# Exercise: PHP CRUD Application

## Users and Emails

This document defines the homework assignments for ["PHP Web Development Basic" Course @ Software University](https://softuni.bg/trainings/1746/php-web-developmentbasics-september-2017). Please submit ……………………………………………….???

|  |
| --- |
| Dowload from <https://softuni.bg/trainings/1746/php-web-developmentbasics-september-2017#lesson-6673>  The file **"14. Session-Handling-Demos.zip"** and install them under a new project in PhpStorm. |

## Preface

Imagine that you are a developer which needs to extend in the proper way given to you code. In this exercise we will use the **same architecture** as in **Session Handling Exercise**. We will extend the User application by extending its functionality with more tables and functions.

## Architecture and Layers

Just to remind the architecture we have **multiple layers**. We will have the following layers:

|  |  |
| --- | --- |
| * **Front layer** – the closest to the user layer. It is the script that will be triggered from the user input (e.g. user.php, index.php and so forth): |  |
| * **HTTP Handling Layer** – the next layer, which the front layer will directly operate. This layer is responsible for the HTTP related decision – whether to redirect, view html or handle form submission: |  |
| * **Service layer** – all the business operations will be here. In order the HTTP Handling Layer to perform real actions, it will operate with so called services. A business operation is an operation by specification (e.g. register, login, logout, check for logged in user and so forth): |  |
| * **Repository layer** – all the queries to the real data will be here. In our case we will use Relational Database, so our concrete classes will operate with the database, but it will be hidden behind interfaces, so we can easily change the operation way or mock it for unit testing: |  |
| * **Data layer** – all the data which is strongly typed. We will never use data as associative arrays or other error-prone way, where the human error is with high chances: |  |
| * **Templates layer** – all the frontend. In our case – HTMLs: |  |

## Loading of Code and passing of instances

You have multiple scripts from which execution starts which are in your root folder in the **Front** Layer (see the first screenshot in point 2). As an example lets take the user profile in **profile.php**:

|  |
| --- |
| profile.php: |
| 1. Load **common.php**   1.1. Start the session  session\_start();   * 1. Auto loads classes according to folders/namespaces   spl\_autoload\_register();  1.3. Load the core template  $template = new \Core\Template();  1.4.Loads the core data binder $binder = new \Core\DataBinder();  1.5.Loads DB ini  $dbInfo = parse\_ini\_file("Config/db.ini");  1.6. Connects to DB $pdo = new PDO($dbInfo['dsn'], $dbInfo['user'], $dbInfo['pass']); $db = new \Database\PDODatabase($pdo); |
| 1. Create new instance of **Repository**   $userRepository = new \App\Repository\UserRepository($db); |
| 1. Create new instance of **Service** $userService = new \App\Service\UserService($userRepository); |
| 1. Create new instance of **Handler**   $userHttpHandler = new \App\Http\UserHttpHandler($template, $binder); |
| 1. Start a particular **handling method** of the Handler   $userHttpHandler->profile($userService, $\_POST); |

Simply put with a few words the architecture can be summed in three words Template, Service and Handler as we use such classes for abstraction. But all these names are **just descriptions of our classes hierarchy**.

## 3.Users Functionality

## Problem 3.1. All Users Paging

Create paging for all users. **What class/classes would change?** Go back to the process of **loading the code and passing of instances of classes to see:**

* What is the **Front Layer file** that loads the users?
* What is the **repository**?
* What is the **service** that is used?
* What is the **handler** that I used and what **handling method** is used?

Use $\_GET to take the number of the pages. If you don't have enough users in the database, register 5 new users. Let the maximal number of results per page be 3, so you will have at least 2 pages. Create links to the pages in the bottom in "user/all.php".

You might have also to work on bindings but this time by passing $\_GET instead of just $\_POST.

## 3.Emails Functionality

## Problem 3.1. Emails of Users

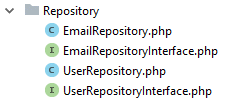
Create new functionality to be able to store emails of users. You will create a new table **emails** which contains only the fields: **id, user\_id, email.**

One user can have more than one email! Every email should have an **ID** and **user\_id** to be related to a particular user.

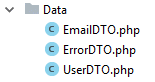
To add the functionality we need to go through:

We have to do the following, going through the different layers in the architecture:

## Step 1. Email Repository and Email DTO

Go to **/App/Repository** and study the functionality in UserRepository.php and UserRepositoryInterface.php. You should pass to your new constructor $db of type DatabaseInterface. You can call your new class **EmailRepository**.

The **EmailRepositoryInterface** defines all the needed methods that need to be implemented. What we want to be able to do is: Insert email, find emails by user\_id and update.

Go to **App/Data**. In order to use the **data as an object** we will need to create **Database Table Objec****t (DTO).** We can create a **EmailDTO** which will represent the data in the Email table. The EmailDTO should have a create() method and all the setters and getters.

We will **use EmailDTO** inside the **EmailRepository** class!

## Step 2. Email Service

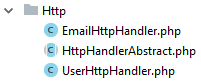
The **EmailService** class and corresponding **EmailServiceInterface** does the actual work with the emails. Notice what you have to pass to the constructor: $EmailRepository of type **EmailRepositoryInterface**.

What we want the classes to implement here is:

* Add new email after user is logged
* View all emails per user

We will need the corresponding methods to do so.

## Step 3. Email HTTP Handler

To handle HTTP requests related to the emails we will need a handler. It will be **EmailHttpHandler class**.

What we have to handle is the following:

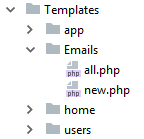
|  |  |  |
| --- | --- | --- |
| **Handler** | **Template** | **Handle Error** |
| allMail | "emails/all\_emails" – if user is logged  "login.php" –if user is not logged | Email already exits. |
| newMail | "emails/new" – if user is logged  "login.php" –if user is not logged |  |
| removeMail | After remove redirect to **allMail**  "login.php" –if user is not logged |  |

## Step 4. Email Templates

To enter in the mail he main entry point to change the email will be email.php in the root of the project. What we have to do in this file is something like:

|  |
| --- |
| <?php *// front layer* require\_once 'common.php';  *// Todo - user functionality* $EmailRepository = new \App\Repository\EmailRepository($db); $EmailService = new \App\Service\EmailService($EmailRepository); $EmailHttpHandler = new \App\Http\EmailHttpHandler($template, $binder); $EmailHttpHandler->allMail($EmailService); |

The template files look as a hierarchy the following way:



## Step 5. Making Emails Work with Users

Because **emails are on per user basis**, we have one more thing to do: combine users and email. We should check if the user is already logged in and only then be able to display, add and remove emails. Somehow we need to **pass the user information** from script to script.