# 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/).

This assignment is about an application that will lay foundations of a multiplayer role play game. It will consist of a login, registration, profile page and page where a player can evolve buildings. The explanations below will walk you through setting up the architecture, building the database and the application itself

## Architecture

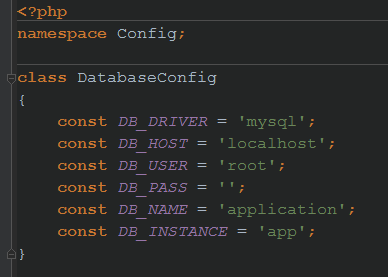
We need some mini-library in order to build our code fast and deliver what we want. Easy way to change the database configuration, register users, check if they are logged and separate the logic from the presentation. In this case we will prepare our code well, using namespaces, autoloaders, database wrappers and common application logic.

### Task 1. Define the directory structure

In the root directory we will place our files that the webserver will hit directly on request (e.g. login.php). Of course the need of configuration calls, so a “config” folder will be placed. All the common logic will be in “core” folder. Let’s create “config” and “core” folders in the main application folder. We will have “templates” folder in order to distinct the presentation (frontend) from the backend.

### Task 2. Create a Database Configuration

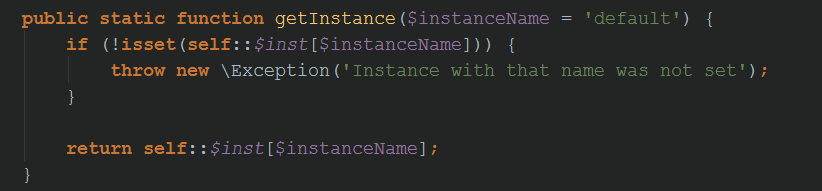
Let’s create a class holding constants with the parameters for connecting the database as well as the instance name we will later keep for the singleton connection. Its place is in the “config” folder, thus its namespace is “Config”



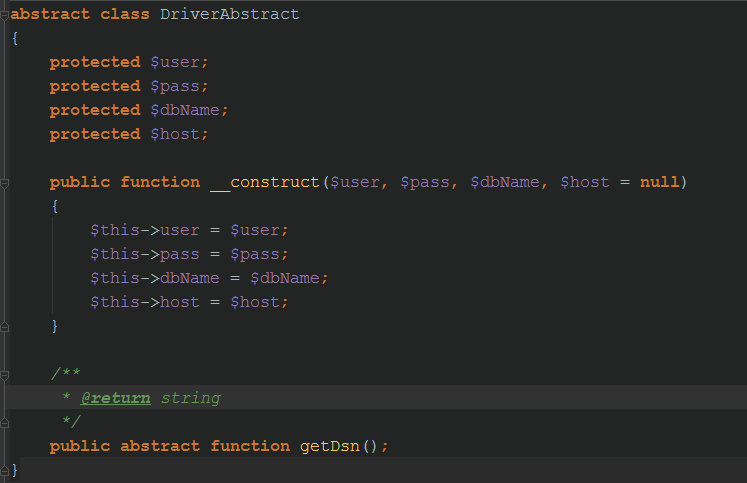
### Task 3. Create a Database Wrapper

We will wrap the common methods we need from the PDO API in our classes. A Database singleton class that accepts configuration parameters and delivers an instance of itself based on that parameters. And a Statement class that represents the PDOStatement object.

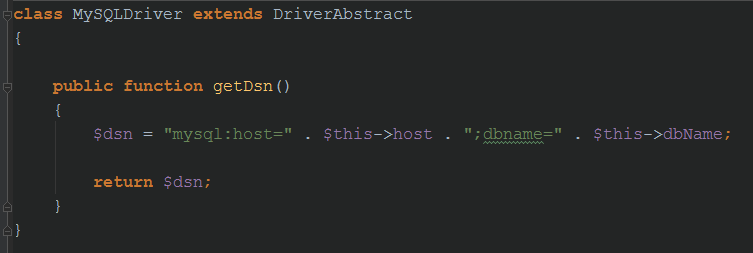
* Create a Database class in the “core” directory.
* It needs a static array of instances in order the singleton pattern to deliver given instance. Let’s call in $inst
* It also needs a private field holding the PDO instance which will be created later through a static method. Let’s call the field $db
* Make a private constructor so nobody will instantiate the class from outside. It needs to accept a PDO instance and assign it to the db field
* Make the getInstance() method to accept instance name and return the given instance from the array of instances



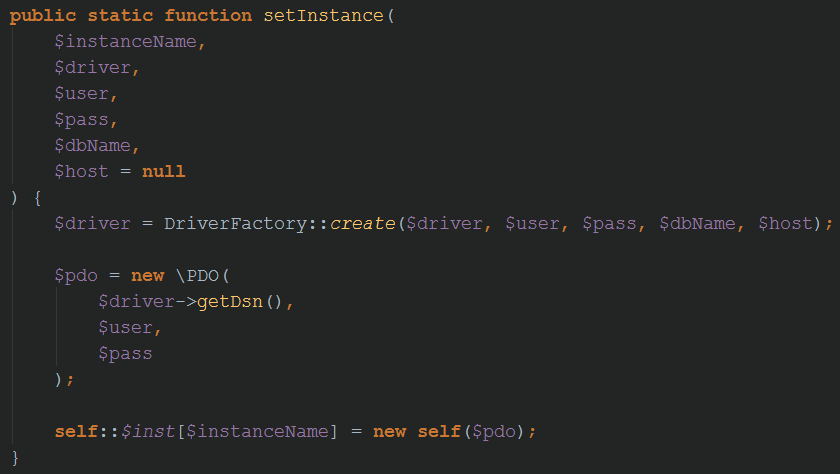
* In order to set an instance with a given name/key we need a method to do it. It will receive the configurations in its signature and kickstart a new PDO instance. The PDO instance needs some driver specific things like a DSN, so let’s expand it a little bit further. We will need classes that upon given database parameters will return driver specific information, e.g. the DSN. We will only create a class for MySQL, but if we change the database, we can easily create new one.
  + Create a “drivers” folder as a child of the “core” folder
  + Create a class named DriverAbstract and assign it the relevant namespace Core\Drivers



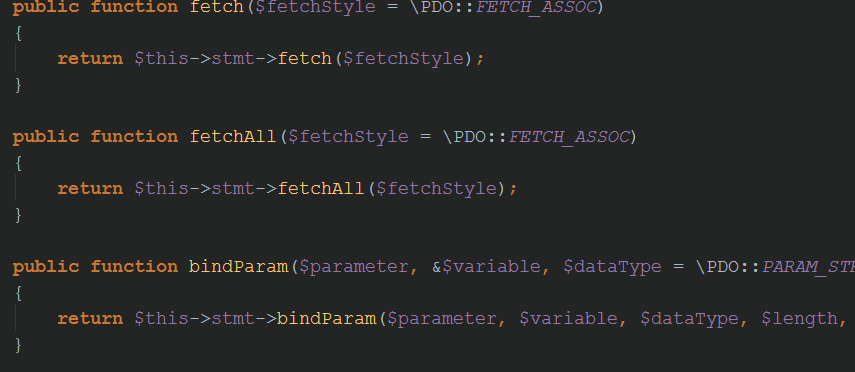
* + Create one implementation of this class named MySQLDriver that extends it, in the same namespace. The class will return the DSN for MySQL database



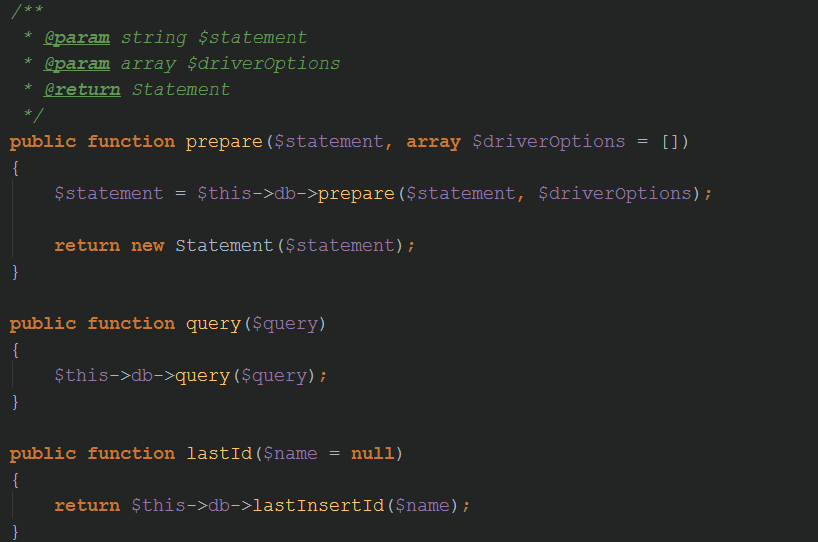
* + In the same namespace create Factory class named DriverFactory that will return upon given database name (e.g. mysql) an instance of concrete implementation of DriverAbstract (e.g. MySQLDriver)
* Let’s get back to the Database class. We are ready to implement the method which will set the instance in the array. It needs to accept the instance name/key, driver type (e.g. mysql), user, pass, dbname and optional host (because some driver does not require it, e.g. sqlite). It will pass the configuration to the factory, so will get an instance of the concrete driver, which will give us the DSN, which later we will pass to the PDO object. The PDO object we will pass to our private constructor so we can create an instance of Database class and store it into the array.



* Create a Statement class (it’s ok to create it in the same file where the Database class is). It needs a field that will hold the PDOStatement instance (let’s call it $stmt) and a public constructor where PDOStatement instance is passed and then assigned to the field. This object will be created from the Database class whenever a method that needs to return a PDOStatement is called. It will return our Statement object instead.
* Wrap the **fetch(), fetchAll(), bindParam(), execute() and rowCount()** methods in it. Just look at the PDOStatement class and make methods with the same name, that returns the value of the original ones as in the example below

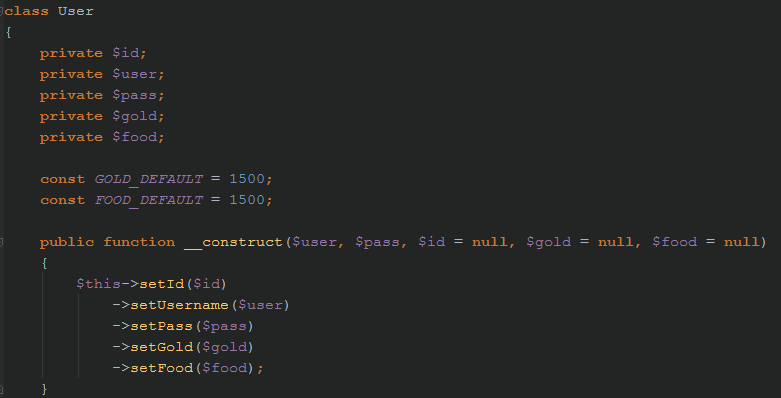


* Go back to the Database class and wrap **prepare(), query() and lastInsertId()** methods from the PDO class. The prepare() method needs to return our Statement object instead of original PDOStatement.



### Task 4. Create a class that describes the User

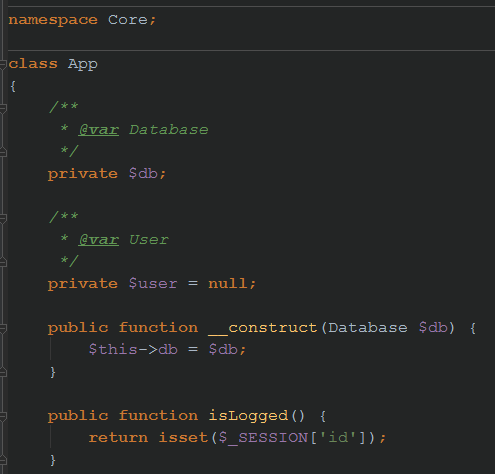
In our game the users will have ID, Username, Password, Gold and Food. Create a class (User) in the Core namespace which has fields for that a properties per each field. Additionally describe in two constants the default value of Gold and Food each user will start with (e.g. 1500). It will be good if you make the setters to return the current instance of the user, so you can apply method chaining.



### Task 5. Create a class holding the common logic

In the Core namespace we need to create a class that will hold some common logic (login, registration, current user, etc…). Let’s call the class App.

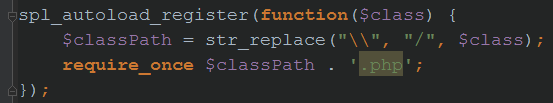
* This class needs to have direct access to the Database (using our Database wrapper)
* Will populate the User object and use its properties when needed
* Can answer whether the user is logged in or not



### Task 6. Create bootstrap file which will have common logic that needs to be executed upon each request

There is some logic we might want to be executed on each request e.g. starting a session, autoloading classes, setting the db configuration and creating the App object.

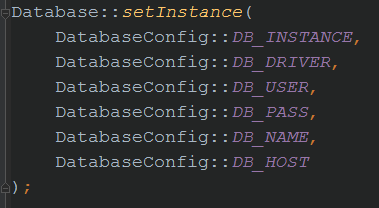
* Let’s create a file in the root directory called index.php which will be our bootstrap file that we will need to include in every file in the root directory from now on.
* Start a session in it
* Create an autoloader using the spl\_autoload\_register hook. It accepts a callback with the full classname (with namespaces) as a parameter each time a class is requested. We will replace the backslash from the namespaces to a normal slash and then require the fullpath. If a class is in Core\Drivers namespace and is named DriverAbstract, when we try to use it, the autoloader hook will receive as a parameter “Core\Drivers\DriverAbstract” string. We need to extract a path from it, and append .php extension in order to create “core/drivers/DriverAbstract.php” from that string and include the file



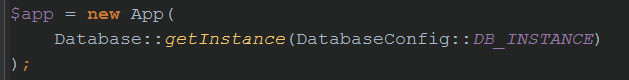
* Include the App class



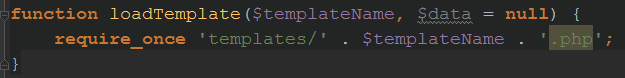
* Set a db instance with the parameters from the config



* Instantiate the App object, passing it the Database object



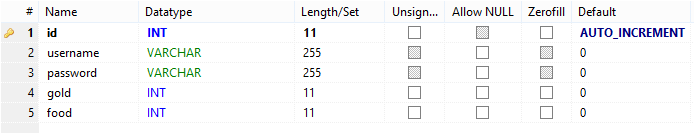
* In the beginning we have said that we will have a “templates” folder holding the files with the presentation. Let’s create function that upon given filename will include a file from “templates” as well as receiving a data parameter with some information that the template will have access to it (e.g. the current user)



## Application: Database

### Task 1. Schema

* Create a table holding user details (e.g. users), which has columns for id, username, password, gold and food



* Create a table describing buildings. They have id and name
* Create a table that describes how many levels each building has and how much they cost. Building\_id, level, gold and food
* Create a table that relates that players to the buildings and their levels. The table should describe each user which building on which level has. Id, user\_id, building\_id and level\_id.

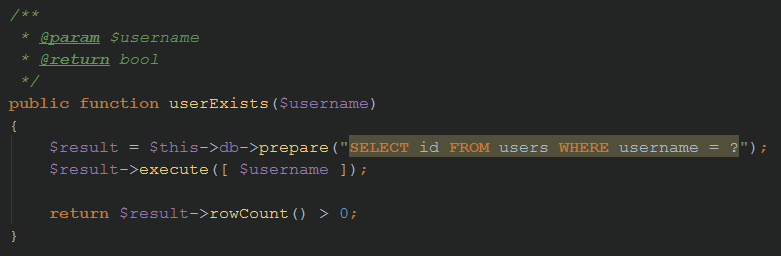
### Task 2. Data

Add some buildings in the buildings table and their level and costs in the building\_levels table. Each building should have level 0 with 0 cost so the players will start initially with level zero.

## Application – login and registration

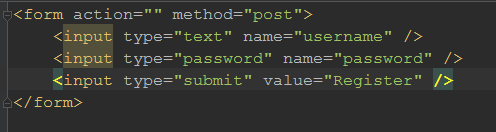
### Task 1. Registration

* We will create in the Application class a method which registers an user. We need to check whether user exists or not, so we need a method that returns this information. Each time an user is registered, one needs to start with level zero on each building. For password hashing we will use the PHP’s built-in functionality for [password hashing](http://php.net/manual/en/function.password-hash.php) and [password verifying](http://php.net/manual/en/function.password-verify.php).

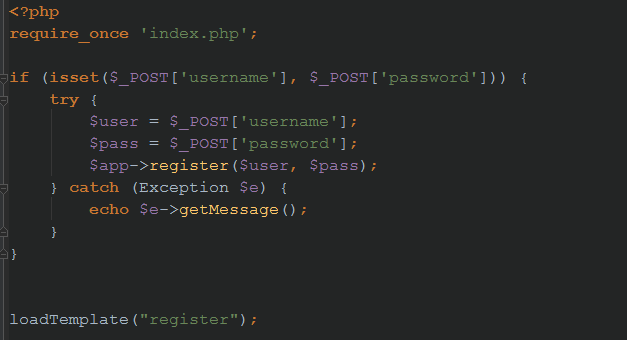




* Create register.php in templates where the presentation for registration will be placed. It will be a form consisting of username and password an a submit button

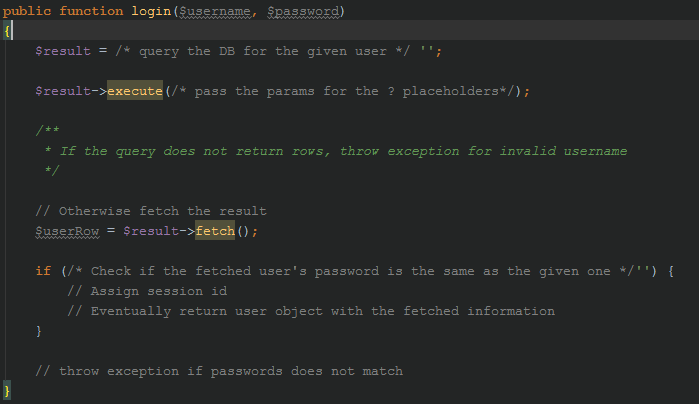


* Now let’s create a backend logic that will handle this and load the register template. Create a register.php file in the root directory. It needs to include the index.php. It should take the POST Request and pass the fields to the register function. It should load the “register.php” template.



### Task 2. Login

* Create a login method that accepts username and password in the App class and assigns session ID if the user has entered correct credentials. Otherwise throw exception. Optionally return Core\User object with the fetched info

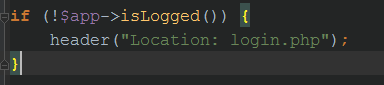


* Create a login template with a form that posts username and password
* Create backend that handles it and if the login is successful to redirect to another page (e.g. profile.php)
* Change the registration backend. If registration is successful, automatically call login and redirect to profile.php if successful.

## Application – profile page

### Task 1. Protect from unwanted visits

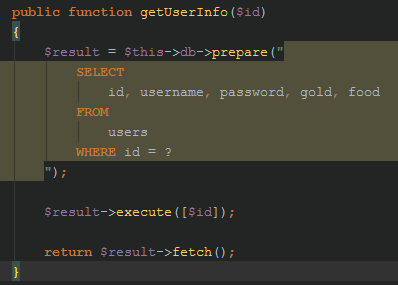
The profile page should be opened only from logged users. Our first task is to redirect the guests to the login page if they try to access it.



### Task 2. Pass user information to the frontend

The template needs information regarding the user in order to display some things like its name and its resources.

* Create a method in the Application class that returns user information by given id



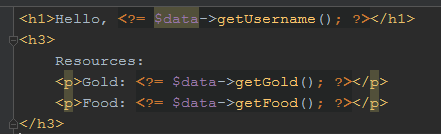
* Create a method that returns a Core\User instance with the information of the logged in user. Make the method pseudo-singleton. It should not call getUserInfo twice or more on the same request. Store the current user in the $user field and if it’s populated, just return the field. Otherwise populate it and return it.



* Go back to the profile frontend and load some template (let’s say profile) and pass to it the user object



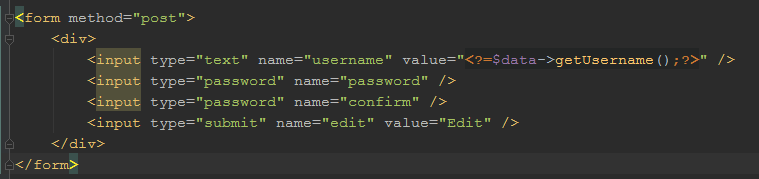
* Now the $data param in loadTemplate receives the value of $app->getUser() which is Core\User instance. So the $data variable is now of type Core\User. Since the templates can access the $data variable, they can call methods from Core\User using the $data variable (e.g. $data->getUsername())
* Create a profile template and show information regarding the username and the user resources.



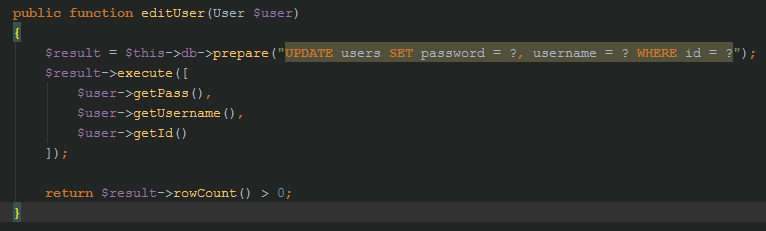
### Task 3. Edit user profile

Let’s provide functionality to the user to edit its profile.

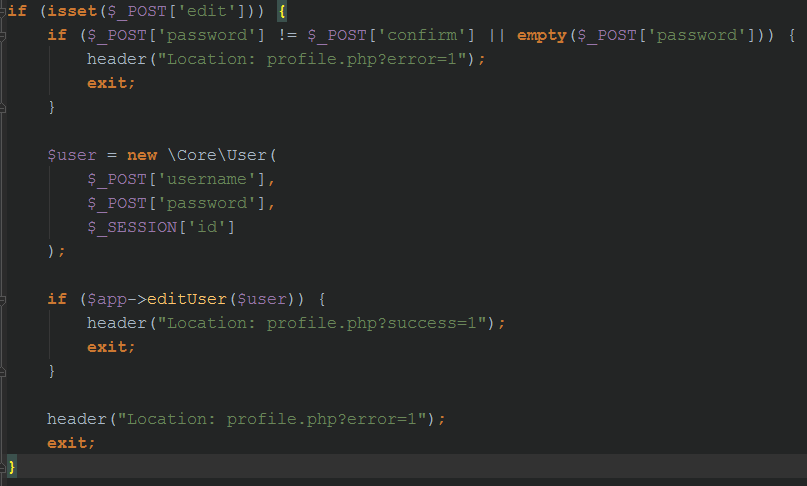
* Create a form consisting of fields for changing username and password. Make a confirmation field for the password. Display the current username in the username field.



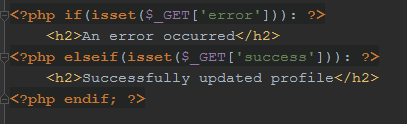
* In the application class create a method that edits user row (username and password) in the db upon given Core\User object. Return true on success and false otherwise.



* Create a backend logic in the profile page which handles the POST request and on valid data – calls the edit method with newly created Core\User object from the Request data. If everything goes OK – redirect to self with ?success=1. Otherwise redirect to self with ?error=1.

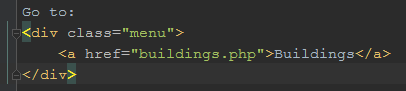


* Display messages in the profile frontend if the page is called with success or with error



### Task 4. Display link to Buildings

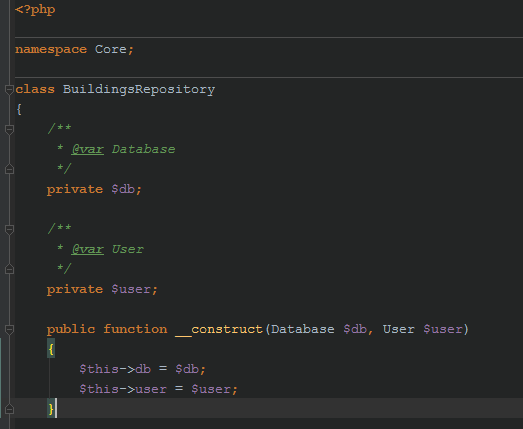
Just make an anchor in the profile frontend to buildings.php which is about to be implemented



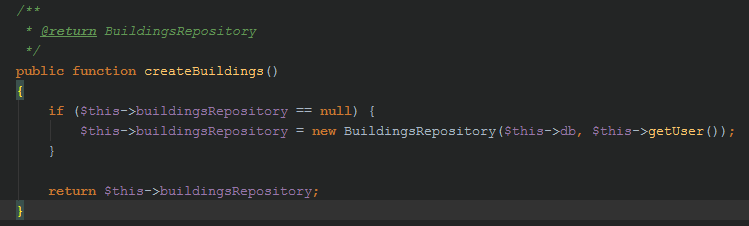
## Application – Buildings

### Task 1. Buildings repository

* Create a class in the Core namespace called BuildingsRepository. It will hold application logic regarding user buildings. It needs to have access to the DB Object and the User Object

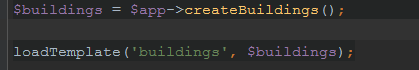


* Make the application class to return its instance injecting it the db and the user object. Make it pseudo singleton too.

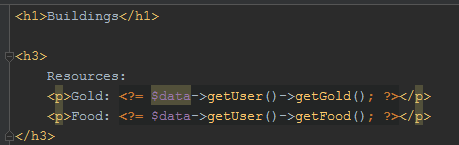


### Task 2. Show basic information

* Create a buildings backend (buildings.php) in the root directory.
* Protect it from an unwanted visit
* Take the buildings repository instance from the Application class and pass it to the frontend



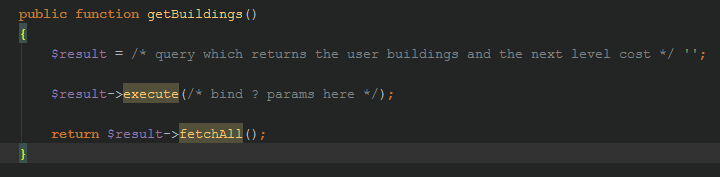
* Create the buildings frontend template and show users resources. Since the $data variable is now Core\BuildingsRepository, the template has an access to this object. One may extract user data from the User object that was passed to the BuildingsRepository.



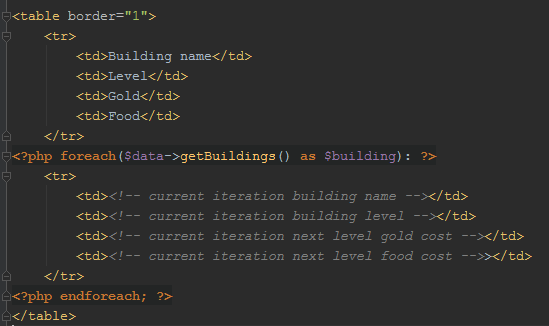
### Task 3. Show buildings information

We need to show in a table view each building. On which level the user has it. And how much the next level costs.

* Go to the buildings repository and create a method that returns such info from the database (building name, next level cost, building id)



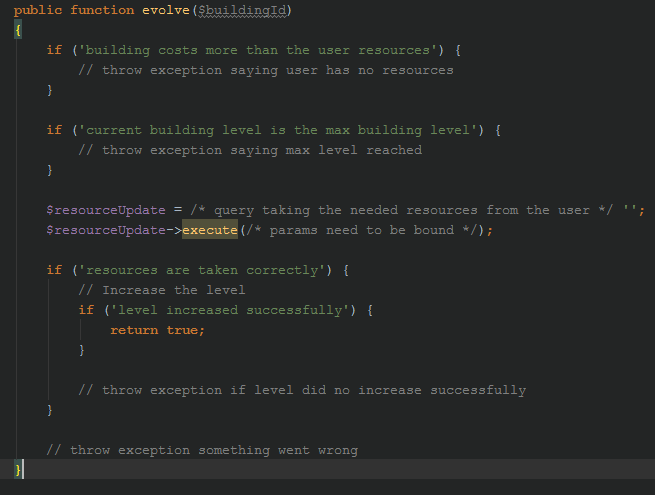
* Using the associative array returned from this method, show the information in the table view



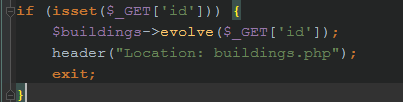
* Add one more column which is a hyperlink leading the user to evolve to next level. Pass the building id as a GET parameter.

### Task 4. Evolve a building

* Create a method evolve() in the buildings repository that evolves certain building for the current player



* In the buildings backend handle the GET request where a building id might be passed and call the evolve method.



## Defend from XSS and CSRF

### Task 1. Defend from XSS

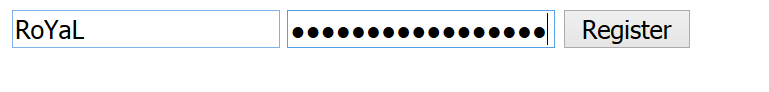
Escape unwanted characters that might be presented in the templates and were initially inserted by user input

### Task 2. Defend from CSRF

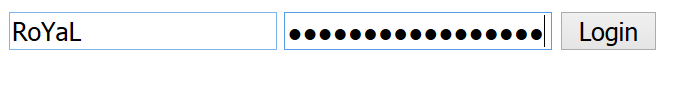
Everything that happens via request on the user’s behalf should be protected by verification token.

# Expected result

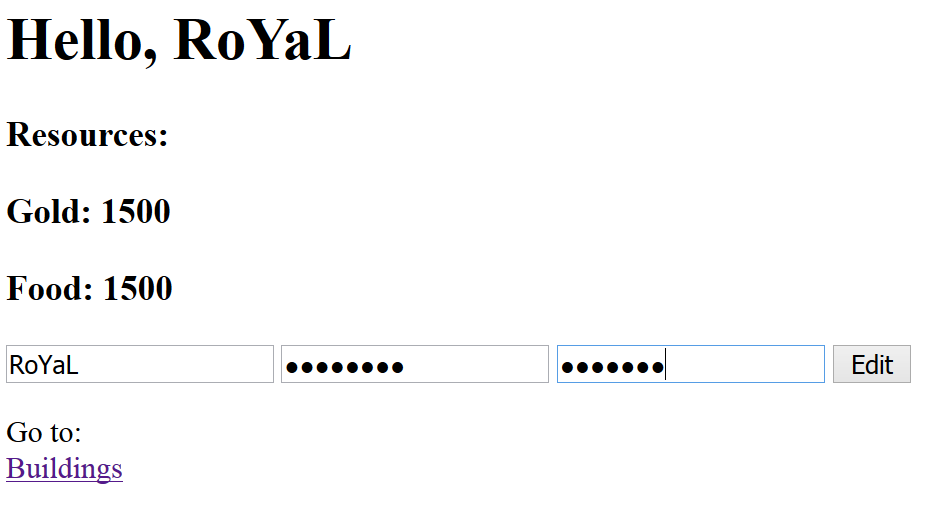
1. User hits /register.php – form appears. Duplicate user – gives error. Valid user – creates the user, executes login and redirects to the profile page



1. User hits /login.php – form appears. Invald user/credentials error is the user/pass is wrong. Valid user – logins and redirects to the profile page



1. User hits /profile.php – if one is not logged in – redirect to login.php. Otherwise shows information regarding the username, its resources and a form for editing username/password. If passwords mismatch and form is submitted – error appears. Otherwise the information is changed successfully.



1. User hits /buildings.php – if one is not logged in – redirect to login.php. Otherwise shows tablic information regarding user buildings. Clicking on evolve – evolves the building and takes the resources from the user. If the user does not have resources to evolve – errors appears.

