micro-framework Documentation

Release 2.0.9

phpmv

Feb 22, 2019

Quick-starts

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 with Semantic-UI integration

```
Ubiquity new quick-start -q=semantic
```

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:

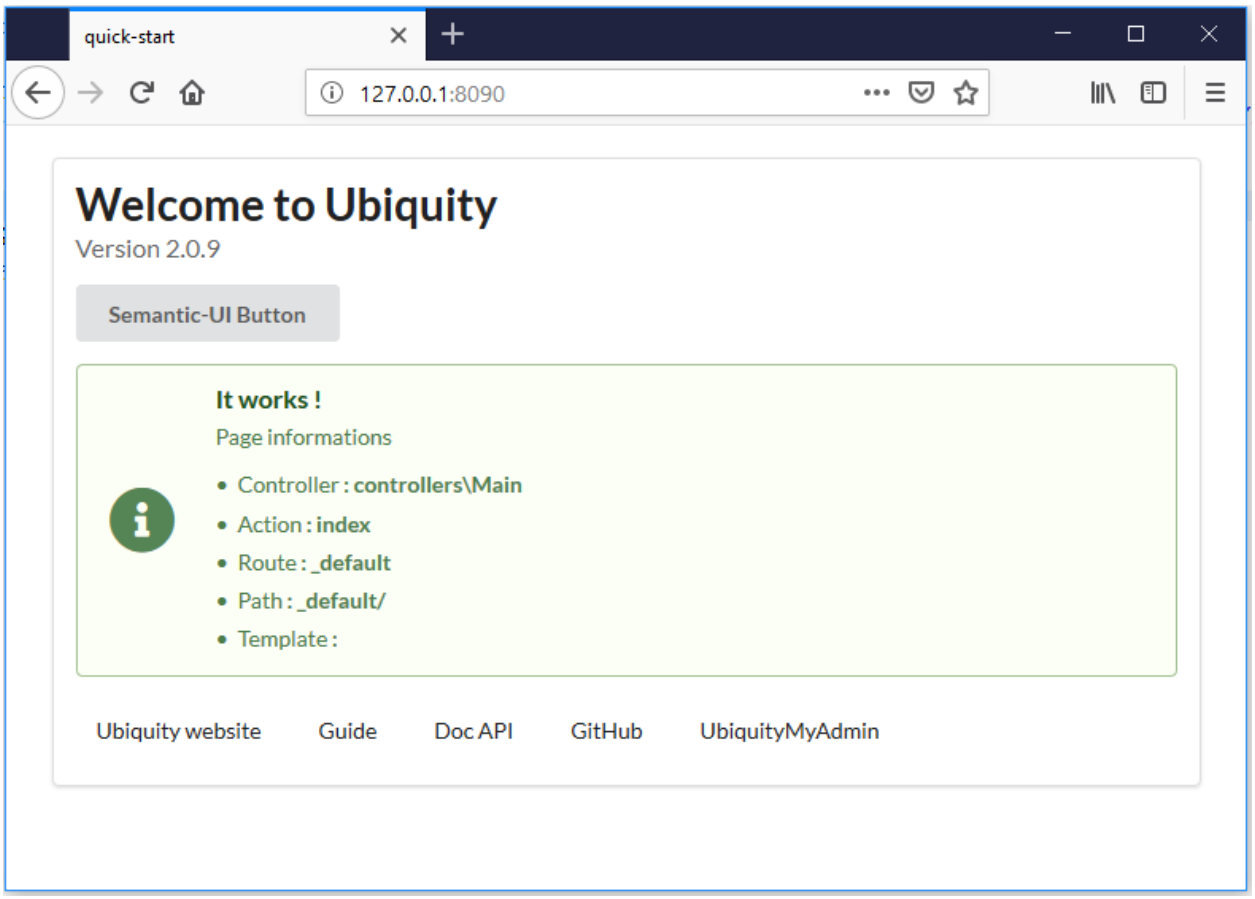
```
app
  ├ cache
  ├ config
  ├ controllers
  ├ models
  └ views
```

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 our 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) and Semantic-UI integration

```
Ubiquity new quick-start -q=semantic -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:

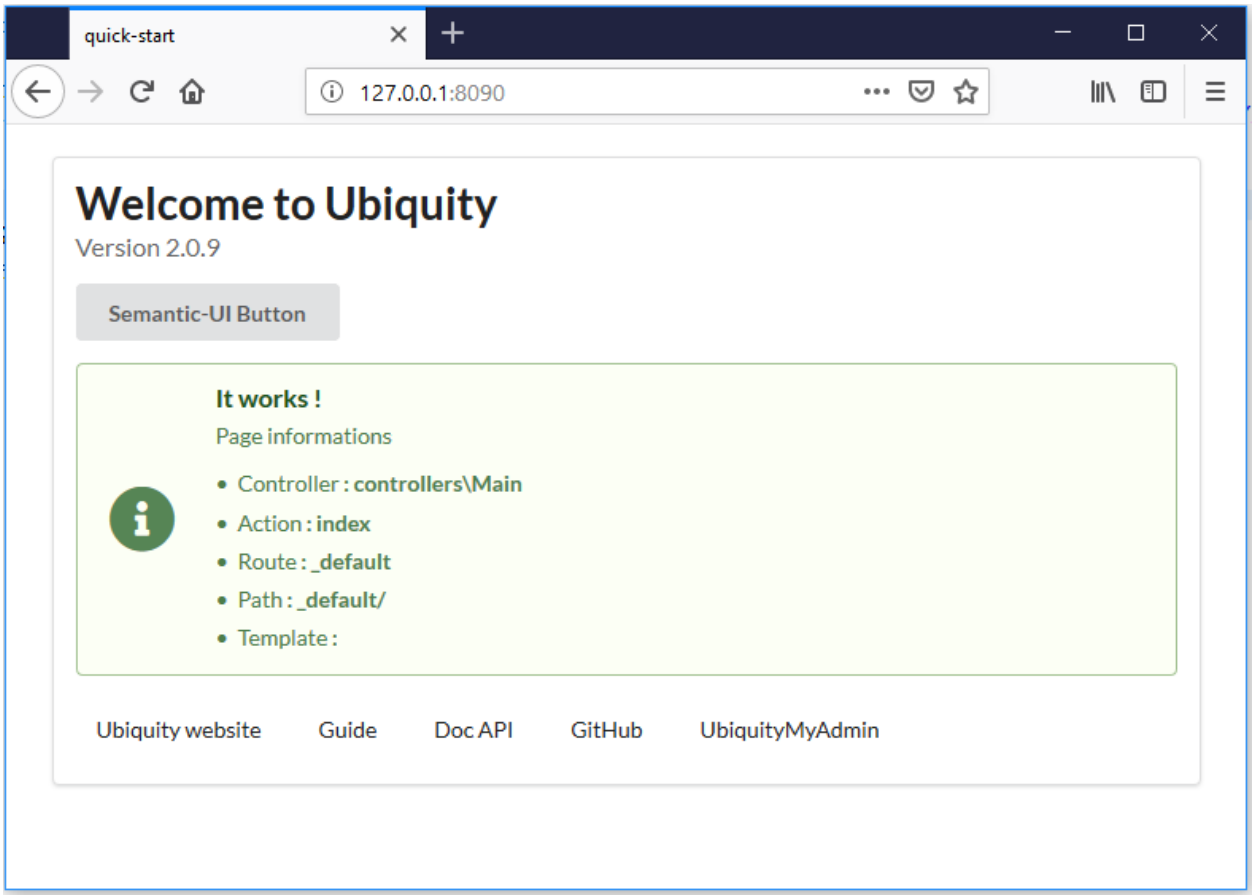
```
app
  ├ cache
  ├ config
  ├ controllers
  ├ models
  └ views
```

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**:

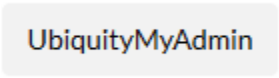


Note: If port 8090 is busy, you can start the server on another port using -p option.

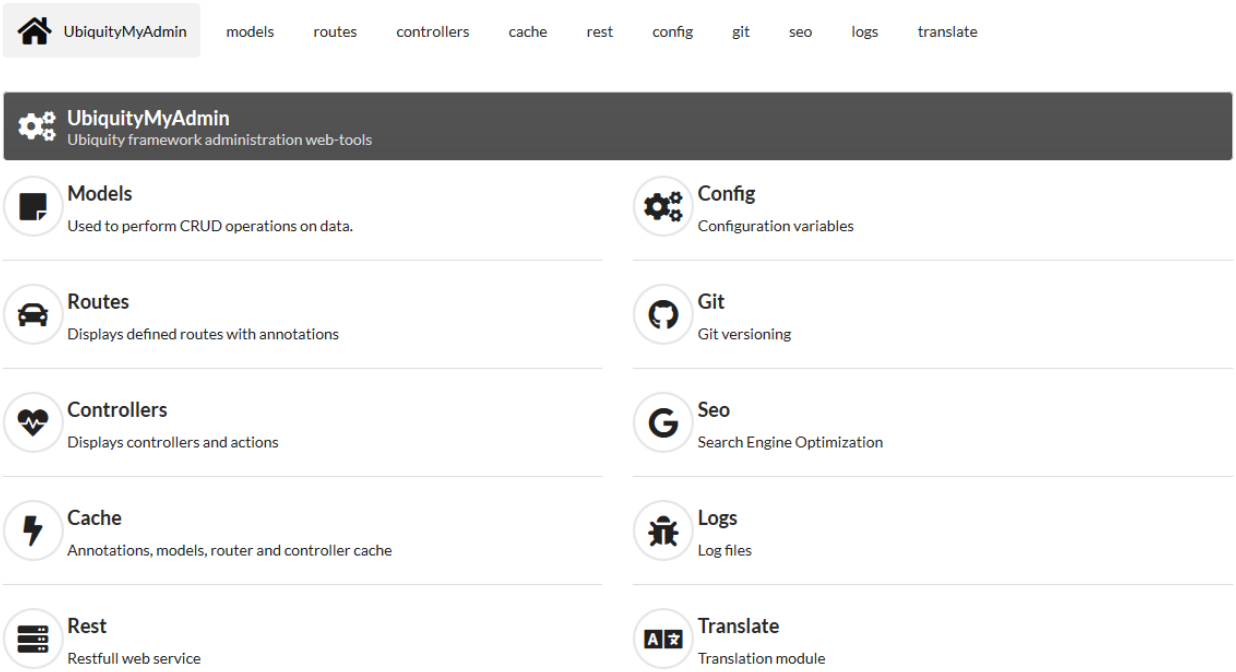
```
Ubiquity serve -p=8095
```

2.6 Controller

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



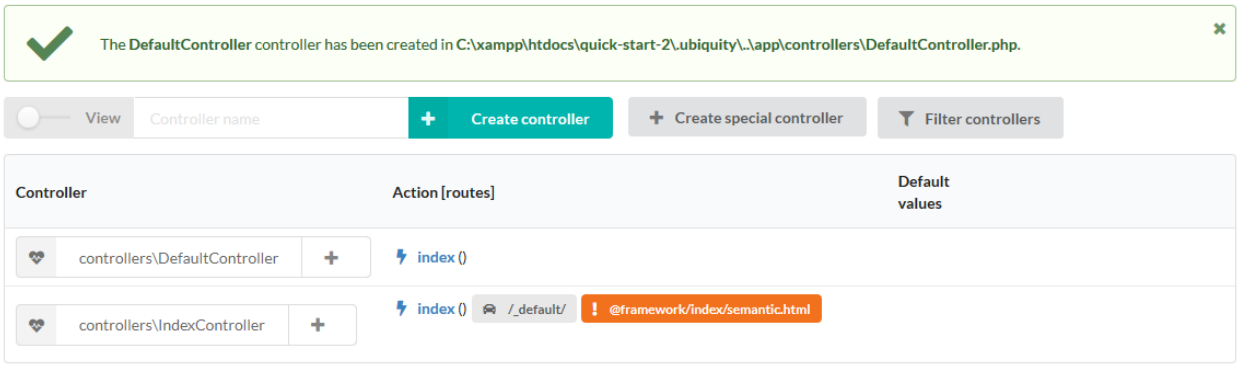
The web application **UbiquityMyAdmin** saves time in repetitive operations.



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     }
```

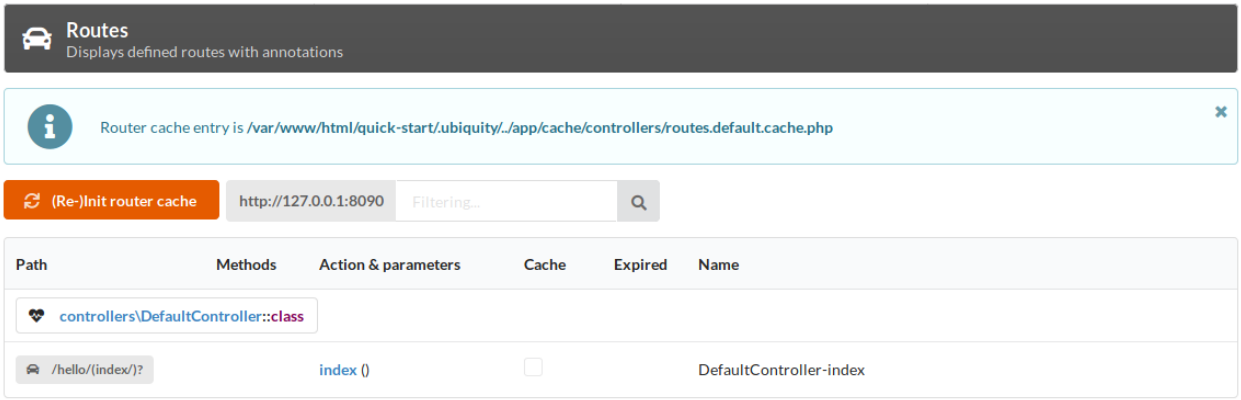
2.7.1 Router cache

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:



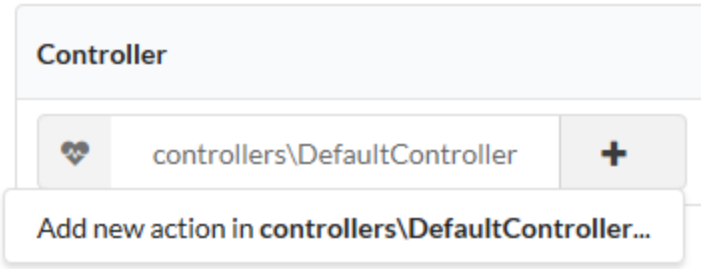
We can now test the page by clicking on the **GET** button or by going to the address `http://127.0.0.1:8090/hello`

2.8 Action & route with parameters

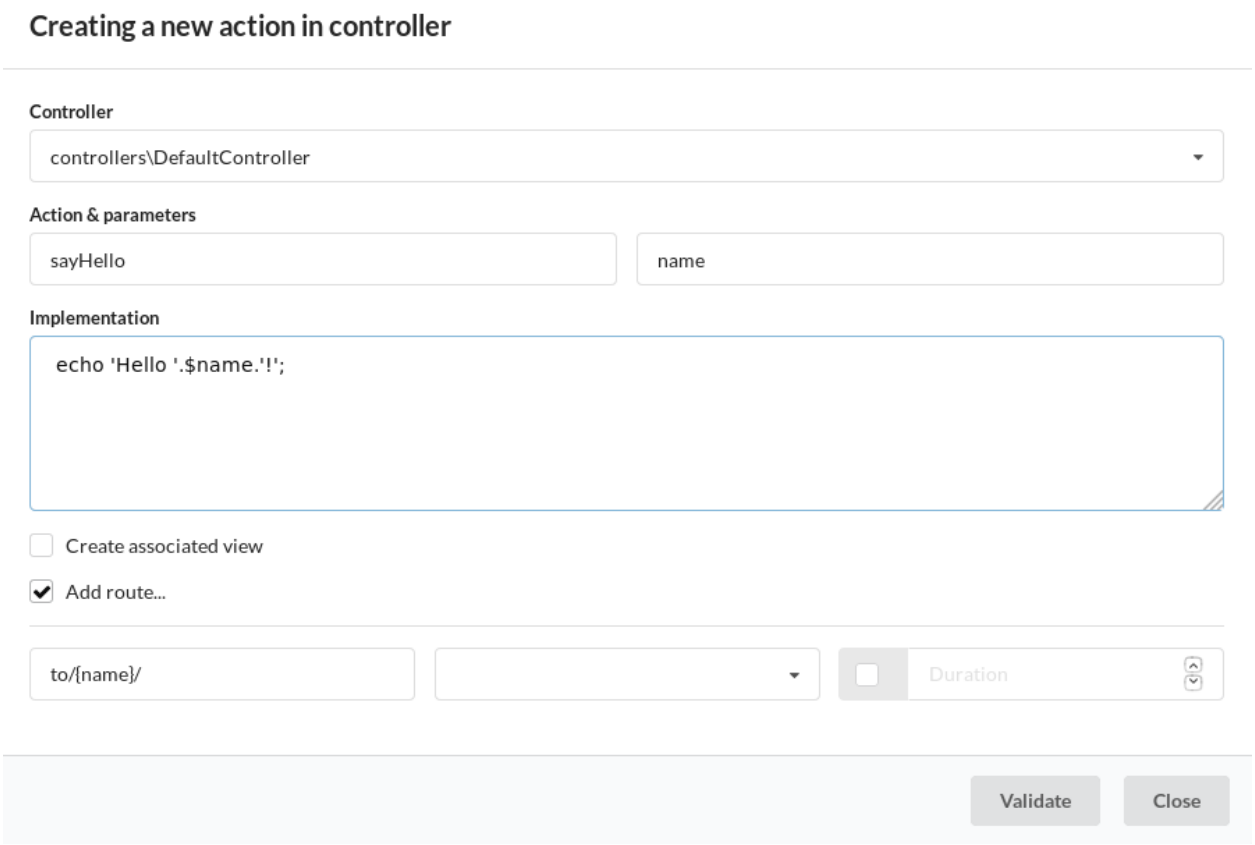
We will now create an action (sayHello) with a parameter (name), and the associated route (to): The route will use the parameter **name** of the action:

Go to the **Controllers** section:

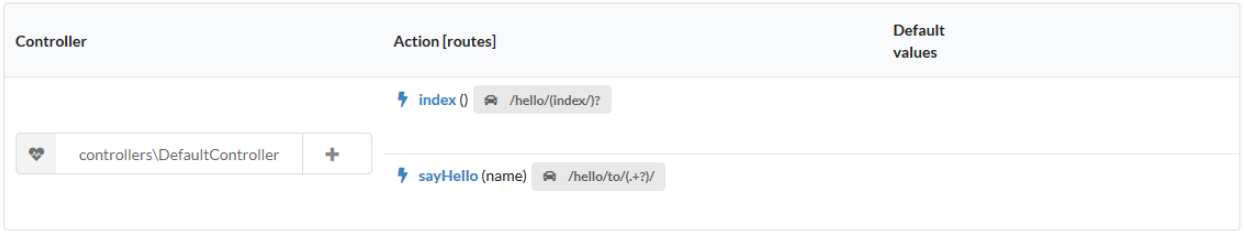
- click on the + button associated with DefaultController,
- then select **Add new action in..** item.



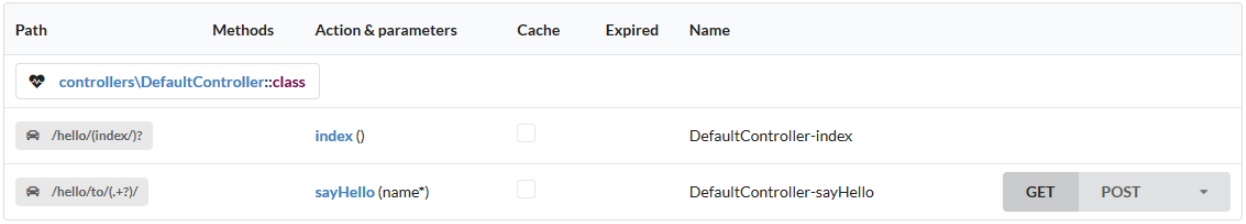
Enter the action information in the following form:



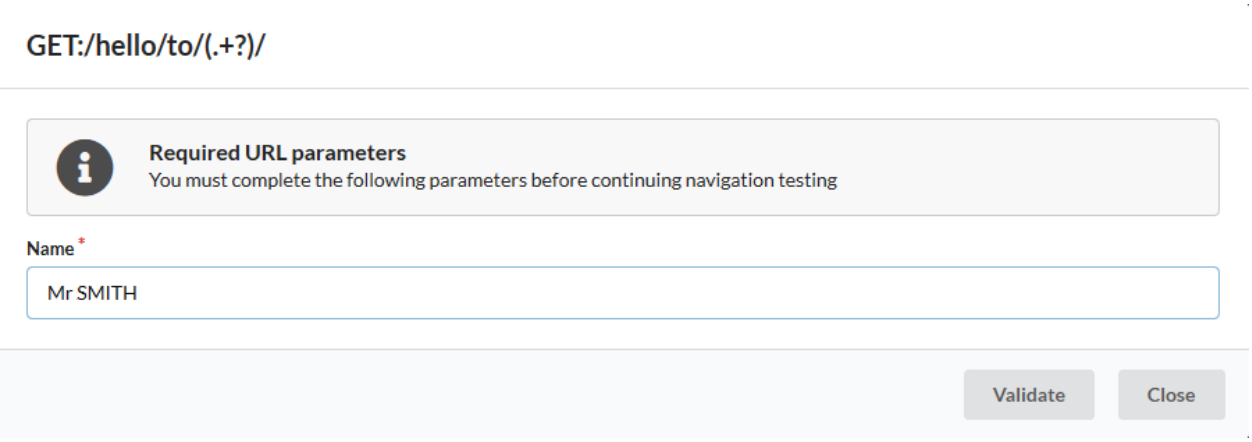
After re-initializing the cache with the orange button, we can see the new route **hello/to/{name}**:



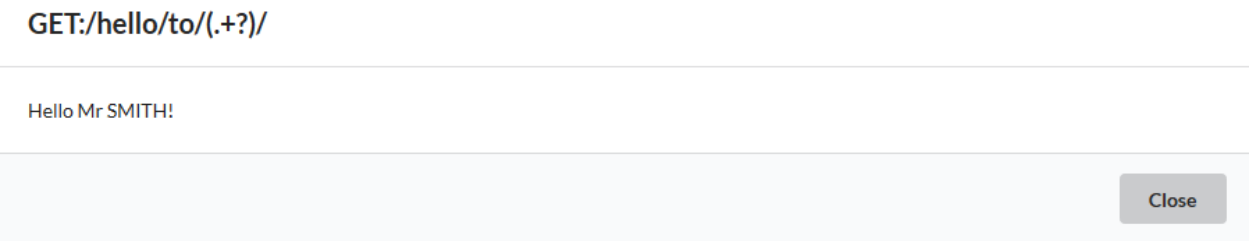
Check the route creation by going to the Routes section:



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



We can see the result:

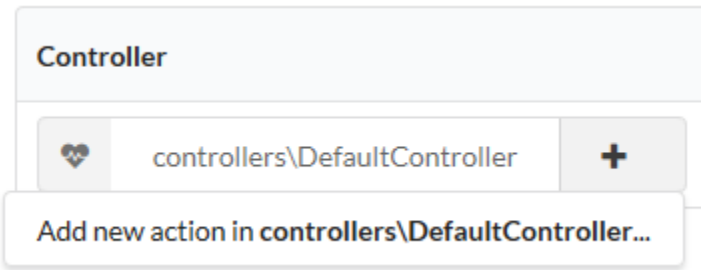


We could directly go to `http://127.0.0.1:8090/hello/to/Mr SMITH` address to test

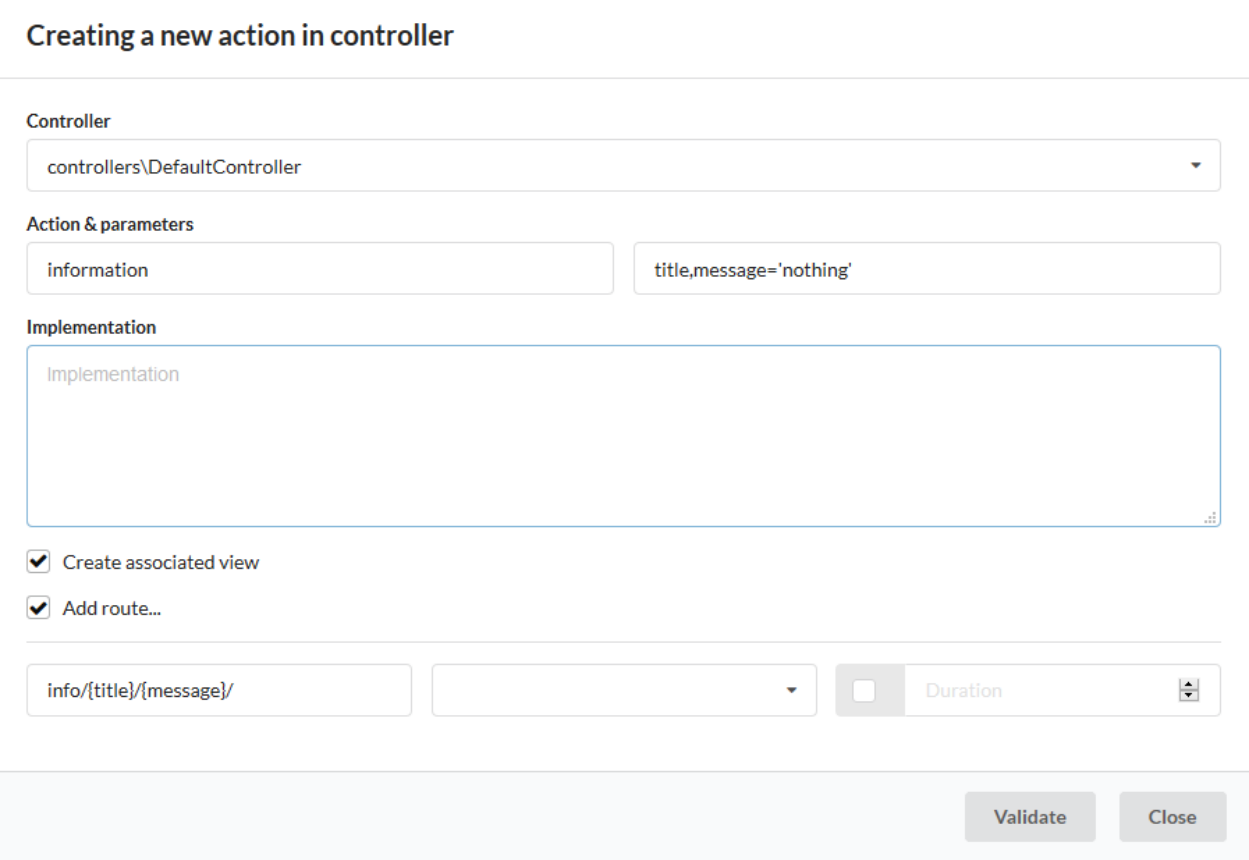
2.9 Action, route parameters & view

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**:

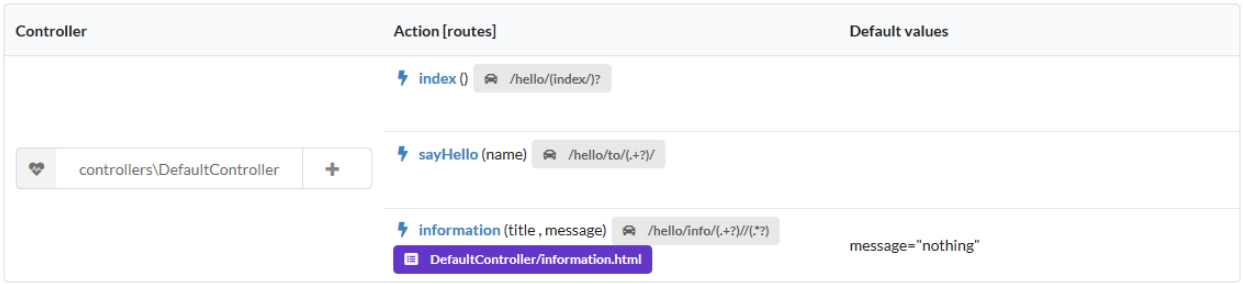


Enter the action information in the following form:



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

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 -q=semantic
```

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 Semantic-UI integration

```
Ubiquity new projectName -q=semantic
```

A project with UbiquityMyAdmin interface and Semantic-UI integration

```
Ubiquity new projectName -q=semantic -a
```

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.	''	
q	phpmv	Integrates phpMv-UI toolkit.	false	semantic,bootstrap,ui
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 phpMv-toolkit

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

4.4 Testing

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

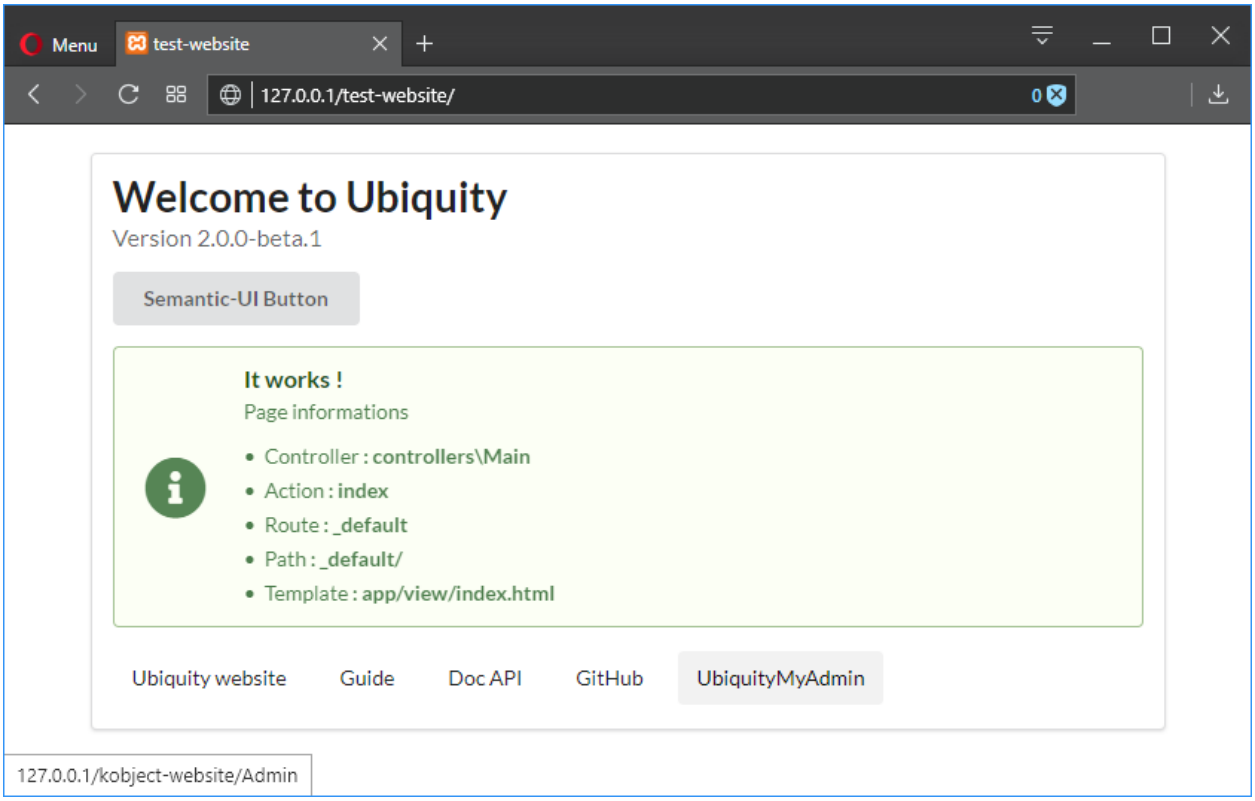
```
Ubiquity serve
```

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;

    /*if($config["test"]){
    \Ubiquity\log\Logger::init($config);
     $config["siteUrl"]="http://127.0.0.1:8090/";
    }*/

    \Ubiquity\cache\CacheManager::startProd($config);
    \Ubiquity\orm\DAO::startDatabase($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 / {
    try_files $uri $uri/ /index.php?c=$query_string;
}
```

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/SCS
```

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

```
namespace controllers;
class First extends ControllerBase{

    public function hello(){
            echo "Hello world!";
    }


}
```

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

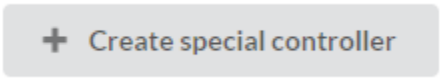
CRUD Controllers

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

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

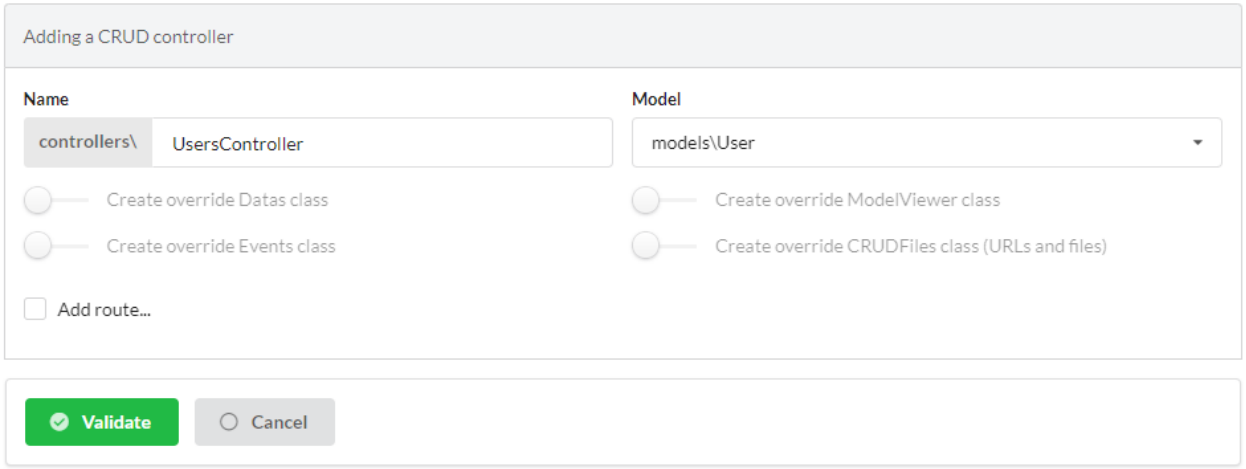
10.1 Creation

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



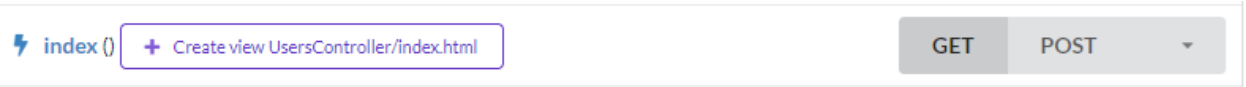
10.2 Description of the features

The generated controller:

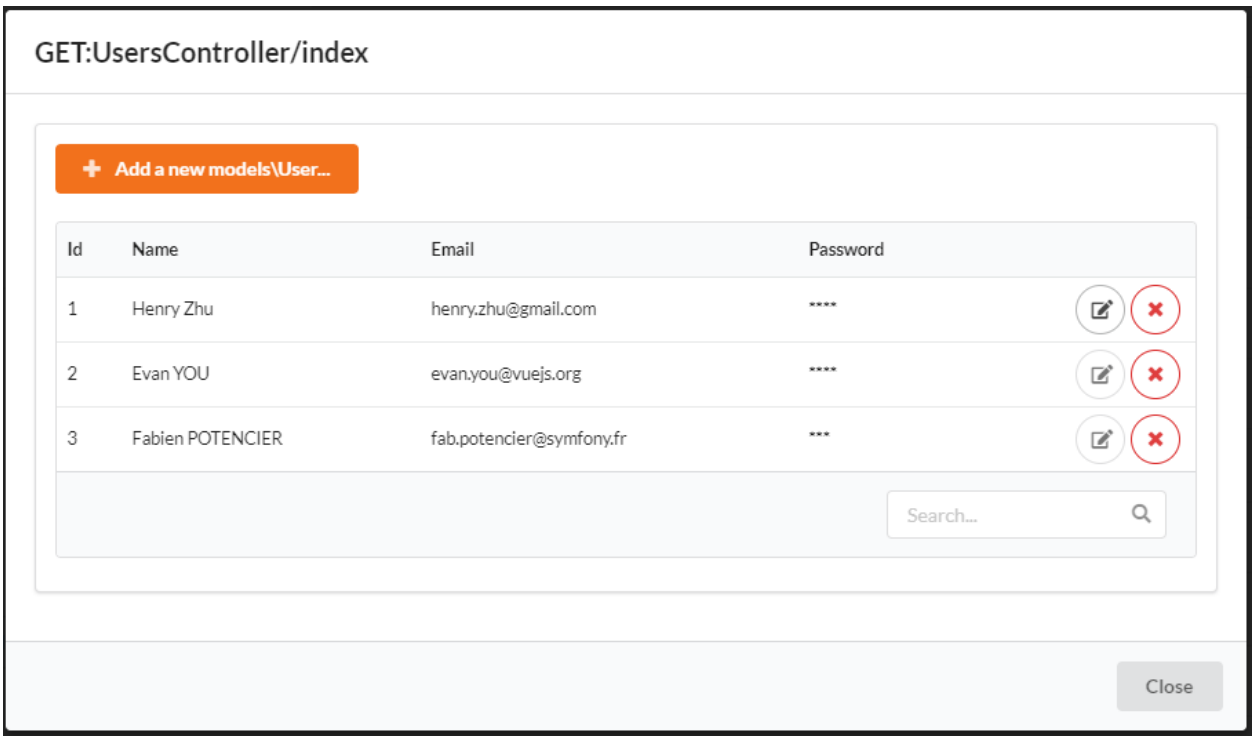
Listing 1: app/controllers/Products.php

```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';
    }
}
```

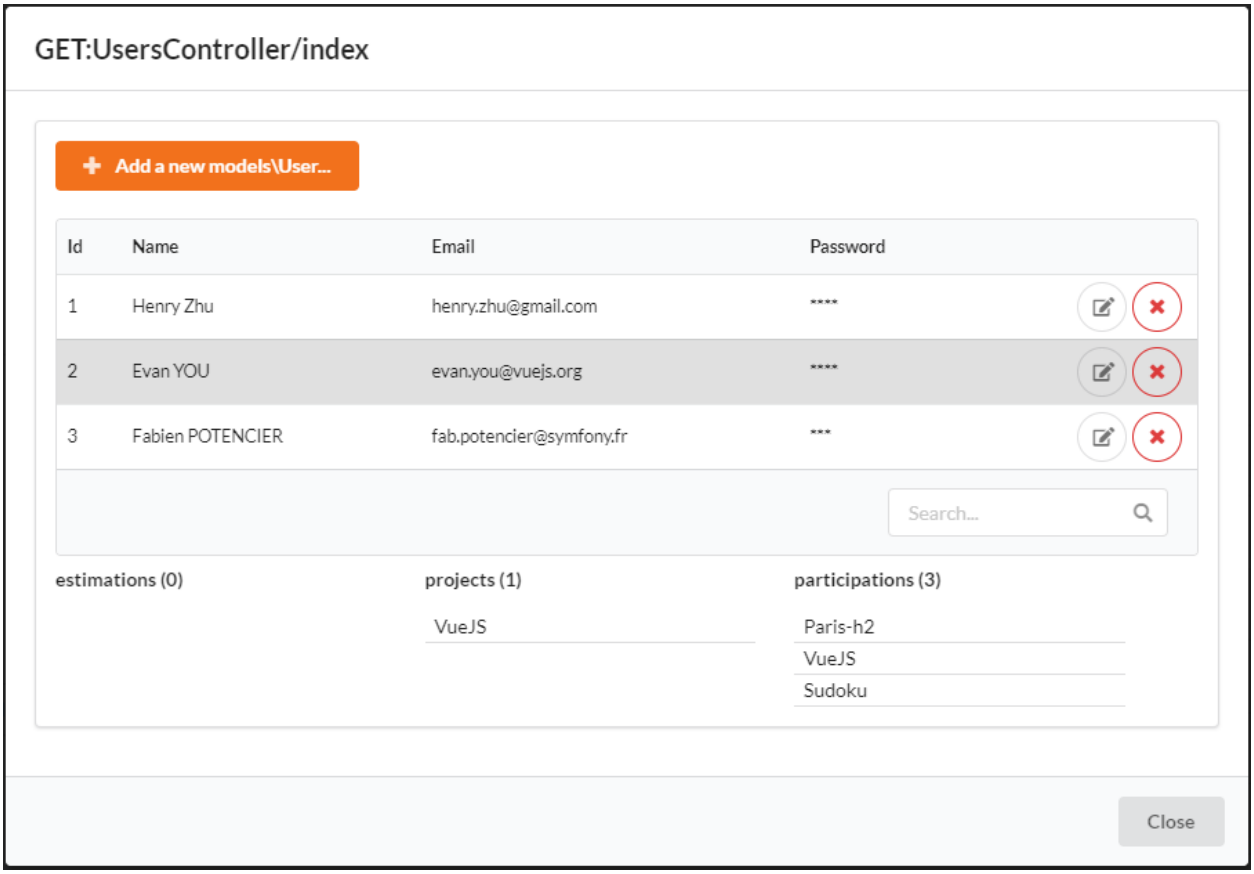
Test the created controller by clicking on the get button in front of the **index** action:



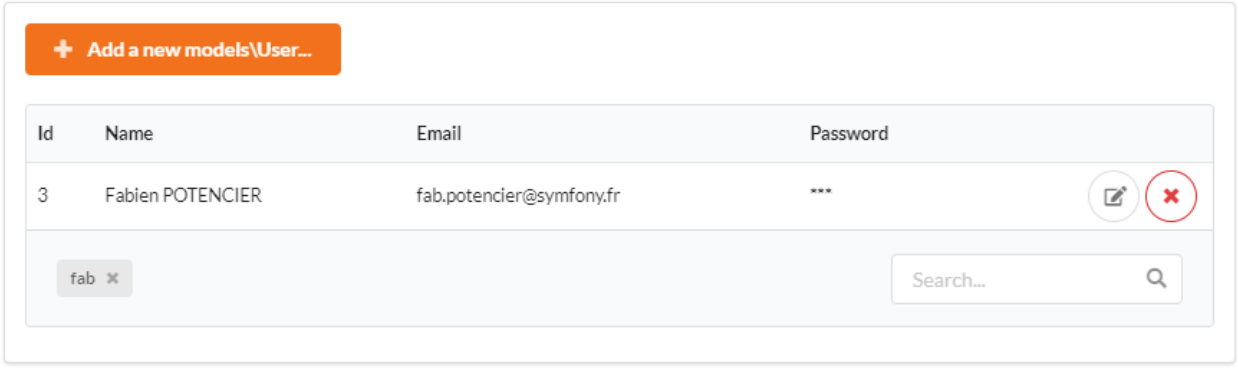
10.2.1 Read (index action)



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



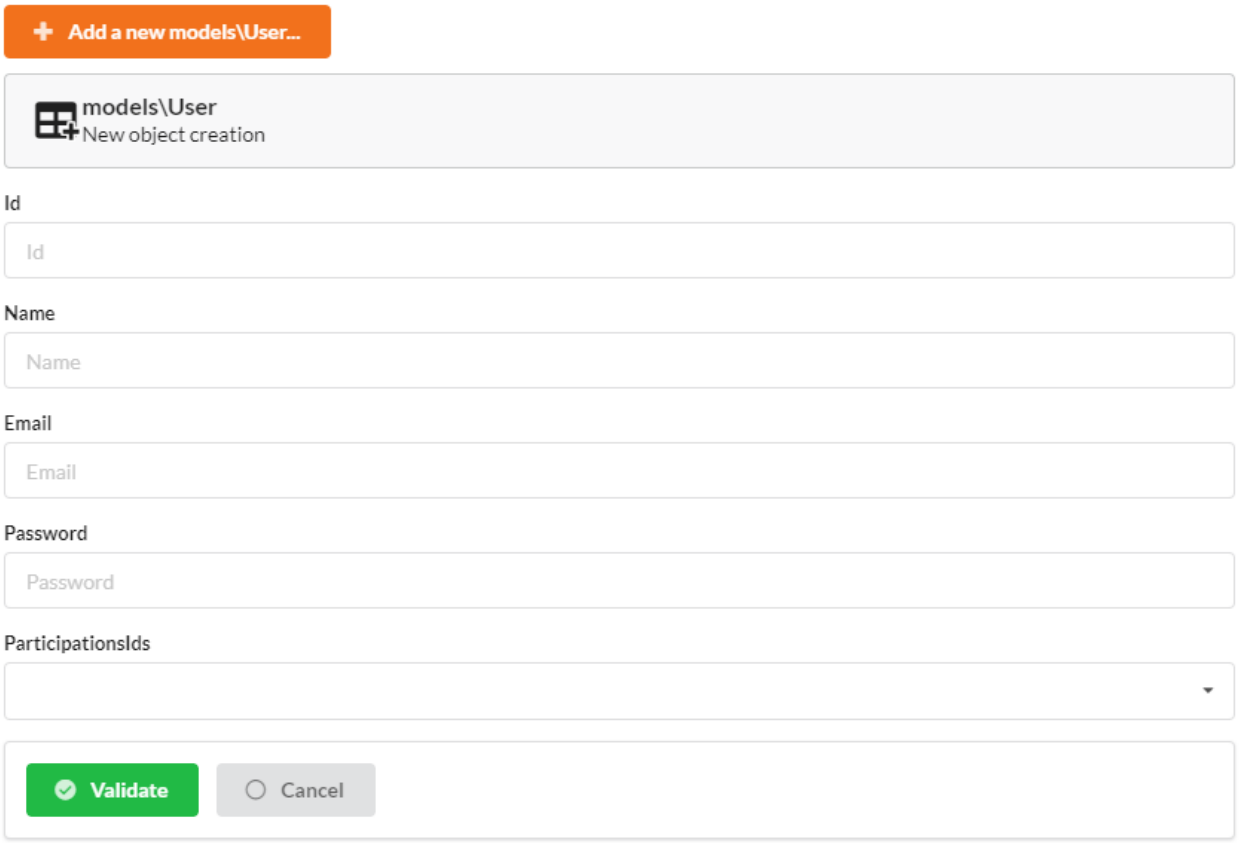
Using the search area:



10.2.2 Create (newModel action)

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

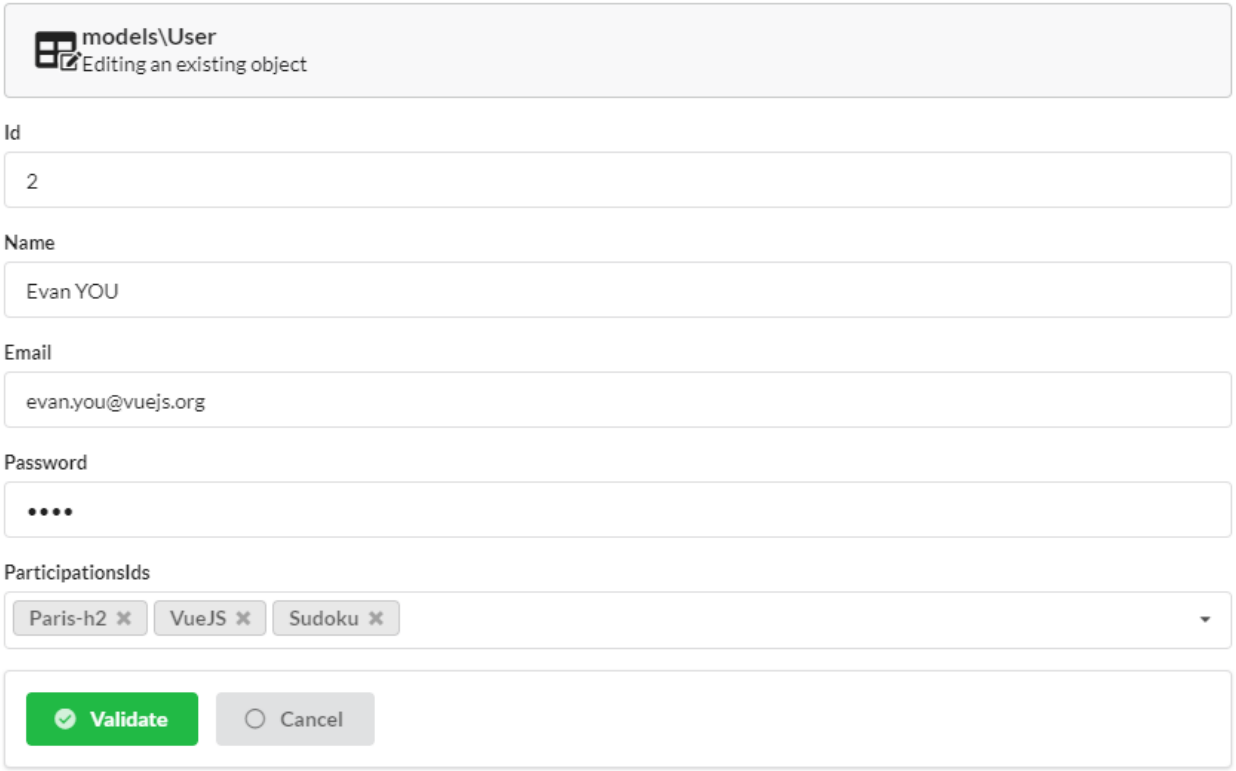
The default form for adding an instance of User:



10.2.3 Update (update action)

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

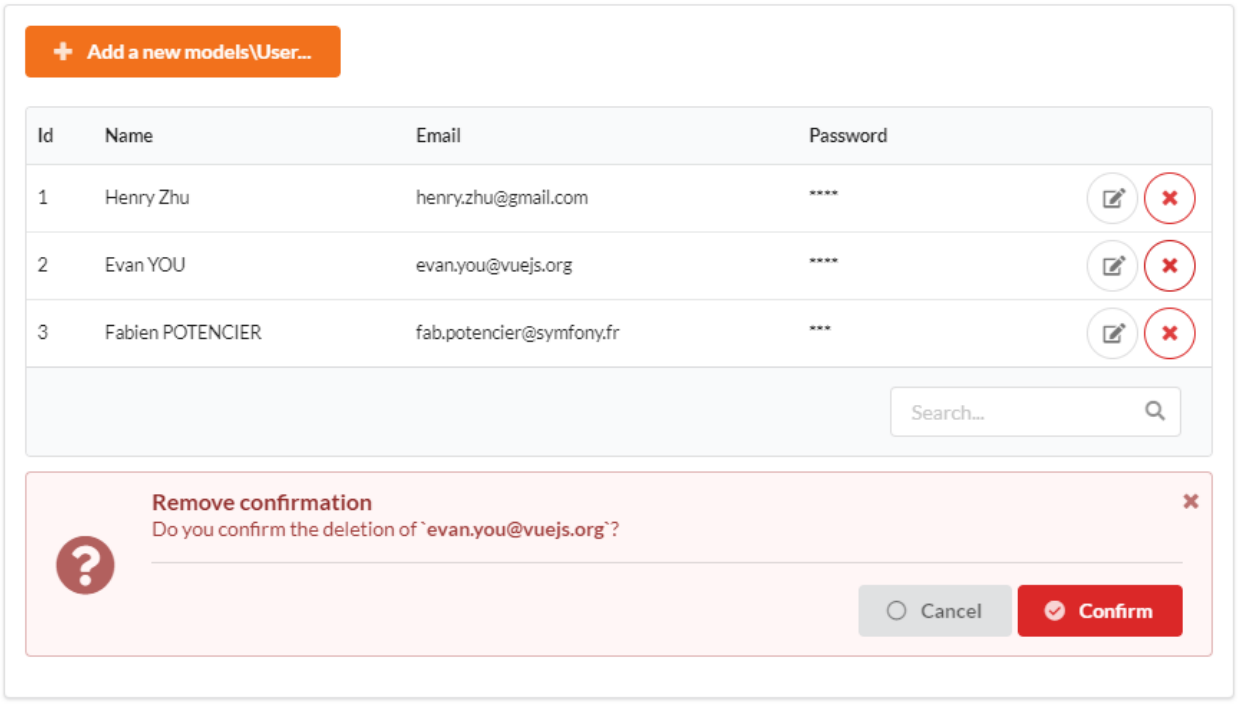
The default form for adding an instance of User:



10.2.4 Delete (delete action)

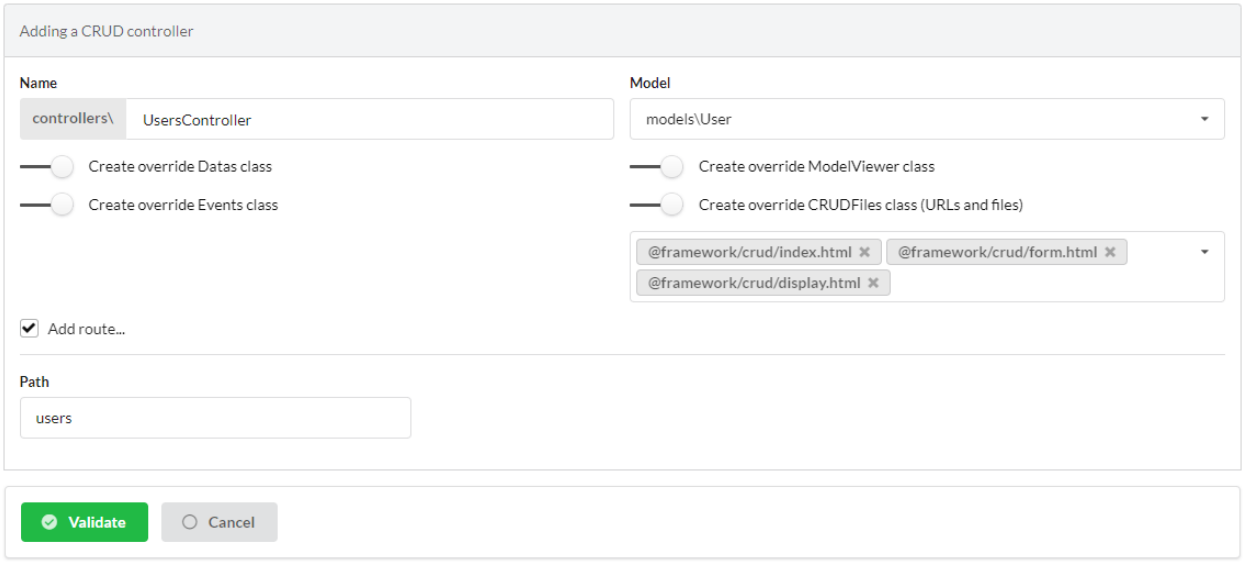
The delete button on each row allows you to edit an instance

Display of the confirmation message before deletion:



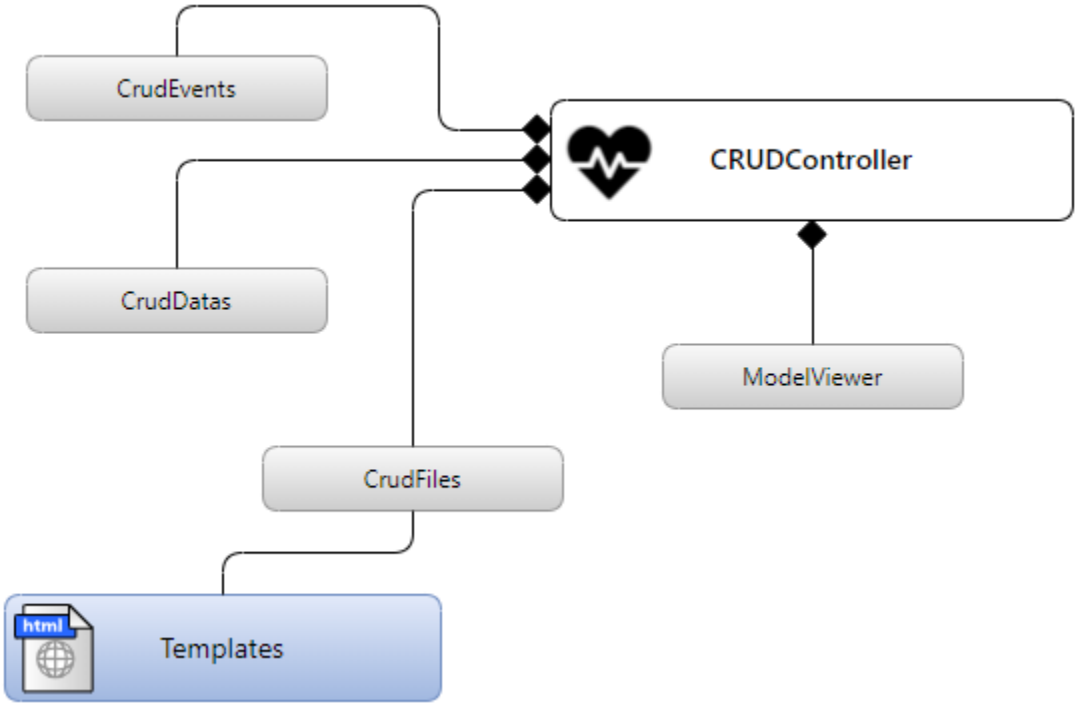
10.3 Customization

Create again a CrudController from the admin interface:



It is now possible to customize the module using overriding.

10.3.1 Overview



10.3.2 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

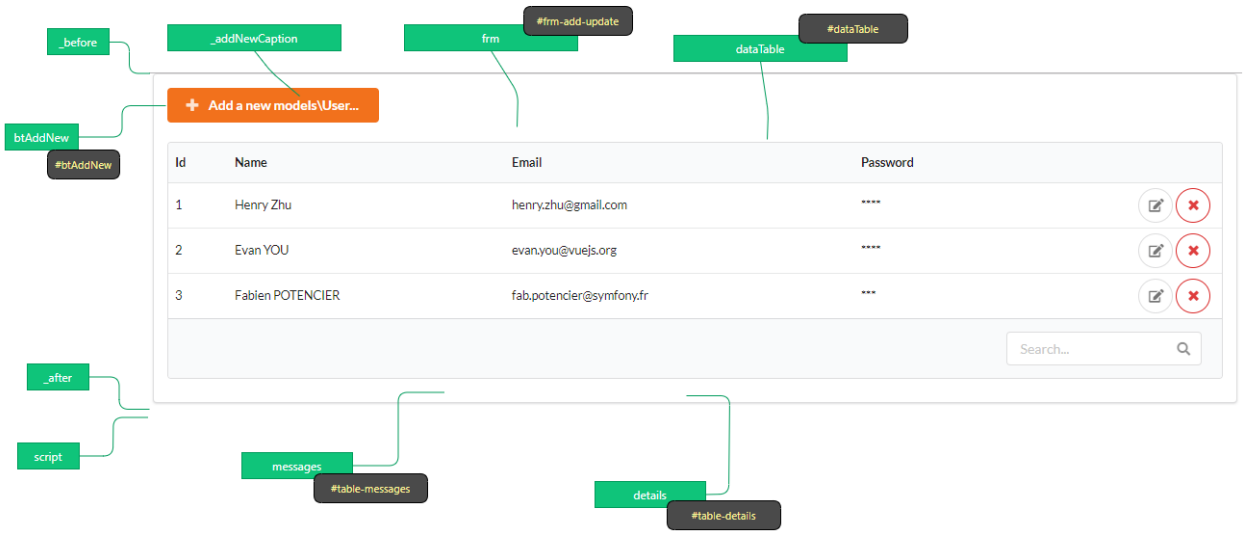
Method	Signification	Default return
index route		
onConfDeleteMessage(CRUDMessage $message,$instance)	Returns the confirmation message displayed before deleting an instance	CRUDMessage
onSuccessDeleteMessage(CRUDMessage $message,$instance)	RReturns the message displayed after a deletion	CRUDMessage
onErrorDeleteMessage(CRUDMessage $message,$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	""

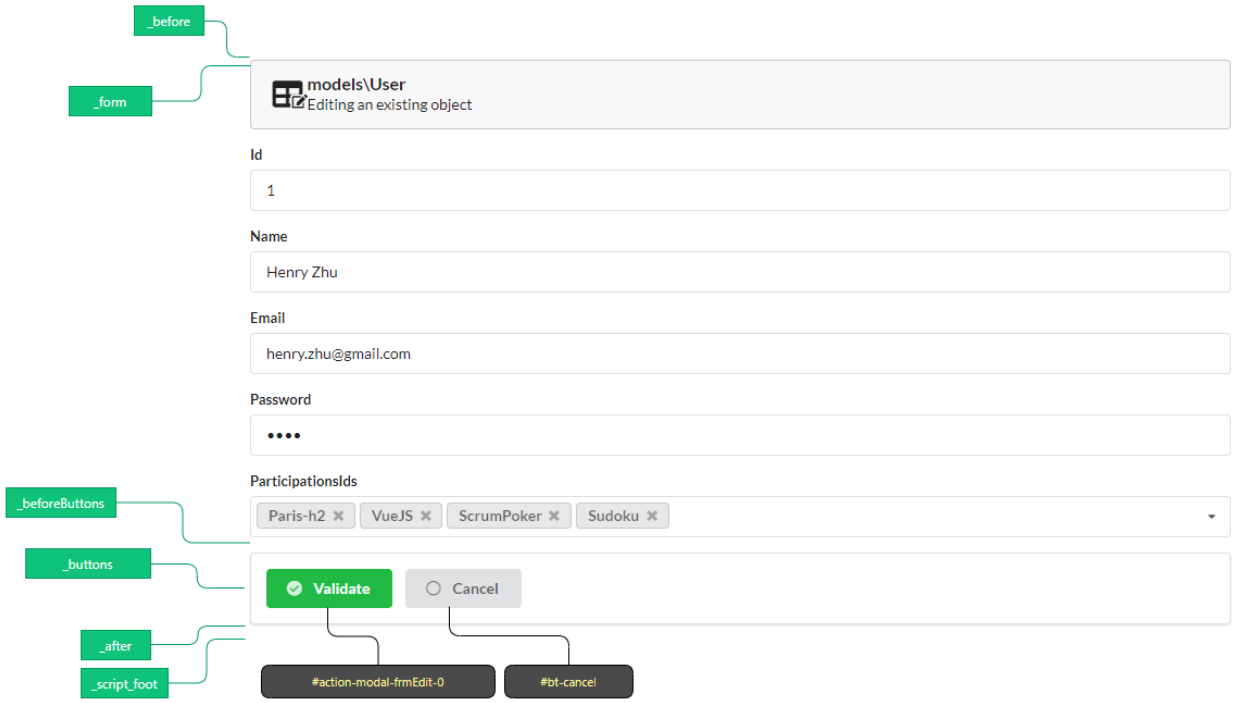
10.3.3 Twig Templates structure

index.html



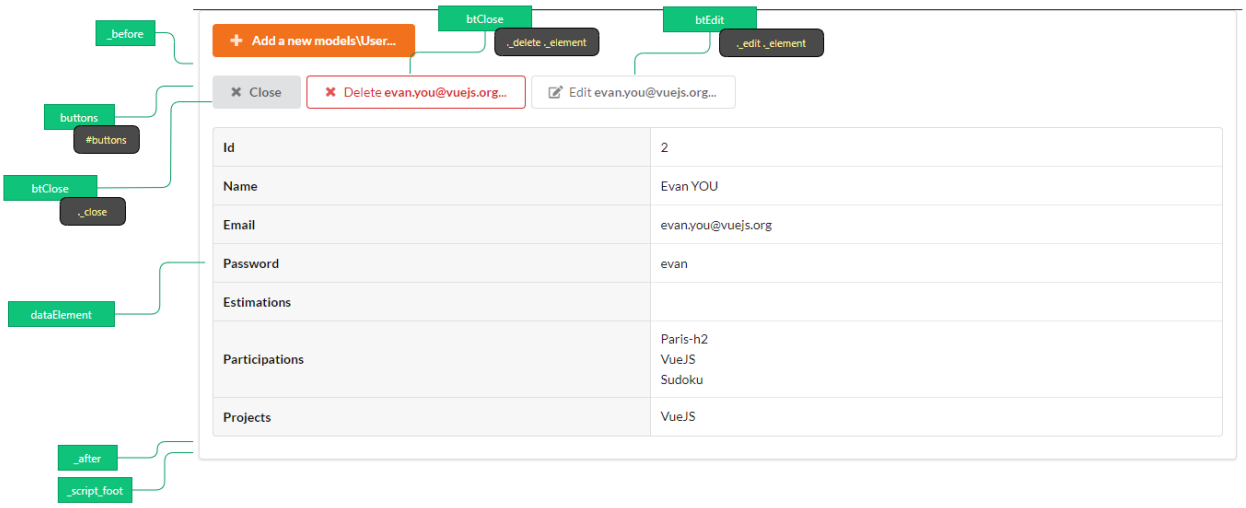
form.html

Displayed in **frm** block



display.html

Displayed in **frm** block



CHAPTER 11

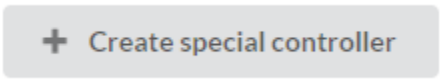
Auth Controllers

The Auth controllers allow you to perform basic authentification with:

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

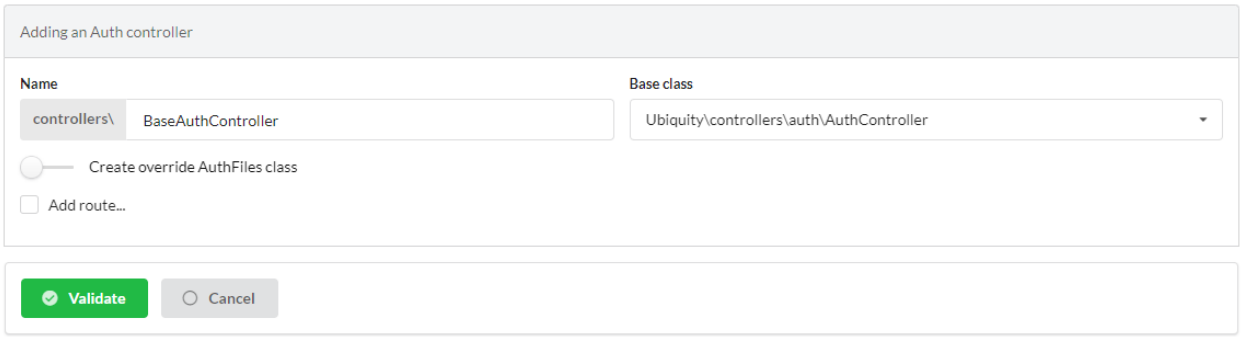
11.1 Creation

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:

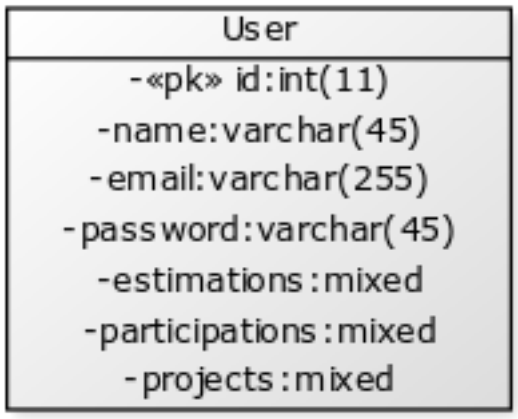
Listing 1: app/controllers/BaseAuthController.php

```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';
    }
}
```

11.2 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**:



11.2.1 BaseAuthController modification

Listing 2: app/controllers/BaseAuthController.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{
                    Startup::forward "admin";
            }
    }

    protected function _connect() {
            if(URequest::isPost()){
                    $email=URequest::post($this->_getLoginInputName());
                    $password=URequest::post($this->_getPasswordInputName());
                    return DAO::uGetOne User::class, "email=? and password= ?" false,
↪ [$email,sha1($password)];
            }
            return;
    }

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

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

(continues on next page)

(continued from previous page)

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

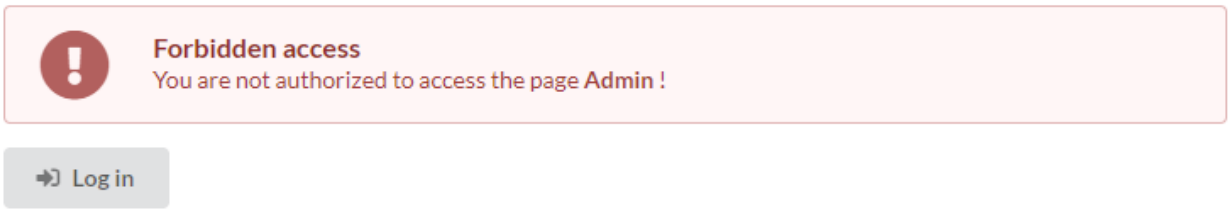
11.2.2 Admin controller modification

Modify the Admin Controller to use BaseAuthController:

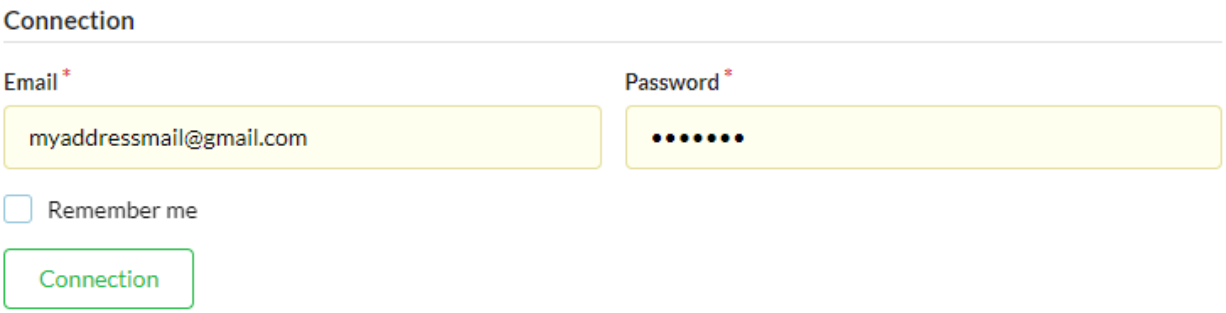
Listing 3: app/controllers/Admin.php

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

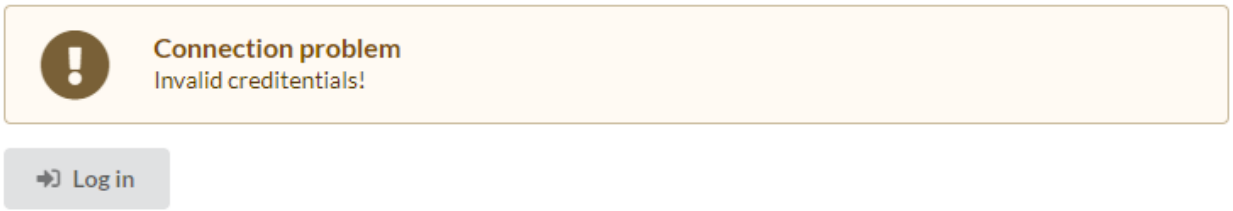
Test the administration interface at **/admin**:



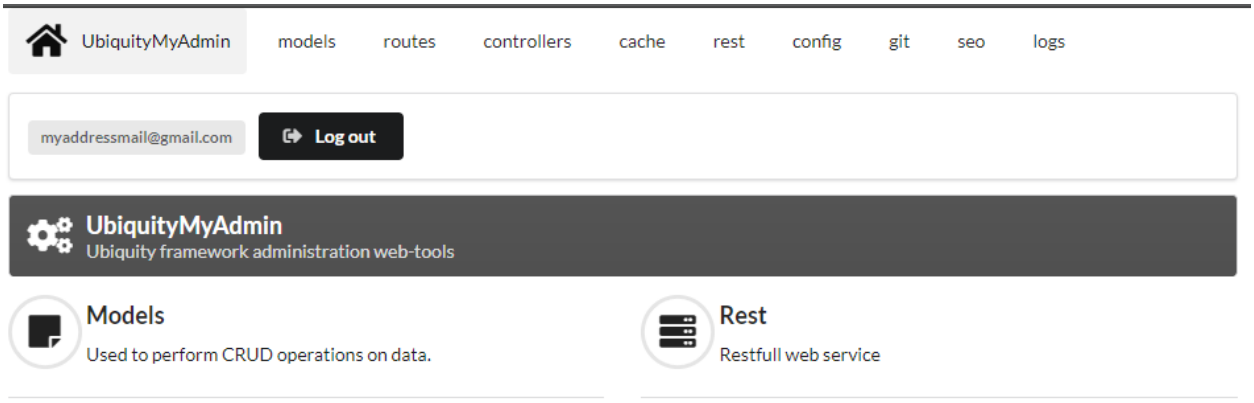
After clicking on **login**:



If the authentication data entered is invalid:



If the authentication data entered is valid:



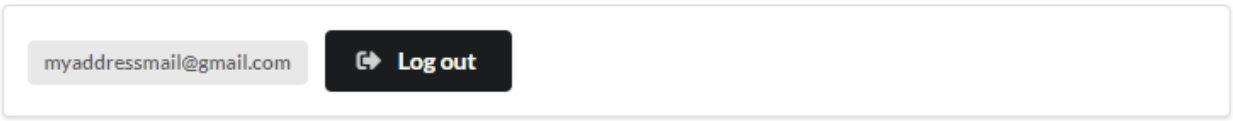
11.2.3 Attaching the zone info-user

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:

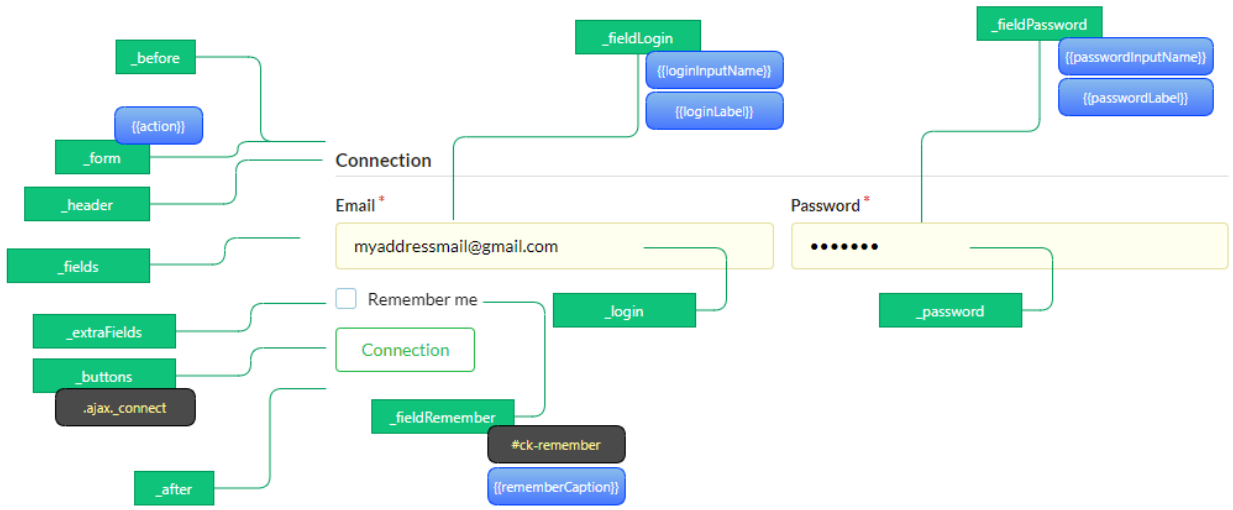
```
{{ _userInfo | raw }}
```

11.3 Description of the features

11.3.1 Customizing templates

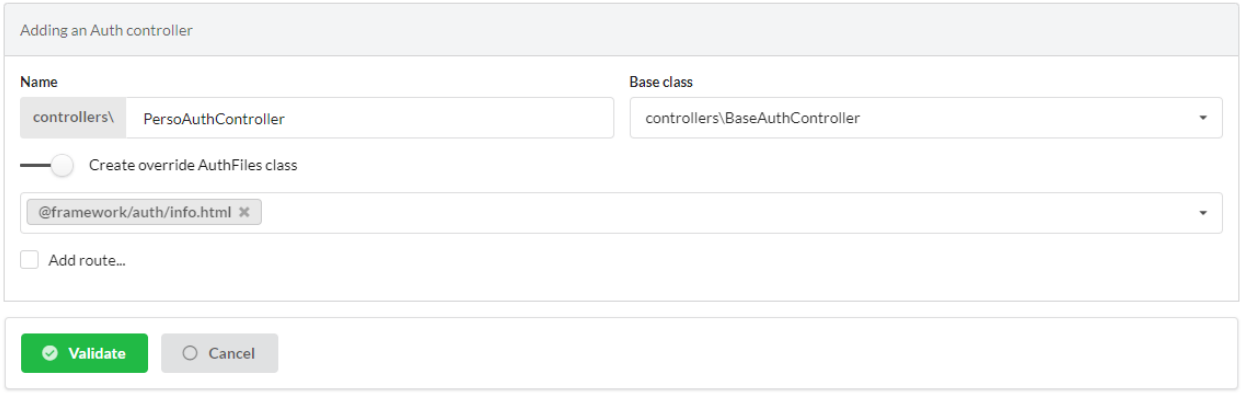
index.html template

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)

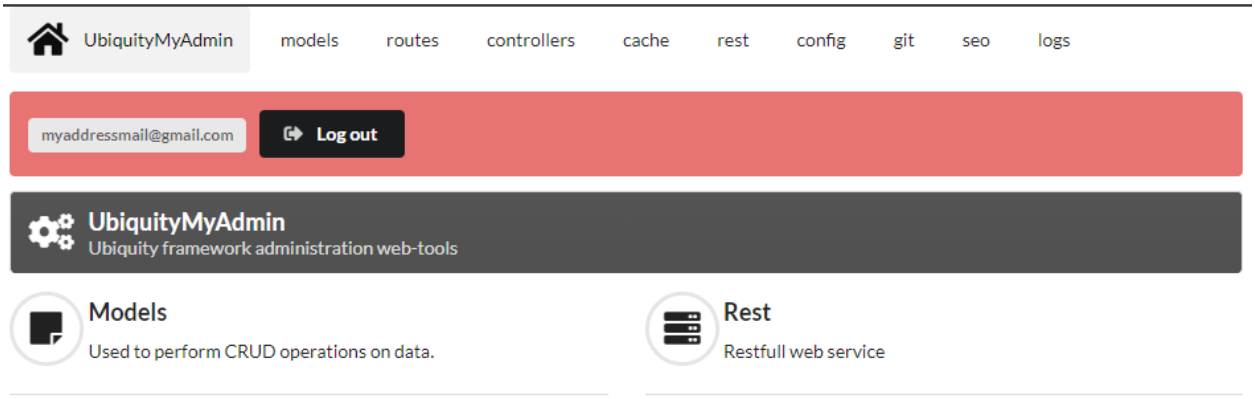
(continued from previous page)

```
{% 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();
    }
}
```



11.3.2 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");

 }
...
}
```

11.3.3 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;
 }
...
}
```

11.3.4 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 12

Models generation

12.1 From existing database

- with console
- with web-tools

CHAPTER 13

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.

13.1 Models definition

13.1.1 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)

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

13.1.2 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

```
namespace models;

/**
* @table("user")
**/
class User{
    /**
     * @id
    **/
    private $id;

    private $firstname;

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

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

```
namespace models;

/**
* @table("user")
**/
class User{
    /**
     * @id
    **/
    private $id;

```

(continues on next page)

(continued from previous page)

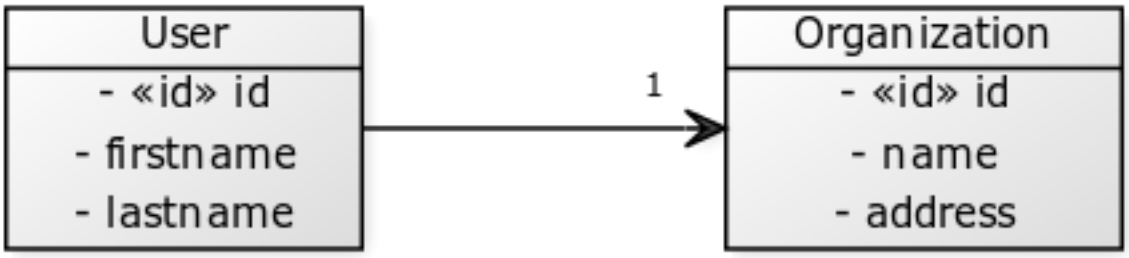
```
    /**
    * column("user_name")
    **/
    private $firstname;

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

13.1.3 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

```
namespace models;

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

    private $firstname;

            /**
             * @manyToOne
             * @joinColumn("className"=>"models\\Organization","name"=>
↪"idOrganization","nullable"=>false)
            **/
            private $organization;
```

(continues on next page)

(continued from previous page)

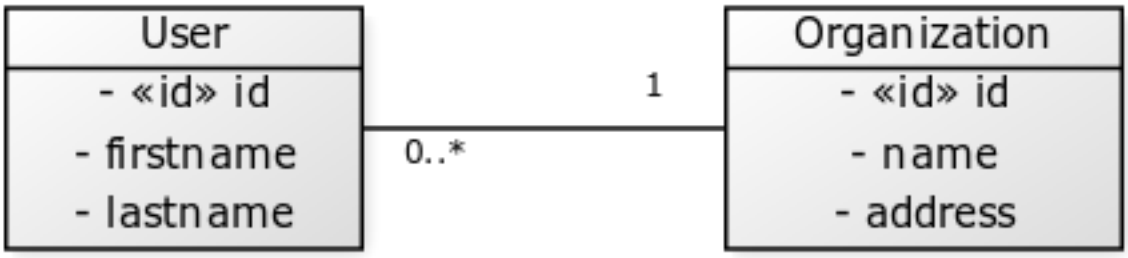
```
16
17              public function getOrganization(){
18                      return $this->organization;
19              }
20
21               public function setOrganization($organization){
22                      $this->organization=$organization;
23              }
24      }
```

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

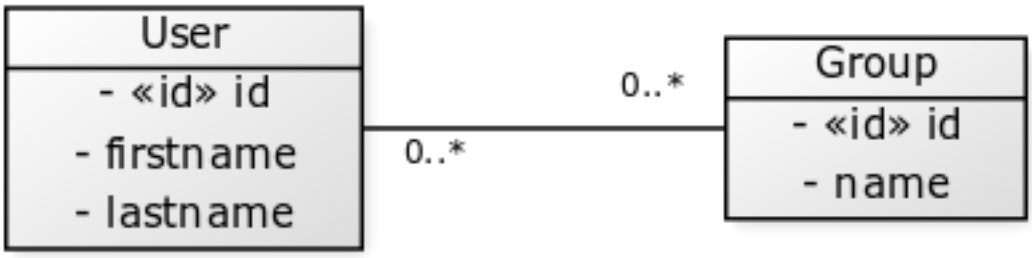
```
1       namespace models;
2
3       class Organization{
4               /**
5                * @id
6               **/
7               private $id;
8
9               private $name;
10
11              /**
12               * @oneToMany("mappedBy"=>"organization","className"=>"models\\User")
13    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


}
```

13.2 ORM Annotations

13.2.1 Annotations for classes

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

13.2.2 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.		

13.2.3 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 14

DAO

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

14.1 Loading data

14.1.1 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>';
```

14.1.2 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 15

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.

15.1 Retrieving data

15.1.1 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("page",1);
```

15.1.2 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("page",1);
```

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

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

15.2 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

```
namespace controllers;

use Ubiquity\orm\DAO;
use Uniquity\utils\http\URequest;

class Users extends BaseController{
    ...
    public function update(){
            $user=DAO::getOne("models\User",URequest::post("id"));
            URequest::setPostValuesToObject($user);
            DAO::update($user);
    }
}
```

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

15.3 Testing the request

15.3.1 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

```
namespace controllers;

use Ubiquity\orm\DAO;
use Uniquity\utils\http\URequest;

class Users extends BaseController{
    ...
    public function update(){
            if URequest::isPost(){
                    $user=DAO::getOne("models\User",URequest::post("id"));
                    URequest::setPostValuesToObject($user);
                    DAO::update($user);
            }
    }
}
```

15.3.2 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");
            }
    }
    ...
```

15.3.3 isCrossSite

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

CHAPTER 16

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 provides additional functionality to more easily manipulate response headers.

CHAPTER 17

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.

17.1 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.

17.2 Creating or editing a session variable

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

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

17.3 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();
```

17.4 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
}
```

17.5 Deleting variables

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

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

17.6 Explicit closing of the session

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

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

CHAPTER 18

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.

CHAPTER 19

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.

19.1 Loading

Views are loaded from controllers:

Listing 1: app/controllers/Users.php

```
namespace controllers;

class Users extends BaseController{
    ...
    public function index(){
                    ->loadView "index.html");
        }
    }
}
```

19.2 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 2: app/controllers/Users.php

```
namespace controllers;

class Users extends BaseController{
    ...
```

(continues on next page)

(continued from previous page)

```
5       public function display($message,$type){
6                               ->loadView "users/display.html"  "message"=> $message "type
    ↪"=> $type);
7               }
8       }
9   }
```

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

Listing 3: app/controllers/Users.php

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

19.3 Displaying in view

The view can then display the variables:

Listing 4: 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'] }}
```

CHAPTER 20

Normalizers

CHAPTER 21

Validators

CHAPTER 22

Translation module

//TODO

See also:

TranslatorManager

CHAPTER 23

Rest

//TODO

CHAPTER 24

External libraries

CHAPTER 25

Ubiquity Caching

CHAPTER 26

Ubiquity dependencies

CHAPTER 27

Indices and tables

- genindex
- modindex
- search