# micro-framework Documentation

*Release 2.0.2*

**phpmv**

**Feb 04, 2019**

# Installation configuration

CHAPTER 1

# Ubiquity-devtools installation

## 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 1.0.x-dev
```

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 micro installation in the directory you specify. For instance, `Ubiquity new blog` would create a directory named **blog** containing an Ubiquity project:

```
Ubiquity new blog
```

You can see more options about installation by reading the *Project creation* section.

CHAPTER 2

# Project creation

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

```
Ubiquity new projectName
```

## 2.1 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 | |

## 2.2 Arguments usage

### 2.2.1 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
```

### 2.2.2 long names

Example of creation of the blog project, connected to the bogDb database, with generation of all models and integration of phpMv-toolkit

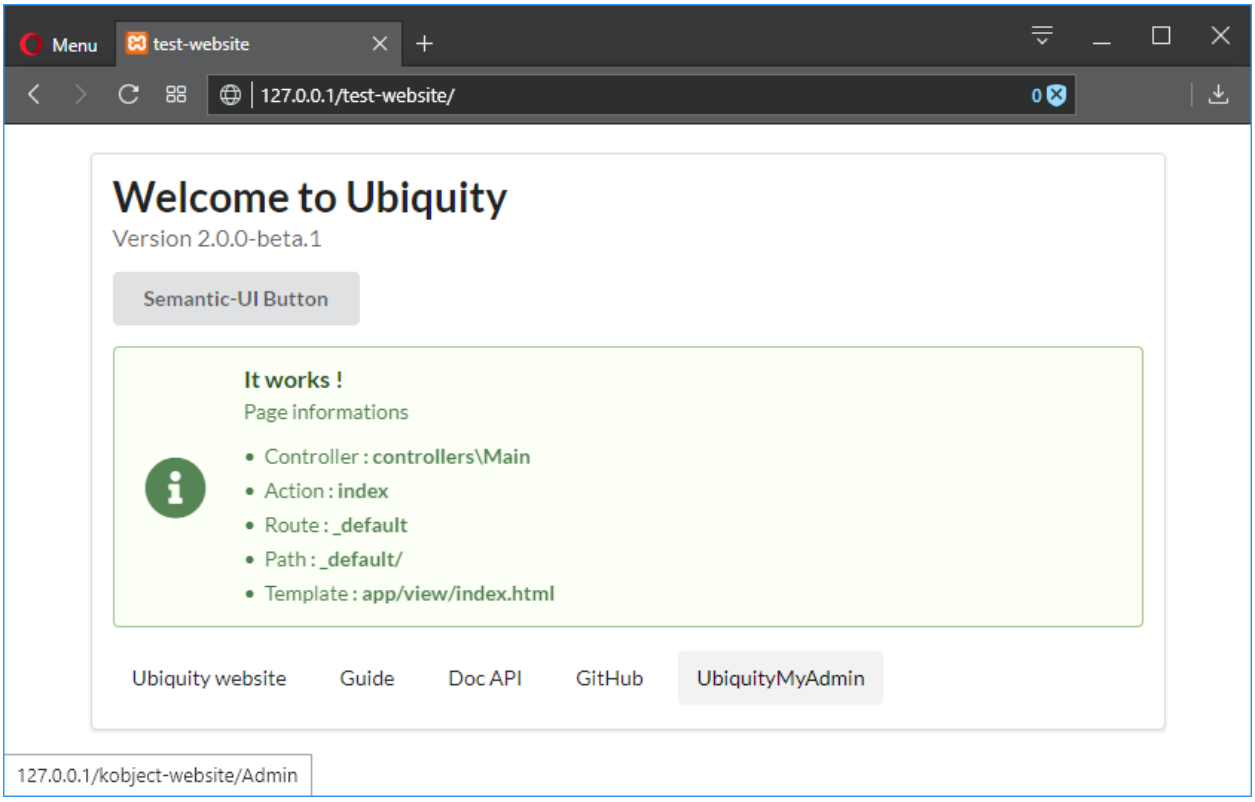
```
Ubiquity new blog --dbName=blogDb --all-models=true --phpmv=semantic
```

**Note:** Markdown doesn't support a lot of the features of Sphinx, like inline markup and directives. However, it works for basic prose content. reStructuredText is the preferred format for technical documentation, please read **'this blog post'_** for motivation.

CHAPTER 3

# Project configuration

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



## 3.1 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",
);
```

## 3.2 Services configuration

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

Listing 2: app/conf/services.php

```
use Ubiquity\cache\CacheManager;
use Ubiquity\controllers\Router;
use Ubiquity\orm\DAO;

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

$db=$config["database"];
if($db["dbName"]!==""){
     DAO::connect($db["dbName"],@$db["serverName"],@$db["port"],@$db["user"],@$db[
↪"password"]);
}
CacheManager::startProd($config);
Router::start();
Router::addRoute("_default", "controllers\Main");
```

## 3.3 Pretty URLs

### 3.3.1 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>
```

### 3.3.2 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 4

# Devtools usage

## 4.1 Project creation

## 4.2 Controller creation

## 4.3 Model creation

## 4.4 All models creation

## 4.5 Cache initialization

CHAPTER 5

# 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
```

## 5.1 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

```
class Products extends ControllerBase{
    public function index(){
        //Default action
    }
}
```

## 5.2 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
```

## 5.3 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
```

## 5.4 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)
```

## 5.5 Routing customization

The *Router* and annotations of controller classes allow you to customize URLs.

CHAPTER 6

# Router

Routing can be used in addition to the default mechanism that associates `controller/action/{parameters}` with an url. Routing works by using the **@route** annotation on controller methods.

## 6.1 Routes definition

### 6.1.1 Creation

Listing 1: 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`.

### 6.1.2 Route parameters

A route can have parameters:

Listing 2: 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'
             // ...
     }
}
```

### 6.1.3 Route optional parameters

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

Listing 3: app/controllers/ProductsController.php

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

### 6.1.4 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 4: app/controllers/ProductsController.php

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

(continues on next page)

(continued from previous page)

```
    ...
    /**
    * 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`

### 6.1.5 Route http methods

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

Listing 5: app/controllers/ProductsController.php

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

### 6.1.6 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 6: app/controllers/ProductsController.php

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

(continues on next page)

(continued from previous page)

```
 **/
class ProductsController extends ControllerBase{

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

}
```

### 6.1.7 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>
```

### 6.1.8 Global route

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

Listing 7: app/controllers/ProductsController.php

```
namespace controllers;
 /**
 * @route("/product")
 * Controller ProductsController
 **/
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 8: app/controllers/ProductsController.php

```
namespace controllers;
 /**
 * @route("/product","automated"=>true)
```

(continues on next page)

(continued from previous page)

```
 * 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 9: app/controllers/ProductsBase.php

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

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

    /**
    *@route("sort/{name}")
    **/
    public function sortBy($name){}

}
```

The derived class using inherited attribute:

Listing 10: app/controllers/ProductsController.php

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

    public function display(){}

}
```

**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.

CHAPTER 7

# 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.

## 7.1 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

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

    public function index(){}

}
```

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

## 7.2 Methods

### 7.2.1 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
```

### 7.2.2 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)
```

### 7.2.3 private

Private or protected methods are not accessible from the URL.

## 7.3 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.

## 7.4 views loading

### 7.4.1 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");
     }
}
```

### 7.4.2 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");
```

### 7.4.3 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;
```

### 7.4.4 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.

### 7.4.5 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");
```

## 7.5 initialize and finalize

## 7.6 Access control

## 7.7 Forwarding

## 7.8 Dependency injection

## 7.9 namespaces

## 7.10 Super class

CHAPTER 8

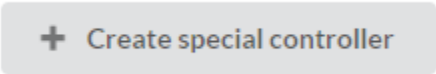
# CRUD Controllers

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

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

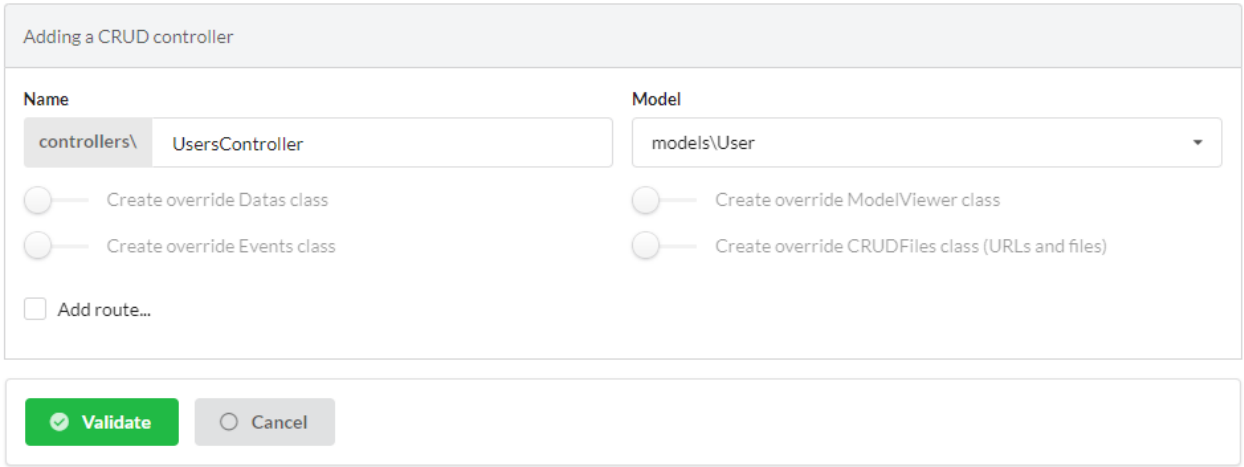
## 8.1 Creation

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



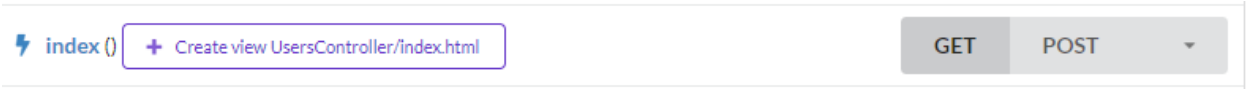

## 8.2 Description of the features

The generated controller:

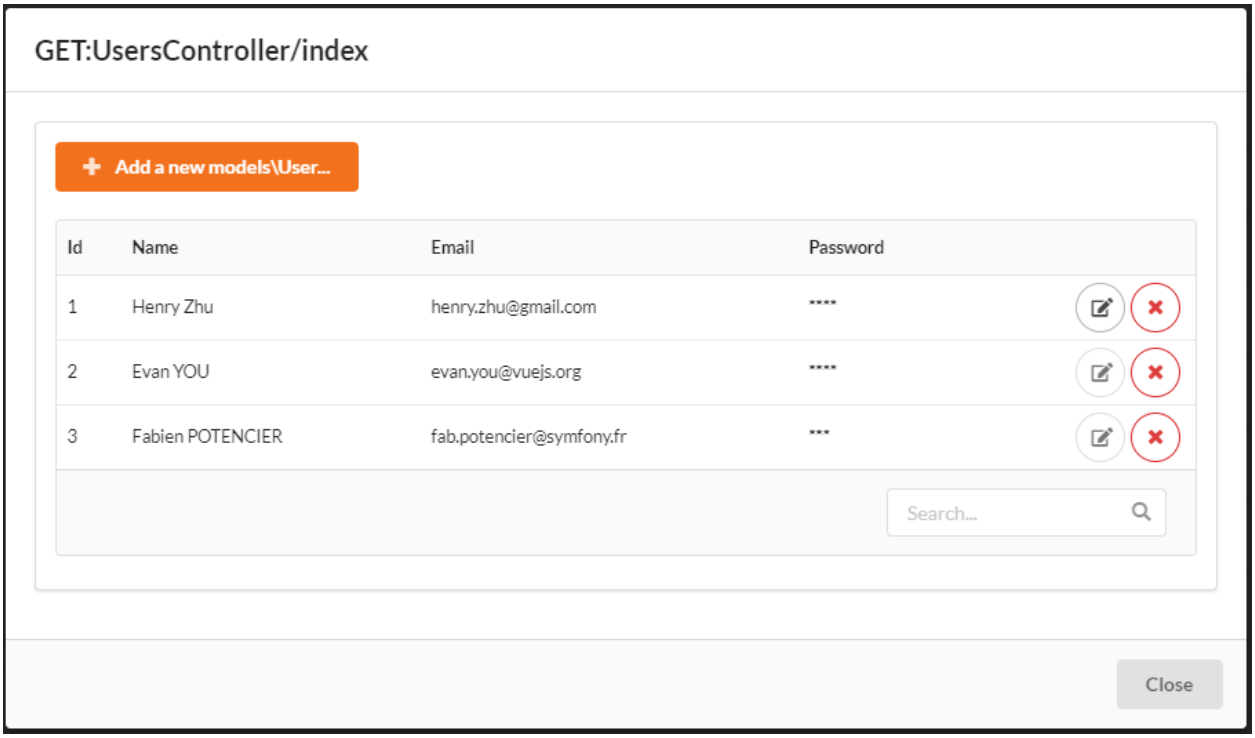
Listing 1: app/controllers/Products.php

```
<?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';
    }

}
```

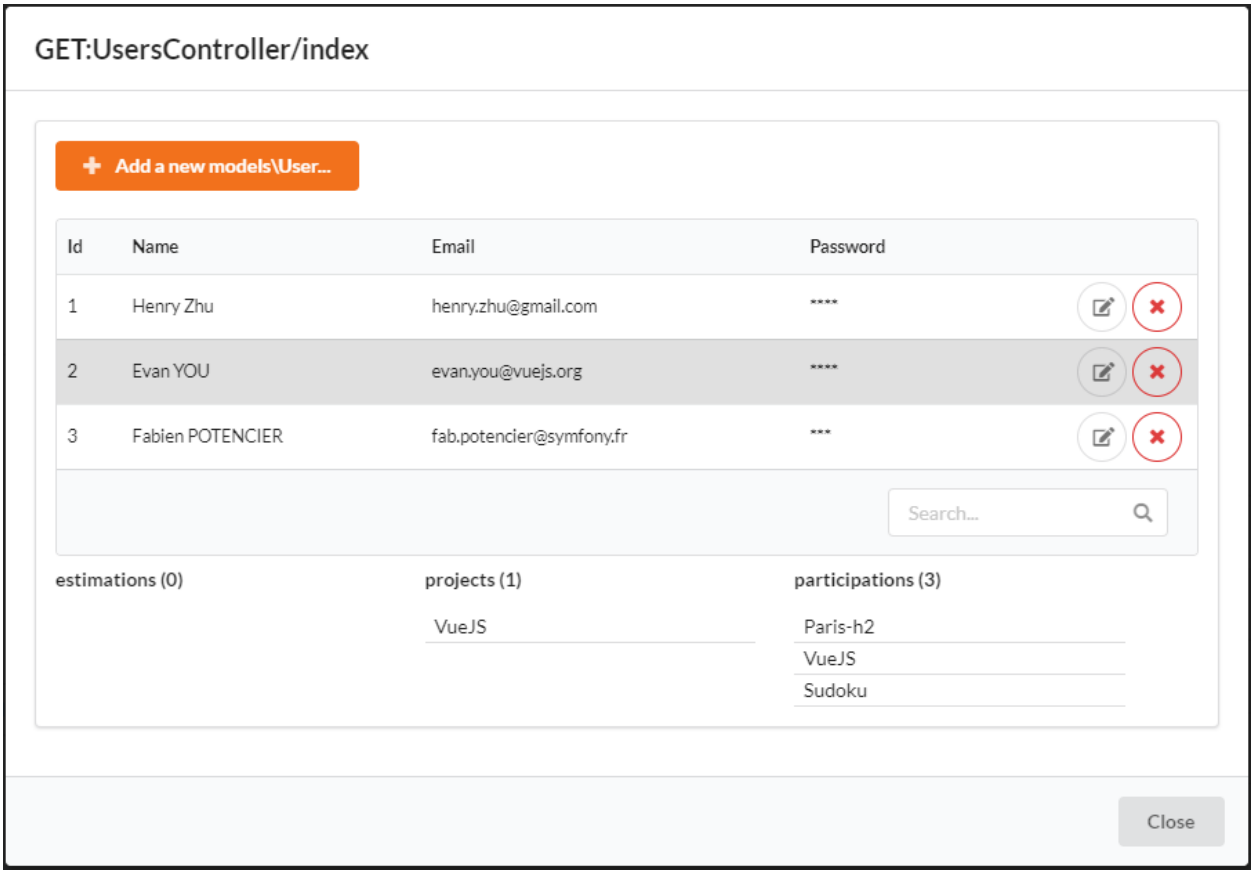
Test the created controller by clicking on the get button in front of the **index** action:



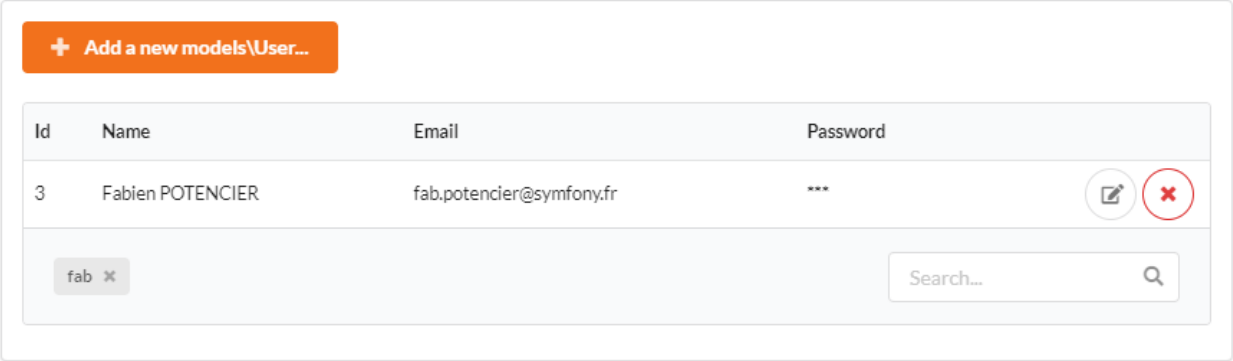
### 8.2.1 Read (index action)




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



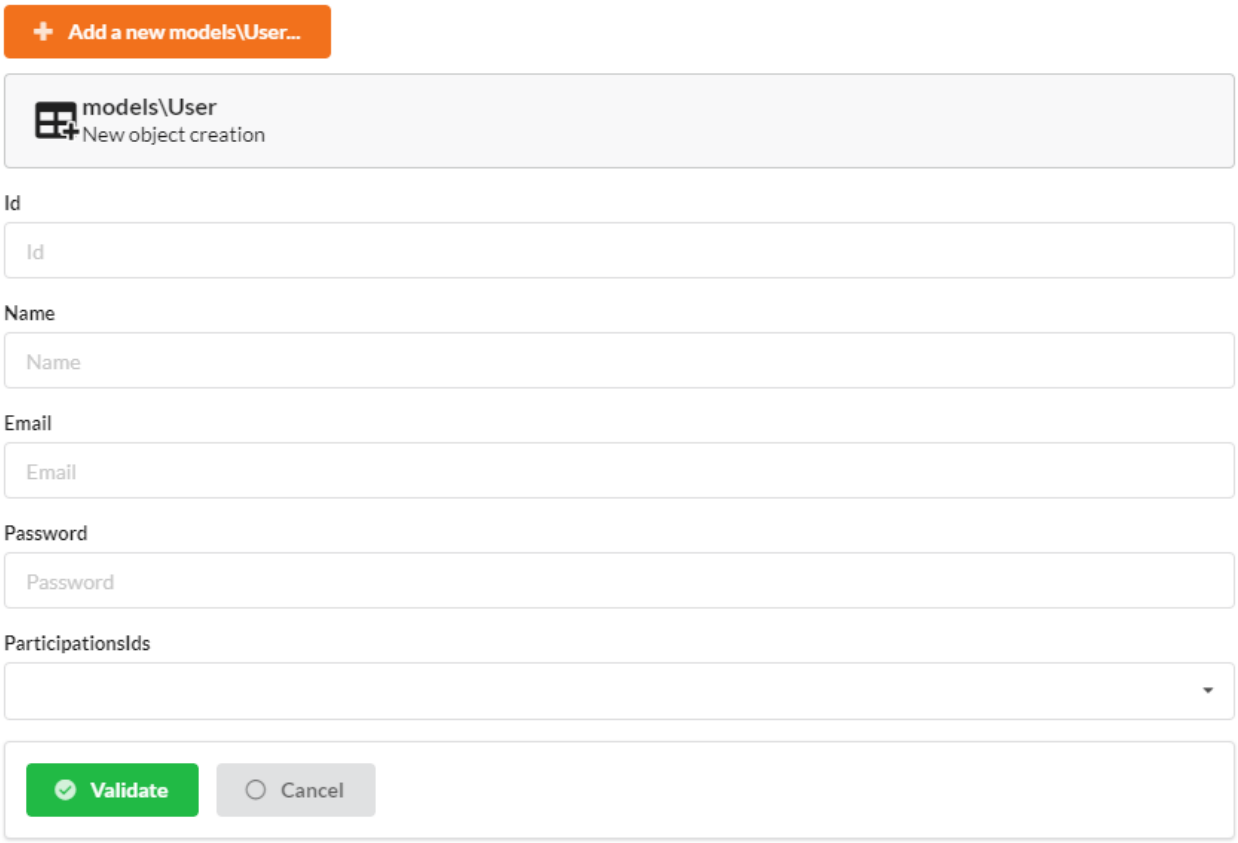

Using the search area:




### 8.2.2 Create (newModel action)

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

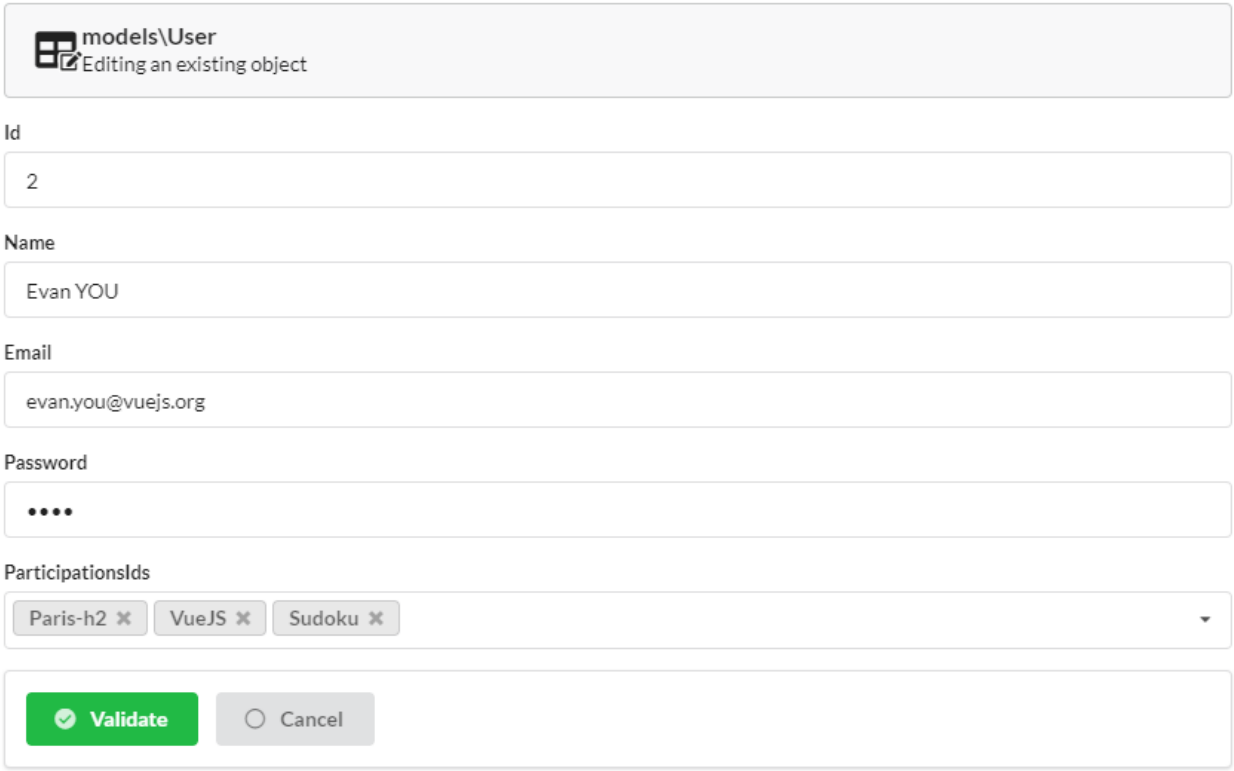

The default form for adding an instance of User:




### 8.2.3 Update (update action)

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

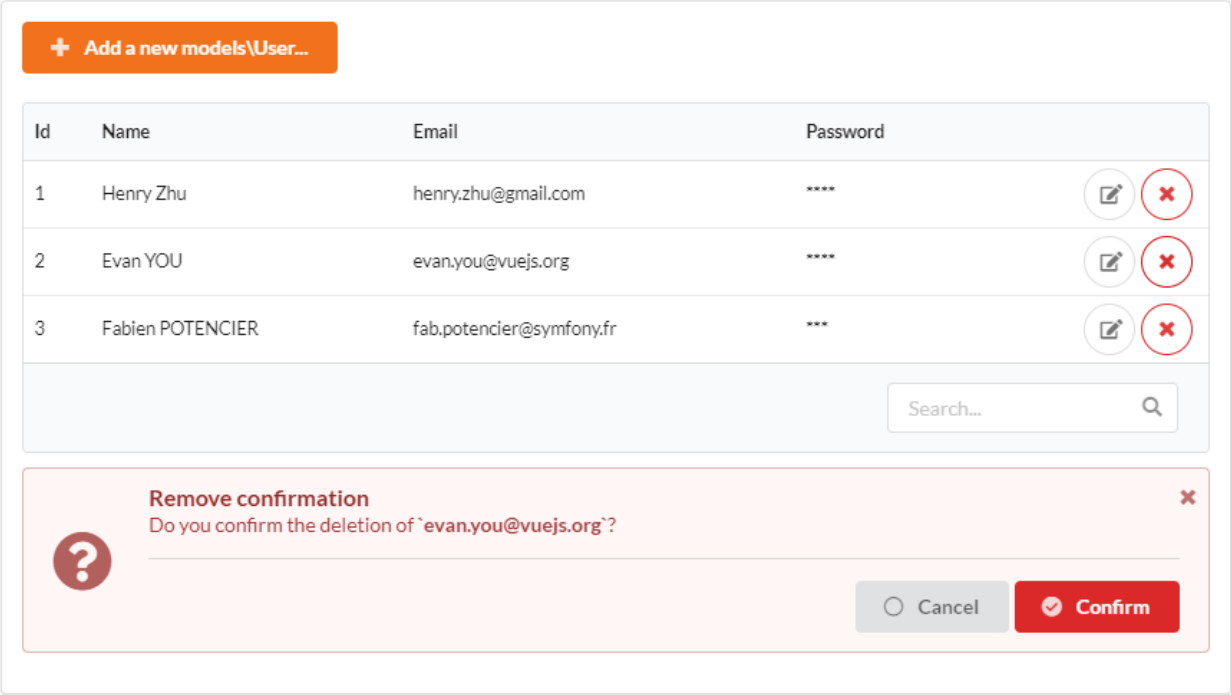
The default form for adding an instance of User:




### 8.2.4 Delete (delete action)

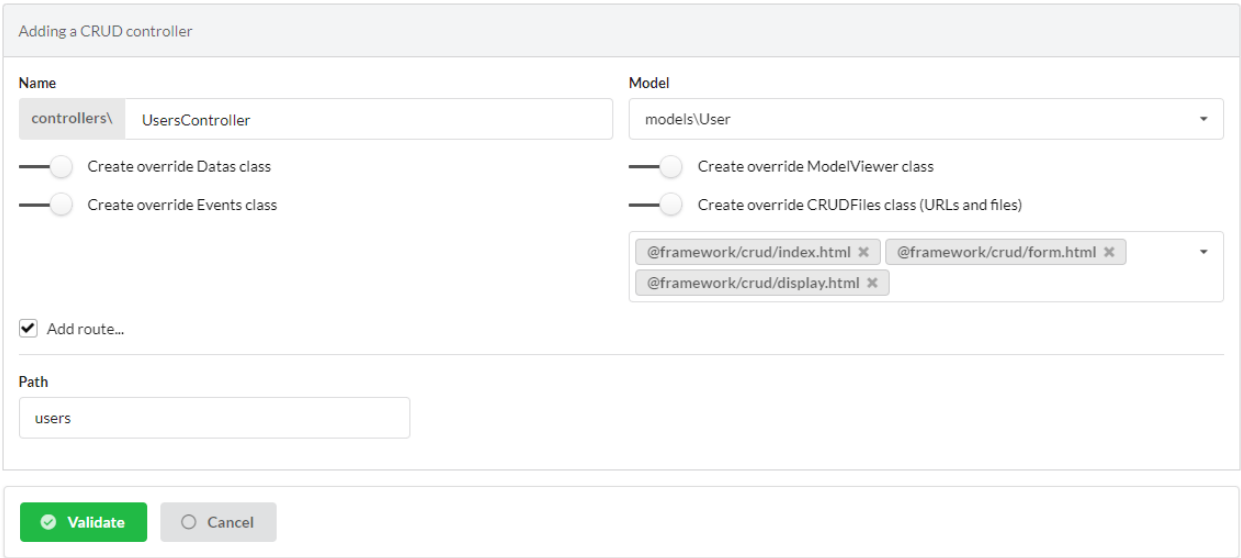
The delete button on each row allows you to edit an instance

Display of the confirmation message before deletion:



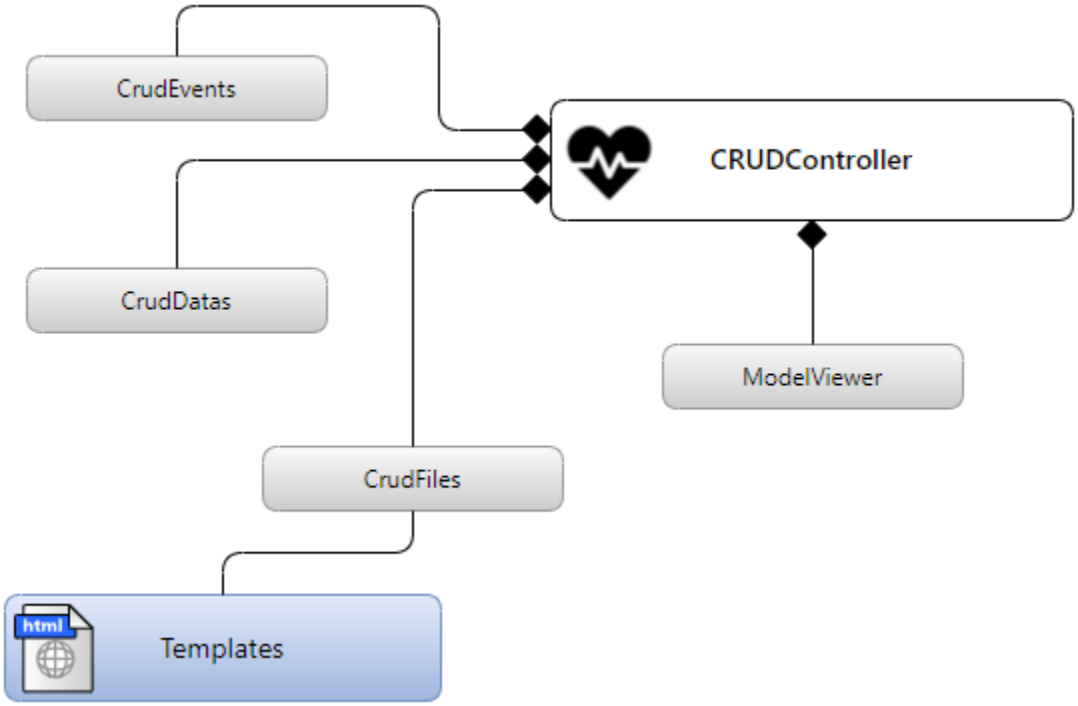

## 8.3 Customization

Create again a CrudController from the admin interface:




It is now possible to customize the module using overriding.

### 8.3.1 Overview




### 8.3.2 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, $class-Name, $object) | Returns a component for displaying a single foreign object (manyToOne relation) | HtmlLabel |
| getFkHeaderListDetails($member, $class-Name, $list) | Returns the header for a list of foreign objects (oneToMany or ManyToMany) | HtmlHeader |
| getFkListDetails($member, $class-Name, $list) | Returns a list component for displaying a collection of foreign objects (many) | HtmlList |
| **edit** and **newModel** routes | | |
| getForm($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, $class-Name, $instance) | Returns the captions for form fields | all member names |
| **display** route | | |
| getModelDataElement($instance,$model,$modal) | Returns a DataElement object for displaying the instance | DataElement |
| getElementCaptions($captions, $class-Name, $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

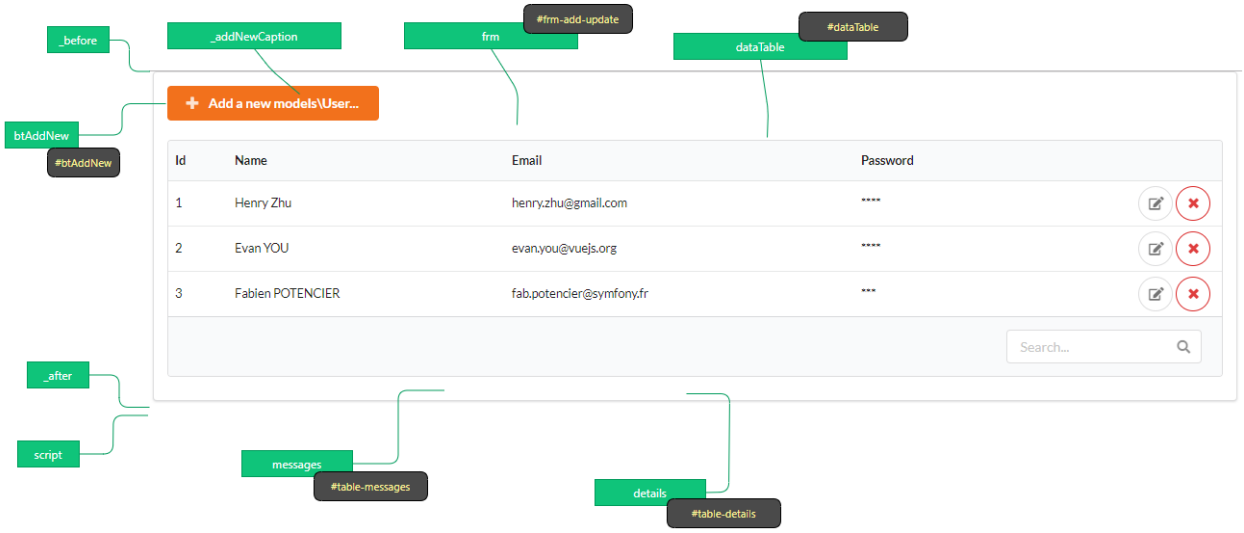
| Method | Signification | Default return |
|---|---|---|
| **index** route | | |
| onConfDeleteMessage(CRUDMessage $message,$instance) | Returns the confirmation message displayed before deleting an instance | CRUDMessage |
| onSuccessDeleteMessage(CRUDMessage $message,$instance) | RReturns the message 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 | “” |

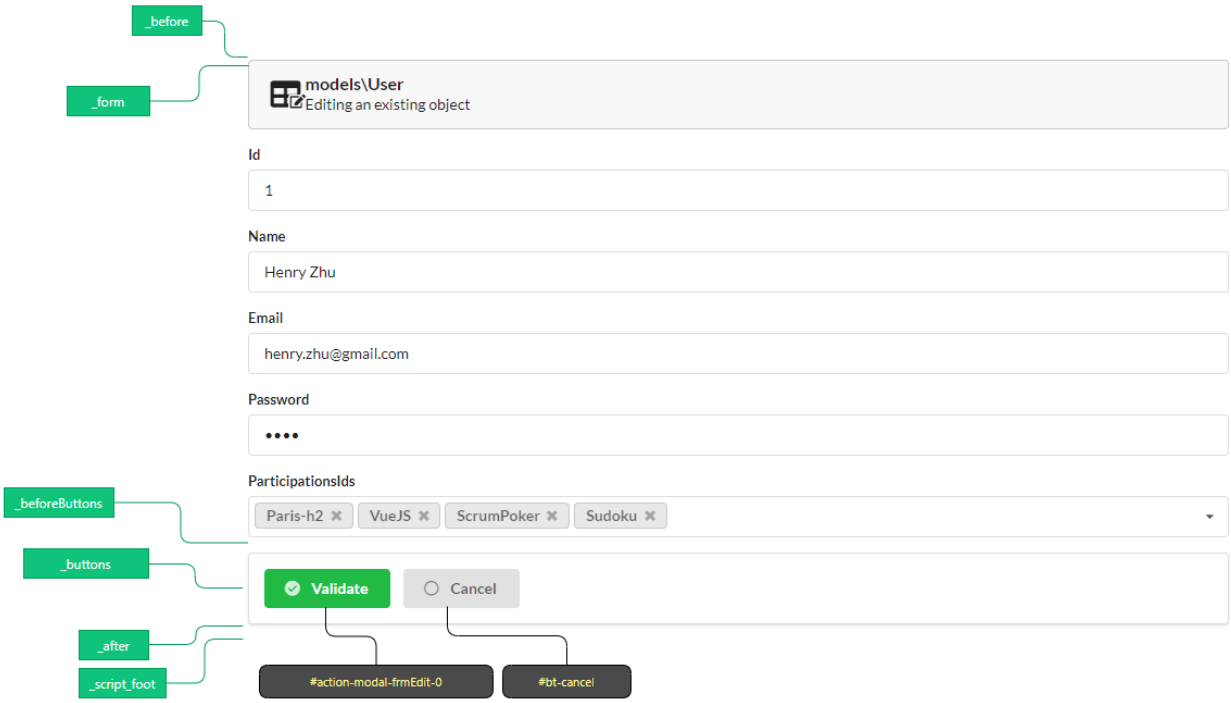
## 8.3.3 Twig Templates structure

### index.html



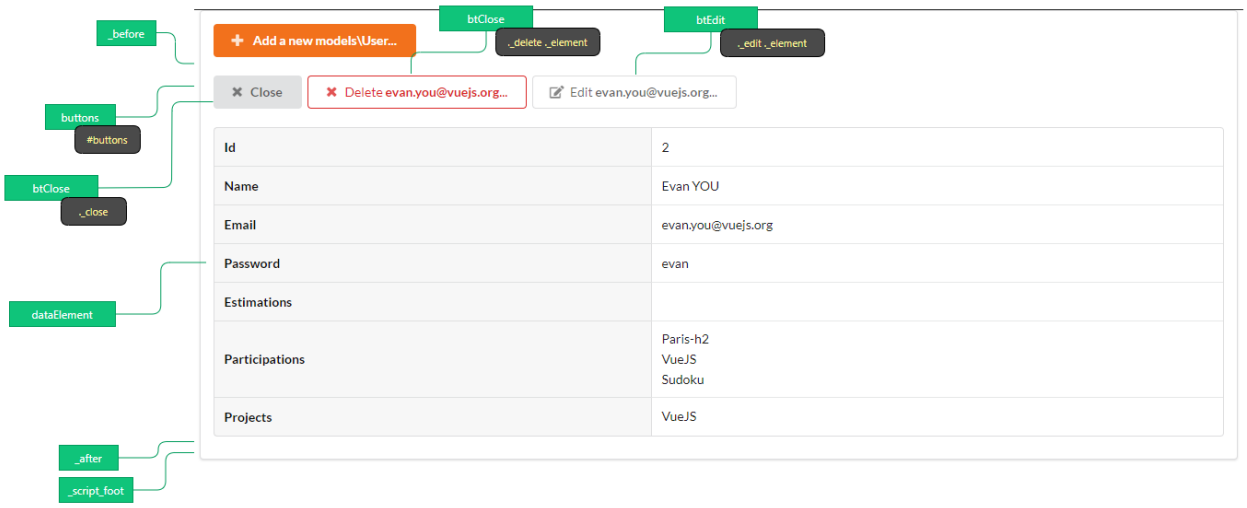

### form.html

Displayed in **frm** block




## display.html

Displayed in **frm** block



CHAPTER 9

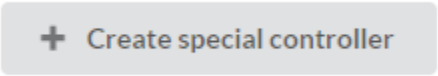
# Auth Controllers

**The Auth controllers allow you to perform basic authentification with:**

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

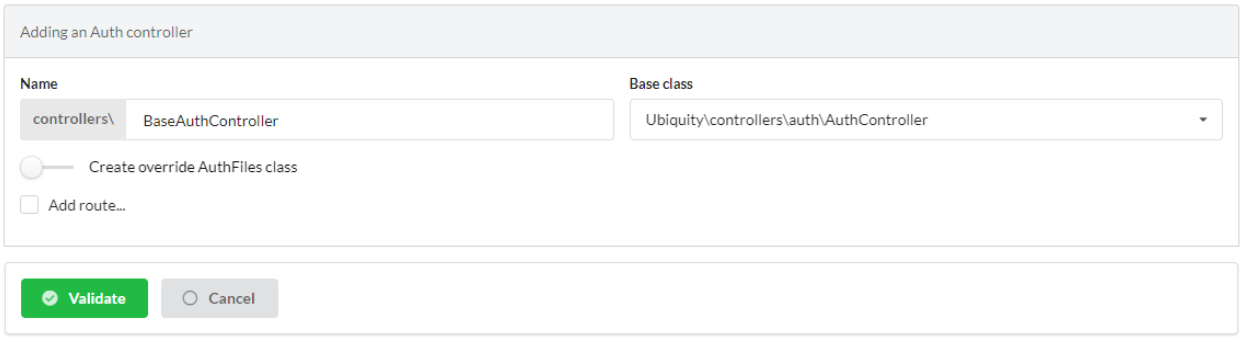
## 9.1 Creation

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:

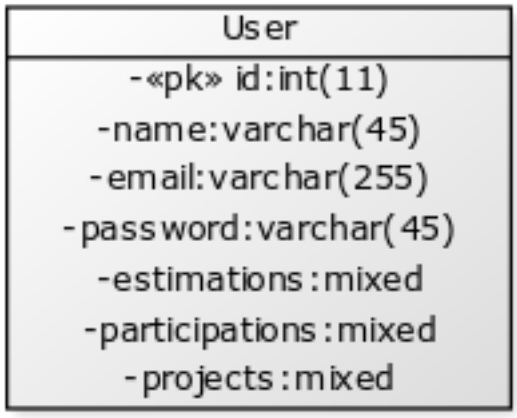
Listing 1: app/controllers/BaseAuthController.php

```
/**
* 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';
    }
}
```

## 9.2 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**:




### 9.2.1 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
↪ array($email,$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)

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

}
```

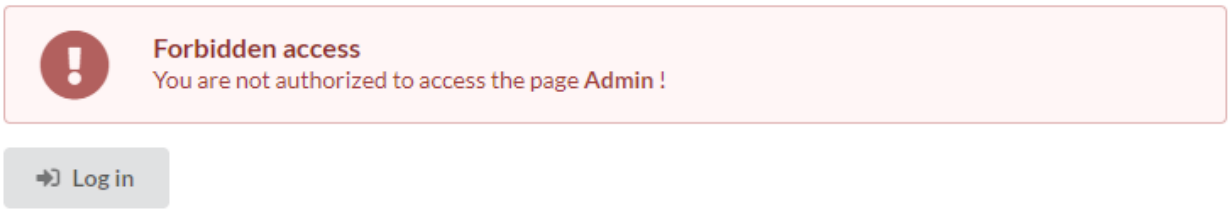
### 9.2.2 Admin controller modification

Modify the Admin Controller to use BaseAuthController:

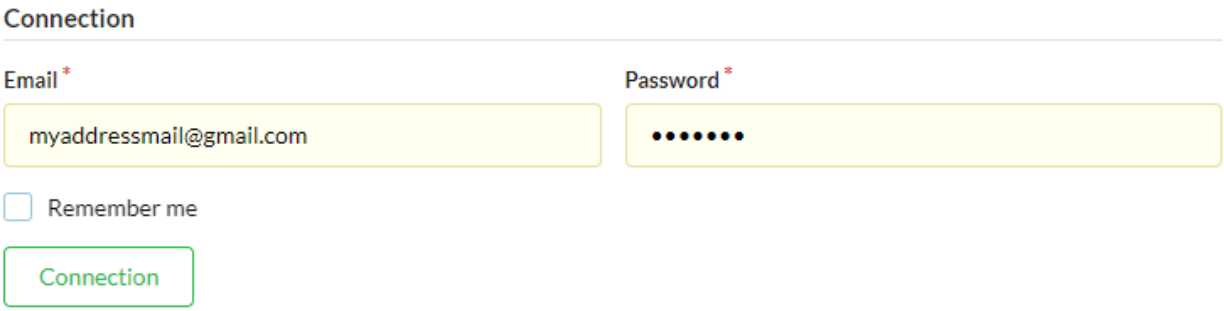
Listing 3: app/controllers/Admin.php

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

}
```

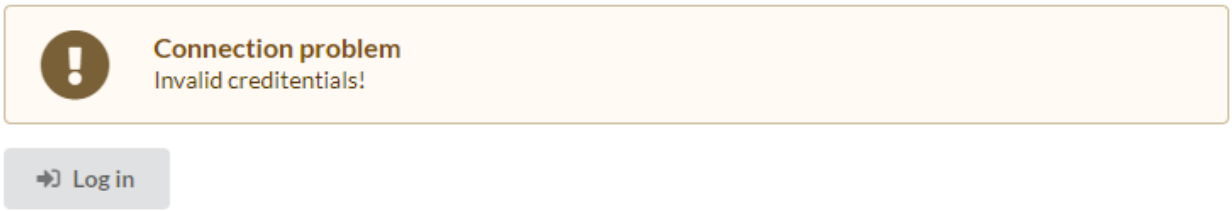
Test the administration interface at **/admin**:



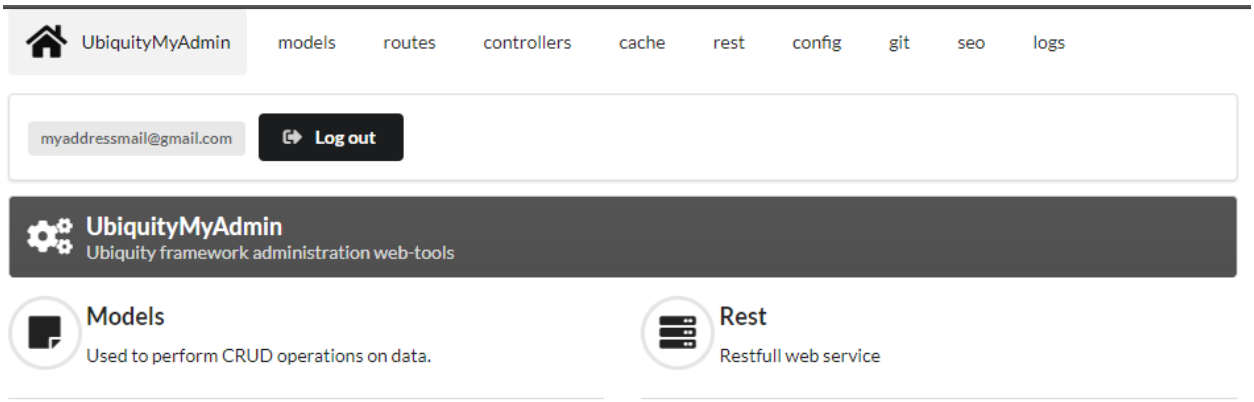

After clicking on **login**:




If the authentication data entered is invalid:



If the authentication data entered is valid:



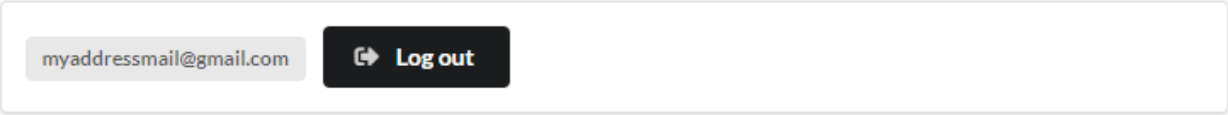

### 9.2.3 Attaching the zone info-user

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:

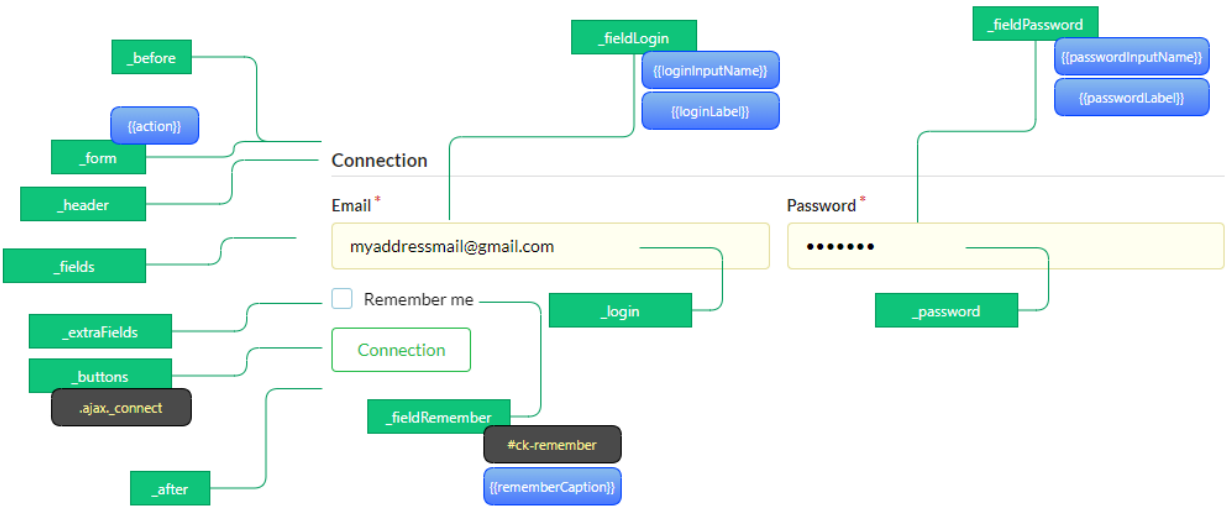
```
{{ _userInfo | raw }}
```

## 9.3 Description of the features

### 9.3.1 Customizing templates

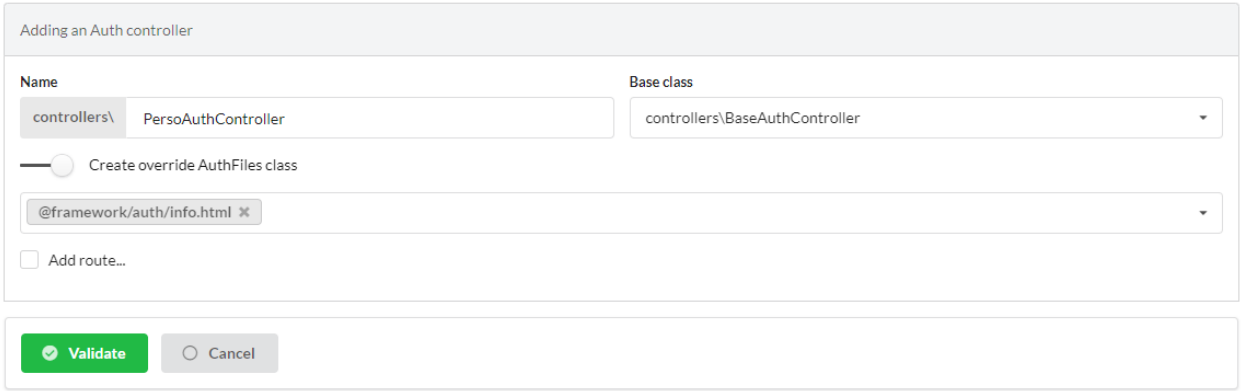
#### index.html template

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)

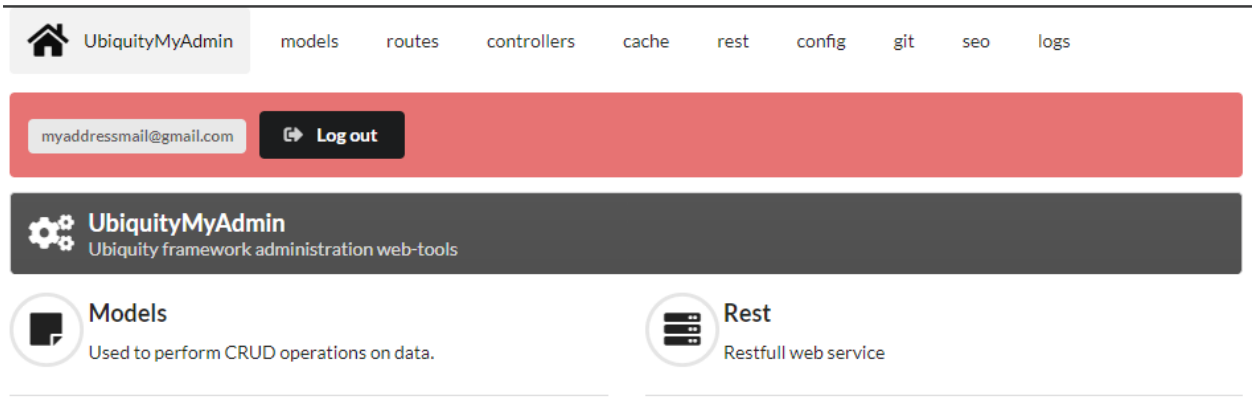
(continued from previous page)

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




### 9.3.2 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");

 }
...
}
```

### 9.3.3 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;
 }
...
}
```

### 9.3.4 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 10

# ORM

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.

## 10.1 Models definition

### 10.1.1 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/Product.php

```
namespace models;
class Product{
    /**
     * @id
    */
    private $id;

    private $name;

    public function getName(){
            return $this->name;
    }
    public function setName($name){
            $this->name=$name;
    }
}
```

//TODO

CHAPTER 11

# DAO

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

## 11.1 Loading data

### 11.1.1 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

```
namespace models;
class ProductDetail{
    /**
     * @id
    */
    private $idProduct;

    /**
     * @id
    */
    private $idCommand;

    ...
}
```

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>';
```

## 11.1.2 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 12

# 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.

## 12.1 Retrieving data

### 12.1.1 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);
```

### 12.1.2 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();
```

## 12.2 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.

## 12.3 Testing the request

### 12.3.1 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);
            }
    }
}
```

### 12.3.2 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");
            }
    }
    ...
```

### 12.3.3 isCrossSite

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

CHAPTER 13

# 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 14

# 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.

## 14.1 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.

## 14.2 Creating or editing a session variable

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

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

## 14.3 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();
```

## 14.4 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
}
```

## 14.5 Deleting variables

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

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

## 14.6 Explicit closing of the session

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

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

CHAPTER 15

# 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 16

# 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.

## 16.1 Loading

Views are loaded from controllers:

Listing 1: app/controllers/Users.php

```
namespace controllers;

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

## 16.2 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)

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

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

Listing 3: app/controllers/Users.php

```
namespace controllers;

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

## 16.3 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 17

# External libraries

CHAPTER 18

# Ubiquity Caching

CHAPTER 19

# Ubiquity dependencies

CHAPTER 20

# Indices and tables

- genindex
- modindex
- search