# Homework: Creating an ORM/View Engine

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

## Preface

The web development as we know it, would not be possible without the existence of the Hypertext Transfer Protocol (HTTP), because it highly relies on the standard set by it: browsers send HTTP verbs, headers and so forth by specification and so do the web servers on the opposite side when accepting them.

The development of the **HTTP Protocol** has started in the late 80s, at the same time the iron curtain started to fall, and the Bulgarian communism was just about to crumble. Tim Berners-Lee initiated the HTTP standard in CERN in the eventful year of 1989.

The **early adopters of the web programming** started creating their application in low level languages as C and doing thin TCP layers, parsing the plain-text of the HTTP payload given by the then-most-popular web browser – Mosaic, and giving the relevant information to the compiled C program.

The cumbersome introduced by the handcrafting everything started to decrease by introducing the first ever web server **– CERN httpd**. Fast-forwarding the story, scripting languages started to get popular in this field, mainly because **web scripting was very much resembling the unix scripting culture**. The then-most popular winner for web development raised – **Perl** and its benevolent dictator for life – **Larry Wall**.

In the mid-90s when web development’s popularity started to increase crazily (and later resulted in the .com boom), Perl was on the top of the chain and **PHP just started its remarkable journey**. Scripts serving this-and-other business case were written here-and-there and copied to the web server’s active directory (once it was **cgi-bin**, but if you are reading this, you are lucky to never have used this directory).

From the above point until the time this document is written, more than two decades have passed. In between these decades, especially the last one, many things have changed: the regular home user started to have regular internet access (do you remember the times, when “internet” was just a term?), users start to find each other all over the world**, browsers start to have more features**, devices evolve too, microcomputer devices started to have been managed by internet (IoT), user’s perception changed – they want it to be beautiful, to change regularly and to be done… now!

In order to adapt to this new status quo, the developer needs to be agile, and cannot be alone. **Teams of developers work on a single software**, and change the codebase by the requirements, every week. As the software grows, the customers do the same, not only the **features need to get more mature, but the security and stability too**.

To achieve this agility, one must obey some **software design principles**, such as **strict object oriented** (or other) paradigm, layers of **abstraction**, **code quality**, **version control** and so forth. While on first sight it might have been a time-consuming event and only slows down the initial development, **the benefits are later seen**, once the **software starts to grow exponentially**.

To try to walk in a **corporate-developer’s shoes**, you need to **imagine the project will grow indefinitely** and so does the team.

In the **steps below**, we will walk through an **application which allows users to register**, **login**, **see** and **edit** their profiles. All of that will be **well abstracted**, without using existing architectures or frameworks. This will be skipped for the next course.

## What we will achieve?

|  |  |
| --- | --- |
| Home Page | Register Page |
| Login Page | Profile (edit) Page |
| Users List Page | Error Page |

## Architecture

The main inspiration in our future architecture is one of the fundamental principles in (object oriented) software development – **single responsibility**. One module (class, method, script, …) should do **exactly one thing**.

The next one we will follow is **abstraction**. In Object Oriented Programming it is mainly achieved by using **interfaces**. So, we will depend only on abstract operations and not on concrete object with their state.

In order to achieve well segregated application, we need **multiple layers**. In our multitier architecture we will have the following layers:

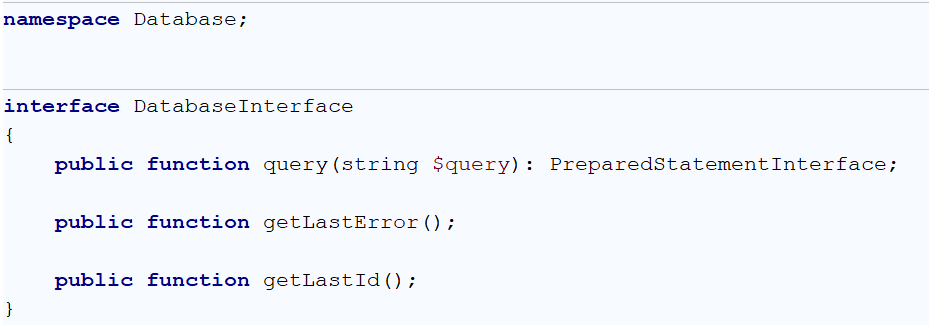
|  |  |
| --- | --- |
| * **Front layer** – the closest to the user layer. It is the script that will be triggered from the user input (e.g. user.php, index.php and so forth): |  |
| * **HTTP Handling Layer** – the next layer, which the front layer will directly operate. This layer is responsible for 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: |  |

## Abstracting Database

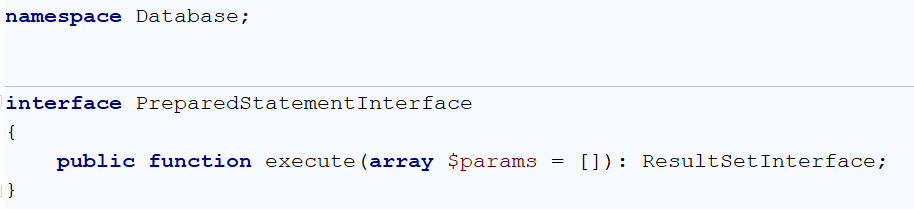
We will **not directly use the PDO object** in our object graph, because **of lack of abstraction** (PDO is concrete object).

As we know from PDO we need first to prepare the query, execute it and then fetch. These three steps will be abstracted in three different objects. Let’s create interfaces for all of them:

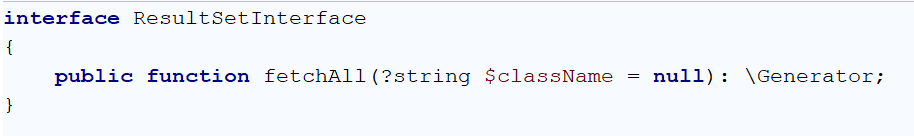
### Query Abstraction



### Prepared Statement Abstraction



### Fetching Abstraction

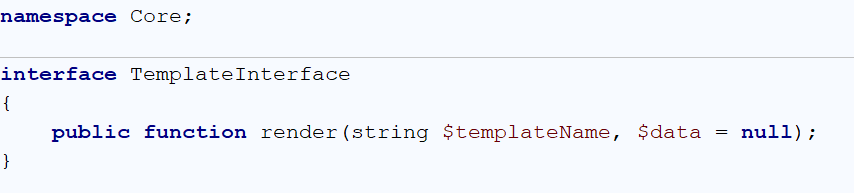


You need to create implementations that:

1. **Starts a query** and **returns a Statement** with the PDOStatement information
2. **Executes the statement** and returns **ResultSet** (the executed statement which can be **later fetched**)

## Templates Inclusion Abstraction

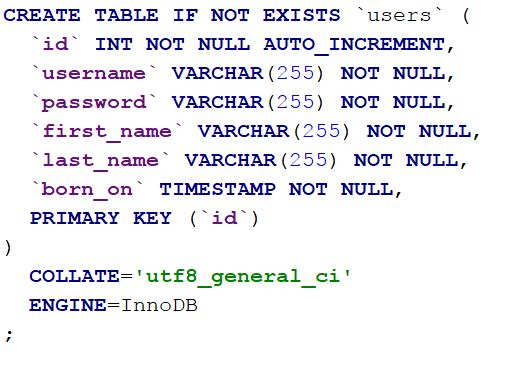
We will abstract the inclusion of HTMLs in a single method which accepts the name of the template and optionally a data which can be passed to the file. No more variables.



Create an implementation of the interface in a class named Template.

## Create Schema

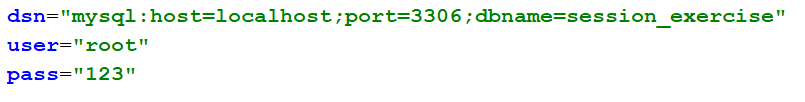
We will need database schema to create our application. Start with table users, that have username, password, first and last name, also a birthday. Add the CREATE TABLE script in a file named export.sql:



## Configure Database

We need our app to know the connection information. Let’s store it in an .ini configuration file (remember how php’s configuration file was named php.ini?). Create a folder “Config” and put a file “db.ini”:

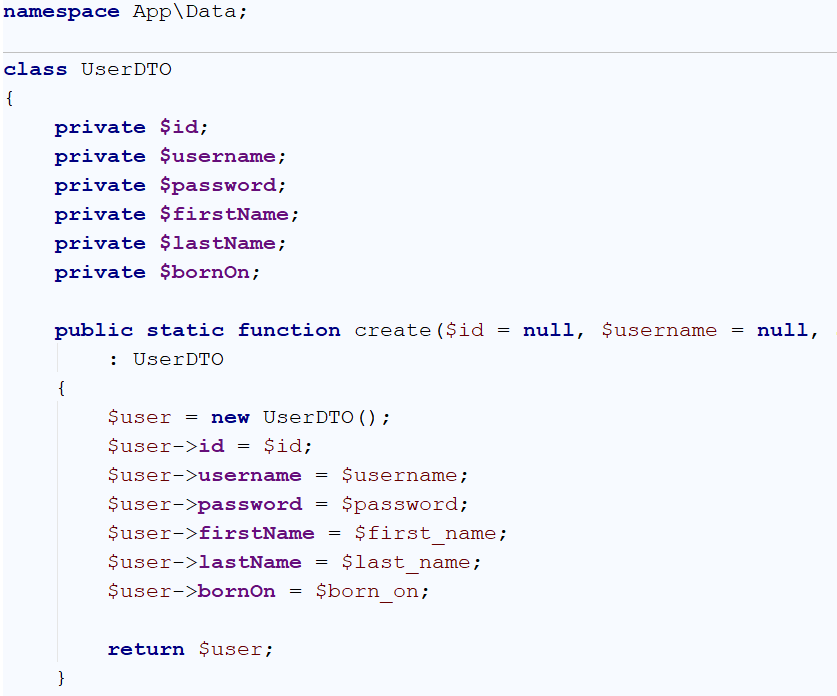




## Repository Layer

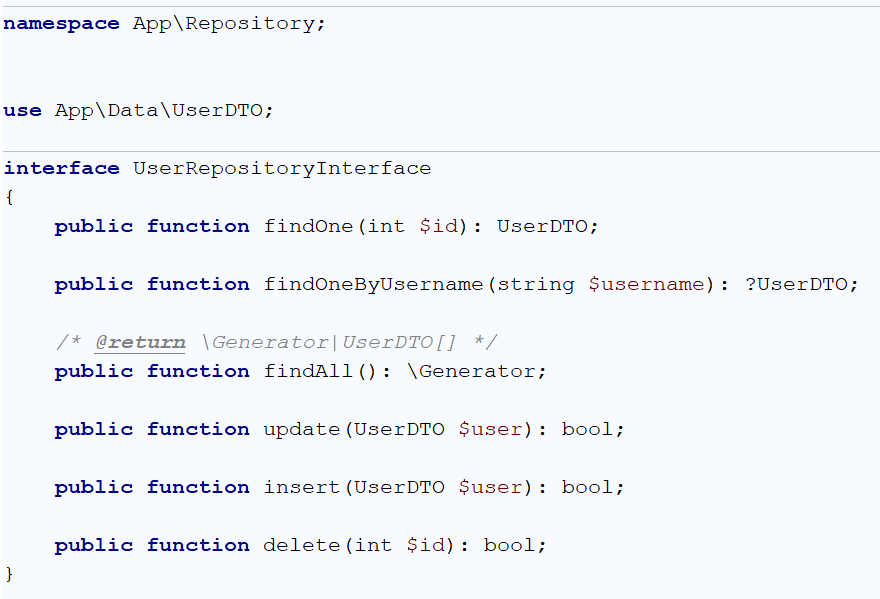
We will operate with **data only for users**, so we will need repository responsible for the users. It will fetch the data in a single data transfer object (DTO) called **UserDTO** which will have the **fields corresponding the columns** from the query.

Let’s create in folder named App, then Data a class UserDTO. Because DTOs in PDO cannot have a constructor, we will add a **single static** create() **method** which returns a populated user for manual usages later:



Create also the corresponding **getters** and **setters**.

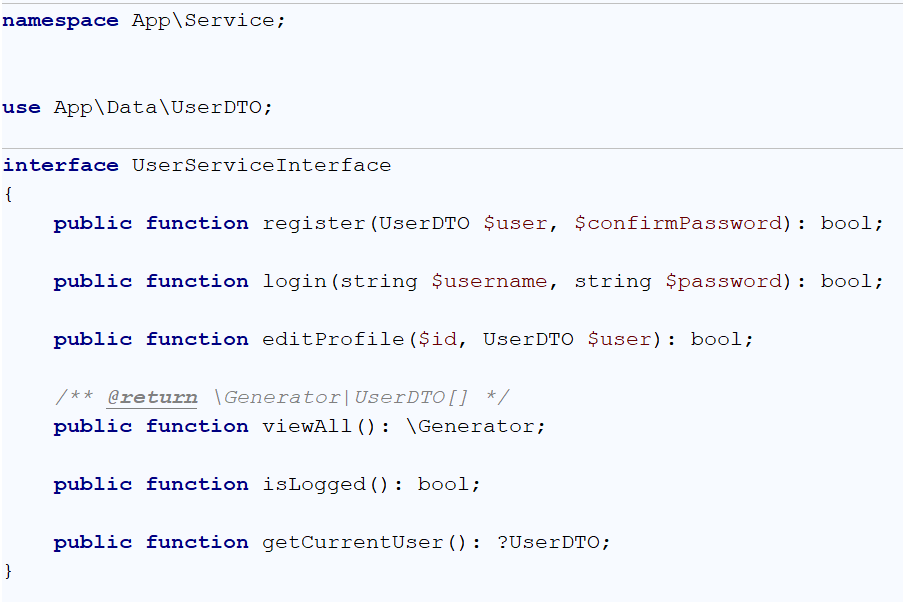
After the DTO we will need the actual repository:



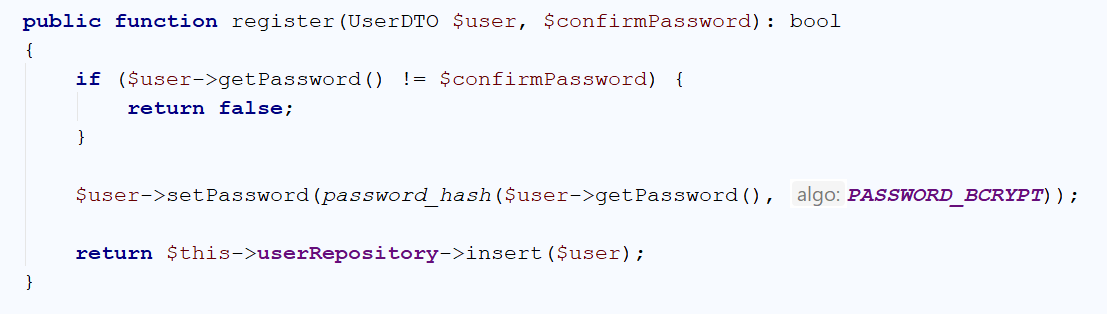
The implementation of this interface will depend on **DatabaseInterface** in order to execute queries. Create this implementation.

## Service Layer

We will have business operations around users as well. It will highly rely on the above repository, but it will **check things** like passwords matching, encrypting them and so forth:

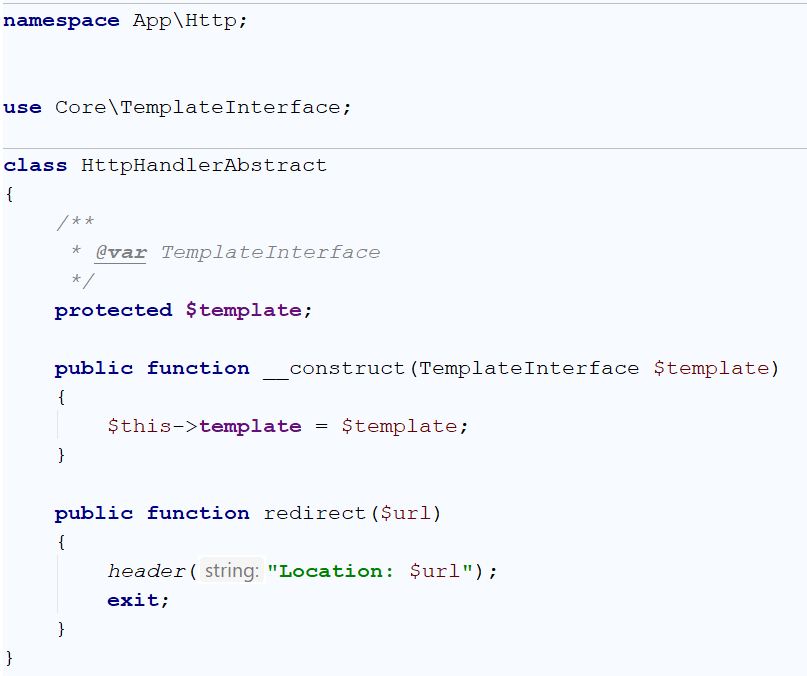


Create implementation of the above interface. An example implementation of the register() method is:



## Http Handlers

Each of our handlers will either include HTML’s or redirect. Let’s create an **abstract http handler** class which needs the **TemplateInterface** and have a function for redirection to some file of the front layer:



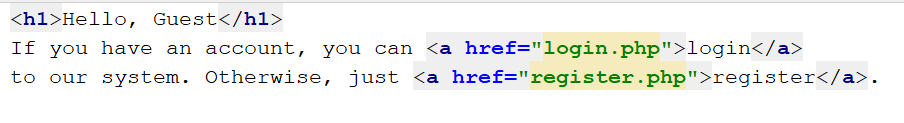
### Index Page

To render our index page, we need to include either the **profile page** (if the user is logged in) or the **home page** (If the user is guest). Following are both HTMLs for profile page and guest page:

Profile HTML located in App/Templates/user/profile.php



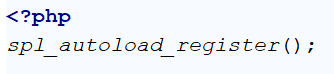
Home HTML located in App/Templates/home/index.php



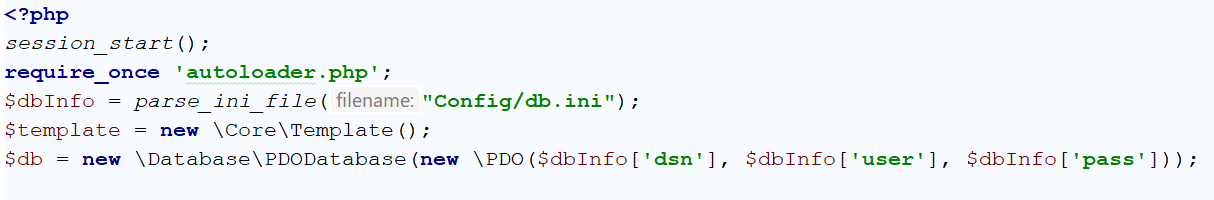
### Rendering decision

First of all, to access the home page, our users will hit **index.php** in the top level directory. This index, needs to prepare all the data for the repositories, services and http handlers. Most of this data will be common things, like autoloading classes, getting database configuration and so forth. In separate files, we will distribute the common configuration:

**autoloader.php**:



**Common.php**:



Before proceeding to index.php, we need to create our **Home handler** to decide which of them to render. Create a class **HomeHttpHandler** in App\Http



Because this handler needs to know whether the user is logged in, it has to depend on the abstract function isLogged() in the **UserServiceInterface**. And because the profile HTML needs an **$data** which is **UserDTO object**, the home handler has to send the second argument with such object. This object is the currently logged user, so we use one more time the service.

Now we are ready to call the handler in **index.php**:

