Authentication With Laravel 4

Authentication is required for virtually any type of web application. In this tutorial, I'd like to show you how you can go about creating a small authentication application using Laravel 4. We'll start from the very beginning by creating our Laravel app using composer, creating the database, loading in the Twitter Bootstrap, creating a main layout, registering users, logging in and out, and protecting routes using filters. We've got a lot of code to cover, so let's get started!

Installation

Let's start off this tutorial by setting up everything that we'll need in order to build our authentication application. We'll first need to download and install Laravel plus all of its dependencies. We'll also utilize the popular Twitter Bootstrap to make our app look pretty. Then we'll do a tad bit of configuration, connect to our database and create the required table and finally, start up our server to make sure everything is working as expected.

Download

Let's use composer to create a new Laravel application. I'll first change directories into my `Sites` folder as that's where I prefer to store all of my apps:

```
1  cd Sites
```

Then run the following command to download and install Laravel (I named my app `laravel-auth`) and all of its dependencies:

```
1  composer create-project laravel/laravel laravel-auth
```

Add In Twitter Bootstrap

Now to keep our app from suffering a horrible and ugly fate of being styled by yours truly, we'll include the Twitter bootstrap within our `composer.json` file:

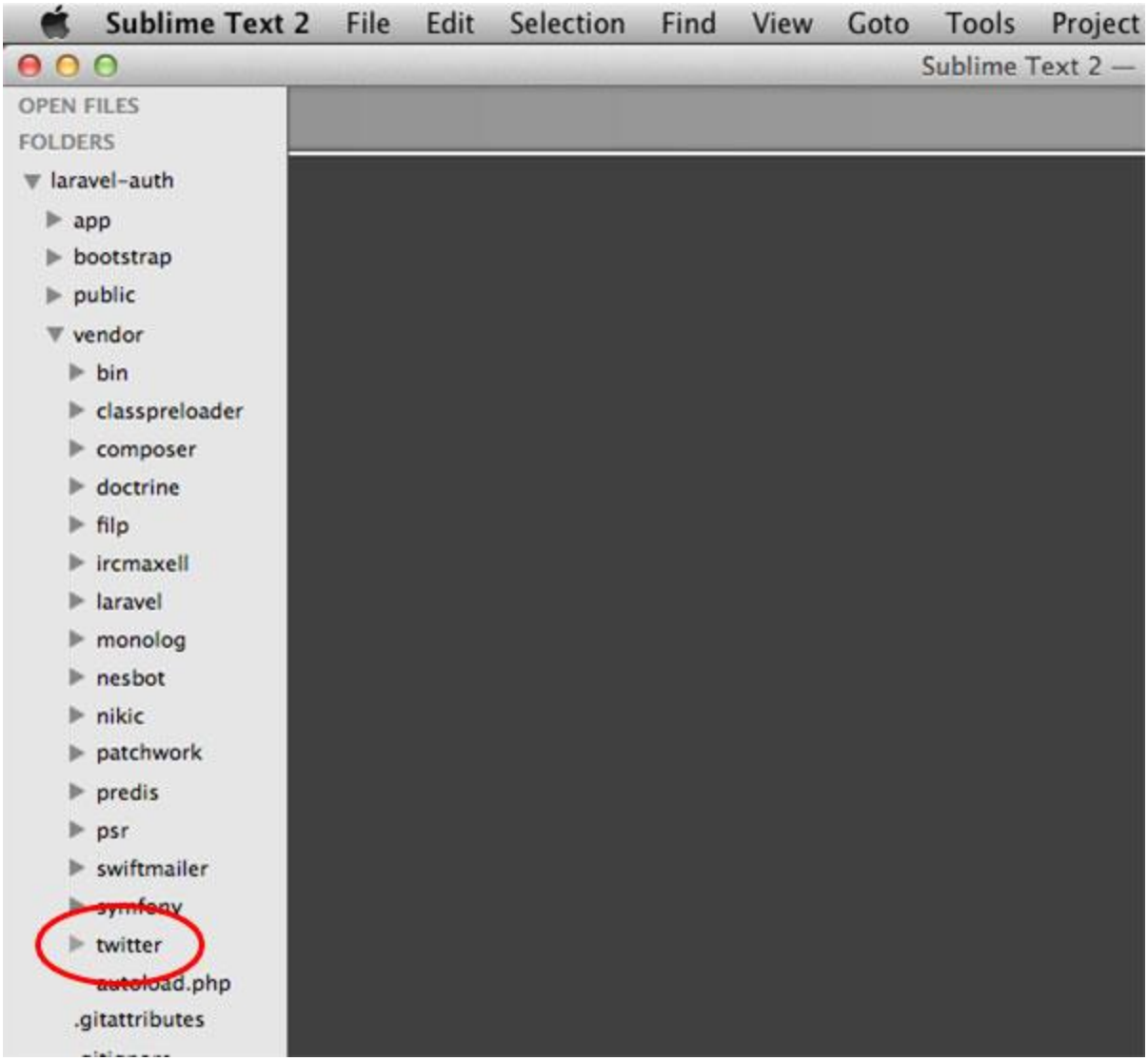
```
{
	"name": "laravel/laravel",
	"description": "The Laravel Framework.",
	"keywords": ["framework", "laravel"],
	"require": {
		"laravel/framework": "4.0.*",
		"twitter/bootstrap": "*"
	},

	// The rest of your composer.json file below ....
```

... and then we can install it:

```
1  composer update
```

Now if you open up your app into your text editor, I'm using Sublime, and if you look in the `vendor` folder you'll see we have the Twitter Bootstrap here.



Now by default our Twitter Bootstrap is composed of `.less` files and before we can compile them into `.CSS` files, we need to install all of the bootstrap dependencies. This will also allow us to use the `Makefile` that is included with the Twitter bootstrap for working with the framework (such as compiling files and running tests).

Note: You will need npm in order to install these dependencies.

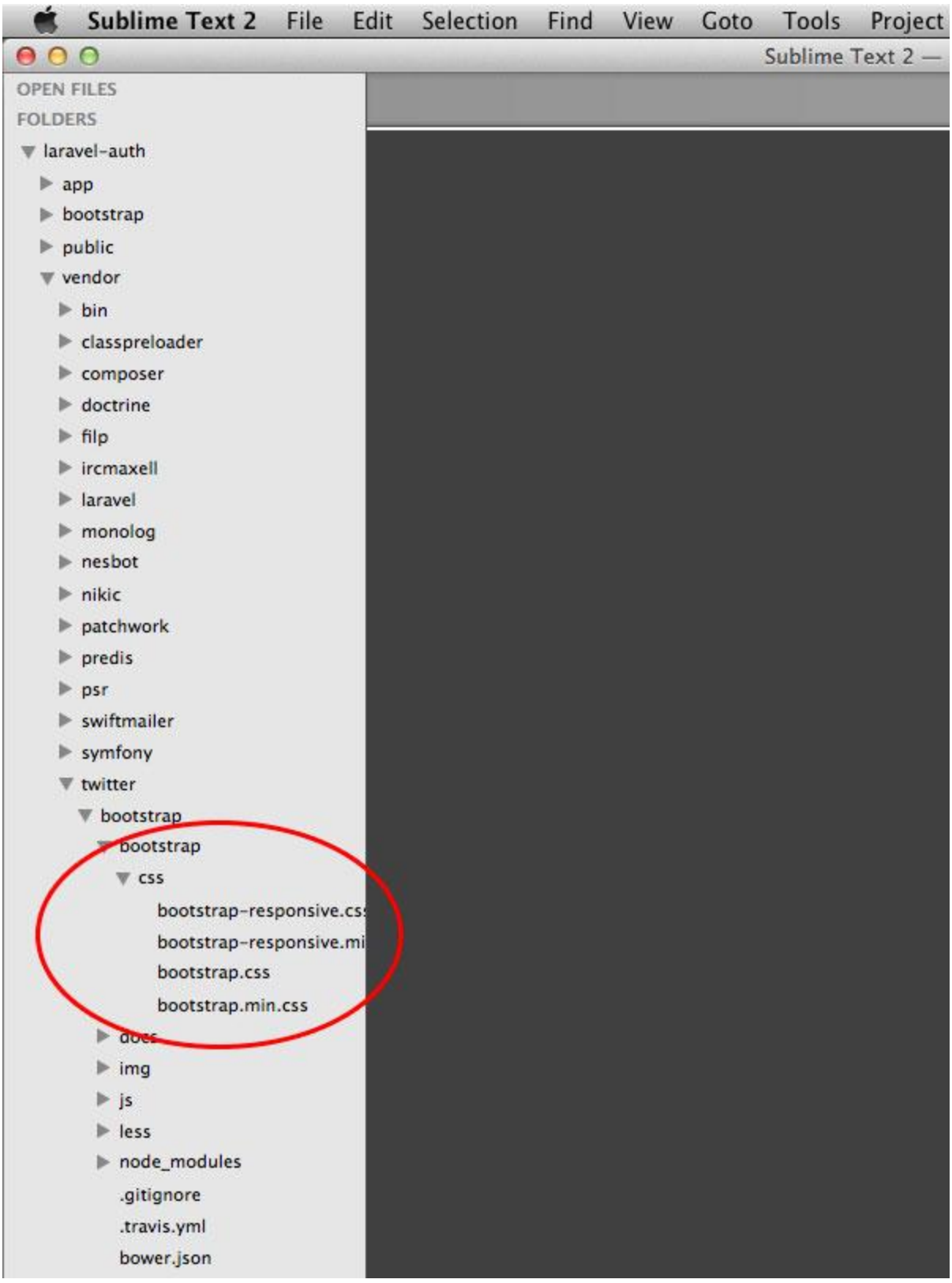
In your terminal, let's change directories into `vendor/twitter/bootstrap` and run `npm install`:

```
cd ~/Sites/laravel-auth/vendor/twitter/bootstrap
npm install
```

With everything ready to go, we can now use the `Makefile` to compile the `.less` files into CSS. Let's run the following command:

```
make bootstrap-css
```

You should now notice that we have two new folders inside our `vendor/twitter/bootstrap` directory named `bootstrap/css` which contain our bootstrap CSS files.



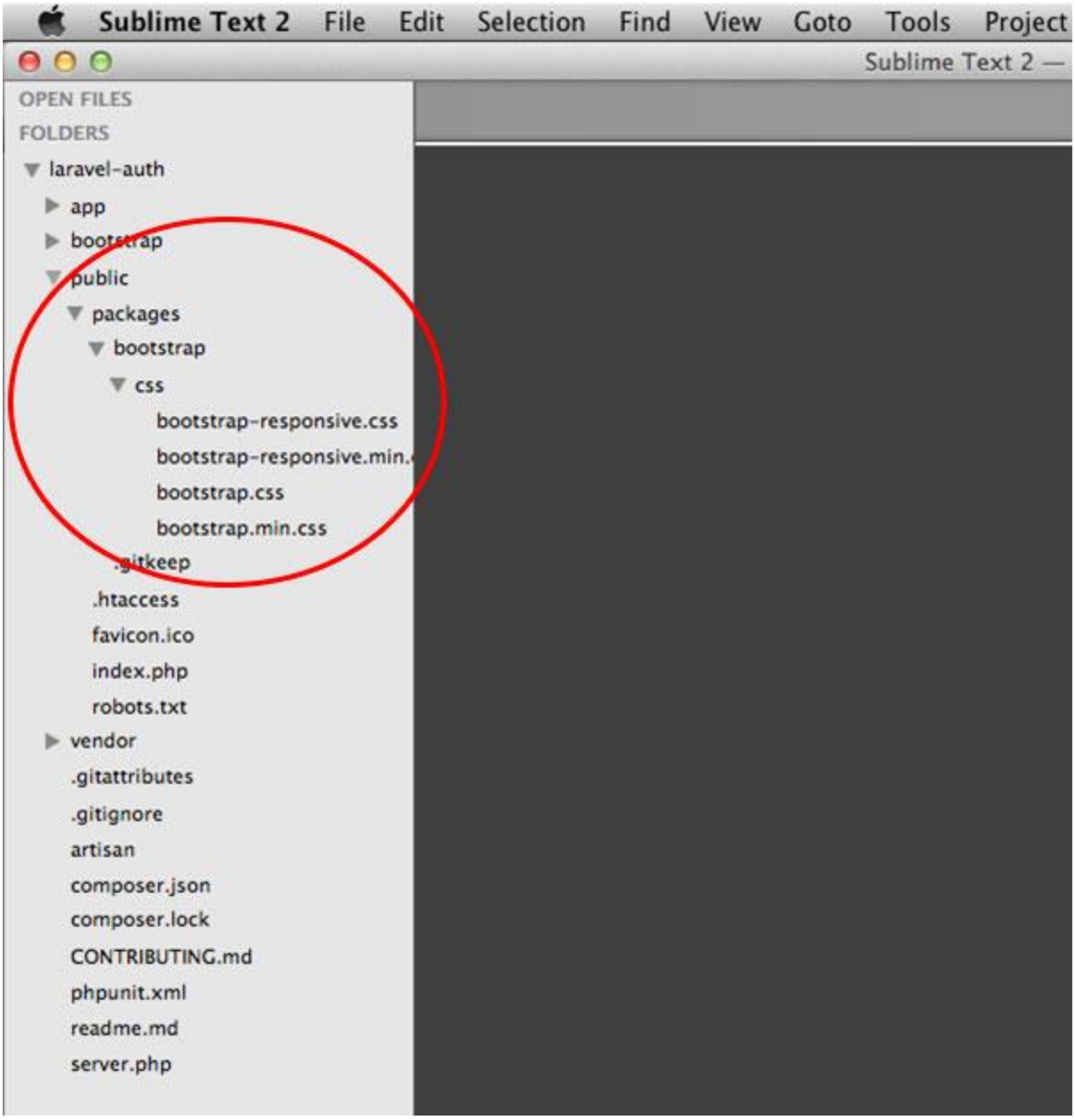
Now we can use the bootstrap CSS files later on, in our layout, to style our app.

But, we have a problem! We need these CSS files to be publicly accessible, currently they are located in our `vendor` folder. But this is an easy fix! We can use artisan to `publish` (move) them to our `public/packages` folder, that way we can link in the required CSS files into our main layout template, which we'll create later on.

First, we'll change back into the root of our Laravel application and then run artisan to move the files:

```
cd ~/Sites/laravel-auth
php artisan asset:publish --path="vendor/twitter/bootstrap/bootstrap/css"
bootstrap/css
```

The artisan command `asset:publish` allows us to provide a `--path` option for which files we want to move into our `public/packages` directory. In this case, we tell it to publish all of the CSS files that we compiled earlier and place them inside of two new folders named `bootstrap/css`. Your `public` directory should now look like the screenshot below, with our Twitter Bootstrap CSS files now publicly accessible:



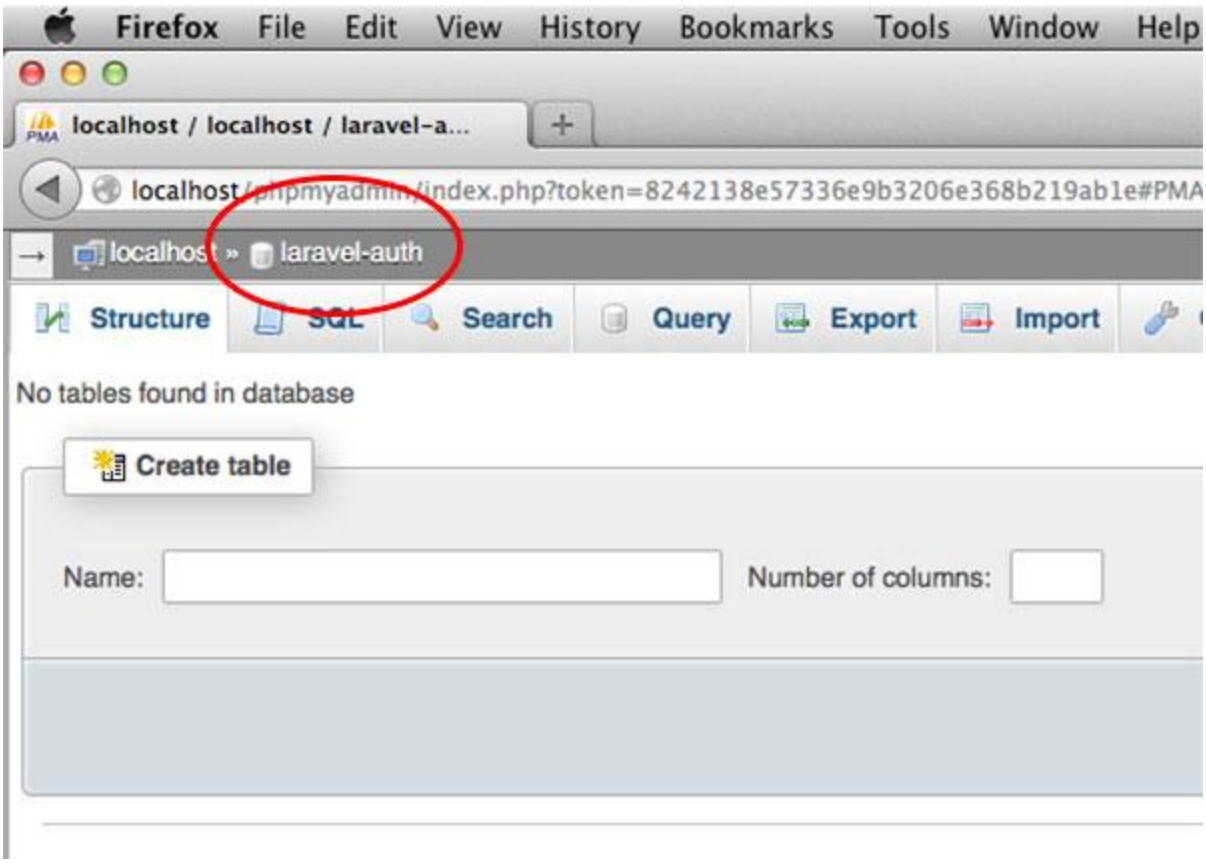
Set Permissions

Next we need to ensure our web server has the appropriate permissions to write to our applications `app/storage` directory. From within your app, run the following command:

```
1  chmod -R 755 app/storage
```

Connect To Our Database

Next, we need a database that our authentication app can use to store our users in. So fire up whichever database you are more comfortable using, personally, I prefer MySQL along with PHPMyAdmin. I've created a new, empty database named: `laravel-auth`.



Now let's connect this database to our application. Under `app/config` open up `database.php`. Enter in your appropriate database credentials, mine are as follows:

```
01  // Default Database Connection Name
02
03  'default' => 'mysql',
04
05  // Database Connections
06
```

```
    'connections' => array(

        'mysql' => array(
            'driver'  => 'mysql',
            'host'   => '127.0.0.1',
            'database' => 'laravel-auth',
            'username' => 'root',
            'password' => '',
            'charset'  => 'utf8',
            'collation' => 'utf8_unicode_ci',
            'prefix'  => '',
        ),

        // the rest of your database.php file's code ...
```

Create the Users Table

With our database created, it won't be very useful unless we have a table to store our users in. Let's use artisan to create a new migration file named: `create-users-table`:

```
php artisan migrate:make create-users-table
```

Let's now edit our newly created migration file to create our `users` table using the Schema Builder. We'll start with the `up()` method:

```
public function up()
{
    $table->increments('id');
    $table->string('firstname', 20);
    $table->string('lastname', 20);
    $table->string('email', 100)->unique();
    $table->string('password', 64);
```

```
7         $table->timestamps();
8     }
9
```

This will create a table named `users` and it will have an `id` field as the primary key, `firstname` and `lastname` fields, an `email` field which requires the email to be unique, and finally a field for the `password` (must be at least 64 characters in length) as well as a few `timestamps`.

Now we need to fill in the `down()` method in case we need to revert our migration, to drop the `users` table:

```
1   public function down()
2   {
3       Schema::drop('users');
4   }
```

And now we can run the migration to create our `users` table:

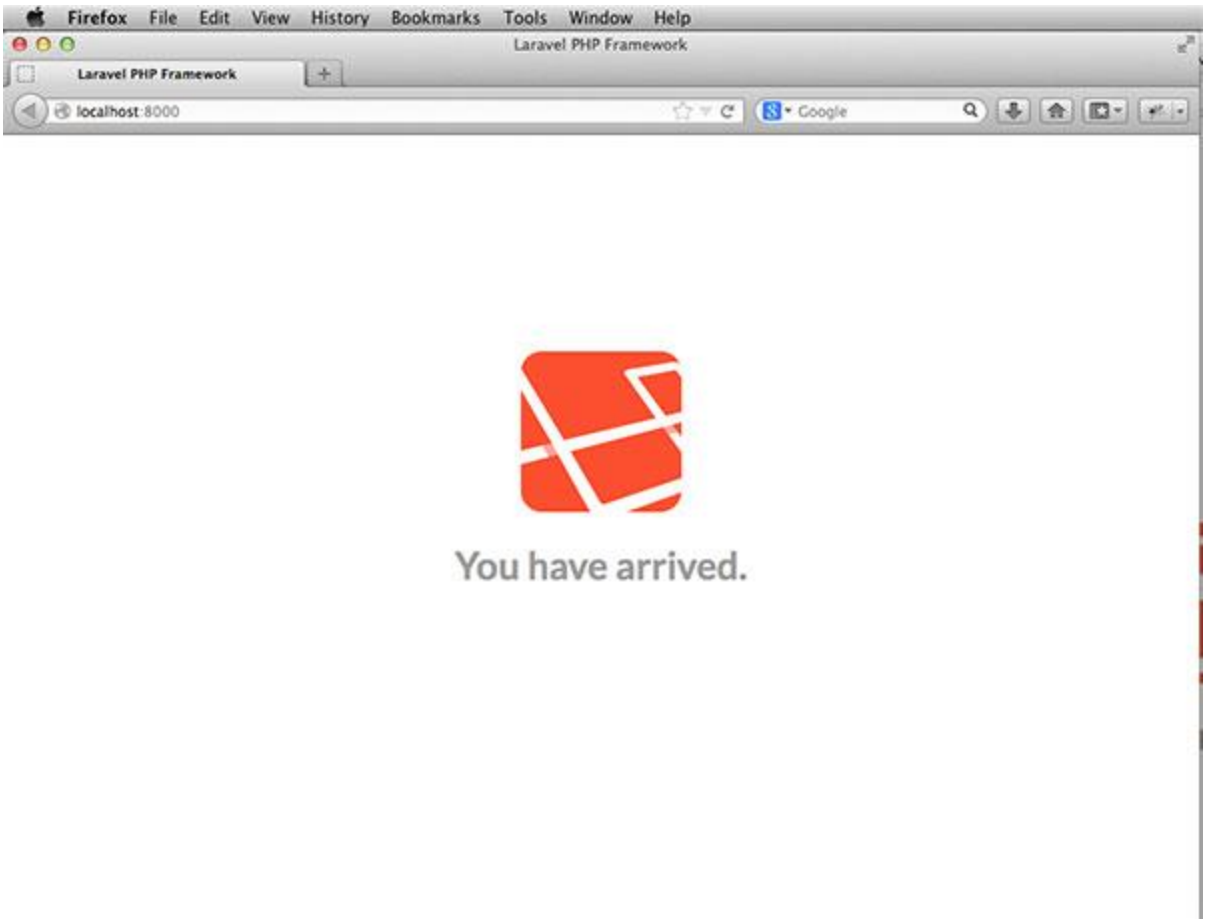
```
1   php artisan migrate
```

Start Server & Test It Out

Alright, our authentication application is coming along nicely. We've done quite a bit of preparation, let's start up our server and preview our app in the browser:

```
1   php artisan serve
```

Great, the server starts up and we can see our home page:



Making the App Look Pretty

Before we go any further, it's time to create a main layout file, which will use the Twitter Bootstrap to give our authentication application a little style!

Creating a Main Layout File

Under `app/views/` create a new folder named `layouts` and inside it, create a new file named `main.blade.php` and let's place in the following basic HTML structure:

```
01  <!DOCTYPE html>
02  <html lang="en">
03    <head>
```

```
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Authentication App With Laravel 4</title>
  </head>

  <body>

  </body>
</html>
```

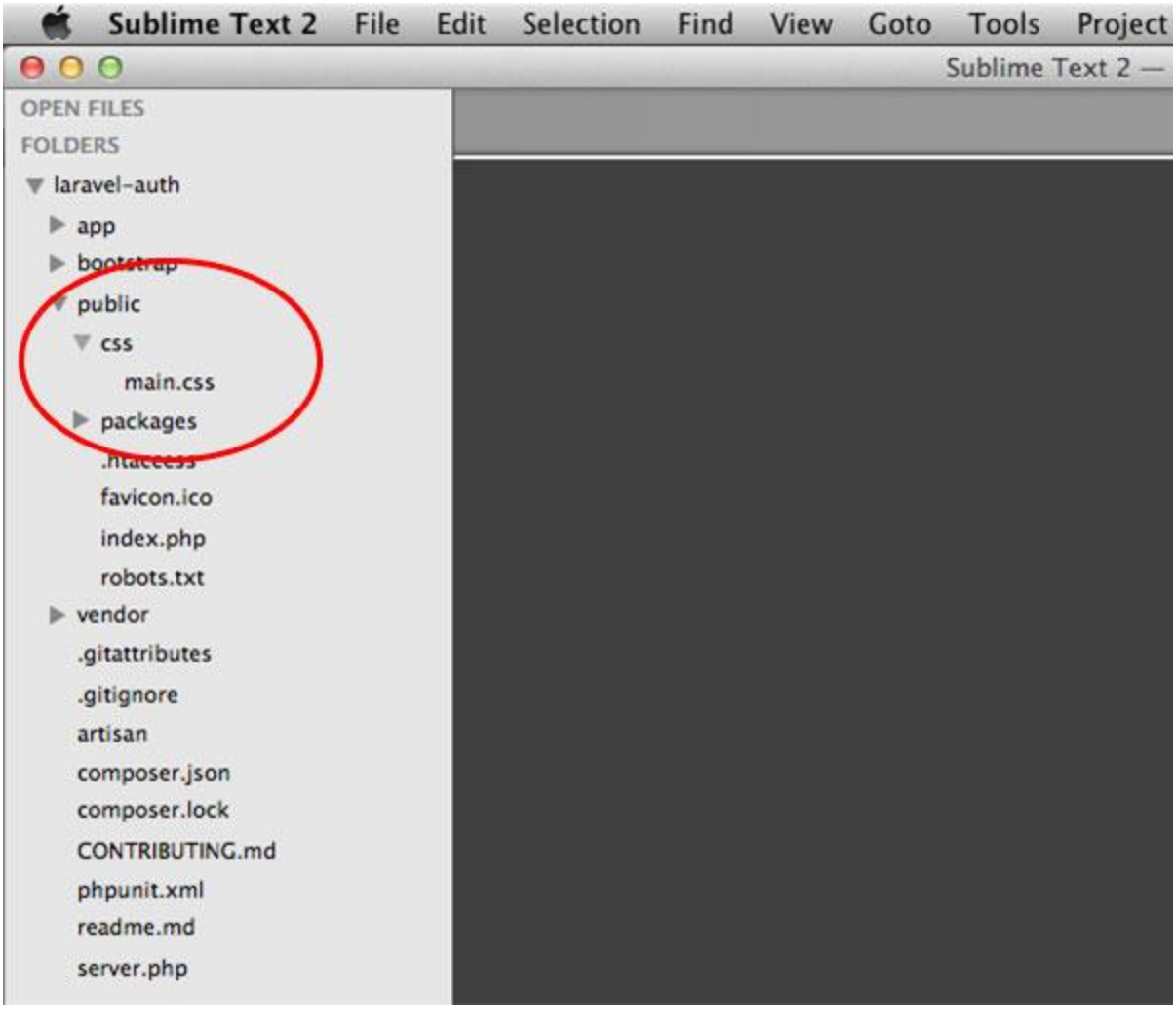
Linking In the CSS Files

Next, we need to link in our bootstrap CSS file as well as our own `main` CSS file, in our `head` tag, right below our `title`:

```
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Authentication App With Laravel 4</title>
    {{ HTML::style('packages/bootstrap/css/bootstrap.min.css') }}
    {{ HTML::style('css/main.css')}}
</head>
```

Now we just need to create this `main.css` file where we can add our own customized styling for our app. Under the `public` directory create a new folder named `css` and within it create a new file named `main.css`.



Finishing the Main Layout

Inside of our `body` tag, let's create a small navigation menu with a few links for registering and logging in to our application:

```
01  <body>
02
03      <div class="navbar navbar-fixed-top">
04          <div class="navbar-inner">
05              <div class="container">
06                  <ul class="nav">
                        <li>{{ HTML::link('users/register', 'Register') }}</li>
```

```
                    <li>{{ HTML::link('users/login', 'Login') }}</li>
                </ul>
            </div>
        </div>
    </div>


</body>


```

Notice the use of several Bootstrap classes in order to style the navbar appropriately. Here we're just using a couple of DIVs to wrap an unordered list of navigation links, pretty simple.

For our application, we're going to want to give our users simple flash messages, like a success message when the user registers. We'll set this flash message from within our controller, but we'll echo out the message's value here in our layout. So let's create another `div` with a class of `.container` and display any available flash messages right after our navbar:

```
<body>


    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <ul class="nav">
                    <li>{{ HTML::link('users/register', 'Register') }}</li>
                    <li>{{ HTML::link('users/login', 'Login') }}</li>
                </ul>
            </div>
        </div>
    </div>


```

```
13
14          <div class="container">
15              @if(Session::has('message'))
16                  <p class="alert">{{ Session::get('message') }}</p>
17              @endif
18          </div>
19
20          </body>
21
```

To display the flash message, I've first used a Blade `if` statement to check if we have a flash message to display. Our flash message will be available in the Session under `message`. So we can use the `Session::has()` method to check for that message. If that evaluates to true, we create a paragraph with the Twitter bootstrap class of `alert` and we call the `Session::get()` method to display the message's value.

Now lastly, at least for our layout file, let's echo out a `$content` variable, right after our flash message. This will allow us to tell our controller to use this layout file, and our views will be displayed in place of this `$content` variable, right here in the layout:

```
01  <body>
02
03      <div class="navbar navbar-fixed-top">
04          <div class="navbar-inner">
05              <div class="container">
06                  <ul class="nav">
07                      <li>{{ HTML::link('users/register', 'Register') }}</li>
08                      <li>{{ HTML::link('users/login', 'Login') }}</li>
09                  </ul>
10              </div>
```

```
    </div>



    <div class="container">
        @if(Session::has('message'))
            <p class="alert">{{ Session::get('message') }}</p>
        @endif

        {{ $content }}
    </div>

    </body>


```

Custom Styling

Now that we have our layout complete, we just need to add a few small custom CSS rules to our `main.css` file to customize our layout a little bit more. Go ahead and add in the following bit of CSS, it's pretty self explanatory:

```
body {
    padding-top: 40px;
}

.form-signup, .form-signin {
    width: 400px;
    margin: 0 auto;
}

```

I added just a small amount of padding to the top of the `body` tag in order to prevent our navbar from overlapping our main content. Then I target the Bootstrap's `.form-signup` and `.form-signin` classes, which we'll be applying to our register and login forms in order to set their width and center them on the page.

Creating the Register Page

It's now time to start building the first part of our authentication application and that is our Register page.

The Users Controller

We'll start by creating a new `UsersController` within our `app/controllers` folder and in it, we define our `UsersController` class:

```
<?php

class UsersController extends BaseController {

}
?>
```

Next, let's tell this controller to use our `main.blade.php` layout. At the top of our controller set the `$layout` property:

```
<?php

class UsersController extends BaseController {
    protected $layout = "layouts.main";
}
?>
```

Now within our `UsersController`, we need an action for our register page. I named my action `getRegister`:

```
1  public function getRegister() {
2      $this->layout->content = View::make('users.register');
3  }
```

Here we just set the `content` layout property (this is the `$content` variable we echo'd out in our layout file) to display a `users.register` view file.

The Users Controller Routes

With our controller created next we need to setup the routes for all of the actions we might create within our controller. Inside of our `app/routes.php` file let's first remove the default `/` route and then add in the following code to create our `UsersController` routes:

```
1  Route::controller('users', 'UsersController');
```

Now anytime that we create a new action, it will be available using a URI in the following format: `/users/actionName`. For example, we have a `getRegister` action, we can access this using the following URI: `/users/register`.

Note that we don't include the "get" part of the action name in the URI, "get" is just the HTTP verb that the action responds to.

Creating the Register View

Inside of `app/views` create a new folder named `users`. This will hold all of our `UsersController`'s view files. Inside the `users` folder create a new file named `register.blade.php` and place the following code inside of it:

```
01  {{ Form::open(array('url'=>'users/create', 'class'=>'form-signup')) }}
02      <h2 class="form-signup-heading">Please Register</h2>
03
        <ul>
```

```
04            @foreach($errors->all() as $error)
05                <li>{{ $error }}</li>
06            @endforeach
07        </ul>
08
09        {{ Form::text('firstname', null,
          array('class'=>'input-block-level', 'placeholder'=>'First Name')) }}
10        {{ Form::text('lastname', null,
11        array('class'=>'input-block-level', 'placeholder'=>'Last Name')) }}
12        {{ Form::text('email', null,
13        array('class'=>'input-block-level', 'placeholder'=>'Email Address')) }}
14        {{ Form::password('password',
15        array('class'=>'input-block-level', 'placeholder'=>'Password')) }}
16        {{ Form::password('password_confirmation',
17        array('class'=>'input-block-level',
          'placeholder'=>'Confirm Password')) }}

          {{ Form::submit('Register', array('class'=>'btn btn-large btn-primary btn-block')
      {{ Form::close() }}
```

Here we use the `Form` class to create our register form. First we call the `open()` method, passing in an array of options. We tell the form to submit to a URI of `users/create` by setting the `url` key. This URI will be used to process the registration of the user. We'll handle this next. After setting the `url` we then give the form a class of `form-signup`.

After opening the form, we just have an `h2` heading with the `.form-signup-heading` class.

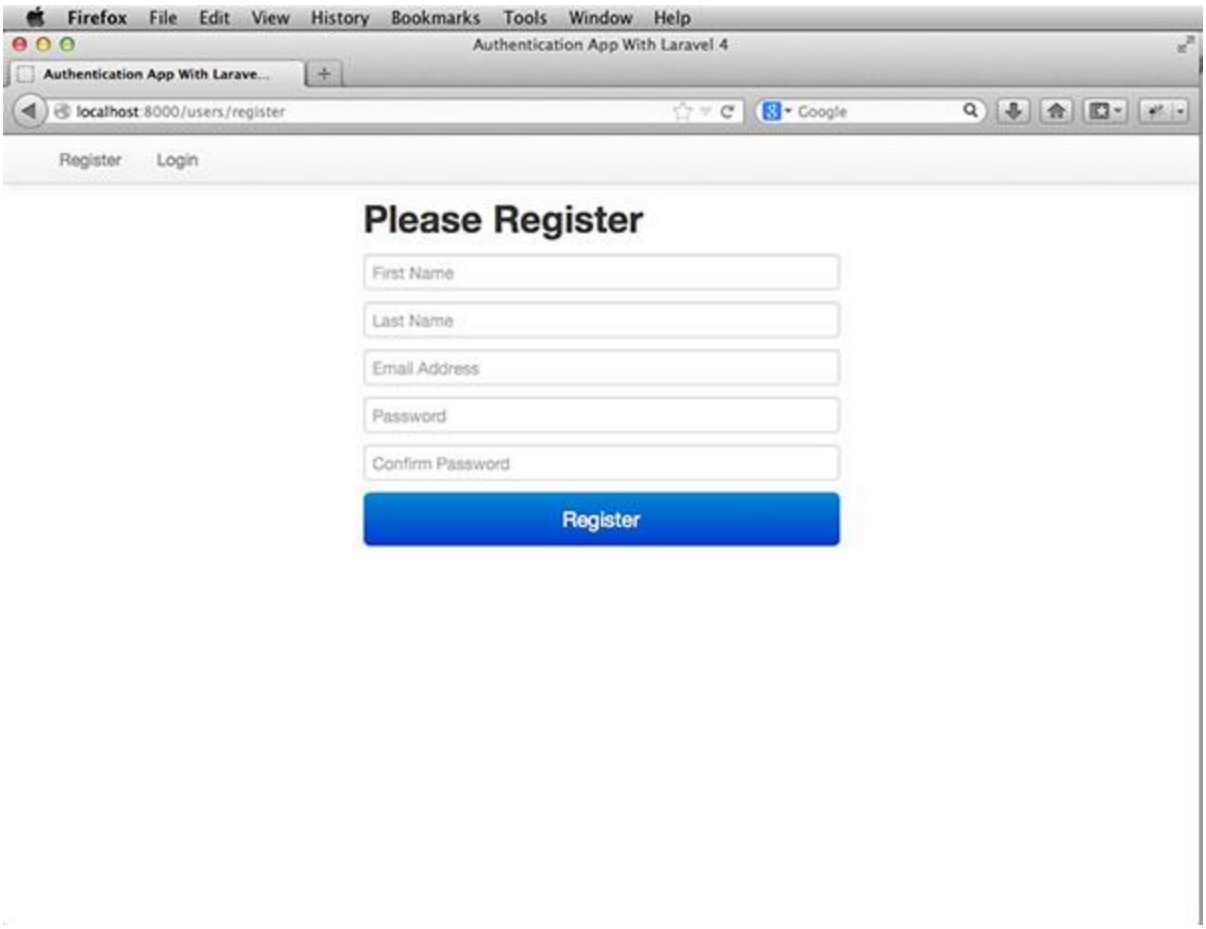
Next, we use a `@foreach` loop, looping over all of the form validation error messages and displaying each `$error` in the unordered list.

After the form validation error messages, we then we create several form input fields, each with a class of `input-block-level` and a placeholder value. We have inputs for the firstname, lastname, email, password, and password confirmation fields. The second argument to the `text()` method is set to `null`, since we're using a `placeholder`, we don't need to set the input fields value attribute, so I just set it to `null` in this case.

After the input fields, we then create our submit button and apply several different classes to it so the Twitter bootstrap handles the styling for us.

Lastly, we just close the form using the `close()` method.

Make sure to start up your server, switch to your favorite browser, and if we browse to `http://localhost:8000/users/register` you should see your register page:



Processing the Register Form Submission

Now if you tried filling out the register form's fields and hitting the **Register** button you would have been greeted with a `NotFoundHttpException`, and this is because we have no route that matches the `users/create` URI, because we do not have an action to process the form submission. So that's our next step!

Creating a `postCreate` Action

Inside of your `UsersController` let's create another action named `postCreate`:

```
1  public function postCreate() {
2
```

```
3    }
```

Now this action needs to handle processing the form submission by validating the data and either displaying validation error messages or it should create the new user, hashing the user's password, and saving the user into the database.

Form Validation

Let's start with validating the form submission's data. We first need to create our validation rules that we'll validate the form data against. I prefer storing my validation rules in my model as that's the convention I'm used to, from working with other frameworks. By default, Laravel ships with a `User.php` model already created for you.

Make sure you don't delete this User model or remove any of the preexisting code, as it contains new code that is required for Laravel 4's authentication to work correctly. Your User model must implement `UserInterface` and `RemindableInterface` as well as implement the `getAuthIdentifier()` and `getAuthPassword()` methods.

Under `app/models` open up that `User.php` file and at the top, add in the following code:

```
1    public static $rules = array(
2        'firstname'=>'required|alpha|min:2',
3        'lastname'=>'required|alpha|min:2',
4        'email'=>'required|email|unique:users',
5        'password'=>'required|alpha_num|between:6,12|confirmed',
6        'password_confirmation'=>'required|alpha_num|between:6,12'
7        );
```

Here I'm validating the `firstname` and `lastname` fields to ensure they are present, only contain alpha characters, and that they are at least two characters in length. Next, I validate the `email` field to ensure that it's present, that it is a valid email address, and that it is unique to the users table, as we don't want to have duplicate email addresses for our users. Lastly, I validate the `password` and `password_confirmation` fields. I ensure they are both present, contain only alpha-numeric characters and that they are

between six and twelve characters in length. Additionally, notice the `confirmed` validation rule, this makes sure that the `password` field is exactly the same as the matching `password_confirmation` field, to ensure users have entered in the correct password.

Now that we have our validation rules, we can use these in our `UsersController` to validate the form submission. In your `UsersController`'s `postCreate` action, let's start by checking if the data passes validation, add in the following code:

```
01
    public function postCreate() {
02
        $validator = Validator::make(Input::all(), User::$rules);
03
04
        if ($validator->passes()) {
05
            // validation has passed, save user in DB
06
        } else {
07
            // validation has failed, display error messages
08
        }
09
    }
10
```

We start by creating a validator object named `$validator` by calling the `User::validate()` method. This accepts the two arguments, the submitted form input that should be validated and the validation rules that the data should be validated against. We can grab the submitted form data by calling the `Input::all()` method and we pass that in as the first argument. We can get our validation rules that we created in our `User` model by accessing the static `User::$rules` property and passing that in as the second argument.

Once we've created our validator object, we call its `passes()` method. This will return either `true` or `false` and we use this within an `if` statement to check whether our data has passed validation.

Within our `if` statement, if the validation has passed, add in the following code:

```
01  if ($validator->passes()) {
02      $user = new User;
03      $user->firstname = Input::get('firstname');
04      $user->lastname = Input::get('lastname');
05      $user->email = Input::get('email');
06      $user->password = Hash::make(Input::get('password'));
07      $user->save();
08
09      return Redirect::to('users/login')->with('message',
        'Thanks for registering!');
10  } else {
11      // validation has failed, display error messages
12  }
```

As long as the data that the user submits has passed validation, we create a new instance of our User model: `new User;` storing it into a `$user` variable. We can then use the `$user` object and set each of the user's properties using the submitted form data. We can grab the submitted data individually using the `Input::get('fieldName')` method. Where `fieldName` is the field's value you want to retrieve. Here we've grabbed the firstname, lastname, and email fields to use for our new user. We also grabbed the password field's value, but we don't just want to store the password in the database as plain text, so we use the `Hash::make()` method to hash the submitted password for us before saving it. Lastly, we save the user into the database by calling the `$user` object's `save()` method.

After creating the new user, we then redirect the user to the login page (we'll create the login page in a few moments) using the `Redirect::to()` method. This just takes in the URI of where you'd like to redirect to. We also chain on the `with()` method call in order to give the user a flash message letting them know that their registration was successful.

Now if the validation does not pass, we need to redisplay the register page, along with some validation error messages, with the old input, so the user can correct their mistakes. Within your `else` statement, add in the following code:

```
01  if ($validator->passes()) {
02      $user = new User;
03      $user->firstname = Input::get('firstname');
04      $user->lastname = Input::get('lastname');
05      $user->email = Input::get('email');
06      $user->password = Hash::make(Input::get('password'));
07      $user->save();
08
09      return Redirect::to('users/login')->with('message',
10      'Thanks for registering!');
11  } else {
12      return Redirect::to('users/register')->with('message',
        'The following errors occurred')->withErrors($validator)->withInput();
    }
```

Here we just redirect the user back to the register page with a flash message letting them know some errors have occurred. We make sure to display the validation error messages by calling the `withErrors($validator)` method and passing in our `$validator` object to it. Finally, we call the `withInput()` method so the form remembers what the user originally typed in and that will make it nice and easy for the user to correct the errors.

Adding In the CSRF Before Filter

Now we need to make sure to protect our POST actions from CSRF attacks by setting the CSRF before filter within our `UsersController`'s constructor method. At the top of your `UsersController` add in the following code:

```
1  public function __construct() {
2      $this->beforeFilter('csrf', array('on'=>'post'));
```

```
3    }
```

Within our constructor, we call the `beforeFilter()` method and pass in the string `csrf`, as the first argument. `csrf` is the filter that we want to apply to our actions. Then we pass in an array as the second argument and tell it to only apply this filter on POST requests. By doing this, our forms will pass along a CSRF token whenever they are submitted. This CSRF before filter will ensure that all POST requests to our app contain this token, giving us confidence that POST requests are not being issued to our application from other external sources.

Creating the Login Page

Before you run off and try out your register page, we first need to create the Login page so that when our register form submission is successful, we don't get an error. Remember, if the form validation passes, we save the user and redirect them to the login page. We currently don't have this login page though, so let's create it!

Still inside of your `UsersController`, create a new action named `getLogin` and place in the following code:

```
1    public function getLogin() {
2        $this->layout->content = View::make('users.login');
3    }
```

This will display a `users.login` view file. We now need to create that view file. Under `app/views/users` create a new file named `login.blade.php` and add in the following code:

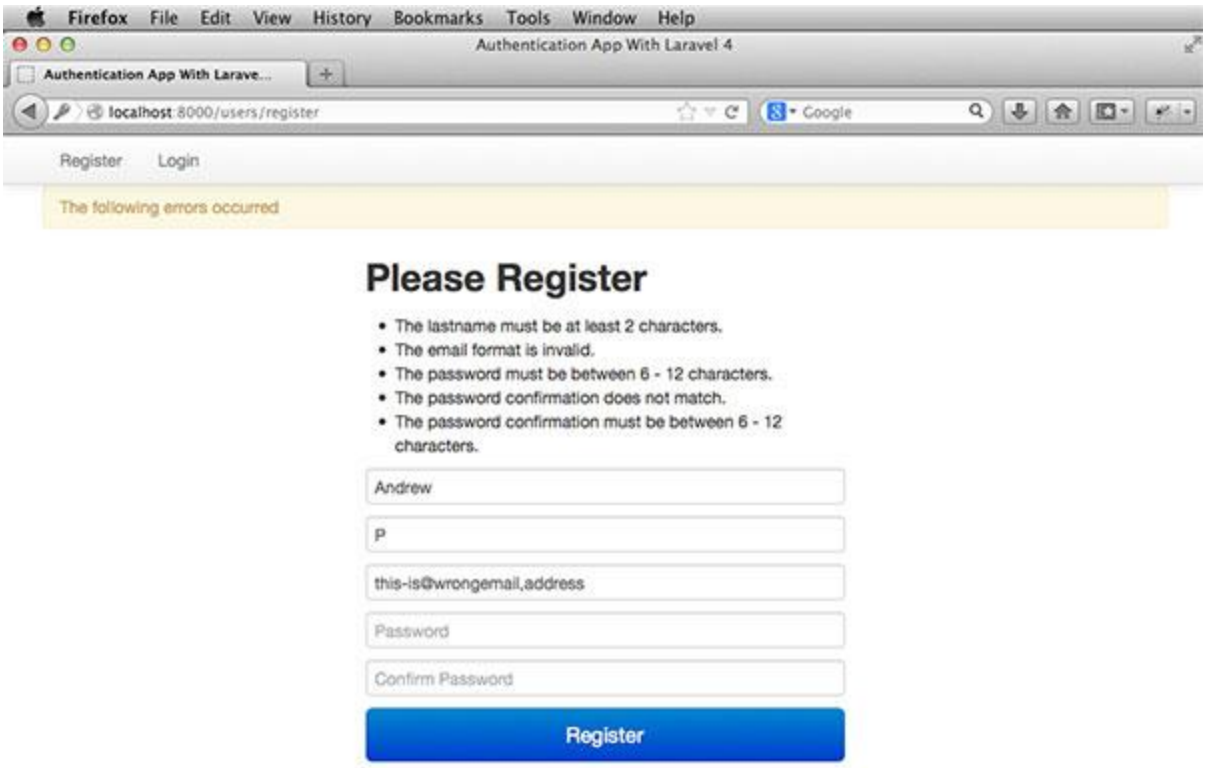
```
1    {{ Form::open(array('url'=>'users/signin', 'class'=>'form-signin')) }}
         <h2 class="form-signin-heading">Please Login</h2>
2
3        {{ Form::text('email', null, array('class'=>'input-block-level',
4        'placeholder'=>'Email Address')) }}
```

```
5       {{ Form::password('password', array('class'=>'input-block-level',
6       'placeholder'=>'Password')) }}
7
8       {{ Form::submit('Login',
        array('class'=>'btn btn-large btn-primary btn-block'))}}
    {{ Form::close() }}
```

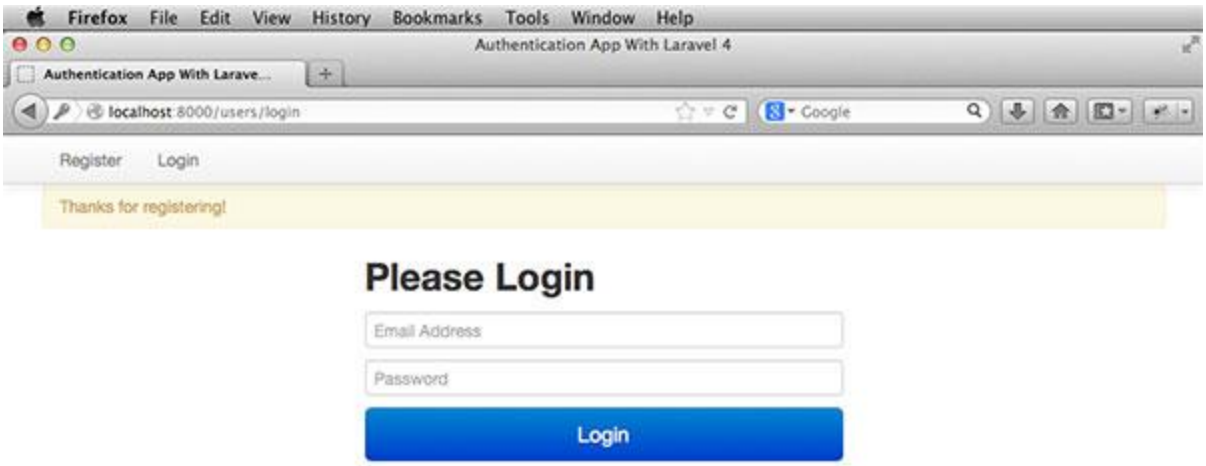
This code is very similar to the code we used in our `register` view, so I'll simplify the explanation this time to only what is different. For this form, we have it submit to a `users/signin` URI and we changed the form's class to `.form-signin`. The `h2` has been changed to say "Please Login" and its class was also changed to `.form-signin-heading`. Next, we have two form fields so the user can enter in their email and password, and then finally our submit button which just says "Login".

Let's Register a New User!

We're finally at a point to where we can try out our registration form. Of course, the login functionality doesn't work just yet, but we'll get to that soon enough. We only needed the login page to exist so that our register page would work properly. Make sure your server is still running, switch into your browser, and visit `http://localhost:8000/users/register`. Try entering in some invalid user data to test out the form validation error messages. Here's what my page looks like with an invalid user:



Now try registering with valid user data. This time we get redirected to our login page along with our success message, excellent!



Logging In

So we've successfully registered a new user and we have a login page, but we still can't login. We now need to create the `postSignin` action for our `users/signin` URI, that our login form submits to. Let's go back into our `UsersController` and create a new action named `postSignin`:

```
1  public function postSignin() {
2
3  }
```

Now let's log the user in, using the submitted data from the login form. Add the following code into your `postSignin()` action:

```
1  if (Auth::attempt(array('email'=>Input::get('email'), 'password'=>Input::get('password'
2      return Redirect::to('users/dashboard')->with('message', 'You are now logged in!');
3  } else {
4      return Redirect::to('users/login')
5          ->with('message', 'Your username/password combination was incorrect')
6          ->withInput();
7  }
```

Here we attempt to log the user in, using the `Auth::attempt()` method. We simply pass in an array containing the user's email and password that they submitted from the login form. This method will return either `true` or `false` if the user's credentials validate. So we can use this `attempt()` method within an `if` statement. If the user was logged in, we just redirect them to a `dashboard` view page and give them a success message. Otherwise, the user's credentials did not validate and in that case we redirect them back to the login page, with an error message, and display the old input so the user can try again.

Creating the Dashboard

Now before you attempt to login with your newly registered user, we need to create that dashboard page and protect it from unauthorized, non logged in users. The dashboard page should only be accessible to those users who have registered and logged in to our application. Otherwise, if a non authorized user attempts to visit the dashboard we'll redirect them and request that they log in first.

While still inside of your `UsersController` let's create a new action named `getDashboard`:

```
1  public function getDashboard() {
2
3  }
```

And inside of this action we'll just display a `users.dashboard` view file:

```
1  public function getDashboard() {
2      $this->layout->content = View::make('users.dashboard');
3  }
```

Next, we need to protect it from unauthorized users by using the `auth` before filter. In our `UsersController`'s constructor, add in the following code:

```
1  public function __construct() {
2      $this->beforeFilter('csrf', array('on'=>'post'));
3      $this->beforeFilter('auth', array('only'=>array('getDashboard')));
4  }
```

This will use the `auth` filter, which checks if the current user is logged in. If the user is not logged in, they get redirected to the login page, essentially denying the user access. Notice that I'm also passing in an array as a second argument, by setting the `only` key, I can tell this before filter to only apply it to the provided actions. In this case, I'm saying to protect only the `getDashboard` action.

Customizing Filters

By default the `auth` filter will redirect users to a `/login` URI, this does not work for our application though. We need to modify this filter so that it redirects to a `users/login` URI instead, otherwise get an error. Open up `app/filters.php` and in the **Authentication Filters** section, change the **auth** filter to redirect to `users/login`, like this:

```
01 /*
02 |--------------------------------------------------------------------------
03 | Authentication Filters
04 |--------------------------------------------------------------------------
05 |
06 | The following filters are used to verify that the user of the current
07 | session is logged into this application. The "basic" filter easily
```

```
|
*/

Route::filter('auth', function()
{
    if (Auth::guest()) return Redirect::guest('users/login');
});
```

Creating the Dashboard View

Before we can log users into our application we need to create that `dashboard` view file. Under `app/views/users` create a new file named `dashboard.blade.php` and insert the following snippet of code:

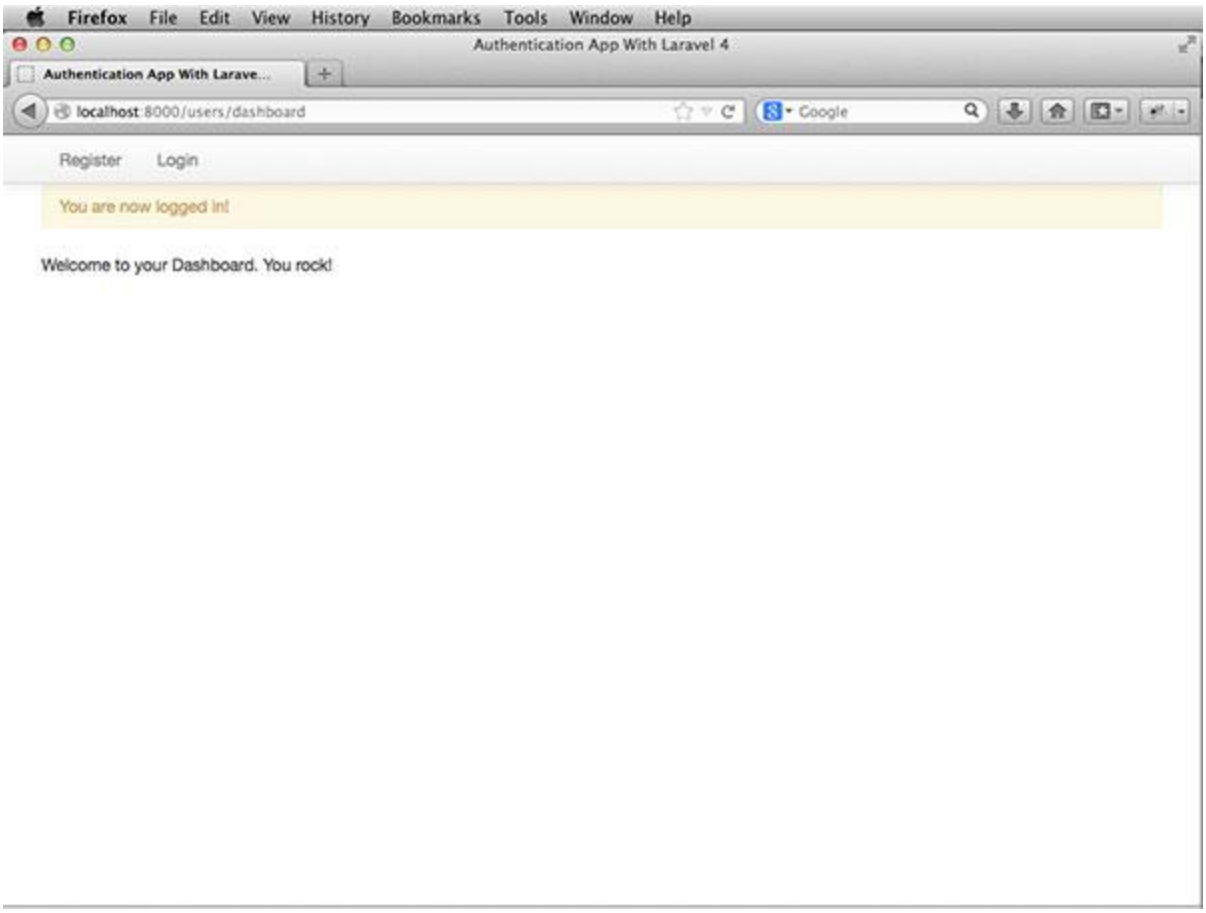
```
<h1>Dashboard</h1>

<p>Welcome to your Dashboard. You rock!</p>
```

Here I'm displaying a very simple paragraph to let the user know they are now in their Dashboard.

Let's Login!

We should now be able to login. Browse to `http://localhost:8000/users/login`, enter in your user's credentials, and give it a try.



Success!

Displaying the Appropriate Navigation Links

Ok, we can now register and login to our application, very cool! But we have a little quirk here, if you look at our navigation menu, even though we're logged in, you can see that the register and login buttons are still viewable. Ideally, we want these to only display when the user is not logged in. Once the user does login though, we want to display a logout link. To make this change, let's open up our `main.blade.php` file again. Here's what our navbar code looks like at the moment:

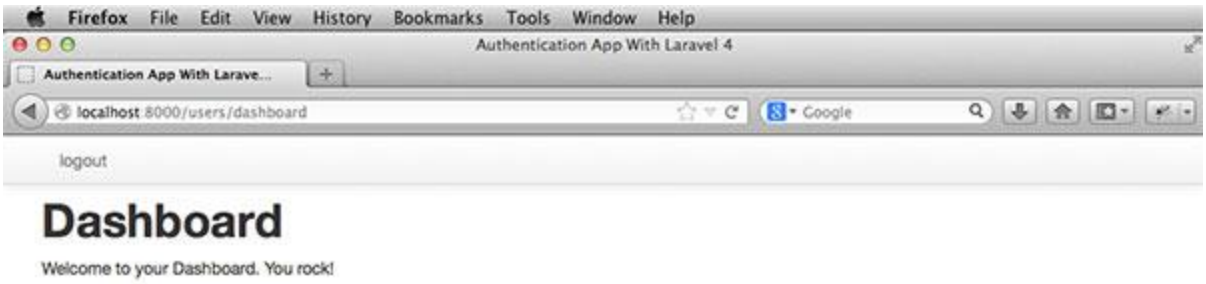
```
01  <div class="navbar navbar-fixed-top">
02      <div class="navbar-inner">
```

```
03          <div class="container">
04              <ul class="nav">
05                  <li>{{ HTML::link('users/register', 'Register') }}</li>
06                  <li>{{ HTML::link('users/login', 'Login') }}</li>
07              </ul>
08          </div>
09      </div>
10  </div>
```

Let's modify this slightly, replacing our original navbar code, with the following:

```
01
02  <div class="navbar navbar-fixed-top">
03      <div class="navbar-inner">
04          <div class="container">
05              <ul class="nav">
06                  @if(!Auth::check())
07                      <li>{{ HTML::link('users/register', 'Register') }}</li>
08                      <li>{{ HTML::link('users/login', 'Login') }}</li>
09                  @else
10                      <li>{{ HTML::link('users/logout', 'logout') }}</li>
11                  @endif
12              </ul>
13          </div>
14      </div>
```

All I've done is wrapped our `li` tags for our navbar in an `if` statement to check if the user is *not* logged in, using the `!Auth::check()` method. This method returns `true` if the user is logged in, otherwise, `false`. So if the user is not logged in, we display the register and login links, otherwise, the user is logged in and we display a logout link, instead.



Logging Out

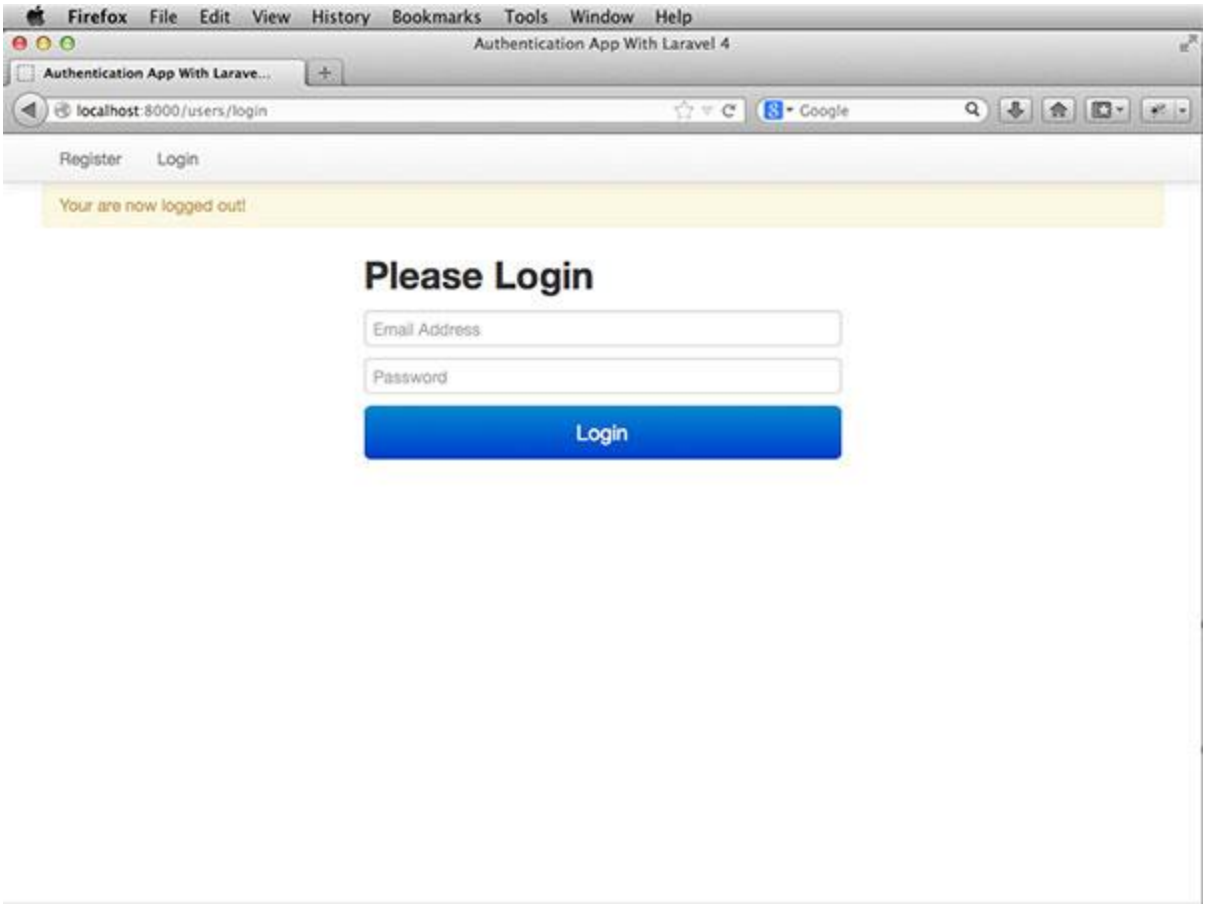
Now that our navbar displays the appropriate links, based on the user's logged in status, let's wrap up this application by creating the `getLogout` action, to actually log the user out. Within your `UsersController` create a new action named `getLogout`:

```
1  public function getLogout() {
2
3  }
```

Now add in the following snippet of code to log the user out:

```
1  public function getLogout() {
2      Auth::logout();
3      return Redirect::to('users/login')->with('message', 'Your are now logged out!');
4  }
```

Here we call the `Auth::logout()` method, which handles logging the user out for us. Afterwards, we redirect the user back to the login page and give them a flash message letting them know that they have been logged out.



Conclusion

And that concludes this Laravel 4 Authentication tutorial. I hope you've found this helpful in setting up auth for your Laravel apps. If you have any problems or questions, feel free to ask in the comments and I'll try my best to help you out. You can checkout the complete source code for the small demo app that we built throughout this tutorial on Github. Thanks for reading.