User Authentication in the Slim Framework

What we have done in class so far is great, but none of it matters if users have information that isn’t protected at least behind a username/password combination.

In this tutorial, we will be providing a small framework for you to get comfortable dealing with usernames and passwords and sending the user to different places depending on if their password is correct or not. So, let’s get started.

First, we need a test database to check user login information against. Log in to mySQL and make and shift focus to a new database called test-auth.

CREATE DATABASE test\_auth;

USE test\_auth.

Next, create a table to store user information. Each entry in the table should have a user\_id uniquely identifying it, a username, a password, and a number indicating the type of user (0 for basic privileges, 1 for moderator, 2 for administrator).

CREATE TABLE users(

user\_id CHAR(8) PRIMARY KEY,

username VARCHAR(50),

password VARCHAR(50),

type TINYINT);

Now, insert a test user into the table.

INSERT INTO users VALUES('12345678', 'fluffybunny1', 'junebug', 0);

We’d like to simulate a basic user’s access to the database, so log in to MySQL as root and type the following:

CREATE USER 'basic\_user'@'localhost' IDENTIFIED BY ‘<password>’;

This will create a MySQL user named “basic\_user” with “<password>” as their password. Users in MySQL allow for you to regulate who has access to what parts of your database. In production, you should store at the very least a hash of the password in the database, but for simplicity’s sake we will present a basic solution for you to build on.

GRANT SELECT ON user\_auth.\* to basic\_user;

This will grant only the permissions to view the database to the basic users. This is not required, but a good thing to know how to do and be comfortable with. In real practice a basic user shouldn’t be able to view the users and their passwords, but for these purposes they can. You can set up your own permissions as they pertain to your project.

The next step is to define connection settings for the database in /src/settings.php. In the associative array called 'settings', add a key value pair for the database.

'login\_dbConn' => [

'username' => '<YOUR\_MYSQL\_USERNAME>',

'password' => '<YOUR\_MYSQL\_PASSWORD>',

'host' => 'localhost',

'dbname' => 'test\_auth',

'db' => 'mysql'

]

Then, in /src/dependencies.php, define a new container for the database settings.

$container['login\_dbConn'] = function ($c) {

$settings = $c->get('settings')['login\_dbConn'];

$connString = $settings['db'] . ':host=' . $settings['host'];

$connString .= ';dbname=' . $settings['dbname'] . ';charset=utf8mb4';

$db = new PDO($connString, $settings['username'], $settings['password']);

return $db;

};

This will allow us to easily establish a connection to the database later in the server-side code.

Next, we need to create a test webpage with a basic form that we can use to log in. In the "templates" directory of your project root, create a new file called "login.phtml". In this file, we'll start with the bare-bones structure of any html page.

<!DOCTYPE>

<html>

<head>

</head>

<body>

</body>

</html>

Between the <body> tags, we're going to create a basic form where the user can enter their username and password in an attempt to log in.

<body>

<p>Login</p>

<form name="loginform" onSubmit="return validateForm();" action="./login" method="post">

<label>Username</label>

<input type="text" name="usr" placeholder="username">

<label>Password</label>

<input type="password" name="pword" placeholder="password">

<input type="submit" value="Login"/>

</form>

</body>

The "name" attribute in the form and input tags assigns a name to the elements so that they can be referred to later.. The "onSubmit" attribute indicates what will be done once the user clicks the Submit button. In this case, when the user clicks the Submit button, a JavaScript function validateForm() will be called. The "action" attribute and the "method" attribute define the API call that will be made, if, when the Submit button is clicked, validateForm() returns True.

Next, we need to write a script to validate the user's login information. To get the javascript without any installation, between the head tags, place the following:

<head>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.0/jquery.min.js"></script>

</head>

Between the closing </body> tag and the closing </html> tag, we're going to write our script.

<script>

function validateForm()

{

var un = document.loginform.usr.value;

var pw = document.loginform.pword.value;

}

</script>

Now, when the user enters their username and password and clicks the Submit button, a POST request will be made to the server. The next step is to write the server-side code to handle it. In the routes.php file in the "src" directory, we need to define a POST function for '/login':

$app->post('/login', function ($request, $response, $args) {

)};

In the body of this function, start by getting the username and password from the POST. $\_POST is a variable that “comes with” the app at the start of the session. It contains pieces of information sent in the request. Similarly, there are variables called $\_GET, $\_PUT, and $\_DELETE, which store information sent in those types of requests.

$uName = $\_POST['usr'];

$pWord = $\_POST['pword'];

Now, establish a connection to the database using the settings we defined earlier in settings.php and query it for the submitted username and password.

$db = $this->login\_dbConn;

$db\_password = $db->query("SELECT password FROM users WHERE username = '" . $uName . "'")->fetch(PDO::FETCH\_ASSOC)['password'];

Finally, we would like to show some proof of concept that the validation of the user credentials works, so we will check the password from the database against the submitted password. Again, this method of password storage and validation is extremely insecure and should never be used in actual production, but this is intended to give you a barebones example that you can flesh out later. So, based on the user’s password, redirect them to different landing pages.

if($db\_password != $pWord)

{

return $response->withRedirect('/test\_invalid\_login');

}

else

{

return $response->withRedirect('/test\_successful\_login');

}

Now, define the two routes that the previous route wants to redirect you to. In practice, if the user’s credentials are incorrect, you could redirect them back to the login page and if they are correct, direct them to some home page. For now, we will just display a message in the browser describing the success or failure of the “login”.

$app->get('/test\_invalid\_login', function ($request, $response, $args)

{

return $response->write('Invalid username/password combination');

});

$app->get('/test\_successful\_login', function ($request, $response, $args)

{

return $response->write('Valid username/password combination. You have successfully logged in.');

});

Finally, we need a route to make it to the simple HTML form we created earlier.

$app->get('/login', function ($request, $response, $args)

{

return $this->renderer->render($response, 'login.phtml', $args);

});

The setup is complete. Now, to test, navigate to “<project\_directory>/login” in your browser and enter the username and password combination you inserted into the database (make sure it is the entry in the database, not the user info for your MySQL user). If you set everything up correctly, you should see “Valid username/password combination. You have successfully logged in.”

show up in the browser.

Congratulations! You now should have a basic idea of how the front-end fits in with the back-end as it pertains to user login information.

Some notes:

* In production, do not send sensitive user information over HTTP, use HTTPS instead.
* Make sure to come up with a secure way to store the users’ passwords.
* In future iterations, rather than redirecting the user to the proper page, when the user logs in, you can generate a random string to be used as a token that you can send back in the HTTP response, which the front-end can then send back on subsequent requests to the server. This allows you to validate your users without them having to login every time they view a page.
  + Make sure your app can handle this validation. This is a good use for Slim middleware.