



C1 - Frameworks MVC

C-COD-130

Rush MVC

It's your own MVC framework

Rush MVC

repository name: PHP_Rush_MVC

repository rights: ramassage-tek



- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

- Introduction
- Phase 0 (Structure of your website)
 - Controllers
 - TWIG
 - ORM (Object-Relational Mapper)
 - Router
 - Apache, XAMPP and .htaccess
 - Summary
- Phase 1 (Users)
- Phase 2 (Login and Registration)
- Phase 3 (Administration of users)
- Phase 4 (Articles)
- Phase 5 (Comments)
- Phase 6 (Administration of articles)
- Phase 7 (Categories and tags)
- Bonuses (Have a lot of fun!)

INTRODUCTION

The project must be done in a team of two.

Expect this rush to be an introduction to how frameworks, like CakePHP or Symfony, works. Now you'll see more advanced notions like a router.

Here is the situation: a client wants you to create a blog with these specifications.

- Your site must respect the **MVC** design pattern (**Model - View - Controller**).



- It has to respect **OOP (Object Oriented Programming)** paradigm.
- The models must contain only functions to manage database requests exclusively with the help of **PDO**.
- You have to manage your views with **TWIG**.
- Your **external libraries** (such as TWIG) must be located in the “/vendor” directory and managed with **Composer**.
- The only page called by the user is “**index.php**” stored in the WebRoot directory.
- “**/Webroot**” is the only directory that can be accessed by the user. It's then up to the “**index.php**” file to make the necessary includes.
- The “**/App/Helpers/Session.php**” file contains a class Session that implements methods read, write, delete, destroy and load. These methods are static. This class has to be used over PHP sessions (\$_SESSION) which it overlays.

There are a lot of types of MVC architecture in programming. But you will see soon the Symfony 4 framework which is using a certain type of MVC. You will learn to create this type and not another.



For the following we ask you to start from the sample example provided during the bootstrap.

PHASE 0 (STRUCTURE OF YOUR WEBSITE)

The structure of the site follows the MVC pattern. This design pattern is crucial as it's one of the most used.



You have to understand well the MVC principles to be able to create one.

Most of the core files of this MVC project are stored in the “/WebFramework” directory.



Understand why these classes are important and how they work.



Some of these classes are **singletons**. You can recognize these with their “public static function getInstance()” method. Search what singletons are and why we use these.



+ +CONTROLLERS

The controllers are one of the most important parts of any MVC models. They act as an interface between Model and View components to process all the business logic and incoming requests.

They must inherit from the parent class defined in “/WebFramework/Controller”. Let’s review some of its methods:

- public function render(string \$view, array \$context = [])
 - Render the TWIG view and send an optional context to it.
- public function redirect(string \$url, string \$status)
 - Redirect current request to another route. This triggers another handler.
 - \$url is the route to redirect to and \$status is the HTTP status code to send.

+ +TWIG

TWIG is used to manage the views.

So, basically, what is it? It's a frontend template engine, to help us design our HTML pages with the PHP content (send as context).



You can find the views in the `"/App/Views"` directory.

You have to create a 404 page. This view will be displayed by the Router when the user tries to access an invalid page.

+ +ORM (OBJECT-RELATIONAL MAPPER)

Object-Relational Mapping is a technique that lets you query and manipulates data from a database using an Object-Oriented Paradigm.

The ORM is a library that implements the Object-Relational Mapping technique.



You can find the ORM class in the “/WebFramework” directory, with the other core components.

Before you begin to use the ORM, you have to modify its configuration.

The “/config/db.php” file must contain every connection information needed to connect to the database(s). Change the connection map with connections information of your database.

You may have noticed that “persist” and “flush” functions still are unimplemented. It’s your job to do it.

These functions are already documented at the place of their definitions. Take a look at it and make some research to understand what those functions may be used for.

+ +ROUTER

The router retrieves information from the URL, parses it and dispatch the requests to the correct controllers and handlers.

The class is declared in the “/WebFramework/Router.php” file.

In order to use the router, you have to configure it. Add each of your routes with the associated controllers and handlers in the “/config/routes.php” file.

You may ask yourself how the router work. In order to explain it, we'll guide you into the creation of a router.

First, let's create the Router class and its principal attributes.

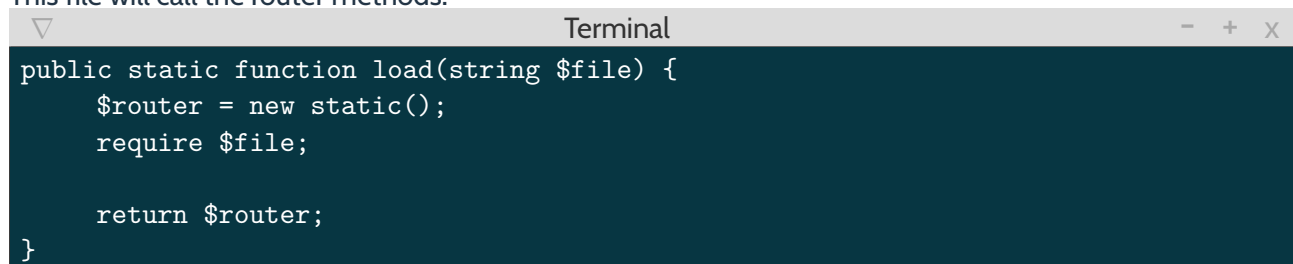
The ‘\$routes’ attributes stores each routes by request type (GET and POST in this case) and associate each of these with an handler.

A terminal window with a dark background and light text. The title bar says "Terminal" with standard window controls. The code defines a class Router with a private \$routes attribute containing two arrays for GET and POST requests.

```
class Router {  
    private $routes = [  
        'GET' => [],  
        'POST' => [],  
    ];  
}
```

Then we have to load the routing file (“/config/routes.php”) in order to initialize the routes. This function simply requires the file specified in the arguments and return a new instance of the router.

This file will call the router methods.

A terminal window with a dark background and light text. The title bar says "Terminal" with standard window controls. The code defines a public static function load that takes a file path, creates a new Router instance, requires the file, and returns the instance.

```
public static function load(string $file) {  
    $router = new static();  
    require $file;  
  
    return $router;  
}
```

Now we must create a function to associate the route declared in the routing file to our handlers and controllers.



```
Terminal
public function use(string $requestType, string $route, Controller $controller,
    string $handler) {
    $this->routes[$requestType][$route] = [
        'controller' => $controller,
        'handler' => $handler
    ];
}
```

Our routes are now defined and associated with the Router class. We have to create a function to dispatch requests to the appropriate controllers and handlers.

```
Terminal
public function dispatch(Request $request) {
    if (array_key_exists($request->method, $this->routes)
        && array_key_exists($request->route,
            $this->routes[$request->method])) {
        $route_handler = $this->routes[$request->method][$request->route];
        $this->handle($request, $route_handler['controller'],
            $route_handler['handler']);
    } else {
        echo '<h1>Page not found</h1>';
    }
}
```

As you can see, the 'dispatch' function above make a call to the 'handle'. This is the last missing function from the controller. This function verifies that the controller and handler are defined. If yes, the functions call the handler and forward the request.

The handler will render the views, interact with the database with the ORM, etc.

```
Terminal
public function handle(Request $request, Controller $controller, string $handler)
{
    if (!method_exists($controller, $handler)) {
        $controller_name = get_class($controller);
        throw new Exception("${$controller_name} does not have {$handler}");
    }

    $controller->$handler($request);
}
```

If you want to see the result of the Router described here, take a look at the `/WebFramework/Router.php` file.



+ +APACHE, XAMPP AND .HTACCESS

.htaccess files provide some security to your website and a **project-wide configuration**. You should search for how they accomplish their roles.

The .htaccess files are already written at the project root and in the “/WebRoot” directory.

You may take a look at those in order to take a better understanding of the URL and routes management.

+ +SUMMARY

Now that you have a better understanding of the project structure and that the project is configured, let's implement some functionalities.



Remember, your structure must be coherent

Your code must be clear and understandable. Don't forget to write comments.

Documenting your code is crucial to keep it maintainable and easily understandable. If you want to, use Swagger to describe your routes.

Check the “/swagger.yaml” file to find sample documentation for this code.

You should also take a look at their documentation and their editor (editor.swagger.io) in order to write great documentation.



PHASE 1 (USERS)

You have to create user entities which will contain at least:

- Username,
- Hashed password,
- Email,
- Group,
- Status of the user (if he/she is banned),
- Status of the account (if the account is activated or not),
- His creation date,
- His last modification date.

There is a hierarchy between users. Indeed, they can be administrators, writers or normal users. Each user group inherits the rights of lesser groups (Example: ADMINISTRATOR can perform all actions of WRITER who can then perform all actions of USER). You decide what rights to each group, but it has to remain consistent.

Bonus: When the account is created, a mail must be sent to the user with an activation link to create his account. If the account isn't activated after 30 days, its deleted.



PHASE 2 (LOGIN AND REGISTRATION)

Obviously, you have to create a connection / registration management. Your users have to register to connect.

You have to handle their errors and build them like this:

- Registration:
 - Username: Between 3 and 10.
 - Email: Handle this part with regex.
 - Password and password confirmation:
 - Between 8 and 20 characters.
 - Have to be the same.
 - The message has to be “Invalid username”, or “Invalid password” or “Invalid email”.
- Connection:
 - Invalid username.
 - Invalid password.
 - The message has to be “Invalid username and/or password.”.
 - *Bonus*: If the account has been deleted since less than 30 days, prompt the user to reactivate his account.
- Logout
- Account deletion:
 - You can delete your account, but not the others.
 - The message has to be “The account has been deleted” and you will be redirected on the index page.
 - *Bonus*: The user has to be informed that is account will be fully deleted after 30 days. Until this, this account is deactivated.

PHASE 3 (ADMINISTRATION OF USERS)

In this part, you have to create an administration interface for user management.

But for now, we only will create a few of them:

- This interface is accessible only by an administrator.
 - A writer and a user can't access to this interface.
- The activation / deactivation of an account: sometimes, an administrator has to kick somebody off the site, and deactivating his account is the best way to do this.
- The creation of a user: it's very important for a lot of reasons. For example, when you have to create a bunch of users to test a feature or to create other administrators.
- The modification of a user, including his promotion to an administrator or a writer.
- The destruction of a user.



PHASE 4 (ARTICLES)

You will now create some articles because we want to publish a lot of them on our website.

So, here are their characteristics:

- Only the users who are connected and have writer rights can write articles. The writer's interface is accessible by writers and administrators.
- Articles are listed by date. It means that you will have the most recent articles on the top of your page.
- Users can see all articles:
 - Their content,
 - Their title,
 - By whom it was created,
 - Creation date,
 - Last modification date
- Users can select articles by their:
 - Title,
 - Creation date.
- We can see the comments of the article.
- They can be identified by tags: look at the associative table.
- They belong to a category, but it's none of your business for now.



PHASE 5 (COMMENTS)

Our articles should have comments because it's a good and simple way to communicate with others about their articles.

However, there are some rules for them.

- Only the users who are connected can write comments on articles.
- We can see when the comment was published and by whom.
- In an article, we can see the comments of this article.
- An article can contain many comments.
- A comment belongs to only one article.

PHASE 6 (ADMINISTRATION OF ARTICLES)

The writers of an article and administrators manage the administration side of articles and comments.

Here is the management to have for them in the administrator's interface:

- An article can be created, modified and deleted.
- A comment can be created (by everyone) and deleted.
- When the article is deleted, all his relative comments are deleted too.



PHASE 7 (CATEGORIES AND TAGS)

For this part, you have to create categories and tags for articles. They are very important because they regroup articles with some criteria. So, what's the difference between them?

It is that the category regroups articles by their principal domain / description, and a tag is like a #hashtag on Twitter and is defining only one domain / description of the article. For example, an apple and a banana can be placed in the "fruits" category while they can have the following hashtags: #fruit #healthy #red #yellow, etc.

Please find just bellow the features to manage in the writer's interface:

- An administrator can create, modify, and delete a category. The modification is the fact of adding or deleting articles in a category, and modifying its name;
- Tags can be created and deleted by writers and administrators;
- Tags can be associated with an article by an administrator and the writer of that article.
- You have to implement the research of articles which can be done by:
 - Author,
 - Title,
 - Date,
 - Category,
 - Tag.

BONUSES (HAVE A LOT OF FUN!)

- Swagger API Documentation
- Global searching
- Docker containers
- Well designed views
- Autocomplete
- ORM creation
- ...