APLICAÇÕES PARA A INTERNET

Engenharia Informática

PHP MVC (Part 1)

Objectives:

(1) Use of Object Oriented Programming approach in PHP for creating a complete CRUD;

(2) Follow the Model-View-Control architectural pattern;

(3) Use composer to autoload classes.

Note the following:

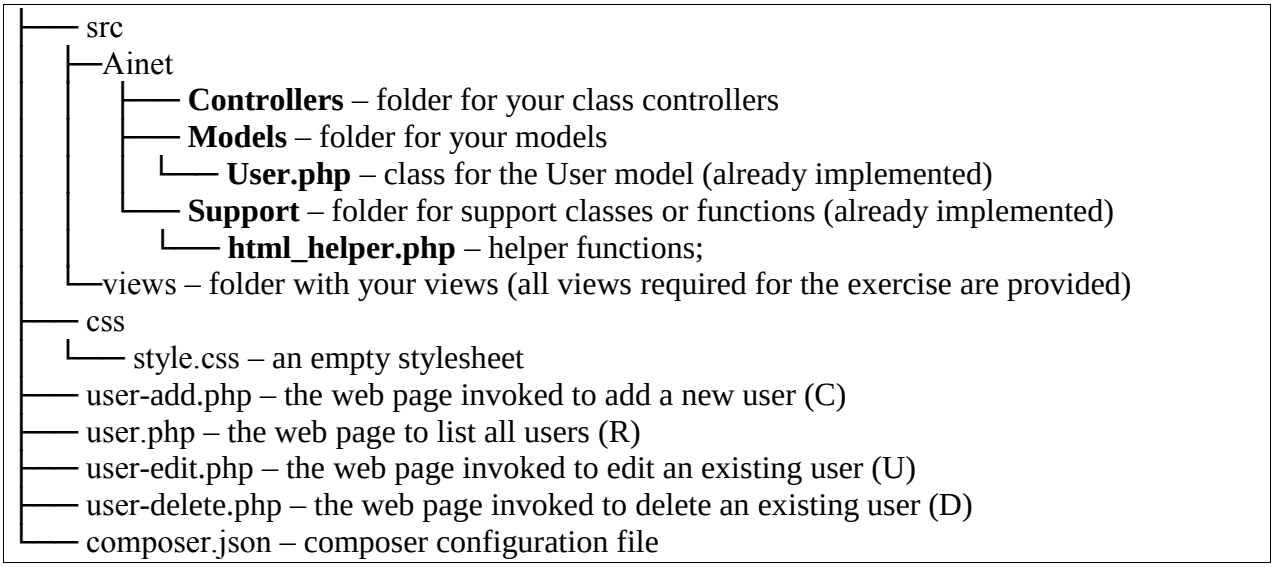
- Every page MUST comply to the HTML5 standard and it must be properly validated as such;
- The web application should follow the Model-View-Control architectural pattern;
- The php code written MUST follow the "coding style guide" PSR [PSR-2, PSR-4];
- There is a video for each exercise showing the intended result;

Setting up the environment:

- If the Vagrant Homestead environment is not installed and configured, follow the instructions of worksheet 1.
- Run “vagrant up”
- Store your exercises on the “C:\<AInet>\pratica\ficha4” folder.
- Note: In alternative, create a site on Laragon (or similar tool) and, if possible, use the same URL to access the content (http://ainet.pratica.4.test/)

In this exercise, you will build a basic CRUD (create, read, update and delete) for user registration. For each question there is a video with the expected result. So, before starting each question please watch the corresponding video.

1. Download from Moodle the base.zip file and extract it to the **ficha4** path. The folder structure is the following:



Discuss with your teacher the folder structure paying particular attention to the **User** class and the functions inside the file "**html_helper.php**" mainly the reasoning behind the **render_view** function;

2. Open the composer.json file and discuss with your teacher its purpose. Now, from command-line (suggestion: git bash), go to the **~/Homestead** folder and type:

   ```
   > vagrant ssh
   ```

   Inside the guest machine, go to the ficha4 folder and run composer:

   ```
   > cd c:/ainet/pratica/ficha4
   > composer install
   ```

   Open the user.php file and discuss the first line with your teacher;

3. The **user.php** page lists all users from the **User** model. So, before proceeding, read carefully the code from the **User.php** class located inside the *src/Ainet/Models* folder. Pay special attention to the namespace of the class, the list of instance variables and the static methods that the class provides. Read also the code from **user.php** page (it will become important later). Detailed steps:
   (a) Create a new controller class called **UserController** in the namespace Ainet\Controllers that should follow PSR-4. The controller should be saved to the folder *src/Ainet/Controllers*;

(b) Add to the **UserController** class a method called **index** that should fetch the list of registered users (obtained by calling the **User::all()** static method) and pass it to the **render_view** function to generate the html output. Example:

```
render_view('users.list', compact('title', 'users'));
```

(c) Update the corresponding view (file **src/views/users/list.view.php**) in order to display a table with all users. Each row should display the email, full name, date/time of registration and type for each single user. You should also include for each user the following actions: edit (invokes the **user-edit.php** web page) and delete (invokes the **user-delete.php** web page).

4. Implement the "Add user" feature by changing the following files: **UserController.php**, **add.view.php**, **add-edit.view.php** and **user-add.php** page. In the process, you can add new support functions or new methods to existing classes besides the methods mentioned below. Detailed steps:

(a) Add to the **UserController** class a method called **add** that should render a view passing the title, the user instance submitted and an associative array with pending errors. If the form has not been submitted yet, pass a new user and an empty array. Example:

```
$title = 'Add user';
$user = new User;
$errors = [];
render_view('users.add', compact('title','user','errors'));
```

(b) Add code to the **user-add.php** page to invoke the **add** method previously defined;

(c) Update the add-edit view (file **src/views/users/partials/add-edit-view.php**) and change the type field option values to correspond to the following list:

i. 0 – Administrator;
ii. 1 – Publisher;
iii. 2 – Client;

(d) Add code to the **add** method defined in (a) to validate the submitted form. Use the following validation rules (same rules as in the previous worksheet):

- fullname is required and must only contain letters and spaces;

- email is required and must be properly formed;
- password must have at least 8 characters;
- password confirmation must be equal to password;
- type must be a valid value;

(e) Add to the corresponding views the code required to post-back the filled form and to show any errors the form may have. You can include the partial view located at **src/views/users/partials/errors.view.php** to show any errors;

(f) Finally, if the form doesn't contain errors you should invoke the **User::add** static method and redirect the browser to the list of users (**user.php**);

5. Implement the “Edit user” feature by changing the following files: **UserController.php**, **edit.view.php** and **user-edit.php** page. Detailed steps:
   (a) Add to the **UserController** class a method called **edit** that should render a view passing through the title, the user instance of the user to edit and an associative array with pending errors. The first time the page is rendered, the user id should be fetched from the GET array. If the form is submitted, the user id should be fetched from the POST array. If the id parameter is missing or the user is not found, the page should redirect the user to the **user.php** page. You can fetch a user instance by calling the **User::find** static method;

   (b) Add code to the **user-edit.php** page to invoke the **edit** method previously defined;

   (c) Update the corresponding view (files **src/views/users/edit.view.php**) in order to correctly display the form (fill up the action attribute and user id hidden field);

   (d) Follow the same guidelines for form validation and error presentation. If the form doesn't contain errors you should invoke the **User::save** static method;

6. Implement the “Delete user” feature by changing the following files: **UserController.php** and **user-delete.php** page. Detailed steps:
   (a) Add to the **UserController** class a method called **delete** that should fetch the user id from the POST array. If the id parameter is present the method should invoke the **User::delete()** static method. At the end the user should be redirected to the **user.php** page.