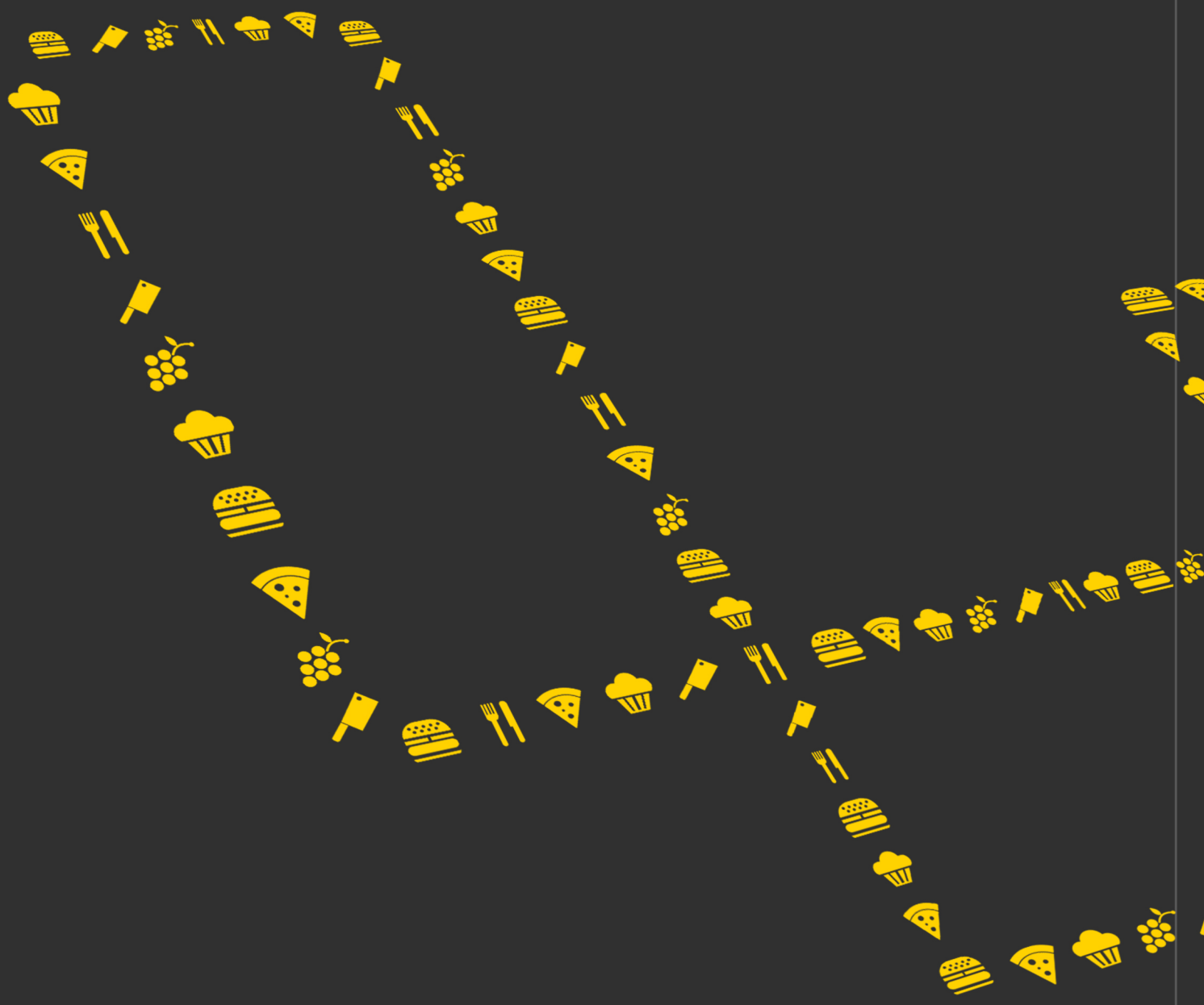

LARAVEL 4
COOKBOOK

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 2014-07-04

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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 . . . . . **1**

Authentication **2**
- Installing Laravel 2
- Configuring The Database 3
  - Connection To The Database 3
  - Database Driver 4
  - Eloquent Driver 5
  - Creating A Migration 5
  - Creating A Model 8
  - Creating A Seeder 9
- Configuring Authentication 11
- Logging In 12
  - Creating A Layout View 12
  - Creating A Login View 15
  - Creating A Login Action 17
  - Authenticating Users 18
  - Redirecting With Input 21
  - Authenticating Credentials 22
- Resetting Passwords 25
  - Creating A Password Reset View 25
  - Creating A Password Reset Action 26
  - Creating Filters 31
  - Creating A Logout Action 33

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**.

> This tutorial requires PHP 5.4 or greater and the PDO/SQLite extension. You also need to have all of the requirements of Laravel 4 met. You can find a list of these at **http://laravel.com/docs/installation#server-requirements**.

Installing Laravel

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 with the following command:

```
 composer create-project laravel/laravel .

Installing laravel/laravel (v4.1.27)
...
```

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 .

Installing laravel/laravel (v4.1.27)
...
```

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.

> The version of Laravel this tutorial is based on is **4.1.27**. It's possible that later versions of Laravel will introduce breaking changes to the code shown here. In that case, clone the Github repository mentioned above, and run `composer install`. The lock file has been included so that Laravel **4.1.27** will be installed for you.

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 components assume you will be using some form of database storage, and they provide 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 SQLite database I use for testing:

```
<?php

return [
  "fetch"       => PDO::FETCH_CLASS,
  "default"     => "sqlite",
  "connections" => [
    "sqlite" => [
      "driver"   => "sqlite",
      "database" => __DIR__ . "/../database/production.sqlite"
    ]
  ],
  "migrations"  => "migration"
];
```

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

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

> Previous versions of this tutorial used a MySQL database for user storage. I decided to switch to SQLite because it's simpler to install and use, when demonstrating Laravel code. If you're using migrations then this shouldn't affect you at all. You can learn more about SQLite at **https://laracasts.com/lessons/maybe-you-should-use-sqlite**.

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 integer PRIMARY KEY NOT null,
  username varchar NOT null,
  password varchar NOT null,
  email varchar NOT null,
  remember_token varchar NOT null,
  created_at datetime NOT null,
  updated_at datetime NOT null
);
```

> 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 provides is called **eloquent**. Eloquent is also the name of the ORM which Laravel 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 using Laravel to build medium-to-large applications, 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 models.

Creating A Migration

Since we're using Eloquent to manage how our application communicates with the database; we may as well use Laravel'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" create_user_table

Created Migration: 2014_05_04_193719_create_user_table
Generating optimized class loader
Compiling common classes
```

> The `--table` flag matches the `$table` 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/****_**_**_******_create_user_table.php`. Yours may be slightly different as the asterisks 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");
      $table->string("password");
      $table->string("email");
      $table->string("remember_token")->nullable();
      $table->timestamps();
    });
  }

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

> This file should be saved as `app/database/migrations/****_**_**_******_create_user_table.php`.

Here; we've added fields for `id`, `username`, `password`, `created_at` and `updated_at`. 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.

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.

> You can learn more about Schema at **http://laravel.com/docs/schema**.

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
<?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 getRememberToken()
  {
    return $this->remember_token;
  }

  public function setRememberToken($value)
  {
    $this->remember_token = $value;
  }

  public function getRememberTokenName()
  {
    return "remember_token";
  }

  public function getReminderEmail()
```

```
39    {
40      return $this->email;
41    }
42  }
```

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

> Note the `$table` 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 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 the `getAuthPassword()` method.

The reminder token methods are used to create and validate account-specific security tokens. The finer details are best left to another lesson…

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 components.

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("7h3iMOST!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`.

The `DatabaseSeeder` class will seed the users table with my account when invoked. 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

Generating autoload files

 php artisan migrate

Migrated: ****_**_**_******_create_user_table

 php artisan db:seed

Seeded: UserSeeder
```

The first command makes sure all the new classes we've created are picked up by the class autoloader. 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 components 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 with the `email` key and the time in which the reset token will expire is specified by the `expire` key.

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

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 template engine.

First off, we need to create the layout view:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link 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.

```
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 {
  clear  : both;
  float  : left;
  margin : 5px 0;
}
```

> 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() }}
    {{ Form::label("username", "Username") }}
    {{ Form::text("username") }}
    {{ Form::label("password", "Password") }}
    {{ Form::password("password") }}
    {{ Form::submit("login") }}
  {{ Form::close() }}
@stop
```

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

The `@extends` tag tells Laravel that this view extends the layout view. The `@section` tag then 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 rendered, as if we were using the `echo` statement. 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**.

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!

> Previous versions of this tutorial included some extra input attributes, and a reference to a JavaScript library that will help you support them. I have removed those attributes to simplify the tutorial, but you can find that script at **http://polyfill.io**.

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@login"
]);
```

> 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've just changed that to use a controller and action.

We specify a name for the route with the `as` key, and give it a destination with the `uses` key. 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

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

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

We define the `UserController`, which extends the `Controller` class. We've also created the `login()` method that 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!

Unless you've got a personal web server set up, you'll probably want to use the built-in web server that Laravel provides. Technically it's just bootstrapping the framework on top of the personal web server that comes bundled with PHP 5.3, but we still need to run the following command to get it working:

```
 php artisan serve

Laravel development server started on http://localhost:8000
```

When you open your browser at **http://localhost:8000**, you should see the login page. If it's not there, you've probably overlooked something leading up to this point.

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 login()
  {
    if ($this->isPostRequest()) {
      $validator = $this->getLoginValidator();

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

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

  protected function isPostRequest()
```

```
22    {
23      return Input::server("REQUEST_METHOD") == "POST";
24    }
25
26    protected function getLoginValidator()
27    {
28      return Validator::make(Input::all(), [
29        "username" => "required",
30        "password" => "required"
31      ]);
32    }
33  }
```

> 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 `login()` 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...

```
public function login()
{
  $data = [];

  if ($this->isPostRequest()) {
    $validator = $this->getLoginValidator();

    if ($validator->passes()) {
      echo "Validation passed!";
    } else {
      $data["error"] = "Username and/or password invalid.";
    }
  }

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

> This was extracted from `app/controllers/UserController.php`.

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() }}
    @if (isset($error))
      {{ $error }}<br />
    @endif
    {{ Form::label("username", "Username") }}
    {{ Form::text("username") }}
    {{ Form::label("password", "Password") }}
    {{ Form::password("password") }}
    {{ Form::submit("login") }}
  {{ Form::close() }}
@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. If validation fails, you will now see the error message above the username 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!

```
1  public function login()
2  {
3    if ($this->isPostRequest()) {
4      $validator = $this->getLoginValidator();
5
6      if ($validator->passes()) {
7        echo "Validation passed!";
8      } else {
9        return Redirect::back()
10         ->withInput()
11         ->withErrors($validator);
12     }
13   }
14
15   return View::make("user/login");
16 }
```

> This was extracted from `app/controllers/UserController.php`.

Instead of assigning errors messages to the view, we redirect back to the same page, passing the posted input data and the validator errors. This also means we will need to change our view:

```
@extends("layout")
@section("content")
  {{ Form::open() }}
    {{ $errors->first("password") }}<br />
    {{ Form::label("username", "Username") }}
    {{ Form::text("username", Input::old("username")) }}
    {{ Form::label("password", "Password") }}
    {{ Form::password("password") }}
    {{ Form::submit("login") }}
  {{ Form::close() }}
@stop
```

We can now hit that refresh button without it asking us for permission to re-submit data.

Authenticating Credentials

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

```
<?php

class UserController
  extends Controller
{
  public function login()
  {
    if ($this->isPostRequest()) {
      $validator = $this->getLoginValidator();

      if ($validator->passes()) {
        $credentials = $this->getLoginCredentials();

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

        return Redirect::back()->withErrors([
          "password" => ["Credentials invalid."]
        ]);
      } else {
        return Redirect::back()
          ->withInput()
```

```
24            ->withErrors($validator);
25        }
26      }
27
28      return View::make("user/login");
29    }
30
31    protected function isPostRequest()
32    {
33      return Input::server("REQUEST_METHOD") == "POST";
34    }
35
36    protected function getLoginValidator()
37    {
38      return Validator::make(Input::all(), [
39        "username" => "required",
40        "password" => "required"
41      ]);
42    }
43
44    protected function getLoginCredentials()
45    {
46      return [
47        "username" => Input::get("username"),
48        "password" => Input::get("password")
49      ];
50    }
51  }
```

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

We simply need to pass the posted form `$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.

Let's set this page up:

```
@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`.

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

> This was extracted from `app/routes.php`.

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

> This was extracted from `app/controllers/UserController.php`.

Once the user is logged in, we can get access to their record by calling the `Auth::user()` method. This returns an instance of the User model (if we're using the Eloquent auth driver, or a plain old PHP object if we're using the Database driver).

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

Resetting Passwords

The password reset components built into Laravel are 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() }}
    {{ Form::label("email", "Email") }}
    {{ Form::text("email", Input::old("email")) }}
    {{ Form::submit("reset") }}
  {{ Form::close() }}
@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() }}
    {{ $errors->first("token") }}<br />
    {{ Form::label("email", "Email") }}
    {{ Form::text("email", Input::get("email")) }}
    {{ $errors->first("email") }}<br />
    {{ Form::label("password", "Password") }}
    {{ Form::password("password") }}
    {{ $errors->first("password") }}<br />
    {{ Form::label("password_confirmation", "Confirm") }}
    {{ Form::password("password_confirmation") }}
    {{ $errors->first("password_confirmation") }}<br />
    {{ Form::submit("reset") }}
  {{ Form::close() }}
@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. 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", compact("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/remin`

Creating A Password Reset Action

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

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

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

> This was extracted from `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-table

Migration created successfully!
Generating optimized class loader
```

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")->index();
      $table->string("token")->index();
      $table->timestamp("created_at");
    });
  }

```

```
18    public function down()
19    {
20      Schema::drop("token");
21    }
22  }
```

> This file should be saved as `app/database/migrations/****_**_**_******_create_token_table.php`.

> Laravel creates the migrations as `app/database/migrations/****_**_**_******_create_password_reminders_table.php` but I have chased something more in=line with the `user` table. So long as your password reminder table matches the `reminder.table` key in your `app/config/auth.php` file, you should be good.

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

```
1   public function request()
2   {
3     if ($this->isPostRequest()) {
4       $response = $this->getPasswordRemindResponse();
5
6       if ($this->isInvalidUser($response)) {
7         return Redirect::back()
8           ->withInput()
9           ->with("error", Lang::get($response));
10      }
11
12      return Redirect::back()
13          ->with("status", Lang::get($response));
14    }
15
16    return View::make("user/request");
17  }
18
19  protected function getPasswordRemindResponse()
20  {
21    return Password::remind(Input::only("email"));
```

```
22  }
23
24  protected function isInvalidUser($response)
25  {
26    return $response === Password::INVALID_USER;
27  }
```

> This was extracted from `app/controllers/UserController.php`.

The main magic in this set of methods is the call to the `Password::remind()` method. This method checks the database for a matching user. If one is found, an email is sent to that user, or else an error message is returned.

We should adjust the reset view to accommodate this error message:

```
1  @extends("layout")
2  @section("content")
3    {{ Form::open() }}
4      @if (Session::get("error"))
5        {{ Session::get("error") }}<br />
6      @endif
7      @if (Session::get("status"))
8        {{ Session::get("status") }}<br />
9      @endif
10     {{ Form::label("email", "Email") }}
11     {{ Form::text("email", Input::old("email")) }}
12     {{ Form::submit("reset") }}
13   {{ Form::close() }}
14 @stop
```

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

When navigating to this route, you should be presented with a form containing an email address field and a submit button. Completing it with an invalid email address should render an error message, while completing it with a valid email address should render a success message. That is, provided the email is sent…

> Laravel includes a ton of configuration options for sending email. I would love to go over them now, but in the interests of keeping this tutorial focussed, I'll simply suggest that you set the `pretend` key to true, in `app/config/mail.php`. This will act as if the application is sending email, though it just skips that step.

```
public function reset($token)
{
  if ($this->isPostRequest()) {
    $credentials = Input::only(
      "email",
      "password",
      "password_confirmation"
    ) + compact("token");

    $response = $this->resetPassword($credentials);

    if ($response === Password::PASSWORD_RESET) {
      return Redirect::route("user/profile");
    }

    return Redirect::back()
      ->withInput()
      ->with("error", Lang::get($response));
  }

  return View::make("user/reset", compact("token"));
}

protected function resetPassword($credentials)
{
  return Password::reset($credentials, function($user, $pass) {
    $user->password = Hash::make($pass);
    $user->save();
  });
}
```

> This was extracted from `app/controllers/UserController.php`.

Similar to the `Password::remind()` method, the `Password::reset()` method accepts an array of user-specific data and does a bunch of magic. That magic includes checking for a valid user account and changing the associated password, or returning an error message.

We need to create the reset view, for this:

```
@extends("layout")
@section("content")
  {{ Form::open() }}
    @if (Session::get("error"))
      {{ Session::get("error") }}<br />
    @endif
    {{ Form::label("email", "Email") }}
    {{ Form::text("email", Input::old("email")) }}
    {{ $errors->first("email") }}<br />
    {{ Form::label("password", "Password") }}
    {{ Form::password("password") }}
    {{ $errors->first("password") }}<br />
    {{ Form::label("password_confirmation", "Confirm") }}
    {{ Form::password("password_confirmation") }}
    {{ $errors->first("password_confirmation") }}<br />
    {{ Form::submit("reset") }}
  {{ Form::close() }}
@stop
```

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

> Tokens expire after 60 minutes, as defined in `app/config/auth.php`.

Creating Filters

Laravel includes a filters file, in which we can define filters to run for single or even groups of routes. The most basic one we're going to look at is the auth filter:

```
<?php

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

> This file should be saved as `app/filters.php`.

This filter, when applied to routes, will check to see whether the user is currently logged in or not. If they are not logged in, they will be directed to the login route.

In order to apply this filter, we need to modify our routes file:

```
<?php

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

Route::group(["before" => "auth"], function() {

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

});

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

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

> This file should be saved as `app/routes.php`.

You'll notice that we've wrapped the profile route in a callback, which executes the auth filter. This means the profile route will only be accessible to authenticated users.

Creating A Logout Action

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

```
public function logout()
{
  Auth::logout();

  return Redirect::route("user/login");
}
```

> This was extracted from `app/controllers/UserController.php`.

Logging a user out is as simple as calling the `Auth::logout()` method. It doesn't clear the session, mind you, but it will make sure our `auth` filter kicks in...

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") }}">
```

```
14            login
15          </a>
16        @endif
17      </div>
18    </div>
19  @show
```

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

Lastly, we should add a route to the logout action:

```
1  Route::any("/logout", [
2    "as"   => "user/logout",
3    "uses" => "UserController@logout"
4  ]);
```

> This was extracted from `app/routes.php`.