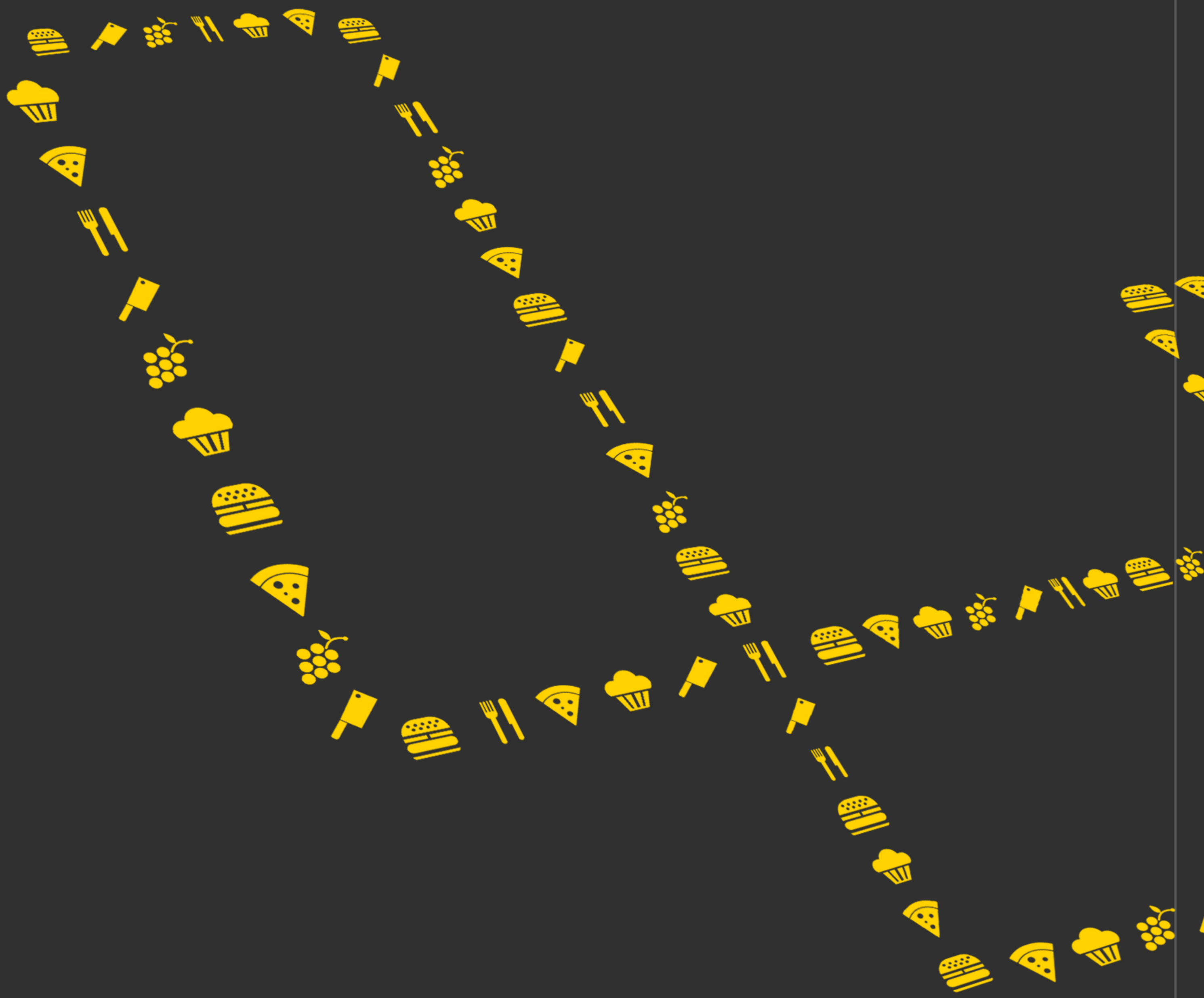

# LARAVEL 4
## COOKBOOK

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 2014-05-04

# 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

# Dedication

I would like to thank Taylor Otwell for the Laravel framework. He may not have written part of this book (in the traditional sense), but without his tireless dedication to Laravel; none of this would have happened. I consider him a co-developer in the code I write daily.

I would like to thank all of my friends as Joe Public[n]. I have never worked anywhere I love more. You give me the freedom and encouragement to create, learn and have fun.

I would like to thank my family for their encouragement, support and general awesomeness.

I would like to thank my wife and children for being patient and loving me even when I'm not loveable.

I would like to thank Jesus. I encourage you to ask me why.

# Forward

Hi, my name is Wayne Ashley Berry and I work with Chris at Joe Public where we write business critical software day in and day out. I've been writing software professionally for years… Chris is the guy I go to when Google doesn't have the answers.

What I love about Chris's work ethic is that he continually pushes the limits of software, frameworks and languages but then manages to hold back and use simple and understandable concepts.

B.B. King once said "Don't use the song to show off your skills, use your skills to show off the song.", Chris is like that Jazz musician who you know could out-play you with one hand but finds immense joy in playing four chord pop tracks.

Each case study in this book comes from hard earned experience. Consider each chapter years of experience, sleepless nights and stressful deadlines distilled into a set of best practices, common sense and good advice.

If you're looking to use Laravel, or even just PHP, for real-world projects then consider this book worth more that its file-size in gold.

# What This Book Teaches

I'm writing this book (and the tutorials) in the hope that people can learn the things I have about Laravel 4. It's not meant as a replacement for any of the great Laravel books, but instead as a complement to the resources, documentation and framework.

This book teaches various aspects of Laravel 4 implementation, configuration and usage; as part of separate projects. The idea is not to demonstrate the only or best way to create any of these projects. It's not to show the only or best way to use Laravel 4. It's simply a different (and subjective) kind of documentation to the modularised version found at: **http://laravel.com/docs**

While this book touches on in the installation and hosting of Laravel applications; it's not an exhaustive reference for how to do these things. There are some instructions; which should be enough to get you up and running, but it assumes you are familiar with how things like LAMP (Linux, Apache, MySQL and PHP) work and are capable of installing and maintaining them.

# Why Write This Book

I was learning how to use Laravel 4 more effectively, and found some subjects which I felt were worth sharing. I picked Medium (which later turned out to be a huge pain) and started putting a tutorial together. A few hours later I hit publish…

Then @laravelphp retweeted a link to the article. I think I spent the rest of the day just watching stats. The tutorial hit Medium's home page. It turns out there are a lot of people who wanted to know about Authentication (in Laravel), and just needed to be exposed to the article through @laravelphp's promotion of it.

Since then; I have been releasing a tutorial every two weeks.

The book grew out of the realisation that; while loads of people were reading the tutorials on Medium, some people weren't happy with the platform.

There are many compelling reasons for me to keep on using Medium to host the tutorials. I don't want to host my own thing because uptime is important, and outages in the night add years onto my life. The simple statistics and text formatting are also great.

I want to stay on Medium, but I also want people to want to read the tutorials and learn from them on other platforms. The book allows both of these things, as well as an important third thing…

The book is also intended as a means to give back to Laravel; in particular the invaluable work of Taylor Otwell. To this end, I have committed to give half of all sales to Taylor. The tutorials will always be free on Medium, and their content will mirror the chapters of this book (with obvious repetition omitted), but by purchasing this book you are helping to fund future Laravel development from him and tutorials from me.

# 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
<?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:

```sql
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()
  {
    return $this->email;
  }
}
```

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("7h3 iMOST!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
<?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
<?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
<?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()
  {
    return Input::server("REQUEST_METHOD") == "POST";
  }

  protected function getLoginValidator()
  {
    return Validator::make(Input::all(), [
      "username" => "required",
      "password" => "required"
    ]);
  }
}
```

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…

```php
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!

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

    if ($validator->passes()) {
      echo "Validation passed!";
    } else {
      return Redirect::back()
        ->withInput()
        ->withErrors($validator);
    }
  }

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

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()
          ->withErrors($validator);
      }
    }

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

  protected function isPostRequest()
  {
    return Input::server("REQUEST_METHOD") == "POST";
  }

  protected function getLoginValidator()
  {
    return Validator::make(Input::all(), [
      "username" => "required",
      "password" => "required"
    ]);
  }

  protected function getLoginCredentials()
  {
    return [
      "username" => Input::get("username"),
      "password" => Input::get("password")
    ];
  }
}
```

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/reminder.blade.php`.

## 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
<?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");
    });
  }

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

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:

```php
public function request()
{
  if ($this->isPostRequest()) {
    $response = $this->getPasswordRemindResponse();

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

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

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

protected function getPasswordRemindResponse()
{
  return Password::remind(Input::only("email"));
}

protected function isInvalidUser($response)
{
  return $response === Password::INVALID_USER;
}
```

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:

```
@extends("layout")
@section("content")
  {{ Form::open() }}
    @if (Session::get("error"))
      {{ Session::get("error") }}<br />
    @endif
    @if (Session::get("status"))
      {{ Session::get("status") }}<br />
    @endif
    {{ 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`.

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.

```php
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
<?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
<?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") }}">
          login
        </a>
      @endif
    </div>
  </div>
@show
```

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

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

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

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

# Access Control List

Previously we looked at how to set up a basic authentication system. In this chapter; we're going to continue to improve the authentication system by adding what's called ACL (Access Control List) to the authentication layer.

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

## Managing Groups

We're going to be creating an interface for adding, modifying and deleting user groups. Groups will be the containers to which we add various users and resources. We'll do that by create a migration and a model for groups, but we're also going to optimise the way we create migrations.

### Refactoring Migrations

We've got a few more migrations to create in this tutorial; so it's a good time for us to refactor our approach to creating them…

```
<?php

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

class BaseMigration
extends Migration
{
    protected $table;

    public function getTable()
    {
        if ($this->table == null)
        {
            throw new Exception("Table not set.");
        }

        return $this->table;
    }

    public function setTable(Blueprint $table)
    {
        $this->table = $table;
        return $this;
    }

    public function addNullable($type, $key)
    {
        $types = [
            "boolean",
            "dateTime",
            "integer",
            "string",
            "text"
        ];

        if (in_array($type, $types))
        {
            $this->getTable()
                ->{$type}($key)
                ->nullable()
                ->default(null);
        }

        return $this;
    }

    public function addTimestamps()
    {
        $this->addNullable("dateTime", "created_at");
        $this->addNullable("dateTime", "updated_at");
        $this->addNullable("dateTime", "deleted_at");
        return $this;
    }

    public function addPrimary()
    {
        $this->getTable()->increments("id");
        return $this;
    }

    public function addForeign($key)
    {
        $this->addNullable("integer", $key);
        $this->getTable()->index($key);
        return $this;
    }

    public function addBoolean($key)
    {
        return $this->addNullable("boolean", $key);
    }

    public function addDateTime($key)
    {
        return $this->addNullable("dateTime", $key);
    }

    public function addInteger($key)
    {
        return $this->addNullable("integer", $key);
    }

    public function addString($key)
    {
        return $this->addNullable("string", $key);
    }

    public function addText($key)
    {
        return $this->addNullable("text", $key);
    }
}
```

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

We're going to base all of our models off of a single **BaseModel** class. This will make it possible for us to reuse a lot of the repeated code we had before.

The **BaseModel** class has a single protected **$table** property, for storing the current **Blueprint** instance we are giving inside our migration callbacks. We have a typical setter for this; and an atypical getter (which throws an exception if **$this->table** hasn't been set). We do this as we need a way to validate that the methods which require a valid **Blueprint** instance have one or throw an exception.

Our **BaseMigration** class also has a factory method for creating fields of various types. If the type provided is one of those defined; a nullable field of that type will be created. This significantly shortens the code we used previously to create nullable fields.

Following this; we have **addPrimary()**, **addForeign()** and **addTimestamps()**. The **addPrimary()** method is a bit clearer than the **increments()** method, the **addForeign()** method adds both a nullable integer field and an index for the foreign key. The **addTimestamps()** method is similar to the **Blueprint**'s **timestamps()** method; except that it also adds the **deleted_at** timestamp field.

Finally; there are a handful of methods which proxy to the **addNullable()** method.

Using these methods, the amount of code required for the migrations we will create (and have already created) is drastically reduced.

```php
<?php

use Illuminate\Database\Schema\Blueprint;

class CreateGroupTable
extends BaseMigration
{
    public function up()
    {
        Schema::create("group", function(Blueprint $table)
        {
            $this
                ->setTable($table)
                ->addPrimary()
                ->addString("name")
                ->addTimestamps();
        });
    }

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

This file should be saved as **app/database/migrations/0000_00_00_000000_CreateGroupTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

The group table has a primary key, timestamp fields (including **created_at**, **updated_at** and **deleted_at**) as well as a name field.

If you're skipping migrations; the following SQL should create the same table structure as the migration:

```
CREATE TABLE `group` (
 `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
 `name` varchar(255) DEFAULT NULL,
 `created_at` datetime DEFAULT NULL,
 `updated_at` datetime DEFAULT NULL,
 `deleted_at` datetime DEFAULT NULL,
 PRIMARY KEY (`id`)
) ENGINE=InnoDB CHARSET=utf8;
```

## Listing Groups

We're going to be creating views to manage group records; more comprehensive than those we created for users previously, but much the same in terms of complexity.

```
@extends("layout")
@section("content")
    @if (count($groups))
        <table>
            <tr>
                <th>name</th>
            </tr>
            @foreach ($groups as $group)
                <tr>
                    <td>{{ $group->name }}</td>
                </tr>
            @endforeach
        </table>
    @else
        <p>There are no groups.</p>
    @endif
    <a href="{{ URL::route("group/add") }}">add group</a>
@stop
```

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

The first view is the index view. This should list all the groups that are in the database. We extend the layout as usual, defining a content block for the markup specific to this page.

The main idea is to iterate over the group records, but before we do that we first check if there are any groups. After all, we don't want to go to the trouble of showing a table if there's nothing to put in it.

If there are groups, we create a (rough) table and iterate over the groups; creating a row for each. We finish off the view by adding a link to create a new group.

```
Route::any("/group/index", [
    "as"   => "group/index",
    "uses" => "GroupController@indexAction"
]);
```

This was extracted from **app/routes.php**.

```
<?php

class Group
extends Eloquent
{
    protected $table = "group";

    protected $softDelete = true;

    protected $guarded = [
        "id",
        "created_at",
        "updated_at",
        "deleted_at"
    ];
}
```

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

```
<?php

class GroupController
extends Controller
{
   public function indexAction()
   {
        return View::make("group/index", [
            "groups" => Group::all()
        ]);
    }
}
```

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

In order to view the index page; we need to define a route to it. We also need to define a model for the group table. Lastly; we render the index view, having passed all the groups to the view. Navigating to this route should now display the message “There are no groups.” as we have yet to add any.

You can test out how the index page looks by adding a group to the database directly.

An important thing to note is the use of **$softDelete**. Laravel 4 provides a new method of ensuring that no data is hastily deleted via Eloquent; so long as this property is set. If **true**; any calls to the **$group->delete()** method will set the **deleted_at** timestamp to the date and time on which the method was invoked. Records with a **deleted_at** timestamp (which is not **null**) will not be returned in normal **QueryBuilder** (including Eloquent) queries.

You can find out more about soft deleting at: **http://laravel.com/docs/eloquent#soft-deleting**

Another important thing to note is the use of **$guarded**. Laravel 4 provides mass assignment protection. What we're doing by specifying this list of fields; is telling Eloquent which fields should not be settable when providing an array of data in the creation of a new **Group** instance.

You can find out more about this mass assignment protection at: **http://laravel.com/docs/eloquent#mass-assignment**

## Adding Groups

We're going to be abstracting much of the validation out of the controllers and into new form classes.

```php
<?php

use Illuminate\Support\MessageBag;

class BaseForm
{
    protected $passes;
    protected $errors;

    public function __construct()
    {
        $errors = new MessageBag();

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

        $this->errors = $errors;
    }

    public function isValid($rules)
    {
        $validator = Validator::make(Input::all(), $rules);
        $this->passes = $validator->passes();
        $this->errors = $validator->errors();
        return $this->passes;
    }

    public function getErrors()
    {
        return $this->errors;
    }

    public function setErrors(MessageBag $errors)
    {
        $this->errors = $errors;
        return $this;
    }

    public function hasErrors()
    {
        return $this->errors->any();
    }

    public function getError($key)
    {
        return $this->getErrors()->first($key);
    }

    public function isPosted()
    {
        return Input::server("REQUEST_METHOD") == "POST";
    }
}
```

This file should be saved as **app/forms/BaseForm.php**.

The **BaseForm** class checks for the error messages we would normally store to flash (session) storage. We would typically pull this data in each action, and now it will happen when each form class instance is created.

The validation takes place in the **isValid()** method, which gets all the input data and compares it to a set of provided validation rules. This will be used later, in **BaseForm** subclasses.

**BaseForm** also has a few methods for managing the **$errors** property, which should always be a **MessageBag** instance. They can be used to set and get the **MessageBag** instance, get an individual message and even tell whether there are any error messages present.

There's also a method to determine whether the request method, for the current request, is POST.

```php
<?php

class GroupForm
extends BaseForm
{
    public function isValidForAdd()
    {
        return $this->isValid([
            "name" => "required"
        ]);
    }

    public function isValidForEdit()
    {
        return $this->isValid([
            "id"   => "exists:group,id",
            "name" => "required"
        ]);
    }

    public function isValidForDelete()
    {
        return $this->isValid([
            "id" => "exists:group,id"
        ]);
    }
}
```

This file should be saved as **app/forms/GroupForm.php**.

The first implementation of **BaseForm** is the **GroupForm** class. It's quite simply by comparison; defining three validation methods. These will be used in their respective actions.

We also need a way to generate not only validation error message markup but also a quicker way to create form markup. Laravel 4 has great utilities for creating form and HTML markup, so let's see how these can be extended.

```
{{ Form::label("name", "Name") }}
{{ Form::text("name", Input::old("name"), [
    "placeholder" => "new group"
]) }}
```

We've already seen this type of Blade template syntax before. The label and text helpers are great for programatically creating the markup we would otherwise have to create; but sometimes it is nice to be able to create our own markup generators for commonly repeated patterns.

What if we, for instance, often use a combination of label, text and error message markup? It would then be ideal for us to create what's called a macro to generate that markup.

```php
<?php

Form::macro("field", function($options)
{
    $markup = "";

    $type = "text";

    if (!empty($options["type"]))
    {
        $type = $options["type"];
    }

    if (empty($options["name"]))
    {
        return;
    }

    $name = $options["name"];

    $label = "";

    if (!empty($options["label"]))
    {
        $label = $options["label"];
    }

    $value = Input::old($name);

    if (!empty($options["value"]))
    {
        $value = Input::old($name, $options["value"]);
    }

    $placeholder = "";

    if (!empty($options["placeholder"]))
    {
        $placeholder = $options["placeholder"];
    }

    $class = "";

    if (!empty($options["class"]))
    {
        $class = " " . $options["class"];
    }

    $parameters = [
        "class"       => "form-control" . $class,
        "placeholder" => $placeholder
    ];

    $error = "";

    if (!empty($options["form"]))
    {
        $error = $options["form"]->getError($name);
    }

    if ($type !== "hidden")
    {
        $markup .= "<div class='form-group";
        $markup .= ($error ? " has-error" : "");
        $markup .= "'>";
    }

    switch ($type)
    {
        case "text":
        {
            $markup .= Form::label($name, $label, [
                "class" => "control-label"
            ]);

          $markup .= Form::text($name, $value, $parameters);

          break;
        }

        case "password":
        {
            $markup .= Form::label($name, $label, [
                "class" => "control-label"
            ]);

            $markup .= Form::password($name, $parameters);

            break;
        }

        case "checkbox":
        {
            $markup .= "<div class='checkbox'>";
            $markup .= "<label>";
            $markup .= Form::checkbox($name, 1, !!$value);
            $markup .= " " . $label;
            $markup .= "</label>";
            $markup .= "</div>";

            break;
        }

        case "hidden":
        {
            $markup .= Form::hidden($name, $value);
            break;
        }
    }

    if ($error)
    {
        $markup .= "<span class='help-block'>";
        $markup .= $error;
        $markup .= "</span>";
    }

    if ($type !== "hidden")
    {
        $markup .= "</div>";
    }

    return $markup;
});
```

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

This macro evaluates an **$options** array, generating a label, input element and validation error message. There's white a lot of checking involved to ensure that all the required data is there, and that optional data affects the generated markup correctly. It supports text inputs, password inputs, checkboxes and hidden fields; but more types can easily be added.

The markup this macro generates is Bootstrap friendly. If you haven't already heard of Bootstrap (where have you been?) then you can find out more about it at: **http://getbootstrap.com/**

To see this in action, we need to include it in the startup processes of the application and then modify the form views to use it:

```
require app_path() . "/macros.php";
```

This was extracted from **app/start/global.php**.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "group/add",
        "autocomplete" => "off"
    ]) }}
        {{ Form::field([
            "name"        => "name",
            "label"       => "Name",
            "form"        => $form,
            "placeholder" => "new group"
        ])}}
        {{ Form::submit("save") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

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

You'll notice how much neater the view is; thanks to the form class handling the error messages for us. This view happens to be relatively short since there's only a single field (name) for groups.

```css
.help-block
{
    float : left;
    clear : left;
}

.form-group.has-error .help-block
{
    color : #ef7c61;
}
```

This was extracted from **public/css/layout.css**.

One last thing we have to do, to get the error messages to look the same as they did before, is to add a bit of CSS to target the Bootstrap-friendly error messages.

With the add view complete; we can create the **addAction()** method:

```php
public function addAction()
{
    $form = new GroupForm();

    if ($form->isPosted())
    {
        if ($form->isValidForAdd())
        {
            Group::create([
                "name" => Input::get("name")
            ]);

            return Redirect::route("group/index");
        }

        return Redirect::route("group/add")->withInput([
            "name"   => Input::get("name"),
            "errors" => $form->getErrors()
        ]);
    }

    return View::make("group/add", [
        "form" => $form
    ]);
}
```

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

You can also see how much simpler our **addAction()** method is; now that we're using the **GroupForm** class. It takes care of retrieving old error messages and handling validation so that we can simply create groups and redirect.

## Editing Groups

The view and action for editing groups is much the same as for adding groups.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "url"          => URL::full(),
        "autocomplete" => "off"
    ]) }}
        {{ Form::field([
            "name"        => "name",
            "label"       => "Name",
            "form"        => $form,
            "placeholder" => "new group",
            "value"       => $group->name
        ]) }}
        {{ Form::submit("save") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

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

The only difference here is the form action we're setting. We need to take into account that a group id will be provided to the edit page, so the URL must be adjusted to maintain this id even after the form is posted. For that; we use the **URL::full()** method which returns the full, current URL.

```php
public function editAction()
{
    $form = new GroupForm();

    $group = Group::findOrFail(Input::get("id"));
    $url   = URL::full();

    if ($form->isPosted())
    {
        if ($form->isValidForEdit())
        {
            $group->name = Input::get("name");
            $group->save();
            return Redirect::route("group/index");
        }

        return Redirect::to($url)->withInput([
            "name"   => Input::get("name"),
            "errors" => $form->getErrors(),
            "url"    => $url
        ]);
    }

    return View::make("group/edit", [
        "form"  => $form,
        "group" => $group
    ]);
}
```

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

In the **editAction()** method; we're still create a new instance of **GroupForm**. Because we're editing a group, we need to get that group to display its data in the view. We do this with Eloquent's **findOrFail()** method; which will cause a 404 error page to be displayed if the id is not found within the database.

The rest of the action is much the same as the **addAction()** method. We'll also need to add the edit route to the **routes.php** file...

```
Route::any("/group/edit", [
    "as"   => "group/edit",
    "uses" => "GroupController@editAction"
]);
```

This was extracted from **app/routes.php**.

## Deleting Groups

There are a number of options we can explore when creating the delete interface, but we'll go with the quickest which is just to present a link on the listing page.

```
@extends("layout")
@section("content")
    @if (count($groups))
        <table>
            <tr>
                <th>name</th>
                <th>&nbsp;</th>
            </tr>
            @foreach ($groups as $group)
                <tr>
                    <td>{{ $group->name }}</td>
                    <td>
                        <a href="{{ URL::route("group/edit") }}?id={{ $group->id \
}}">edit</a>
                        <a href="{{ URL::route("group/delete") }}?id={{ $group->i\
d }}" class="confirm" data-confirm="Are you sure you want to delete this group?">\
delete</a>
                    </td>
                </tr>
            @endforeach
        </table>
    @else
        <p>There are no groups.</p>
    @endif
    <a href="{{ URL::route("group/add") }}">add group</a>
@stop
```

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

We've modified the **group/index** view to include two links; which will redirect users either to the edit page or the delete action. Notice the **class="confirm"** and **data-confirm="..."** attributes we've added to the delete link—we'll use these shortly. We'll also need to add the delete route to the **routes.php** file...

```
Route::any("/group/delete", [
    "as"   => "group/delete",
    "uses" => "GroupController@deleteAction"
]);
```

This was extracted from **app/routes.php**.

Since we've chosen such an easy method of deleting groups, the action is pretty straightforward:

```
public function deleteAction()
{
    $form = new GroupForm();

    if ($form->isValidForDelete())
    {
        $group = Group::findOrFail(Input::get("id"));
        $group->delete();
    }

    return Redirect::route("group/index");
}
```

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

We simply need to find a group with the provided id (using the **findOrFail()** method we saw earlier) and delete it. After that; we redirect back to the listing page. Before we take this for a spin, let's add the following JavaScript:

```
(function($){
    $(".confirm").on("click", function() {
        return confirm($(this).data("confirm"));
    });
}(jQuery));
```

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

```
@section("footer")
    @parent
    <script src="/js/jquery.js"></script>
    <script src="/js/layout.js"></script>
@stop
```

This was extracted from **app/views/group/index.blade.php**.

You'll notice I have linked to **jquery.js** (any recent version will do). The code in **layout.js** adds a click event handler on to every element with **class="confirm"** to prompt the user with the message in **data-confirm**="…". If "OK" is clicked; the callback returns **true** and the browser will redirect to the page on the other end (in this case the **deleteAction()** method on our **GroupController** class). Otherwise the click will be ignored.

## Adding Users And Resources

Next on our list is making a way for us to specify resource information and add users to our groups. Both of these thing will happen on the group edit page; but before we get there we will need to deal with migrations, models and relationships…

### Adding Migrations, Models And Relationships

```php
<?php

use Illuminate\Database\Schema\Blueprint;

class CreateResourceTable
extends BaseMigration
{
    public function up()
    {
        Schema::create("resource", function(Blueprint $table)
        {
            $this
                ->setTable($table)
                ->addPrimary()
                ->addString("name")
                ->addString("pattern")
                ->addString("target")
                ->addBoolean("secure")
                ->addTimestamps();
        });
    }

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

This file should be saved as **app/database/migrations/0000_00_00_000000_CreateResourceTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

We are calling them resources to avoid the name collision with the existing Route class.

If you're skipping migrations; the following SQL should create the same table structure as the migration:

```
CREATE TABLE `resource` (
 `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
 `name` varchar(255) DEFAULT NULL,
 `pattern` varchar(255) DEFAULT NULL,
 `target` varchar(255) DEFAULT NULL,
 `secure` tinyint(1) DEFAULT NULL,
 `created_at` datetime DEFAULT NULL,
 `updated_at` datetime DEFAULT NULL,
 `deleted_at` datetime DEFAULT NULL,
 PRIMARY KEY (`id`)
) ENGINE=InnoDB CHARSET=utf8;
```

The resource table has fields for the things we usually store in our routes file. The idea is that we keep the route information in the database so we can both programatically generate the routes for our application; and so that we can link various routes to groups for controlling access to various parts of our application.

```
<?php

class Resource
extends Eloquent
{
    protected $table = "resource";

    protected $softDelete = true;

    protected $guarded = [
        "id",
        "created_at",
        "updated_at",
        "deleted_at"
    ];

    public function groups()
    {
        return $this->belongsToMany("Group")->withTimestamps();
    }
}
```

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

The Resource model is similar to those we've seen before; but it also specifies a many-to-many relationship (in the **groups()** method). This will allows us to return related groups with **$this->groups**. We'll use that later!

The **withTimestamps()** method will tell Eloquent to update the timestamps of related groups when resources are updated. You can find out more about it at: **http://laravel.com/docs/eloquent#working-with-pivot-tables**

We also need to add the reverse relationship to the **Group** model:

```
public function resources()
{
    return $this->belongsToMany("Resource")->withTimestamps();
}
```

This was extracted from **app/models/Group.php**.

There really is a lot to relationships in Eloquent; more than we have time to cover now. I will be going into more detail about these relationships in future tutorials; exploring the different types and configuration options. For now, this is all we need to complete this chapter.

We can also define relationships for users and groups, as in the following examples:

```php
public function users()
{
    return $this->belongsToMany("User")->withTimestamps();
}
```

This was extracted from **app/models/Group.php**.

```php
public function groups()
{
    return $this->belongsToMany("Group")->withTimestamps();
}
```

This was extracted from **app/models/User.php**.

Before we're quite done with the database work; we'll also need to remember to set up the pivot tables in which the relationship data will be stored.

```php
<?php

use Illuminate\Database\Schema\Blueprint;

class CreateGroupUserTable
extends BaseMigration
{
    public function up()
    {
        Schema::create("group_user", function(Blueprint $table)
        {
            $this
                ->setTable($table)
                ->addPrimary()
                ->addForeign("group_id")
                ->addForeign("user_id")
                ->addTimestamps();
        });
    }

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

> This file should be saved as **app/database/migrations/0000_00_00_000000_CreateGroupUserTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

```
<?php

use Illuminate\Database\Schema\Blueprint;

class CreateGroupResourceTable
extends BaseMigration
{
    public function up()
    {
        Schema::create("group_resource", function(Blueprint $table)
        {
            $this
                ->setTable($table)
                ->addPrimary()
                ->addForeign("group_id")
                ->addForeign("resource_id")
                ->addTimestamps();
        });
    }

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

This file should be saved as **app/database/migrations/0000_00_00_000000_CreateGroupResourceTable.php**. Yours may be slightly different as the 0's are replaced with other numbers.

We now have a way to manage the data relating to groups; so let's create the views and actions through which we can capture this data.

If you're skipping migrations; the following SQL should create the same table structures as the migrations:

```
CREATE TABLE `group_resource` (
 `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
 `group_id` int(11) DEFAULT NULL,
 `resource_id` int(11) DEFAULT NULL,
 `created_at` datetime DEFAULT NULL,
 `updated_at` datetime DEFAULT NULL,
 `deleted_at` datetime DEFAULT NULL,
 PRIMARY KEY (`id`),
 KEY `group_resource_group_id_index` (`group_id`),
 KEY `group_resource_resource_id_index` (`resource_id`)
) ENGINE=InnoDB CHARSET=utf8;

CREATE TABLE `group_user` (
 `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
 `group_id` int(11) DEFAULT NULL,
 `user_id` int(11) DEFAULT NULL,
 `created_at` datetime DEFAULT NULL,
 `updated_at` datetime DEFAULT NULL,
 `deleted_at` datetime DEFAULT NULL,
 PRIMARY KEY (`id`),
 KEY `group_user_group_id_index` (`group_id`),
 KEY `group_user_user_id_index` (`user_id`)
) ENGINE=InnoDB CHARSET=utf8;
```

## Adding Views

The views we need to create are those in which we will select which users and resources should be assigned to a group.

```
<div class="assign">
    @foreach ($resources as $resource)
        <div class="checkbox">
            {{ Form::checkbox("resource_id[]", $resource->id, $group->resources->\
contains($resource->id)) }}
            {{ $resource->name }}
        </div>
    @endforeach
</div>
```

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

```
<div class="assign">
    @foreach ($users as $user)
        <div class="checkbox">
            {{ Form::checkbox("user_id[]", $user->id, $group->users->contains($us\
er->id)) }}
            {{ $user->username }}
        </div>
    @endforeach
</div>
```

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

These views similarly iterate over resources and users (passed to the group edit view) and render markup for checkboxes. It's important to note the names of the checkbox inputs ending in **[]**—this is the recommended way to passing array-like data in HTML forms.

The first parameter of the **Form::checkbox()** method is the input's name. The second is its value. The third is whether of not the checkbox should initially be checked. Eloquent models provide a useful **contains()** method which searches the related rows for those matching the provided id(s).

```
@extends("layout")
@section("content")
    {{ Form::open([
        "url"          => URL::full(),
        "autocomplete" => "off"
    ]) }}
        {{ Form::field([
            "name"        => "name",
            "label"       => "Name",
            "form"        => $form,
            "placeholder" => "new group",
            "value"       => $group->name
        ])}}
        @include("user/assign")
        @include("resource/assign")
        {{ Form::submit("save") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

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

We've modified the **group/edit** view to include the new assign views. If you try to edit a group, at this point, you might see an error. This is because we still need to pass the users and resources to the view…

```
return View::make("group/edit", [
    "form"      => $form,
    "group"     => $group,
    "users"     => User::all(),
    "resources" => Resource::where("secure", true)->get()
]);
```

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

## Seeding Resources

We return all the users (so that any user can be in any group) and the resources that need to be secure. Right now, that database table is empty, but we can easily create a seeder for it:

```
<?php

class ResourceSeeder
extends DatabaseSeeder
{
    public function run()
    {
        $resources = [
            [
                "pattern" => "/",
                "name"    => "user/login",
                "target"  => "UserController@loginAction",
                "secure"  => false
            ],
            [
                "pattern" => "/request",
                "name"    => "user/request",
                "target"  => "UserController@requestAction",
                "secure"  => false
            ],
            [
                "pattern" => "/reset",
                "name"    => "user/reset",
                "target"  => "UserController@resetAction",
                "secure"  => false
            ],
            [
                "pattern" => "/logout",
                "name"    => "user/logout",
                "target"  => "UserController@logoutAction",
                "secure"  => true
            ],
            [
                "pattern" => "/profile",
                "name"    => "user/profile",
                "target"  => "UserController@profileAction",
                "secure"  => true
            ],
            [
                "pattern" => "/group/index",
                "name"    => "group/index",
                "target"  => "GroupController@indexAction",
                "secure"  => true
            ],
            [
                "pattern" => "/group/add",
                "name"    => "group/add",
                "target"  => "GroupController@addAction",
                "secure"  => true
            ],
            [
                "pattern" => "/group/edit",
                "name"    => "group/edit",
                "target"  => "GroupController@editAction",
                "secure"  => true
            ],
            [
                "pattern" => "/group/delete",
                "name"    => "group/delete",
                "target"  => "GroupController@deleteAction",
                "secure"  => true
            ]
        ];

        foreach ($resources as $resource)
        {
            Resource::create($resource);
        }
    }
}
```

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

We should also add this seeder to the **DatabaseSeeder** class so that the Artisan commands which deal with seeding pick it up:

```php
<?php

class DatabaseSeeder
extends Seeder
{
    public function run()
    {
        Eloquent::unguard();

        $this->call("ResourceSeeder");
        $this->call("UserSeeder");
    }
}
```

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

## Saving Relationships

Now you should be seeing the lists of resources and users when you try to edit a group. We need to save selections when the group is saved; so that we can successfully assign both users and resources to groups.

```
if ($form->isValidForEdit())
{
    $group->name = Input::get("name");
    $group->save();

    $group->users()->sync(Input::get("user_id", []));
    $group->resources()->sync(Input::get("resource_id", []));

    return Redirect::route("group/index");
}
```

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

Laravel 4 provides and excellent method for synchronising related database records—the **sync()** method. You simply provide it with the id(s) of the related records and it makes sure there is a record for each relationship. It couldn't be easier!

Finally, we will add a bit of CSS to make the lists less of a mess…

```
.assign
{
    padding     : 10px 0 0 0;
    line-height : 22px;
}
.checkbox, .assign
{
    float : left;
    clear : left;
}
.checkbox input[type='checkbox']
{
    margin : 0 10px 0 0;
    float  : none;
}
```

This was extracted from **public/css/layout.css**.

Take it for a spin! You will find that the related records are created (in the pivot) tables, and each time you submit it; the edit page will remember the correct relationships and show them back to you.

## Advanced Routes

The final thing we need to do is manage how resources are translated into routes and how the security behaves in the presence of our simple ACL.

```php
<?php

Route::group(["before" => "guest"], function()
{
    $resources = Resource::where("secure", false)->get();

    foreach ($resources as $resource)
    {
        Route::any($resource->pattern, [
            "as"   => $resource->name,
            "uses" => $resource->target
        ]);
    }
});

Route::group(["before" => "auth"], function()
{
    $resources = Resource::where("secure", true)->get();

    foreach ($resources as $resource)
    {
        Route::any($resource->pattern, [
            "as"   => $resource->name,
            "uses" => $resource->target
        ]);
    }
});
```

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

There are some significant changes to the routes file. Firstly, all the routes are being generated from resources. We no longer need to hard-code routes in this file because we can save them in the database.

It's more efficient hard-coding them, and we really should be caching them if we have to read them from the database; but that's the subject of future tutorials—we've running out of time here!

All the "insecure" routes are rendered in the first block—the block in which routes are subject to the **guest** filter. All the "secure" routes are rendered in the secure; where they are subject to the **auth** filter.

```
Route::filter("auth", function()
{
    if (Auth::guest())
    {
        return Redirect::route("user/login");
    }
    else
    {
        foreach (Auth::user()->groups as $group)
        {
            foreach ($group->resources as $resource)
            {
                $path = Route::getCurrentRoute()->getPath();

                if ($resource->pattern == $path)
                {
                    return;
                }
            }
        }

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

This was extracted from **app/filters.php**.

The new **auth** filter needs not only to make sure the user is authenticated, but also that one of the group to which they are assigned has the current route assigned to it also. Users can belong to multiple groups and so can resources; so this is the only (albeit inefficient way) to filter allowed resources from those which the user is not allowed access to.

To test this out; alter the group to which your user account belongs to disallow access to the **group/add** route. When you try to visit it you will be redirected first to the **user/login** route and the not the **user/profile** route.

You need to make sure you're not disallowing access to a route specified in the auth filter. That will probably lead to a redirect loop!

Lastly, we need a way to hide links to disallowed resources…

```php
<?php

if (!function_exists("allowed"))
{
    function allowed($route)
    {
        if (Auth::check())
        {
            foreach (Auth::user()->groups as $group)
            {
                foreach ($group->resources as $resource)
                {
                    if ($resource->name == $route)
                    {
                        return true;
                    }
                }
            }
        }

        return false;
    }
}
```

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

```
require app_path() . "/helpers.php";
```

This was extracted from **app/start/global.php**.

Once we’ve included that **helpers.php** file in the startup processes of our application; we can check whether the authenticated user is allowed access to resources simply by passing the resource name to the **allowed()** method.

The first time you run the migrations (if you’re installing from GitHub); you may see errors relating to the resource table. This is likely caused by the routes.php file trying to load routes from an empty database table before seeding takes place. Try commenting out the code in routes.php until you’ve successfully migrated and seeded your database. There are nicer ways to do this; all of which I will not cover now.

Try this out by wrapping the links of your application in a condition which references this method.

For the sake of brevity; I have not included any examples of this, though many can be found, in the source code, on GitHub.

# Deployment

There are few things which have improved my life quite as much as learning how to create a custom deployment process for my projects. Nothing is worse than having to worry about how to get your files onto a remote server, when you've got an important bug to fix.

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

Deployment processes are one of the most subjective things about development. Everyone's got their own ideas about what should and shouldn't be done. There are sometimes best practises; though these tend only to apply to subsets of the whole process.

Things get even more tricky when it comes to deploying Laravel 4 applications because there aren't really any best practises to speak of, when it comes to working with remote servers and deploying code.

Remember this as you continue—this tutorial isn't the only approach to deploying Laravel 4 applications. It's simply a process I've found works for me.

## Dependencies

Our deployment processes will need to handle JavaScript and CSS files. Assetic is an asset management library which we will use to do all the heavy lifting in this area.

To install it; we need to add two requirements to our composer.json file:

```
"kriswallsmith/assetic"    : "1.2.*@dev",
"toopay/assetic-minifier" : "dev-master"
```

> This was extracted from **composer.json**.

Lastly, we will also be using Jason Lewis' Resource Watcher library, which integrates with Laravel 4's **Filesystem** classes to enable notification of file changes. You'll see why that's useful in a bit…

```
"jasonlewis/resource-watcher" : "dev-master"
```

This was extracted from **composer.json**.

Once these requirements are added to the **composer.json** file, we need to update the vendor folder:

```
composer update
```

We'll look at how to integrate these into our deployment workflow shortly.

## Environment Commands

Environments are a small, yet powerful, aspect of any Laravel 4 application. They primarily allow the specification of machine-based configuration options.

There are a few things you need to know about environments, in Laravel 4:

1. The files contained in the root of **app/config** are merged or overridden by environment-based configuration files.
2. Configuration files that are specific to an environment are stored in folders matching their environment name.
3. Environments are determined by an array specified in **bootstrap/start.php** and are matched according to the name of the machine on which the application is being run.
4. An application can have any number of environments; each with their own configuration files. There can also be multiple machine names (hosts) in each environment. You can have two staging servers, a production server and a testing server (for instance). If their machine names match those in **bootstrap/start.php** then their individual configuration files will be loaded.
5. All Artisan commands can be given an **–env** option which will override the environment settings of the machine on which the commands are run. I'm sure there are other ways in which environments affect application execution, but you get the point: environments are a big-little thing.

## Checking Environments

As I mentioned earlier; environments are usually specified in **boostrap/start.php**. This is probably going to be ok for the 99% of Laravel 4 applications that will ever be made, but we can improve upon it slightly still.

We're going to make the list of environments somewhat dynamic, and load them in a slightly different way to how they are loaded out-the-box.

The first thing we're going to do is learn how to make a command to tell us what the current environment is. Commands are the Laravel 4 way of extending the power and functionality of the Artisan command line tool. There are some commands already available when installing Laravel 4 (and we've seen some of them already, in previous tutorials).

To make our own, we can use an Artisan command:

```
php artisan command:make FooCommand
```

This command will make a new file at **app/commands/FooCommand .php**. Inside this file you will begin to see what commands look like under the hood. Here's an example of the file that gets generated:

```
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;

class FooCommand extends Command {

    /**
    * The console command name.
    *
    * @var string
    */
    protected $name = 'command:name';

    /**
    * The console command description.
    *
    * @var string
    */
    protected $description = 'Command description.';

    /**
    * Create a new command instance.
    *
    * @return void
    */
    public function __construct()
    {
        parent::__construct();
    }

    /**
    * Execute the console command.
    *
    * @return void
    */
    public function fire()
    {
        //
    }

    /**
    * Get the console command arguments.
    *
    * @return array
    */
    protected function getArguments()
    {
        return array(
            array(
                'example',
                InputArgument::REQUIRED,
                'An example argument.'
            ),
        );
    }

    /**
    * Get the console command options.
    *
    * @return array
    */
    protected function getOptions()
    {
        return array(
            array(
                'example',
                null,
                InputOption::VALUE_OPTIONAL,
                'An example option.',
                null
            ),
        );
    }

}
```

This file should be saved as **app/commands/FooCommand.php**.

There are a few things of importance here:

1. The **$name** property is used both to describe the command as well as invoke it. If we change it to **foo**, and register it correctly (as we'll do in a moment), then we would be able to call it with: **php artisan foo**
2. The description property is only descriptive. When all the registered command are displayed (by a call to: **php artisan**) then this description will be shown next to the name of the command.
3. The **fire()** method is where all the action happens. If you want your command to do anything; there is where it needs to get done.
4. The **getArguments()** method should return an array of arguments (or parameters) to the command. If our command was: **php artisan foo bar**, then **bar** would be an argument. Arguments are named (which we will see shortly).
5. The **getOptions()** method should return an array of options (or flags) to the command. If out command was: **php artisan foo –baz**, then **–baz** would be an option. Options are also named (which we will also see shortly). This file gives us a good starting point from which to build our own set of commands.

We begin our commands by creating the **EnvironmentCommand** class:

```php
<?php

use Illuminate\Console\Command;

class EnvironmentCommand
extends Command
{
    protected $name = "environment";

    protected $description = "Lists environment commands.";

    public function fire()
    {
        $this->line(trim("
            <comment>environment:get</comment>
            <info>gets host and environment.</info>
        "));

        $this->line(trim("
            <comment>environment:set</comment>
            <info>adds host to environment.</info>
        "));

        $this->line(trim("
            <comment>environment:remove</comment>
            <info>removes host from environment.</info>
        "));
    }

    protected function getArguments()
    {
        return [];
    }

    protected function getOptions()
    {
        return [];
    }
}
```

This file should be saved as **app/commands/EnvironmentCommand.php**.

The command class begins with us setting a useful name and description for the following commands we will create. The **fire()** method includes three calls to the **line()** method; which prints text to the command line. The **getArguments()** and **getOptions()** methods return empty arrays because we do not expect to handle any arguments or options.

You may have noticed the XML notation within the calls to the line() method. Laravel 4's console library extends Symphony 2's console library; and Symphony 2's console library allows these definitions in order to alter the meaning and appearance of text rendered to the console.

While the appearance will be changed, by using this notation, it's not the best thing to be doing (semantically speaking). The bits in the comment elements aren't comments any more than the bit in the info elements are.

We're simply using it for a refreshing variation in the console text colours. If this sort of jacky behaviour offends your senses, feel free to omit the XML notation altogether!

Before we can run any commands; we need to register them with Artisan:

```
Artisan::add(new EnvironmentCommand);
```

This was extracted from **app/start/artisan.php**.

There's nothing much more to say than; this code registers the command with Artisan, so that it can be invoked. We should be able to call the command now, and it should show us something like the following:

```
 php artisan environment
environment:get gets host and environment.
environment:set adds host to environment.
environment:remove removes host from environment.
```

Congratulations! We've successfully created and registered our first Artisan command. Let's go ahead and make a few more.

To make the first described command (**environment:get**); we're going to subclass the **Environment-Command** class we just created. We'll be adding reusable code in the **EnvironmentCommand** class so it's a means of accessing this code in related commands.

```php
<?php

use Illuminate\Console\Command;

class EnvironmentGetCommand
extends EnvironmentCommand
{
    protected $name = "environment:get";

    protected $description = "Gets host and environment.";

    public function fire()
    {
        $this->line(trim("
            <comment>Host:</comment>
            <info>" . $this->getHost() . "</info>
        "));

        $this->line(trim("
            <comment>Environment:</comment>
            <info>" . $this->getEnvironment() . "</info>
        "));
    }
}
```

This file should be saved as **app/commands/EnvironmentGetCommand.php**.

The **EnvironmentGetCommand** does slightly more than the previous command we made. It fetches the host name and the environment name from functions we must define in the **EnvironmentCommand** class:

```
protected function getHost()
{
    return gethostname();
}

protected function getEnvironment()
{
    return App::environment();
}
```

This was extracted from **app/commands/EnvironmentCommand.php**.

The **gethostname()** function returns the name of the machine on which it is invoked. Similarly; the **App::environment()** method returns the name of the environment in which Laravel is being run.

After registering and running this command, I see the following:

```
□ php artisan environment:get
Host: formativ.local
Environment: local
```

## Setting Environments

The next command is going to allow us to alter these values from the command line (without changing hard-coded values)…

```
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class EnvironmentSetCommand
extends EnvironmentCommand
{
    protected $name = "environment:set";

    protected $description = "Adds host to environment.";

    public function fire()
    {
        $host        = $this->getHost();
        $config      = $this->getConfig();
        $overwrite   = $this->option("host");
        $environment = $this->argument("environment");

        if (!isset($config[$environment]))
        {
            $config[$environment] = [];
        }

        $use = $host;

        if ($overwrite)
        {
            $use = $overwrite;
        }

        if (!in_array($use, $config[$environment]))
        {
            $config[$environment][] = $use;
        }

        $this->setConfig($config);

        $this->line(trim("
            <info>Added</info>
            <comment>" . $use . "</comment>
            <info>to</info>
            <comment>" . $environment . "</comment>
            <info>environment.</info>
        "));
    }

    protected function getArguments()
    {
        return [
            [
                "environment",
                InputArgument::REQUIRED,
                "Environment to add the host to."
            ]
        ];
    }

    protected function getOptions()
    {
        return [
            [
                "host",
                null,
                InputOption::VALUE_OPTIONAL,
                "Host to add.",
                null
            ]
        ];
    }
}
```

This file should be saved as **app/commands/EnvironmentSetCommand.php**.

The **EnvironmentSetCommand** class' **fire()** method begins by getting the host name and a configuration array (using inherited methods). It also checks for a host option and an environment argument.

If the host option is provided; it will be added to the list of hosts for the provided environment. If no host option is provided; it will default to the machine the code is being executed on.

We also need to add the inherited methods to the **EnvironmentCommand** class:

```
protected function getHost()
{
    return gethostname();
}

protected function getEnvironment()
{
    return App::environment();
}

protected function getPath()
```

```
{
    return app_path() . "/config/environment.php";
}

protected function getConfig()
{
    $environments = require $this->getPath();

    if (!is_array($environments))
    {
        $environments = [];
    }

    return $environments;
}

protected function setConfig($config)
{
    $config = "<?php return " . var_export($config, true) . ";";
    File::put($this->getPath(), $config);
}
```

This was extracted from **app/commands/EnvironmentCommand.php**.

The **getConfig()** method fetches the contents of the **app/config/environment.php** file (a list of hosts per environment) and the **setConfig()** method writes back to it. We use the **var_export()** to re-create the array that's stored in memory; but it's possible to get a more aesthetically-pleasing configuration file. I've customised the **setConfig()** method to match my personal taste:

```
protected function setConfig($config)
{

    $code = "<?php\n\nreturn [";

    foreach ($config as $environment => $hosts)
    {
        $code .= "\n \"" . $environment . "\" => [";

        foreach ($hosts as $host)
        {
            $code .= "\n \"" . $host . "\",";
        }

        $code = trim($code, ",");
        $code .= "\n ],";
    }

    $code = trim($code, ",");
    File::put($this->getPath(), $code . "\n];");
}
```

This was extracted from **app/commands/EnvironmentCommand.php**.

In order for these environments to be of any use to us; we need to replace those defined in **bootstrap/start.php** with the following lines:

```
$env = $app->detectEnvironment(
    require __DIR__ . "/../app/config/environment.php"
);
```

This was extracted from **bootstrap/start.php**.

This ensures that the environments we set (using our environment commands), and not those hard-coded in the **bootstrap/start.php** file are used in determining the current machine environment.

## Unsetting Environments

The last environment command we will create will provide us a way to remove hosts from an environment (in much the same way as they were added):

```php
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class EnvironmentRemoveCommand
extends EnvironmentCommand
{
    protected $name = "environment:remove";

    protected $description = "Removes host from environment.";

    public function fire()
    {
        $host        = $this->getHost();
        $config      = $this->getConfig();
        $overwrite   = $this->option("host");
        $environment = $this->argument("environment");

        if (!isset($config[$environment]))
        {
            $config[$environment] = [];
        }

        $use = $host;

        if ($overwrite)
        {
            $use = $overwrite;
        }

        foreach ($config[$environment] as $index => $item)
        {
            if ($item == $use)
            {
                unset($config[$environment][$index]);
            }
        }

        $this->setConfig($config);

        $this->line(trim("
            <info>Removed</info>
            <comment>" . $use . "</comment>
            <info>from</info>
            <comment>" . $environment . "</comment>
            <info>environment.</info>
        "));
    }

    protected function getArguments()
    {
        return [
            [
                "environment",
                InputArgument::REQUIRED,
                "Environment to remove the host from."
            ]
        ];
    }

    protected function getOptions()
    {
        return [
            [
                "host",
                null,
                InputOption::VALUE_OPTIONAL,
                "Host to remove.",
                null
            ]
        ];
    }
}
```

This file should be saved as **app/commands/EnvironmentRemoveCommand.php**.

It's pretty much the same as the **EnvironmentSetCommand** class, but instead of adding them to the configuration file; we remove them from the configuration file. It uses the same formatter as the **EnvironmentSetCommand** class.

Commands make up a lot of this tutorial. It may, therefore, surprise you to know that there's not much more to them than this. Sure, there are different types (and values) of **InputOption**'s and **InputArgument**'s, but we're not going into that level of detail here.

You can find out more about these at: **http://symfony.com/doc/current/components/console/introdu**

## Asset Commands

There are two parts to managing assets. The first is how they are stored and referenced in views, and the second is how they are combined/minified.

Both of these operations will require a sane method of specifying asset containers and the assets contained therein. For this; we will create a new configuration file (in each environment we will be using):

```php
<?php

return [
    "header-css" => [
        "css/bootstrap.css",
        "css/shared.css"
    ],
    "footer-js" => [
        "js/jquery.js",
        "js/bootstrap.js",
        "js/shared.js"
    ]
];
```

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

```php
<?php

return [
    "header-css" => [
        "css/shared.min.css" => [
            "css/bootstrap.css",
            "css/shared.css"
        ]
    ],
    "footer-js" => [
        "js/shared.min.js" => [
            "js/jquery.js",
            "js/bootstrap.js",
            "js/shared.js"
        ]
    ]
];
```

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

The difference between these environment-based asset configuration files is how the files are combined in the production environment vs. how they are simply listed in the local environment. This makes development easier because you can see unaltered files while still developing and testing; while the production environment will have smaller file sizes.

To use these in our views; we're going to make a form macro. It's not technically what form macros were made for (since we're not using them to make forms) but it's such a convenient/clean method for generating view markup that we're going to use it anyway.

```php
<?php

Form::macro("assets", function($section)
{
    $markup = "";
    $assets = Config::get("asset");

    if (isset($assets[$section]))
    {
        foreach ($assets[$section] as $key => $value)
        {
            $use = $value;

            if (is_string($key))
            {
                $use = $key;
            }

            if (ends_with($use, ".css"))
            {
                $markup .= "<link
                    rel='stylesheet'
                    type='text/css'
                    href='" . asset($use) . "'
                />";
            }

            if (ends_with($use, ".js"))
            {
                $markup .= "<script
                    type='text/javascript'
                    src='" . asset($use) . "'
                ></script>";
            }
        }
    }

    return $markup;
});
```

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

This macro accepts a single parameter—the name of the section—and renders HTML markup elements for each asset file. It doesn't matter if there are a combination of stylesheets and scripts as the applicable tag is rendered for each asset type.

We also need to make sure this file gets included in our application's startup processes:

```
require app_path() . "/macros.php";
```

This was extracted from **app/start/global.php**.

We can now use this in our templates...

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Laravel 4 — Deployment Tutorial</title>
        {{ Form::assets("header-css") }}
    </head>
    <body>
        <h1>Hello World!</h1>
        {{ Form::assets("footer-js") }}
    </body>
</html>
```

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

Now, depending on the environment we have set; we will either see a list of asset files in each section, or single (production) asset files.

You will notice the difference by changing your environment from local to production. Give the environment commands a go—this is the kind of thing they were made for!

## Combining Assets

The simplest asset operation is combining. This is where we put two or more stylesheets or scripts together into a single file. Similarly to how we arranged the environment classes; the asset commands will inherit form a master command:

```php
<?php

use Assetic\Asset\AssetCollection;
use Assetic\Asset\FileAsset;
use Illuminate\Console\Command;

class AssetCommand
extends Command
{
    protected $name = "asset";

    protected $description = "Lists asset commands.";

    public function fire()
    {
        $this->line(trim("
            <comment>asset:combine</comment>
            <info>combines resource files.</info>
        "));

        $this->line(trim("
            <comment>asset:minify</comment>
            <info>minifies resource files.</info>
        "));
    }

    protected function getArguments()
    {
        return [];
    }

    protected function getOptions()
    {
        return [];
    }

    protected function getPath()
    {
        return public_path();
    }

    protected function getCollection($input, $filters = [])
    {
        $path       = $this->getPath();
        $input      = explode(",", $input);
        $collection = new AssetCollection([], $filters);

        foreach ($input as $asset)
        {
            $collection->add(
                new FileAsset($path . "/" . $asset)
            );
        }

        return $collection;
    }

    protected function setOutput($file, $content)
    {
        $path = $this->getPath();
        return File::put($path . "/" . $file, $content);
    }
}
```

This file should be saved as **app/commands/AssetCommand.php**.

Here's where we make use of Assetic. Among the many utilities Assetic provides; the **AssetCollection** and **FileAsset** classes will be out main focus. The **getCollection()** method accepts a comma-delimited list of asset files (relative to the **public** folder) and returns a populated **AssetCollection** instance.

```php
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class AssetCombineCommand
extends AssetCommand
{
    protected $name = "asset:combine";

    protected $description = "Combines resource files.";

    public function fire()
    {
        $input    = $this->argument("input");
        $output   = $this->option("output");
        $combined = $this->getCollection($input)->dump();

        if ($output)
        {
            $this->line(trim("
                <info>Successfully combined</info>
                <comment>" . $input . "</comment>
                <info>to</info>
                <comment>" . $output . "</comment>
                <info>.</info>
            "));

            $this->setOutput($output, $combined);
        }
        else
        {
            $this->line($combined);
        }
    }

    protected function getArguments()
    {
        return [
            [
                "input",
                InputArgument::REQUIRED,
                "Names of input files."
            ]
        ];
    }

    protected function getOptions()
    {
        return [
            [
                "output",
                null,
                InputOption::VALUE_OPTIONAL,
                "Name of output file.",
                null
            ]
        ];
    }
}
```

This file should be saved as **app/commands/AssetCombineCommand.php**.

The **AssetCombineCommand** class expects the aforementioned list of assets and will output the results to console, by default. If we want to save the results to a file we can provide the **–output** option with a path to the combined file.

You can learn more about Assetic at: **https://github.com/kriswallsmith/assetic/**

## Minifying Assets

Minifying asset files is just as easy; thanks to the Assetic-Minifier filters. Assetic provides filters which can be used to transform the output of asset files.

Usually some of these filters depend on third-party software being installed on the server, or things like Java applets. Fortunately, the Assetic-Minifier library provides pure PHP alternatives to CssMin and JSMin filters (which would otherwise need additional software installed).

```php
<?php

use Minifier\MinFilter;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class AssetMinifyCommand
extends AssetCommand
{
    protected $name = "asset:minify";

    protected $description = "Minifies resource files.";

    public function fire()
    {
        $type    = $this->argument("type");
        $input   = $this->argument("input");
        $output  = $this->option("output");
        $filters = [];

        if ($type == "css")
        {
            $filters[] = new MinFilter("css");
        }

        if ($type == "js")
        {
            $filters[] = new MinFilter("js");
        }

        $collection = $this->getCollection($input, $filters);
        $combined   = $collection->dump();

        if ($output)
        {
            $this->line(trim("
                <info>Successfully minified</info>
                <comment>" . $input . "</comment>
                <info>to</info>
                <comment>" . $output . "</comment>
                <info>.</info>
            "));

            $this->setOutput($output, $combined);
        }
        else
        {
            $this->line($combined);
        }
    }

    protected function getArguments()
    {
        return [
            [
                "type",
                InputArgument::REQUIRED,
                "Code type."
            ],
            [
                "input",
                InputArgument::REQUIRED,
                "Names of input files."
            ]
        ];
    }

    protected function getOptions()
    {
        return [
            [
                "output",
                null,
                InputOption::VALUE_OPTIONAL,
                "Name of output file.",
                null
            ]
        ];
    }
}
```

This file should be saved as **app/commands/AssetMinifyCommand.php**.

The minify command accepts two arguments—the type of assets (either **css** or **js**) and the comma-delimited list of asset files to minify. Like the combine command; it will output to console by default, unless the —output option is specified.

You can learn more about Assetic-Minifier at: **https://github.com/toopay/assetic-minifier**

## Building Assets

Commands that accept inputs and outputs are really useful for the combining and minifying asset files, but it's a pain to have to specify inputs and outputs each time (especially when we have gone to the effort of defining asset lists in configuration files). For this purpose; we need a command which will combine and minify all the asset files appropriately.

```php
<?php

use Illuminate\Console\Command;

class BuildCommand
extends Command
{
    protected $name = "build";

    protected $description = "Builds resource files.";

    public function fire()
    {
        $sections = Config::get("asset");

        foreach ($sections as $section => $assets)
        {
            foreach ($assets as $output => $input)
            {
                if (!is_string($output))
                {
                    continue;
                }

                if (!is_array($input))
                {
                   $input = [$input];
                }

                $input = join(",", $input);

                $options = [
                    "--output" => $output,
                    "input"    => $input
                ];

                if (ends_with($output, ".min.css"))
                {
                    $options["type"] = "css";
                    $this->call("asset:minify", $options);
                }
                else if (ends_with($output, ".min.js"))
                {
                    $options["type"] = "js";
                    $this->call("asset:minify", $options);
                }
                else
                {
                    $this->call("asset:combine", $options);
                }
            }
        }
    }
}
```

This file should be saved as **app/commands/BuildCommand.php**.

The **BuildCommand** class fetches the asset lists, from the configuration files, and iterates over them; combining/minifying as needed. It does this by checking file extensions. If the file ends in **.min.js** then the minify command is run in js mode. Files ending in **.min.css** are minified in css mode, and so forth.

The **app/config/*/asset.php** files are environment-based. This means running the build command will build the assets for the current environment. Often this will not be the environment you want to build assets for. I often build assets on my local machine, yet I want assets built for production. When that is the case; I provide the **–env=production** flag to the build command.

## Watching Assets

So we have the tools to combine, minify and even build our asset files. It's a pain to have to remember to do those things all the time (especially when files are being updated at a steady pace), so we're going to take it a step further by adding a file watcher.

Remember the Resource Watcher library we added to **composer.json**? Well we need to add it to the list of service providers:

```
"providers" => [
    // other service providers here
   "JasonLewis\ResourceWatcher\Integration\LaravelServiceProvider"
]
```

This was extracted from **app/config/app.php**.

The Resource Watcher library requires Laravel 4's **Filesystem** library, and this service provider will allow us to utilise dependency injection.

```
<?php

use Illuminate\Console\Command;

class WatchCommand
extends Command
{
    protected $name = "watch";

    protected $description = "Watches for file changes.";

    public function fire()
    {
        $path     = $this->getPath();
        $watcher  = App::make("watcher");
        $sections = Config::get("asset");

        foreach ($sections as $section => $assets)
        {
            foreach ($assets as $output => $input)
            {
                if (!is_string($output))
                {
                    continue;
                }

                if (!is_array($input))
                {
                    $input = [$input];
                }

                foreach ($input as $file)
                {
                    $watch    = $path . "/" . $file;
                    $listener = $watcher->watch($watch);

                    $listener->onModify(function() use (
                        $section,
                        $output,
                        $input,
                        $file
                    )
                    {
                        $this->build(
                            $section,
                            $output,
                            $input,
                            $file
                        );
                    });
                }
            }
        }

        $watcher->startWatch();
    }

    protected function build($section, $output, $input, $file)
    {
        $options = [
            "--output" => $output,
            "input"    => join(",", $input)
        ];

        $this->line(trim("
            <info>Rebuilding</info>
            <comment>" . $output . "</comment>
            <info>after change to</info>
            <comment>" . $file . "</comment>
            <info>.</info>
        "));

        if (ends_with($output, ".min.css"))
        {
            $options["type"] = "css";
            $this->call("asset:minify", $options);
        }
        else if (ends_with($output, ".min.js"))
        {
            $options["type"] = "js";
            $this->call("asset:minify", $options);
        }
        else
        {
            $this->call("asset:combine", $options);
        }
    }

    protected function getArguments()
    {
        return [];
    }

    protected function getOptions()
    {
        return [];
    }

    protected function getPath()
    {
        return public_path();
    }
}
```

This file should be saved as **app/commands/WatchCommand.php**.

The **WatchCommand** class is a step up from the **BuildCommand** class in that it processes asset files similarly. Where it excels is in how it is able to watch the individual files in **app/config/*/asset.php**.

When a **Watcher** targets a specific file (with the **$watcher->watch()** method); it generates a **Listener**. Listeners can have events bound to them (as is the case with the **onModify()** event listener method).

When these files change, the processed asset files they form part of are rebuilt, using the same logic as in the build command.

You have to have the watcher running in order for updates to cascade into combined/minified asset files. Do this by running: **php artisan watch**

You can learn more about the Resource Watcher library at: **https://github.com/jasonlewis/resource-watcher**

## Resource Watcher Integration Bug

While creating this tutorial; I found a bug with the service provider. It relates to class resolution, and was probably caused by a reshuffling of the library. I have since contacted Jason Lewis to inform him of the bug, and submitted a pull request which resolves it.

If you are having issues relating to the Resource Watcher library, try replacing the service provider file with this one:

```php
<?php namespace JasonLewis\ResourceWatcher\Integration;

use Illuminate\Support\ServiceProvider;
use JasonLewis\ResourceWatcher\Tracker;
use JasonLewis\ResourceWatcher\Watcher;

class LaravelServiceProvider extends ServiceProvider {

    /**
    * Indicates if loading of the provider is deferred.
    *
    * @var bool
    */
    protected $defer = false;

    /**
    * Register the service provider.
    *
    * @return void
    */
    public function register()
    {
        $this->app['watcher'] = $this->app->share(function($app)
        {
            $tracker = new Tracker;
            return new Watcher($tracker, $app['files']);
        });
    }

    /**
    * Get the services provided by the provider.
    *
    * @return array
    */
    public function provides()
    {
        return array('watcher');
    }
}
```

This file should be saved as **vendor/jasonlewis/resource-watcher/src/JasonLewis/ResourceWatcher/Integration/LaravelServiceProvider.php**.

## Rsync

Rsync is a file synchronisation utility which we will use to synchronise our distribution folder to a remote server. It requires a valid private/public key and some configuration.

To set up SSH access, for your domain, follow these steps:

1. Back up any keys you already have in ~/**.ssh**
2. Generate a new key with: **ssh-keygen -t rsa -C "your_email@example.com"**
3. Remember the name you set here.
4. Copy the contents of the new public key file (the name from step 3, ending in **.pub**).
5. SSH into your remove server.
6. Add the contents of the new public key file (which you copied in step 3) to ~/**.ssh/authorized_-keys**. Add the following lines to ~/**.ssh/config** (on your local machine):

```
host example.com
    User your_user
    IdentityFile your_key_file
```

This was extracted from ~/**.ssh/config**.

The **your_user** account needs to be the same as the one with which you SSH'd into the remote server and added the authorised key.

The **your_identity_file** is the name from step 3 (not ending in **.pub**).

Now, when you type **ssh example.com** (where **example.com** is the name of the domain you've been accessing with SSH); you should be let in without even having to provide a password. Don't worry—your server is still secure. You've just let it know (ahead of time) what an authentic connection from you looks like.

With this configuration in place; you won't need to do anything tricky in order to get Rsync to work correctly. The hard part is getting a good connection…

## Distribute Command

In order for us to distribute our code, we need to make a copy of it and perform some operations on the copy. This involves optimisation and cleanup.

### Copying Files For Distribution

First, let's make the copy command:

```php
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;

class CopyCommand
extends Command
{
    protected $name = "copy";

    protected $description = "Creates distribution files.";

    public function fire()
    {
        $target = $this->option("target");

        if (!$target)
        {
            $target = "../distribution";
        }

        File::copyDirectory("./", $target);

        $this->line(trim("
            <info>Successfully copied source files to</info>
            <comment>" . realpath($target) . "</comment>
            <info>.</info>
        "));
    }

    protected function getArguments()
    {
        return [];
    }

    protected function getOptions()
    {
        return [
            [
                "target",
                null,
                InputOption::VALUE_OPTIONAL,
                "Distribution path.",
                null
            ]
        ];
    }
}
```

This file should be saved as **app/commands/CopyCommand.php**.

The copy command uses Laravel 4's **File** methods to copy the source directly recursively. It initially targets ../**distribute** but this can be changed with the –**target** option.

It's important that you copy the distribution files to a target outside of your source folder. The copy command copies ./ which means a target inside will lead to an "infinite copy loop" where the command tries to copy the distribution folder into itself an infinite number of times.

To get around this; I guess you could target sources files individually (or in folders). It's likely that you will be running the copy command from a local machine, so there's little harm in copying the distribution files into a directory outside of the one your application files are in.

## Removing Development Files

Next up, we need a command to remove any temporary/development files. We'll start by creating a config file in which these files are listed:

```php
<?php

return [
    "app/storage/cache/*",
    "app/storage/logs/*",
    "app/storage/sessions/*",
    "app/storage/views/*",
    ".DS_Store",
    ".git*",
    ".svn*",
    "public/js/jquery.js",
    "public/js/bootstrap.js",
    "public/js/shared.js",
    "public/css/bootstrap.css",
    "public/css/shared.css"
];
```

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

I've just listed a few common temporary/development files. You can use any syntax that works with PHP's **glob()** function, as this is what Laravel 4 is using behind the scenes.

I'm also removing the development scripts and styles, as these will be built into minified asset files by the build command.

```php
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputOption;

class CleanCommand
extends Command
{
    protected $name = "clean";

    protected $description = "Cleans distribution folder.";

    public function fire()
    {
        $target = $this->option("target");

        if (!$target)
        {
            $target = "../distribution";
        }

        $cleaned = Config::get("clean");

        foreach ($cleaned as $pattern)
        {
            $paths = File::glob($target . "/" . $pattern);

            foreach ($paths as $path)
            {
                if (File::isFile($path))
                {
                    File::delete($path);
                }

                if (File::isDirectory($path))
                {
                    File::deleteDirectory($path);
                }
            }

            $this->line(trim("
                <info>Deleted all files/folders matching</info>
                <comment>" . $pattern . "</comment>
                <info>.</info>
            "));
        }
    }

    protected function getArguments()
    {
        return [];
    }

    protected function getOptions()
    {
        return [
            [
                "target",
                null,
                InputOption::VALUE_OPTIONAL,
                "Distribution path.",
                null
            ]
        ];
    }
}
```

This file should be saved as **app/commands/CleanCommand.php**.

The **CleanCommand** class gets all files matching the patterns in **app/config/clean.php** and deletes them. It handles folders as well.

Be very careful when deleting files. It’s always advisable to make a full backup before you test these kinds of scripts. I nearly nuked all the tutorial code because I didn’t make a backup!

## Synchronising Files To A Remote Server

We’ve set up SSH access and we have code ready for deployment. Before we sync, we should set up a config file for target remote servers:

```php
<?php

return [
    "production" => [
        "url"  => "example.com",
        "path" => "/var/www"
    ]
];
```

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

The normal SSH access we've set up takes care of the username and password, so all we need to be able to configure is the url of the remote server and the path on it to upload the files to.

```php
<?php

use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class DistributeCommand
extends Command
{
    protected $name = "distribute";

    protected $description = "Synchronises files with target.";

    public function fire()
    {
        $host   = $this->argument("host");
        $target = $this->option("target");

        if (!$target)
        {
            $target = "../distribution";
        }

        $url  = Config::get("host." . $host . ".url");
        $path = Config::get("host." . $host . ".path");

        $command = "rsync --verbose --progress --stats --compress --recursive --t\
imes --perms -e ssh " . $target . "/ " . $url . ":" . $path . "/";

        $escaped = escapeshellcmd($command);

        $this->line(trim("
            <info>Synchronizing distribution files to</info>
            <comment>" . $host . " (" . $url . ")</comment>
            <info>.</info
        "));

        exec($escaped, $output);

        foreach ($output as $line)
        {
            if (starts_with($line, "Number of files transferred"))
            {
                $parts = explode(":", $line);

                $this->line(trim("
                    <comment>" . trim($parts[1]) . "</comment>
                    <info>files transferred.</info>
                "));
             }

            if (starts_with($line, "Total transferred file size"))
            {
                $parts = explode(":", $line);
                $this->line(trim("
                    <comment>" . trim($parts[1]) . "</comment>
                    <info>transferred.</info>
                "));
            }
        }
    }

    protected function getArguments()
    {
        return [
            [
                "host",
                InputArgument::REQUIRED,
                "Destination host."
            ]
        ];
    }

    protected function getOptions()
    {
        return [
            [
                "target",
                null,
                InputOption::VALUE_OPTIONAL,
                "Distribution path.",
                null
            ]
        ];
    }
}
```

This file should be saved as **app/commands/DistributeCommand.php**.

The **DistributeCommand** class accepts a host (remote server name) argument and a **–target** option. Like the copy and clean commands; the **–target** option will override the default **../distribute** folder with one of your choosing.

The host argument needs to match a key in your **app/config/host.php** configuration file. It carefully constructs and escapes a shell command instruction rsync to synchronise files from the distribution folder to a path on the remote server.

Once the files have been synchronised, it will inspect the messy output of the rsync command and return number number of bytes and files transferred.

The intricacies of SSH and Rsync are outside the scope of this tutorial. You can learn more about Rsync at: **https://calomel.org/rsync_tips.html**

You can watch the progress of an upload (to the remote server) by connecting via SSH, navigating to the folder and running the command: **du -hs**

## Command Portability

Portability refers to how easily code will run on any environment. Most of the commands created in this tutorial are portable. The parts that aren't too portable are those relating to Rsync. Rsync is a *nix utility, and is therefore not found on Windows machines.

If you find the deploy command gives you headaches; feel free to jump in just before it and deploy the distribution folder (after build, copy and clean commands) by whatever means works for you.

## Preprocessors

Preprocessors are things which convert some intermediate languages into common languages (such as converting Less, SASS, Stylus into CSS). These are often combined with deployment workflows.

Due to time constraints; I've not covered them in this tutorial. If you need to add them then a good place would be as a filter in the **asset:combine** or **asset:minify** commands. You may also want to add another command (to be executed before those two) which preprocesses any of these languages.

## Images

Image optimisation is a huge topic. I might go into detail on how to optimise your images (as part of the deployment process) in a future tutorial. Assetic does provide filters for this sort of thing; so check its documentation if you feel up to the challenge!

# API

I seldom think of MVC in terms of applications which don't really have views. Turns out Laravel 4 is stocked with features which make REST API's a breeze to create and maintain.

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

## Dependencies

One of the goals, of this tutorial, is to speed up our prototyping. To that end; we'll be installing Jeffrey Way's Laravel 4 Generators. This can be done by amending the **composer.json** file to including the following dependency:

```
"way/generators" : "dev-master"
```

> This was extracted from **composer.json**.

We'll also need to add the **GeneratorsServiceProvider** to our app config:

```
"Way\Generators\GeneratorsServiceProvider"
```

> This was extracted from **app/config/app.php**.

You should now see the generate methods when you run artisan.

## Creating Resources With Artisan

Artisan has a few tasks which are helpful in setting up resourceful API endpoints. New controllers can be created with the command:

```
php artisan controller:make EventController
```

New migrations can also be created with the command:

```
php artisan migrate:make CreateEventTable
```

These are neat shortcuts but they're a little limited considering our workflow. What would make us even more efficient is if we had a way to also generate models and seeders. Enter Jeffrey Way's Laravel 4 Generators…

> You can learn more about Laravel 4's built-in commands by running php artisan in your console, or at: **http://laravel.com/docs**.

## Creating Resources With Generators

With the generate methods installed; we can now generate controllers, migrations, seeders and models.

### Generating Migrations

Let's begin with the migrations:

```
php artisan generate:migration create_event_table
```

This simple command will generate the following migration code:

```php
<?php

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

class CreateEventTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
        Schema::create('event', function(Blueprint $table) {
            $table->increments('id');

            $table->timestamps();
        });
    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop('event');
    }
}
```

This file should be saved as **app/database/migrations/0000*00*00_000000_create_event_table.php**.

We've seen these kinds of migrations before, so there's not much to say about this one. The generators allow us to take it a step further by providing field names and types:

```
php artisan generate:migration --fields="name:string, description:text, started_a\
t:timestamp, ended_at:timestamp" create_event_table
```

This command alters the **up()** method previously generated:

```
public function up()
{
    Schema::create('event', function(Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->text('description');
        $table->timestamp('started_at');
        $table->timestamp('ended_at');
        $table->timestamps();
    });
}
```

This was extracted from **app/database/migrations/0000*000*000_0000000_create_event_table.php**.

Similarly, we can create tables for sponsors and event categories:

```
php artisan generate:migration --fields="name:string, description:text" create_ca\
tegory_table

php artisan generate:migration --fields="name:string, url:string, description:tex\
t" create_sponsor_table
These commands generate the following migrations:
```

```
<?php

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

class CreateCategoryTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
        Schema::create('category', function(Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->text('description');
            $table->timestamps();
        });
    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop('category');
    }
}
```

This file should be saved as **app/database/migrations/00000000_000000_create_category_table.php**.

```
<?php

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

class CreateSponsorTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
        Schema::create('sponsor', function(Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('url');
            $table->text('description');
            $table->timestamps();
        });
    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop('sponsor');
    }
}
```

This file should be saved as **app/database/migrations/0000*00*00_000000_create_sponsor_table.php**.

The last couple of migrations we need to create are for pivot tables to connect sponsors and categories to events. Pivot tables are common in HABTM (Has And Belongs To Many) relationships, between database entities.

The command for these is just as easy:

```
php artisan generate:pivot event category

php artisan generate:pivot event sponsor
These commands generate the following migrations:
```

```php
<?php

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

class PivotCategoryEventTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
        Schema::create('category_event', function(Blueprint $table) {
            $table->increments('id');
            $table->integer('category_id')->unsigned()->index();
            $table->integer('event_id')->unsigned()->index();
            $table->foreign('category_id')->references('id')->on('category')->onD\
elete('cascade');
            $table->foreign('event_id')->references('id')->on('event')->onDelete(\
'cascade');
        });
    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop('category_event');
    }
}
```

This file should be saved as **app/database/migrations/00000000_000000_pivot_category_-event_table.php**.

```
<?php

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

class PivotEventSponsorTable extends Migration {

    /**
    * Run the migrations.
    *
    * @return void
    */
    public function up()
    {
        Schema::create('event_sponsor', function(Blueprint $table) {
            $table->increments('id');
            $table->integer('event_id')->unsigned()->index();
            $table->integer('sponsor_id')->unsigned()->index();
            $table->foreign('event_id')->references('id')->on('event')->onDelete(\
'cascade');
            $table->foreign('sponsor_id')->references('id')->on('sponsor')->onDel\
ete('cascade');
        });
    }

    /**
    * Reverse the migrations.
    *
    * @return void
    */
    public function down()
    {
        Schema::drop('event_sponsor');
    }
}
```

This file should be saved as **app/database/migrations/00000000_000000_pivot_event_sponsor_-table.php**.

Pay special attention to the names of the tables in the calls to the **on()** method. I have changed them from plural tables names to singular table names. The templates used to construct these constraints don't seem to take into account the name of the table on which the constrains are being created.

Apart from the integer fields (which we've seen before); these pivot tables also have foreign keys, with constraints. These are common features of relational databases, as they help to maintain referential integrity among database entities.

These migrations specify foreign key constraints. This means the database will require valid foreign keys when you insert and update rows in these tables. Your seeder class needs to provide these valid ID values, or you should remove the constraints (**references('id')**…**onDelete('cascade')**).

With all these migration files created; we have only to migrate them to the database with:

```
php artisan migrate
```

(This assumes you have already configured the database connection details, in the configuration files.)

## Generating Seeders

Seeders are next on our list. These will provide us with starting data; so our API responses don't look so empty.

```
php artisan generate:seed Category
php artisan generate:seed Sponsor
```

These commands will generate stub seeders, and add them to the **DatabaseSeeder** class. I have gone ahead and customised them to include some data (and better formatting):

```php
<?php

class CategoryTableSeeder
extends Seeder
{
    public function run()
    {
        DB::table("category")->truncate();

        $categories = [
            [
                "name"        => "Concert",
                "description" => "Music for the masses.",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ],
            [
                "name"        => "Competition",
                "description" => "Prizes galore.",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ],
            [
                "name"        => "General",
                "description" => "Things of interest.",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ]
        ];

        DB::table("category")->insert($categories);
    }
}
```

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

```php
<?php

class SponsorTableSeeder
extends Seeder
{
    public function run()
    {
        DB::table("sponsor")->truncate();

        $sponsors = [
            [
                "name"        => "ACME",
                "description" => "Makers of quality dynomite.",
                "url"         => "http://www.kersplode.com",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ],
            [
                "name"        => "Cola Company",
                "description" => "Making cola like no other.",
                "url"         => "http://www.cheerioteeth.com",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ],
            [
                "name"        => "MacDougles",
                "description" => "Super sandwiches.",
                "url"         => "http://www.imenjoyingit.com",
                "created_at"  => date("Y-m-d H:i:s"),
                "updated_at"  => date("Y-m-d H:i:s")
            ]
        ];

        DB::table("sponsor")->insert($sponsors);
    }
}
```

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

To get this data into the database, we need to run the seed command:

```
php artisan db:seed
```

If you kept the foreign key constraints (in the pivot table migrations); you will need to modify your seeder classes to include valid foreign keys.

At this point; we should have the database tables set up, and some test data should be in the category and sponsor tables.

## Generating Models

Before we can output data, we need a way to interface with the database. We're going to use models for this purpose, so we should generate some:

```
php artisan generate:model Event

php artisan generate:model Category

php artisan generate:model Sponsor
```

These commands will generate stub models which resemble the following:

```php
<?php

class Event extends Eloquent {

    protected $guarded = array();

    public static $rules = array();
}
```

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

We need to clean these up a bit, and add the relationship data in…

```php
<?php

class Event
extends Eloquent
{
    protected $table = "event";

    protected $guarded = [
        "id",
        "created_at",
        "updated_at"
    ];

    public function categories()
    {
        return $this->belongsToMany("Category", "category_event", "event_id", "ca\
tegory_id");
    }

    public function sponsors()
    {
        return $this->belongsToMany("Sponsor", "event_sponsor", "event_id", "spon\
sor_id");
    }
}
```

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

```php
<?php

class Category
extends Eloquent
{
    protected $table = "category";

    protected $guarded = [
        "id",
        "created_at",
        "updated_at"
    ];

    public function events()
    {
        return $this->belongsToMany("Event", "category_event", "category_id", "ev\
ent_id");
    }
}
```

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

```
<?php

class Sponsor
extends Eloquent
{
    protected $table = "sponsor";

    protected $guarded = [
        "id",
        "created_at",
        "updated_at"
    ];

    public function events()
    {
        return $this->belongsToMany("Event", "event_sponsor", "sponsor_id", "even\
t_id");
    }
}
```

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

As I mentioned before; we've gone with a **belongsToMany()** relationship to connect the entities together. The arguments for each of these is (1) the model name, (2) the pivot table name, (3) the local key and (4) the foreign key.

I refer to them as the local and foreign keys but they are actually both foreign keys on the pivot table. Think of local as the key closest to the model in which the relationship is defined and the foreign as the furthest.

Our models' $table property should match what's specified in the migrations and seeders.

The last step in creating our API is to create the client-facing controllers.

## Generating Controllers

The API controllers are different from those you might typically see, in a Laravel 4 application. They don't load views; rather they respond to the requested content type. They don't typically cater for multiple request types within the same action. They are not concerned with interface; but rather translating and formatting model data.

Creating them is a bit more tricky than the other classes we've done so far:

```
php artisan generate:controller EventController
```

The command isn't much different, but the generated file is far from ready:

```
<?php
class EventController extends BaseController {

    /**
    * Display a listing of the resource.
    *
    * @return Response
    */
    public function index()
    {
        return View::make('events.index');
    }

    /**
    * Show the form for creating a new resource.
    *
    * @return Response
    */
    public function create()
    {
        return View::make('events.create');
    }

    /**
    * Store a newly created resource in storage.
    *
    * @return Response
    */
    public function store()
    {
        //
    }

    /**
    * Display the specified resource.
    *
    * @param int $id
    * @return Response
    */
    public function show($id)
    {
        return View::make('events.show');
    }

    /**
    * Show the form for editing the specified resource.
    *
    * @param int $id
    * @return Response
    */
    public function edit($id)
    {
        return View::make('events.edit');
    }

    /**
    * Update the specified resource in storage.
    *
    * @param int $id
    * @return Response
    */
    public function update($id)
    {
        //
    }

    /**
    * Remove the specified resource from storage.
    *
    * @param int $id
    * @return Response
    */
    public function destroy($id)
    {
        //
    }
}
```

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

We've not going to be rendering views, so we can remove those statements/actions. We're also not going to deal just with integer ID values (we'll get to the alternative shortly).

For now; what we want to do is list events, create them, update them and delete them. Our controller should look similar to the following:

```php
<?php

class EventController
extends BaseController
{
    public function index()
    {
        return Event::all();
    }

    public function store()
    {
        return Event::create([
            "name"        => Input::get("name"),
            "description" => Input::get("description"),
            "started_at"  => Input::get("started_at"),
            "ended_at"    => Input::get("ended_at")
        ]);
    }

    public function show($event)
    {
        return $event;
    }

    public function update($event)
    {
        $event->name        = Input::get("name");
        $event->description = Input::get("description");
        $event->started_at  = Input::get("started_at");
        $event->ended_at    = Input::get("ended_at");
        $event->save();
        return $event;
    }

    public function destroy($event)
    {
        $event->delete();
        return Response::json(true);
    }
}
```

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

We've deleted a bunch of actions and added some simple logic in others. Our controller will list (**index**) events, show them individually, create (**store**) them, update them and delete (**destroy**) them.

We need to return a JSON response differently, for the **destroy()** action, as boolean return values do not automatically map to JSON responses in the same way as collections and models do.

You can optionally specify the template to be used in the creation of seeders, models and controllers. you do this by providing the **–template** option with the path to a Mustache template file. Unfortunately migrations do not have this option. If you find yourself using these generators often enough, and wanting to customise the way in which new class are created; you will probably want to take a look at this option (and the template files in **vendor/way/generators/src/Way/Generators/Generators/templates**).

You can find out more about Jeffrey Way's Laravel 4 Generators at: **https://github.com/JeffreyWay/Laravel-4-Generators**.

## Binding Models To Routes

Before we can access these; we need to add routes for them:

```
Route::model("event", "Event");

Route::get("event", [
    "as"   => "event/index",
    "uses" => "EventController@index"
]);

Route::post("event", [
    "as"   => "event/store",
    "uses" => "EventController@store"
]);

Route::get("event/{event}", [
    "as"   => "event/show",
    "uses" => "EventController@show"
]);

Route::put("event/{event}", [
    "as"   => "event/update",
    "uses" => "EventController@update"
]);

Route::delete("event/{event}", [
    "as"   => "event/destroy",
    "uses" => "EventController@destroy"
]);
```

This was extracted from **app/routes.php**.

There are two important things here:

1. We're binding a model parameter to the routes. What that means is that we are telling Laravel to look for a specific parameter (in this case **event**) and treat the value found as an ID. Laravel will attempt to return a model instance if it finds that parameter in a route.
2. We're using different request methods to do perform different actions on our events. **GET** requests retrieve data, **POST** requests store data. **PUT** requests update data and **DELETE** requests delete data. This is called REST (Representational State Transfer).

Laravel 4 does support the **PATCH** method, though it's undocumented. Many API's support/use the **PUT** method when what they implement is the closer to the **PATCH** method. That's a discussion for elsewhere; but just know you could use both without modification to Laravel 4.

You can find out more about Route Model Binding at: **http://laravel.com/docs/routing#route-model-binding**.

## Troubleshooting Aliases

If you go to the index route; you will probably see an error. It might say something like "Call to undefined method IlluminateEventsDispatcher::all()". This is because there is already a class (or rather an alias) called **Event**. **Event** is the name of our model, but instead of calling the **all()** method on our model; it's trying to call it on the event disputer class baked into Laravel 4.

I've lead us to this error intentionally, to demonstrate how to overcome it if you ever have collisions in your applications. Most everything in Laravel 4 is in a namespace. However, to avoid lots of extra keystrokes; Laravel 4 also offers a configurable list of aliases (in **app/config/app.php**).

In the list of aliases; you will see an entry which looks like this:

```
'Event' => 'Illuminate\Support\Facades\Event',
```

This was extracted from **app/config/app.php**.

I changed the key of that entry to **Events** but you can really call it anything you like.

It's often just easier to change the name of the classes you create than it is to alter the aliases array. There are many Laravel 4 tutorials that will refer to these classes by their alias, so you'll need to keep a mental log-book of the aliases you've changed and where they might be referred to.

## Testing Endpoints

It’s not always easy to test REST API’s simply by using the browser. Often you will need to use an application to perform the different request methods. Thankfully modern *nix systems already have the Curl library, which makes these sorts of tests easier.

You can test the index endpoint with the console command:

```
curl http://dev.tutorial-laravel-4-api/event
```

Your host (domain) name will differ based on your own setup.

You should only be seeing a JSON response - if you’re seeing HTML there’s a good chance it’s a Laravel error page. Check your application logs for more details.

Unless you’ve manually populated the event table, or set up a seeder for it; you should see an empty JSON array. This is a good thing, and also telling of some Laravel 4 magic. Our **index()** action returns a collection, but it’s converted to JSON output when passed in a place where a Response is expected.

Let’s add a new event:

```
curl -X POST -d "name=foo&description=a+day+of+foo&started_at=2013-10-03+09:00&en\
ded_at=2013-10-03+12:00" http://dev.tutorial-laravel-4-api:2080/event
```

..now, when we request the **index()** action, we should see the new event has been added. We can retrieve this event individually with a request similar to this:

```
curl http://dev.tutorial-laravel-4-api:2080/event/1
```

There’s a lot going on here. Remember how we bound the model to a specific parameter name (in **app/routes.php**)? Well Laravel 4 sees that ID value, matches it to the bound model parameter and fetches the record from the database. If the ID does not match any of the records in the database; Laravel will respond with a 404 error message.

If the record is found; Laravel returns the model representation to the action we specified, and we get a model to work with.

Let’s update this event:

```
curl -X PUT -d "name=best+foo&description=a+day+of+the+best+foo&started_at=2013-1\
0-03+10:00&ended_at=2013-10-03+13:00" http://dev.tutorial-laravel-4-api:2080/even\
t/1
```

Notice how all that's changed is the data and the request type—even though we're doing something completely different behind the scenes.

Lastly, let's delete the event:

```
curl -X DELETE http://dev.tutorial-laravel-4-api:2080/event/1
```

Feel free to set the same routes and actions up for categories and sponsors. I've not covered them here, for the sake of time, but they only differ in terms of the fields.

## Authenticating Requests

So far we've left the API endpoints unauthenticated. That's ok for internal use but it would be far more secure if we were to add an authentication layer.

We do this by securing the routes in filtered groups, and checking for valid credentials within the filter:

```
Route::group(["before" => "auth"], function()
{
    // ...routes go here
});
```

This was extracted from **app/routes.php**.

```
Route::filter("auth", function()
{
    // ...get database user

    if (Input::server("token") !== $user->token)
    {
        App::abort(400, "Invalid token");
    }
});
```

This was extracted from **app/filters.php**.

Your choice for authentication mechanisms will greatly affect the logic in your filters. I’ve opted not to go into great detail with regards to how the tokens are generated and users are stored. Ultimately; you can check for token headers, username/password combos or even IP addresses.

What’s important to note here is that we check for this thing (tokens in this case) and if they do not match those stored in user records, we abort the application execution cycle with a 400 error (and message).

You can find out more about filters at: http://laravel.com/docs/routing#route-filters.

## Using Accessors And Mutators

There are times when we need to customise how model attributes are stored and retrieved. Laravel 4 lets us do that by providing specially named methods for accessors and mutators:

```
public function setNameAttribute($value)
{
    $clean = preg_replace("/\W/", "", $value);
    $this->attributes["name"] = $clean;
}

public function getDescriptionAttribute()
{
    return trim($this->attributes["description"]);
}
```

This was extracted from **app/models/Event.php**.

You can catch values, before they hit the database, by creating public **set*Attribute()** methods. These should transform the **$value** in some way and commit the change to the internal **$attributes** array.

You can also catch values, before they are returned, by creating **get*Attribute()** methods.

In the case of these methods; I am removing all non-word characters from the name value, before it hits the database; and trimming the description before it's returned by the property accessor. Getters are also called by the **toArray()** and **toJson()** methods which transform model instances into either arrays or JSON strings.

You can also add attributes to models by creating accessors and mutators for them, and mentioning them in the **$appends** property:

```php
protected $appends = ["hasCategories", "hasSponsors"];

public function getHasCategoriesAttribute()
{
    $hasCategories = $this->categories()->count() > 0;
    return $this->attributes["hasCategories"] = $hasCategories;
}

public function getHasSponsorsAttribute()
{
    $hasSponsors = $this->sponsors()->count() > 0;
    return $this->attributes["hasSponsors"] = $hasSponsors;
}
```

This was extracted from **app/models/Event.php**.

Here we've created two new accessors which check the count for categories and sponsors. We've also added those two attributes to the **$appends** array so they are returned when we list (**index**) all events or specific (**show**) events.

You can find out more about attribute accessor and mutators at: **http://laravel.com/docs/eloquent#accessors-and-mutators**.

## Using Cache

Laravel 4 provides a great cache mechanism. It's configured in the same was as the database:

```php
<?php

return [
    "driver"    => "memcached",
    "memcached" => [
        [
            "host"   => "127.0.0.1",
            "port"   => 11211,
            "weight" => 100
        ]
    ],
    "prefix" => "laravel"
];
```

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

I've configured my cache to use the Memcached provider. This needs to be running on the specified host (at the specified port) in order for it to work.

Installing and running Memcached are outside the scope of this tutorial. It's complicated.

No matter the provider you choose to use; the cache methods work the same way:

```php
public function index()
{
    return Cache::remember("events", 15, function()
    {
        return Event::all();
    });
}
```

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

The **Cache::remember()** method will store the callback return value in cache if it's not already there. We've set it to store the events for 15 minutes.

The primary use for cache is in key/value storage:

```php
public function total()
{
    if (($total = Cache::get("events.total")) == null)
    {
        $total = Event::count();
        Cache::put("events.total", $total, 15);
    }

    return Response::json((int) $total);
}
```

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

You can also invoke this cache on Query Builder queries or Eloquent queries:

```
public function today()
{
    return Event::where(DB::raw("DAY(started_at)"), date("d"))
        ->remember(15)
        ->get();
}
```

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

…we just need to remember to add the **remember()** method before we call the **get()** or **first()** methods.

You can find out more about cache at: **http://laravel.com/docs/cache**.

# Packages

Packages are the recommended way of extending the functionality provided by Laravel 4. They're nothing more than Composer libraries with some particular bootstrapping code.

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

This chapter follows on from a previous chapter which covered the basics of creating a deployment process for your applications. It goes into detail about creating a Laravel 4 installation; so you should be familiar with it before spending a great deal of time in this one. You'll also find the source-code we created there to be a good basis for understanding the source code of this chapter.

I learned the really technical bits of this chapter from reading Taylor's book. If you take away just one thing from this it should be to get and read it.

## Composer

While the previous chapter shows you how to create a Laravel 4 project; we need to do a bit more work in this area if we are to understand one of the fundamental ways in which packages differ from application code.

Laravel 3 had the concept of bundles; which were downloadable folders of source code. Laravel 4 extends this idea, but instead of rolling its own download process, it makes use of Composer. Composer should not only be used to create new Laravel 4 installations; but also as a means of adding packages to existing Laravel 4 applications.

Laravel 4 packages are essentially the same things as normal Composer libraries. They often utilise functionality built into the Laravel 4 framework, but this isn't a requirement of packages in general.

It follows that the end result of our work today is a stand-alone Composer library including the various deployment tools we created last time. For now, we will be placing the package code inside our application folder to speed up iteration time.

## Dependency Injection

One of the darlings of modern PHP development is Dependency Injection. PHP developers have recently grown fond of unit testing and class decoupling. I'll explain both of these things shortly;

but it's important to know that they are greatly improved by the use of dependency injection.

Let's look at some non-injected dependencies, and the troubles they create for us.

```php
<?php

class Archiver
{
    protected $database;

    public function __construct()
    {
        $this->database = Database::connect(
            "host",
            "username",
            "password",
            "schema"
        );
    }

    public function archive()
    {
        $entries = $this->database->getEntries();

        foreach ($entries as $entry)
        {
            if ($entry->aggregated)
            {
                $entry->archive();
            }
        }
    }
}

$archiver = new Archiver();
$archiver->archive();
```

Assuming, for a moment, that **Database** has been defined to pull all entries in such a way that they have an aggregated property and an **archive()** method; this code should be straightforward to follow.

The **Archiver** constructor initialises a database connection and stores a reference to it within a protected property. The **archive()** method iterates over the entries and archives them if they are already aggregated. Easy stuff.

Not so easy to test. The problem is that testing the business logic means testing the database connection and the entry instances as well. They are dependencies to any unit test for this class.

Furthermore; the Archiver class needs to know that these things exist and how they work. It's not enough just to know that the database instance is available or that the **getEntries()** method returns entries. If we have thirty classes all depending on the database; something small (like a change to the database password) becomes a nightmare.

Dependency injection takes two steps, in PHP. The first is to declare dependencies outside of classes and pass them in:

```php
<?php

class Archiver
{
    protected $database;

    public function __construct(Database $database)
    {
        $this->database = $database;
    }

    // ...then the archive() method
}

$database = Database::connect(
    "host",
    "username",
    "password",
    "schema"
);

$archiver = new Archiver($database);
$archiver->archive();
```

This may seem like a small, tedious change but it immediately enables independent unit testing of the database and the **Archiver** class.

The second step to proper dependency injection is to abstract the dependencies in such a way that they provide a minimum of functionality. Another way of looking at it is that we want to reduce the impact of changes or the leaking of details to classes with dependencies:

```php
<?php

interface EntryProviderInterface
{
    public function getEntries();
}

class EntryProvider implements EntryProviderInterface
{
    protected $database;

    public function __construct(Database $database)
    {
        $this->database = $database;
    }

    public function getEntries()
    {
        // ...get the entries from the database
    }
}

class Archiver
{
    protected $provider;

    public function __construct(
        EntryProviderInterface $provider
    )
    {
        $this->provider = $provider;
    }

    // ...then the archive() method
}

$database = Database::connect(
    "host",
    "username",
    "password",
    "schema"
);
```

```

$provider = new EntryProvider($database);
$archiver = new Archiver($provider);
$archiver->archive();
```

Woah Nelly! That's a lot of code when compared to the first snippet; but it's worth it. Firstly; we're exposing so few details to the **Archiver** class that the entire entry dependency could be replaced in a heartbeat. This is possible because we've moved the database dependency out of the **Archiver** and into the **EntryProvider**.

We're also type-hinting an interface instead of a concrete class; which lets us swap the concrete class out with anything else that implements EntryProviderInterface. The concrete class can fetch the entries from the database, or the filesystem or whatever.

We can test the **EntryProvider** class by swapping in a fake **Database** instance. We can test the **Archiver** class by swapping in a fake **EntryProvider** instance.

So, to recap the requirements for dependency injection:

1. Don't create class instances in other classes (if they are dependencies)—pass them into the class from outside.
2. Don't type-hint concrete classes—create interfaces.

> "When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.
>
> What you should be doing is stating a need, 'I need something to drink with lunch,' and then we will make sure you have something when you sit down to eat."
>
> —John Munsch

You can learn more about Dependency Injection, from Taylor's book, at: **https://leanpub.com/laravel**.

## Inversion Of Control

Inversion of Control (IoC) is the name given to the process that involves assembling class instances (and the resolution of class instances in general) within a container or registry. Where the registry pattern involves defining class instances and then storing them in some global container; IoC involves telling the container how and where to find the instances so that they can be resolved as required.

This naturally aids dependency injection as class instances adhering to abstracted requirements can easily be resolved without first creating. To put it another way; if our classes adhere to certain requirements (via interfaces) and the IoC container can resolve them to class instances, then we don't have to do it beforehand.

The easiest way to understand this is to look at some code:

```php
<?php

interface DatabaseInterface
{
    // ...
}

class Database implements DatabaseInterface
{
    // ...
}

interface EntryProviderInterface
{
    // ...
}

class EntryProvider implements EntryProviderInterface
{
    protected $database;

    public function __construct(DatabaseInterface $database)
    {
        $this->database = $database;
    }

    // ...
}

interface ArchiverInterface
{
    // ...
}
```

```php

class Archiver implements ArchiverInterface
{
    protected $provider;

    public function __construct(
        EntryProviderInterface $provider
    )
    {
        $this->provider = $provider;
    }

    // ...
}

$database = new Database();
$provider = new EntryProvider($database);
$archiver = new Archiver($provider);
```

This shallowly represents a dependency (injection) chain. The last three lines are where the problem starts to become clear; the more we abstract our dependencies, the more “bootstrapping” code needs to be done every time we need the **Archiver** class.

We could abstract this by using Laravel 4’s IoC container:

```php
<?php

// ...define classes

App::bind("DatabaseInterface", function() {
    return new Database();
});

App::bind("EntryProviderInterface", function() {
    return new EntryProvider(
        App::make("DatabaseInterface")
    );
});

App::bind("ArchiverInterface", function() {
    return new Archiver(
        App::make("EntryProviderInterface")
    );
});

$archiver = App::make("ArchiverInterface");
```

These extra nine lines (using the **App::bind()** and **App::make()** methods) tell Laravel 4 how to find/make new class instances, so we can get to the business of using them!

> You can learn more about IoC container at: **http://laravel.com/docs/ioc**.

## Service Providers

The main purpose of services providers is to collect and organise the bootstrapping requirements of your package. They're not strictly a requirement of package development; but rather a Really Good Idea™.

There are three big things to service providers. The first is that they are registered in a common configuration array (in **app/config/app.php**):

```
'providers' => array(
    'Illuminate\Foundation\Providers\ArtisanServiceProvider',
    'Illuminate\Auth\AuthServiceProvider',
    'Illuminate\Cache\CacheServiceProvider',
    // ...
    'Illuminate\Validation\ValidationServiceProvider',
    'Illuminate\View\ViewServiceProvider',
    'Illuminate\Workbench\WorkbenchServiceProvider',
),
```

> This was extracted from **app/config/app.php**.

You can also add your own service providers to this list; as many packages will recommend you do. There there's the **register()** method:

```php
<?php namespace Illuminate\Cookie;

use Illuminate\Support\ServiceProvider;

class CookieServiceProvider extends ServiceProvider {

    /**
    * Register the service provider.
    *
    * @return void
    */
    public function register()
    {
        $this->app['cookie'] = $this->app->share(function($app)
        {
            $cookies = new CookieJar(
                $app['request'],
                $app['encrypter']
            );

            $config = $app['config']['session'];

            return $cookies->setDefaultPathAndDomain(
                $config['path'],
                $config['domain']
            );
        });
    }

}
```

This file should be saved as **vendor/laravel/framework/src/Illuminate/Cookie/CookieServiceProvider.php**.

When we take a look at the **register()** method of the **CookieServiceProvider** class; we can see a call to **$this->app->share()** which is similar to the **App::bind()** method but instead of creating a new class instance every time it's resolved in the IoC container; **share()** wraps a callback so that it's shared with every resolution.

The name of the **register()** method explains exactly what it should be used for; registering things with the IoC container (which **App** extends). If you need to do other bootstrapping stuff then the method you need to use is **boot()**:

```
public function boot()
{
    Model::setConnectionResolver($this->app['db']);
    Model::setEventDispatcher($this->app['events']);
}
```

This was extracted from **vendor/laravel/framework/src/Illuminate/Database/DatabaseServiceProvider.php**.

This **boot()** method sets two properties on the **Model** class. The same class also has a register method but these two settings are pet for the boot method.

You can learn more about service providers at: **http://laravel.com/docs/packages#service-providers**.

## Organising Code

Now that we have some tools to help us create packages; we need to port our code over from being application code to being a package. Laravel 4 provides a useful method for creating some structure to our package:

```
php artisan workbench Formativ/Deployment
```

You'll notice a new folder has been created, called workbench. Within this folder you'll find an assortment of files arranged in a similar way to those in the **vendor/laravel/framework/src/Illuminate/*** folders.

We need to break all of the business logic (regarding deployment) into individual classes, making use of the IoC container and dependency injection, and make a public API.

We're not going to spend a lot of time discussing the intricacies of the deployment classes/scripts we made in the last tutorial. Refer to it if you want more of those details.

To recap; the commands we need to abstract are:

- asset:combine
- asset:minify
- clean
- copy
- distribute
- environment:get
- environment:remove
- environment:set
- watch

Many of these were cluttering up the list of commands which ship with Laravel. Our organisation should collect all of these in a common "namespace".

The first step is to create a few contracts (interfaces) for the API we want to expose through our package:

```php
<?php

namespace Formativ\Deployment\Asset;

interface ManagerInterface
{
    public function add($source, $target);
    public function remove($target);
    public function combine();
    public function minify();
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Asset/ManagerInterface.php**.

```php
<?php

namespace Formativ\Deployment\Asset;

interface WatcherInterface
{
    public function watch();
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Asset/WatcherInterface.php**.

```php
<?php

namespace Formativ\Deployment;

interface DistributionInterface
{
    public function prepare($source, $target);
    public function sync();
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/DistributionInterface.php**.

```php
<?php

namespace Formativ\Deployment;

interface MachineInterface
{
    public function getEnvironment();
    public function setEnvironment($environment);
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/MachineInterface.php**.

The methods required by these interfaces encompass the functionality our previous commands provided for us. The next step is to fulfil these contracts with concrete class implementations:

```php
<?php

namespace Formativ\Deployment\Asset;

use Formativ\Deployment\MachineInterface;
use Illuminate\Filesystem\Filesystem;

class Manager
implements ManagerInterface
{
    protected $files;

    protected $machine;

    protected $assets = [];

    public function __construct(
        Filesystem $files,
        MachineInterface $machine
    )
    {
        $this->files = $files;
        $this->machine = $machine;
    }

    public function add($source, $target)
    {
        // ...add files to $assets
    }

    public function remove($target)
    {
        // ...remove files from $assets
    }

    public function combine()
    {
        // ...combine assets
    }

    public function minify()
    {
        // ...minify assets
    }
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Asset/Manager.php**.

```
<?php

namespace Formativ\Deployment\Asset;

use Formativ\Deployment\MachineInterface;
use Illuminate\Filesystem\Filesystem;

class Watcher
implements WatcherInterface
{
    protected $files;

    protected $machine;

    public function __construct(
        Filesystem $files,
        MachineInterface  $machine
    )
    {
        $this->files = $files;
        $this->machine = $machine;
    }

    public function watch()
    {
        // ...watch assets for changes
    }
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Asset/Watcher.php**.

```
<?php

namespace Formativ\Deployment;

use Formativ\Deployment\MachineInterface;
use Illuminate\Filesystem\Filesystem;

class Distribution
implements DistributionInterface
{
    protected $files;

    protected $machine;

    public function __construct(
        Filesystem $files,
        MachineInterface $machine
    )
```

```php
    {
        $this->files = $files;
        $this->machine = $machine;
    }

    public function prepare($source, $target)
    {
        // ...copy + clean files for distribution
    }

    public function sync()
    {
        // ...sync distribution files to remote server
    }
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Distribution.php**.

```php
<?php

namespace Formativ\Deployment;

use Illuminate\Filesystem\Filesystem;

class Machine
implements MachineInterface
{
    protected $environment;

    protected $files;

    public function __construct(Filesystem $files)
    {
        $this->files = $files;
    }

    public function getEnvironment()
    {
        // ...get the current environment
    }

    public function setEnvironment($environment)
    {
        // ...set the current environment
    }
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Machine.php**.

The main to note about these implementations is that they use dependency injection instead of creating class instances in their constructors. As I pointed out before; we're not going to go over the actually implementation code, but feel free to port it over from the application-based commands.

As you can probably tell; I've combined related functionality into four main classes. This becomes even more apparent when you consider what the service provider has become:

```php
<?php

namespace Formativ\Deployment;

use App;
use Illuminate\Support\ServiceProvider;

class DeploymentServiceProvider
extends ServiceProvider
{
    protected $defer = true;

    public function register()
    {
        App::bind("deployment.asset.manager", function()
        {
            return new Asset\Manager(
                App::make("files"),
                App::make("deployment.machine")
            );
        });

        App::bind("deployment.asset.watcher", function()
        {
            return new Asset\Watcher(
                App::make("files"),
                App::make("deployment.machine")
            );
        });

        App::bind("deployment.distribution", function()
        {
            return new Distribution(
                App::make("files"),
                App::make("deployment.machine")
            );
        });

        App::bind("deployment.machine", function()
        {
            return new Machine(
                App::make("files")
            );
        });

        App::bind("deployment.command.asset.combine", function()
        {
            return new Command\Asset\Combine(
                App::make("deployment.asset.manager")
            );
        });

        App::bind("deployment.command.asset.minify", function()
        {
            return new Command\Asset\Minify(
                App::make("deployment.asset.manager")
            );
        });

        App::bind("deployment.command.asset.watch", function()
        {
            return new Command\Asset\Watch(
                App::make("deployment.asset.manager")
            );
        });

        App::bind("deployment.command.distribute.prepare", function()
        {
            return new Command\Distribute\Prepare(
                App::make("deployment.distribution")
            );
        });

        App::bind("deployment.command.distribute.sync", function()
        {
            return new Command\Distribute\Sync(
                App::make("deployment.distribution")
            );
        });

        App::bind("deployment.command.machine.add", function()
        {
            return new Command\Machine\Add(
                App::make("deployment.machine")
            );
        });

        App::bind("deployment.command.machine.remove", function()
        {
            return new Command\Machine\Remove(
                App::make("deployment.machine")
            );
        });

        $this->commands(
            "deployment.command.asset.combine",
            "deployment.command.asset.minify",
            "deployment.command.asset.watch",
            "deployment.command.distribute.prepare",
            "deployment.command.distribute.sync",
            "deployment.command.machine.add",
            "deployment.command.machine.remove"
        );
```

```
    }

    public function boot()
    {
        $this->package("formativ/deployment");
        include __DIR__ . "/../../helpers.php";
        include __DIR__ . "/../../macros.php";
    }

    public function provides()
    {
        return [
            "deployment.asset.manager",
            "deployment.asset.watcher",
            "deployment.distribution",
            "deployment.machine",
            "deployment.command.asset.combine",
            "deployment.command.asset.minify",
            "deployment.command.asset.watch",
            "deployment.command.distribute.prepare",
            "deployment.command.distribute.sync",
            "deployment.command.machine.add",
            "deployment.command.machine.remove"
        ];
    }
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/DeploymentServiceProvider.php**.

I’ve folded the **copy()** and **clean()** methods into a single **prepare()** method.

The first four **bind()** calls bind our four main classes to the IoC container. The remaining **bind()** methods bind the artisan commands we still have to create (to replace those we make last time).

There's also a call to **$this->commands()**; which registers commands (bound to the IoC container) with artisan. Finally; we define the **provides()** method, which coincides with the **$defer = true** property, to inform Laravel which IoC container bindings are returned by this service provider. By setting **$defer = true**; we're instructing Laravel to not immediately load the provided classes and commands, but rather wait until they are required.

This is the first time we're using the **boot()** method. The call to **$this->package()** is so that we can have views, language files and other resources mapped to our package. We would be able to call them by prefixing the view names, language keys etc. with the name of the package. We're not going to be using that this time round; but it's important to know how.

We also include **helpers.php** and **macros.php** in the boot method.

```php
<?php

namespace Formativ\Deployment\Command\Asset;

use Formativ\Deployment\Asset\ManagerInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Combine
extends Command
{
    protected $name = "deployment:asset:combine";

    protected $description = "Combines multiple resource files.";

    protected $manager;

    public function __construct(ManagerInterface $manager)
    {
        parent::__construct();
        $this->manager = $manager;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Asset/Combine.php**.

```php
<?php

namespace Formativ\Deployment\Command\Asset;

use Formativ\Deployment\Asset\ManagerInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Minify
extends Command
{
    protected $name = "deployment:asset:minify";

    protected $description = "Minifies multiple resource files.";

    protected $manager;

    public function __construct(ManagerInterface $manager)
    {
        parent::__construct();
        $this->manager = $manager;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Asset/Minify.php**.

```php
<?php

namespace Formativ\Deployment\Command\Asset;

use Formativ\Deployment\Asset\ManagerInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Watch
extends Command
{
    protected $name = "deployment:asset:watch";

    protected $description = "Watches files for changes.";

    protected $manager;

    public function __construct(ManagerInterface $manager)
    {
        parent::__construct();
        $this->manager = $manager;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Asset/Watch.php**.

```php
<?php

namespace Formativ\Deployment\Command\Distribute;

use Formativ\Deployment\DistributionInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Prepare
extends Command
{
    protected $name = "deployment:distribute:prepare";

    protected $description = "Prepares the distribution folder.";

    protected $distribution;

    public function __construct(
        DistributionInterface $distribution
    )
    {
        parent::__construct();
        $this->distribution = $distribution;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Distribute/Prepare.php**.

```php
<?php

namespace Formativ\Deployment\Command\Distribute;

use Formativ\Deployment\DistributionInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Sync
extends Command
{
    protected $name = "deployment:distribute:sync";

    protected $description = "Syncs changes to a target.";

    protected $distribution;

    public function __construct(
        DistributionInterface $distribution
    )
    {
        parent::__construct();
        $this->distribution = $distribution;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Distribute/Sync.php**.

```php
<?php

namespace Formativ\Deployment\Command\Machine;

use Formativ\Deployment\MachineInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Add
extends Command
{
    protected $name = "deployment:machine:add";

    protected $description = "Adds the current machine to an environment.";

    protected $machine;

    public function __construct(MachineInterface $machine)
    {
        parent::__construct();
        $this->machine = $machine;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Machine/Add.php**.

```php
<?php

namespace Formativ\Deployment\Command\Machine;

use Formativ\Deployment\MachineInterface;
use Illuminate\Console\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputOption;

class Remove
extends Command
{
    protected $name = "deployment:machine:remove";

    protected $description = "Removes the current machine from an environment.";

    protected $machine;

    public function __construct(MachineInterface $machine)
    {
        parent::__construct();
        $this->machine = $machine;
    }

    // ...
}
```

This file should be saved as **workbench/formativ/deployment/src/Formativ/Deployment/Command/Machine/Remove.php**.

I've also omitted the **fire()** method from the new commands; consider it an exercise to add these in yourself.

Now, when we run the artisan list command; we should see all of our new package commands neatly grouped. Feel free to remove the old commands, after you've ported their functionality over to the new ones.

## Publishing Configuration Files

Often you'll want to add new configuration files to the collection which ship with new Laravel applications. There's an artisan command to help with this:

```
php artisan config:publish formativ/deployment --path=workbench/Formativ/Deployme\
nt/src/config
```

This should copy the package configuration files into the **app/config/packages/formativ/deployment** directory.

> Since we have access to a whole new kind of configuration; we no longer need to override the configuration files for things like environment. As long as our helpers/macros use the package configuration files instead of the the default ones; we can leave the underlying Laravel 4 application structure (and bootstrapping code) untouched.

## Creating Composer.json

Before we can publish our package, so that others can use it in their applications; we need to add a few things to the **composer.json** file that the workbench command created for us:

```
{
    "name"        : "formativ/deployment",
    "description" : "All sorts of cool things with deployment.",
    "authors"     : [
        {
            "name"  : "Christopher Pitt",
            "email" : "cgpitt@gmail.com"
        }
    ],
    "require" : {
        "php"                 : ">=5.4.0",
        "illuminate/support"  : "4.0.x"
    },
    "autoload" : {
        "psr-0" : {
            "Formativ\\Deployment" : "src/"
        }
    },
    "minimum-stability" : "dev"
}
```

This file should be saved as **workbench/formativ/deployment/composer.json**.

I've added a description, my name and email address. I also cleaned up the formatting a bit; but that's not really a requirement. I've also set the minimum PHP version to 5.4.0 as we're using things like short array syntax in our package.

You should also create a **readme.md** file, which contains installation instructions, usage instructions, license and contribution information. I can't understate the import ants of these things—they are your only hope to attract contributors.

You can learn more about the composer.json file at: **http://getcomposer.org/doc/01-basic-usage.md#composer-json-project-setup**.

## Submitting A Package To Packagist

To get others using your packages; you need to submit them to packagist.org. Go to **http://packagist.org** and sign in. If you haven't already got an account, you should create it and link to your GitHub account.

You'll find a big "Submit Package" button on the home page. Click it and paste the URL to your GitHub repository in the text field. Submit the form and you should be good to go.

You can learn more about package development at: **http://laravel.com/docs/packages**.

## Note On Testing

Those of you who already know about unit testing will doubtlessly wonder where the section on testing is. I ran short on time for that in this chapter, but it will be the subject of a future chapter; which will demonstrate the benefits of dependency injection as it relates to unit testing.

# Real Time Chat

One of the most pervasive and least understood technologies that underpin the internet is socket programming. It's been a dark art for decades, but the recent standardisation of web sockets (in relation to HTML 5) has made this type of programming a little easier to get into.

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

## Dependencies

This tutorial depends heavily on client-side resources. We're developing a Laravel 4 application which has lots of server-side aspects; but it's also a chat app. There be scripts!

### Bootstrap

For this; we're using Bootstrap and EmberJS. Download Bootstrap at: **http://getbootstrap.com/** and unpack it into your public folder. Where you put the individual files makes little difference, but I have put the scripts in **public/js**, the stylesheets in **public/css** and the fonts in **public/fonts**. Where you see those paths in my source-code; you should substitute them with your own.

### EmberJS

Next up, download EmberJS at: **http://emberjs.com/** and unpack it into your public folder. You'll also need the Ember.Data script at: **http://emberjs.com/guides/getting-started/obtaining-emberjs-and-dependencies/**.

### Ratchet

For the server-side portion of dependencies, we need to download a library called Ratchet. I'll explain it shortly, but in the meantime we need to add it to our **composer.json** file:

```
"require" : {
    "laravel/framework" : "4.0.*",
    "cboden/Ratchet"    : "0.3.*"
},
```

This was extracted from **composer.json**.

Follow that up with:

```
composer update
```

Ratchet isn't built specifically for Laravel 4, so there are no service providers for us to add.

We'll now have access to the Ratchet library for client-server communication, Bootstrap for styling the interface and EmberJS for connecting these two things together.

## ReactPHP

Before we can understand Ratchet, we need to understand ReactPHP. ReactPHP was born out of the need to develop event-based, asynchronous PHP applications. If you've worked with Node.JS you'll feel right at home developing applications with ReactPHP; as they share a similar approaches to code. We're not going to develop our chat application in ReactPHP, but it's a dependency for Ratchet…

You can learn more about ReactPHP at: **http://reactphp.org/**.

## Ratchet

One of the many ways in which real-time client-server applications are made possible is by what's called socket programming. Believe it or not; most of what you do on the internet depends on socket programming. From simple browsing to streaming—your computer opens a socket connection to a server and the server sends data back through it.

PHP supports this type of programming but PHP websites have not typically been developed with this kind of model in mind. PHP developers have preferred the typical request/response model, and it's comparatively easier than low-level socket programming.

Enter ReactPHP. One of the requirements for building a fully-capable socket programming framework is creating what's called an Event Loop. ReactPHP has this and Ratchet uses it, along with the Publish/Subscribe model to accept and maintain open socket connections.

ReactPHP wraps the low-level PHP functions into a nice socket programming API and Ratchet wraps that API into another API that's even easier to use.

You can learn more about Ratchet at: **http://socketo.me/**.

## Creating An Interface

Let's get to the code! We're going to need an interface (kind of like a wireframe) so we know what to build with our application. Let's set up a simple view and plug it into EmberJS.

I should mention that I am by no means an EmberJS expert. I learned all I know of it, while writing this tutorial, by following various guides. The point of this is not to teach EmberJS so much as it is to show EmberJS integration with Laravel 4.

### Creating A View

Let's change the default routes.php file to load a custom view:

```php
<?php

Route::get("/", function()
{
    return View::make("index/index");
});
```

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

Then we need to add this view:

```
<!DOCTYPE html>
<html lang="en">
    <head>
    <meta charset="UTF-8" />
        <link
            rel="stylesheet"
            type="text/css"
            href="{{ asset("css/bootstrap.3.0.0.css") }}"
        />
        <link
            rel="stylesheet"
            type="text/css"
            href="{{ asset("css/bootstrap.theme.3.0.0.css") }}"
        />
        <title>Laravel 4 Chat</title>
    </head>
    <body>
        <script type="text/x-handlebars">
            @{{outlet}}
        </script>
        <script
            type="text/x-handlebars"
            data-template-name="index"
        >
            <div class="container">
                <div class="row">
                    <div class="col-md-12">
                        <h1>Laravel 4 Chat</h1>
                        <table class="table table-striped">
                            @{{#each}}
                                <tr>
                                    <td>
                                        @{{user}}
                                    </td>
                                    <td>
```

```
                                        @{{text}}
                                    </td>
                                </tr>
                            @{{/each}}
                        </table>
                    </div>
                </div>
                <div class="row">
                    <div class="col-md-12">
                        <div class="input-group">
                            <input
                                type="text"
                                class="form-control"
                            />
                            <span class="input-group-btn">
                                <button class="btn btn-default">
                                    Send
                                </button>
                            </span>
                        </div>
                    </div>
                </div>
            </div>
        </script>
        <script
            type="text/javascript"
            src="{{ asset("js/jquery.1.9.1.js") }}"
        ></script>
        <script
            type="text/javascript"
            src="{{ asset("js/handlebars.1.0.0.js") }}"
        ></script>
        <script
            type="text/javascript"
            src="{{ asset("js/ember.1.1.1.js") }}"
        ></script>
        <script
            type="text/javascript"
            src="{{ asset("js/ember.data.1.0.0.js") }}"
        ></script>
        <script
            type="text/javascript"
            src="{{ asset("js/bootstrap.3.0.0.js") }}"
        ></script>
        <script
            type="text/javascript"
            src="{{ asset("js/shared.js") }}"
        ></script>
    </body>
</html>
```

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

Both Blade and EmberJS use double-curly-brace syntax for variable and logic substitution. Luckily Blade includes a mechanism to ignore curly brace blocks, by prepending them with **@** symbols. Thus our template includes **@** symbols before all EmberJS blocks.

The scripts and stylesheets need to be relative to where you saved them or you're going to see errors.

## Creating An EmberJS App

You'll notice I've specified **shared.css** and **shared.js** files—the CSS file is blank, but the JavaScript file contains:

```
// 1
var App = Ember.Application.create();

// 2
App.Router.map(function() {
    this.resource("index", {
        "path" : "/"
    });
});

// 3
App.Message = DS.Model.extend({
    "user" : DS.attr("string"),
    "text" : DS.attr("string")
});

// 4
App.ApplicationAdapter = DS.FixtureAdapter.extend();

// 5
App.Message.FIXTURES = [
    {
        "id"   : 1,
        "user" : "Chris",
        "text" : "Hello World."
    },
    {
        "id"   : 2,
        "user" : "Wayne",
        "text" : "Don't dig it, man."
    },
    {
        "id"   : 3,
        "user" : "Chris",
        "text" : "Meh."
    }
];
```

This file should be saved as **public/js/shared.js**.

If you're an EmberJS noob, like me, then it will help to understand what each piece of this script is doing.

1. We create a new Ember application with **Ember.Application.create()**.
2. Routes are defined in the **App.Route.map()** method, and we tell the application to equate the path / to the index resource.
3. We define a **Message** model. These are similar to the Eloquent models we have in Laravel 4, but they're built to work with EmberJS (and are obviously on the client-side of the application).
4. We specify a fixture-based data store for our application. We're using this, temporarily, to fill our interface with some dummy data, but we'll add a dynamic data store before too long…
5. Here we add the fixture data. Notice that, in addition to the two model fields we defined, we also specify ID values for the fixture rows. This data is used to single out individual Message objects.

When you browse to the base URL of the application; you should now see an acceptably styled list of message objects, along with a heading and input form. Let's make it dynamic!

You should also see some console log messages (depending on your browser) which show that EmberJS is running.

## Creating A Service Provider

Following on from a previous tutorial; we're going to be creating a service provider which will provide the various classes for our application. Create a new workbench:

```
php artisan workbench formativ/chat
```

This will produce (amongst other things) a service provider template. I've added a few IoC bindings to it:

```
<?php

namespace Formativ\Chat;

use Evenement\EventEmitter;
use Illuminate\Support\ServiceProvider;
use Ratchet\Server\IoServer;

class ChatServiceProvider
extends ServiceProvider
{
    protected $defer = true;

    public function register()
    {
        $this->app->bind("chat.emitter", function()
        {
            return new EventEmitter();
        });

        $this->app->bind("chat.chat", function()
        {
            return new Chat(
                $this->app->make("chat.emitter")
            );
        });

        $this->app->bind("chat.user", function()
        {
            return new User();
        });

        $this->app->bind("chat.command.serve", function()
        {
            return new Command\Serve(
                $this->app->make("chat.chat")
            );
        });

        $this->commands("chat.command.serve");
    }

    public function provides()
    {
        return [
            "chat.chat",
            "chat.command.serve",
            "chat.emitter",
            "chat.server"
        ];
    }
}
```

This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/ChatService-Provider.php**.

The first binding is a simple alias to the **EvenementEventEmitter** class (which Ratchet requires). We bind it here as we cannot guarantee that Ratchet will continue to use **EvenementEventEmitter** and we'll need a reliable way to unit test possible alternatives in the future.

## Creating A Chat Handler

Let's look closer at the second and third bindings. The first is to the **Chat** class. It implements the **ChatInterface** interface:

```php
<?php

namespace Formativ\Chat;

use Evenement\EventEmitterInterface;
use Ratchet\ConnectionInterface;
use Ratchet\MessageComponentInterface;

interface ChatInterface
extends MessageComponentInterface
{
    public function getUserBySocket(ConnectionInterface $socket);
    public function getEmitter();
    public function setEmitter(EventEmitterInterface $emitter);
    public function getUsers();
}
```

This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/ChatInterface.php**.

It's interesting to note that PHP actually supports interfaces which extend other interfaces. This is useful if you want to expect a certain level of functionality, provided by a third-party library (such as SPL), but want to add your own requirements on top.

The concrete implementation looks like this:

```php
<?php

namespace Formativ\Chat;

use Evenement\EventEmitterInterface;
use Exception;
use Ratchet\ConnectionInterface;
use SplObjectStorage;

class Chat
implements ChatInterface
{
    protected $users;
    protected $emitter;
    protected $id = 1;

    public function getUserBySocket(ConnectionInterface $socket)
    {
        foreach ($this->users as $next)
        {
            if ($next->getSocket() === $socket)
            {
                return $next;
            }
        }

        return null;
    }

    public function getEmitter()
    {
        return $this->emitter;
    }

    public function setEmitter(EventEmitterInterface $emitter)
    {
        $this->emitter = $emitter;
    }

    public function getUsers()
    {
        return $this->users;
    }

    public function __construct(EventEmitterInterface $emitter)
    {
        $this->emitter = $emitter;
        $this->users   = new SplObjectStorage();
    }

    public function onOpen(ConnectionInterface $socket)
    {
        $user = new User();
        $user->setId($this->id++);
        $user->setSocket($socket);

        $this->users->attach($user);
        $this->emitter->emit("open", [$user]);
    }

    public function onMessage(
        ConnectionInterface $socket,
        $message
    )
    {
        $user = $this->getUserBySocket($socket);
        $message = json_decode($message);

        switch ($message->type)
        {
            case "name":
            {
                $user->setName($message->data);
                $this->emitter->emit("name", [
                    $user,
                    $message->data
                ]);
                break;
            }

            case "message":
            {
                $this->emitter->emit("message", [
                    $user,
                    $message->data
                ]);
                break;
            }
        }

        foreach ($this->users as $next)
        {
            if ($next !== $user)
            {
                $next->getSocket()->send(json_encode([
                    "user" => [
                        "id"   => $user->getId(),
                        "name" => $user->getName()
                    ],
                    "message" => $message
                ]));
            }
        }
    }

    public function onClose(ConnectionInterface $socket)
    {
        $user = $this->getUserBySocket($socket);

        if ($user)
        {
            $this->users->detach($user);
            $this->emitter->emit("close", [$user]);
        }
    }

    public function onError(
        ConnectionInterface $socket,
        Exception $exception
    )
    {
        $user = $this->getUserBySocket($socket);

        if ($user)
        {
            $user->getSocket()->close();
```

```
                $this->emitter->emit("error", [$user, $exception]);
            }
        }
}
```

This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/Chat.php**.

It's fairly simple: the (delegate) **onOpen** and **onClose** methods handle creating new User objects and disposing of them. The **onMessage** method translates JSON-encoded message objects into required actions and responds back to the other socket connections with further details.

## Creating A Socket Wrapper

Additionally; the **UserInterface** interface and **User** class look like this:

```php
<?php

namespace Formativ\Chat;

use Evenement\EventEmitterInterface;
use Ratchet\ConnectionInterface;
use Ratchet\MessageComponentInterface;

interface UserInterface
{
    public function getSocket();
    public function setSocket(ConnectionInterface $socket);
    public function getId();
    public function setId($id);
    public function getName();
    public function setName($name);
}
```

This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/UserInterface.php**.

```php
<?php

namespace Formativ\Chat;

use Ratchet\ConnectionInterface;

class User
implements UserInterface
{
    protected $socket;
    protected $id;
    protected $name;

    public function getSocket()
    {
        return $this->socket;
    }

    public function setSocket(ConnectionInterface $socket)
    {
        $this->socket = $socket;
        return $this;
    }

    public function getId()
    {
        return $this->id;
    }

    public function setId($id)
    {
        $this->id = $id;
        return $this;
    }

    public function getName()
    {
        return $this->name;
    }

    public function setName($name)
    {
        $this->name = $name;
        return $this;
    }
}
```

This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/User.php**.

The **User** class is a simple wrapper for a socket resource and name string. The way we've chosen to implement the Ratchet server requires that we have a class which implements the **MessageComponentInterface** interface; and this interface specifies that **ConnectionInterface** objects are passed back and forth. There's no way to identify these, by name (and id), so we're adding that functionality with the extra layer.

## Creating A Serve Command

All these classes lead us to the artisan command which will kick things off:

```
<?php

namespace Formativ\Chat\Command;

use Illuminate\Console\Command;
use Formativ\Chat\ChatInterface;
use Formativ\Chat\UserInterface;
use Ratchet\ConnectionInterface;
use Ratchet\Http\HttpServer;
use Ratchet\Server\IoServer;
use Ratchet\WebSocket\WsServer;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;

class Serve
extends Command
{
    protected $name        = "chat:serve";
    protected $description = "Command description.";
    protected $chat;

```

```php
    protected function getUserName($user)
    {
        $suffix = " (" . $user->getId() . ")";

        if ($name = $user->getName())
        {
            return $name . $suffix;
        }

        return "User" . $suffix;
    }

    public function __construct(ChatInterface $chat)
    {
        parent::__construct();

        $this->chat = $chat;

        $open = function(UserInterface $user)
        {
            $name = $this->getUserName($user);
            $this->line("
                <info>" . $name . " connected.</info>
            ");
        };

        $this->chat->getEmitter()->on("open", $open);

        $close = function(UserInterface $user)
        {
            $name = $this->getUserName($user);
            $this->line("
                <info>" . $name . " disconnected.</info>
            ");
        };

        $this->chat->getEmitter()->on("close", $close);

        $message = function(UserInterface $user, $message)
        {
            $name = $this->getUserName($user);
            $this->line("
                <info>New message from " . $name . ":</info>
                <comment>" . $message . "</comment>
                <info>.</info>
            ");
        };

        $this->chat->getEmitter()->on("message", $message);

        $name = function(UserInterface $user, $message)
        {
            $this->line("
                <info>User changed their name to:</info>
                <comment>" . $message . "</comment>
                <info>.</info>
            ");
        };

        $this->chat->getEmitter()->on("name", $name);

        $error = function(UserInterface $user, $exception)
        {
            $message = $exception->getMessage();

            $this->line("
                <info>User encountered an exception:</info>
                <comment>" . $message . "</comment>
                <info>.</info>
            ");
        };

        $this->chat->getEmitter()->on("error", $error);
    }

    public function fire()
    {
        $port = (integer) $this->option("port");

        if (!$port)
        {
            $port = 7778;
        }

        $server = IoServer::factory(
            new HttpServer(
                new WsServer(
                    $this->chat
                )
            ),
            $port
        );

        $this->line("
            <info>Listening on port</info>
            <comment>" . $port . "</comment>
            <info>.</info>
        ");

        $server->run();
    }

    protected function getOptions()
    {
        return [
            [
                "port",
                null,
                InputOption::VALUE_REQUIRED,
                "Port to listen on.",
                null
            ]
        ];
    }
}
```

This file should be saved as **workbench/formativ/chat/src/Formativ/Chat/Command/Serve.php**.

The reason for us adding the event emitter to the equation should now be obvious—we need a way to tie into the delegated events, of the **Chat** class, without leaking the abstraction we gain from it. In other words; we don't want the **Chat** class to know of the existence of the artisan command. Similarly; we don't want the artisan command to know of the **onOpen**, **onMessage**, **onError** and

**onMessage** methods so instead we use a publish/subscribe model for notifying the command of changes. The result is a clean abstraction.

The **fire()** method gets (or defaults) the port and starts the Ratchet web socket server.

> The method we're using, to start the server, is not the only way it can be started. You can learn more about the web socket server at: **http://socketo.me/docs/websocket**.

## Connecting To The Socket Server

To connect to the web socket server; we need to add a bit of vanilla JavaScript:

```
try {
    if (!WebSocket) {
        console.log("no websocket support");
    } else {
        var socket = new WebSocket("ws://127.0.0.1:7778/");

        socket.addEventListener("open", function (e) {
            console.log("open: ", e);
        });

        socket.addEventListener("error", function (e) {
            console.log("error: ", e);
        });

        socket.addEventListener("message", function (e) {
            console.log("message: ", JSON.parse(e.data));
        });

        console.log("socket:", socket);

        window.socket = socket;
    }
} catch (e) {
    console.log("exception: " + e);
}
```

This was extracted from **public/js/shared.js**.

We've wrapped this code in a try-catch block as not all browsers support Web Sockets yet. There are a number of libraries which will shim this functionality, but their use is outside the scope of this tutorial.

You can find a couple of these libraries at: **https://github.com/gimite/web-socket-js** and **https://github.com/sockjs**.

This code will attempt to open a socket connection to **127.0.0.1:7778** (the address and port used in the serve command) and write some console messages depending on the events that are emitted. You'll notice we're also assigning the socket instance to the window object; so we can send some debugging commands through it.

This allows us to see both the server-side of things, as well as the client-side…

## Wiring Up The Interface

Getting our interface talking to our socket server is relatively straightforward. We begin by disabling our fixture data and modifying our model slightly:

```
App.Message = DS.Model.extend({
    "user_id"       : DS.attr("integer"),
    "user_name"     : DS.attr("string"),
    "user_id_class" : DS.attr("string"),
    "message"       : DS.attr("string")
});

App.ApplicationAdapter = DS.FixtureAdapter.extend();

App.Message.FIXTURES = [
    // {
    //     "id"   : 1,
    //     "user" : "Chris",
    //     "text" : "Hello World."
    // },
    // {
    //     "id"   : 2,
    //     "user" : "Wayne",
    //     "text" : "Don't dig it, man."
    // },
    // {
    //     "id"   : 3,
    //     "user" : "Chris",
    //     "text" : "Meh."
    // }
];
```

This was extracted from **public/js/shared.js**.

If you want to pre-populate your chat application with a history; you could feed this fixture configuration with data from your server.

## Showing Chat Messages

Now the index template will show only the heading and form elements, but no chat messages. In order to populate these; we need to store a reference to the application data store:

```
var store;

App.IndexRoute = Ember.Route.extend({
    "init" : function() {
        store = this.store;
    },
    "model" : function () {
        return store.find("message");
    }
});
```

This was extracted from **public/js/shared.js**.

We store this reference because we will need to push rows into the store once we receive them from the open socket. This leads us to the changes to web sockets:

```
try {
    var id = 1;

    if (!WebSocket) {
        console.log("no websocket support");
    } else {

        var socket = new WebSocket("ws://127.0.0.1:7778/");
        var id     = 1;

        socket.addEventListener("open", function (e) {
            // console.log("open: ", e);
        });

        socket.addEventListener("error", function (e) {
            console.log("error: ", e);
        });

        socket.addEventListener("message", function (e) {

            var data      = JSON.parse(e.data);
            var user_id   = data.user.id;
            var user_name = data.user.name;
            var message   = data.message.data;

            switch (data.message.type) {

                case "name":
                    $(".name-" + user_id).html(user_name);
                    break;

                case "message":
                    store.push("message", {
                        "id"            : id++,
                        "user_id"       : user_id,
                        "user_name"     : user_name || "User",
                        "user_id_class" : "name-" + user_id,
                        "message"       : message
                    });
                    break;

            }

        });

        // console.log("socket:", socket);

        window.socket = socket; // debug
    }
} catch (e) {
    console.log("exception: " + e);
}
```

This was extracted from **public/js/shared.js**.

We start by defining an id variable, to store the id's of message objects as they are passed through the socket. Inside the **onMessage** event handler; we parse the JSON data string and determine the type of message being received. If it's a name message (a user changing their name) then we update all the table cells matching the user's server id. If it's a normal message object; we push it into the data store.

## Sending Chat Messages

This gets us part of the way, since console message commands will visually affect the UI. We still need to wire up the input form…

```
App.IndexController = Ember.ArrayController.extend({
    "command" : null,
    "actions" : {
        "send" : function(key) {

            if (key && key != 13) {
                return;
            }

            var command = this.get("command") || "";

            if (command.indexOf("/name") === 0) {
                socket.send(JSON.stringify({
                    "type" : "name",
                    "data" : command.split("/name")[1]
                }));
            } else {
                socket.send(JSON.stringify({
                    "type" : "message",
                    "data" : command
                }));
            }

            this.set("command", null);
        }
    }
});

App.IndexView = Ember.View.extend({
    "keyDown" : function(e) {
        this.get("controller").send("send", e.keyCode);
    }
});
```

This was extracted from **public/js/shared.js**.

We create **IndexController** and **IndexView** objects. The **IndexView** object intercepts the **keyDown** event and passes it to the **IndexController** object. The first bit of logic tells the **send()** method to ignore all keystrokes that aren't the enter key. This means enter will trigger the **send()** method.

It continues by checking for the presence of a **/name** command switch. If that's present in the input value (via **this.get("command")**) then it sends a message to the server to change the user's name. Otherwise it sends a normal message to the server. In order for the UI to update for the person sending the message; we need to also slightly modify the **Chat** class:

```
public function onMessage(ConnectionInterface $socket, $message)
{
    $user = $this->getUserBySocket($socket);
    $message = json_decode($message);

    switch ($message->type)
    {
        case "name":
        {
            $user->setName($message->data);
            $this->emitter->emit("name", [
                $user,
                $message->data
            ]);
            break;
        }

        case "message":
        {
            $this->emitter->emit("message", [
                $user,
                $message->data
            ]);
            break;
        }
    }

    foreach ($this->users as $next)
    {
        // if ($next !== $user)
        // {
            $next->getSocket()->send(json_encode([
                "user" => [
                    "id"   => $user->getId(),
                    "name" => $user->getName()
                ],
                "message" => $message
            ]));
        // }
    }
}
```

This was extracted from **workbench/formativ/chat/src/Formativ/Chat/Chat.php**.

The change we've made is to exclude the logic which prevented messages from being sent to the user from which they came. All messages will essentially be sent to everyone on the server now.

## Finishing Up The Template

The final change is to the index template, as we changed the model structure and need to adjust for this in the template:

```
<script
    type="text/x-handlebars"
    data-template-name="index"
>
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <h1>Laravel 4 Chat</h1>
                <table class="table table-striped">
                    @{{#each message in model}}
                        <tr>
                            <td @{{bind-attr class="message.user_id_class"}}>
                                @{{message.user_name}}
                            </td>
                            <td>
                                @{{message.message}}
                            </td>
                        </tr>
                    @{{/each}}
                </table>
            </div>
        </div>
        <div class="row">
            <div class="col-md-12">
```

```
                <div class="input-group">
                    @{{input
                        type="text"
                        value=command
                        class="form-control"
                    }}
                    <span class="input-group-btn">
                        <button
                            class="btn btn-default"
                            @{{action "send"}}
                        >
                            Send
                        </button>
                    </span>
                </div>
            </div>
        </div>
    </div>
</script>
```

This was extracted from **app/views/index/index.blade.php**.

You'll notice, apart from us using different field names; that we've change how the loop is done, the class added to each "label" cell and how the input field is generated.

The reason for the change to the loop structure is simply so that you can see another way to represented enumerable data in handlebars. Where previously we used a simple **{{#each}}** tag, we're now being more explicit about the data we want to iterate.

We add a special class on each "label" cell as we need to target these cells and change their contents, in the event that a user decides to change their name.

Finally, we change how the input field is generated because we need to bind its value to the **IndexController**'s command property. This format allows that succinctly.

You can learn more about Ember JS at: **http://emberjs.com/**.

## Note On Nginx

For persistent sockets to remain open, Apache needs to keep a single process thread occupied. This is a problem as it will consume vast amounts of RAM over a shorter time period than non-persistent connections would. For this reason; I highly recommend using either Nginx or an event-based language to create your chat application.

It’s not that Apache sucks; this is just not the sort of thing it was designed for. Nginx, on the other hand, is event-based and doesn’t hold onto a whole thread while it waits for activity through the open socket.

You can learn more about Nginx at: **http://wiki.nginx.org/Main**.

# Multisites

If you've ever had to build an application that shares some business logic, and even interface logic, across multiple interfaces then you will undoubtedly have debated the merits of instead creating multiple applications or finding some way to handle each interface within the same codebase.

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

## Note on Operating Systems

I work on a Macbook, and deal with Linux servers every day. I seldom interact with Windows-based machines. As a result; most of the work I do is never run on Windows. Remember this as you follow this chapter—I will not show how to do things on Windows because it's dead to me. If you are forced to use it; then you have my sympathies.

## Note on Server Setup

We're going to look at how to create virtual hosts in Apache2 and Nginx, but we won't look at how to get those systems running int he first place. It's not something I consider particularly tricky, and the internet is filled with wonderful tutorials on the subject.

## Note on Dutch

I use it in this chapter, but I don't really speak it. Google Translate does. Blame Google Translate.

## Virtual Hosts

It may surprise you to know that single domain names do not translate into single web servers. Modern web servers have the ability to load many different domains, and these domains are often referred to as Virtual Hosts or Virtual Domains. We're going to see how to use them with Laravel 4.

## Adding Virtual Host Entries

When you type an address into your browser, and hit enter; your browser goes through a set of steps in order to get you the web page you want. First, it queries what's called a hosts file to see if the address you typed in is a reference to a local IP address. If not found, the browser then queries whatever DNS servers are available to the operating system.

A DNS server compares an address (e.g. example.com) with a list of IP addresses it has on file. If an IP address is found; the browser's request is forwarded to it and the web page is delivered.

> This is a rather simplified explanation of the steps involved. You can probably find more details at: **http://en.wikipedia.org/wiki/Internet**.

In a sense; the hosts file acts as a DNS server before any remote requests are made. It follows that the best place to add references to local IP addresses (local web servers) is in the hosts file.

To do that, open the hosts file:

```
sudo vim /etc/hosts
```

> If that command gives you errors then you can try running it without the **sudo** part. Sudo is (simply speaking) a way to run a command at the highest permission level possible; to ensure it executes correctly. If may be that you do not have sufficient privileges to run commands as root (with **sudo**).

> Vim is a teminal-based text editor. If you're uneasy using it; what we're after is editing **/etc/hosts**.

On a new line, at the bottom of **/etc/hosts**, add:

```
127.0.0.1 dev.tutorial
```

This was extracted from **/etc/hosts**.

This line tells the operating system to send requests to dev.tutorial to **127.0.0.1**. You'll want to replace **127.0.0.1** with the IP address of the server you are using for this tutorial (assuming it's not LAMP/MAMP on your local machine) and **dev.tutorial** with whatever you want the virtual host to be.

You'll need to repeat this process for each virtual host you add. If your application has (for example) three different domains pointing to it; then you will need to set three of these up to test them locally.

## Creating Apache 2 Virtual Hosts

Apache2 virtual host entries are created by manipulating the configuration files usually located in **/etc/apache2/sites-available/**. You can edit the default file, and add the entries to it, or you can create new files.

If you choose to create new files for your virtual host entries, then you will need to symlink them to **/etc/apache2/sites-enabled/*** where ***** matches the name of the files you create in the **sites-available** folder.

The expanded syntax of virtual host entries can be quite a bit to type/maintain; so I have a few go-to lines I usually use:

```
<VirtualHost *:80>
    DocumentRoot /var/www/site1
    ServerName dev.site1
</VirtualHost>
```

This was extracted from **/etc/apache2/sites-available/default**.

The **ServerName** property can be anything you like—it's the address you will use in your browser. The **DocumentRoot** property needs to point to an existing folder in the filesystem of the web server machine.

> You can learn more about virtual host entries, in Apache2, at: **http://httpd.apache.org/docs/current/vhosts/examples.html**.

## Creating Nginx Virtual Hosts

Similar to Apache2, Nginx virtual host entries are created by manipulating the configuration files usually located in **/etc/nginx/sites-available/**. You can edit the **default** file, and add the entries to it, or you can create new files.

> If you choose to create new files for your virtual host entries, then you will need to symlink them to **/etc/nginx/sites-enabled/*** where ***** matches the name of the files you create in the **sites-available** folder.

The expanded syntax of virtual host entries can be quite a bit to type/maintain; so I have a few go-to lines I usually use:

```
server {
    listen 80;
    server_name dev.site1;
    root /var/www/site1;
    index index.php;

    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_index index.php
        include fastcgi_params;
    }
}
```

This was extracted from **/etc/nginx/sites-available/default**.

This site definition assumes you are using PHP5 FPM. Getting this installed and running is out of the scope of this tutorial, but there are plenty of great tutorials on the subject.

You can learn more about virtual host entries, in Nginx, at: **https://www.digitalocean.com/community/articles/how-to-set-up-nginx-virtual-hosts-server-blocks-on-ubuntu-12-04-lts--3**.

## Environments

Laravel 4 employs a system of execution environments. Think of these as different contexts in which different configuration files will be loaded; determined by the host name of the machine or command flags.

To illustrate this concept; think of the differences between developing and testing your applications locally and running them on a production server. You will need to target different databases, use different caching schemes etc.

This can be achieved, deterministically, by setting the environments to match the host names of the machines the code will be executed (local and production respectively) and having different sets of configuration files for each environment.

When a Laravel 4 application is executed, it will determine the environment that you are running it in, and adjust the path to configuration files accordingly. This approach has two caveats.

The first caveat is that the PHP configuration files in **app/config** are always loaded and environment-based configuration files are then loaded on top of them. If you have a database set up in **app/config/database.php** and nothing in **app/config/production/database.php**, the global database configuration details will apply.

The second caveat is that you can override the environment Laravel 4 would otherwise use by supplying an environment flag to artisan commands:

```
php artisan migrate --env=local
```

This will tell artisan to execute the database migrations in the local environment, whether or not the environment you are running it in is local.

This is important to multisites because Laravel 4 configuration files have the ability to connect to different database, load different views and send different emails based on environmental configuration.

## Note on Running Commands in Local Environment

The moment you add multiple environments to your application, you create the possibility that artisan commands might be run on the incorrect environment.

Just because Laravel 4 is capable of executing in the correct environment doesn't mean you will always remember which environment you are in or which environment you should be in…

Start learning to provide the environment for every artisan command, when you're working with multiple environments. It's a good habit to get into.

## Using Site-Specific Views

One of the benefits of multiple environment-specific configuration sets is that we can load different views for different sites. Let's begin by creating a few:

```
127.0.0.1 dev.www.tutorial-laravel-4-multisites
127.0.0.1 dev.admin.tutorial-laravel-4-multisites
```

This was extracted from **/etc/hosts**.

You can really create any domains you want, but for this part we're going to need multiple domains pointing to our testing server so that we can actually see different view files being loaded, based on domain.

Next, update your **app/bootstrap/start.php** file to include the virtual host domains you've created:

```
$env = $app->detectEnvironment([
    "www"   => ["dev.www.tutorial-laravel-4-multisites"],
    "admin" => ["dev.admin.tutorial-laravel-4-multisites"]
]);
```

This was extracted from **/bootstrap/start.php**.

Phil Sturgeon made an excellent point[a] about not using URL's to toggle environments, as somebody can easily set a hostname to match your development environment and point it to your production environment to cause unexpected results. This can lead to people gleaning more information than you would expect via debugging information or backtraces.

[a]http://www.reddit.com/r/PHP/comments/1qm8e3/tutorial_multisites_with_laravel_4/cdecd88

If you would rather determine the current environment via a server configuration property; you can pass a callback to the **detectEnvironment()** method:

```
$env = $app->detectEnvironment(function()
{
    return Input::server("environment", "development");
});
```

This was extracted from **/bootstrap/start.php**.

Clean out the **app/views** folder and create **www** and **admin** folders, each with their own **layout.blade.php** and **index/index.blade.php** files.

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Laravel 4 Multisites</title>
    </head>
    <body>
        @yield("content")
    </body>
</html>
```

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

```
@extends("layout")
@section("content")
    Welcome to our website!
@stop
```

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

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title>Laravel 4 Multisites – Admin</title>
    </head>
    <body>
        @yield("content")
    </body>
</html>
```

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

```
@extends("layout")
@section("content")
    Please log in to use the admin.
@stop
```

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

Let’s also prepare the routes and controllers for the rest of the tutorial, by updating both:

```
<?php

Route::any("/", [
    "as"   => "index/index",
    "uses" => "IndexController@indexAction"
]);
```

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

```
<?php

class IndexController
extends BaseController
{
    public function indexAction()
    {
        return View::make("index/index");
    }
}
```

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

In order for Laravel to know which views to use for each environment, we should also create the configuration files for the environments.

```
<?php

return [
    "paths" => [app_path() . "/views/www"]
];
```

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

```
<?php

return [
    "paths" => [app_path() . "/views/admin"]
];
```

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

These new configuration files tell Laravel 4 not only to look for views in the default directory but also to look within the environment-specific view folders we've set up. Going to each of these virtual host domains should now render different content.

I would recommend this approach for sites that need to drastically change their appearance in ways mere CSS couldn't achieve. If, for example, your different sites need to load different HTML or extra components then this is a good approach to take.

Another good time to use this kind of thing is when you want to separate the markup of a CMS from that of a client-facing website. Both would need access to the same business logic and storage system, but their interfaces should be vastly different.

I’ve also used this approach when I’ve needed to create basic mobile interfaces and rich interactive interfaces for the same brands…

> You can learn more about environments at: **http://laravel.com/docs/configuration#environment-configuration**.

## Using Site-Specific Routes

Another aspect to multiple-domain applications is how they can affect (and be affected by) the routes file. Consider the following route groups:

```php
Route::group([
    "domain" => "dev.www.tutorial-laravel-4-multisites"
], function()
{
    Route::any("/about", function()
    {
        return "This is the client-facing website.";
    });
});

Route::group([
    "domain" => "dev.admin.tutorial-laravel-4-multisites"
], function()
{
    Route::any("/about", function()
    {
        return "This is the admin site.";
    });
});
```

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

Aside from the basic routes Laravel 4 supports, it’s also possible to group routes within route groups. These groups provide an easy way of applying common logic, filtering and targeting specific domains.

Not only can we explicitly target our different virtual host domains, we can target all subdomains with sub-domain wildcard:

```
Route::group([
    "domain" => "dev.{sub}.tutorial-laravel-4-multisites"
], function()
{
    Route::any("/whoami", function($sub)
    {
        return "You are in the '" . $sub . "' sub-domain.";
    });
});
```

This was extracted from **app/routes.php**.

This functionality allows some pretty powerful domain-related configuration and logical branching!

## Translation

Laravel 4 includes a translation system that can greatly simplify developing multisites. Translated phrases are stored in configuration files, and these can be returned in views.

### Using Language Lookups

The simplest example of this requires two steps: we need to add the translated phrases and we need to recall them in a view. To begin with; we're going to add a few English and Dutch phrases to the configuration files:

```
<?php

return [
    "instructions" => "Follow these steps to operate cheese:",
    "step1"        => "Cut the cheese.",
    "step2"        => "Eat the :product!",
    "product"      => "cheese"
];
```

This file should be saved as **app/lang/en/steps.php**.

```php
<?php

return [
    "instructions" => "Volg deze stappen om kaas te bedienen:",
    "step1"        => "Snijd de kaas.",
    "step2"        => "Eet de :product!",
    "product"      => "kaas"
];
```

This file should be saved as **app/lang/nl/steps.php**.

```php
<?php

class IndexController
extends BaseController
{
    public function indexAction()
    {
        App::setLocale("en");

        if (Input::get("lang") === "nl")
        {
            App::setLocale("nl");
        }

        return View::make("index/index");
    }
}
```

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

This will toggle the language based on a querystring parameter. The next step is actually using the phrases in views:

```
@extends("layout")
@section("content")
    <h1>
        {{ Lang::get("steps.instructions") }}
    </h1>
    <ol>
        <li>
            {{ trans("steps.step1") }}
        </li>
        <li>
            {{ trans("steps.step2", [
                "product" => trans("steps.product")
            ]) }}
        </li>
    </ol>
@stop
```

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

The **Lang::get()** method gets translated phrases out of the configuration files (in this case steps.instructions). The **trans()** method serves as a helpful alias to this.

You may also have noticed that **step2** has a strange **:product** placeholder. This allows the insertion of variable data into translation phrases. We can pass these in the optional second parameter of **Lang::get()** and **trans()**.

You can learn more about the Localization class at: **http://laravel.com/docs/localization**.

## Using Language Lookups in Packages

We're not going to go into the details of how to create packages in Laravel 4, except to say that it's possible to have package-specific translation. If you've set a package up, and registered its service provider in the application configuration, then you should be able to insert the following lines:

```php
public function boot()
{
    $this->package("formativ/multisite", "multisite");
}
```

This was extracted from **workbench/formativ/multisite/src/Formativ/Multisite/MultisiteServiceProvider.php**.

…in the service provider. Your package will probably have a different vendor/package name, and you need to pay particular attention to the second parameter (which is the alias to your package assets).

Add these translation configuration files also:

```php
<?php

return [
    "instructions" => "Do these:"
];
```

This file should be saved as **workbench/formativ/multisite/src/lang/en/steps.php**.

```php
<?php

return [
    "instructions" => "Hebben deze:"
];
```

This file should be saved as **workbench/formativ/multisite/src/lang/nl/steps.php**.

Finally, let's adjust the view to reflect the new translation phrase's location:

```
<h1>
    {{ Lang::get("multisite::steps.instructions") }}
</h1>
```

This was extracted from **app/views/www/index/index.blade.php**.

It's as easy at that!

You can learn more about package resources at: **http://laravel.com/docs/packages#package-configuration**.

## Caching Language Lookups

Getting translated phrases from the filesystem can be an expensive operation, in a big system. What would be even cooler is if we could use Laravel 4's built-in cache system to make repeated lookups more efficient.

To do this, we need to create a few files, and change some old ones:

```
// 'Lang' => 'Illuminate\Support\Facades\Lang',
"Lang" => "Formativ\Multisite\Facades\Lang",
```

This was extracted from **app/config/app.php**.

This tells Laravel 4 to load our own Lang facade in place of the one that ships with Laravel 4. We've got to make this facade…

```php
<?php

namespace Formativ\Multisite\Facades;

use Illuminate\Support\Facades\Facade;

class Lang
extends Facade
{
    protected static function getFacadeAccessor()
    {
        return "multisite.translator";
    }
}
```

> This file should be saved as **workbench/formativ/multisite/src/Formativ/Multisite/Facades/Lang.php**.

This is basically the same facade that ships with Laravel 4, but instead of returning **translator** it will return **multisite.translator**. We need to register this in our service provider as well:

```php
public function register()
{
    $this->app["multisite.translator"] =
         $this->app->share(function($app)
    {
        $loader = $app["translation.loader"];
        $locale = $app["config"]["app.locale"];
        $trans  = new Translator($loader, $locale);

        return $trans;
    });
}
```

This was extracted from **workbench/formativ/multisite/src/Formativ/Multisite/MultisiteServiceProvider.php**.

I've used very similar code to what can be found in **vendor/laravel/framework/src/Illuminate/-Translation/TranslationServiceProvider.php**. That's because I'm still using the old file-based loading system to get the translated phrases initially. We'll only return cached data in subsequent retrievals.

Lastly; we need to override how the translator fetches the data.

```php
<?php

namespace Formativ\Multisite;

use Cache;
use Illuminate\Translation\Translator as Original;

class Translator
extends Original
{
    public function get($key, array $replace = array(),
        $locale = null)
    {
        $cached = Cache::remember($key, 15,
            function() use ($key, $replace, $locale)
        {
            return parent::get($key, $replace, $local
        });

        return $cached;
    }
}
```

This file should be saved as **workbench/formativ/multisite/src/Formativ/Multisite/Translator.php**.

The process is as simple as subclassing the Translator class and caching the results of the first call to the **get()** method.

You can learn more about facades at: **http://laravel.com/docs/facades**.

## Creating Multi-Language Routes

Making multi-language routes may seem needless, but link are important to search engines, and the users who have to remember them.

To make multi-language routes, we first need to create some link terms:

```php
<?php

return [
    "cheese" => "cheese",
    "create" => "create",
    "update" => "update",
    "delete" => "delete"
];
```

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

```php
<?php

return [
    "cheese" => "kaas",
    "create" => "creëren",
    "update" => "bijwerken",
    "delete" => "verwijderen"
];
```

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

Next, we need to modify the **app/routes.php** file to dynamically create the routes:

```php
$locales = [
    "en",
    "nl"
];

foreach ($locales as $locale)
{
    App::setLocale($locale);

    $cheese = trans("routes.cheese");
    $create = trans("routes.create");
    $update = trans("routes.update");
    $delete = trans("routes.delete");

    Route::any($cheese . "/" . $create,
        function() use ($cheese, $create)
    {
        return $cheese . "/" . $create;
    });

    Route::any($cheese . "/" . $update,
        function() use ($cheese, $update)
    {
        return $cheese . "/" . $update;
    });

    Route::any($cheese . "/" . $delete,
        function() use ($cheese, $delete)
    {
        return $cheese . "/" . $delete;
    });
}
```

This was extracted from **app/routes.php**.

We hard-code the locale names because it's the most efficient way to return them in the routes file. Routes are determined on each application request, so we dare not do a file lookup…

What this is basically doing is looping through the locales and creating routes based on translated phrases specific to the locale that's been set. It's a simple, but effective, mechanism for implementing multi-language routes.

If you would like to review the registered routes (for each language), you can run the following command:

```
php artisan routes
```

You can learn more about routes at: **http://laravel.com/docs/routing**.

## Creating Multi-Language Content

Creating multi-language content is nothing more than having a few extra database table fields to hold the language-specific data. To do this; let's make a migration and seeder to populate our database:

```php
<?php

use Illuminate\Database\Migrations\Migration;

class CreatePostTable
extends Migration
{
    public function up()
    {
        Schema::create("post", function($table)
        {
            $table->increments("id");
            $table->string("title_en");
            $table->string("title_nl");
            $table->text("content_en");
            $table->text("content_nl");
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists("post");
    }
}
```

This file should be saved as **app/database/migrations/0000*00*00_000000_CreatePostTable.php**.

```php
<?php

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

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

```php
<?php

class PostTableSeeder
extends DatabaseSeeder
{
    public function run()
    {
        $posts = [
            [
                "title_en"   => "Cheese is the best",
                "title_nl"   => "Kaas is de beste",
                "content_en" => "Research has shown...",
                "content_nl" => "Onderzoek heeft aangetoond..."
            ]
        ];
        DB::table("post")->insert($posts);
    }
}
```

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

To get all of this in the database, we need to check the settings in app/config/database.php and run the following command:

```
php artisan migrate --seed --env=local
```

This should create the post table and insert a single row into it. To access this table/row, we'll make a model:

```php
<?php

class Post
extends Eloquent
{
    protected $table = "post";
}
```

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

We'll not make a full set of views, but let's look at what this data looks like straight out of the database. Update your **IndexController** to fetch the first post:

```php
<?php

class IndexController
extends BaseController
{
    public function indexAction($sub)
    {
        App::setLocale($sub);
        return View::make("index/index", [
            "post" => Post::first()
        ]);
    }
}
```

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

Next, update the **index/index.blade.php** template:

```
<h1>
    {{ $post->title_en }}
</h1>
<p>
    {{ $post->content_en }}
</p>
```

This was extracted from **app/views/www/index/index.blade.php**.

If you managed to successfully run the migrations, have confirmed you have at least one table/row in the database, and made these changes; then you should be seeing the english post content in your **index/index** view.

That's fine for the English site, but what about the other languages? We don't want to have to add additional logic to determine which fields to show. We can't use the translation layer for this either, because the translated phrases are in the database.

The answer is to modify the **Post** model:

```
public function getTitleAttribute()
{
    $locale = App::getLocale();
    $column = "title_" . $locale;
    return $this->{$column};
}

public function getContentAttribute()
{
    $locale = App::getLocale();
    $column = "content_" . $locale;
    return $this->{$column};
}
```

This was extracted from **app/models/Post.php**.

We've seen these attribute accessors before. They allow us to intercept calls to **$post->title** and **$post->content**, and provide our own return values. In this case; we return the locale-specific field value. Naturally we can adjust the view use:

```
<h1>
    {{ $post->title }}
</h1>
<p>
    {{ $post->content }}
</p>
```

This was extracted from **app/views/www/index/index.blade.php**.

We can use this in all the domain-specific views, to render locale-specific database data.

# E-Commerce

One of the benchmarks of any framework is how well it fares in the creation of an e-commerce application. Laravel 4 is up to the challenge; and, in this chapter, we're going to create an online shop.

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

## Note on Sanity

There is no way that an e-commerce platform, built in 40 minutes, can be production-ready. Please do not attempt to conduct any real business on the basis of this chapter; without having taken the necessary precautions.

This chapter is a guide, an introduction, a learning tool. It is not meant to be the last word in building e-commerce platforms. I do not want to hear about how all your customers want to sue you because you slapped your name and logo on the GitHub source-code and left all reason behind.

## Getting Started

In this chapter; we will create a number of database objects, which will later be made available through API endpoints. We'll then use these, together with AngularJS, to create an online shop. We'll finish things off with an overview of creating PDF documents dynamically.

> You need to understand a bit of JavaScript for this chapter. There's too much functionality to also cover the basics, so you should learn them elsewhere.

## 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.

## Installing Other Dependencies

Our application will do loads of things, so we'll need to install a few dependencies to lighten the workload.

### AngularJS

> AngularJS is an open-source JavaScript framework, maintained by Google, that assists with running single-page applications. Its goal is to augment browser-based applications with model–view–controller capability, in an effort to make both development and testing easier.

Angular allows us to create a set of interconnected components for things like product listings, shopping carts and payment pages. That's not all it can do; but that's all we're going to do with it (for now).

To get started, all we need to know is how to link the AngularJS library to our document:

```
<script
  type="text/javascript"
  src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.0rc1/angular.js"
></script>
```

If this seems too easy to be enough, fret not. AngularJS includes no stylesheets or any other resources. It's purely a JavaScript framework, so that script is all you need. If you prefer to keep scripts loading from your local machine, just download the contents of the file at the end of that src attribute.

## Bootstrap

> Sleek, intuitive, and powerful mobile first front-end framework for faster and easier web development.

Bootstrap has become somewhat of a standard in modern applications. It's often used as a CSS reset, a wire-framing tool and even as the baseline for all application CSS. We're going to use it to neaten up our simple HTML.

It's available for linking (as we did with AngularJS):

```
<link
  type="text/css"
  rel="stylesheet"
  href="//netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap.min.css"
/>
<link
  type="text/css"
  rel="stylesheet"
  href="//netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap-theme.min.css"
/>
<script
        type="text/javascript"
        src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"
></script>
```

You can also download it, and serve it directly from the public folder. As it contains CSS and fonts; be sure to update all paths to the relevant images and/or fonts that come bundled with it.

## DOMPDF

> At its heart, dompdf is (mostly) CSS 2.1 compliant HTML layout and rendering engine written in PHP. It is a style-driven renderer: it will download and read external stylesheets, inline style tags, and the style attributes of individual HTML elements. It also supports most presentational HTML attributes.

DOMPDF essentially takes HTML documents and converts them into PDF files. If you've ever had to produce PDF files programatically (at least in PHP) then this library should be like the singing of angels to your ears. It really is epic.

To install this guy; we need to add a composer dependency:

```
"require" : {
  "dompdf/dompdf" : "dev-master"
}
```

This was extracted from **composer.json**.

Obviously you're going to have other dependencies in your **composer.js** file (especially when working on Laravel 4 projects), so just make it play nice with the other dependencies already in there.

## Stripe

> Stripe is a simple, developer-friendly way to accept payments online. We believe that enabling transactions on the web is a problem rooted in code, not finance, and we want to help put more websites in business.

We're going to look at accepting payments with Stripe. It's superior to the complicated payment processes of other services, like PayPal.

Installation is similar to DOMPDF:

```
"require" : {
  "stripe/stripe-php" : "dev-master"
}
```

This was extracted from **composer.json**.

### Faker

> Faker is a PHP library that generates fake data for you. Whether you need to bootstrap your database, create good-looking XML documents, fill-in your persistence to stress test it, or anonymize data taken from a production service, Faker is for you.

We're going to use Faker for populating our database tables (through seeders) so we have a fresh set of data to use each time we migrate our database objects.

To install Faker; add another Composer dependency:

```
"require" : {
  "fzaninotto/faker" : "dev-master"
}
```

This was extracted from **composer.json**.

Remember to make this dependency behave nicely with the others in **composer.json**. A simple **composer update** should take us the rest of the way to being able to use DOMPDF, Stripe and Faker in our application.

## Creating Database Objects

For our online shop; we're going to need categories for products to be sorted into, products and accounts. We'll also need orders and order items, to track which items have been sold.

### Creating Migrations

We listed five migrations, which we need to create. Fire up a terminal window and use the following command to create them:

```
php artisan migrate:make CreateCategoryTable
```

You can repeat this process five times, or you can use the first as a copy-and-paste template for the remaining four migrations.

After a bit of modification; I have the following migrations:

```
<?php

use Illuminate\Database\Migrations\Migration;

class CreateAccountTable
extends Migration
{
  public function up()
  {
    Schema::create("account", function($table)
    {
      $table->engine = "InnoDB";

      $table->increments("id");
      $table->string("email");
      $table->string("password");
      $table->dateTime("created_at");
      $table->dateTime("updated_at");
      $table->dateTime("deleted_at");
    });
  }

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

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

If you're wondering why I am creating the timestamp fields explicitly, instead of using **$table->timestamps()** and **$table->softDeletes()**; it's because I prefer to know what the field names are. I would rather depend on these three statements than the two magic methods. Perhaps the field names will be configurable in future. Perhaps it will burn me. For now I'm declaring them.

```php
<?php

use Illuminate\Database\Migrations\Migration;

class CreateCategoryTable
extends Migration
{
  public function up()
  {
    Schema::create("category", function($table)
    {
      $table->engine = "InnoDB";

      $table->increments("id");
      $table->string("name");
      $table->dateTime("created_at");
      $table->dateTime("updated_at");
      $table->dateTime("deleted_at");
    });
  }

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

This file should be saved as **app/database/migrations/nnnn_nn_nn_nnnnnn_CreateCategory-Table.php**.

```php
<?php

use Illuminate\Database\Migrations\Migration;

class CreateOrderItemTable
extends Migration
{
  public function up()
  {
    Schema::create("order_item", function($table)
    {
      $table->engine = "InnoDB";

      $table->increments("id");
      $table->integer("order_id");
      $table->integer("product_id");
      $table->integer("quantity");
      $table->float("price");
      $table->dateTime("created_at");
      $table->dateTime("updated_at");
      $table->dateTime("deleted_at");
    });
  }

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

This file should be saved as **app/database/migrations/nnnn_nn_nn_nnnnnn_Create-OrderItemTable.php**.

We add another price field for each order item because so that changes in product pricing don't affect orders that have already been placed.

```
<?php

use Illuminate\Database\Migrations\Migration;

class CreateOrderTable
extends Migration
{
  public function up()
  {
    Schema::create("order", function($table)
    {
      $table->engine = "InnoDB";

      $table->increments("id");
      $table->integer("account_id");
      $table->dateTime("created_at");
      $table->dateTime("updated_at");
      $table->dateTime("deleted_at");
    });
  }

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

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

```php
<?php

use Illuminate\Database\Migrations\Migration;

class CreateProductTable
extends Migration
{
  public function up()
  {
    Schema::create("product", function($table)
    {
      $table->engine = "InnoDB";

      $table->increments("id");
      $table->string("name");
      $table->integer("stock");
      $table->float("price");
      $table->dateTime("created_at");
      $table->dateTime("updated_at");
      $table->dateTime("deleted_at");
    });
  }

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

This file should be saved as **app/database/migrations/nnnn_nn_nn_nnnnnn_CreateProduct-Table.php**.

There's nothing particularly special about these - we've create many of them before. What is important to note is that we're calling the traditional user table account.

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

The relationships might not yet be apparent, but we'll see them more clearly in the models…

## Creating Models

We need to create the same amount of models. I've gone ahead and created them with table names matching those defined int he migrations. I've also added the relationship methods:

```php
<?php

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

class Account
extends Eloquent
implements UserInterface, RemindableInterface
{
  protected $table = "account";

  protected $hidden = ["password"];

  protected $guarded = ["id"];

  protected $softDelete = true;

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

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

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

  public function orders()
  {
    return $this->hasMany("Order");
  }
}
```

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

```
<?php

class Category
extends Eloquent
{
  protected $table = "category";

  protected $guarded = ["id"];

  protected $softDelete = true;

  public function products()
  {
    return $this->hasMany("Product");
  }
}
```

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

```
<?php

class Order
extends Eloquent
{
  protected $table = "order";

  protected $guarded = ["id"];

  protected $softDelete = true;

  public function account()
  {
    return $this->belongsTo("Account");
  }

  public function orderItems()
  {
    return $this->hasMany("OrderItem");
  }

  public function products()
  {
    return $this->belongsToMany("Product", "order_item");
  }
}
```

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

```
<?php

class OrderItem
extends Eloquent
{
  protected $table = "order_item";

  protected $guarded = ["id"];

  protected $softDelete = true;

  public function product()
  {
    return $this->belongsTo("Product");
  }

  public function order()
  {
    return $this->belongsTo("Order");
  }
}
```

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

```
<?php

class Product
extends Eloquent
{
  protected $table = "product";

  protected $guarded = ["id"];

  protected $softDelete = true;

  public function orders()
  {
    return $this->belongsToMany("Order", "order_item");
  }

  public function orderItems()
  {
    return $this->hasMany("OrderItem");
  }

  public function category()
  {
    return $this->belongsTo("Category");
  }
}
```

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

I'm using a combination of one-to-many relationships and many-to-many relationships (through the **order_item**) table. These relationships can be expressed as:

1. Categories have many products.
2. Accounts have many orders.
3. Orders have many items (directly) and many products (indirectly).

We can now being to populate these tables with fake data, and manipulate them with API endpoints.

> You can learn more about models at: **http://laravel.com/docs/eloquent**.

## Creating Seeders

Having installed Faker; we're going to use it to populate the database tables with fake data. We do this for two reasons. Firstly, using fake data is safer than using production data.

> Have you ever been writing a script that sends out emails and used some dummy copy while you're building it? Ever used some cheeky words in that content? Ever accidentally sent that email out to 10,000 real customers email addresses? Ever been fired for losing a company north of £200,000?
>
> I haven't, but I know a guy that has. Don't be that guy.

- Phil Sturgeon, Build APIs You Won't Hate

Secondly, Faker provides random fake data so we get to see what our models look like with random variable data. This will show us the oft-overlooked field limits and formatting errors that we tend to miss while using the same set of pre-defined seed data.

Using Faker is easy:

```php
<?php

class DatabaseSeeder
extends Seeder
{
  protected $faker;

  public function getFaker()
  {
    if (empty($this->faker))
    {
      $this->faker = Faker\Factory::create();
    }

    return $this->faker;
  }

  public function run()
  {
    $this->call("AccountTableSeeder");
  }
}
```

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

```php
<?php

class AccountTableSeeder
extends DatabaseSeeder
{
  public function run()
  {
    $faker = $this->getFaker();

    for ($i = 0; $i < 10; $i++)
    {
      $email    = $faker->email;
      $password = Hash::make("password");

      Account::create([
        "email"    => $email,
        "password" => $password
      ]);
    }
  }
}
```

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

The first step is to create an instance of the **FakerGenerator** class. We do this by calling the **FakerFactory::create()** method and assigning it to a protected property.

Then, in **AccountTableSeeder**, we loop ten times; creating different accounts. Each account has a random email address, but all of them share the same hashed password. This is so that we will be able to log in with any of these accounts to interact with the rest of the application.

We can actually test this process, to see how the data is created, and how we can authenticate against it. Seed the database, using the following command:

```
php artisan migrate:refresh --seed
```

Depending on whether you have already migrated the schema; this may fail. If this happens, you can try the following commands:

```
php artisan migrate
php artisan db:seed
```

The refresh migration method actually reverses all migrations and re-migrates them. If you've not migrated before using it - there's a change that the missing tables will cause problems.

You should see ten account records, each with a different email address and password hash. We can attempt to authenticate with one of these. To do this; we need to adjust the auth settings:

```php
<?php

return [
  "driver"   => "eloquent",
  "model"    => "Account",
  "table"    => "account",
  "reminder" => [
    "email"  => "email/request",
    "table"  => "token",
    "expire" => 60
  ]
];
```

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

Fire up a terminal window and try the following commands:

```
php artisan tinker
```

```php
dd(Auth::attempt([
  "email"    => [one of the email addresses],
  "password" => "password"
]));
```

You'll need to type it in a single line. I have added the whitespace to make it more readable.

If you see bool(true) then the details you entered will allow a user to log in. Now, let's repeat the process for the other models:

```php
public function getFaker()
{
  if (empty($this->faker))
  {
    $faker = Faker\Factory::create();
    $faker->addProvider(new Faker\Provider\Base($faker));
    $faker->addProvider(new Faker\Provider\Lorem($faker));
  }

  return $this->faker = $faker;
}
```

This was extracted from **app/database/seeds/DatabaseSeeder.php**.

We've modified the **getFaker()** method to add things called providers. Providers are like plugins for Faker; which extend the base array of properties/methods that you can query.

```php
<?php

class CategoryTableSeeder
extends DatabaseSeeder
{
  public function run()
  {
    $faker = $this->getFaker();

    for ($i = 0; $i < 10; $i++)
    {
      $name = ucwords($faker->word);

      Category::create([
        "name" => $name
      ]);
    }
  }
}
```

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

```php
<?php

class ProductTableSeeder
extends DatabaseSeeder
{
  public function run()
  {
    $faker = $this->getFaker();

    $categories = Category::all();

    foreach ($categories as $category)
    {
      for ($i = 0; $i < rand(-1, 10); $i++)
      {
        $name  = ucwords($faker->word);
        $stock = $faker->randomNumber(0, 100);
        $price = $faker->randomFloat(2, 5, 100);

        Product::create([
          "name"     => $name,
          "stock"     => $stock,
          "price"     => $price,
          "category_id" => $category->id
        ]);
      }
    }
  }
}
```

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

Here, we use the **randomNumber()** and **randomFloat()** methods. What's actually happening, when you request a property value, is that Faker invokes a method of the same name (on one of the providers). We can just as easily use the **$faker->word()** means the same as **$faker->word**. Some of the methods (such as the **random*()** methods we've used here) take arguments, so we provide them in the method form.

```php
<?php

class OrderTableSeeder
extends DatabaseSeeder
{
  public function run()
  {
    $faker = $this->getFaker();

    $accounts = Account::all();

    foreach ($accounts as $account)
    {
      for ($i = 0; $i < rand(-1, 5); $i++)
      {
        Order::create([
          "account_id" => $account->id
        ]);
      }
    }
  }
}
```

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

```php
<?php

class OrderItemTableSeeder
extends DatabaseSeeder
{
  public function run()
  {
    $faker = $this->getFaker();

    $orders   = Order::all();
    $products = Product::all()->toArray();

    foreach ($orders as $order)
    {
      $used = [];

      for ($i = 0; $i < rand(1, 5); $i++)
      {
        $product = $faker->randomElement($products);

        if (!in_array($product["id"], $used))
        {
          $id       = $product["id"];
          $price    = $product["price"];
          $quantity = $faker->randomNumber(1, 3);

          OrderItem::create([
            "order_id"   => $order->id,
            "product_id" => $id,
            "price"      => $price,
            "quantity"   => $quantity
          ]);

          $used[] = $product["id"];
        }
      }
    }
  }
}
```

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

```php
public function run()
{
  $this->call("AccountTableSeeder");
  $this->call("CategoryTableSeeder");
  $this->call("ProductTableSeeder");
  $this->call("OrderTableSeeder");
  $this->call("OrderItemTableSeeder");
}
```

This was extracted from **app/database/seeds/DatabaseSeeder.php**.

You can learn more about seeders at: **http://laravel.com/docs/migrations#database-seeding** and more about Faker at: **https://github.com/fzaninotto/Faker**.

The order in which we call the seeders is important. We can't start populating orders and order items if we have no products or accounts in the database…

## Creating API Endpoints

We don't have time to cover all aspects of **creating APIs with Laravel 4**, so we'll confine our efforts to creating endpoints for the basic interactions that need to happen for our interface to function.

### Managing Categories And Products

The endpoints for categories and products are read-only in nature. We're not concentrating on any sort of administration interface, so we don't need to add or update them. We will need to adjust the quantity of available products, but we can do that from the OrderController, later on. For now, all we need is:

```php
<?php

class CategoryController
extends BaseController
{
  public function indexAction()
  {
    return Category::with(["products"])->get();
  }
}
```

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

```php
<?php

class ProductController
extends BaseController
{
  public function indexAction()
  {
    $query    = Product::with(["category"]);
    $category = Input::get("category");

    if ($category)
    {
      $query->where("category_id", $category);
    }

    return $query->get();
  }
}
```

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

The **CategoryController** has a single **index()** method which returns all categories, and the ProductController has a single **index()** method which returns all the products. If **?category=n** is provided to the **product/index** route, the products will be filtered by that category.

We do, of course, still need to add these routes:

```
Route::any("category/index", [
  "as"    => "category/index",
  "uses" => "CategoryController@indexAction"
]);

Route::any("product/index", [
  "as"    => "product/index",
  "uses" => "ProductController@indexAction"
]);
```

This was extracted from **app/routes.php**.

## Managing Accounts

For users to be able to buy products, they will need to log in. We've added some users to the database, via the **UserTableSeeder** class, but we should create an authentication endpoint:

```
<?php

class AccountController
extends BaseController
{
  public function authenticateAction()
  {
    $credentials = [
      "email"    => Input::get("email"),
      "password" => Input::get("password")
    ];

    if (Auth::attempt($credentials))
    {
      return Response::json([
        "status"  => "ok",
        "account" => Auth::user()->toArray()
      ]);
    }

    return Response::json([
      "status" => "error"
    ]);
  }
}
```

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

We'll also need to add a route for this:

```
Route::any("account/authenticate", [
  "as"   => "account/authenticate",
  "uses" => "AccountController@authenticateAction"
]);
```

This was extracted from **app/routes.php**.

It should now be possible to determine whether login credentials are legitimate; through the browser:

```
/account/authenticate?email=x&password=y
```

This will return an object with a status value. If the details were valid then an account object will also be returned.

## Managing Orders

Orders are slightly more complicated. We will need to be able to get all orders as well as orders by account. We'll also need to create new orders.

Let's begin by getting the orders:

```php
<?php

class OrderController
extends BaseController
{
  public function indexAction()
  {
    $query = Order::with([
      "account",
      "orderItems",
      "orderItems.product",
      "orderItems.product.category"
    ]);

    $account = Input::get("account");

    if ($account)
    {
      $query->where("account_id", $account);
    }

    return $query->get();
  }
}
```

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

This looks similar to the **indexAction()** method, in **ProductController**. We're also eager-loading the related "child" entities and querying by account (if that's given).

For this; we will need to add a route:

```php
Route::any("order/index", [
  "as"   => "order/index",
  "uses" => "OrderController@indexAction"
]);
```

This was extracted from **app/routes.php**.

You can learn more about controllers at: **http://laravel.com/docs/controllers** and more about routes at: **http://laravel.com/docs/routing**.

We'll deal with creating orders once we have the shopping and payment interfaces completed. Let's not get ahead of ourselves…

## Creating The Site With AngularJS

Will the API in place; we can begin the interface work. We're using AngularJS, which creates rich interfaces from ordinary HTML.

### Creating The Interface

AngularJS allows much of the functionality, we would previous have split into separate pages, to be in the same single-page application interface. It's not a unique feature of AngularJS, but rather the preferred approach to interface structure.

Because of this; we only need a single view:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Laravel 4 E-Commerce</title>
    <link
      type="text/css"
      rel="stylesheet"
      href="{{ asset("css/bootstrap.3.0.3.min.css") }}"
    />
    <link
      type="text/css"
      rel="stylesheet"
      href="{{ asset("css/bootstrap.theme.3.0.3.min.css") }}"
    />
    <link
      type="text/css"
      rel="stylesheet"
      href="{{ asset("css/shared.css") }}"
    />
    <script
      type="text/javascript"
      src="{{ asset("js/angularjs.1.2.4.min.js") }}"
    ></script>
    <script
      type="text/javascript"
      src="{{ asset("js/angularjs.cookies.1.2.4.min.js") }}"
    ></script>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col-md-12">
          <h1>
            Laravel 4 E-Commerce
          </h1>
        </div>
      </div>
      <div class="row">
        <div class="col-md-8">
          <h2>
            Products
          </h2>
          <div class="categories btn-group">
            <button
              type="button"
              class="category btn btn-default active"
            >
              All
            </button>
            <button
              type="button"
              class="category btn btn-default"
            >
              Category 1
            </button>
            <button
              type="button"
              class="category btn btn-default"
            >
              Category 2
            </button>
            <button
              type="button"
              class="category btn btn-default"
            >
              Category 3
            </button>
          </div>
          <div class="products">
            <div class="product media">
              <button
                type="button"
                class="pull-left btn btn-default"
              >
                Add to basket
              </button>
              <div class="media-body">
                <h4 class="media-heading">Product 1</h4>
                <p>
                  Price: 9.99, Stock: 10
                </p>
              </div>
            </div>
            <div class="product media">
              <button
                type="button"
                class="pull-left btn btn-default"
              >
                Add to basket
              </button>
              <div class="media-body">
                <h4 class="media-heading">Product 2</h4>
                <p>
                  Price: 9.99, Stock: 10
                </p>
              </div>
            </div>
            <div class="product media">
              <button
                type="button"
                class="pull-left btn btn-default"
              >
                Add to basket
              </button>
              <div class="media-body">
                <h4 class="media-heading">Product 3</h4>
                <p>
                  Price: 9.99, Stock: 10
                </p>
              </div>
            </div>
          </div>
        </div>
        <div class="col-md-4">
          <h2>
            Basket
          </h2>
          <form class="basket">
            <table class="table">
              <tr class="product">
                <td class="name">
                  Product 1
                </td>
                <td class="quantity">
                  <input
                    class="quantity form-control col-md-2"
                    type="number"
                    value="1"
                  />
                </td>
                <td class="product">
                  9.99
                </td>
                <td class="product">
                  <a
                    class="remove glyphicon glyphicon-remove"
                    href="#"
                  ></a>
                </td>
              </tr>
              <tr class="product">
                <td class="name">
                  Product 2
                </td>
                <td class="quantity">
                  <input
                    class="quantity form-control col-md-2"
                    type="number"
                    value="1"
                  />
                </td>
                <td class="product">
                  9.99
                </td>
                <td class="product">
                  <a
                    class="remove glyphicon glyphicon-remove"
                    href="#"
                  ></a>
                </td>
              </tr>
              <tr class="product">
                <td class="name">
                  Product 3
                </td>
                <td class="quantity">
                  <input
                    class="quantity form-control col-md-2"
                    type="number"
                    value="1"
                  />
                </td>
                <td class="product">
                  9.99
                </td>
                <td class="product">
                  <a
                    class="remove glyphicon glyphicon-remove"
                    href="#"
                  ></a>
                </td>
              </tr>
            </table>
          </form>
        </div>
      </div>
    </div>
    <script
      type="text/javascript"
      src="{{ asset("js/shared.js") }}"
    ></script>
  </body>
</html>
```

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

You'll notice that we also reference a **shared.css** file:

```
.products {
  margin-top: 20px;
}
.basket td {
  vertical-align: middle !important;
}
.basket .quantity input {
  width: 50px;
}
```

> This file should be saved as **public/css/shared.css**.

These changes to the view coincide with a modified **IndexController**:

```php
<?php

class IndexController
extends BaseController
{
  public function indexAction()
  {
    return View::make("index");
  }
}
```

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

## Making The Interface Dynamic

So far; we've set up the API and static interface, for our application. It's not going to be much use without the JavaScript to drive purchase functionality, and to interact with the API. Let's dive into AngularJS!

I should mention that I am by no means an AngularJS expert. I learned all I know of it, while writing this chapter, by following various guides. The point of this is not to teach AngularJS so much as it is to show AngularJS integration with Laravel 4.

AngularJS interfaces are just regular HTML and JavaScript. To wire the interface into the beginnings of an AngularJS application architecture; we have to add a script, and a few directives:

```html
<body ng-controller="main">
```

This was extracted from **app/views/index.blade.php**.

```
<div class="col-md-8" ng-controller="products">
```

This was extracted from **app/views/index.blade.php**.

```
<div class="col-md-4" ng-controller="basket">
```

This was extracted from **app/views/index.blade.php**.

```
<script
  type="text/javascript"
  src="{{ asset("js/shared.js") }}"
></script>
```

This was extracted from **app/views/index.blade.php**.

This script should be placed just before the **</body>** tag.

In addition to these modifications, we should also create the **shared.js** file:

```
var app = angular.module("app", ["ngCookies"]);

app.controller("main", function($scope) {
  console.log("main.init");

  this.shared = "hello world";

  $scope.main = this;
});

app.controller("products", function($scope) {
  console.log("products.init:", $scope.main.shared);

  $scope.products = this;
});

app.controller("basket", function($scope) {
  console.log("basket.init:", $scope.main.shared);

  $scope.basket = this;
});
```

This file should be saved as **public/js/shared.js**.

AngularJS implements the concept of modules - contains for modularising business and interface logic. We begin by creating a module (called app). This correlates with the **ng-app="app"** directive.

This demonstrates a powerful feature of AngularJS: the ability to have multiple applications on a single HTML page, and to make any element an application.

The remaining **ng-controller** directives define which controllers apply to which element. These match the names of the controllers which we have created. Controllers are nothing more than scoped functions. We assign the controller instances, and some shared data, to the **$scope** variable. This provides a consistent means of sharing data.

If you've linked everything correctly; you should see three console messages, letting you know that everything's working.

Let's popular the interface with real products. To achieve this; we need to request the products from the API, and render them (in a loop).

```
app.factory("CategoryService", function($http) {
  return {
    "getCategories": function() {
      return $http.get("/category/index");
    }
  };
});

app.factory("ProductService", function($http) {
  return {
    "getProducts": function() {
      return $http.get("/product/index");
    }
  };
});

app.controller("products", function(
  $scope,
  CategoryService,
  ProductService
) {

  var self = this;
  var categories = CategoryService.getCategories();

  categories.success(function(data) {
    self.categories = data;
  });

  var products = ProductService.getProducts();

  products.success(function(data) {
    self.products = data;
  });

  $scope.products = this;

});
```

This was extracted from **public/js/shared.js**.

There are some awesome things happening in this code. Firstly, we abstract the logic by which we get categories and products (from the API) into AngularJS's implementation of services. We also have access to the **$http** interface; which is a wrapper for **XMLHTTPRequest**, and acts as a replacement for other libraries (think jQuery) which we would have used before.

The two services each have a method for returning the API data, for categories and products. These methods return things, called promises, which are references to future-completed data. We attach callbacks to these, within the ProductController, which essentially update the controller data.

So we have the API data, but how do we render it in the interface? We do so with directives and data-binding:

```html
<div class="col-md-8" ng-controller="products">
  <h2>
    Products
  </h2>
  <div class="categories btn-group">
    <button
      type="button"
      class="category btn btn-default active"
    >
      All
    </button>
    <button
      type="button"
      class="category btn btn-default"
      ng-repeat="category in products.categories"
    >
      @{{ category.name }}
    </button>
  </div>
  <div class="products">
    <div
```

```
      class="product media"
      ng-repeat="product in products.products"
    >
      <button
        type="button"
        class="pull-left btn btn-default"
      >
        Add to basket
      </button>
      <div class="media-body">
        <h4 class="media-heading">@{{ product.name }}</h4>
        <p>
          Price: @{{ product.price }}, Stock: @{{ product.stock }}
        </p>
      </div>
    </div>
  </div>
</div>
```

This was extracted from **app/views/index.blade.php**.

If you're wondering how the interface is updated when the data is fetched asynchronously, but the good news is you don't need to. AngularJS takes care of all interface updates; so you can focus on the actual application! Open up a browser and see it working…

Now that we have dynamic categories and products, we should implement a filter so that products are swapped out whenever a user selects a category of products.

```
  <button
    type="button"
    class="category btn btn-default active"
    ng-click="products.setCategory(null)"
    ng-class="{ 'active' : products.category == null }"
  >
    All
  </button>
  <button
    type="button"
    class="category btn btn-default"
    ng-repeat="category in products.categories"
    ng-click="products.setCategory(category)"
    ng-class="{ 'active' : products.category.id == category.id }"
  >
    @{{ category.name }}
  </button>
</div>
<div class="products">
  <div
    class="product media"
    ng-repeat="product in products.products | filter : products.filterByCategory"
  >
```

This was extracted from **app/views/index.blade.php**.

We've added three new concepts here:

1. We're filtering the **ng-repeat** directive with a call to **products.filterByCategory()**. We'll create this in a moment, but it's important to understand that filter allows functions to define how the items being looped are filtered.
2. We've added **ng-click** directives. These directives allow the execution of logic when the element is clicked. We're targeting another method we're about to create; which will set the current category filter.
3. We've added **ng-class** directives. These will set the defined class based on controller/scope logic. If the set category filter matches that which the button is being created from; the active class will be applied to the button.

These directives, in isolation, will only cause errors. We need to add the JavaScript logic to back them up:

```
app.controller("products", function(
  $scope,
  CategoryService,
  ProductService
) {

  var self = this;

  // ...

  this.category = null;

  this.filterByCategory = function(product) {

    if (self.category !== null) {
      return product.category.id === self.category.id;
    }

    return true;

  };

  this.setCategory = function(category) {
    self.category = category;
  };

  // ...

});
```

This was extracted from **public/js/shared.js**.

Let's move on to the shopping basket. We need to be able to add items to it, remove items from it and change quantity values.

```
app.factory("BasketService", function($cookies) {

  var products = JSON.parse($cookies.products || "[]");

  return {

    "getProducts" : function() {
      return products;
    },

    "add" : function(product) {

      products.push({
        "id"       : product.id,
        "name"     : product.name,
        "price"    : product.price,
        "total"    : product.price * 1,
        "quantity" : 1
      });

      this.store();

    },

    "remove" : function(product) {

      for (var i = 0; i < products.length; i++) {

        var next = products[i];

        if (next.id == product.id) {
          products.splice(i, 1);
        }

      }

      this.store();

    },

    "update": function() {

      for (var i = 0; i < products.length; i++) {

        var product = products[i];
        var raw     = product.quantity * product.price;

        product.total = Math.round(raw * 100) / 100;

      }

      this.store();

    },

    "store" : function() {
      $cookies.products = JSON.stringify(products);
    },

    "clear" : function() {
      products.length = 0;
      this.store();
    }

  };

});
```

This was extracted from **public/js/shared.js**.

We're grouping all the basket-related logic together in **BasketService**. You may have noticed the reference to **ngCookies** (when creating the **app** module) and the extra script file reference (in **index.blade.php**). These allow us to use AngularJS's cookies module; for storing the basket items.

The **getProducts()** method returns the products. We need to store them as a serialised JSON array, so when we initially retrieve them; we parse them (with a default value of **"[]"**). The **add()** and **remove()** methods create and destroy special item objects. After each basket item operation; we need to persist the products array back to **$cookies**.

The **update()** method works out the total cost of each item; by taking into account the original price and the updated quantity. It also rounds this value to avoid floating-point calculation irregularities.

There's also a **store()** method which persists the products to **$cookies**, and a **clear()** method which removes all products.

The HTML, compatible with all this, is:

```
<div class="col-md-4" ng-controller="basket">
  <h2>
    Basket
  </h2>
  <form class="basket">
    <table class="table">
      <tr
        class="product"
        ng-repeat="product in basket.products track by $index"
      >
        <td class="name">
          @{{ product.name }}
        </td>
        <td class="quantity">
          <input
            class="quantity form-control col-md-2"
            type="number"
            ng-model="product.quantity"
            ng-change="basket.update()"
            min="1"
          />
        </td>
        <td class="product">
          @{{ product.total }}
        </td>
        <td class="product">
          <a
            class="remove glyphicon glyphicon-remove"
            ng-click="basket.remove(product)"
          ></a>
        </td>
      </tr>
    </table>
  </form>
</div>
```

This was extracted from **app/views/index.blade.php**.

We've change the numeric input element to use **ng-model** and **ng-change** directives. The first tells the input which dynamic (quantity) value to bind to, and the second tells the basket what to do if the input's value has changed. We already know that this means re-calculating the total cost of that product, and storing the products back in **$cookies**.

We've also added an **ng-click** directive to the remove link; so that the product can be removed from the basket.

You may have noticed track by **$index**, in the hg-repeat directive. This is needed as **ng-repeat** will error when it tries to iterate over the parsed JSON value (which is stored in $cookies). I found out about this at: **http://docs.angularjs.org/error/ngRepeat:dupes**.

We need to be able to remove the basket items, also. Let's modify the JavaScript/HTML to allow for this:

```
app.controller("products", function(
  $scope,
  CategoryService,
  ProductService,
  BasketService
) {

  // ...

  this.addToBasket = function(product) {
    BasketService.add(product);
  };

  // ...

});
```

This was extracted from **public/js/shared.js**.

```
<button
  type="button"
  class="pull-left btn btn-default"
  ng-click="products.addToBasket(product)"
>
  Add to basket
</button>
```

This was extracted from **app/views/index.blade.php**.

Try it out in your browser. You should be able to add items into the basket, change their quantities and remove them. When you refresh, all should display correctly.

## Completing Orders

To complete orders; we need to send the order item data to the server, and process a payment. We need to pass this endpoint an account ID (to link the orders to an account) which means we also need to add authentication…

```
app.factory("AccountService", function($http) {

  var account = null;

  return {
    "authenticate": function(email, password) {

      var request = $http.post("/account/authenticate", {
        "email"    : email,
        "password" : password
      });

      request.success(function(data) {
        if (data.status !== "error") {
          account = data.account;
        }
      });

      return request;

    },
    "getAccount": function() {
      return account;
    }
  };
});

app.factory("OrderService", function(
  $http,
  AccountService,
  BasketService
) {
  return {
    "pay": function(number, expiry, security) {

      var account  = AccountService.getAccount();
      var products = BasketService.getProducts();
      var items    = [];

      for (var i = 0; i < products.length; i++) {

        var product = products[i];

        items.push({
          "product_id" : product.id,
          "quantity"   : product.quantity
        });

      }

      return $http.post("/order/add", {
        "account"  : account.id,
        "items"    : JSON.stringify(items),
        "number"   : number,
        "expiry"   : expiry,
        "security" : security
      });
    }
  };
});
```

This was extracted from **public/js/shared.js**.

The **AccountService** object has a method for authenticating (with a provided email and password) and it returns the result of a **POST** request to **/account/authenticate**. It also has a **getAccount()** method which just returns whatever's in the account variable.

The **OrderService** object as a single method for sending order details to the server. I've bundled the payment particulars in with this method to save some time. The idea is that the order is created and paid for in a single process.

We need to amend the basket controller:

```
app.controller("basket", function(
  $scope,
  AccountService,
  BasketService,
  OrderService
) {

  // ...

  this.state    = "shopping";
  this.email    = "";
  this.password = "";
  this.number   = "";
  this.expiry   = "";
  this.secutiry = "";

  this.authenticate = function() {

    var details = AccountService.authenticate(self.email, self.password);

    details.success(function(data) {
      if (data.status == "ok") {
        self.state = "paying";
      }
    });

  }

  this.pay = function() {

    var details = OrderService.pay(
      self.number,
      self.expiry,
      self.security
    );

    details.success(function(data) {
      BasketService.clear();
      self.state = "shopping";
    });

  }

  // ...

});
```

This was extracted from **public/js/shared.js**.

We've added a state variable which tracks progress through checkout. We're also keeping track of the account email address and password as well as the credit card details. In addition; there are two new methods which will be triggered by the interface:

```
<div class="col-md-4" ng-controller="basket">
  <h2>
    Basket
  </h2>
  <form class="basket">
    <table class="table">
      <tr
        class="product"
        ng-repeat="product in basket.products track by $index"
        ng-class="{ 'hide' : basket.state != 'shopping' }"
      >
        <td class="name">
          @{{ product.name }}
        </td>
        <td class="quantity">
          <input
            class="form-control"
            type="number"
            ng-model="product.quantity"
            ng-change="basket.update()"
            min="1"
          />
        </td>
        <td class="product">
          @{{ product.total }}
        </td>
        <td class="product">
          <a
            class="remove glyphicon glyphicon-remove"
            ng-click="basket.remove(product)"
          ></a>
        </td>
      </tr>
      <tr>
        <td
          colspan="4"
          ng-class="{ 'hide' : basket.state != 'shopping' }"
        >
          <input
            type="text"
            class="form-control"
            placeholder="email"
            ng-model="basket.email"
          />
        </td>
      </tr>
      <tr>
        <td
          colspan="4"
          ng-class="{ 'hide' : basket.state != 'shopping' }"
        >
          <input
            type="password"
            class="form-control"
            placeholder="password"
            ng-model="basket.password"
          />
        </td>
      </tr>
      <tr>
        <td
          colspan="4"
          ng-class="{ 'hide' : basket.state != 'shopping' }"
        >
          <button
            type="button"
            class="pull-left btn btn-default"
            ng-click="basket.authenticate()"
          >
            Authenticate
          </button>
        </td>
      </tr>
      <tr>
        <td
          colspan="4"
          ng-class="{ 'hide' : basket.state != 'paying' }"
        >
          <input
            type="text"
            class="form-control"
            placeholder="card number"
            ng-model="basket.number"
          />
        </td>
      </tr>
      <tr>
        <td
          colspan="4"
          ng-class="{ 'hide' : basket.state != 'paying' }"
        >
          <input
            type="text"
            class="form-control"
            placeholder="expiry"
            ng-model="basket.expiry"
          />
        </td>
      </tr>
      <tr>
        <td
          colspan="4"
          ng-class="{ 'hide' : basket.state != 'paying' }"
        >
          <input
            type="text"
            class="form-control"
            placeholder="security number"
            ng-model="basket.security"
          />
        </td>
      </tr>
      <tr>
        <td
          colspan="4"
          ng-class="{ 'hide' : basket.state != 'paying' }"
        >
          <button
            type="button"
            class="pull-left btn btn-default"
            ng-click="basket.pay()"
          >
            Pay
          </button>
        </td>
      </tr>
    </table>
  </form>
</div>
```

This was extracted from **app/views/index.blade.php**.

We're using those **ng-class** directives to hide/show various table rows (in our basket). This lets us toggle the fields that users need to complete; and provides us with different buttons to dispatch the different methods in our basket controller.

You can learn more about AngularJS at: **http://angularjs.org**.

Finally; we need to tie this into our **OrderController**, where the orders are completed and the payments are processed…

## Accepting Payments

We're going to create a service provider to handle the payment side of things, and while we could go into great detail about how to do this; we don't have the time. Read Taylor's book, or mime, or the docs.

### Creating Orders

Before we start hitting Stripe up; we should create the endpoint for creating orders:

```
public function addAction()
{
  $validator = Validator::make(Input::all(), [
    "account" => "required|exists:account,id",
    "items"   => "required"
  ]);

  if ($validator->passes())
  {
    $order = Order::create([
      "account_id" => Input::get("account")
    ]);

    try
    {
      $items = json_decode(Input::get("items"));
    }
    catch (Exception $e)
    {
      return Response::json([
        "status" => "error",
        "errors" => [
          "items" => [
            "Invalid items format."
          ]
        ]
      ]);
    }

    $total = 0;

    foreach ($items as $item)
    {
      $orderItem = OrderItem::create([
        "order_id"   => $order->id,
        "product_id" => $item->product_id,
        "quantity"   => $item->quantity
      ]);

      $product = $orderItem->product;

      $orderItem->price = $product->price;
      $orderItem->save();

      $product->stock -= $item->quantity;
      $product->save();

      $total += $orderItem->quantity * $orderItem->price;
    }

    $result = $this->gateway->pay(
      Input::get("number"),
      Input::get("expiry"),
      $total,
      "usd"
    );

    if (!$result)
    {
      return Response::json([
        "status" => "error",
        "errors" => [
          "gateway" => [
            "Payment error"
          ]
        ]
      ]);
    }

    $account = $order->account;

    $document = $this->document->create($order);
    $this->messenger->send($order, $document);

    return Response::json([
      "status" => "ok",
      "order"  => $order->toArray()
    ]);
  }

  return Response::json([
    "status" => "error",
    "errors" => $validator->errors()->toArray()
  ]);
}
```

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

There are a few steps taking place here:

1. We validate that the account and items details have been provided.
2. We create an order item, and assign it to the prodded account.
3. We **json_decode()** the provided items and return an error if an invalid format has been provided.
4. We create individual order items for each provided item, and add the total value up.
5. We pass this value, and the order to a **GateWayInterface** class (which we'll create in a bit).
6. If this **pay()** method returns true; we create a document (with the **DocumentInterface** we're about to make) and send it (with the **MessengerInterface** we're also about to make).
7. Finally we return a status of **ok**.

The main purpose of this endpoint is to create the order (and order items) while passing the payment off to the service provider classes.

It would obviously be better to separate these tasks into their own classes/methods but there's simply not time for that sort of thing. Feel free to do it in your own application!

## Working The Service Provider

This leaves us with the service-provider part of things. I've gone through the motions to hook everything up (as you might have done following on from the chapter which covered this); and here is a list of the changes:

```
"providers" => array(

  // ...

  "Formativ\Billing\BillingServiceProvider"

),
```

This was extracted from **app/config/app.php**.

```
"autoload" : {

  // ...

  "psr-0": {
    "Formativ\\Billing": "workbench/formativ/billing/src/"
  }
}
```

This was extracted from **composer.json**.

```
<?php

namespace Formativ\Billing;

use App;
use Illuminate\Support\ServiceProvider;

class BillingServiceProvider
extends ServiceProvider
{
  protected $defer = true;

  public function register()
  {
    App::bind("billing.stripeGateway", function() {
      return new StripeGateway();
    });

    App::bind("billing.pdfDocument", function() {
      return new PDFDocument();
    });

    App::bind("billing.emailMessenger", function() {
      return new EmailMessenger();
    });
  }

```

```php
  public function provides()
  {
    return [
      "billing.stripeGateway",
      "billing.pdfDocument",
      "billing.emailMessenger"
    ];
  }
}
```

This file should be saved as **workbench/formativ/billing/src/Formativ/Billing/BillingServiceProvider.php**.

```php
<?php

namespace Formativ\Billing;

interface GatewayInterface
{
  public function pay(
    $number,
    $expiry,
    $amount,
    $currency
  );
}
```

This file should be saved as **workbench/formativ/billing/src/Formativ/Billing/GatewayInterface.php**.

```php
<?php

namespace Formativ\Billing;

interface DocumentInterface
{
  public function create($order);
}
```

This file should be saved as **workbench/formativ/billing/src/Formativ/Billing/DocumentInterface.php**.

```php
<?php

namespace Formativ\Billing;

interface MessengerInterface
{
  public function send(
    $order,
    $document
  );
}
```

This file should be saved as **workbench/formativ/billing/src/Formativ/Billing/MessengerInterface.php**.

These are all the interfaces (what some might consider the scaffolding logic) that we need. Let's make some payments!

You can learn more about service providers at: **http://laravel.com/docs/packages#service-providers**.

## Making Payments

As I've mentioned; we're going to receive payments through Stripe. You can create a new Stripe account at: **https://manage.stripe.com/register**.

You should already have the Stripe libraries installed, so let's make a GatewayInterface implementation:

```php
<?php

namespace Formativ\Billing;

use Stripe;
use Stripe_Charge;

class StripeGateway
implements GatewayInterface
{
  public function pay(
    $number,
    $expiry,
    $amount,
    $currency
  )
  {
    Stripe::setApiKey("...");

    $expiry = explode("/", $expiry);

    try
    {
      $charge = Stripe_Charge::create([
        "card" => [
          "number"    => $number,
          "exp_month" => $expiry[0],
          "exp_year"  => $expiry[1]
        ],
        "amount"   => round($amount * 100),
        "currency" => $currency
      ]);

      return true;
    }
    catch (Exception $e)
    {
      return false;
    }
  }
}
```

This file should be saved as **workbench/formativ/billing/src/Formativ/Billing/StripeGateway.php**.

Using the document (found at: https://github.com/stripe/stripe-php); we're able to create a test charge which goes through the Stripe payment gateway. You should be able to submit orders through the the interface we've created and actually see them on your Stripe dashboard.

You can learn more about Stripe at: **https://stripe.com/docs**.

## Generating PDF Documents

The last thing left to do is generate and email the invoice. We'll begin with the PDF generation, using DOMPDF and ordinary views:

```
public function getTotalAttribute()
{
  return $this->quantity * $this->price;
}
```

This was extracted from **app/models/OrderItem.php**.

```
public function getTotalAttribute()
{
  $total = 0;

  foreach ($this->orderItems as $orderItem)
  {
   $total += $orderItem->price * $orderItem->quantity;
  }

  return $total;
}
```

This was extracted from **app/models/Order.php**.

These two additional model methods allow us to get the totals of orders and order items quickly. You can learn more about Eloquent attribute getters at: **http://laravel.com/docs/eloquent#accessors-and-mutators**.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Laravel 4 E-Commerce</title>
    <style type="text/css">

      body {
        padding     : 25px 0;
        font-family : Helvetica;
      }

      td {
        padding : 0 10px 0 0;
      }

      * {
        float : none;
      }

    </style>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col-md-8">

        </div>
        <div class="col-md-4 well">
          <table>
            <tr>
              <td class="pull-right">
                <strong>Account</strong>
              </td>
              <td>
                {{ $order->account->email }}
              </td>
            </tr>
            <tr>
              <td class="pull-right">
                <strong>Date</strong>
              </td>
              <td>
                {{ $order->created_at->format("F jS, Y");  }}
              </td>
            </tr>
          </table>
        </div>
      </div>
      <div class="row">
        <div class="col-md-12">
          <h2>Invoice {{ $order->id }}</h2>
        </div>
      </div>
      <div class="row">
        <div class="col-md-12">
          <table class="table table-striped">
          <thead>
            <tr>
              <th>Product</th>
              <th>Quantity</th>
              <th>Amount</th>
            </tr>
          </thead>
          <tbody>
            @foreach ($order->orderItems as $orderItem)
              <tr>
                <td>
                  {{ $orderItem->product->name }}
                </td>
                <td>
                  {{ $orderItem->quantity }}
                </td>
                <td>
                  $ {{ number_format($orderItem->total, 2) }}
                </td>
              </tr>
            @endforeach
            <tr>
              <td>&nbsp;</td>
              <td>
                <strong>Total</strong>
              </td>
              <td>
                <strong>$ {{ number_format($order->total, 2) }}</strong>
              </td>
            </tr>
          </tbody>
        </table>
        </div>
      </div>
    </div>
  </body>
</html>
```

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

This view just displays information about the order, including items, totals and a grand total. I've avoided using Bootstrap since it seems to kill DOMDPF outright. The magic, however, is in how the PDF document is generated:

```php
<?php

namespace Formativ\Billing;

class PDFDocument
implements DocumentInterface
{
  public function create($order)
  {
    $view = View::make("email/invoice", [
      "order" => $order
    ]);

    define("DOMPDF_ENABLE_AUTOLOAD", false);

    require_once base_path() . "/vendor/dompdf/dompdf/dompdf_config.inc.php";

    $dompdf = new DOMPDF();
    $dompdf->load_html($view);
    $dompdf->set_paper("a4", "portrait");

    $dompdf->render();
    $results = $dompdf->output();

    $temp = storage_path() . "/order-" . $order->id . ".pdf";
    file_put_contents($temp, $results);

    return $temp;
  }
}
```

This file should be saved as **workbench/formativ/billing/src/Formativ/Billing/PDFDocument.php**.

We generate a PDF invoice by rendering the invoice view; setting the page size (and orientation) and rendering the document. We're also saving the PDF document to the app/storage/cache directory.

You can learn more about DOMPDF at: **https://github.com/dompdf/dompdf**.

Last thing to hook up is the MessengerInterface implementation:

```
Here's your invoice!
```

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

```php
<?php

namespace Formativ\Billing;

use Mail;

class EmailMessenger
implements MessengerInterface
{
  public function send(
    $order,
    $document
  )
  {
    Mail::send("email/wrapper", [], function($message) use ($order, $document)
    {
      $message->subject("Your invoice!");
      $message->from("info@example.com", "Laravel 4 E-Commerce");
      $message->to($order->account->email);

      $message->attach($document, [
        "as"   => "Invoice " . $order->id,
        "mime" => "application/pdf"
      ]);
    });
  }
}
```

This file should be saved as **workbench/formativ/billing/src/Formativ/Billing/EmailMessenger.php**.

The EmailMessenger class sends a simple email to the account, attaching the PDF invoice along the way.

You can learn more about sending email at: **http://laravel.com/docs/mail**.

# Embedded Systems

What do you think of when you hear the name Laravel? Laravel 4 powers an increasing number of websites, but that's not the only place it can be used. In this tutorial we're going to use it to control embedded systems.

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

## Gathering Parts

This tutorial demonstrates the use of an Arduino and a web cam. There are other smaller parts, but these are the main ones. The Arduino needs firmata installed (explained later) and the webcam needs to be compatible with Linux and Motion (installed later).

We will go over using things like LEDs and resisters, but none of those are particularly important. We will also go over using servos and these are important. You can find the bracket I use at: **https://www.sparkfun.com/products/10335** and the servos I use at: **https://www.sparkfun.com/products/9065**. You can get these parts, and an Arduino Uno, for less than $55.

## Installing Dependencies

We're developing a Laravel 4 application which has lots of server-side aspects; but there's also an interactive interface. There be scripts!

For this; we're using Bootstrap and jQuery. Download Bootstrap at: **http://getbootstrap.com/** and unpack it into your public folder. Where you put the individual files makes little difference, but I have put the scripts in **public/js**, the stylesheets in **public/css** and the fonts in **public/fonts**. Where you see those paths in my source code; you should substitute them with your own.

Next up, download jQuery at: **http://jquery.com/download/** and unpack it into your public folder.

For the server-side portion of dependencies, we need to download a library called Ratchet. I'll explain it shortly, but in the meantime we need to add it to our **composer.json** file:

```
"require" : {
    "laravel/framework" : "4.0.*",
    "cboden/Ratchet"    : "0.3.*"
},
```

This was extracted from **composer.json**.

Follow that up with:

```
composer update
```

Ratchet isn't built specifically for Laravel 4, so there are no service providers for us to add.

We'll now have access to the Ratchet library for client-server communication, Bootstrap for styling the interface and jQuery for connecting these two things together.

## Note About Errata

I love all things Arduino and Raspberry Pi. When I'm not programming; I'm trying to build things (in the real world) which can interact with my programming. Don't let that fool you though: I am not an expert in electronics. There are bound to be better ways to do all of this stuff.

If you can't get a specific program installed because apt or permissions or network settings or aliens; I cannot help you. I have neither the time nor the experience to guide you through the many potential problems. Hit me up on Twitter and, if I have not yet already done so, I will connect you to a forum where you can discuss this with other (smarter) people.

## Creating An Interface

The first step to creating an interface is returning static HTML for a web request. We handle this by creating a route, a controller and a view:

```php
<?php

Route::get("/", [
  "as"   => "index/index",
  "uses" => "IndexController@indexAction"
]);
```

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

```php
<?php

class IndexController
extends BaseController
{
  public function indexAction()
  {
    return View::make("index/index");
  }
}
```

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

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Laravel 4 Embedded Systems</title>
    <link rel="stylesheet" href="/css/bootstrap.3.0.0.css" />
    <link rel="stylesheet" href="/css/bootstrap.theme.3.0.0.css" />
    <link rel="stylesheet" href="/css/shared.css" />
  </head>
  <body>
    <div class="container">
```

```
      <div class="row">
        <div class="col-md-12">
          <h1>Laravel 4 Embedded Systems</h1>
          <input type="number" max="255" min="0" step="5" name="led" />
          <input type="number" max="255" min="0" readonly name="sensor" />
          <input type="number" max="180" min="0" step="10" name="x" />
          <input type="number" max="180" min="0" step="10" name="y" />
        </div>
      </div>
    </div>
    <script src="/js/jquery.1.9.1.js"></script>
    <script src="/js/shared.js"></script>
  </body>
</html>
```

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

If you've created all of these files (and configured your web server to serve the Laravel application) then you should see a set of controls in your browser, when you visit /. Before we move on; let's just quickly set up a socket server for this interface to talk to the PHP server with….

## Creating A Socket Server

We've covered this topic before; so I won't go into too much detail. We're going to set up a Ratchet socket server, to pass messages back and forth between the interface and the PHP server.

```
try {
  if (!WebSocket) {
      console.log("no websocket support");
  } else {

    var socket = new WebSocket("ws://127.0.0.1:8081/");

    socket.addEventListener("open", function (e) {
      console.log("open: ", e);
    });

    socket.addEventListener("error", function (e) {
      // console.log("error: ", e);
    });

    socket.addEventListener("message", function (e) {
      var data = JSON.parse(e.data);
      console.log("message: ", data);
    });

    // console.log("socket:", socket);

    window.socket = socket;

  }
} catch (e) {
  // console.log("exception: " + e);
}
```

This file should be saved as **public/js/shared.js**.

```
<?php

namespace Formativ\Embedded;

use Evenement\EventEmitter;
use Illuminate\Support\ServiceProvider;

class EmbeddedServiceProvider
extends ServiceProvider
{
  protected $defer = true;

  public function register()
  {
    $this->app->bind("formativ.embedded.emitter", function()
    {
      return new EventEmitter();
    });

    $this->app->bind("formativ.embedded.command.serve", function()
    {
      return new Command\Serve(
        $this->app->make("formativ.embedded.socket")
      );
    });

    $this->app->bind("formativ.embedded.socket", function()
    {
      return new Socket(
        $this->app->make("formativ.embedded.emitter")
      );
    });

    $this->commands(
      "formativ.embedded.command.serve"
    );
  }

  public function provides()
  {
    return [
      "formativ.embedded.emitter",
      "formativ.embedded.command.serve",
      "formativ.embedded.socket"
    ];
  }
}
```

This file should be saved as **workbench/formativ/embedded/src/Formativ/Embedded/EmbeddedServiceController.php**.

```php
<?php

namespace Formativ\Embedded;

use Evenement\EventEmitterInterface;
use Ratchet\MessageComponentInterface;

interface SocketInterface
extends MessageComponentInterface
{
    public function getEmitter();
    public function setEmitter(EventEmitterInterface $emitter);
}
```

This file should be saved as **workbench/formativ/embedded/src/Formativ/Embedded/Socket-Interface.php**.

```php
<?php

namespace Formativ\Embedded;

use Evenement\EventEmitterInterface as Emitter;
use Exception;
use Ratchet\ConnectionInterface as Connection;

class Socket
implements SocketInterface
{
  protected $emitter;

  protected $connection;

  public function getEmitter()
  {
    return $this->emitter;
  }

  public function setEmitter(Emitter $emitter)
  {
    $this->emitter = $emitter;
  }

  public function __construct(Emitter $emitter)
  {
    $this->emitter = $emitter;
  }

  public function onOpen(Connection $connection)
  {
    $this->connection = $connection;
    $this->emitter->emit("open");
  }

  public function onMessage(Connection $connection, $message)
  {
    $this->emitter->emit("message", [$message]);
  }

  public function onClose(Connection $connection)
  {
    $this->connection = null;
  }

  public function onError(Connection $connection, Exception $e)
  {
    $this->emitter->emit("error", [$e]);
  }

  public function send($message)
  {
    if ($this->connection)
    {
      $this->connection->send($message);
    }
  }
}
```

This file should be saved as **workbench/formativ/embedded/src/Formativ/Embedded/-Socket.php**.

The **Socket** class only has room for a single connection. Feel free to change this so that it can handle any number of controlling connections.

```php
<?php

namespace Formativ\Embedded\Command;

use Formativ\Embedded\SocketInterface;
use Illuminate\Console\Command;
use Ratchet\ConnectionInterface;
use Ratchet\Http\HttpServer;
use Ratchet\Server\IoServer;
use Ratchet\WebSocket\WsServer;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Input\InputArgument;

class Serve
extends Command
{
  protected $name = "embedded:serve";

  protected $description = "Creates a firmata socket server.";

  public function __construct(SocketInterface $socket)
  {
    parent::__construct();

    $this->socket = $socket;

    $socket->getEmitter()->on("message", function($message) {
      $this->info("Message: " . $message . ".");
    });

    $socket->getEmitter()->on("error", function($e) {
      $this->line("Exception: " . $e->getMessage() . ".");
    });
  }

  public function fire()
  {
    $port = (integer) $this->option("port");

    $server = IoServer::factory(
      new HttpServer(
        new WsServer(
          $this->socket
        )
      ),
      $port
    );

    $this->info("Listening on port " . $port . ".");
    $server->run();
  }

  protected function getOptions()
  {
    return [
      [
        "port",
        null,
        InputOption::VALUE_REQUIRED,
        "Port to listen on.",
        8081
      ]
    ];
  }
}
```

This file should be saved as **workbench/formativ/embedded/src/Formativ/Embedded/Command/Serve.php**.

These four files create a means with which we can start a socket server, listen for messages and respond to them. The shared.js file connects to this server and allows us to send and receive these messages. It's simpler than last time; but sufficient for our needs!

## Connection To Arduino

Real-time communication is typically done through a software layer called Firmata. There are Firmata libraries for many programming languages, though PHP is not one of them. While there is one library listed on the official Firmata website, I could not get it to work.

So we're going to create our own simple Firmata-like protocol. I managed to put one together that handles analog reads, analog writes and servo control. It does this by parsing strings received over serial connection. This is what it looks like:

```
#include <Servo.h>

String buffer = "";
String parts[3];
Servo servos[13];
int index = 0;

void setup()
{
  Serial.begin(9600);
  buffer.reserve(200);
}

void loop()
{

}

void handle()
{
  int pin = parts[1].toInt();
  int value = parts[2].toInt();

  if (parts[0] == "pinMode")
  {
    if (parts[2] == "output")
    {
      pinMode(pin, OUTPUT);
    }

    if (parts[2] == "servo")
    {
      servos[pin].attach(pin);
    }
  }

  if (parts[0] == "digitalWrite")
  {
    if (parts[2] == "high")
    {
      digitalWrite(pin, HIGH);
    }
    else
    {
      digitalWrite(pin, LOW);
    }
  }

  if (parts[0] == "analogWrite")
  {
    analogWrite(pin, value);
  }

  if (parts[0] == "servoWrite")
  {
    servos[pin].write(value);
  }

  if (parts[0] == "analogRead")
  {
    value = analogRead(pin);
  }

  Serial.print(parts[0] + "," + parts[1] + "," + value + ".\n");
}

void serialEvent()
{
  while (Serial.available())
  {
    char in = (char) Serial.read();

    if (in == '.' || in == ',')
    {
      parts[index] = String(buffer);
      buffer = "";

      index++;

      if (index > 2)
      {
        index = 0;
      }
    }
    else
    {
      buffer += in;
    }

    if (in == '.')
    {
      index = 0;
      buffer = "";
      handle();
    }
  }
}
```

This code needs to be uploaded to the Arduino you're working with for any communication between our server and the Arduino to take place.

The Arduino sketch is only one side of the puzzle. We also need to modify our socket server to communicate with the Arduino:

```php
protected $device;

public function __construct(SocketInterface $socket)
{
  parent::__construct();

  $this->socket = $socket;

  $socket->getEmitter()->on("message", function($message)
  {
    fwrite($this->device, $message);

    $data = trim(stream_get_contents($this->device));
    $this->info($data);

    $this->socket->send($data);
  });

  $socket->getEmitter()->on("error", function($e)
  {
    $this->line("exception: " . $e->getMessage() . ".");
  });
}

public function fire()
{
  $this->device = fopen($this->argument("device"), "r+");
  stream_set_blocking($this->device, 0);

  $port = (integer) $this->option("port");

  $server = IoServer::factory(
    new HttpServer(
      new WsServer(
        $this->socket
      )
    ),
    $port
  );

  $this->info("Listening on port " . $port . ".");
  $server->run();
}

protected function getArguments()
{
  return [
    [
      "device",
      InputArgument::REQUIRED,
      "Device to use."
    ]
  ];
}

public function __destruct()
{
  if (is_resource($this->device)) {
    fclose($this->device);
  }
}
```

This file should be saved as **workbench/formativ/embedded/src/Formativ/Embedded/Command/Serve.php**.

We're using PHP file read/write methods to communicate with the Arduino. In our **fire()** method, we open a connection to the Arduino and store it in **$this->device**. We also set the stream to non-blocking. This is so that read requests to the Arduino don't wait until data is returned before allowing the rest of the PHP processing to take place.

In the **__construct()** method we also modify the message event listener to write the incoming message (from the browser) to the Arduino. We read any info the Arduino has and send it back to the socket. This effectively creates a bridge between the Arduino and the socket.

We've also added two additional methods. The **getArguments()** method stipulates a required device argument. We want the serve command to be able to use any device name, so this argument makes sense. We've also added a **__destruct()** method to close the connection to the Arduino.

Finally, we need to add some JavaScript to read and write from the Arduino:

```
try {
  if (!WebSocket) {
      console.log("no websocket support");
  } else {

    var socket = new WebSocket("ws://127.0.0.1:8081/");
    var sensor = $("[name='sensor']");
    var led = $("[name='led']");
    var x = $("[name='x']");
    var y = $("[name='y']");

    socket.addEventListener("open", function (e) {
      socket.send("pinMode,6,output.");
      socket.send("pinMode,3,servo.");
      socket.send("pinMode,5,servo.");
    });

    socket.addEventListener("error", function (e) {
      // console.log("error: ", e);
    });

    socket.addEventListener("message", function (e) {

      var data  = e.data;
      var parts = data.split(",");

      if (parts[0] == "analogRead") {
        sensor.val(parseInt(parts[2], 10));
      }

    });

    window.socket = socket;

    led.on("change", function() {
      socket.send("analogWrite,6," + led.val() + ".");
    });

    x.on("change", function() {
      socket.send("servoWrite,5," + x.val() + ".");
    });

    y.on("change", function() {
      socket.send("analogWrite,3," + y.val() + ".");
    });

    setInterval(function() {
      socket.send("analogRead,0,void.");
    }, 1000);

  }
} catch (e) {
  // console.log("exception: " + e);
}
```

This was extracted from **public/js/shared.js**.

This script does a number of cool things. When it loads, we set the pinMode of the LED pin (6) and the two servo pins (3, 5) to the correct modes. We also attach a message event listener to receive and parse messages from the Arduino.

We then attach some event listeners for the input elements in the interface. These will respond to changes in value by sending signals to the Arduino to adjust its various pin values.

Finally, we set a timer to ping the Arduino with analogRead requests on the sensor pin. Thanks to the message event listener, the return values will be received and reflected in the sensor input field.

## Spinning Up

To connect to the Arduino, you should run the following command:

```
php artisan embedded:serve /dev/cu.usbmodem1411
```

On my system, the Arduino is reflected as **/dev/cu.usbmodem1411** by this may differ on yours. For instance, my Ubuntu machine shows it as **/dev/ttyACM0**. The easiest way to find out what it is, is to unplug it, then run the following command:

```
ls /dev/tty*
```

Look over the list to familiarise yourself with the devices you see there. The plug the Arduino in, wait a couple seconds and run that command again. This time you should see a new addition to the list of devices. This is most likely your Arduino.

> You may be wondering why my device starts with **cu** and not **tty**. I first tried the **tty** version of the Arduino and it would cause the server to hang. I experimented and found that the **cu** version let me connect and communicate with the Arduino. Go figure…

I usually like to include Vagrant configurations for running these projects. This time I found it confusing to try and communicate with an Arduino through the host machine. So instead I decided to try the server.php script bundled with Laravel applications. Turns out you can run the server with the following command:

```
php -S 127.0.0.1:8080 -t ./public  server.php
```

Now you can go to **http://127.0.0.1:8080** and see the interface we created earlier. You need to start the serve command successfully before this interface will work. If everything is running correctly, you should immediately start to see the sensor values being updated in the interface.

You can now adjust the LED input to control the brightness of the LED, and the x and y inputs to control the rotation of the servos.

## Adding A Webcam

Streaming video from a web cam can be tricky, so we're going to take time-lapse photos instead…

### Installing ImageSnap On OSX

The utility we'll be using on OSX is called ImageSnap. We can install it with homebrew:

brew install imagesnap

Once that's installed; we can periodically take photos with it by using a command resembling:

```
imagesnap -d "Microsoft LifeCam" -w 3
```

The web cam I am using is **Microsoft LifeCam** but you can find the appropriate one for you with the command:

```
imagesnap -l
```

I also had to add some warm-up time, with the -**w** switch.

### Installing Streamer On Ubuntu/Debian

The utility we'll be using on Ubuntu/Debian is called Streamer. We can install it with the command:

```
apt-get install -y streamer
```

You can take a photo with the web cam, using the following command:

```
streamer -o snapshot.jpeg
```

## Displaying Photos In The Interface

Depending on whether you are using OSX or Ubuntu/Debian; you will need to set up a shell script to periodically take photos. Save them to a web-accessible location and display them with the following changes:

```
<img name="photo" />
```

This was extracted from **app/views/index/index.blade.php**.

```
setInterval(function() {
  $("[name='photo']").attr("src", "[path to photo]");
}, 1000);
```

This was extracted from **public/js/shared.js**.

# File-Based CMS

October is a Laravel-based, pre-built CMS which was recently announced. I have yet to see the code powering what looks like a beautiful and efficient CMS system. So I thought I would try to implement some of the concepts presented in the introductory video as they illustrate valuable tips for working with Laravel.

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

## Installing Dependencies

We're developing a Laravel 4 application which has lots of server-side aspects; but there's also an interactive interface. There be scripts!

For this; we're using Bootstrap and jQuery. Download Bootstrap at: **http://getbootstrap.com/** and unpack it into your public folder. Where you put the individual files makes little difference, but I have put the scripts in **public/js**, the stylesheets in **public/css** and the fonts in **public/fonts**. Where you see those paths in my source code; you should substitute them with your own.

Next up, download jQuery at: **http://jquery.com/download** and unpack it into your public folder.

On the server-side, we're going to be using Flysystem for reading and writing files. Add it to the Composer dependencies:

```
"require" : {
  "laravel/framework" : "4.1.*",
  "league/flysystem"  : "0.2.*"
},
```

> This was extracted from **composer.json**.

Follow that up with:

```
composer update
```

## Rendering Templates

We’ve often used **View::make()** to render views. It’s great for when we have pre-defined view files and we want Laravel to manage how they are rendered and stored. In this tutorial, we’re going to be rendering templates from strings. We’ll need to encapsulate some of how Laravel rendered templates, but it’ll also give us a good base for extending upon the Blade template syntax.

Let’s get started by creating a service provider:

```
php artisan workbench formativ/cms
```

This will generate the usual scaffolding for a new package. We need to add it in a few places, to be able to use it in our application:

```
"providers" => [
  "Formativ\Cms\CmsServiceProvider",
  // …remaining service providers
],
```

This was extracted from **app/config/app.php**.

```
"autoload" : {
  "classmap" : [
    // …
  ],
  "psr-0" : {
    "Formativ\\Cms" : "workbench/formativ/cms/src/"
  }
}
```

This was extracted from **composer.json**.

Then we need to rebuild the composer autoloader:

```
composer dump-autoload
```

All this gets us to a place where we can start to add classes for encapsulating and extending Blade rendering. Let's create some wrapper classes, and register them in the service provider:

```
<?php

namespace Formativ\Cms;

interface CompilerInterface
{
  public function compileString($template);
}
```

This file should be saved as **workbench/formativ/cms/src/Formativ/Cms/CompilerInterface.php**.

```
<?php

namespace Formativ\Cms\Compiler;

use Formativ\Cms\CompilerInterface;
use Illuminate\View\Compilers\BladeCompiler;

class Blade
extends BladeCompiler
implements CompilerInterface
{

}
```

This file should be saved as **workbench/formativ/cms/src/Formativ/Cms/Compiler/Blade.php**.

```php
<?php

namespace Formativ\Cms;

interface EngineInterface
{
  public function render($template, $data);
}
```

This file should be saved as **workbench/formativ/cms/src/Formativ/Cms/EngineInterface.php**.

```php
<?php

namespace Formativ\Cms\Engine;

use Formativ\Cms\CompilerInterface;
use Formativ\Cms\EngineInterface;

class Blade
implements EngineInterface
{
  protected $compiler;

  public function __construct(CompilerInterface $compiler)
  {
    $this->compiler = $compiler;
  }

  public function render($template, $data)
  {
    $compiled = $this->compiler->compileString($template);

    ob_start();
    extract($data, EXTR_SKIP);

    try
    {
      eval("?>" . $compiled);
    }
    catch (Exception $e)
    {
      ob_end_clean();
      throw $e;
    }

    $result = ob_get_contents();
    ob_end_clean();

    return $result;
  }
}
```

This file should be saved as **workbench/formativ/cms/src/Formativ/Cms/Engine/Blade.php**.

```
<?php

namespace Formativ\Cms;

use Illuminate\Support\ServiceProvider;

class CmsServiceProvider
extends ServiceProvider
{
  protected $defer = true;

  public function register()
  {
    $this->app->bind(
      "Formativ\Cms\CompilerInterface",
      function() {
        return new Compiler\Blade(
          $this->app->make("files"),
          $this->app->make("path.storage") . "/views"
        );
      }
    );

    $this->app->bind(
      "Formativ\Cms\EngineInterface",
      "Formativ\Cms\Engine\Blade"
    );
  }

  public function provides()
  {
    return [
      "Formativ\Cms\CompilerInterface",
      "Formativ\Cms\EngineInterface"
    ];
  }
}
```

This file should be saved as **workbench/formativ/cms/src/Formativ/Cms/CmsServiceProvider.php**.

The **CompilerBlade** class encapsulates the **BladeCompiler** class, allowing us to implement the **CompilerInterface** interface. This is a way of future-proofing our package so that the code which depends on methods Blade currently implements won't fail if future versions of Blade were to remove that implementation.

We'll also be adding additional template tags in via this class, so it's not all overhead.

The **EngineBlade** class contains the method which we will use to render template strings. It implements the **EngineInterface** interface for that same future-proofing.

We register all of these in the **CmsServiceProvider**. We can now inject these dependencies in our controller:

```php
<?php

use Formativ\Cms\EngineInterface;

class IndexController
extends BaseController
{
  protected $engine;

  public function __construct(EngineInterface $engine)
  {
    $this->engine = $engine;
  }

  public function indexAction()
  {
    // ...use $this->engine->render() here
  }
}
```

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

As we bound **FormativCmsEngineInterface** in our service provider, we can now specify it in our controller constructor and Laravel will automatically inject it for us.

## Gathering Metadata

One of the interesting things October does is store all of the page and layout meta data at the top of the template file. This allows changes to metadata (which would normally take place elsewhere) to be version-controlled. Having gained the ability to render template strings, we're now in a position to be able to isolate this kind of metadata and render the rest of the file as a template.

Consider the following example:

```
protected function minify($html)
{
  $search = array(
      "/\>[^\S ]+/s",
      "/[^\S ]+\</s",
      "/(\s)+/s"
  );

  $replace = array(
      ">",
      "<",
      "\\1"
  );

  $html = preg_replace($search, $replace, $html);

  return $html;
}

public function indexAction()
{
  $template = $this->minify("
    <!doctype html>
    <html lang='en'>
      <head>
        <title>
          Laravel 4 File-Based CMS
        </title>
      </head>
      <body>
        Hello world
      </body>
    </html>
  ");

  return $this->engine->render($template, []);
}
```

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

Here we’re rendering a page template (with the help of a minify method). It’s just like what we did before. Let’s add some metadata, and pull it out of the template before rendering:

```php
protected function extractMeta($html)
{
  $parts = explode("==", $html, 2);

  $meta = "";
  $html = $parts[0];

  if (count($parts) > 1)
  {
    $meta = $parts[0];
    $html = $parts[1];
  }

  return [
    "meta" => $meta,
    "html" => $html
  ];
}

protected function parseMeta($meta)
{
  $meta  = trim($meta);
  $lines = explode("\n", $meta);
  $data  = [];

  foreach ($lines as $line)
  {
    $parts = explode("=", $line);
    $data[trim($parts[0])] = trim($parts[1]);
  }

  return $data;
}

public function indexAction()
{
  $parts = $this->extractMeta("
    title   = Laravel 4 File-Based CMS
    message = Hello world
    ==
    <!doctype html>
    <html lang='en'>
      <head>
        <title>
          {{ \$title }}
        </title>
      </head>
      <body>
        {{ \$message }}
      </body>
    </html>
  ");

  $data     = $this->parseMeta($parts["meta"]);
  $template = $this->minify($parts["html"]);

  return $this->engine->render($template, $data);
}
```

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

This time round, we're using an **extractMeta()** method to pull the meta data string out of the template string, and a **parseMeta()** method to split the lines of metadata into key/value pairs.

The result is a functional means of storing and parsing meta data, and rendering the remaining template from and to a string.

The minify method is largely unmodified from the original, which I found at: **http://stackoverflow.com/questions/6225351/how-to-minify-php-page-html-output**.

## Creating Layouts

We need to create some sort of admin interface, with which to create and/or modify pages and layouts. Let's skip the authentication system (as we've done that before and it will distract from the focus of this tutorial).

I’ve chosen for us to use Flysystem when working with the filesystem. It would be a good idea to future-proof this dependency by wrapping it in a subclass which implements an interface we control.

```php
<?php

namespace Formativ\Cms;

interface FilesystemInterface
{
  public function has($file);
  public function listContents($folder, $detail = false);
  public function write($file, $contents);
  public function read($file);
  public function put($file, $content);
  public function delete($file);
}
```

This file should be saved as **workbench/formativ/cms/src/Formativ/Cms/FilesystemInterface.php**.

```php
<?php

namespace Formativ\Cms;

use League\Flysystem\Filesystem as Base;

class Filesystem
extends Base
implements FilesystemInterface
{

}
```

This file should be saved as **workbench/formativ/cms/src/Formativ/Cms/Filesystem.php**.

```php
public function register()
{
  $this->app->bind(
    "Formativ\Cms\CompilerInterface",
    function() {
      return new Compiler\Blade(
        $this->app->make("files"),
        $this->app->make("path.storage") . "/views"
      );
    }
  );

  $this->app->bind(
    "Formativ\Cms\EngineInterface",
    "Formativ\Cms\Engine\Blade"
  );

  $this->app->bind(
    "Formativ\Cms\FilesystemInterface",
    function() {
      return new Filesystem(
        new Local(
          $this->app->make("path.base") . "/app/views"
        )
      );
    }
  );
}
```

This was extracted from **workbench/formativ/cms/src/Formativ/Cms/CmsService-Provider.php**.

We're not really adding any extra functionality to that which Flysystem provides. The sole purpose of us wrapping the **Local** Flysystem adapter is to make provision for swapping it with another filesystem class/library.

We should also move the metadata-related functionality into a better location.

```php
<?php

namespace Formativ\Cms;

interface EngineInterface
{
  public function render($template, $data);
  public function extractMeta($template);
  public function parseMeta($meta);
  public function minify($template);
}
```

This file should be saved as **workbench/formativ/cms/src/Formativ/Cms/EngineInterface.php**.

```php
<?php

namespace Formativ\Cms\Engine;

use Formativ\Cms\CompilerInterface;
use Formativ\Cms\EngineInterface;

class Blade
implements EngineInterface
{
  protected $compiler;

  public function __construct(CompilerInterface $compiler)
  {
    $this->compiler = $compiler;
  }

  public function render($template, $data)
  {
    $extracted = $this->extractMeta($template)["template"];
    $compiled  = $this->compiler->compileString($extracted);

    ob_start();
    extract($data, EXTR_SKIP);

    try
    {
      eval("?>" . $compiled);
    }
    catch (Exception $e)
    {
      ob_end_clean();
      throw $e;
    }

    $result = ob_get_contents();
    ob_end_clean();

    return $result;
  }

  public function minify($template)
  {
    $search = array(
        "/\>[^\S ]+/s",
        "/[^\S ]+\</s",
        "/(\s)+/s"
    );

    $replace = array(
        ">",
        "<",
        "\\1"
    );

    $template = preg_replace($search, $replace, $template);

    return $template;
  }

  public function extractMeta($template)
  {
    $parts = explode("==", $template, 2);

    $meta     = "";
    $template = $parts[0];

    if (count($parts) > 1)
    {
      $meta     = $parts[0];
      $template = $parts[1];
    }

    return [
      "meta"     => $meta,
      "template" => $template
    ];
  }

  public function parseMeta($meta)
  {
    $meta  = trim($meta);
    $lines = explode("\n", $meta);
    $data  = [];

    foreach ($lines as $line)
    {
      $parts = explode("=", $line);
      $data[trim($parts[0])] = trim($parts[1]);
    }

    return $data;
  }
}
```

This file should be saved as **workbench/formativ/cms/src/Formativ/Cms/Engine/Blade.php**.

The only difference can be found in the argument names (to bring them more in line with the rest of the class) and integrating the meta methods into the **render()** method.

Next up is layout controller class:

```php
<?php

use Formativ\Cms\EngineInterface;
use Formativ\Cms\FilesystemInterface;

class LayoutController
extends BaseController
{
  protected $engine;
  protected $filesystem;

  public function __construct(
    EngineInterface $engine,
    FilesystemInterface $filesystem
  )
  {
    $this->engine     = $engine;
    $this->filesystem = $filesystem;

    Validator::extend(
      "add",
      function($attribute, $value, $params) {
        return !$this->filesystem->has("layouts/" . $value);
      }
    );

    Validator::extend(
      "edit",
      function($attribute, $value, $params) {
        $new  = !$this->filesystem->has("layouts/" . $value);
        $same = $this->filesystem->has("layouts/" . $params[0]);

        return $new or $same;
      }
    );
  }

  public function indexAction()
  {
    $layouts = $this->filesystem->listContents("layouts");
    $edit    = URL::route("admin/layout/edit") . "?layout=";
    $delete  = URL::route("admin/layout/delete") . "?layout=";

    return View::make("admin/layout/index", compact(
      "layouts",
      "edit",
      "delete"
    ));
  }
}
```

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

This is the first time we're using Dependency Injection in our controller. Laravel is injecting our engine interface (which is the Blade wrapper) and our filesystem interface (which is the Flysystem wrapper). As usual, we assign the injected dependencies to protected properties. We also define two custom validation rules, which we'll use when adding and editing the layout files.

We've also defined an **indexAction()** method which will be used to display a list of layout files which can then be edited or deleted. For the interface to be complete, we are going to need the following files:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Laravel 4 File-Based CMS</title>
    <link
      rel="stylesheet"
      href="{{ asset("css/bootstrap.min.css"); }}"
    />
    <link
      rel="stylesheet"
      href="{{ asset("css/shared.css"); }}"
    />
  </head>
  <body>
    @include("admin/include/navigation")
    <div class="container">
      <div class="row">
        <div class="column md-12">
```

```
          @yield("content")
        </div>
      </div>
    </div>
    <script src="{{ asset("js/jquery.min.js"); }}"></script>
    <script src="{{ asset("js/bootstrap.min.js"); }}"></script>
  </body>
</html>
```

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

```
<nav
  class="navbar navbar-inverse navbar-fixed-top"
  role="navigation"
>
  <div class="container-fluid">
    <div class="navbar-header">
    <button type="button"
      class="navbar-toggle"
      data-toggle="collapse"
      data-target="#navbar-collapse"
    >
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
  </div>
  <div
    class="collapse navbar-collapse"
    id="navbar-collapse"
  >
    <ul class="nav navbar-nav">
      <li class="@yield("navigation/layout/class")">
        <a href="{{ URL::route("admin/layout/index") }}">
          Layouts
        </a>
      </li>
    </div>
  </div>
</nav>
```

This file should be saved as **app/views/admin/include/navigation.blade.php**.

```
<ol class="breadcrumb">
  <li>
    <a href="{{ URL::route("admin/layout/index") }}">
      List Layouts
    </a>
  </li>
  <li>
    <a href="{{ URL::route("admin/layout/add") }}">
      Add New Layout
    </a>
  </li>
</ol>
```

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

```
@extends("admin/layout")
@section("navigation/layout/class")
  active
@stop
@section("content")
  @include("admin/include/layout/navigation")
  @if (count($layouts))
    <table class="table table-striped">
      <thead>
        <tr>
          <th class="wide">
            File
          </th>
          <th class="narrow">
            Actions
          </th>
        </tr>
      </thead>
      <tbody>
        @foreach ($layouts as $layout)
          @if ($layout["type"] == "file")
            <tr>
              <td class="wide">
                <a href="{{ $edit . $layout["basename"] }}">
                  {{ $layout["basename"] }}
                </a>
              </td>
              <td class="narrow actions">
                <a href="{{ $edit . $layout["basename"] }}">
                  <i class="glyphicon glyphicon-pencil"></i>
                </a>
                <a href="{{ $delete . $layout["basename"] }}">
                  <i class="glyphicon glyphicon-trash"></i>
                </a>
              </td>
            </tr>
          @endif
        @endforeach
      </tbody>
    </table>
    @else
      No layouts yet.
      <a href="{{ URL::route("admin/layout/add") }}">
        create one now!
      </a>
    @endif
@stop
```

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

```php
Route::any("admin/layout/index", [
  "as"   => "admin/layout/index",
  "uses" => "LayoutController@indexAction"
]);
```

This was extracted from **app/routes.php**.

These are quite a few files, so let's go over them individually:

1. The first file is the main admin layout template. Every page in the admin area should be rendered within this layout template. As you can see, we've linked the jQuery and Bootstrap assets to provide some styling and interactive functionality to the admin area.
2. The second file is the main admin navigation template. This will also be present on every page in the admin area, though it's better to include it in the main layout template than to clutter the main layout template with secondary markup.
3. The third file is a sub-navigation template, only present in the pages concerning layouts.
4. The fourth file is the layout index (or listing) page. It includes the sub-navigation and renders a table row for each layout file it finds. If none can be found, it will present a cheeky message for the user to add one.
5. Finally we add the index route to the routes.php file. At this point, the page should be visible via the browser. As there aren't any layout files yet, you should see the cheeky message instead.

Let's move onto the layout add page:

```php
public function addAction()
{
  if (Input::has("save"))
  {
    $validator = Validator::make(Input::all(), [
      "name" => "required|add",
      "code" => "required"
    ]);

    if ($validator->fails())
    {
      return Redirect::route("admin/layout/add")
        ->withInput()
        ->withErrors($validator);
    }

    $meta = "
      title       = " . Input::get("title") . "
      description = " . Input::get("description") . "
      ==
    ";

    $name = "layouts/" . Input::get("name") . ".blade.php";

    $this->filesystem->write($name, $meta . Input::get("code"));

    return Redirect::route("admin/layout/index");
  }

  return View::make("admin/layout/add");
}
```

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

```
@extends("admin/layout")
@section("navigation/layout/class")
  active
@stop
@section("content")
  @include("admin/include/layout/navigation")
  <form role="form" method="post">
    <div class="form-group">
      <label for="name">Name</label>
      <span class="help-text text-danger">
        {{ $errors->first("name") }}
      </span>
      <input
        type="text"
        class="form-control"
        id="name"
        name="name"
        placeholder="new-layout"
        value="{{ Input::old("name") }}"
      />
    </div>
    <div class="form-group">
      <label for="title">Meta Title</label>
      <input
        type="text"
        class="form-control"
        id="title"
        name="title"
        value="{{ Input::old("title") }}"
      />
    </div>
    <div class="form-group">
      <label for="description">Meta Description</label>
      <input
        type="text"
        class="form-control"
        id="description"
        name="description"
        value="{{ Input::old("description") }}"
      />
    </div>
    <div class="form-group">
      <label for="code">Code</label>
      <span class="help-text text-danger">
        {{ $errors->first("code") }}
      </span>
      <textarea
        class="form-control"
        id="code"
        name="code"
        rows="5"
        placeholder="&lt;div&gt;Hello world&lt;/div&gt;"
      >{{ Input::old("code") }}</textarea>
    </div>
    <input
      type="submit"
      name="save"
      class="btn btn-default"
      value="Save"
    />
  </form>
@stop
```

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

```
Route::any("admin/layout/add", [
  "as"   => "admin/layout/add",
  "uses" => "LayoutController@addAction"
]);
```

This was extracted from **app/routes.php**.

The form processing, in the **addAction()** method, is wrapped in a check for the **save** parameter. This is the name of the submit button on the add form. We specify the validation rules (including one of those we defined in the constructor). If validation fails, we redirect back to the add page, bringing along the errors and old input. If not, we create a new file with the default meta title and default meta description as metadata. Finally we redirect to the index page.

The view is fairly standard (including the bootstrap tags we've used). The name and code fields have error messages and all of the fields have their values set to the old input values. We've also added a route to the add page.

Edit follows a similar pattern:

```
public function editAction()
{
  $layout           = Input::get("layout");
  $name             = str_ireplace(".blade.php", "", $layout);
  $content          = $this->filesystem->read("layouts/" . $layout);
  $extracted        = $this->engine->extractMeta($content);
  $code             = trim($extracted["template"]);
  $parsed           = $this->engine->parseMeta($extracted["meta"]);
  $title            = $parsed["title"];
  $description      = $parsed["description"];

  if (Input::has("save"))
  {
    $validator = Validator::make(Input::all(), [
      "name" => "required|edit:" . Input::get("layout"),
      "code" => "required"
    ]);

    if ($validator->fails())
    {
      return Redirect::route("admin/layout/edit")
        ->withInput()
        ->withErrors($validator);
    }

    $meta = "
      title       = " . Input::get("title") . "
      description = " . Input::get("description") . "
      ==
    ";

    $name = "layouts/" . Input::get("name") . ".blade.php";

    $this->filesystem->put($name, $meta . Input::get("code"));

    return Redirect::route("admin/layout/index");
  }

  return View::make("admin/layout/edit", compact(
    "name",
    "title",
    "description",
    "code"
  ));
}
```

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

```
@extends("admin/layout")
@section("navigation/layout/class")
  active
@stop
@section("content")
  @include("admin/include/layout/navigation")
  <form role="form" method="post">
    <div class="form-group">
      <label for="name">Name</label>
      <span class="help-text text-danger">
        {{ $errors->first("name") }}
      </span>
      <input
        type="text"
        class="form-control"
        id="name"
        name="name"
        placeholder="new-layout"
        value="{{ Input::old("name", $name) }}"
      />
    </div>
    <div class="form-group">
      <label for="title">Meta Title</label>
      <input
        type="text"
        class="form-control"
        id="title"
        name="title"
        value="{{ Input::old("title", $title) }}"
      />
    </div>
    <div class="form-group">
      <label for="description">Meta Description</label>
      <input
        type="text"
        class="form-control"
        id="description"
        name="description"
        value="{{ Input::old("description", $description) }}"
      />
    </div>
    <div class="form-group">
```

```
        <label for="code">Code</label>
        <span class="help-text text-danger">
          {{ $errors->first("code") }}
        </span>
        <textarea
          class="form-control"
          id="code"
          name="code"
          rows="5"
          placeholder="&lt;div&gt;Hello world&lt;/div&gt;"
        >{{ Input::old("code", $code) }}</textarea>
      </div>
      <input
        type="submit"
        name="save"
        class="btn btn-default"
        value="Save"
      />
    </form>
@stop
```

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

```
Route::any("admin/layout/edit", [
  "as"   => "admin/layout/edit",
  "uses" => "LayoutController@editAction"
]);
```

This was extracted from **app/routes.php**.

The **editAction()** method fetches the layout file data and extracts/parses the metadata, so that we can present it in the edit form. Other than utilising the second custom validation function, we define in the constructor, there's nothing else noteworthy in this method.

The edit form is also pretty much the same, except that we provide default values to the **Input::old()** method calls, giving the data extracted from the layout file. We also add a route to the edit page.

You may notice that the file name remains editable, on the edit page. When you change this value, and save the layout it won't change the name of the current layout file, but rather generate a new layout file. This is an interesting (and in this case useful) side-effect of using the file name as the unique identifier for layout files.

Deleting layout files is even simpler:

```php
public function deleteAction()
{
  $name = "layouts/" . Input::get("layout");
  $this->filesystem->delete($name);

  return Redirect::route("admin/layout/index");
}
```

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

```php
Route::any("admin/layout/delete", [
  "as"   => "admin/layout/delete",
  "uses" => "LayoutController@deleteAction"
]);
```

This was extracted from **app/routes.php**.

We link straight to the **deleteAction()** method in the index view. This method simply deletes the layout file and redirects back to the index page. We've added the appropriate route to make this page accessible.

We can now list the layout files, add new ones, edit existing ones and delete those layout files we no longer require. It's basic, and could definitely be polished a bit, but it's sufficient for our needs.

## Creating Pages

Pages are handled in much the same way, so we're not going to spend too much time on them. Let's begin with the controller:

```php
<?php

use Formativ\Cms\EngineInterface;
use Formativ\Cms\FilesystemInterface;

class PageController
extends BaseController
{
  protected $engine;
  protected $filesystem;

  public function __construct(
    EngineInterface $engine,
    FilesystemInterface $filesystem
  )
  {
    $this->engine     = $engine;
    $this->filesystem = $filesystem;

    Validator::extend(
      "add",
      function($attribute, $value, $params) {
        return !$this->filesystem->has("pages/" . $value);
      }
    );

    Validator::extend(
      "edit",
      function($attribute, $value, $params) {
        $new  = !$this->filesystem->has("pages/" . $value);
        $same = $this->filesystem->has("pages/" . $params[0]);

        return $new or $same;
      }
    );
  }

```

```
public function indexAction()
{
  $pages  = $this->filesystem->listContents("pages");
  $edit   = URL::route("admin/page/edit") . "?page=";
  $delete = URL::route("admin/page/delete") . "?page=";

  return View::make("admin/page/index", compact(
    "pages",
    "edit",
    "delete"
  ));
}

public function addAction()
{
  $files   = $this->filesystem->listContents("layouts");
  $layouts = [];

  foreach ($files as $file)
  {
    $name = $file["basename"];
    $layouts[$name] = $name;
  }

  if (Input::has("save"))
  {
    $validator = Validator::make(Input::all(), [
      "name"   => "required|add",
      "route"  => "required",
      "layout" => "required",
      "code"   => "required"
    ]);

    if ($validator->fails())
    {
      return Redirect::route("admin/page/add")
        ->withInput()
        ->withErrors($validator);
    }

    $meta = "
      title       = " . Input::get("title") . "
```

```
        description = " . Input::get("description") . "
        layout      = " . Input::get("layout") . "
        route       = " . Input::get("route") . "
        ==
      ";

      $name = "pages/" . Input::get("name") . ".blade.php";
      $code = $meta . Input::get("code");

      $this->filesystem->write($name, $code);

      return Redirect::route("admin/page/index");
    }

    return View::make("admin/page/add", compact(
      "layouts"
    ));
  }

  public function editAction()
  {
    $files   = $this->filesystem->listContents("layouts");
    $layouts = [];

    foreach ($files as $file)
    {
      $name = $file["basename"];
      $layouts[$name] = $name;
    }

    $page            = Input::get("page");
    $name            = str_ireplace(".blade.php", "", $page);
    $content         = $this->filesystem->read("pages/" . $page);
    $extracted       = $this->engine->extractMeta($content);
    $code            = trim($extracted["template"]);
    $parsed          = $this->engine->parseMeta($extracted["meta"]);
    $title           = $parsed["title"];
    $description     = $parsed["description"];
    $route           = $parsed["route"];
    $layout          = $parsed["layout"];

    if (Input::has("save"))
    {
      $validator = Validator::make(Input::all(), [
        "name"    => "required|edit:" . Input::get("page"),
        "route"   => "required",
        "layout"  => "required",
        "code"    => "required"
      ]);

      if ($validator->fails())
      {
        return Redirect::route("admin/page/edit")
          ->withInput()
          ->withErrors($validator);
      }

      $meta = "
        title       = " . Input::get("title") . "
        description = " . Input::get("description") . "
        layout      = " . Input::get("layout") . "
        route       = " . Input::get("route") . "
        ==
      ";

      $name = "pages/" . Input::get("name") . ".blade.php";

      $this->filesystem->put($name, $meta . Input::get("code"));

      return Redirect::route("admin/page/index");
    }

    return View::make("admin/page/edit", compact(
      "name",
      "title",
      "description",
      "layout",
      "layouts",
      "route",
      "code"
    ));
  }

  public function deleteAction()
  {
    $name = "pages/" . Input::get("page");
    $this->filesystem->delete($name);

    return Redirect::route("admin/page/index");
  }
}
```

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

The constructor method accepts the same injected dependencies as our layout controller did. We also define similar custom validation rules to check the names of files we want to save.

The **addAction()** method differs slightly in that we load the existing layout files so that we can designate the layout for each page. We also add this (and the route parameter) to the metadata saved to the page file.

The **editAction()** method loads the route and layout parameters (in addition to the other fields) and passes them to the edit page template, where they will be used to populate the new fields.

```
<nav
  class="navbar navbar-inverse navbar-fixed-top"
  role="navigation"
>
  <div class="container-fluid">
    <div class="navbar-header">
      <button type="button"
        class="navbar-toggle"
        data-toggle="collapse"
        data-target="#navbar-collapse"
      >
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
    </div>
    <div class="collapse navbar-collapse" id="navbar-collapse">
      <ul class="nav navbar-nav">
        <li class="@yield("navigation/layout/class")">
```

```
          <a href="{{ URL::route("admin/layout/index") }}">
            Layouts
          </a>
        </li>
        <li class="@yield("navigation/page/class")">
          <a href="{{ URL::route("admin/page/index") }}">
            Pages
          </a>
        </li>
      </ul>
    </div>
  </div>
</nav>
```

This file should be saved as **app/views/admin/include/navigation.blade.php**.

```
<ol class="breadcrumb">
  <li>
    <a href="{{ URL::route("admin/page/index") }}">
      List Pages
    </a>
  </li>
  <li>
    <a href="{{ URL::route("admin/page/add") }}">
      Add New Page
    </a>
  </li>
</ol>
```

This file should be saved as **app/views/admin/include/page/navigation.blade.php**.

```
@extends("admin/layout")
@section("navigation/page/class")
  active
@stop
@section("content")
  @include("admin/include/page/navigation")
  @if (count($pages))
    <table class="table table-striped">
      <thead>
        <tr>
          <th class="wide">
            File
          </th>
          <th class="narrow">
            Actions
          </th>
        </tr>
      </thead>
      <tbody>
        @foreach ($pages as $page)
          @if ($page["type"] == "file")
            <tr>
              <td class="wide">
                <a href="{{ $edit . $page["basename"] }}">
                  {{ $page["basename"] }}
                </a>
              </td>
              <td class="narrow actions">
                <a href="{{ $edit . $page["basename"] }}">
                  <i class="glyphicon glyphicon-pencil"></i>
                </a>
                <a href="{{ $delete . $page["basename"] }}">
                  <i class="glyphicon glyphicon-trash"></i>
                </a>
              </td>
            </tr>
          @endif
        @endforeach
      </tbody>
    </table>
    @else
      No pages yet.
      <a href="{{ URL::route("admin/page/add") }}">
        create one now!
      </a>
    @endif
@stop
```

This file should be saved as **app/views/admin/page/index.blade.php**.

```
@extends("admin/layout")
@section("navigation/page/class")
  active
@stop
@section("content")
  @include("admin/include/page/navigation")
  <form role="form" method="post">
    <div class="form-group">
      <label for="name">Name</label>
      <span class="help-text text-danger">
        {{ $errors->first("name") }}
      </span>
      <input
        type="text"
        class="form-control"
        id="name"
        name="name"
        placeholder="new-page"
        value="{{ Input::old("name") }}"
      />
    </div>
    <div class="form-group">
      <label for="route">Route</label>
      <span class="help-text text-danger">
        {{ $errors->first("route") }}
      </span>
      <input
        type="text"
        class="form-control"
        id="route"
        name="route"
        placeholder="/new-page"
        value="{{ Input::old("route") }}"
      />
    </div>
    <div class="form-group">
      <label for="layout">Layout</label>
      <span class="help-text text-danger">
        {{ $errors->first("layout") }}
      </span>
      {{ Form::select(
        "layout",
        $layouts,
        Input::old("layout"),
        [
          "id"    => "layout",
          "class" => "form-control"
        ]
      ) }}
    </div>
    <div class="form-group">
      <label for="title">Meta Title</label>
      <input
        type="text"
        class="form-control"
        id="title"
        name="title"
        value="{{ Input::old("title") }}"
      />
    </div>
    <div class="form-group">
      <label for="description">Meta Description</label>
      <input
        type="text"
        class="form-control"
        id="description"
        name="description"
        value="{{ Input::old("description") }}"
      />
    </div>
    <div class="form-group">
      <label for="code">Code</label>
```

```
        <span class="help-text text-danger">
          {{ $errors->first("code") }}
        </span>
        <textarea
          class="form-control"
          id="code"
          name="code"
          rows="5"
          placeholder="&lt;div&gt;Hello world&lt;/div&gt;"
        >{{ Input::old("code") }}</textarea>
      </div>
      <input
        type="submit"
        name="save"
        class="btn btn-default"
        value="Save"
      />
    </form>
@stop
```

This file should be saved as **app/views/admin/page/add.blade.php**.

```
@extends("admin/layout")
@section("navigation/page/class")
  active
@stop
@section("content")
  @include("admin/include/page/navigation")
  <form role="form" method="post">
    <div class="form-group">
      <label for="name">Name</label>
      <span class="help-text text-danger">
        {{ $errors->first("name") }}
      </span>
      <input
        type="text"
        class="form-control"
        id="name"
        name="name"
        placeholder="new-page"
        value="{{ Input::old("name", $name) }}"
      />
    </div>
    <div class="form-group">
      <label for="route">Route</label>
      <span class="help-text text-danger">
        {{ $errors->first("route") }}
      </span>
      <input
        type="text"
        class="form-control"
        id="route"
        name="route"
        placeholder="/new-page"
        value="{{ Input::old("route", $route) }}"
      />
    </div>
    <div class="form-group">
      <label for="layout">Layout</label>
      <span class="help-text text-danger">
        {{ $errors->first("layout") }}
      </span>
      {{ Form::select("layout", $layouts, Input::old("layout", $layout), [
        "id"    => "layout",
        "class" => "form-control"
      ]) }}
    </div>
    <div class="form-group">
      <label for="title">Meta Title</label>
      <input
        type="text"
        class="form-control"
        id="title"
        name="title"
        value="{{ Input::old("title", $title) }}"
      />
    </div>
    <div class="form-group">
      <label for="description">Meta Description</label>
      <input
```

```
        type="text"
        class="form-control"
        id="description"
        name="description"
        value="{{ Input::old("description", $description) }}"
      />
    </div>
    <div class="form-group">
      <label for="code">Code</label>
      <span class="help-text text-danger">
        {{ $errors->first("code") }}
      </span>
      <textarea
        class="form-control"
        id="code"
        name="code"
        rows="5"
        placeholder="&lt;div&gt;Hello world&lt;/div&gt;"
      >{{ Input::old("code", $code) }}</textarea>
    </div>
    <input type="submit" name="save" class="btn btn-default" value="Save" />
  </form>
@stop
```

This file should be saved as **app/views/admin/page/edit.blade.php**.

The views follow a similar pattern to those which we created for managing layout files. The exception is that we add the new layout and route fields to the add and edit page templates. We've used the **Form::select()** method to render and select the appropriate layout.

```
Route::any("admin/page/index", [
  "as"   => "admin/page/index",
  "uses" => "PageController@indexAction"
]);

Route::any("admin/page/add", [
  "as"   => "admin/page/add",
  "uses" => "PageController@addAction"
]);

Route::any("admin/page/edit", [
  "as"   => "admin/page/edit",
  "uses" => "PageController@editAction"
]);

Route::any("admin/page/delete", [
  "as"   => "admin/page/delete",
  "uses" => "PageController@deleteAction"
]);
```

This was extracted from **app/routes.php**.

Finally, we add the routes which will allow us to access these pages. With all this in place, we can work on displaying the website content.

## Displaying Content

Aside from our admin pages, we need to be able to catch the all requests, and route them to a single controller/action. We do this by appending the following route to the **routes.php** file:

```
Route::any("{all}", [
  "as"   => "index/index",
  "uses" => "IndexController@indexAction"
])->where("all", ".*");
```

This was extracted from **app/routes.php**.

We pass all route information to a named parameter (**{all}**), which will be mapped to the **IndexController::indexAction()** method. We also need to specify the regular expression with which the route data should be matched. With ".*" we're telling Laravel to match absolutely anything. This is why this route needs to come right at the end of the **app/routes.php** file.

```php
<?php

use Formativ\Cms\EngineInterface;
use Formativ\Cms\FilesystemInterface;

class IndexController
extends BaseController
{
  protected $engine;
  protected $filesystem;

  public function __construct(
    EngineInterface $engine,
    FilesystemInterface $filesystem
  )
  {
    $this->engine     = $engine;
    $this->filesystem = $filesystem;
  }

  protected function parseFile($file)
  {
    return $this->parseContent(
      $this->filesystem->read($file["path"]),
      $file
    );
  }

  protected function parseContent($content, $file = null)
  {
    $extracted = $this->engine->extractMeta($content);
    $parsed    = $this->engine->parseMeta($extracted["meta"]);

    return compact("file", "content", "extracted", "parsed");
  }

  protected function stripExtension($name)
  {
    return str_ireplace(".blade.php", "", $name);
  }

  protected function cleanArray($array)
  {
    return array_filter($array, function($item) {
      return !empty($item);
    });
  }

  public function indexAction($route = "/")
  {
    $pages = $this->filesystem->listContents("pages");

    foreach ($pages as $page)
    {
      if ($page["type"] == "file")
      {
        $page = $this->parseFile($page);

        if ($page["parsed"]["route"] == $route)
        {
          $basename   = $page["file"]["basename"];
          $name       = "pages/extracted/" . $basename;
          $layout     = $page["parsed"]["layout"];
          $layoutName = "layouts/extracted/" . $layout;
          $extends    = $this->stripExtension($layoutName);

          $template = "
            @extends('" . $extends . "')
            @section('page')
              " . $page["extracted"]["template"] . "
            @stop
          ";

          $this->filesystem->put($name, trim($template));

          $layout = "layouts/" . $layout;

          $layout = $this->parseContent(
            $this->filesystem->read($layout)
          );

          $this->filesystem->put(
            $layoutName,
            $layout["extracted"]["template"]
          );

          $data = array_merge(
            $this->cleanArray($layout["parsed"]),
            $this->cleanArray($page["parsed"])
          );

          return View::make(
            $this->stripExtension($name),
            $data
           );
        }
      }
    }
  }
}
```

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

In the IndexController class, we've injected the same two dependencies: **$filesystem** and **$engine**. We need these to fetch and extract the template data.

We begin by fetching all the files in the **app/views/pages** directory. We iterate through them filtering out all the returned items which have a **type** of **file**. From these, we fetch the metadata and check if the route defined matches that which is being requested.

If there is a match, we extract the template data and save it to a new file, resembling **app/views/-pages/extracted/[original name]**. We then fetch the layout defined in the metadata, performing a similar transformation. We do this because we still want to run the templates through Blade (so that **@extends**, **@include**, **@section** etc.) all still work as expected.

We filter the page metadata and the layout metadata, to omit any items which do not have values, and we pass the merged array to the view. Blade takes over and we have a rendered view!

## Extending The CMS

We've implemented the simplest subset of October functionality. There's a lot more going on that I would love to implement, but we've run out of time to do so. If you've found this project interesting, perhaps you would like to take a swing at implementing partials (they're not much work if you prevent them from having metadata). Or perhaps you're into JavaScript and want to try your hand at emulating some of the Ajax framework magic that October's got going on…

You can learn more about OctoberCMS's features at: **http://octobercms.com/docs/cms/themes**.

# Controller Testing

Testing is a hot topic these days. Everyone knows that having tests is a good thing, but often they're either "too busy" to write them, or they just don't know how.

Even if you do write tests, it may not always be clear what the best way is to prepare for them. Perhaps you're used to writing them against a particular framework (which I'm assuming is not Laravel 4) or you tend to write fewer when you can't figure out how exactly to test a particular section of your application.

I have recently spent much time writing tests, and learning spades about how to write them. I have Jeffrey Way to thank, both for **Laravel: Testing Decoded**[1] and **Laracasts.com**[2]. These resources have taught me just about everything I know about unit/functional testing.

This article is about some of the practical things you can do to make your code easier to test, and how you would go about writing functional/unit tests against that code. If you take nothing else away, after reading this, you should be subscribed to **Laracasts.com** and you should read **Laravel: Testing Decoded**.

Since there's already so much in the way of testing, I thought it would be better just to focus on the subject of how to write testable controllers, and then how to test that they are doing what they are supposed to be doing.

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

## Installing Dependencies

For this chapter, we're going to be using PHPUnit and Mockery. Both of these libraries are used in testing, and can be installed at the same time, by running the following commands:

[1]https://leanpub.com/laravel-testing-decoded
[2]http://laracasts.com

```
□ composer require --dev "phpunit/phpunit:4.0.*"

./composer.json has been updated
Loading composer repositories with package information
...

□ composer require --dev "mockery/mockery:0.9.*"

./composer.json has been updated
Loading composer repositories with package information
...
```

We'll get into the specifics of each later, but this should at least install them and add them to the **require-dev** section of your **composer.json** file.

## Unit vs. Functional vs. Acceptance

There are many kinds of tests. There are three which we are going to look at:

1. Unit
2. Functional
3. Acceptance

### Unit Tests

Unit tests are tests which cover individual functions or methods. They are meant to run the functions or methods in complete isolation, with no dependencies or side-effects.

You can find a boring description at: **http://en.wikipedia.org/wiki/Unit_testing**[a].

[a] http://en.wikipedia.org/wiki/Unit_testing

### Functional Tests

Functional tests are tests which concentrate on the input given to some function or method, and the output returned. They don't care about isolation or state.

You can find a boring description at: **http://en.wikipedia.org/wiki/Functional_testing**[a].

[a]http://en.wikipedia.org/wiki/Functional_testing

## Acceptance Tests

Acceptance tests are test which look at a much broader range of functionality. These are often called end-to-end tests because they are concerned with the correct functionality over a range of functions, methods classes etc.

You can find a boring description at: **http://en.wikipedia.org/wiki/Acceptance_testing**[a].

[a]http://en.wikipedia.org/wiki/Acceptance_testing

# Am I Writing Unit Or Functional Tests?

When it comes to writing tests in Laravel 4, developers often think they are writing unit tests when they are actually writing functional tests. The difference is small but significant.

Laravel provides a set of test classes (a base TestCase class, and an ExampleTest class). If you base your tests off of these, you are probably writing functional tests. The TestCase class actually initialises the Laravel 4 framework. If your code depends on that being the case (accessing the aliases, service providers etc.) then your tests aren't isolated. They have dependencies and they need to be run in order.

When your tests only require the underlying PHPUnit or PHPSpec classes, then you may be writing unit tests.

How does this affect us? Well - if your goal is to write functional tests, and you have decent test coverage then you're doing ok. But if you want to write true unit tests, then you need to pay attention to how your code is constructed. If you're using the handy aliases, which Laravel provides, then writing unit tests may be tough.

That should be enough theory to get us started. Let's take a look at some code…

# Fat Controllers

It's not uncommon to find controllers with actions resembling the following:

```php
public function store()
{
  $validator = Validator::make(Input::all(), [
    "title"    => "required|max:50",
    "subtitle" => "required|max:100",
    "body"     => "required",
    "author"   => "required|exists:authors"
  ]);

  if ($validator->passes()) {
    Posts::create([
      "title"     => Input::get("title"),
      "subtitle"  => Input::get("subtitle"),
      "body"      => Input::get("body"),
      "author_id" => Input::get("author"),
      "slug"      => Str::slug(Input::get("title"))
    ]);

    Mail::send("emails.post", Input::all(), function($email) {
      $email
        ->to("cgpitt@gmail.com", "Chris")
        ->subject("New post");
    });

    return Redirect::route("posts.index");
  }

  return Redirect::back()
    ->withErrors($validator)
    ->withInput();
}
```

This was extracted from **app/controllers/PostController.php**. I generated the file with the **controller:make** command.

This is how we first learn to use MVC frameworks, but there comes a point where we are familiar with how they work, and need to start writing tests. Testing this action would be a nightmare. Fortunately, there are a few improvements we can make.

## Service Providers

We’ve used service providers to organise and package our code. Now we’re going to use them to help us thin our controllers out. Create a new service provider, resembling the following:

```php
<?php

namespace Formativ;

use Illuminate\Support\ServiceProvider;

class PostServiceProvider
extends ServiceProvider
{
  protected $defer = true;

  public function register()
  {
    $this->app->bind(
      "Formativ\\PostRepositoryInterface",
      "Formativ\\PostRepository"
    );

    $this->app->bind(
      "Formativ\\PostValidatorInterface",
      "Formativ\\PostValidator"
    );

    $this->app->bind(
      "Formativ\\PostMailerInterface",
      "Formativ\\PostMailer"
    );
  }

  public function provides()
  {
    return [
      "Formativ\\PostRepositoryInterface",
      "Formativ\\ValidatorInterface",
      "Formativ\\MailerInterface"
    ];
  }
}
```

This file should be saved as **app/Formativ/PostServiceProvider.php**.

You will also need to load the **Formativ** namespace through the **composer.json** file. You can do that by using either PSR specification, or even through a classmap. More info on that at: **https://getcomposer.org/doc/01-basic-usage.md#autoloading**[a].

[a]https://getcomposer.org/doc/01-basic-usage.md#autoloading

You can find out more about making service providers at: **http://laravel.com/docs/packages**[a].

[a]http://laravel.com/docs/packages

This does a couple of important things:

1. Interfaces are connected to concrete implementations, so that we can type-hint the interfaces in our controller, and they will be resolved automatically, with the Laravel IoC container.
2. We specify which interfaces are provided (in the **provides()** method) so that we can defer the loading of this service provider until the concrete implementations are called.

We need to define the interfaces and concrete implementations:

```php
<?php

namespace Formativ;

interface PostRepositoryInterface
{
  public function all(array $modifiers);
  public function first(array $modifiers);
  public function insert(array $data);
  public function update(array $data, array $modifiers);
  public function delete(array $modifiers);
}
```

This file should be saved as **app/Formativ/PostRepositoryInterface.php**.

```php
<?php

namespace Formativ;

class PostRepository implements PostRepositoryInterface
{
  public function all(array $modifiers)
  {
    // return all the posts filtered by $modifiers...
  }

  public function first(array $modifiers)
  {
    // return the first post filtered by $modifiers...
  }

  public function insert(array $data)
  {
    // insert posts with $data...
  }

  public function update(array $data, array $modifiers)
  {
    // update posts filtered by $modifiers, with $data...
  }

  public function delete(array $modifiers)
  {
    // delete posts filtered by $modifiers...
  }
}
```

This file should be saved as **app/Formativ/PostRepository.php**.

```php
<?php

namespace Formativ;

interface PostValidatorInterface
{
  public function passes($event);
  public function messages($event);
  public function on($event);
}
```

This file should be saved as **app/Formativ/PostValidatorInterface.php**.

```php
<?php

namespace Formativ;

class PostValidator implements PostValidatorInterface
{
  public function passes($event)
  {
    // validate the event instance...
  }

  public function messages($event)
  {
    // fetch the error messages for the event instance...
  }

  public function on($event)
  {
    // set up the event instance and return it for method chaining...
  }
}
```

This file should be saved as **app/Formativ/PostValidator.php**.

```php
<?php

namespace Formativ;

interface PostMailerInterface
{
  public function send($to, $view, $data);
}
```

This file should be saved as **app/Formativ/PostMailerInterface.php**.

```php
<?php

namespace Formativ;

class PostMailer implements PostMailerInterface
{
  public function send($to, $view, $data)
  {
    // send an email about the post...
  }
}
```

This file should be saved as **app/Formativ/PostMailer.php**.

## Dependency Injection

With all of these interfaces and concrete implementations in place, we can simply type-hint the interfaces in our controller. This is essentially dependency injection. We don't create or use dependencies in our controller - rather they are passed in when the controller is instantiated.

The dependencies are resolved automatically, via the IoC container and reflection.

This makes the controller thinner, and helps us break the logic up into a number of smaller, easier-to-test classes:

```php
<?php

use Formativ\PostRepositoryInterface;
use Formativ\PostValidatorInterface;
use Formativ\PostMailerInterface;
use Illuminate\Support\Facades\Response;

class PostController
extends BaseController
{
  public function __construct(
    PostRepositoryInterface $repository,
    PostValidatorInterface $validator,
    PostMailerInterface $mailer,
    Response $response
  )
  {
    $this->repository = $repository;
    $this->validator  = $validator;
    $this->mailer     = $mailer;
    $this->response   = $response;
  }

  public function store()
  {
    if ($this->validator->passes("store")) {
      $this->repository->insert([
        "title"     => Input::get("title"),
        "subtitle"  => Input::get("subtitle"),
        "body"      => Input::get("body"),
        "author_id" => Input::get("author"),
        "slug"      => Str::slug(Input::get("title"))
      ]);

      $this->mailer->send("cgpitt@gmail.com", "emails.post");

      return $this->response
        ->route("posts.index")
        ->with("success", true);
    }

    return $this->response
      ->back()
      ->withErrors($this->validator->messages("store"))
      ->withInput();
  }
```

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

Another thing you can do, to further modularise your logic, is to dispatch events at critical points in execution:

```
<?php

use Formativ\PostRepositoryInterface;
use Formativ\PostValidatorInterface;
use Formativ\PostMailerInterface;
use Illuminate\Support\Facades\Response;
use Illuminate\Events\Dispatcher;

class PostController
extends BaseController
{
  public function __construct(
    PostRepositoryInterface $repository,
    PostValidatorInterface $validator,
    PostMailerInterface $mailer,
    Response $response,
    Dispatcher $dispatcher
  )
  {
    $this->repository = $repository;
    $this->validator  = $validator;
    $this->mailer     = $mailer;
    $this->response   = $response;
    $this->dispatcher = $dispatcher;

    $this->dispatcher->listen(
      "post.store",
      [$this->repository, "insert"]
    );

    $this->dispatcher->listen(
      "post.store",
      [$this->mailer, "send"]
    );
  }

  public function store()
  {
    if ($this->validator->passes("store")) {
      $this->dispatcher->fire("post.store");

      return $this->response
        ->route("posts.index")
        ->with("success", true);
    }

    return $this->response
      ->back()
      ->withErrors($this->validator->messages("store"))
      ->withInput();
  }
```

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

```php
<?php

namespace Formativ;

use Illuminate\Http\Request;
use Str;

class PostRepository implements PostRepositoryInterface
{
  public function __construct(Request $request)
  {
    $this->request = $request;
  }

  public function insert()
  {
    $data = [
      "title"     => $this->request->get("title"),
      "subtitle"  => $this->request->get("subtitle"),
      "body"      => $this->request->get("body"),
      "author_id" => $this->request->get("author"),
      "slug"      => Str::slug($this->request->get("title"))
    ];

    // insert posts with $data...
  }
```

This was extracted from **app/Formativ/PostRepository.php**.

Using this approach, you can delegate method calls based on events, rather than explicit method calls and variable manipulation.

There are loads of different ways to use events, so you should definitely check out the official docs at: **http://laravel.com/docs/events**[a].

[a] http://laravel.com/docs/events

## This Isn’t Testing!

If you’re wondering how this is related to testing, consider how you would have tested the original controller code. There are many different responsibilities, and places for errors to emerge.

Splitting off your logic into classes of single responsibility is a good thing. Generally following SOLID principles is a good thing. The smaller your classes are, the fewer things each of them do, the easier they are to test.

So how would we write tests for these new classes? Let’s begin with one of the concrete implementations:

```php
<?php

namespace Formativ;

class PostMailerTest
extends TestCase
{
  public function testSend()
  {
    // ...your test here
  }
}
```

This file should be saved as **app/tests/Formativ/PostMailerTest.php**.

In order for this first test case to run, we’ll need to set up a phpunit config file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<phpunit backupGlobals="false"
  backupStaticAttributes="false"
  bootstrap="bootstrap/autoload.php"
  colors="true"
  convertErrorsToExceptions="true"
  convertNoticesToExceptions="true"
  convertWarningsToExceptions="true"
  processIsolation="false"
  stopOnFailure="false"
  syntaxCheck="false"
>
  <testsuites>
    <testsuite name="Application Test Suite">
      <directory>./app/tests/</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

This file should be saved as **phpunit.xml**.

This will get the tests running, and serves as a template in which to start writing our tests. You can actually see this test case working, by running the following command:

```
 phpunit

PHPUnit 4.0.14 by Sebastian Bergmann.

Configuration read from /path/to/phpunit.xml

.

Time: 76 ms, Memory: 8.75Mb

OK (1 test, 0 assertions)
```

This test is functional because it only happens after a Laravel instance is spun up, and it doesn't care about what state it leaves this application instance in.

Since we haven't implemented the body of the **send()** method, it's difficult for us to know what the return value will be. What we can test for is what methods (on the mailer's underlying mail transport/interface) are being called…

Imagine we're using the underlying Laravel mail class to send the emails. We used it before we started optimising the controller layer:

```php
Mail::send("emails.post", Input::all(), function($email) {
  $email
    ->to("cgpitt@gmail.com", "Chris")
    ->subject("New post");
});
```

You can find out more about the Mailer class at: **http://laravel.com/docs/mail**[a].

[a] http://laravel.com/docs/mail

We'd essentially like to use this logic inside the PostMailer class. We should also dependency-inject our Mail provider:

```php
<?php

namespace Formativ;

use Illuminate\Mail\Mailer;

class PostMailer implements PostMailerInterface
{
  public function __construct(Mailer $mailer)
  {
    $this->mailer = $mailer;
  }

  public function send($to, $view, $data)
  {
    $this->mailer->send(
      $view, $data,
      function($email) use ($to) {
        $email->to($to);
      }
    );
  }
}
```

This file should be saved as **app/Formativ/PostMailer.php**.

Now we call the send() method on an injected mailer instance, instead of directly on the facade. This is still a little tricky to test (thanks to the callback), but thankfully much easier than if it was still using the facade (and in the controller):

```
<?php

namespace Formativ;

use Mockery;
use TestCase;

class PostMailerTest
extends TestCase
{
  public function tearDown()
  {
    Mockery::close();
  }

  public function testSend()
  {
    $mailerMock = $this->getMailerMock();

    $mailerMock
      ->shouldReceive("send")
      ->atLeast()->once()
      ->with(
        "bar", ["baz"],
        $this->getSendCallbackMock()
      );

    $postMailer = new PostMailer($mailerMock);
    $postMailer->send("foo", "bar", ["baz"]);
  }

  protected function getSendCallbackMock()
  {
    return Mockery::on(function($callback) {
      $emailMock = Mockery::mock("stdClass");

      $emailMock
        ->shouldReceive("to")
        ->atLeast()->once()
        ->with("foo");

      $callback($emailMock);

      return true;
    });
  }

  protected function getMailerMock()
  {
    return Mockery::mock("Illuminate\Mail\Mailer");
  }
}
```

> This file should be saved as **app/tests/Formativ/PostMailerTest.php**.

Phew! Let's break that up so it's easier to digest…

```
Mockery::mock("Illuminate\Mail\Mailer");
```

Part of trying to test in the most isolated manner is substituting dependencies with things that don't perform any significant function. We do this by creating a new mock instance, via the **Mockery::mock()** method.

We use this again with:

```
$emailMock = Mockery::mock("stdClass");
```

We can use **stdClass** the second time around because the provided class isn't type-hinted. Our **PostMailer** class type-hints the **IlluminateMailMailer** class.

We then tell the test to expect that certain methods are called using certain arguments:

```
$emailMock
  ->shouldReceive("to")
  ->atLeast()
  ->once()
  ->with("foo");
```

This tells the mock to expect a call to an as-yet undefined **to()** method, and to expect that it will be passed "**foo**" as the first (and single) argument. If your production code expects a stubbed method to return a specific kind of data, you can add the **andReturn()** method.

We can provide expected callbacks, though it's slightly trickier:

```
Mockery::on(function($callback) {
  $emailMock = Mockery::mock("stdClass");

  $emailMock
    ->shouldReceive("to")
    ->atLeast()
    ->once()
    ->with("foo");

  $callback($emailMock);

  return true;
});
```

We set up a mock, which expects calls to it's own methods, and then the original callback is run with the provided mock. Don't forget to add **return true** - that tells mockery that it's ok to run the callback with the mock you've set up.

At the end of all of this; we're just testing that methods were called in the correct way. The test doesn't worry about making sure the Laravel **Mailer** class actually sends the mail correctly - that has it's own tests.

The repository class is slightly simpler to test:

```php
<?php

namespace Formativ;

use Mockery;
use TestCase;

class PostRepositoryTest
extends TestCase
{
  public function tearDown()
  {
    Mockery::close();
  }

  public function testSend()
  {
    $requestMock = $this->getRequestMock();

    $requestMock
      ->shouldReceive("get")
      ->atLeast()
      ->once()
      ->with("title");

    $requestMock
      ->shouldReceive("get")
      ->atLeast()
      ->once()
      ->with("subtitle");

    $requestMock
      ->shouldReceive("get")
      ->atLeast()
      ->once()
      ->with("body");

    $requestMock
      ->shouldReceive("get")
      ->atLeast()
      ->once()
      ->with("author");

    $postRepository = new PostRepository($requestMock);
    $postRepository->insert();
  }

  protected function getRequestMock()
  {
    return Mockery::mock("Illuminate\Http\Request");
  }
}
```

This file should be saved as **app/tests/Formativ/PostRepositoryTest.php**.

All we're doing here is making sure the get method is called four times, on the request dependency. We could extend this to accommodate requests against the underlying database connector object, and the test code would be similar.

We can't completely test the **Str::slug()** method because it's not a facade but water a static method on the Str class. Every facade allows you to mock methods (facades subclass the **MockObject** class), and you can even swap them out with your own mocks (using the **Validator::swap($validatorMock)** method).

You can test static method calls using the AspectMock library, which you can learn more about at: **https://github.com/Codeception/AspectMock**[a].

[a]https://github.com/Codeception/AspectMock

You can learn more about facades at: **http://laravel.com/docs/facades**[a].

[a]http://laravel.com/docs/facades

Finally, let's test the controller:

```php
<?php

class PostControllerTest
extends TestCase
{
  public function tearDown()
  {
    Mockery::close();
  }

  public function testConstructor()
  {
    $repositoryMock = $this->getRepositoryMock();

    $mailerMock = $this->getMailerMock();

    $dispatcherMock = $this->getDispatcherMock();

    $dispatcherMock
      ->shouldReceive("listen")
      ->atLeast()
      ->once()
      ->with(
        "post.store",
        [$repositoryMock, "insert"]
      );

    $dispatcherMock
      ->shouldReceive("listen")
      ->atLeast()
      ->once()
      ->with(
        "post.store",
        [$mailerMock, "send"]
      );

    $postController = new PostController(
      $repositoryMock,
      $this->getValidatorMock(),
      $mailerMock,
      $this->getResponseMock(),
      $dispatcherMock
    );
  }

  public function testStore()
  {
    $validatorMock = $this->getValidatorMock();

    $validatorMock
      ->shouldReceive("passes")
      ->atLeast()
      ->once()
      ->with("store")
      ->andReturn(true);

    $responseMock = $this->getResponseMock();

    $responseMock
      ->shouldReceive("route")
      ->atLeast()
      ->once()
      ->with("posts.index")
      ->andReturn($responseMock);

    $responseMock
      ->shouldReceive("with")
      ->atLeast()
      ->once()
      ->with("success", true);

    $dispatcherMock = $this->getDispatcherMock();

    $dispatcherMock
      ->shouldReceive("fire")
      ->atLeast()
      ->once()
      ->with("post.store");

    $postController = new PostController(
      $this->getRepositoryMock(),
      $validatorMock,
      $this->getMailerMock(),
      $responseMock,
      $dispatcherMock
    );

    $postController->store();
  }

  public function testStoreFails()
  {
    $validatorMock = $this->getValidatorMock();

    $validatorMock
      ->shouldReceive("passes")
      ->atLeast()
      ->once()
      ->with("store")
      ->andReturn(false);

    $validatorMock
      ->shouldReceive("messages")
      ->atLeast()
      ->once()
      ->with("store")
      ->andReturn(["foo"]);

    $responseMock = $this->getResponseMock();

    $responseMock
      ->shouldReceive("back")
      ->atLeast()
      ->once()
      ->andReturn($responseMock);

    $responseMock
      ->shouldReceive("withErrors")
      ->atLeast()
      ->once()
      ->with(["foo"])
      ->andReturn($responseMock);

    $responseMock
      ->shouldReceive("withInput")
      ->atLeast()
      ->once()
      ->andReturn($responseMock);

    $postController = new PostController(
      $this->getRepositoryMock(),
      $validatorMock,
      $this->getMailerMock(),
      $responseMock,
      $this->getDispatcherMock()
    );

    $postController->store();
  }

  protected function getRepositoryMock()
  {
    return Mockery::mock("Formativ\PostRepositoryInterface")
      ->makePartial();
  }

  protected function getValidatorMock()
  {
    return Mockery::mock("Formativ\PostValidatorInterface")
      ->makePartial();
  }

  protected function getMailerMock()
  {
    return Mockery::mock("Formativ\PostMailerInterface")
      ->makePartial();
  }

  protected function getResponseMock()
  {
    return Mockery::mock("Illuminate\Support\Facades\Response")
      ->makePartial();
  }

  protected function getDispatcherMock()
  {
    return Mockery::mock("Illuminate\Events\Dispatcher")
      ->makePartial();
    }
}
```

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

Here we're still testing method calls, but we also test multiple paths through the **store()** method.

We haven't used any assertions, even though they are very useful for unit and functional testing. Feel free to use them to check output values…

You can find out more about Mockery at: **https://github.com/padraic/mockery**[a].

[a] https://github.com/padraic/mockery

You can find out more about PHPUnit at: **http://phpunit.de**[a].

[a] http://phpunit.de

## The Rabbit Hole

The process of test-writing can take as much time as you want to give it. It's best just to decide exactly what you need to test, and step away after that.

We've just looked at a very narrow area of testing, in our applications. You're likely to have a richer data layer, and need a ton of tests for that. You're probably going to want to test the rendering of views.

Don't think this is an exhaustive reference for how to test, or even that this is the only way to functionally test your controller code. It's simply a method I've found works for the applications I write.

## Alternatives

The closest alternative to testing with PHPUnit is probably PHPSpec (**http://www.phpspec.net**[3]). It uses a similar dialect of assertions and mocking.

If you're looking to test, in a broader sense, consider looking into Behat (**http://behat.org**[4]). It uses a descriptive, text-based language to define what behaviour a service/library should have.

[3]http://www.phpspec.net
[4]http://behat.org