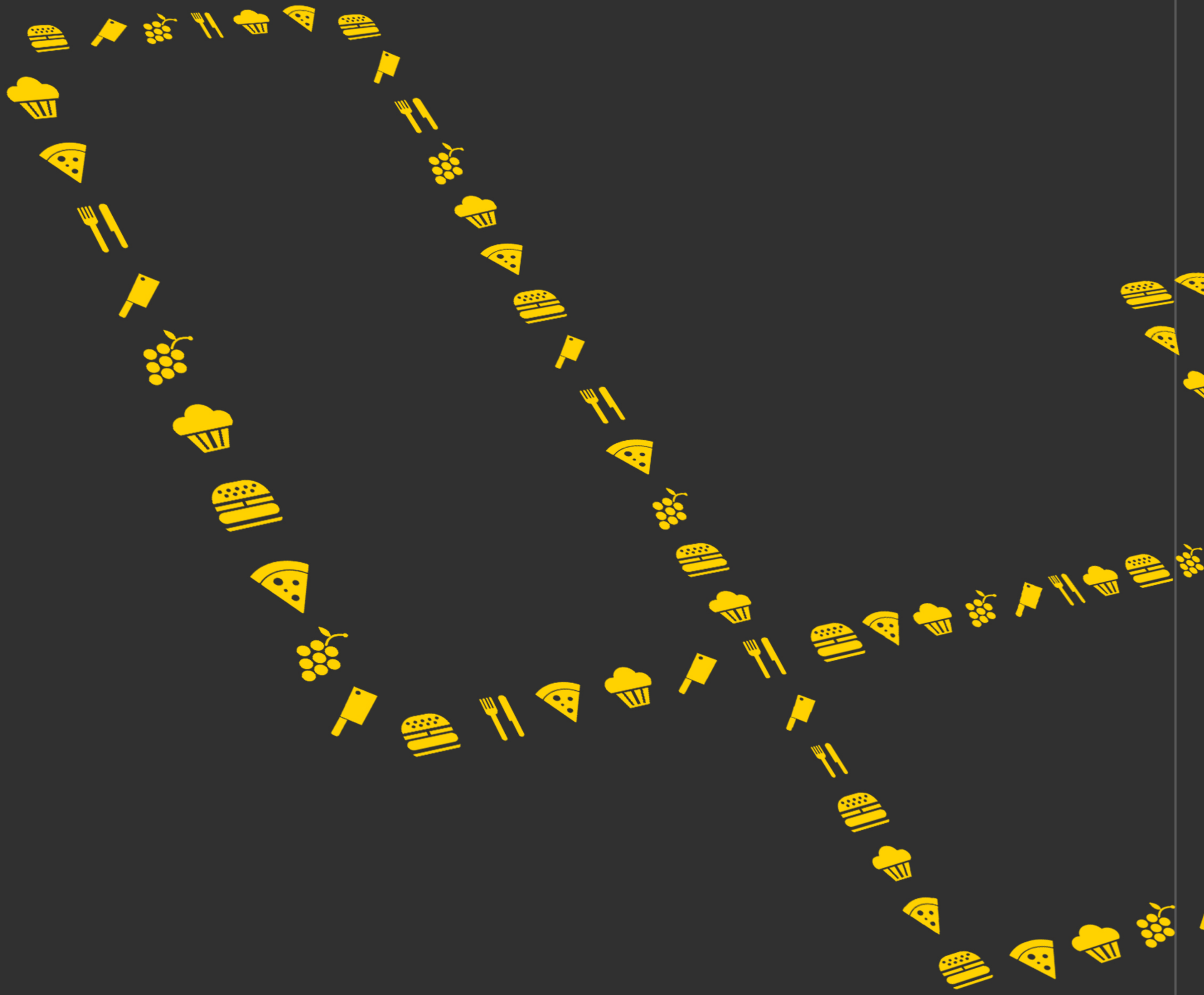

LARAVEL 4
COOKBOOK

BY CHRISTOPHER PITT

Laravel 4 Cookbook

Christopher Pitt and Taylor Otwell

This book is for sale at http://leanpub.com/laravel4cookbook

This version was published on 2013-12-16

Tweet This Book!

Please help Christopher Pitt and Taylor Otwell by spreading the word about this book on Twitter!

The suggested hashtag for this book is #laravel4cookbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#laravel4cookbook

Contents

Installing Laravel 4

Laravel 4 uses Composer to manage its dependencies. You can install Composer by following the instructions at **http://getcomposer.org/doc/00-intro.md#installation-nix**.

Once you have Composer working, make a new directory or navigation to an existing directory and install Laravel 4 with the following command:

```
composer create-project laravel/laravel ./ --prefer-dist
```

If you chose not to install Composer globally (though you really should), then the command you use should resemble the following:

```
php composer.phar create-project laravel/laravel ./ --prefer-dist
```

Both of these commands will start the process of installing Laravel 4. There are many dependencies to be sourced and downloaded; so this process may take some time to finish.

Authentication

If you're anything like me; you've spent a great deal of time building password-protected systems. I used to dread the point at which I had to bolt on the authentication system to a CMS or shopping cart. That was until I learned how easy it was with Laravel 4.

> The code for this chapter can be found at: **https://github.com/formativ/tutorial-laravel-4-authentication**

Configuring The Database

One of the best ways to manage users and authentication is by storing them in a database. The default Laravel 4 authentication mechanisms assume you will be using some form of database storage, and provides two drivers with which these database users can be retrieved and authenticated.

Connection To The Database

To use either of the provided drivers, we first need a valid connection to the database. Set it up by configuring and of the sections in the **app/config/database.php** file. Here's an example of the MySQL database I use for testing:

```php
<?php

return [
    "fetch"       => PDO::FETCH_CLASS,
    "default"     => "mysql",
    "connections" => [
        "mysql" => [
                "driver"    => "mysql",
                "host"      => "localhost",
                "database"  => "tutorial",
                "username"  => "dev",
                "password"  => "dev",
                "charset"   => "utf8",
                "collation" => "utf8_unicode_ci",
                "prefix"    => ""
            ]
        ],
    "migrations" => "migration"
];
```

This file should be saved as **app/config/database.php**.

I have removed comments, extraneous lines and superfluous driver configuration options.

Database Driver

The first driver which Laravel 4 provides is a called **database**. As the name suggests; this driver queries the database directly, in order to determine whether users matching provided credentials exist, and whether the appropriate authentication credentials have been provided.

If this is the driver you want to use; you will need the following database table in the database you have already configured:

```
CREATE TABLE `user` (
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `username` varchar(255) DEFAULT NULL,
    `password` varchar(255) DEFAULT NULL,
    `email` varchar(255) DEFAULT NULL,
    `created_at` datetime DEFAULT NULL,
    `updated_at` datetime DEFAULT NULL,
    PRIMARY KEY (`id`)
) CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Here, and further on, I deviate from the standard of plural database table names. Usually, I would recommend sticking with the standard, but this gave me an opportunity to demonstrate how you can configure database table names in both migrations and models.

Eloquent Driver

The second driver which Laravel 4 provides is called eloquent. Eloquent is the name of the ORM which Laravel 4 also provides, for abstracting model data. It is similar in that it will ultimately query a database to determine whether a user is authentic, but the interface which it uses to make that determination is quite different from direct database queries.

If you're building medium-to-large applications, using Laravel 4, then you stand a good chance of using Eloquent models to represent database objects. It is with this in mind that I will spend some time elaborating on the involvement of Eloquent models in the authentication process.

If you want to ignore all things Eloquent; feel free to skip the following sections dealing with migrations and models.

Creating A Migration

Since we're using Eloquent to manage how our application communicates with the database; we may as well use Laravel 4's database table manipulation tools.

To get started, navigate to the root of your project and type the following command:

```
php artisan migrate:make --table="user" CreateUserTable
```

The **–table="user"** flag matches the **$table=user** property we will define in the **User** model.

This will generate the scaffolding for the users table, which should resemble the following:

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUserTable
extends Migration
{
    public function up()
    {
        Schema::table('user', function(Blueprint $table)
        {
            //
        });
    }
    public function down()
    {
        Schema::table('user', function(Blueprint $table)
        {
            //
        });
    }
}
```

This file should be saved as **app/database/migrations/0000_00_00_000000_CreateUserTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

The file naming scheme may seem odd, but it is for a good reason. Migration systems are designed to be able to run on any server, and the order in which they must run is fixed. All of this is to allow changes to the database to be version-controlled.

The migration is created with just the most basic scaffolding, which means we need to add the fields for the users table:

```php
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateUserTable
extends Migration
{
    public function up()
    {
        Schema::create("user", function(Blueprint $table)
        {
            $table->increments("id");

            $table
                ->string("username")
                ->nullable()
                ->default(null);

            $table
                ->string("password")
                ->nullable()
                ->default(null);

            $table
                ->string("email")
                ->nullable()
                ->default(null);

            $table
                ->dateTime("created_at")
                ->nullable()
                ->default(null);

            $table
                ->dateTime("updated_at")
                ->nullable()
                ->default(null);
        });
    }

    public function down()
    {
        Schema::dropIfExists("user");
    }
}
```

> This file should be saved as **app/database/migrations/0000_00_00_000000_CreateUserTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

Here; we've added fields for id, username, password, date created and date updated. There are methods to shortcut the timestamp fields, but I prefer to add these fields explicitly. All the fields are nullable and their default value is null.

We've also added the drop method, which will be run if the migrations are reversed; which will drop the users table if it exists.

> The shortcut for adding the timestamp fields can be found at: **http://laravel.com/docs/schema#adding-columns**

This migration will work, even if you only want to use the database driver, but it's usually part of a larger setup; including models and seeders.

Creating A Model

Laravel 4 provides a **User** model, with all the interface methods it requires. I have modified it slightly, but the basics are still there…

```
<?php

use Illuminate\Auth\UserInterface;
use Illuminate\Auth\Reminders\RemindableInterface;

class User
extends Eloquent
implements UserInterface, RemindableInterface
{
    protected $table  = "user";
    protected $hidden = ["password"];

    public function getAuthIdentifier()
    {
        return $this->getKey();
    }

    public function getAuthPassword()
    {
        return $this->password;
    }

    public function getReminderEmail()
    {
        return $this->email;
    }
}
```

This file should be saved as **app/models/User.php**.

Note the **$table=user** property we have defined. It should match the table we defined in our migrations.

The **User** model extends **Eloquent** and implements two interfaces which ensure the model is valid for authentication and reminder operations. We'll look at the interfaces later, but its important to note the methods these interfaces require.

Laravel 4 allows the user of either email address or username with which to identify the user, but it is a different field from that which the **getAuthIdentifier()** returns. The **UserInterface** interface does specify the password field name, but this can be changed by overriding/changing the **getAuthPassword()** method.

The **getReminderEmail()** method returns an email address with which to contact the user with a password reset email, should this be required.

You are otherwise free to specify any model customisation, without fear it will break the built-in authentication mechanisms.

Creating A Seeder

Laravel 4 also includes seeding system, which can be used to add records to your database after initial migration. To add the initial users to my project, I have the following seeder class:

```php
<?php

class UserSeeder
extends DatabaseSeeder
{
    public function run()
    {
        $users = [
            [
                "username" => "christopher.pitt",
                "password" => Hash::make("7h3¡MOST!53cu23"),
                "email"    => "chris@example.com"
            ]
        ];

        foreach ($users as $user)
        {
            User::create($user);
        }
    }
}
```

This file should be saved as **app/database/seeds/UserSeeder.php**.

Running this will add my user account to the database, but in order to run this; we need to add it to the main **DatabaseSeeder** class:

```php
<?php

class DatabaseSeeder
extends Seeder
{
    public function run()
    {
        Eloquent::unguard();
        $this->call("UserSeeder");
    }
}
```

This file should be saved as **app/database/seeds/DatabaseSeeder.php**.

Now, when the **DatabaseSeeder** class is invoked; it will seed the users table with my account. If you've already set up your migration and model, and provided valid database connection details, then the following commands should get everything up and running.

```
composer dump-autoload
php artisan migrate
php artisan db:seed
```

The first command makes sure all the new classes we've created are correctly autoloaded. The second creates the database tables specified for the migration. The third seeds the user data into the users table.

Configuring Authentication

The configuration options for the authentication mechanisms are sparse, but they do allow for some customisation.

```
<?php

return [
    "driver"   => "eloquent",
    "model"    => "User",
    "reminder" => [
        "email"  => "email.request",
        "table"  => "token",
        "expire" => 60
    ]
];
```

This file should be saved as **app/config/auth.php**.

All of these settings are important, and most are self-explanatory. The view used to compose the request email is specified by **email ⇒ email.request** and the time in which the reset token will expire is specified by **expire ⇒ 60**.

Pay particular attention to the view specified by **email ⇒ email.request**—it tells Laravel to load the file **app/views/email/request.blade.php** instead of the default **app/views/emails/auth/reminder.blade.php**.

There are various things that would benefit from configuration options; which are currently being hard-coded in the providers. We will look at some of these, as they come up.

Logging In

To allow authentic users to use our application, we're going to build a login page; where users can enter their login details. If their details are valid, they will be redirected to their profile page.

Creating A Layout View

Before we create any of the pages for out application; it would be wise to abstract away all of our layout markup and styling. To this end; we will create a layout view with various includes, using the Blade tempting engine.

First off, we need to create the layout view.

```
<!DOCTYPE html>
<html lang=”en”>
    <head>
        <meta charset="UTF-8" />
        <link
            type="text/css"
            rel="stylesheet"
            href="/css/layout.css" />
        <title>
            Tutorial
        </title>
    </head>
    <body>
        @include("header")
        <div class="content">
            <div class="container">
                @yield("content")
            </div>
        </div>
        @include("footer")
    </body>
</html>
```

This file should be saved as **app/views/layout.blade.php**.

The layout view is mostly standard HTML, with two Blade-specific tags in it. The **@include()** tags tell Laravel to include the views (named in those strings; as **header** and **footer**) from the views directory.

Notice how we've omitted the **.blade.php** extension? Laravel automatically adds this on for us. It also binds the data provided to the layout view to both includes.

The second Blade tag is **yield()**. This tag accepts a section name, and outputs the data stored in that section. The views in our application will extend this layout view; while specifying their own **content** sections so that their markup is embedded in the markup of the layout. You'll see exactly how sections are defined shortly.

```
@section("header")
    <div class="header">
        <div class="container">
            <h1>Tutorial</h1>
        </div>
    </div>
@show
```

This file should be saved as **app/views/header.blade.php**.

The header include file contains two blade tags which, together, instruct Blade to store the markup in the named section, and render it in the template.

```
@section("footer")
    <div class="footer">
        <div class="container">
            Powered by <a href="http://laravel.com/">Laravel</a>
        </div>
    </div>
@show
```

This file should be saved as **app/views/footer.blade.php**.

Similarly, the footer include wraps its markup in a named section and immediately renders it in the template.

You may be wondering why we would need to wrap the markup, in these include files, in sections. We are rendering them immediately, after all. Doing this allows us to alter their contents. We will see this in action soon.

```css
body
{
    margin      : 0;
    padding     : 0 0 50px 0;
    font-family : "Helvetica", "Arial";
    font-size   : 14px;
    line-height : 18px;
    cursor      : default;
}
a
{
    color : #ef7c61;
}
.container
{
    width    : 960px;
    position : relative;
    margin   : 0 auto;
}
.header, .footer
{
    background  : #000;
    line-height : 50px;
    height      : 50px;
    width       : 100%;
    color       : #fff;
}
.header h1, .header a
{
    display : inline-block;
}
.header h1
{
    margin      : 0;
    font-weight : normal;
}
.footer
{
    position : absolute;
    bottom   : 0;
}
.content
```

```
{
    padding : 25px 0;
}
label, input, .error
{
    clear  : both;
    float  : left;
    margin : 5px 0;
}
.error
{
    color : #ef7c61;
}
```

This file should be saved as **public/css/layout.css**.

We finish by adding some basic styles; which we linked to in the **head** element. These alter the default fonts and layout. Your application would still work without them, but it would just look a little messy.

Creating A Login View

The login view is essentially a form; in which users enter their credentials.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "user/login",
        "autocomplete" => "off"
    ]) }}
        {{ Form::label("username", "Username") }}
        {{ Form::text("username", Input::old("username"), [
            "placeholder" => "john.smith"
        ]) }}
        {{ Form::label("password", "Password") }}
        {{ Form::password("password", [
            "placeholder" => "□□□□□□□□□□"
        ]) }}
        {{ Form::submit("login") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

This file should be saved as **app/views/user/login.blade.php**.

The first Blade tag, in the login view, tells Laravel that this view extends the layout view. The second tells it what markup to include in the content section. These tags will form the basis for all the views (other than layout) we will be creating.

We then use **{{** and **}}** to tell Laravel we want the contained code to be interpreted as PHP. We open the form with the **Form::open()** method; providing a route for the form to post to, and optional parameters in the second argument.

We then define two labels and three inputs. The labels accept a name argument, followed by a text argument. The text input accepts a name argument, a default value argument and and optional parameters. The password input accepts a name argument and optional parameters. Lastly, the submit input accepts a name argument and a text argument (similar to labels).

We close out the form with a call to **Form::close()**.

You can find out more about the **Form** methods Laravel offers at: **http://laravel.com/docs/html**

The last part of the login view is where we override the default footer markup (specified in the footer include we created earlier). We use the same section name, but we don't end the section with **@show**. It will already render due to how we defined the include, so we just use **@stop** in the same way as we closed the content section.

We also use the **@parent** Blade tag to tell Laravel we want the markup we defined in the default footer to display. We're not completely changing it, only adding a script tag.

You can find out more about Blade tags at: **http://laravel.com/docs/templates#blade-templating**

The script we included is called polyfill.io. It's a collection of browser shims which allow things like the **placeholder** attribute (which aren't always present in older browsers).

You can find out more about Polyfill.io at: **https://github.com/jonathantneal/polyfill**

Our login view is now complete, but basically useless without the server-side code to accept the input and return a result. Let's get that sorted!

Creating A Login Action

The login action is what glues the authentication logic to the views we have created. If you have been following along, you might have wondered when we were going to try any of this stuff out in a browser. Up to this point; there was nothing telling our application to load that view.

To begin with; we need to add a route for the login action.

```
<?php

Route::any("/", [
    "as"   => "user/login",
    "uses" => "UserController@loginAction"
]);
```

This file should be saved as **app/routes.php**.

The routes file displays a holding page for a new Laravel 4 application, by rendering a view directly. We need to change that to use a controller/action. It's not that we have to—we could just as easily perform the logic in the routes file—it just wouldn't be very tidy.

We specify a name for the route with **as** ⇒ **user/login**, and give it a destination with **uses** ⇒ **UserController@loginAction**. This will match all calls to the default route /, and even has a name which we can use to refer back to this route easily.

Next up, we need to create the controller.

```php
<?php

class UserController
extends Controller
{
    public function loginAction()
    {
        return View::make("user/login");
    }
}
```

This file should be saved as **app/controllers/UserController.php**.

We define the **UserController** (to extend the **Controller** class). In it; we have the single **loginAction()** method we specified in the routes file. All this currently does is render the login view to the browser, but it's enough for us to be able to see our progress!

Authenticating Users

Right, so we've got the form and now we need to tie it into the database so we can authenticate users correctly.

```php
<?php

class UserController
extends Controller
{
    public function loginAction()
    {
        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                echo "Validation passed!";
            }
            else
            {
                echo "Validation failed!";
            }
        }

        return View::make("user/login");
    }
}
```

This file should be saved as **app/controllers/UserController.php**.

Our **UserController** class has changed somewhat. Firstly, we need to act on data that is posted to the **loginAction()** method; and to do that we check the server property **REQUEST_METHOD**. If this value is **POST** we can assume that the form has been posted to this action, and we proceed to the validation phase.

It's also common to see separate get and post actions for the same page. While this makes things a little neater, and avoids the need for checking the **REQUEST_METHOD** property; I prefer to handle both in the same action.

Laravel 4 provides a great validation system, and one of the ways to use it is by calling the **Validator::make()** method. The first argument is an array of data to validate, and the second argument is an array of rules.

We have only specified that the username and password fields are required, but there are many other validation rules (some of which we will use in a while). The **Validator** class also has a **passes()** method, which we use to tell whether the posted form data is valid.

Sometimes it's better to store the validation logic outside of the controller. I often put it in a model, but you could also create a class specifically for handling and validating input.

If you post this form; it will now tell you whether the required fields were supplied or not, but there is a more elegant way to display this kind of message…

```php
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $data = [];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                //
            }
            else
            {
                $data["errors"] = new MessageBag([
                    "password" => [
                        "Username and/or password invalid."
                    ]
                ]);
            }
        }

        return View::make("user/login", $data);
    }
}
```

This file should be saved as **app/controllers/UserController.php**.

With the changes above; we're using the **MessageBag** class to store validation error messages. This is similar to how the Validation class implicitly stores its errors, but instead of showing individual error messages for either username or password; we're showing a single error message for both. Login forms are a little more secure that way!

To display this error message, we also need to change the login view.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "user/login",
        "autocomplete" => "off"
    ]) }}
        {{ Form::label("username", "Username") }}
        {{ Form::text("username", Input::get("username"), [
            "placeholder" => "john.smith"
        ]) }}
        {{ Form::label("password", "Password") }}
        {{ Form::password("password", [
            "placeholder" => "●●●●●●●●●●"
        ]) }}
        @if ($error = $errors->first("password"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::submit("login") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

This file should be saved as **app/views/user/login.blade.php**.

As you can probably see; we've added a check for the existence of the error message, and rendered it within a styled div element. If validation fails, you will now see the error message below the password field.

Redirecting With Input

One of the common pitfalls of forms is how refreshing the page most often re-submits the form. We can overcome this with some Laravel magic. We'll store the posted form data in the session, and redirect back to the login page!

```
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $errors = new MessageBag();

        if ($old = Input::old("errors"))
        {
            $errors = $old;
        }

        $data = [
            "errors" => $errors
        ];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                //
            }
            else
            {
                $data["errors"] = new MessageBag([
                    "password" => [
                        "Username and/or password invalid."
                    ]
                ]);

                $data["username"] = Input::get("username");

                return Redirect::route("user/login")
                    ->withInput($data);
            }
        }

        return View::make("user/login", $data);
    }
}
```

This file should be saved as **app/controllers/UserController.php**.

The first thing we've done is to declare a new **MessageBag** instance. We do this because the view will still check for the errors **MessageBag**, whether or not it has been saved to the session. If it is, however, in the session; we overwrite the new instance we created with the stored instance.

We then add it to the **$data** array so that it is passed to the view, and can be rendered.

If the validation fails; we save the username to the **$data** array, along with the validation errors, and we redirect back to the same route (also using the **withInput()** method to store our data to the session).

Our view remains unchanged, but we can refresh without the horrible form re-submission (and the pesky browser messages that go with it).

Authenticating Credentials

The last step in authentication is to check the provided form data against the database. Laravel handles this easily for us.

```php
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $errors = new MessageBag();

        if ($old = Input::old("errors"))
        {
            $errors = $old;
        }

        $data = [
            "errors" => $errors
        ];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                $credentials = [
                    "username" => Input::get("username"),
                    "password" => Input::get("password")
                ];

                if (Auth::attempt($credentials))
                {
                    return Redirect::route("user/profile");
                }
            }

            $data["errors"] = new MessageBag([
                "password" => [
                    "Username and/or password invalid."
                ]
            ]);

            $data["username"] = Input::get("username");

            return Redirect::route("user/login")
                ->withInput($data);
        }

        return View::make("user/login", $data);
    }
}
```

This file should be saved as **app/controllers/UserController.php**.

We simply need to pass the posted form data (**$credentials**) to the **Auth::attempt()** method and, if the user credentials are valid, the user will be logged in. If valid, we return a redirect to the user profile page.

We have also removed the errors code outside of the else clause. This is so that it will occur both on validation errors as well as authentication errors. The same error message (in the case of login pages) is fine.

Resetting Passwords

The password reset mechanism built into Laravel 4 is great! We're going to set it up so users can reset their passwords just by providing their email address.

Creating A Password Reset View

We need two views for users to be able to reset their passwords. We need a view for them to enter their email address so they can be sent a reset token, and we need a view for them to enter a new password for their account.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "user/request",
        "autocomplete" => "off"
    ]) }}
        {{ Form::label("email", "Email") }}
        {{ Form::text("email", Input::get("email"), [
            "placeholder" => "john@example.com"
        ]) }}
        {{ Form::submit("reset") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

This file should be saved as **app/views/user/request.blade.php**.

This view is similar to the login view, except it has a single field for an email address.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "url"          => URL::route("user/reset") . $token,
        "autocomplete" => "off"
    ]) }}
        @if ($error = $errors->first("token"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::label("email", "Email") }}
        {{ Form::text("email", Input::get("email"), [
            "placeholder" => "john@example.com"
        ]) }}
        @if ($error = $errors->first("email"))
            <div class="error">
```

```
                {{ $error }}
            </div>
        @endif
        {{ Form::label("password", "Password") }}
        {{ Form::password("password", [
            "placeholder" => "●●●●●●●●●●"
        ]) }}
        @if ($error = $errors->first("password"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::label("password_confirmation", "Confirm") }}
        {{ Form::password("password_confirmation", [
            "placeholder" => "●●●●●●●●●●"
        ]) }}
        @if ($error = $errors->first("password_confirmation"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::submit("reset") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

This file should be saved as **app/views/user/reset.blade.php**.

Ok, you get it by now. There's a form with some inputs and error messages. One important thing to note is the change in form action; namely the use of **URL::route()** in combination with a variable assigned to the view. We will set that in the action, so don't worry about it for now.

I've also slightly modified the password token request email, though it remains mostly the same as the default view provided by new Laravel 4 installations.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
    </head>
    <body>
        <h1>Password Reset</h1>
        To reset your password, complete this form:
        {{ URL::route("user/reset") . "?token=" . $token }}
    </body>
</html>
```

This file should be saved as **app/views/email/request.blade.php**.

Remember we changed the configuration options for emailing this view from the default **app/views/emails/auth/reminder.blade.php**.

Creating A Password Reset Action

In order for the actions to be accessible; we need to add routes for them.

```
<?php

Route::any("/", [
    "as"   => "user/login",
    "uses" => "UserController@loginAction"
]);

Route::any("/request", [
    "as"   => "user/request",
    "uses" => "UserController@requestAction"
]);

Route::any("/reset", [
    "as"   => "user/reset",
    "uses" => "UserController@resetAction"
]);
```

This file should be saved as **app/routes.php**.

Remember; the request route is for requesting a reset token, and the reset route is for resetting a password.

We also need to generate the password reset tokens table; using artisan.

```
php artisan auth:reminders
```

This will generate a migration template for the reminder table.

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateTokenTable
extends Migration
{
    public function up()
    {
        Schema::create("token", function(Blueprint $table)
        {
            $table
                ->string("email")
                ->nullable()
                ->default(null);

            $table
                ->string("token")
                ->nullable()
                ->default(null);

            $table
                ->timestamp("created_at")
```

```
                    ->nullable()
                    ->default(null);
        });
    }

    public function down()
    {
        Schema::dropIfExists("token");
    }
}
```

This file should be saved as **app/database/migrations/0000_00_00_000000_CreateTokenTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

I've modified the template slightly, but the basics are all the same. This will create a table with **email**, **token** and **created_at** fields; which the authentication mechanisms use to generate and validate password reset tokens.

With these in place, we can begin to add our password reset actions.

```
public function requestAction()
{
    $data = [
        "requested" => Input::old("requested")
    ];

    if (Input::server("REQUEST_METHOD") == "POST")
    {
        $validator = Validator::make(Input::all(), [
            "email" => "required"
        ]);

        if ($validator->passes())
        {
            $credentials = [
                "email" => Input::get("email")
            ];

            Password::remind($credentials,
                function($message, $user)
                {
                    $message->from("chris@example.com");
                }
            );

            $data["requested"] = true;

            return Redirect::route("user/request")
                ->withInput($data);
        }
    }

    return View::make("user/request", $data);
}
```

This was extracted from **app/controllers/UserController.php**.

The **requestAction()** method validates the posted form data in much the same was as the **loginAction()** method, but instead of passing the form data to **Auth::attempt()**, it passes it to **Password::remind()**. This method accepts an array of credentials (which usually just includes an email address), and also allows an optional callback in which you can customise the email that gets sent out.

```
public function resetAction()
{
    $token = "?token=" . Input::get("token");

    $errors = new MessageBag();

    if ($old = Input::old("errors"))
    {
        $errors = $old;
    }

    $data = [
        "token"  => $token,
        "errors" => $errors
    ];

    if (Input::server("REQUEST_METHOD") == "POST")
    {
        $validator = Validator::make(Input::all(), [
            "email"                 => "required|email",
            "password"              => "required|min:6",
            "password_confirmation" => "same:password",
            "token"                 => "exists:token,token"
        ]);

        if ($validator->passes())
        {
            $credentials = [
                "email" => Input::get("email")
            ];

            Password::reset($credentials,
                function($user, $password)
                {
                    $user->password = Hash::make($password);
                    $user->save();

                    Auth::login($user);
                    return Redirect::route("user/profile");
                }
            );
        }

        $data["email"] = Input::get("email");

        $data["errors"] = $validator->errors();

        return Redirect::to(URL::route("user/reset") . $token)
            ->withInput($data);
    }

    return View::make("user/reset", $data);
}
```

This was extracted from **app/controllers/UserController.php**.

The **resetAction()** method is much the same. We begin it by creating the token query string (which we use for redirects, to maintain the token in all states of the reset page). We fetch old error messages, as we did for the login page, and we validate the posted form data.

If all the data is valid, we pass it to **Password::reset()**. The second argument is the logic used to update the user's database record. We're updating the password, saving the record and then automatically logging the user in.

If all of that went down without a hitch; we redirect to the profile page. If not; we redirect back to the reset page, passing along the error messages.

There is one strange thing about the authentication mechanisms here; the password/token field names are hard-coded and there is hard-coded validation built into the **Password::reset()** function which does not use the **Validation** class. So long as your field names are **password**, **password_-confirmation** and **token**, and your password is longer than 6 characters, you shouldn't notice this strange thing.

Alternatively, you can modify field names and validation applied in the **vendor/laravel/frame-work/src/Illuminate/Auth/Reminders/PasswordBroker.php** file or implement your own **Re-minderServiceProvider** to replace that which Laravel 4 provides. The details for both of those approaches are beyond the scope of this tutorial. You can find details for creating service providers in Taylor Otwell's excellent book, at: **https://leanpub.com/laravel**

As I mentioned before, you can set the amount of time after which the password reset tokens expire; in the **app/config/auth.php** file.

You can find out more about the authentication methods at: **http://laravel.com/docs/security#authenticating-users**

You can find out more about the mail methods at: **http://laravel.com/docs/mail**

Working With Authenticated Users

Ok. We've got login and password reset under our belt. The final part of this tutorial is for us to use the user session data in our application, and protect unauthenticated access to secure parts of our application.

Creating A Profile Page

To show off some of the user session data we have access to; we've going to implement the profile view.

```
@extends("layout")
@section("content")
    <h2>Hello {{ Auth::user()->username }}</h2>
    <p>Welcome to your sparse profile page.</p>
@stop
```

This file should be saved as **app/views/user/profile.blade.php**.

This incredibly sparse profile page shows off a single thing; you can get data from the user model by accessing object returned by the **Auth::user()** method. Any fields you have defined on this model (or database table) are accessible in this way.

```
public function profileAction()
{
    return View::make("user/profile");
}
```

This was extracted from **app/controllers/UserController.php**.

The **profileAction()** method is just a simple as the view. We don’t need to pass any data to the view, or even get hold of the user session using any special code. **Auth::user()** does it all!

In order for this page to be accessible, we do need to add a route for it. We’re going to do that in a minute; but now would be a good time to talk about protecting sensitive pages of our application…

Creating Filters

Laravel 4 includes a filters file, in which we can define filters to run for single (or even groups of) routes.

```php
<?php

Route::filter("auth", function()
{
    if (Auth::guest())
    {
        return Redirect::route("user/login");
    }
});

Route::filter("guest", function()
{
    if (Auth::check())
    {
        return Redirect::route("user/profile");
    }
});

Route::filter("csrf", function()
{
    if (Session::token() != Input::get("_token"))
    {
        throw new Illuminate\Session\TokenMismatchException;
    }
});
```

This file should be saved as **app/filters.php**.

The first filter is for routes (or pages if you prefer) for which a user must be authenticated. The second is for the exact opposite; for which users must not be authenticated. The last filter is one that we have been using all along.

When we use the **Form::open()** method; Laravel automatically adds a hidden field to our forms. This field contains a special security token which is checked each time a form is submitted. You don't really need to understand why this is more secure…

…but if you want to, read this: **http://blog.ircmaxell.com/2013/02/preventing-csrf-attacks.html**

In order to apply these filters, we need to modify our routes file.

```php
<?php

Route::group(["before" => "guest"], function()
{
    Route::any("/", [
        "as"   => "user/login",
        "uses" => "UserController@loginAction"
    ]);

    Route::any("/request", [
        "as"   => "user/request",
        "uses" => "UserController@requestAction"
    ]);

    Route::any("/reset", [
        "as"   => "user/reset",
        "uses" => "UserController@resetAction"
    ]);
});

Route::group(["before" => "auth"], function()
{
    Route::any("/profile", [
        "as"   => "user/profile",
        "uses" => "UserController@profileAction"
    ]);

    Route::any("/logout", [
        "as"   => "user/logout",
        "uses" => "UserController@logoutAction"
    ]);
});
```

This file should be saved as **app/routes.php**.

To protect parts of our application, we group groups together with the **Route::group()** method. The first argument lets us specify which filters to apply to the enclosed routes. We want to group all of our routes in which users should not be authenticated; so that users don't see them when logged in. We do the opposite for the profile page because only authenticated users should get to see their profile pages.

Creating A Logout Action

To test these new security measures out (and to round off the tutorial) we need to create a **logoutAction()** method and add links to the header so that users can log out.

```
public function logoutAction()
{
    Auth::logout();
    return Redirect::route("user/login");
}
```

This was extracted from **app/controllers/UserController.php**.

The **logoutAction()** method calls the **Auth::logout()** method to close the user session, and redirects back to the login screen. Easy as pie!

This is what the new header include looks like:

```
@section("header")
    <div class="header">
        <div class="container">
            <h1>Tutorial</h1>
            @if (Auth::check())
                <a href="{{ URL::route("user/logout") }}">
                    logout
                </a>
                |
                <a href="{{ URL::route("user/profile") }}">
                    profile
                </a>
            @else
                <a href="{{ URL::route("user/login") }}">
                    login
                </a>
            @endif
        </div>
    </div>
@show
```

This file should be saved as **app/views/header.blade.php**.