			//TODO
			//Loading from the database the user corresponding to the parameters
			//Checking user creditentials
			//Returning the user
		}
		return;
	}

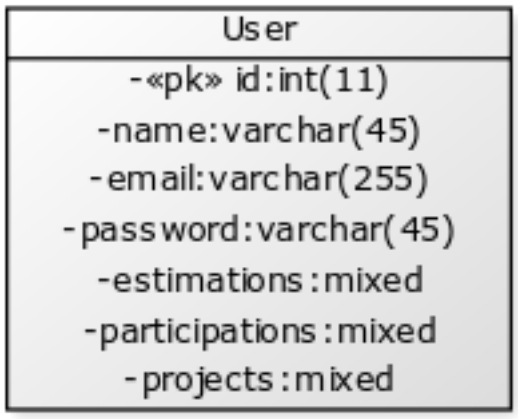
	/**
	 * {@inheritDoc}
	 * @see \Ubiquity\controllers\auth\AuthController::isValidUser()
	 */
	public function _isValidUser() {
		return USession::exists($this->_getUserSessionKey());
	}

	public function _getBaseRoute() {
		return 'BaseAuthController';
	}
}
```

13.2 Implementation of the authentification

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,
↪[$email,$password]);
21             }
22             return;
23     }
24
25     /**
26      * {@inheritDoc}
27      * @see \Ubiquity\controllers\auth\AuthController::isValidUser()
28      */
29     public function _isValidUser() {
30             return USession::exists($this->_getUserSessionKey());
31     }
32
33     public function _getBaseRoute() {
34             return 'BaseAuthController';
35     }
36     /**
```

(continues on next page)

(continued from previous page)

```
     * {@inheritDoc}
     * @see \Ubiquity\controllers\auth\AuthController::_getLoginInputName()
     */
    public function _getLoginInputName() {
            return "email";
    }
}
```

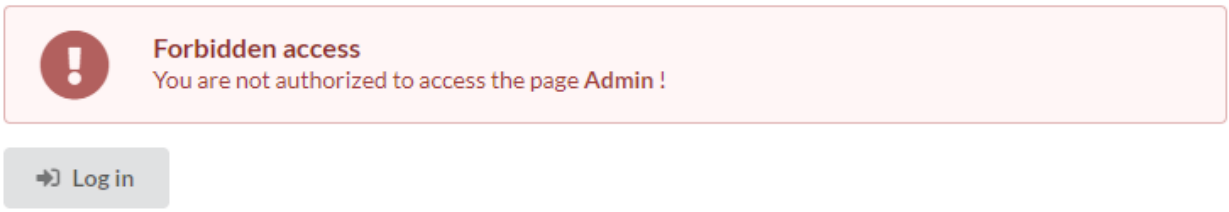
13.2.2 Admin controller modification

Modify the Admin Controller to use BaseAuthController:

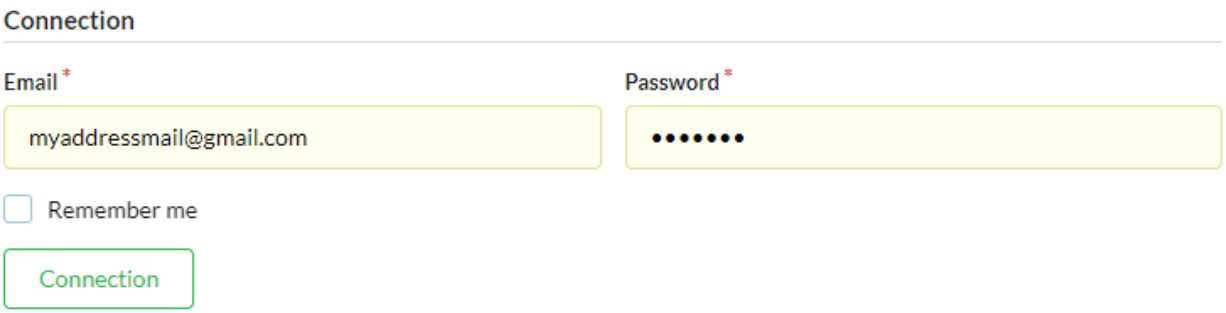
Listing 3: app/controllers/Admin.php

```
class Admin extends UbiquityMyAdminBaseController{
    use WithAuthTrait;
    protected function getAuthController(): AuthController {
            return new BaseAuthController();
    }
}
```

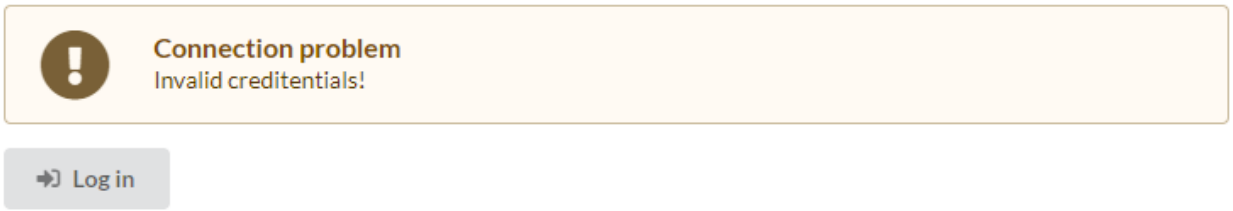
Test the administration interface at **/admin**:



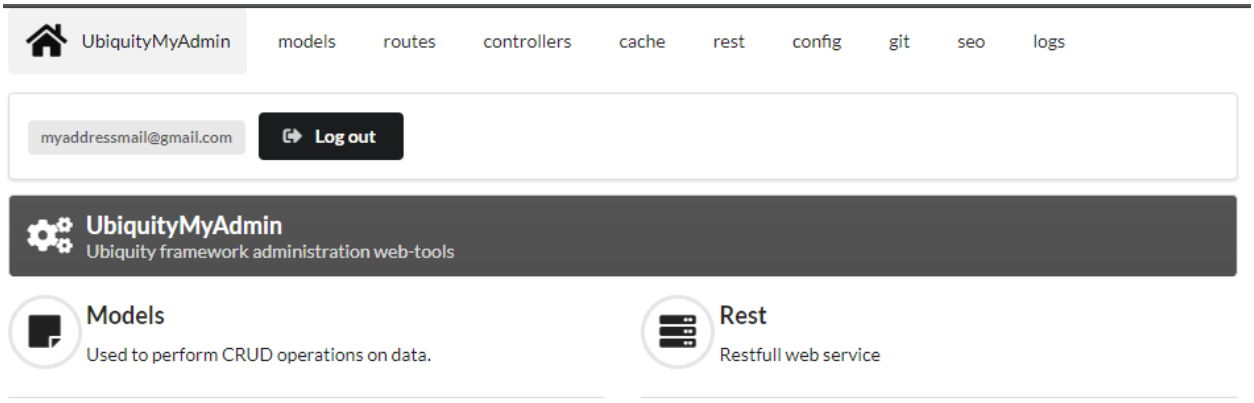
After clicking on **login**:



If the authentication data entered is invalid:



If the authentication data entered is valid:



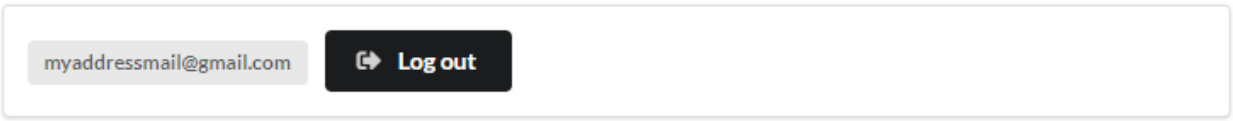
13.2.3 Attaching the zone info-user

Modify the **BaseAuthController** controller:

Listing 4: app/controllers/BaseAuthController.php

```
/**
 * Auth Controller BaseAuthController
 **/
class BaseAuthController extends \Ubiquity\controllers\auth\AuthController{
...
    public function _displayInfoAsString() {
            return true;
    }
}
```

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



It can be displayed in any twig template:

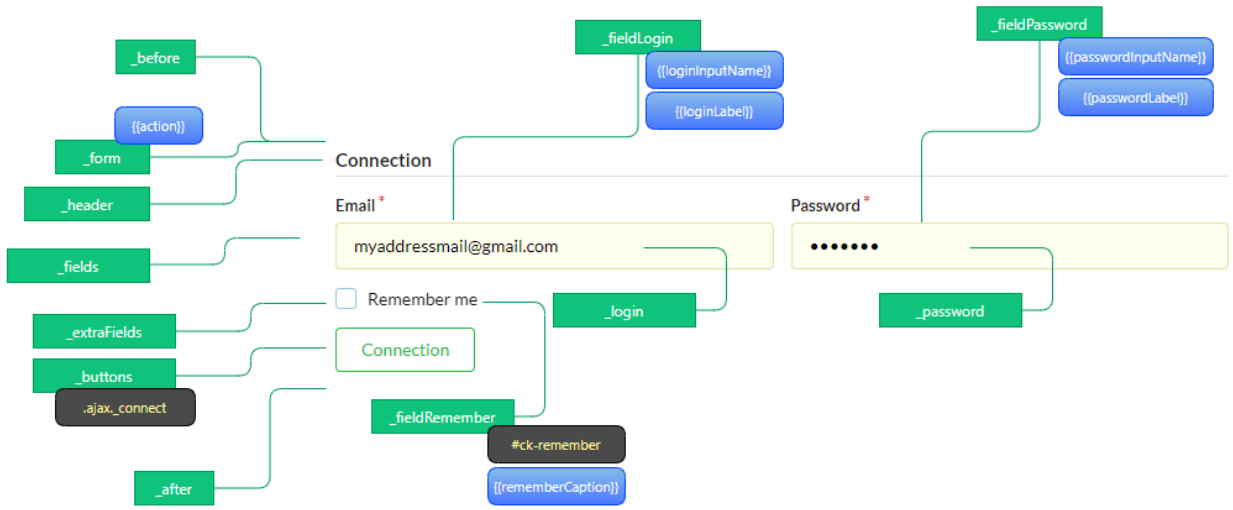
```
{{ _userInfo | raw }}
```

13.3 Description of the features

13.3.1 Customizing templates

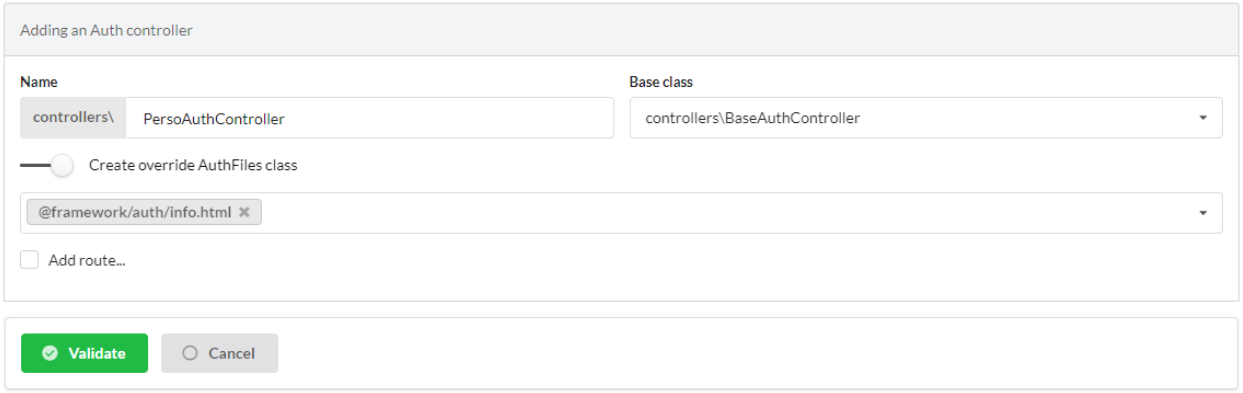
index.html template

The index.html template manages the connection:



Example with the **_userInfo** aera:

Create a new AuthController named **PersoAuthController**:



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

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

```
{% extends "@framework/auth/info.html" %}
{% block _before %}
    <div class="ui tertiary inverted red segment">
{% endblock %}
{% block _userInfo %}
    {{ parent() }}
{% endblock %}
{% block _logoutButton %}
    {{ parent() }}
{% endblock %}
{% block _logoutCaption %}
    {{ parent() }}
{% endblock %}
{% block _loginButton %}
    {{ parent() }}
{% endblock %}
{% block _loginCaption %}
    {{ parent() }}
```

(continues on next page)

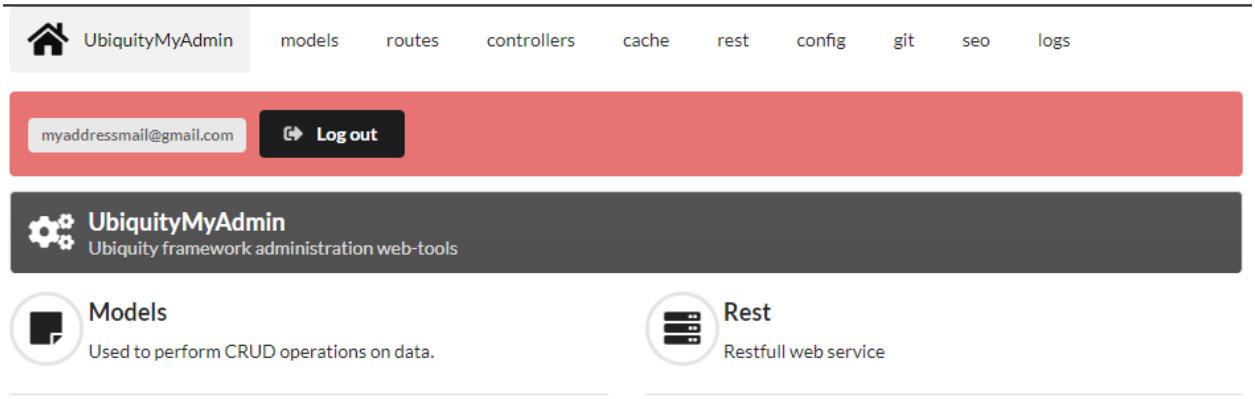
(continued from previous page)

```
{% endblock %}
{% block _after %}
            </div>
{% endblock %}
```

Change the AuthController **Admin** controller:

Listing 6: app/controllers/Admin.php

```
class Admin extends UbiquityMyAdminBaseController{
    use WithAuthTrait;
    protected function getAuthController(): AuthController {
            return new PersoAuthController();
    }
}
```



13.3.2 Customizing messages

Listing 7: app/controllers/PersoAuthController.php

```
class PersoAuthController extends \controllers\BaseAuth{
...
 /**
  * {@inheritDoc}
  * @see \Ubiquity\controllers\auth\AuthController::badLoginMessage()
  */
 protected function badLoginMessage(\Ubiquity\utils\flash\FlashMessage $fMessage) {
     $fMessage->setTitle("Erreur d'authentification");
     $fMessage->setContent("Login ou mot de passe incorrects !");
     $this->_setLoginCaption("Essayer à nouveau");

 }
...
}
```

13.3.3 Self-check connection

Listing 8: app/controllers/PersoAuthController.php

```
class PersoAuthController extends \controllers\BaseAuth{
...
 /**
  * {@inheritDoc}
  * @see \Ubiquity\controllers\auth\AuthController::_checkConnectionTimeout()
  */
 public function _checkConnectionTimeout() {
     return 10000;
 }
...
}
```

13.3.4 Limitation of connection attempts

Listing 9: app/controllers/PersoAuthController.php

```
class PersoAuthController extends \controllers\BaseAuth{
...
 /**
  * {@inheritDoc}
  * @see \Ubiquity\controllers\auth\AuthController::attemptsNumber()
  */
 protected function attemptsNumber() {
     return 3;
 }
...
}
```

CHAPTER 14

Models generation

14.1 From existing database

- with console
- with web-tools

CHAPTER 15

ORM

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.

15.1 Models definition

15.1.1 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  }
```

15.1.2 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)

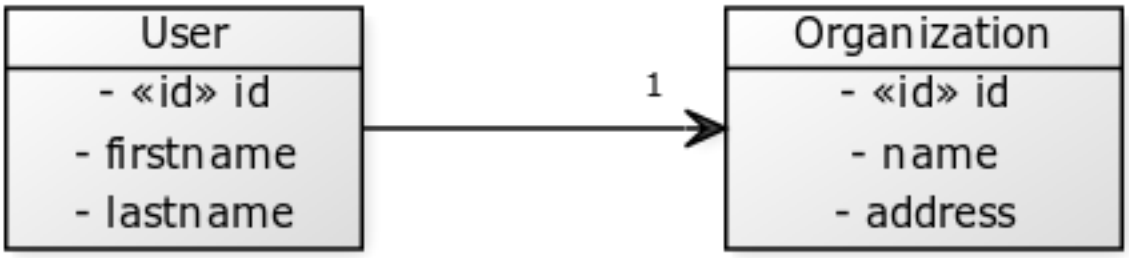
```
12      /**
13      * column("user_name")
14      **/
15      private $firstname;
16
17      public function getFirstname(){
18              return $this->firstname;
19      }
20      public function setFirstname($firstname){
21              $this->firstname=$firstname;
22      }
23  }
```

15.1.3 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"=>
    ↪"idOrganization","nullable"=>false)
14              **/
15              private $organization;
```

(continues on next page)

(continued from previous page)

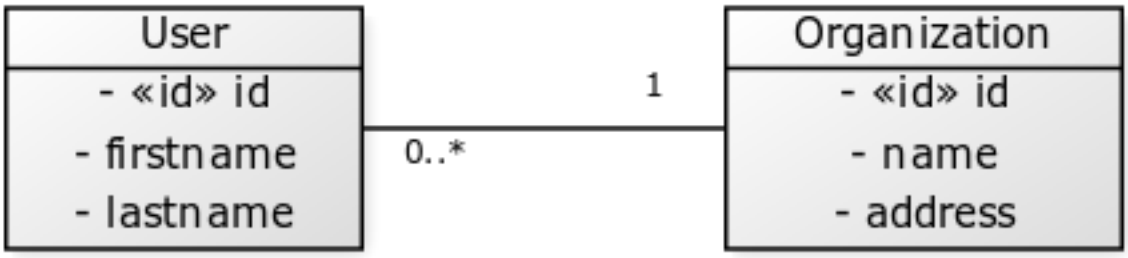
```
16
17              public function getOrganization(){
18                      return $this->organization;
19              }
20
21               public function setOrganization($organization){
22                      $this->organization=$organization;
23              }
24  }
```

The **@joinColumn** annotation specifies that:

- The member **$organization** is an instance of **modelsOrganization**
- The table **user** has a foreign key **idOrganization** refering 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

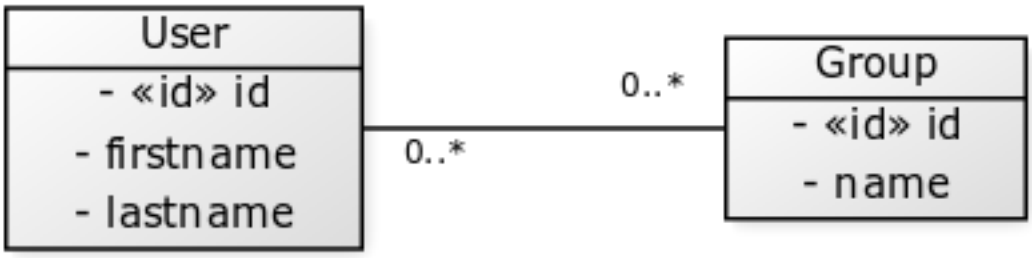
```
1       namespace models;
2
3       class Organization{
4               /**
5                * @id
6               **/
7               private $id;
8
9               private $name;
10
11              /**
12               * @oneToMany("mappedBy"=>"organization","className"=>"models\\User")
13    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

```php
namespace models;

class User{
    /**
     * @id
    **/
    private $id;

    private $firstname;

            /**
             * @manyToMany("targetEntity"=>"models\\Group","inversedBy"=>"users")
             * @joinTable("name"=>"groupusers")
            **/
            private $groups;

}
```

Listing 7: app/models/Group.php

```php
namespace models;

class Group{
    /**
     * @id
    **/
    private $id;

    private $name;

            /**
             * @manyToMany("targetEntity"=>"models\\User","inversedBy"=>"groups")
             * @joinTable("name"=>"groupusers")
            **/
            private $users;

}
```

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

Listing 8: app/models/Group.php

```php
namespace models;

class Group{
    /**
     * @id
    **/
    private $id;

    private $name;

    /**
     * @manyToMany("targetEntity"=>"models\\User","inversedBy"=>"groupes")
     * @joinTable("name"=>"groupeusers",
     * "joinColumns"=>["name"=>"id_groupe","referencedColumnName"=>"id"],
     * "inverseJoinColumns"=>["name"=>"id_user","referencedColumnName"=>"id"])
    **/
    private $users;

}
```

15.2 ORM Annotations

15.2.1 Annotations for classes

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

15.2.2 Annotations for members

@annotation	role	properties	role
@id	Defines the primary key(s).		
@column	Specify the associated field caracteristics.	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.		

15.2.3 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

CHAPTER 16

DAO

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

16.1 Loading data

16.1.1 Loading an instance

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

```
use Ubiquity\orm\DAO;

$user=DAO::getOne("models\User",5);
```

BelongsTo loading

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

Each user belongs to only one category:

```
$user=DAO::getOne("models\User",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("models\User",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:

```
$user=DAO::getOne("models\User",5,["groupes"]);
foreach($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/Products.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:

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

16.1.2 Loading multiple objects

Loading instances of the *User* class:

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

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

```
$users=DAO::getAll("models\User",["groupes","category"]);
foreach($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>";
    }
```

(continues on next page)

(continued from previous page)

```
    echo "</ul>";
}
```

CHAPTER 17

Request

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.

17.1 Retrieving data

17.1.1 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);
```

17.1.2 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();
```

17.2 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 Uniquity\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.

17.3 Testing the request

17.3.1 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 Uniquity\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 }
```

17.3.2 isAjax

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

Listing 3: app/controllers/Users.php

```
...
    public function initialize(){
            if(!URequest::isAjax()){
                    $this->loadView("main/vHeader.html");
            }
    }
    ...
```

17.3.3 isCrossSite

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

CHAPTER 18

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.

18.1 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);
```

18.2 Defining specific headers

18.2.1 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');
```

18.3 Cache

Forcing the disabling of the browser cache:

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

18.4 Accept

Define which content types, expressed as MIME types, the client is able to understand. See Accept default values

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

18.5 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.

18.5.1 Access-Control-Allow-Origin

Setting allowed origin:

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

18.5.2 Access-Control-Allow-methods

Defining allowed methods:

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

18.5.3 Access-Control-Allow-headers

Defining allowed headers:

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

18.5.4 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/');
```

18.6 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
}
```

CHAPTER 19

Session

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.

19.1 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.

19.2 Creating or editing a session variable

```
use Ubiquity\utils\http\USession;

USession::set("name","SMITH");
USession::set("activeUser",$user);
```

19.3 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();
```

19.4 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
}
```

19.5 Deleting variables

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

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

19.6 Explicit closing of the session

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

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

CHAPTER 20

Cookie

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.

20.1 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
}
```

20.2 Retrieving a Cookie

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

20.2.1 Testing the existence

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

20.2.2 Using a default value

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

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

20.3 Deleting a cookie

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

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

20.4 Deleting all cookies

Deleting all cookies from the entire domain:

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

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

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

CHAPTER 21

Views

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.

21.1 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  }
```

21.1.1 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

```
namespace controllers;

class Users extends BaseController{
    ...
    public function info(){
                    $this->loadDefaultView();
            }
    }
}
```

21.2 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

```
namespace controllers;

class Users extends BaseController{
    ...
    public function display($message,$type){
                    $this->loadView("users/display.html",["message"=>$message,"type
↪"=>$type]);
            }
    }
}
```

In this case, it is usefull to call Compact for creating an array containing variables and their values :

Listing 4: app/controllers/Users.php

```
namespace controllers;

class Users extends BaseController{
    ...
    public function display($message,$type){
                    $this->loadView("users/display.html",compact("message","type"));
            }
    }
}
```

21.3 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'] }}
```

21.4 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.

CHAPTER 22

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+...'}) }}
```

CHAPTER 23

Themes

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**.

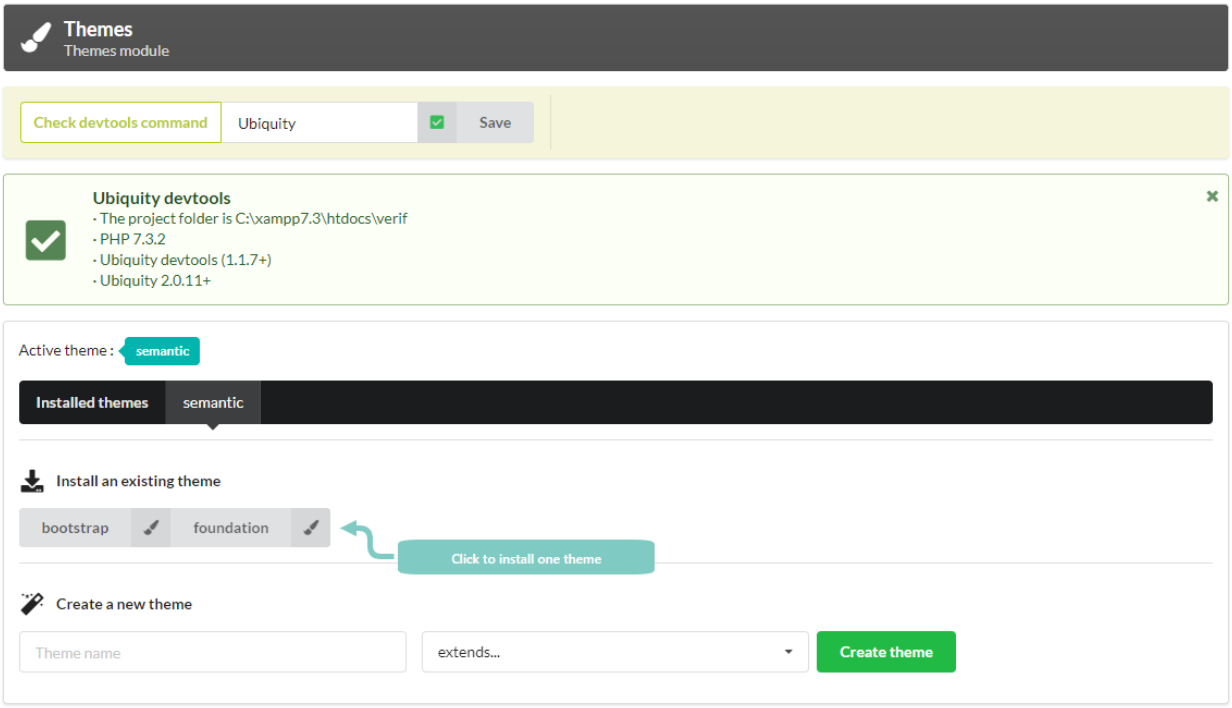
23.1 Installing a theme

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) :



23.2 Creating a new theme

With devtools, run :

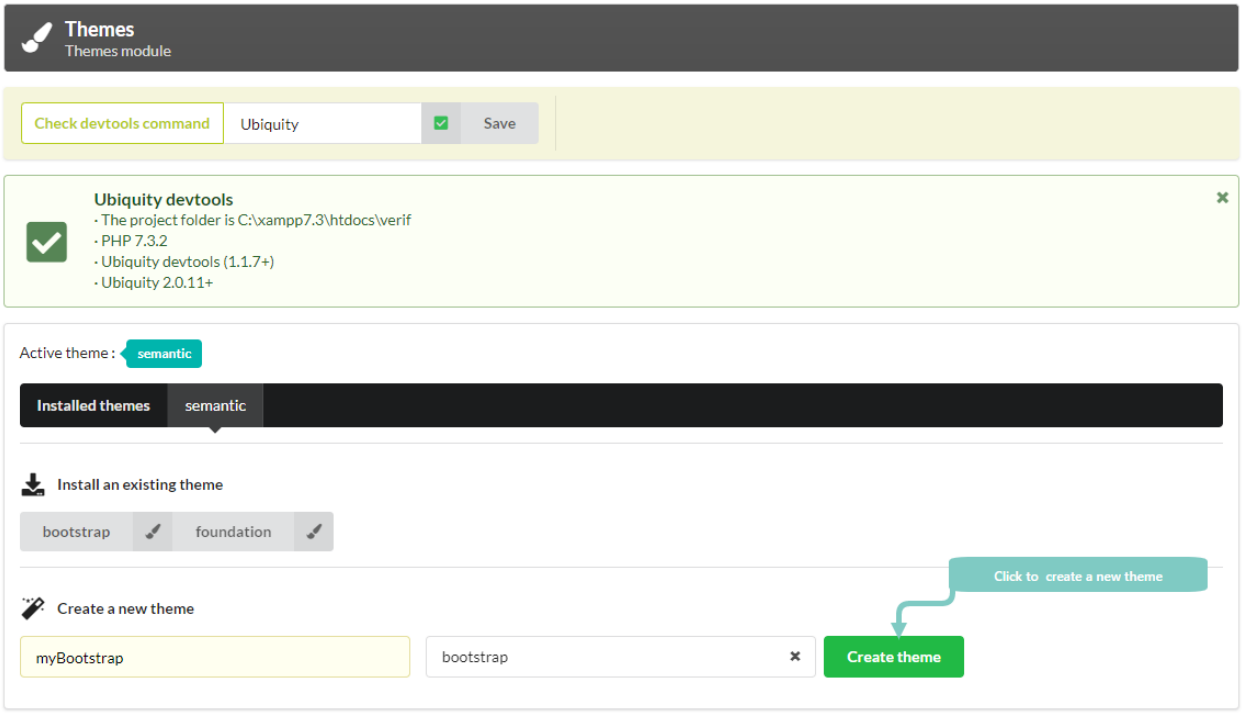
```
Ubiquity create-theme myTheme
```

Creating a new theme from Bootstrap, Semantic...

With devtools, run :

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

With **webtools** :



23.3 Theme functioning and structure

23.3.1 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)

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

        public function initialize() {
                if (! URequest::isAjax ()) {
                        $this->loadView ( $this->headerView );
                }
        }
        public function finalize() {
                if (! URequest::isAjax ()) {
                        $this->loadView ( $this->footerView );
                }
        }
}
```

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
```

23.4 Change of the active theme

23.4.1 Persistent change

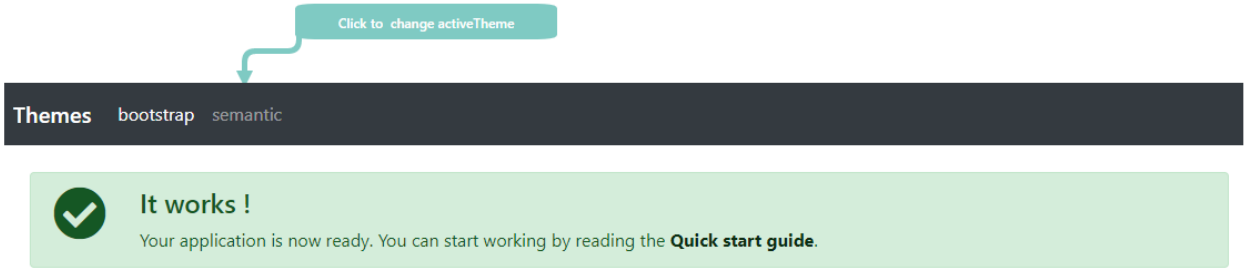
activeTheme is defined in `app/config/config.php` with `templateEngineOptions => array("activeTheme" => "semantic")`

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');
```

23.4.2 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::intialize();
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)

```
            ...
        }
    }
```

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

```
use Ubiquity\themes\ThemesManager;
use Ubiquity\utils\http\URequest;

...

ThemesManager::onBeforeRender(function(){
            if(URequest::get("th")=='bootstrap'){
                    ThemesManager::setActiveTheme("bootstrap");
            }
    });
```

23.5 View and assets loading

23.5.1 Views

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

Listing 5: app/controllers/Users.php

```
namespace controllers;

class Users extends BaseController{

        public function action(){
            $this->loadView('@activeTheme/action.html');
            ...
        }
    }
```

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

23.5.2 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 loacated in `app/views/themes/bootstrap/Users/display.html`.

Listing 6: app/controllers/Users.php

```
namespace controllers;

```

(continues on next page)

(continued from previous page)

```
class Users extends BaseController{

        public function display(){
            $this->loadDefaultView();
            ...
        }
    }
```

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
```

23.6 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/`.

23.7 Css compilation

CHAPTER 24

Normalizers

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

CHAPTER 25

Validators

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.

25.1 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.

25.2 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`.

25.3 Validating instances

25.3.1 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 :

The **validate** method returns an array of **ConstraintViolation** instances.

25.3.2 multiple instances

```
public function testValidateAuthors(){
        $authors=DAO::getAll(Author::class);
        $violations=ValidatorsManager::validateInstances($author);
        foreach($violations as $violation){
                echo $violation.'<br>';
        }
}
```

25.4 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
```

(continues on next page)

(continued from previous page)

```
     * @column("name"=>"id","nullable"=>false,"dbType"=>"int(11)")
     * @validator("id","constraints"=>array("autoinc"=>true))
    **/
    private $id;

    /**
     * @column("name"=>"firstname","nullable"=>false,"dbType"=>"varchar(65)")
     * @validator("length","constraints"=>array("max"=>65,"notNull"=>true))
    **/
    private $firstname;

    /**
     * @column("name"=>"lastname","nullable"=>false,"dbType"=>"varchar(65)")
     * @validator("length","constraints"=>array("max"=>65,"notNull"=>true))
    **/
    private $lastname;

    /**
     * @column("name"=>"email","nullable"=>false,"dbType"=>"varchar(255)")
     * @validator("email","constraints"=>array("notNull"=>true))
     * @validator("length","constraints"=>array("max"=>255))
    **/
    private $email;

    /**
     * @column("name"=>"password","nullable"=>true,"dbType"=>"varchar(255)")
     * @validator("length","constraints"=>array("max"=>255))
    **/
    private $password;

    /**
     * @column("name"=>"suspended","nullable"=>true,"dbType"=>"tinyint(1)")
     * @validator("isBool")
    **/
    private $suspended;
}
```

These validators can then be modified. [br| Modifications must always be folowed 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
```

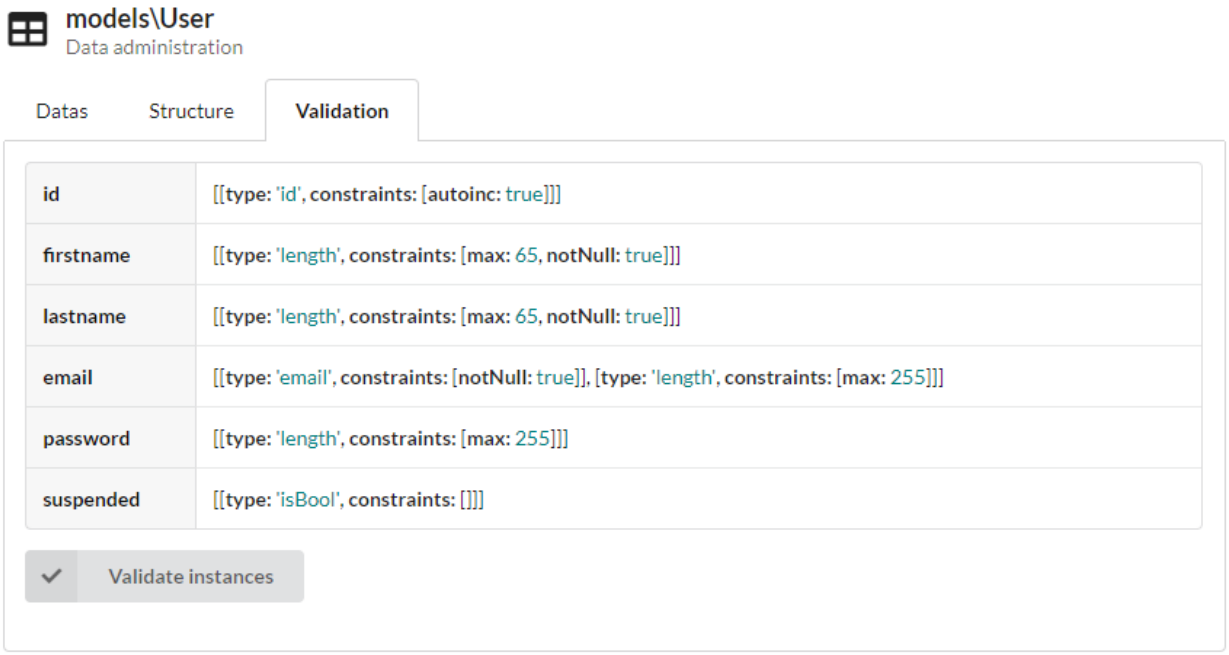
field	value
id	· [type: 'id',constraints: [autoinc: true]]
firstname lastname	· [type: 'length',constraints: [max: 65,notNull: true]] ≡
email	· [type: 'email',constraints: [notNull: true]] · [type: 'length',constraints: [max: 255]]
password	· [type: 'length',constraints: [max: 255]]
suspended	· [type: 'isBool',constraints: []]

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:



25.5 Validator types

25.5.1 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}	

25.5.2 Comparison

25.5.3 Dates

25.5.4 Multiples

25.5.5 Strings

CHAPTER 26

Translation module

Note: The Translation module uses the static class **TranslatorManager** to manage translations.

Important: This module is under development. It is operational for the use of existing translation files. It still lacks the module to manage the translation files in the administration part (devtools or webtools).

26.1 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.']
];
```

26.2 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.

26.2.1 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');
```

26.2.2 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');
```

26.3 Defining the root translations dir

If the **rootDir** parameter is missing, the default directory used is `app/translations`.

Listing 5: app/config/services.php

```
Ubiquity\cache\CacheManager::startProd($config);
Ubiquity\translation\TranslatorManager::start('fr_FR','en_EN','myTranslations');
```

26.4 Make a translation

26.4.1 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%!',...];
```

26.4.2 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) }}
```

CHAPTER 27

Rest

//TODO

CHAPTER 28

Contributing

28.1 System requirements

Before working on Ubiquity, setup your environment with the following software:

- Git
- PHP version 7.1 or above.

28.2 Get Ubiquity source code

On Ubiquity github repository :

- Click *Fork* Ubiquity project
- Clone your fork locally:

```
git clone git@github.com:USERNAME/ubiquity.git
```

28.3 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.

28.3.1 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
```

28.3.2 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, fonctional 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.

28.4 Submit your Patch

Update the [Unrelease] part of the CHANGELOG.md file by integrating your changes into the appropriate parts:

- Added
- Changed
- Removed
- Fixed

Eventualy 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
```

28.5 Make a Pull Request

You can now make a pull request on Ubiquity github repository .

CHAPTER 29

Coding guide

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.

29.1 Design choices

29.1.1 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 member of a class.

In addition to this problematic resource consumption, the dependency injection poses another problem during development. Access to injected resources returns an element without type, not easy to handle for the developer.

For example, It's hard to manipulate the untyped return of `$this->serviceContainer->get('translator')`, and know which methods to call on it.

The use of classes with static methods avoids all the disadvantages mentioned above:

For a developer, the `TranslatorManager` class is accessible from an entire project. It exposes its public interface and allows code completion.

29.2 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

29.3 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

29.3.1 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;

29.3.2 Code duplications

Absolutely avoid duplication of code, except if duplication is minimal, and is justified by performance.

29.3.3 Bugs

Try to solve all the bugs reported as you go, without letting them accumulate.

29.4 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.

29.5 Code Documentation

//TODO

29.6 Coding standards

Ubiquity coding standards are based on the PSR-1 , PSR-2 and PSR-4 standards, so you may already know most of them.

29.6.1 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, TanslatorManager);
- 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'

29.6.2 Indentation, tabs, braces

- Use Tabs, not spaces;
- Use brace always on the same line;
- Use braces to indicate control structure body regardless of the number of statements it contains;

29.6.3 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;

29.6.4 Operators

- Use identical comparison and equal when you need type juggling;

Example

```php
<?php
namespace Ubiquity\namespace;

use Ubiquity\othernamespace\Foo;

/**
 * Class description.
 * Ubiquity\namespace$Example
 * This class is part of Ubiquity
 *
 * @author jcheron <myaddressmail@gmail.com>
 * @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: If you work with Eclipse, you can import this standardization file that integrates all these rules: `phpMv-coding-standards.xml`

CHAPTER 30

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 31

External libraries

CHAPTER 32

Ubiquity Caching

CHAPTER 33

Ubiquity dependencies

CHAPTER 34

Indices and tables

- genindex
- modindex
- search