# Exercise: Web Development Basics

This document defines the in class assignments from the ["Web Development Basics" Course @ Software University](https://softuni.bg/courses/web-development-basics/).

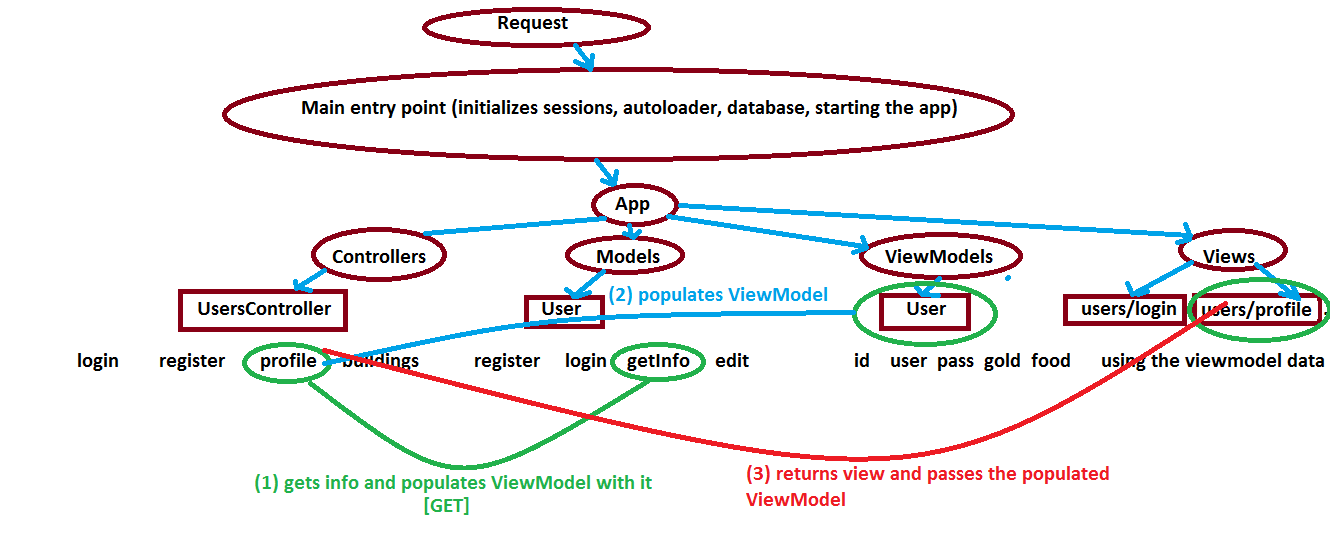
The purpose of this exercise is to create a minimalistic framework that dispatches requests upon given url in order to implement a basic MVC pattern. By solving the problems below, we will migrate the project from the State Management and Web Security Lab to use the MVC architecture.

## MVC Framework

#### Prerequisite:

1. We will use some of our old codebase – the database class, the drivers namespace and the configuration. We will follow the [PSR-4](http://www.php-fig.org/psr/psr-4/) standard for namespaces, thus we need to prefix all our namespaces with vendor. Prefix them with SoftUni namespace.
2. We will use the User object which will be a ViewModel passed to views
3. We will keep the core logic (registration, login, etc…) but will distribute it to Controllers and Models.
4. All of our folders will be named PascalCase, same applies for files that are classes

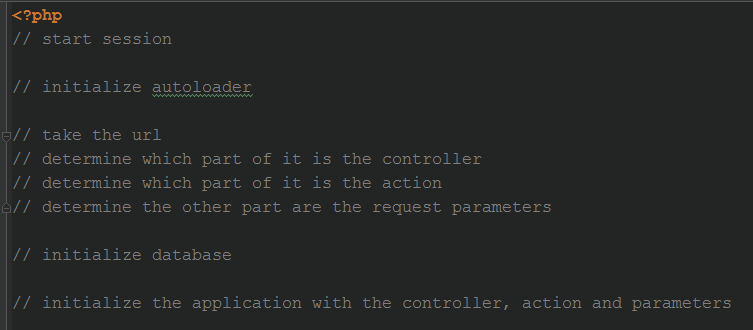
#### How does it look like:



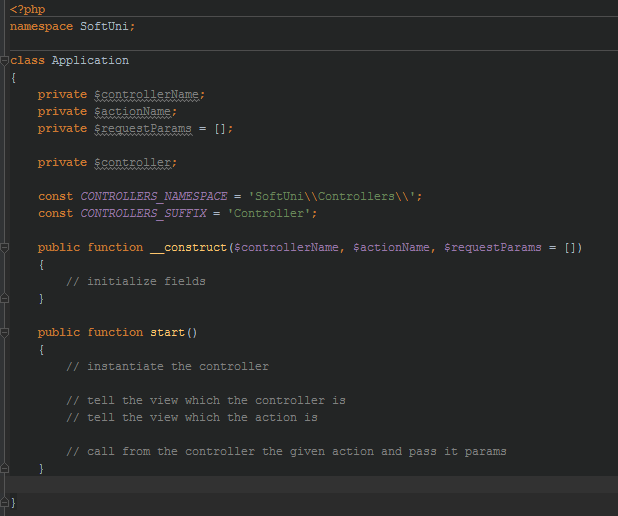
#### Constraints:

1. The routing follows up the pattern <http://site.com/controller/action/paramX/paramY/paramZ> ...
2. The framework gives the params ($params[0] = paramX, $params[1] = paramY, $params[2] = paramZ) to the action through its signature
3. Controllers returns View object which includes view templates. A view object can be instantiated with two overloads:
   1. With view name (e.g. “users/login”) and view model with data
   2. Only with view model with data

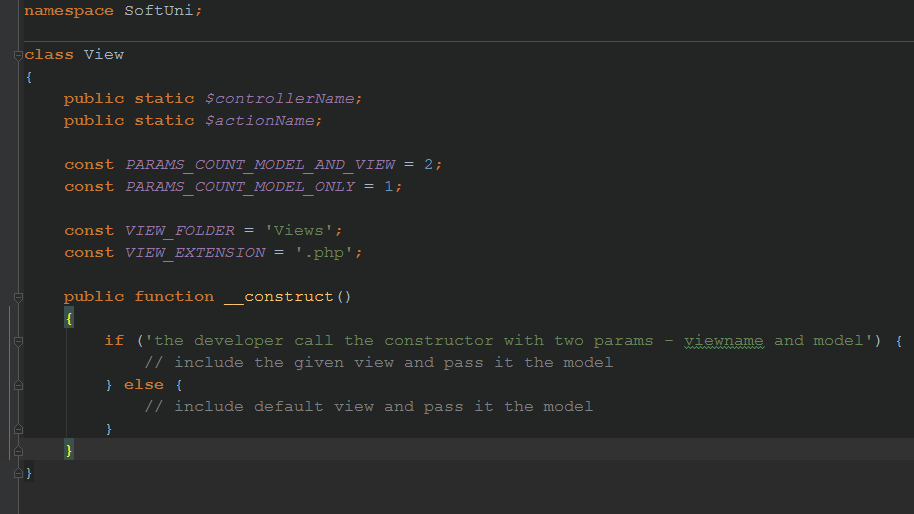
Index.php



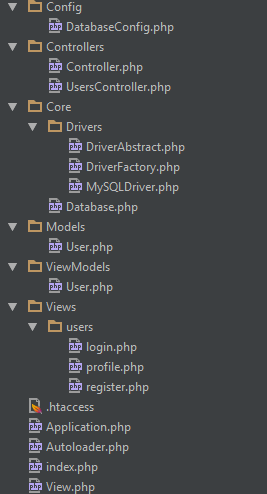
Application.php



View.php



The structure of the project including the framework at some moment might look like:

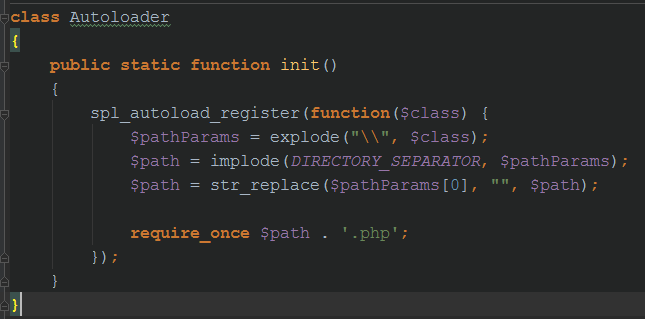


### Task 1. Create the Autoloader

Most of our work in this project will be in the terms of objects. Many new classes will be instantiated for different reasons.

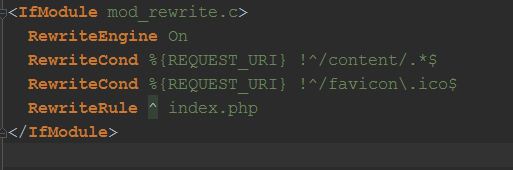
All of our classes have a template in their namespaces. They start with vendor (SoftUni) and continue with the folder structure they are in e.g. SoftUni\Core\Drivers\DriverAbstract. Thus our autoloader should clear the SoftUni part and load the file Core/Drivers/DriverAbstract.php when we try to use this class.

* Create a class in the vendor namespace called Autoloader
* Create a static method called init() which registers a new autoloader
* In the callback of spl\_autoload\_register remove the zero part from the url and include the path with the php extension



### Task 2. Redirect all requests to a single point

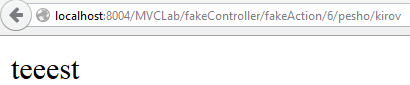
In order to redirect each request to only one file, we need to enable the rewrite module in our web server (mod\_rewrite in Apache). When we are done we need to create local configurations in the project which tells the webserver where to redirect the requests. In apache that’s the .htaccess file. Create an .htaccess file that redirects everything that is not resource to index.php



### Task 3. Main entry point

After we have redirected every request to index.php, we need to test this. Let’s create an index.php file, write something in it and open <http://oursite.com/ourproject/fakeController/fakeAction/6/pesho/kirov>

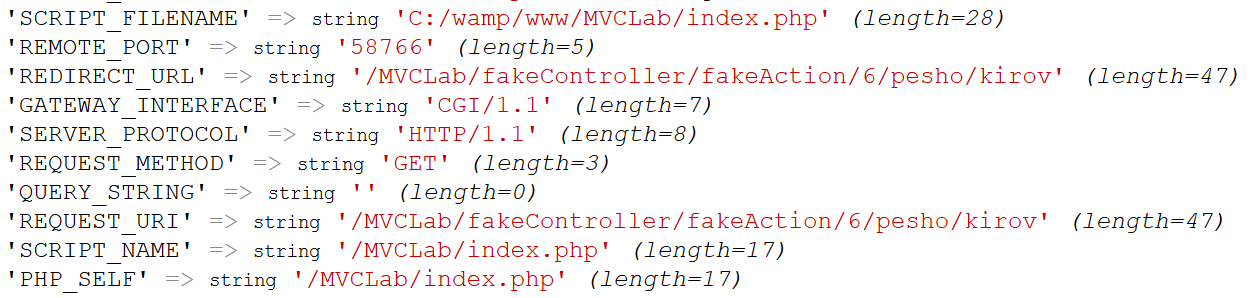




### Task 4. Routing

Each request is now redirected to index.php. We now need to read the URL in order to determine which one is the controller, the action and the other part – parameters.

Luckily, we have many of that meta-information registered in the $\_SERVER superglobal. If you dump it’s content you will see some cool things.



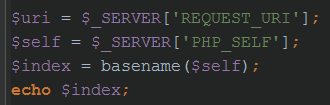
As you can see, everything after the hostname is registered in the REQUEST\_URI key. The real path and the file we have hit is registered in the PHP\_SELF key



What we need to do is to remove the path (in our case the folder MVCLab) from the REQUEST\_URI so the other part will be the whole user request.

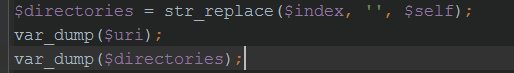
We can first remove the index.php from PHP\_SELF (the last part) so we will leave with the directories only (e.g. /dir1/dir2/MVCLab/test/, but in our case it’s just /MVCLab/)

The index.php from the path is the so called basename

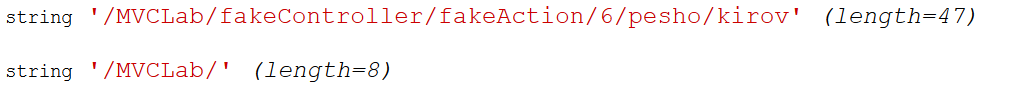


Returns ‘index.php’

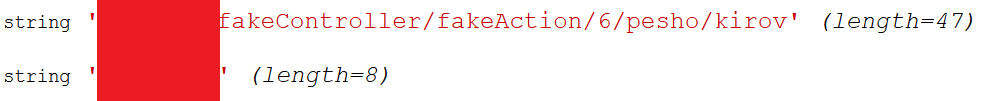
We need to remove that from PHP\_SELF so we will apply str\_replace with empty string



The result is



So now if we remove /MVCLab/ FROM $uri we will receive our desired request



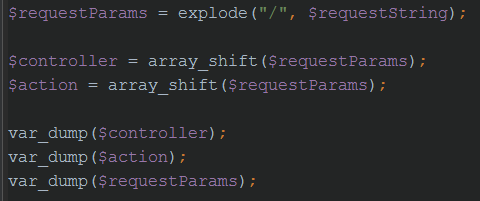
Let’s apply string replace with empty string to the directories from the uri too



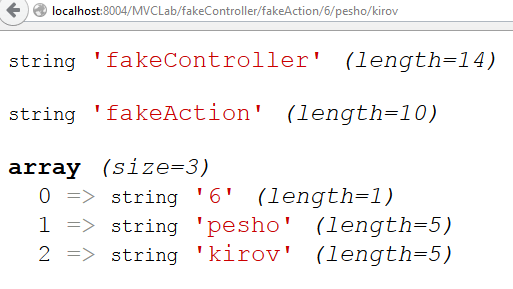
The result is



We now have the request string extracted. In order to determine which one is the controller, the action and the request params we need to split by slash, dequeue the first from the array which will be the controller. Dequeue the second from the array which will be the action and the part that has left after two dequeues will be the request parameters array



The result is



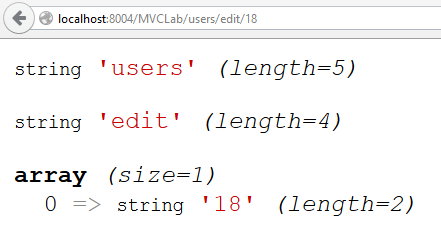
At last! We have determined our desired information. Now we need to make controller object from the string. Luckily PHP has powerful reflection API as well as native dynamic reflection. Creating an object from string is easy

**$string = “DateTime”;**

**$dateTimeObject = new $string();**

Will return a DateTime instance

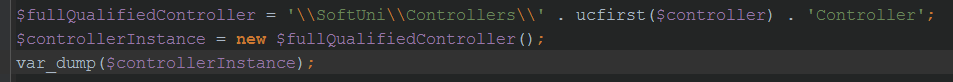
We have our controllers put in the Controllers namespace. One more convention: we need all controller classes to be suffixed with “Controller”. For example



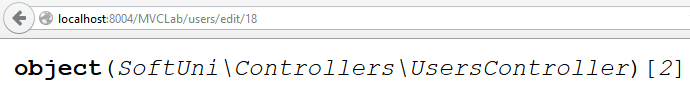
Needs to instantiate \SoftUni\Controllers\UsersController

And call ->edit(18) on it.

In order to test it we need to include the autoloader and register it, otherwise the file will not be found. Then let’s just try it



Results in



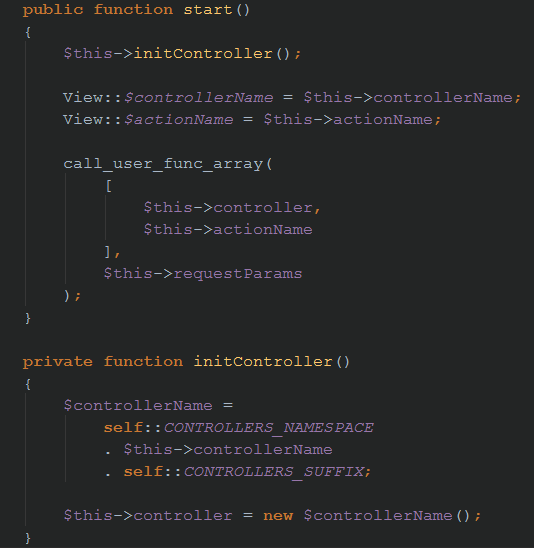
Let’s not do this in index.php. It’s enough that we have determined the controller, action and params.

### Task 5. Application class

The application class will accept the controller, action and params and upon command will run it. As we will have Views and these views might need to load default views in some case (for default views we mean a file from the Views folder, then from a folder with the same name as the controller name, then a file that is named the same as the action name), they need to know about which the controller and the action is.

So when we tell to app to start dispatching, it also needs to tell the View the controller name and the action name.

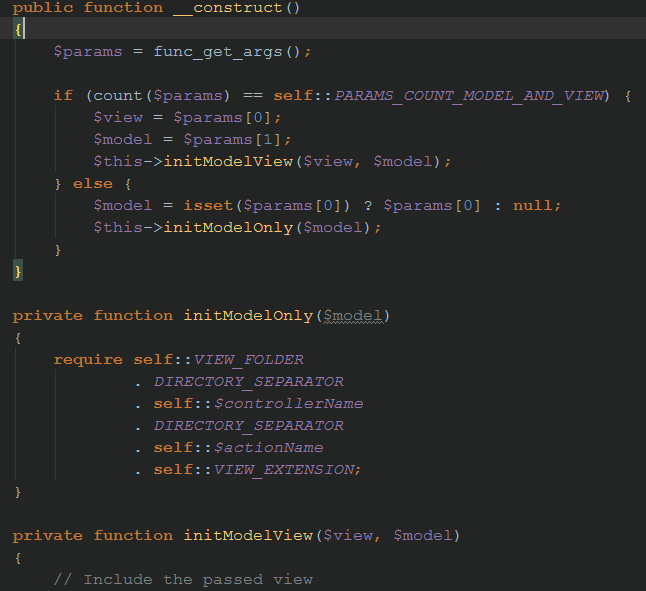
Using call\_user\_func\_array we can pass arbitrary number of parameters horizontally to a method. First we will instantiate the controller, then call the action with the params array extracted horizontally.



### Task 6. The View class

Not so long ago we have said that the View will be instantiated through two different overloads. Unfortunately PHP does not support overloads by default. Luckily there is a way to determine which are the parameters passed to the function, no matter if the function accepts any.

This can be done through the func\_get\_args() functions. It returns a numeric array with the parameter values passed to the function. We will check if it’s called with two arguments – if it is, then we include a view with a given path and pass it a model. If not – we include a default view and pass it a model, if any is passed before that.

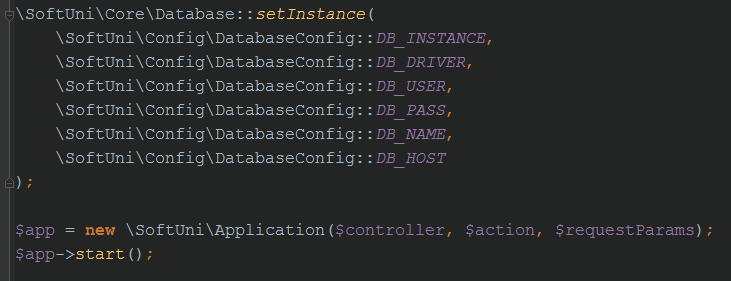


### Task 7. The Controller class

The controller abstract class should be the parent of all controllers and has all the shared logic between controllers. In our minimalistic example we have nothing to share, only if user is logged or not. So create an abstract controller in the controllers namespace and put a function that checks whether an user is logged or not

### Task 8. Set DB instance and start the Application

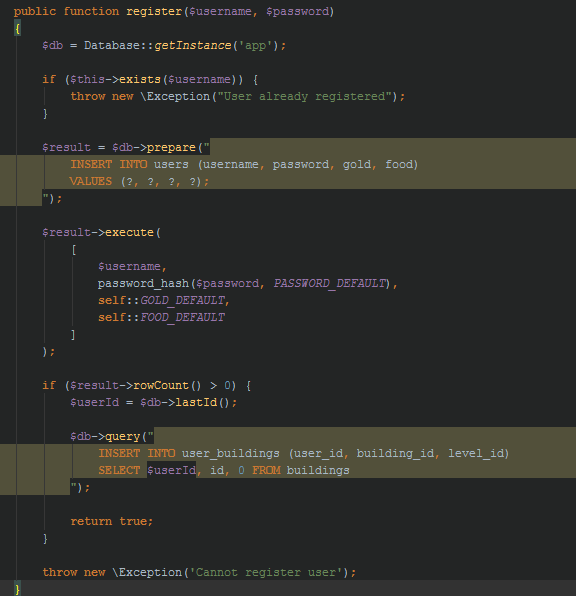
In the index file all we need to do is now to set the DB instance and start the application through its start() method



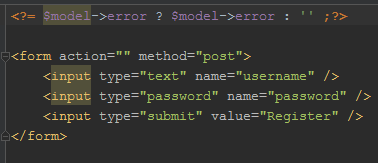
## Migrate the project to the MVC framework

### Task 1. Registration

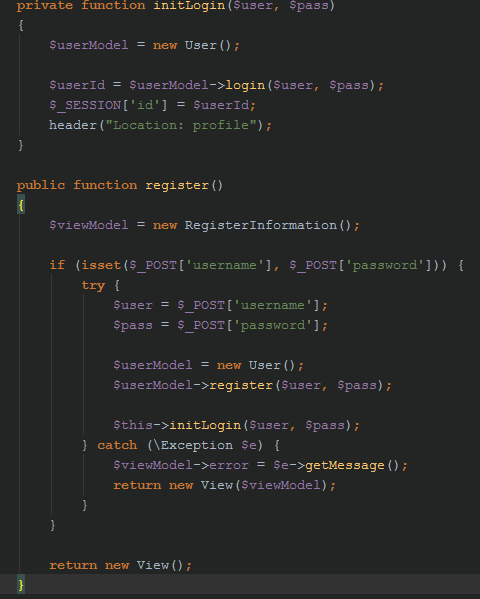
* Create a namespace Models and place a class named User there. It will be the user model which will deliver information from somewhere. In our case it’s from the database. Use the db singleton to retrieve the db instance, so the model will have access to the database and just copy the register method from the previous lab and make it use the retrieved db instance. Move the constants for gold and food there.



* Do the same for the login, exists, getInfo and edit methods (do not use ‘user’ in their names, they are members of the User class, of course they will provide e.g. Info for User). As this is the information layer do not set sessions here. Just return the user id instead.
* Go to the controllers namespace and create a class UsersController that extends the abstract controller.
* Create a ViewModels namespace if you don’t have one and create a class called RegisterInformation and create two public fields $error and $success assigned to false;
* Create Views folder. Create a subfolder called users (it’s called after the users controller). Then create a file called register.php (called after the register action)
* Put the form from the register template from the previous lab there. The register view will receive RegisterInformation view model. If the error field is not set to false – print the error on the top of the view.



* Go back to the users controller and create a register method. Create a bare RegisterInformation viewmodel. Check if the post has both username and password fields and send them to the User Model register method. If the registration passes ok, apply login and redirect. If not, assign the message of the error to the $error fields of the view model and return the view

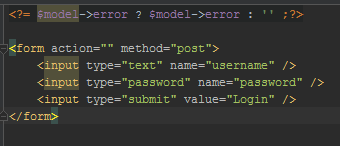


### Task 2. Login

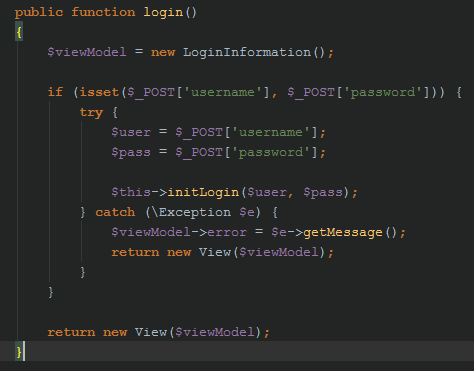
Now we need a login page. It will be <http://site.com/users/login> so we need a login() method in the users controller.

The login method will call the User model for login and if everything goes right will assign a session id and redirect. Otherwise an error will occur. We will create a LoginInformation view model which will hold error/success messages too. If an error occur, we will assign the error to the $error fields and pass the view model to the view so the last one can print it

* Login view (Views/users/login.php)



* Login action



### Task 3. Profile

Migrate the profile page. The route should be users/profile. Use viewmodels to pass information to the view.

### Task 4. Buildings

Migrate the buildings page. The route should be users/buildings. Use viewmodels to pass information to the view.