# Lab Exercise: Creating a Blog with PHP and MySQL

This document holds a step-by-step tutorial on **how to create a Blog system** based on **PHP**, **Apache** and **MySQL** with a simple custom **MVC framework**. This lab exercise is part of the [“Software Technologies” course @ SoftUni](https://softuni.bg/courses/software-technologies).

## Project Specification

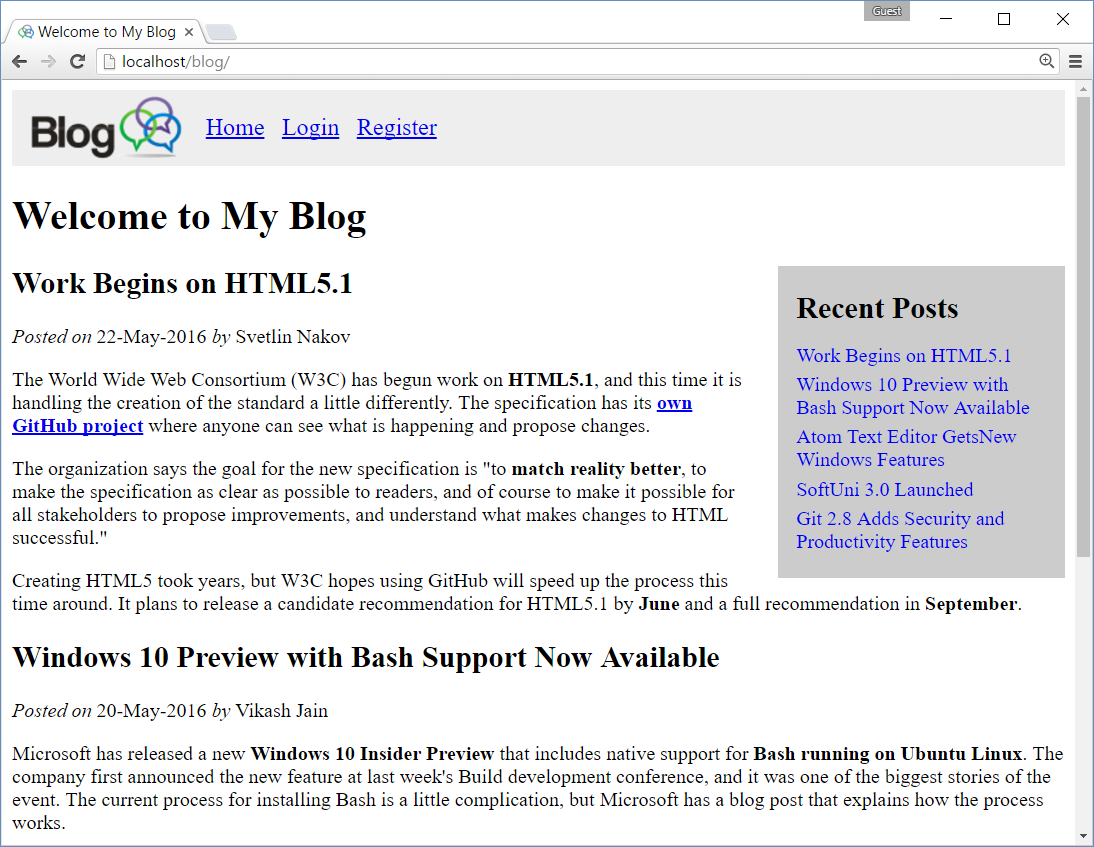
Design and implement a **“Blog” Web application** in PHP + MySQL. Implement the following functionality:

* **Home**
  + Show the **last 3 posts** at the home page, ordered by date (from the most recent).
  + Show also the **last 5 post titles** at the home page (as a **sidebar**) with a link to the post.
  + Show **[Login]** and **[Register]** buttons (when no user is logged in).
* **Login**
  + Login in the blog with existing account (username + password).
  + Show a success message after login or error message in case of problem.
* **Register**
  + Register a new user in the MySQL database (by username + password + full name).
  + Show a success message after registration or error message in case of problem.
* **Logout**
  + Logout the current user.
  + This [Logout] button is available after successful login only.
* **View / Create / Edit / Delete Posts (CRUD Operations)**
  + Logged in users should be able to **view** all posts, **create** new post (by title + content) / **edit** / **delete** their own posts.
  + Posts are **displayed in a table** (one row for each post). At each row a link **[Edit]** and **[Delete]** should be displayed.
  + **Create post** shows a **form** to enter the post data (title + content). After the form submission, the post is created in the database. Implement field validation (non-empty fields are required).
  + **Edit post** fills an existing post data in a form and allows it to be edited. After successful form submission, the post is edited. Implement field validation.
  + **Delete post** shows the post to be deleted and asks for confirmation.
* **View All Users**
  + Logged in users should be able to **view** all users (username + full name) in a table.

## Screenshots

This is how your HTML pages may look like.

The **home page** looks like this:



The **login form** looks like this:



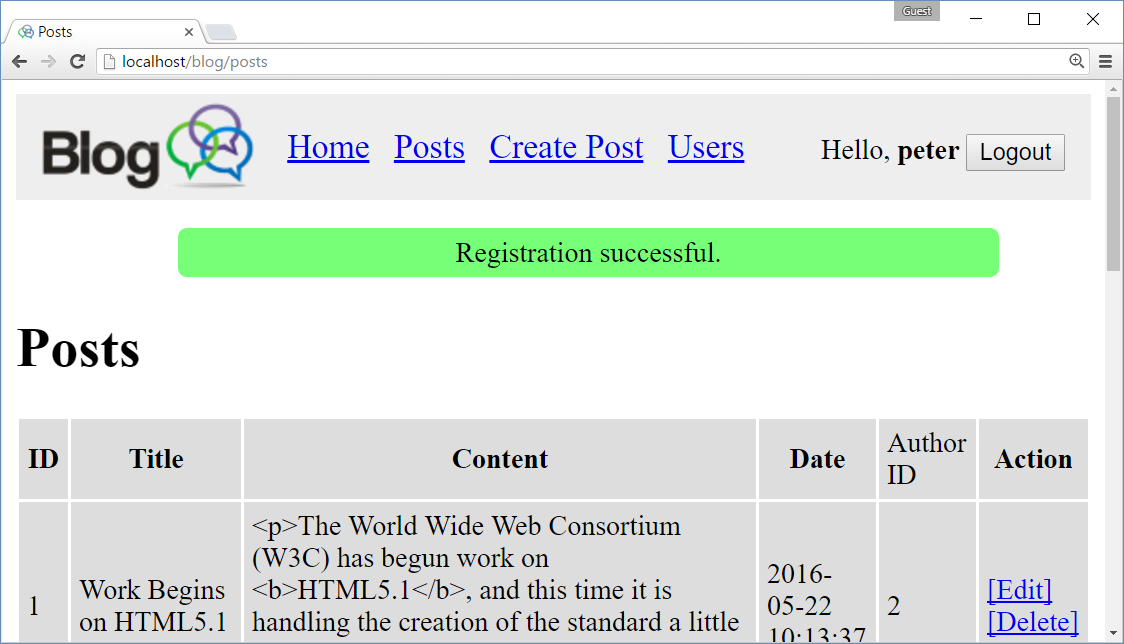
Trying to **login with invalid user**:



Form for **registering new user**:



After **successful user registration**:



Trying to **register with existing username** shows an error:

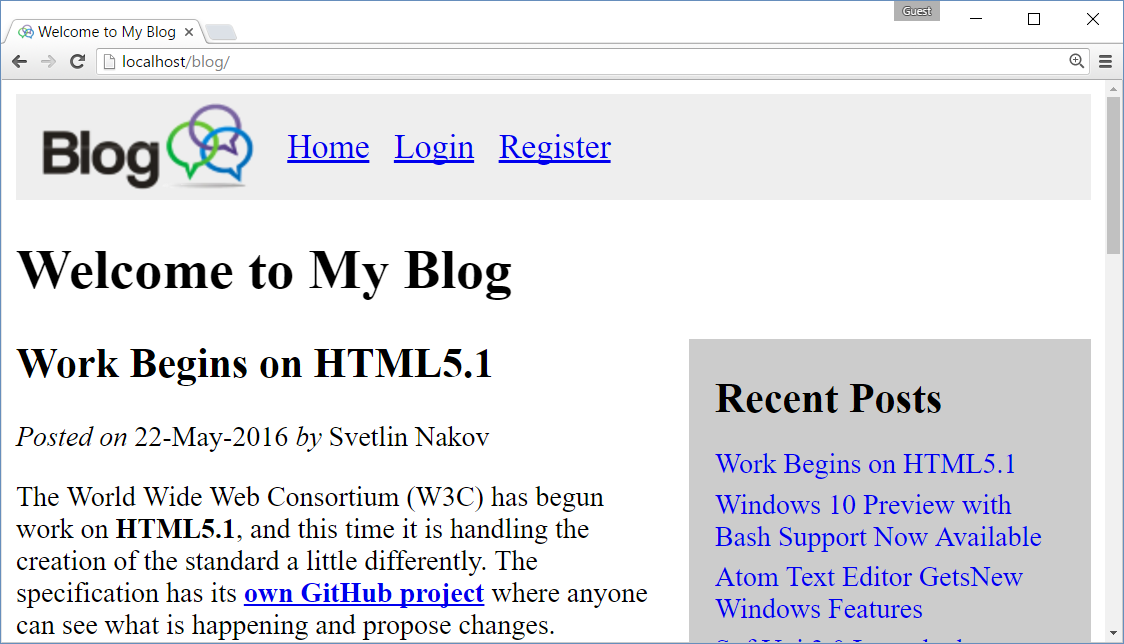


After successful **login**, redirect to the **posts page**:

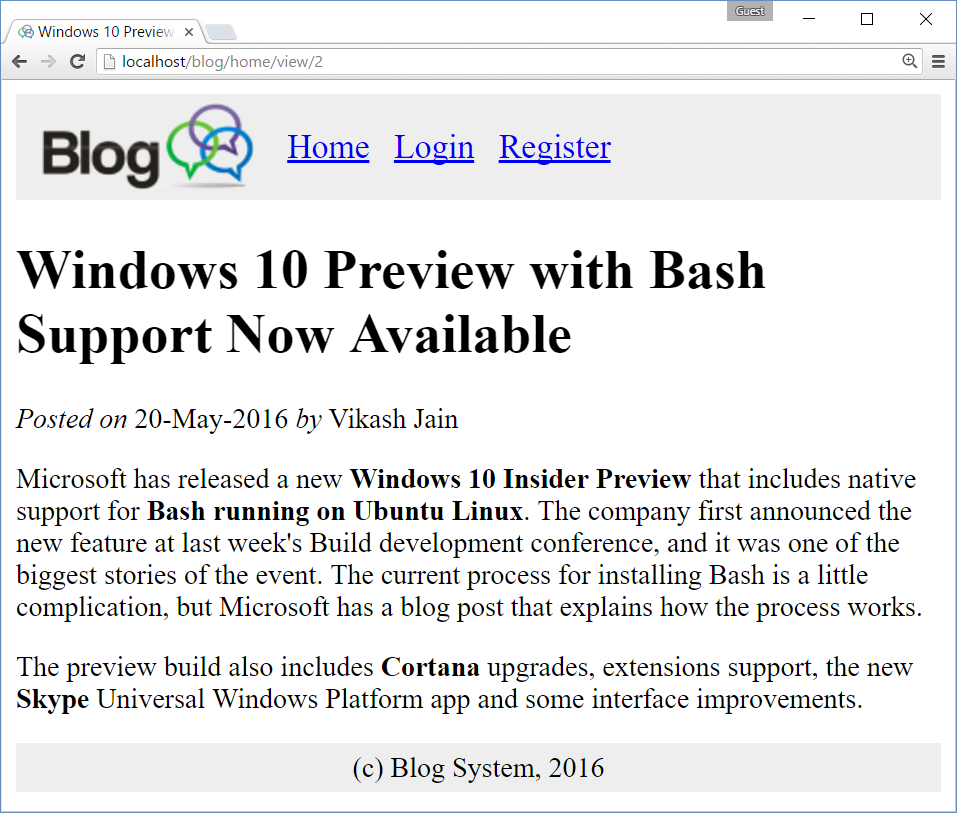




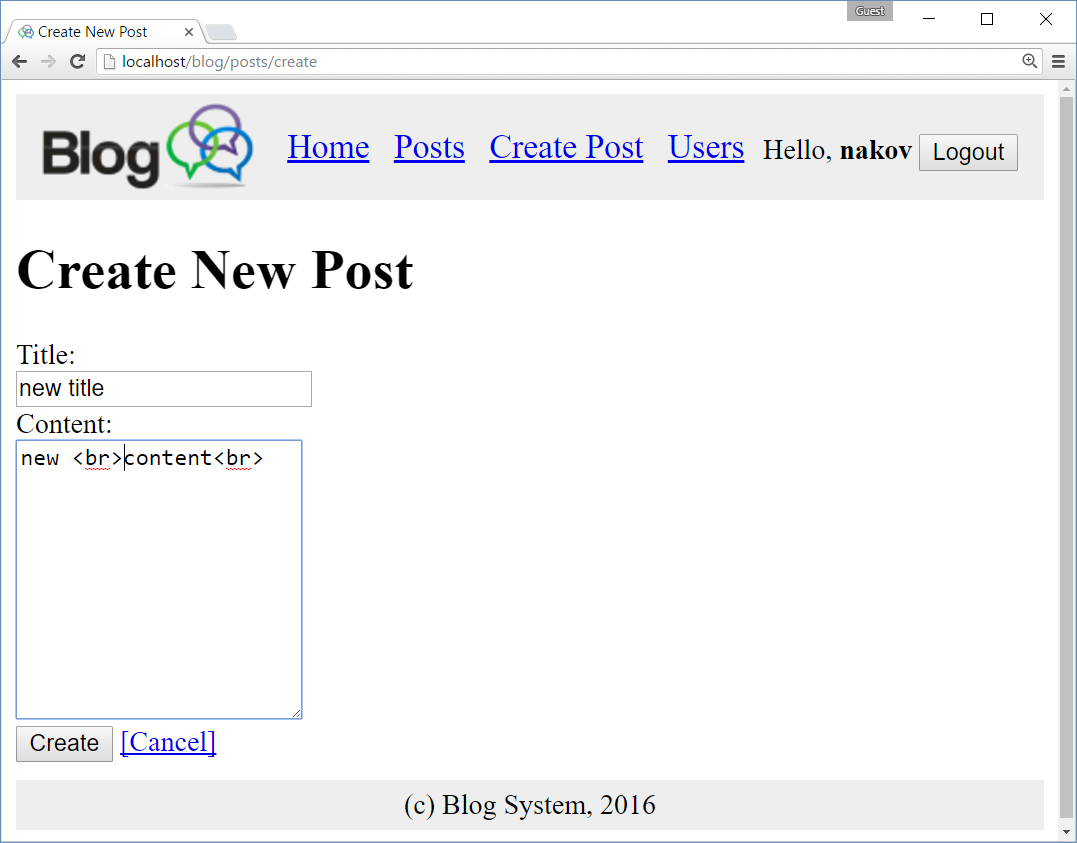
Clicking on **logout** shows the anonymous home page:

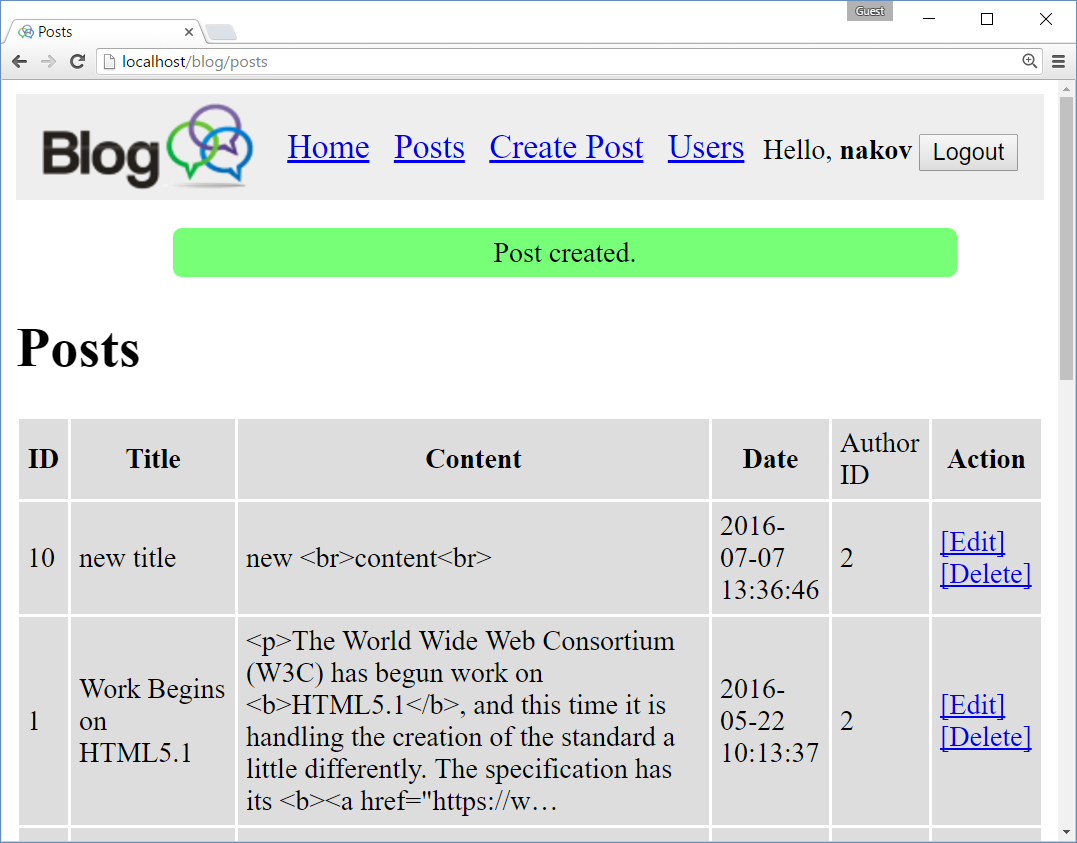


Clicking on a links from the “**Recent Posts**” shows the **blog publication** in separate page:



**Create** new blog post:



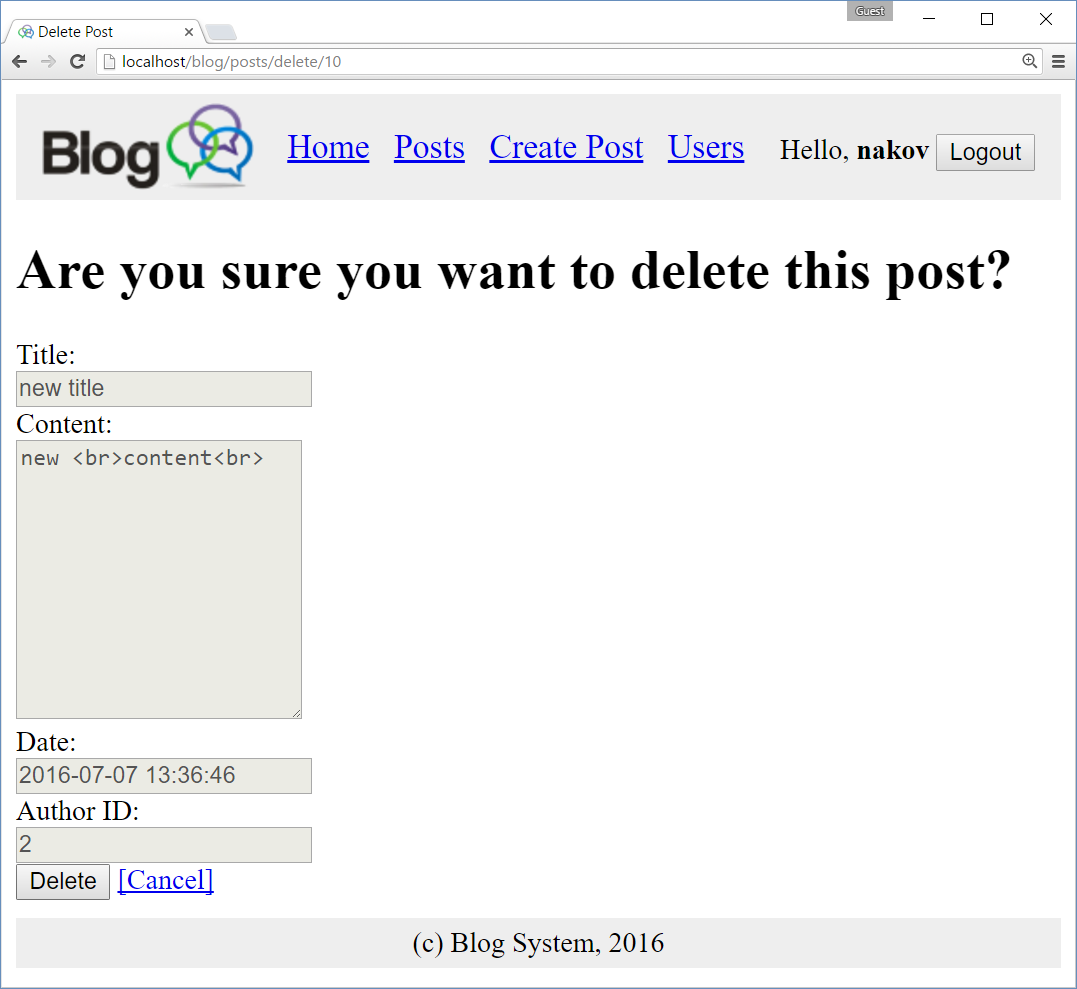


**Editing** an existing post:





**Deleting** a post:

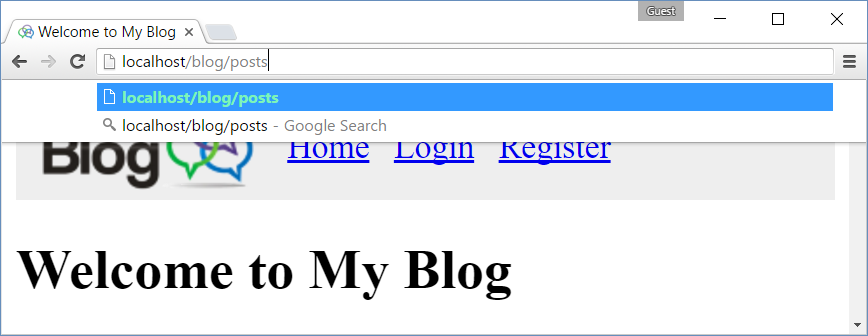




View all existing **users**:



Trying to access the posts / users pages after logout (without logged in user) causes **“Authentication” error**:

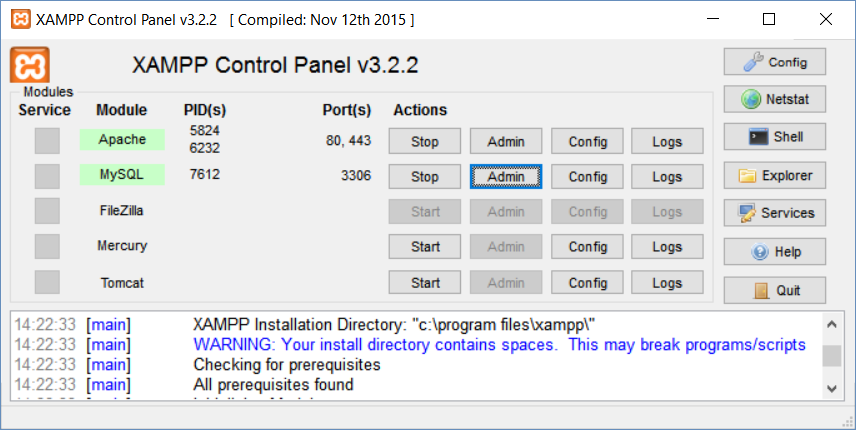




## Implementing the Blog from Scratch

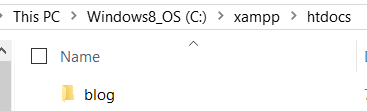
Let’s implement the blog system step by step, starting from scratch.

### Start XAMPP, Apache and MySQL



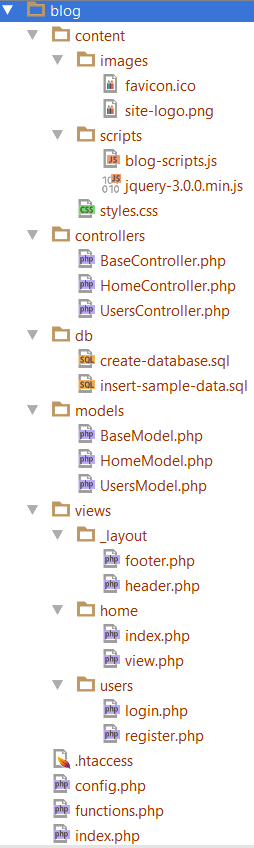
### Blog Project Directory

Create “blog” directory in the XAMPP htdocs, e.g. c:\xampp\htdocs\blog.



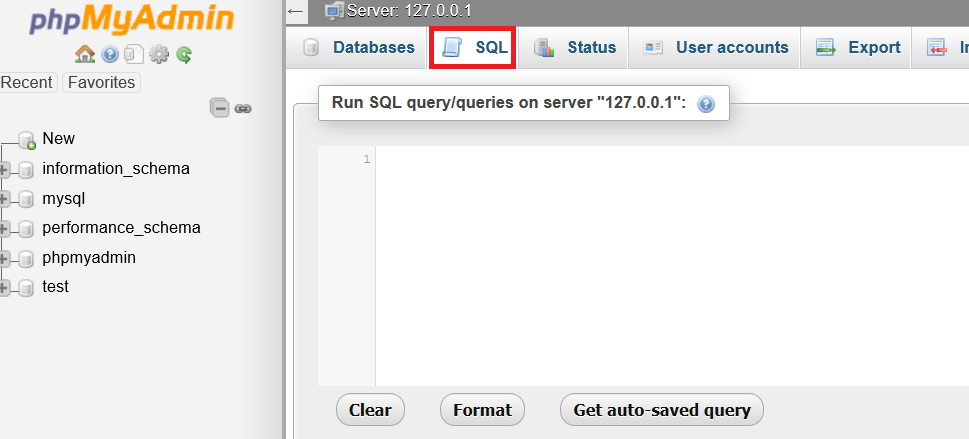
### Project Skeleton

Create (or download and unzip) the project skeleton, consisting of the following files and directories:

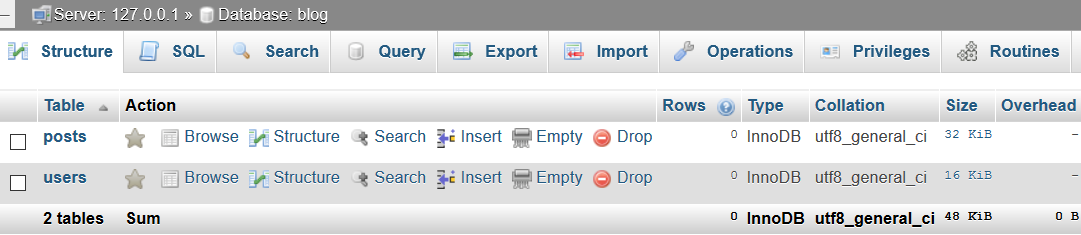


### Create Database

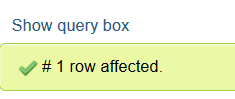
Go to phpMyAdmin and go to the ‘SQL’ tab.



Copy the script from ‘db/create-database.sql’ and paste it to the text field. Click on ‘Go’ and you should receive this database when you go to the ‘databases’ tab.



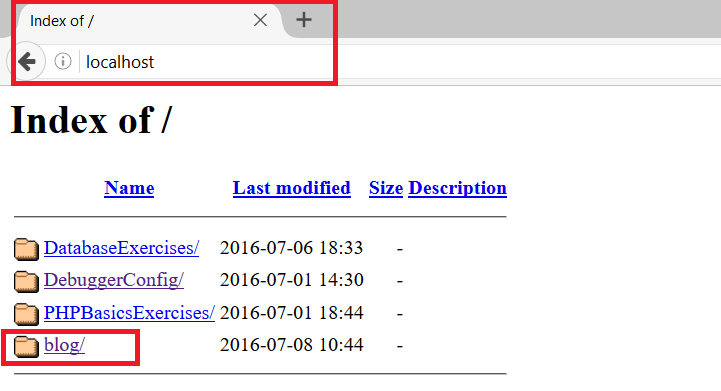
Go back to the ‘SQL’ tab and paste the other script from the db folder. When you execute the script, you should see this notification:



If everything worked correctly, we will have fully operational database.

### Run the Project for the First Time

Go to localhost in your browser and choose the blog folder.

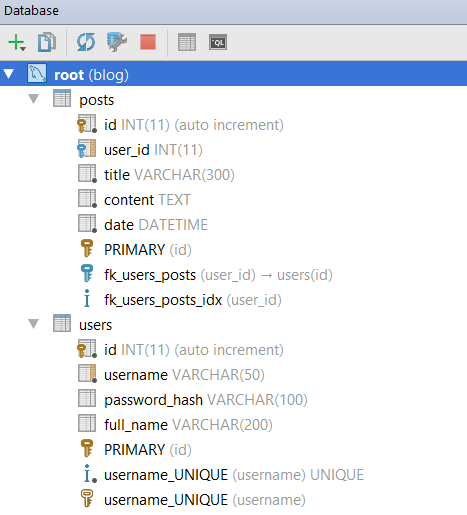


You should see this screen:

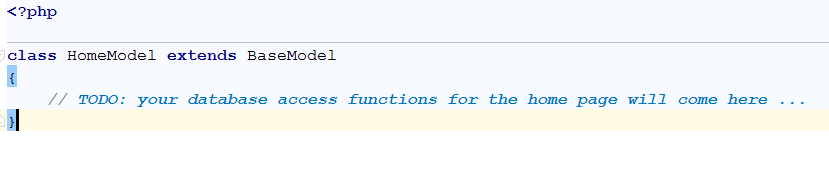


### List Posts at the Home Page

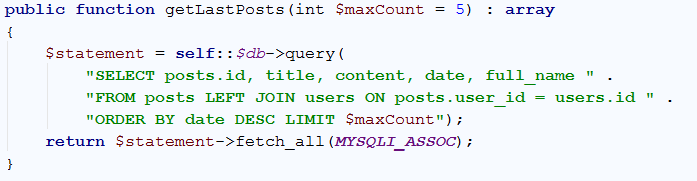
First we need to connect the database with PhpStorm. If you don’t know how to do it, check the previous lab “MySQL for PHP devs”. When we connect the database, we should see this:



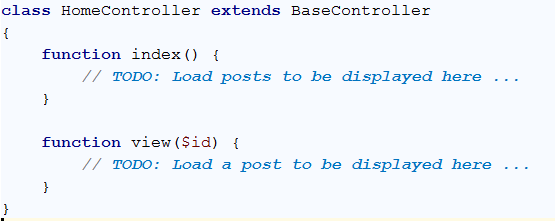
Let’s go to the **Models/HomeModel.php** file. Right now we have this:



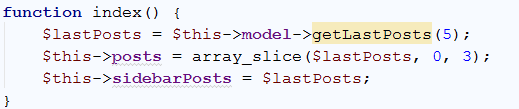
Let’s write a simple function:



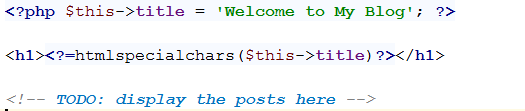
This function creates a SELECT query, that should give us the last $maxCount posts. We should call this function now. In order to do that, go to **controllers/HomeController.php**. You should see the following:



Modify the index function like this:



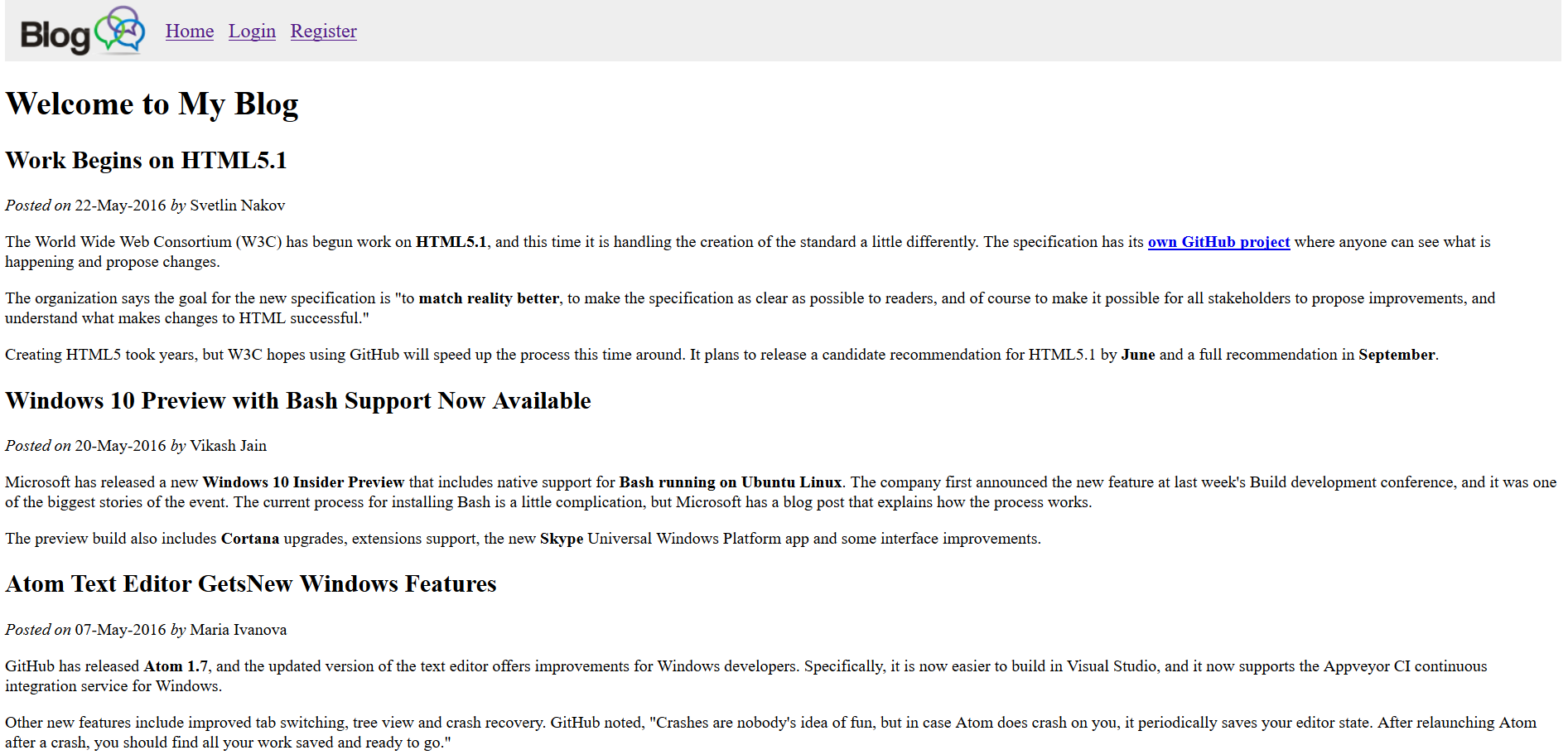
In order to see if it works, we should go to **views/home/index.php**. We have this at the moment:



We should write a simple foreach loop, that traverses our query result:



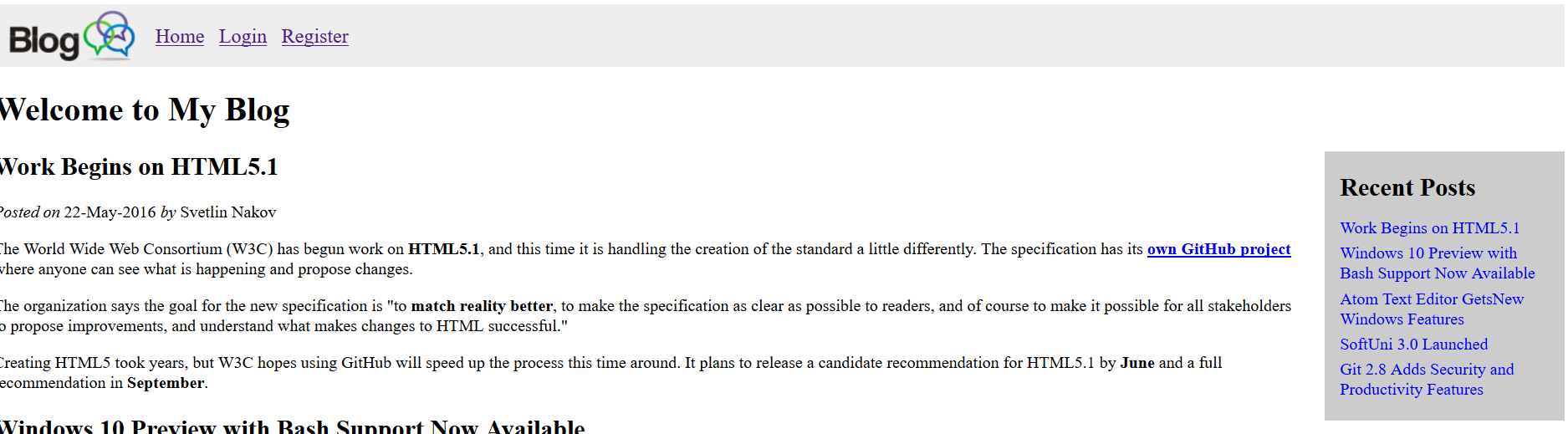
If we go to our blog home page we should see this now:



They are more posts, but we want to list only the last 3. However, we are missing the little sidebar on the right. In order to print it, we must go back to our **home/index.php** file and write another foreach loop above the previous one:

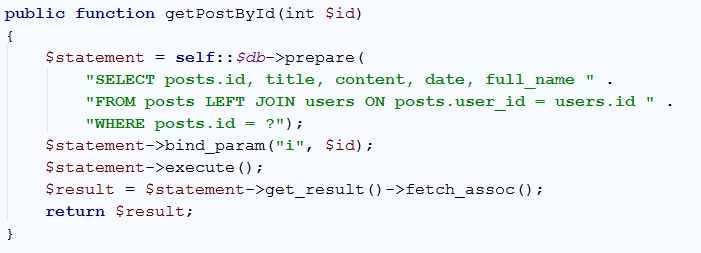


Refresh the home page in your browser, and see if you got this:

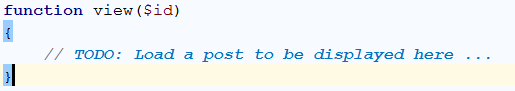


### List Single Post

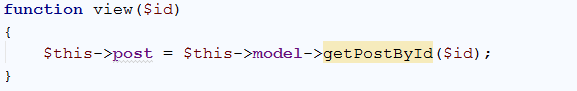
First we need to go back to our **HomeModel**, in which we should create one more function:



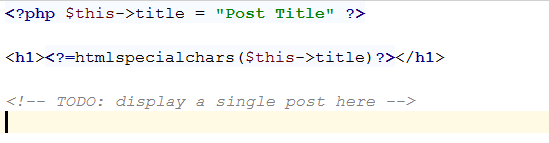
This function will return the information for a single post, that we select by id. We need to call that function, which we are going to do in the HomeController. We still have the empty view() function:



Write the following code:



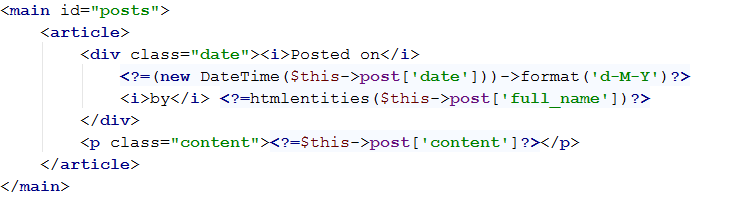
We are done with the HomeModel and HomeController. Now we should go to the **home/view.php** file. It contains the following code:



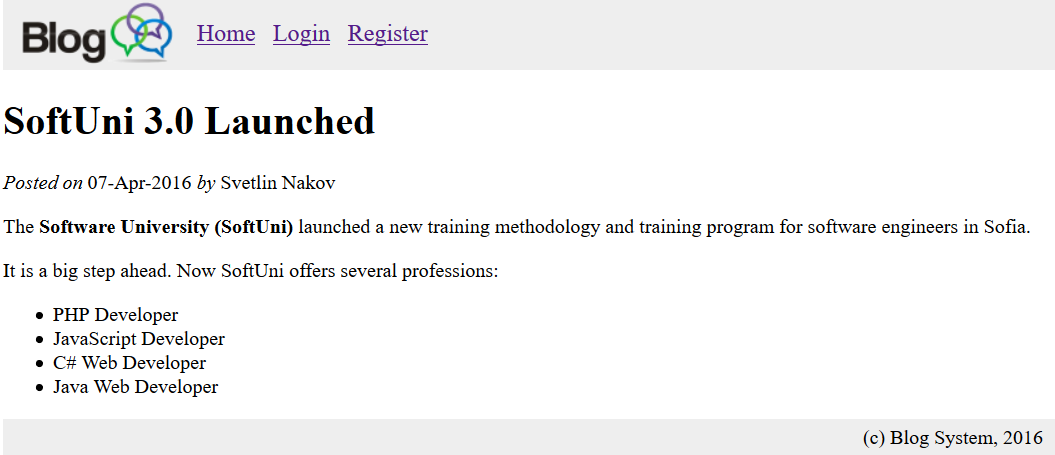
Replace the first line with this one:



Add this code in order to list the information about a single post:

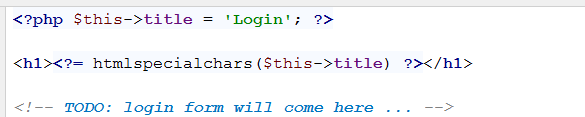


Now if click on a link in the sidebar, you will be redirected to a single post page, which should look like this:



### User Login

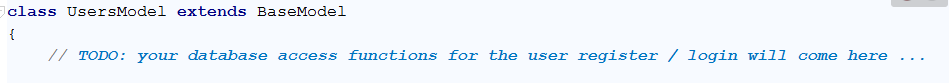
Now we need to create the login page. We start from the login form. Go to the **views/users/login.php**. It should contain the following:



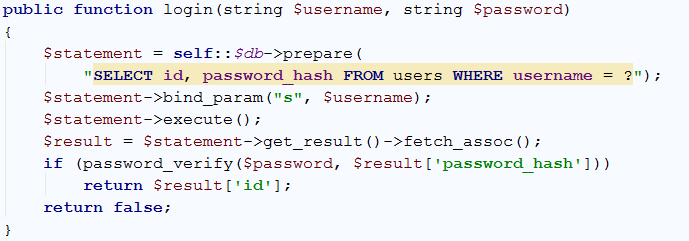
Write a simple form, that has 2 textboxes and a button:



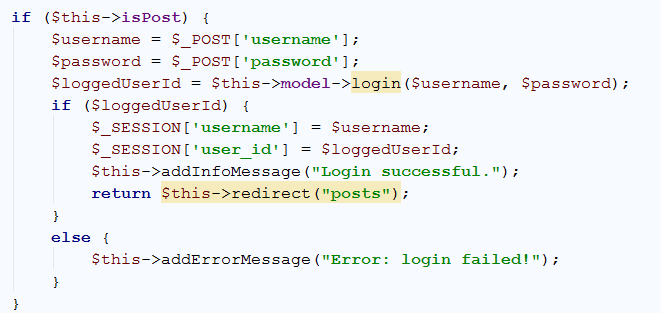
Now we need to create the logic behind the login. Let’s start by going to the UsersModel. Right now we have this:



Write the following function, that checks if the user exists:



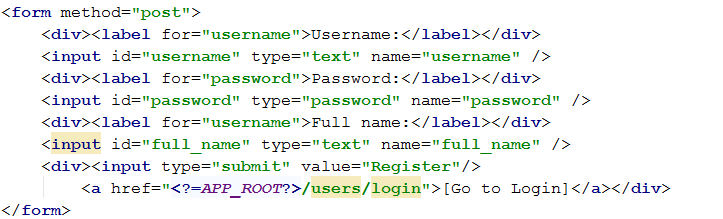
Finally, we need to call that function from the UsersController. In the empty login function we need to write the following code:



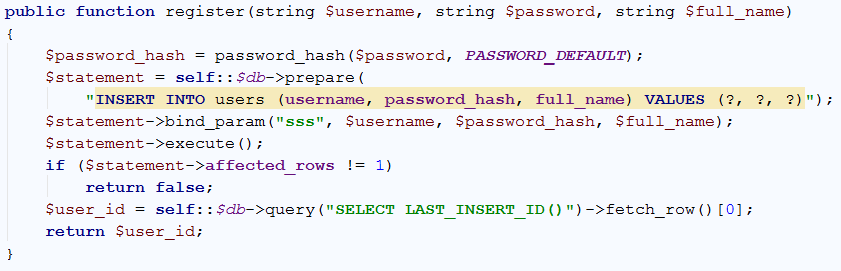
The login form should work now. It will redirect you to an error message when you successfully login, but if go back to the home page, you will see the success message.

### User Registration

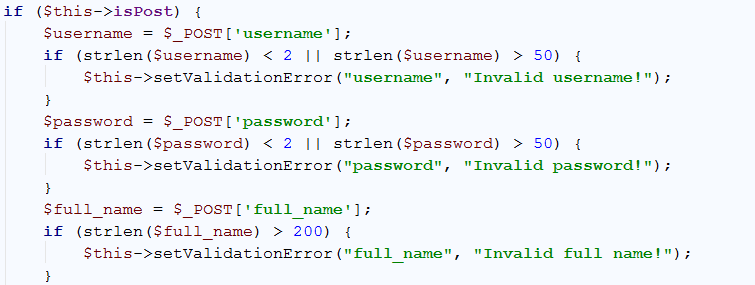
Right now we can login, but we cannot register. We should fix this issue. Let’s start by creating the form in **users/register.php**. Creating a simple form like this will work like a charm:



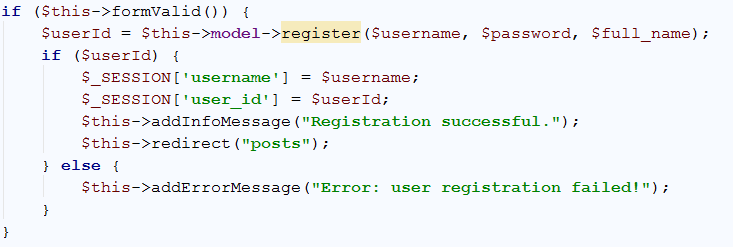
We should go to the UsersModel and create another function, that will create a query that inserts new row to our users table:



Now that we have the model and the view, the only thing left is to tell the controller what to do. Go to the UsersController and add that code to the empty **register** **function**:



This code validates our input, and if the input is invalid, throws an error message. After the validations, we should write the following code:



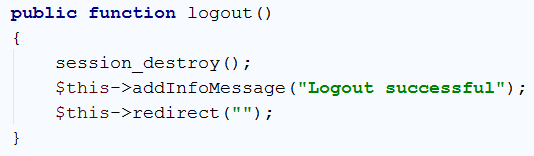
Our registration is ready now. After successful registration this error might come up, but don’t worry, we will fix it soon.



Just go to the home page, and you should be logged in.

### User Logout

Aside from Registration and Login logic, we should provide the user with the ability to log out and use a different username. Go to logout() function in the UsersController, and write the following code.



Essentially, the “**login**” logic is done by saving data about the user, which is being logged in, in the **current session**. That data is the only thing that indicates that there is a currently logged in user. If we are to delete that data, there will practically be no logged in user. For that reason, we just clear the session, with the session\_destroy() function, which works as the main “**logout**” logic.

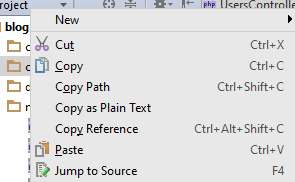
After that, we just print an information message, and we redirect the page to the home.

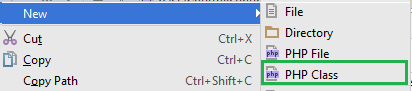
### Implementing the Main Posts Logic

Now that we are done with the main authorization elements in the blog, we can concentrate on the logic around the blog posts. In order for all actions concentrated around the posts to function normally, you’ll need to integrate the posts logic with the other logic. That means that the posts need their own Model, Views and Controller.

Let’s prepare those 3 elements, and integrate them with all the others.

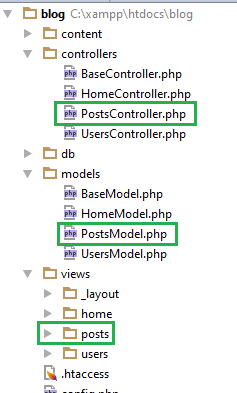
Create a PHP class named PostsController.php, in the controllers folder. Creating a class is done by right-clicking on the folder, in which you want to add the class, selecting **[New]** and then **[PHP Class]**.

e



Follow the same steps and create a PHP Class named PostsModel.php in the models folder.

And lastly, we need the **Views**. Create a folder – posts, in the views folder, to hold our posts views. Your structure should look like this, if you’ve done everything correctly.



There are several main actions which work with the blog’s posts.

* Listing all the posts in a table – this is done on the main index.php page.
* Creating a new post with given data
* Editing a post, with new data, by given post id
* Deleting a post, by given post id

We must implement all of these actions, using the files and folders, we’ve just created. Let’s start with the PostsController.

### Implementing the Posts Controller

Go to the PostsController.php file and initialize the class so that it extends the base class – BaseController.php.



We are performing a simple Inheritance – The class PostsController is extending (inheriting) the class BaseController. Now that we have the successfully created class – PostsController, we can fill it with functions, so that each function corresponds to one of the actions, which work with the posts, as we specified above.



As you see we have functions for every action we specified above, but we also have an additional function – onInit(). This function will be executed when the class is instantiated. It is a function that initializes base data and operations, which are needed for the class to function normally.

And with this, we have the base structure of our PostsController. We can start filling the functions, but first, we must implement the models and views too. Let’s start with the Model.

### Implementing the Posts Model

First, initialize the class so that it extends the BaseModel class.



And now let’s implement the main functions in the Posts Model.



We would need to get all posts from the Database in order to print them on the index.php page. We would need every post from the Database to be accessible via **id**. Given that, we have created 2 functions which do that for us. The Create, Edit and Delete functions remain with the same context.

Now that we have even the Posts Model, the only thing left to do is the Views.

### Implementing the Post Views

Just the Controller and the Model aren’t enough for our blog posts. We need Views to present the data to the client. We already have a folder posts in the views folder, but it is empty. Let’s add a few views to it.

Create the following files, in the posts folder, with exactly the same names as they are in the picture.



And this should be all about the basic structure of the posts. Now we can start implementing the actions around the posts, one by one.

### Lists Posts in a Table

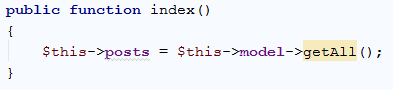
Now that we have our Posts Controller, Posts Model and Posts Views, we can implement the different actions with the posts. The first action is – listing all the posts in a table. This is done on the index.php page.

Go to the PostsController class. Before implementing the logic about the post listing, we must first implement the onInit() function in the controller, so that we know that our class is fully functional when instantiated.



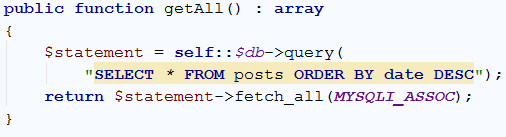
The authorize() function comes from the base controller class, and is used to authorize the client as a logged in user.

Now we can continue with the listing of the posts. This logic is controlled by the index() function in the PostsController class.



This will use the current controller’s model (the **Posts Model**) to get all posts from the Database, and store them. However, we do not have such functionality in the Posts Model – therefore we need to implement it.

Go to the getAll() function, which we just used, in the PostsModel class and write the following:



The function returns an **array**, as specified in the function declaration. This array consists of all the posts, **fetched** from the Database using a **query**, and ordered in **descending** order by the **date** they were posted on.

With this we are ready with the functionality of extracting all the posts from the Database, but how will we present them to the client. This is where the View comes. Go to the index.php file in the posts views, and write the following:



We set the title of the page. It is good to do it this way, so that we are sure that it is a constant, not just some magic value. After that we create a simple table with a head row, which will hold the captions of our columns. The next step is filling the table. Write the following PHP script right after the head table row.



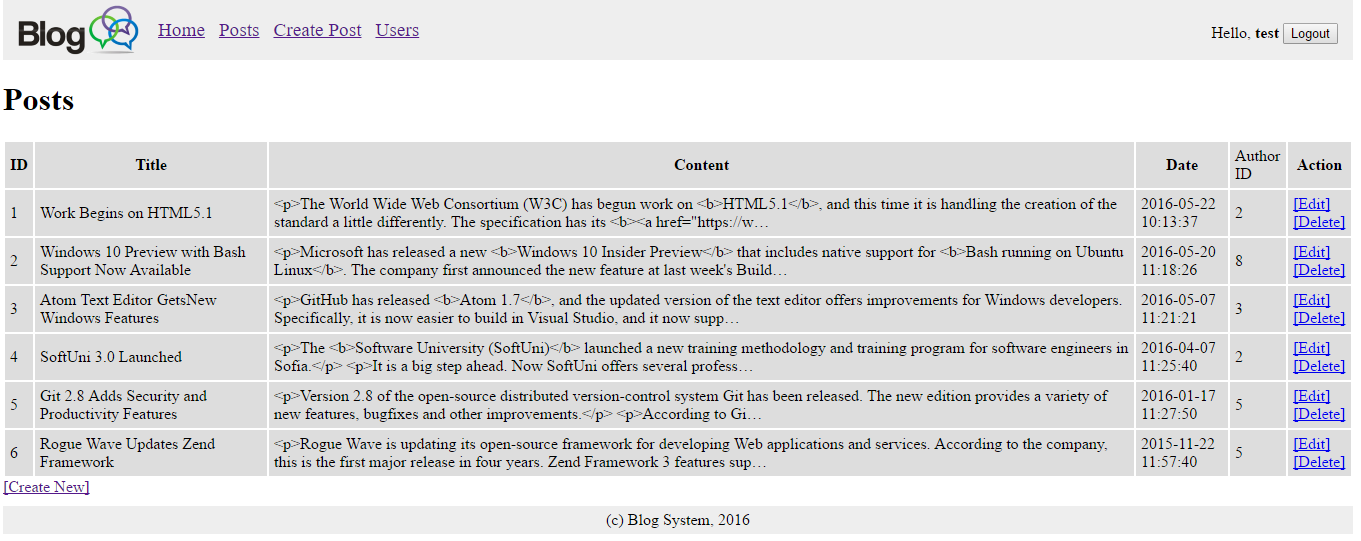
This is how a foreach loop is made in PHP scripts. The loop will traverse all the posts from the controller, which we previously extracted, and will print info about each of those posts, on each table row, separating the values into the columns of that row. Also, this will add two links, which will later point to the edit and delete functionalities. We will implement them later.

Let’s also give the user the ability to create a new post, on this page – just a simple link after the table.



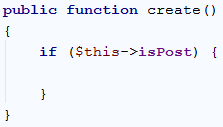
We will implement the functionality for this, in the next step.

If you’ve done everything correctly, when you click on the Posts link, you should see this on the posts page.

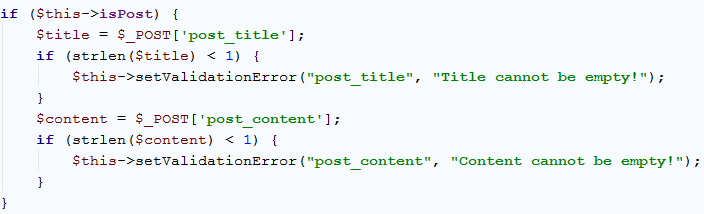


### Create New Post

Let’s implement the functionality for creating new posts. Go to the create() function and write the following:



This will check if the isPost field is set to true. This field is only set to true if there is a “POST” request to the server, and since our only “POST” in the Posts context is about creating posts, it is perfect for this function.



After that, we perform several simple checks to validate the given post data. If it is not valid, we simply notify the client with an error.

If there are no errors, we simply extract all the data, and we use the Post Model’s create() function to create a new post. Then we notify the client with an information message if the post creation succeeded or with an error message if the post creation failed. In case of **success**, we redirect the client to the posts page.

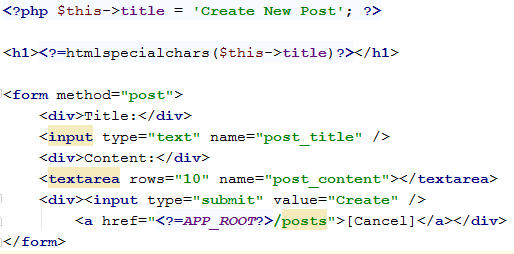


Now let’s implement the create post functionality in the PostModel class. Go to the create() function of the class and write the following:



This will create a query, which will insert a post into the posts table. The query will be applied the, given by the controller, parameters and will be executed, returning a Boolean result, indicating whether it succeeded or failed.

Lastly, we need to implement the Create Post View. Go to the create.php file in the posts views, and write the following:

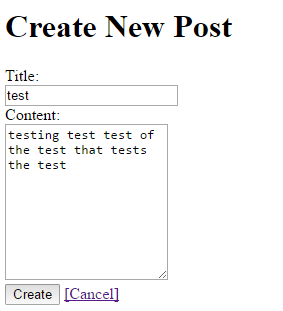


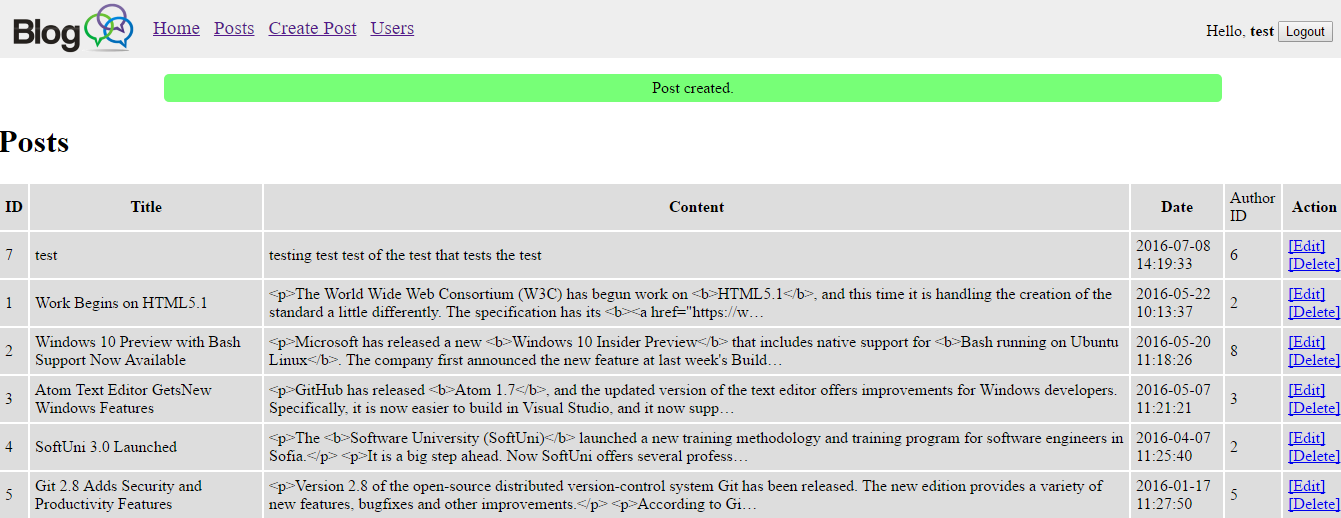
The logic with the constant title remains the same. The other is just a simple form, containing a submit request method, and two input fields – the title and the content of the newly created post. We also provide the user with the ability to cancel his action, and return to the posts page.

If you’ve done everything in the right way, you should see the following results, when trying to create a post:









### Delete Post

We have the functionality to create posts, now let us implement the logic to delete posts. Go to the delete() function of the PostsController class and write the following:

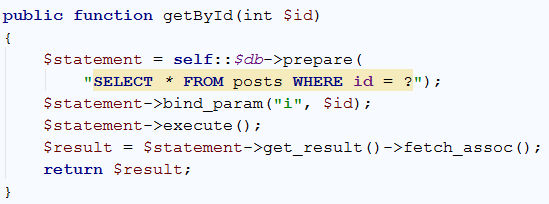


As you see we have an if / else statement above all other statements. This statement is put in order to first provide the client with a confirmation message of deletion, and only after that delete the given post. In the confirmation, the controller extracts the post, using the model, by the given id and questions the user. In the deletion phase the controller deletes the post, using the mode, by the given id.

In both cases, messages shown to notify the user of the status of his actions.

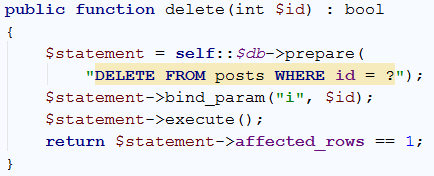
Now, however, we have two things to implement in the model – the getById() function and the delete() function. Go to the PostModel class.

First we will implement the getById() function.



It prepares a query, applies an integer id to it and extracts all posts from the database with that id, in our case it will get only 1 post, because we have unique ids. The result is fetched and returned.

And now the delete() function.



Same as the create() function it returns a Boolean result, indicating whether it has succeeded or failed.

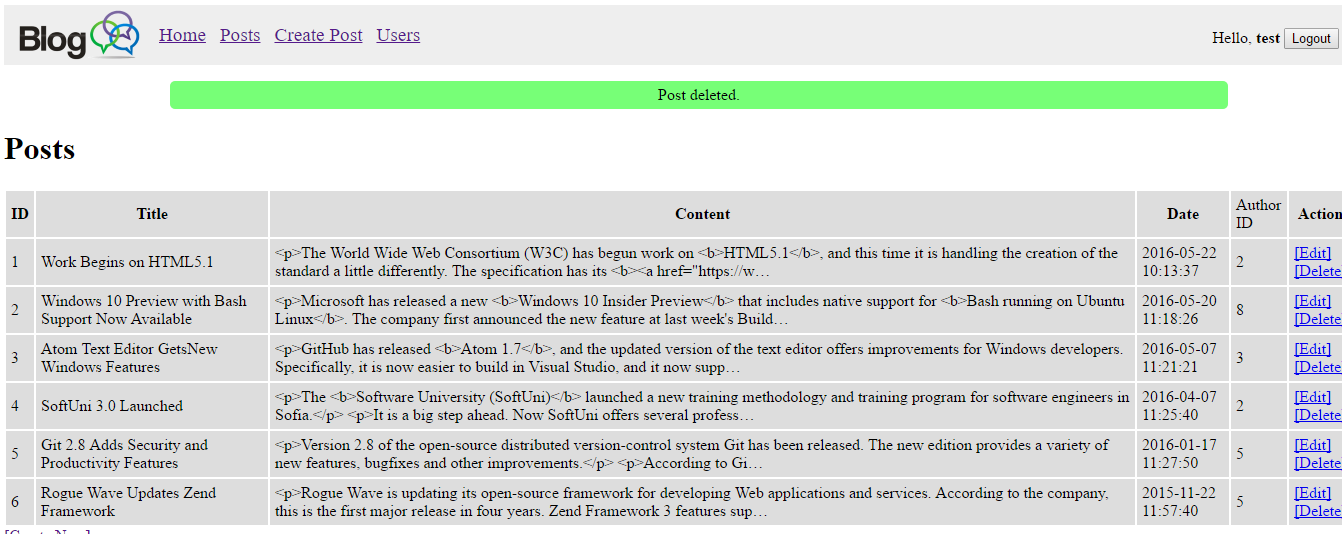
Now, go to the delete.php view in the posts folder in the views folder, and write the following in it:



As you’ve noticed most of the elements in this form are disabled, this is due to the fact that it is only a confirmation form – it will hold predefined values – the values of the post that is currently being deleted, and will confirm if the user is sure to delete exactly this post with this data. In case the data is submitted, the post will be immediately deleted.

If you’ve implemented everything correctly, you should see the following result when try to delete a post.

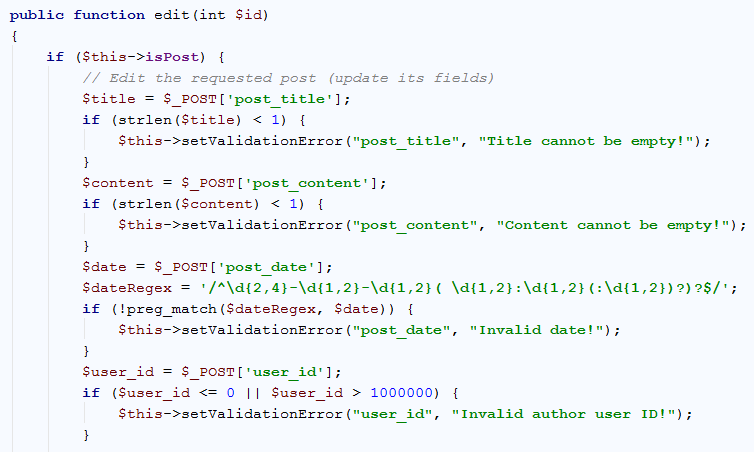




### Edit Existing Post

We have creation of posts and deletion of posts, but what if the user wants just a minor change on an existing post. Then we must provide the user with the functionality of editing posts.

Go to the edit() function of the PostsController class and write the following:

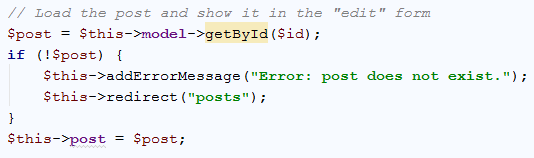


The same if which validates if the request method is “POST”, and several form value, validations in it. Notice how the date is validated using regular expressions (REGEX).

If all validations have succeeded we must edit the post, using the model, with the new data:

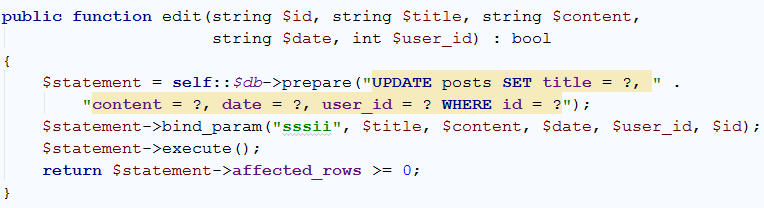


After the big if statement which checks if the request method, put the following:



Now, this will preload the post into the edit form, enabling us to edit it, and only when we submit the new values, the code in the if statement will execute. This should finalize our work in the PostController class.

Go to the PostModel class, to the edit() function, and write the following:



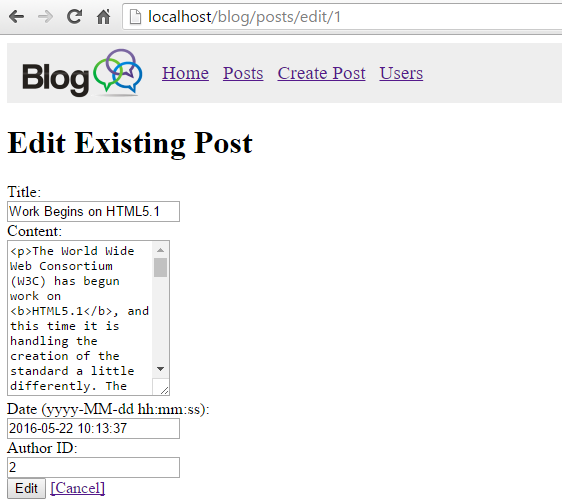
This will prepare a query, which updates the posts table, setting every title, content, date and user id, to a newly given value, but will apply the following changes only to those posts which correspond to the given id.

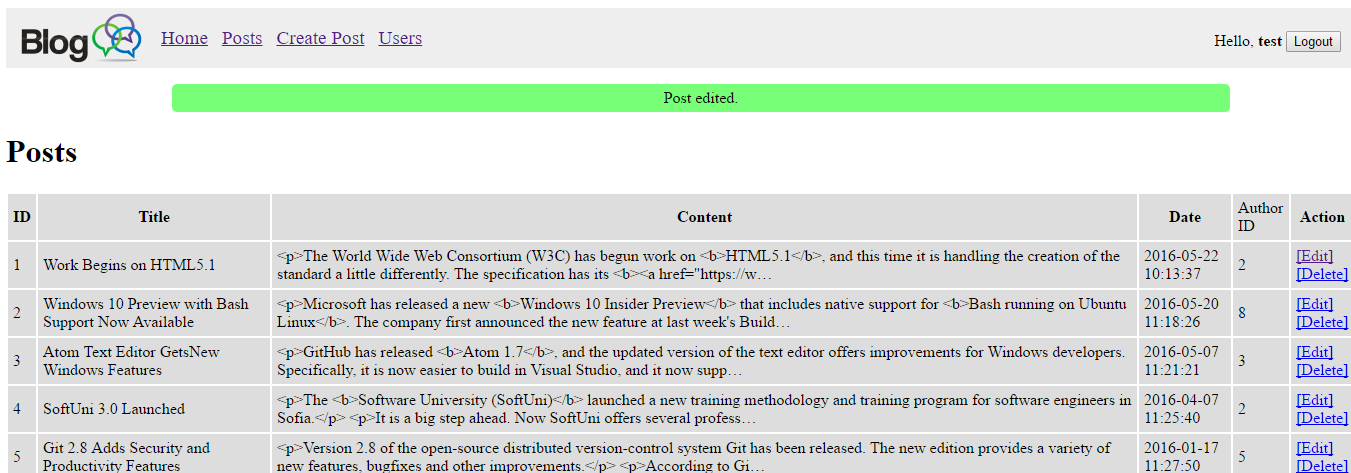
And with this we finalize our work in the PostsModel class too. The final piece of the posts logic is the final view. Go to edit.php file in the posts views and write the following:



The form is incredibly similar to the delete form, except this time everything is enabled and we can freely change it. When the edit form is submitted, it will override the old data in the Database with the newly given values.

If you’ve done everything correctly, you should see the following results:

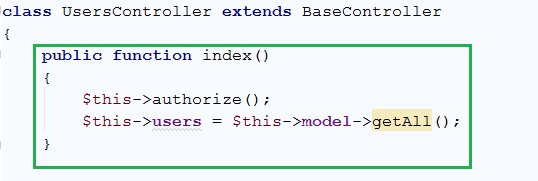




This is the final functionality we had to implement for the posts. With this we have almost finished our simple blog application.

### List Users

The last thing we need to do is list all the currently registered users.  
Let’s go to the UsersController class and add an index() function, which will implement the logic for the home users page.



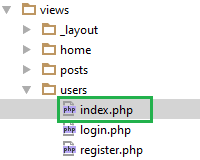
Similar to the posts, it just gets all the users from the Database, using the User Model, and stores them. It also authorizes the client, before all else.

Next, we go to the UsersModel class, because we need a function called getAll().

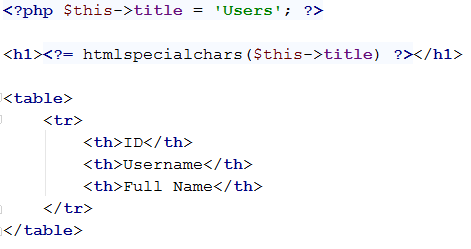


The function fetches all the users from the Database and orders them in alphabetical order.

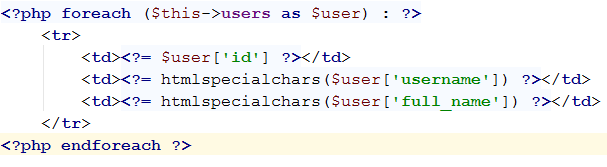
The last thing we need to do is to present the users data to the user. Go to the users views, and create a file called index.php.



Write the following in the file:



We create a simple table, same as the posts, in which we will present data about our users. After that we just make a foreach loop, to loop over all the users we extracted with the controller.



This was the last functionality we needed to implement. If you have done everything by the steps, you should see the following results:

