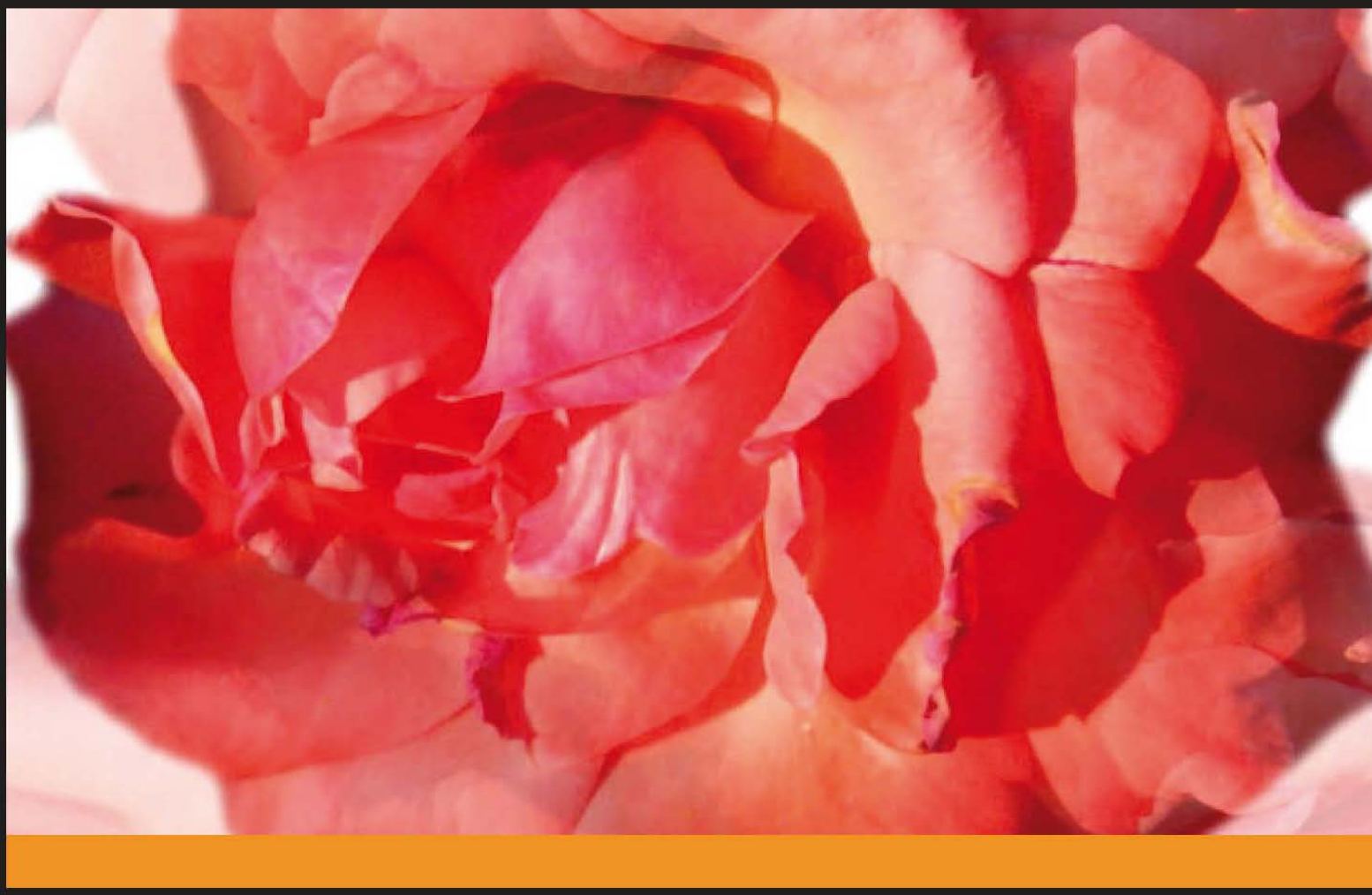
open source
community experience distilled



Laravel Starter

The definitive introduction to the Laravel PHP web development framework

Foreword by Taylor Otwell, Creator of the Laravel PHP framework

Shawn McCool

[PACKT] PUBLISHING

Laravel Starter

The definitive introduction to the Laravel PHP web development framework

Shawn McCool

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Laravel Starter

First published: November 2012

Production Reference: 1151112

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK..

ISBN 978-1-78216-090-8

www.packtpub.com

Credits

Author

Shawn McCool

Reviewer

Alexander Cogneau

Terry Matula

Acquisition Editor

Mary Nadar

Commissioning Editor

Maria D'souza

Technical Editor

Prasad Dalvi

Project Coordinator

Priya Sharma

Proofreader

Maria Gould

Graphics

Aditi Gajjar

Production Coordinator

Prachali Bhiwandkar

Cover Work

Prachali Bhiwandkar

Cover Image

Sheetal Aute

Foreword

When I first wrote the Laravel framework, it was because I needed a way to quickly develop my ideas for great web applications. Even more, I wanted to enjoy developing them. It turns out that the things I built to make my development enjoyable others have found enjoyable as well. Laravel has grown into a mature framework with a vibrant, friendly community, and shows promise of continuing to do so for years to come.

Laravel is a simple, yet powerful framework. Its syntax is terse, yet expressive. It is fitting that an introduction to Laravel would be just as sharp and straightforward, helping developers quickly learn how to build their web applications with the framework. I'm very pleased that Shawn has undertaken this work of providing a wonderful introduction to Laravel.

Shawn has been a key part of the Laravel community for some time, and has dedicated much of his time to helping other developers learn to build great applications. But, Shawn's experience isn't "all talk". He has proven himself as a Laravel developer by delivering production-ready applications to the world. I trust he will faithfully light the path on your journey to do the same.

Taylor Otwell,
Creator of the Laravel PHP framework

About the author

Shawn McCool is the co-founder of the web agency Big Name. He has been involved in professional web-application development for 15 years. His work emphasizes a philosophy of long-term software maintainability.

Shawn joined the Laravel team in 2012 and has been involved in updating and improving the official documentation.

Shawn is highly involved in community education and regularly creates education content and makes it freely available on his company site `http://heybigname.com`. Shawn spends much of his time involved with the Laravel community and can usually be found providing help in the official *#Laravel IRC* channel on FreeNode.

I'd like to thank my life partner Danielle and my business partner and favorite designer Justin for all of their encouragement.

About the reviewers

Alexander Cogneau is a student, who makes websites as a hobby. He is a developer for the FluxBB Forum Software. He has extensive knowledge of Laravel (the PHP MVC framework).

Terry Matula is a web developer and Laravel advocate based in Austin, TX.

He's been a passionate computer enthusiast since he first played Oregon Trail on an Apple computer. He started programming in BASIC, making simple Scott Adams-like games on a Commodore Vic-20, at a young age. Since then, he's worked as a developer using Flash/ActionScript, ASP.NET, PHP, and numerous PHP frameworks, with Laravel being his favorite by far.

He blogs web development tips and tricks at his website `http://terrymatula.com`.

I'd like to thank my beautiful wife Michelle for all her encouragement and support. I'd also like to thank my son Evan, for keeping me grounded and being a shining light in my life.

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version atwww.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Table of Contents

Laravel Starter

Welcome to the Laravel Starter. This book has been specially created to provide you with all the information that you need to get started with the Laravel web development framework. You will learn the basics of Laravel, get started with building your first web application, and discover some tips and tricks for using Laravel.

This guide contains the following sections:

So, what is Laravel? – find out what Laravel actually is, what you can do with it, and why it's so great.

Installation – this section will get you started on programming with Laravel. We'll go over installation and basic configuration so that we can get started on our application!

Quick start: Creating your first web application – let's get started by making our own application. In this section, we'll develop a basic application that receives input from forms and then update a database accordingly.

Top 5 features you need to know about – here we will go more in depth about what we've covered in the *Quick start* section. We'll learn more about Eloquent, authentication, filters, validation, and bundles.

People and places you should get to know – the Laravel community is one of its most potent educational assets. This section provides you with useful links to the project's page and forums, as well as a number of helpful articles, tutorials, blogs, and the Twitter feeds of Laravel super-contributors.

So, what is Laravel?

Laravel is an MVC web-development framework written in PHP. It has been designed to improve the quality of your software by reducing both the cost of initial development and ongoing maintenance costs, and to improve the experience of working with your applications by providing clear expressive syntax and a core set of functionality that will save you hours of implementation time.

Laravel was designed with the philosophy of using convention over configuration. This means that it makes intelligent assumptions about what you're trying to accomplish so that in most situations you'll be able to accomplish your goals with much less code. Not every application and database that you'll work with will be designed using these conventions. Thankfully, Laravel is flexible enough to work with your system—no matter how unique it is.

Laravel has been designed to target the sweet-spot between minimalism and functionality. It's easier to understand smaller code bases and Laravel is all about implementing solutions in a way that is clean, simple, and elegant. Long-time PHP developers will find many aspects of Laravel familiar as it is an evolution of the PHP development frameworks that have come before it.

Laravel is one of the few PHP frameworks that offers true code modularity. This is achieved through a combination of drivers and its bundles system. Drivers allow you to easily change and extend caching, session, database, and authentication functionality. Using bundles, you're able to package up any kind of code for either your own re-use or to provide to the rest of the Laravel community. This is very exciting because anything that can be written in Laravel can be packaged as a bundle, from simple libraries to entire web-applications. The Laravel bundle website allows you to browse bundles that have been built by the community as well as to showcase your own. It is a valuable resource of third-party libraries and subsystems that can dramatically ease the development of your web-application.

Laravel also provides a cutting-edge suite of tools for interacting with databases. Database migrations enable you to easily design and modify a database in a platform-independent way. The migrations can then be run against any of the database types that Laravel supports (MySQL, PostgreSQL, MSSQL, and SQLite) and you won't have any compatibility issues.

Laravel's Fluent Query Builder abstracts away the differences between different database types. Use it to build and execute robust queries.

Laravel's ActiveRecord implementation is called **Eloquent**. Interacting with a database in an object-oriented way is the modern standard. With Eloquent, we can create, retrieve, update, and delete the database records without needing to write a single line of SQL. In addition to this, Eloquent provides powerful relationship management and it can even handle pagination automatically for you.

Laravel also ships with a command-line interface tool called **Artisan**. With Artisan, a developer can interact with their application to trigger actions such as running migrations, running unit tests, and running scheduled tasks. Artisan is also completely extendable so that you can write any type of functionality that you'd like. Laravel's easy-to-manage routing system allows you to easily manage your site's URLs. By using the built-in HTML helper, you can create links within your site that will automatically update themselves if you change the URLs that make the job of maintaining your site much easier.

The Blade templating engine cleans up your views by providing aesthetically pleasing replacements for inline PHP and by including powerful new features.

Installation

In five easy steps, you can install Laravel and get it set up on your system.

Step 1 – What do I need?

Before you install Laravel, you will need to check that you have all of the required elements, as follows:

- Laravel requires a web server environment and will run in Apache, IIS, and Nginx easily. Laravel should run in any server environment that supports PHP. The easiest way to set up a local webserver for development is to install XAMPP (Windows), MAMP (Mac OSX), or Apache with PHP5 on through a package manager on Linux.
- Laravel is written in the PHP scripting language. Currently, Laravel v3.2.5 requires a minimum of PHP v5.3 to run.
- Laravel requires that you have the FileInfo and Mcrypt libraries installed. Conveniently, they are almost always installed by default.
- .For our QuickStart application we require a database. Out of the box, Laravel supports MySQL, MSSQL, PostgreSQL, and SQLite.

Step 2 – Downloading Laravel

The easiest way to download Laravel is as a compressed package from `http://laravel.com/download`.

Alternatively, you can download Laravel by cloning its `git` repository from `GitHub.com` with the following command.

```
git clone git@github.com:laravel/laravel.git
```

It would be better that you download the most current stable build.

Extract the contents of the compressed package into the directory that you store your web-applications. Typical locations include `/Users/Shawn/Sites`, `c:\sites`, and `/var/www/vhosts/` depending on your operating system.

We'll assume that your first Laravel installation is in `c:\sites\myfirst\`.

Step 3 – Configuring hosts

Let's go ahead and set up our web server to host our site. We need to choose a host name for our example application. This is our first application and we're working on our local development environment, so let's use `http://myfirst.dev?`.

In Linux and OSX, simply add the following line to your `/etc/hosts` file:

```
127.0.0.1 myfirst.dev
```

Windows users should add that line to their `c:\windows\system32\drivers\etc\hosts` file.

Now you should be able to ping `myfirst.dev` and see that it resolves to `127.0.0.1`.

Step 4 – Setting up your VirtualHost

Now that we have a host name we need to tell our web server where to find the Laravel installation. Add the following `VirtualHost` configuration to your Apache web server configuration and replace `DocumentRoot` with the path to your Laravel installation's public directory.

```
<VirtualHost *:80>
  ServerName myfirst.dev
  DocumentRoot c:/sites/myfirst/public
</VirtualHost>
```

It's very important to note that `DocumentRoot` points to the Laravel's public directory. There are multiple security reasons for this. A momentary server misconfiguration can expose the secure information, such as your database passwords.

Step 5 – Restarting your web server and testing

Now that you've installed the Laravel files, added your host declaration, and updated your web-server configuration you're ready to go! Restart your web-server software and go to `http://myfirst.dev` in your browser. You should see the Laravel splash page!

And that's it!

By this point, you should have a working installation of Laravel and are free to play around and discover more about it.

Quick start: Creating your first web application

In this section we'll be building a user administration tool. User admins are one of the most common components to be found in web applications. They also use a number of important systems that we want to explore with Laravel including database interactions, forms, and routing.

We'll be storing and retrieving records from a database so now it would be a good time for you to create a database for this application. Keep the database name, user, and password handy.

Let's get started.

Step 1 – Database configuration

Armed with our database name, user, and password, we can now tell Laravel how to connect to our database.

Open the file `application/config/database.php` and scan the contents. You'll find example configurations for each database driver. Determine which driver you're going to use from the available options (`sqlite`, `mysql`, `pgsql`, and `sqlserv`) and enter the name of the driver as the default database connection.

Then, in the **Connections** section add your database name, user, and password.

Ok! We're just where we want to be. Let's get started by creating the users table.

Step 2 – Creating the users table using migrations

You might typically create the users table with a tool like phpMyAdmin or Navicat. But, Laravel provides a fancy migrations system for us and we should use it because it improves our workflow and reduces deployment bugs.

Migrations are version control for your schema modifications. Migrations reduce the amount of headache that we face by requiring us to only define the schema alteration once. Afterwards, we can deploy our changes to any number of systems without the potential for human error.

Migrations are especially useful for projects in which you're collaborating with others. Like using source control, migrations are always a good idea.

Developers who are new to migrations might believe them unnecessary or believe that they add too much additional work at first. But, stick with it and their value will become quickly apparent.

Let's start by creating our migration template with Laravel's command-line tool Artisan.

In order to use Artisan, we will need to add the directory which contains the PHP binary to our `PATH` environment variable. This lets us execute the PHP binary from anywhere, as the system will know where to find it. You can test this by running the following command from your command-line terminal:

```
# php -v
```

You should see a nice readout telling you which version of PHP you're running.

If you find that your command-line interface doesn't know where the PHP binary is, you'll need to update your system's `PATH`. You can modify your `PATH` variable on OS X and Linux with the following command line:

```
# export PATH=/path/to/php/binary:$PATH
```

Windows users will need to right-click on **Computer** from the start menu and click on **Properties**. Click on **Advanced system settings**, and then click on **Environment Variables**. Here you can find the system variable `PATH` and add the directory that contains your PHP binary.

Now that the PHP binary is in our path, let's navigate to our `project` folder. Now we can go ahead and install our migrations database table. Execute the following command:

```
# php artisan migrate:install
```

You should see the message, **Migration table created successfully**. If you get errors, verify that your database connection information is correct.

Next, let's create a new migration template in our `application/migrations` folder.

```
# php artisan migrate:makecreate_users_table
```

The migration files are formatted with the year, month, day, time, and the text that you added that enable it to be identified by a human, in this case `create_users_table`. Mine looks like `2012_08_16_112327_create_users_table.php`. The structure of the filename is important as it helps Laravel to understand the order in which it should run the migrations. By using a convention for naming your migrations, you'll be helping your team to better understand your work (and vice-versa). An example convention might consist of entries like `create_users_table`, `add_fields_to_users_table`, or `rename_blog_posts_table_to_posts`.

The migrations file contains a single class with the human readable name that we entered before. The class has two methods, `up()` and `down()`.

When migrations are run, Laravel reads in each migration file one by one and runs its `up()` method. If you feel that a mistake has been made, you can rollback migrations. When rolling back a change, Laravel runs the `down()` method. The `up()` method is for making your desired changes to the database. The `down()` method is for reverting your changes.

Let's go ahead and look at what our `create_users_table migration` should look like:

```
classCreate_Users_Table
{

  /**
  * Make changes to the database.
  *
  * @return void
  */
  public function up()
  {
    Schema::table('users', function($table)
    {
      $table->create();

      $table->increments('id');

      $table->string('email');
      $table->string('real_name');
      $table->string('password');

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

  /**
  * Revert the changes to the database.
  *
  * @return void
  */
  public function down()
  {
    Schema::drop('users');
  }

}
```

Let's first discuss our `up()` method. Our goal is to create the `users` table and to define the fields within it. In order to accomplish this goal, we'll use the Laravel `Schema` class. When creating or making modifications to tables, we use the `Schema` class' `table()` method.

`Schema::table()` accepts two arguments. The first is the name of the table that you'll be interacting with, in this case it's `users`. The second argument is a closure which contains your table definition. The closure receives the argument `$table` and this is the object that we'll be interacting with to define the table.

```
$table->create();
```

This line tells Laravel that the table will need to be created. If we omit this line, `Schema` will generate the `ALTER TABLE` syntax rather than the `CREATE TABLE` syntax.

```
$table->increments('id');
```

The `increments()` method tells `Schema` that the specified field should be an auto-incremented primary key. With Laravel, you'll want to use simple field names such as `id`, `email`, and `password`. If you aren't familiar with using **Object-Relational Mapping** (**ORM**), you may be in the habit of creating those same field names with the table name as a prefix. For example, `user_id`, `user_email`, `user_password`. The purpose behind defining field names with the table name as a prefix is to simplify query generation when using a query builder. This is no longer necessary and it's best to follow the more simple convention as it manages redundant interactions for you, removing the need for you to continuously write the boilerplate code.

```
$table->string('email');
$table->string('real_name');
$table->string('password');
```

Next we have a few string declarations. These will be created as the `VARCHAR` fields with the default length of `200`. You can override the length of these fields by passing a second argument that represents the intended length. For example:

```
$table->string('email', 300);
```

This line creates a `VARCHAR` field named `email` with a length of `300`.

It's important to note that we shouldn't reduce the size of the `password` field as we'll need that length for the output from Laravel's `Hash` class.

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

Finally, we come to the `timestamps()` method. This will create two `DATETIME` fields (`created_at` and `updated_at`). It is not unreasonable to create the `timestamp` fields for every table in the database as they can be very useful for troubleshooting down the road. The Eloquent ORM will automatically manage these `timestamp` fields for us. So, we can forget about them for now.

The `down()` method should revert any changes made to the `up()` method. In this case, the `up()` method creates a database table called users. So, the `down()` method should remove the table.

```
Schema::drop('users');
```

This is done with the method `Schema::drop()`. `drop()` takes a single argument, a string value containing the name of the table that you wish to drop.

That's it! We have our first migration. Once you memorize the commonly used methods such as `increments()`, `string()`, `decimal()`, `timestamps()`, and `date()`, you'll be able to make migrations just as fast as you were able to modify your database with your preferred database management tool. But, now we gain the added benefit from using them in versioned and collaborative situations.

Now, we're ready to run our migrations. From this point on, running migrations will always be done in the same way. Let's go ahead and give it a shot:

```
# php artisan migrate
```

Now, we should see the message, **Migrated: 2012_08_16_112327_create_users_table**.

It's very important to test our migrations. If we don't test our migrations, it could come back to bite us later in the project when we need to roll back migrations and run into an error. Proper migration testing verifies that the `up()` and `down()` methods both function as intended.

To test the `up()` method, run the migration and open your preferred database management application. Then, verify that everything is as you intended. Then, test the `down()` method by rolling back the migration and doing the same. Roll back your migration now by using the following command:

```
# php artisan migrate:rollback
```

Optimally, you will be notified that the migration was rolled back successfully. Double-check that your database no longer contains the `users` table. That's it! This migration is good to go. Run your migrations for one last time and let's move on to the next step.

```
# php artisan migrate
```

Step 3 – Creating an Eloquent user model

Now that we have created our `users` table, we should go ahead and create our user model. In the context of MVC, a model is a class that represents various types of data interactions. The data can include information stored in a database such as users, blog posts, and comments or interactions with many other types of data sources such as files, forms, or web services. For the sake of this document, we'll primarily be using models to represent the data that we store in our database.

Models sit in the `application/models` folder. So go ahead and create the file `application/models/user.php` with the following code:

```
class User extends Eloquent
{

}
```

This is all we need! This tells Laravel that our user model represents data in the users table. Wait! How does Laravel know that? Well, because we're following Laravel's conventions of course! Our database table `users` is plural because it signifies that it stores more than one user record. The model class is named `User` singular because it represents one single user record in the `users` table. The `User` class name is capitalized because of the standard for using Pascal case for class names. If your database table was named `user_profiles`, your model's class name would be `UserProfile`.

We can see how using conventions prevents us from having to make a bunch of configurations. Well, what if we must use a database table that doesn't follow conventions? No problem! We can just define the table name manually. Just add the following line to the `User` class:

```
public static $table = 'my_users_table';
```

That's all it takes. Now, Laravel knows that when interacting with this model, it should use the table named `my_users_table`. Most conventions in Laravel can be overridden with configuration when necessary.

There's one important thing that we should add to our user model. We're storing the user's e-mail address, real name, and password. We want to make sure that the user's password isn't stored in plain text. We need to hash their password before it is stored in the database. For this we'll create a setter.

A **setter** is a method that intercepts the assignment of an attribute. In this case, we're going to intercept the assignment of the password attribute, hash the value that we received, and then store the hashed value in the database.

Let's look at some code.

```
class User extends Eloquent
{
  public function set_password($string)
  {
    $this->set_attribute('password', Hash::make($string));
  }
}
```

As you can see, the convention for declaring setters is to prefix the name of the attribute whose assignments you want to intercept with `set_`. The user's password will be passed to the setter as the argument `$string`.

We use the `set_attribute()` method to store a hashed version of the user's password into the model. Typically the `set_attribute()` method is not necessary. But, we don't want our setter to be stuck in an endless loop as we continuously attempt to assign `$this->password`. The `set_attribute()` method accepts two arguments. The first is the name of the attribute and the second is the value that we want to assign to it. When assigning values with `set_attribute()`, setter methods will not be called and the data will be directly modified within the model.

We're using the `make()` method from Laravel's `Hash` class to create a salted hash of the user's password.

Step 4 – Routing to a closure

Before we can move on and test our user model, we need to know a few things about routing in Laravel. **Routing** is the act of linking a URL to a function in your application. In Laravel, it's possible to route in two ways. You can either route to a closure or a controller action. As we'll be going over controllers in more detail later, let's start by looking at how we can route to a closuRoutes in Laravel are declared in `application/routes.php`. This file will represent the connection between your site's URLs and the functions that contain application logic for your site. This is very handy as other developers will be able to come into your project and know how requests are routed, simply by reviewing this file.

Here is a simple example of routing to a closure:

```
Route::get('test', function()
{
  return "This is the test route.";
});
```

We're using the `Route::get()` method to define the route. `Route::get()` registers a closure with the router that specifically responds to a `GET` request at the specified URI. To register a closure for the `POST`, `PUT`, and `DELETE` requests, you'd use `Route::post()`, `Route::put()`, and `Route::delete()` respectively. These methods correspond to what are commonly referred to as the **HTTP verbs**.

Typically, developers only interact with the `GET` and `POST` requests. When a user clicks on a link or enters a URL in their address bar, they're creating a `GET` request. When a user submits a form, they're typically creating a `POST` request.

The first argument for the `Route::get()` method is the URI for the route (the part of the URL after the domain name), and the second argument is the closure which contains the desired application logic.

Let's update the example and test the route.

Notice that instead of using echo to output the string we're returning it. That's because whether you route to a closure or route to a controller action, you should always return your response. This allows Laravel to handle many situations in a robust way.

Now go ahead and navigate to `http://myfirst.dev/test`. You will see the message, **This is the test route**.

Step 5 – Creating users with Eloquent

Now, let's test the `User` model and learn a bit about Eloquent in the process. In this application, we're going to interact with the `User` model in a few ways. We'll want to use the `Create`, `Retrieve`, `Update`, and `Delete` methods for user records. These common methods are referred to as **CRUD methods**.

Eloquent simplifies development by removing the need to manually implement CRUD methods for your models. If you've ever designed models without an ORM, you are already aware that this alone can save you many hours on large sites.

Now, let's explore the various ways in which you can create new user records. We'll repurpose our test route from the previous step to help us get to know Eloquent. Update the route declaration with the following code:

```
Route::get('test', function()
{
  $user = new User;

  $user->email = "test@test.com";
  $user->real_name = "Test Account";
  $user->password = "test";

  $user->save();

  return "The test user has been saved to the database.";
});
```

Let's review:

```
$user = new User;
```

First, we create a new instance of our `User` model and store it in the `$user` variable:

```
$user->email = "test@test.com";
$user->real_name = "Test Account";
$user->password = "test";
```

Then, we set some attributes in our `User` model. These attributes directly correspond to the fields in our users database table.

```
$user->save();
```

Next, we tell Eloquent that we want to save the contents of this model to the database.

```
return "The test user has been saved to the database.";
```

Finally, we output this string to the browser so that we know that all is well.

Go ahead and navigate to `http://myfirst.dev/test` in your browser. You should see the confirmation message that the user has been saved to the database.

Now, take a look at the contents of your database's `users` table. You will see a new record filled with our data. Notice that the `timestamps` fields have been automatically pre-populated for you. It's that easy to create new database records with Eloquent!

Step 6 – The users controller

Now it's time for us to create our first controller. You've already learned how we can route to a closure and you can use this method to make an entire web-application. So, what are controllers and why should we use them?

Controllers are containers for methods that contain application logic related to a common domain. A domain is simply a categorization of purpose. In the context of our web application, we will be working solely with administrating user data. Therefore, our domain is `users`. Our `users` controller will contain the application logic that controls our application flow and delivers our database data to the view for formatting.

Since controllers allow us to group logic, we can also apply configurations to a controller that will affect all of the methods inside it. We'll explore more of this concept later.

Let's create the file `application/controllers/users.php` and fill it with our controller class' skeleton:

```
<?php

class Users_Controller extends Base_Controller
{

  public function action_index()
  {
    return "Welcome to the users controller.";
  }

}
```

Our `users` controller is a class whose name is constructed from the domain of the methods contained within and is suffixed with `_Controller`. Since this controller's domain is user accounts, our controller is named `Users_Controller`. The `_Controller` suffix is required because it prevents controller classes from having name collisions with other classes in the system.

A controller's domain should always be plural when applicable.

You'll notice that our `Users_Controller` class extends Laravel's default `Base_Controller` class. This is a good practice because if we need some code or configurations to affect all of our controllers, we can just edit the file `application/controllers/base.php` and make changes to the `Base_Controller` class. Every controller that extends the `Base_Controller` class will be affected.

You'll also notice that we have defined a controller action named `index`. A **controller action** is a method within a controller class that we intend to be the destination for a route. You may decide to create methods within a controller class that will only be called from other methods within that class; these would not be actions.

Controller actions are named with the prefix `action_`. This is an important distinction because we do not want users to be able to access methods within our controller that aren't actions.

So, now that we have this controller how can we access the `index` action from our browser? For now, we can't. We haven't routed any URL to that controller action. So, let's do that. Open up `application/routes.php` and add the following line:

```
Route::controller('users');
```

As you can see, we can register an entire controller with the router with one command. Now, we can access our `users` controller's `index` action with `http://myfirst.dev/users/index`. The `index` action is also considered to be the default action for a controller, so we can also access our `index` action at `http://myfirst.dev/users`.

It's important to note that while routing to closures is convenient, routing to controllers is generally considered better practice for a few reasons. Controllers are not loaded into memory until their routes are accessed, which helps to reduce the memory footprint of your application. They also make maintenance easier by making it quite clear where the developer can find the code for the route. Controllers are derived from a base class, so it's simple to make a change in one class and through inheritance have that change affect other classes. Finally, since controllers are actions grouped by purpose, it's often quite convenient to assign filters on a per-controller basis. We'll talk more about filters in the section *Top 5 features you need to know about*.

Step 7 – Creating the users index view

Now we can go to `http://myfirst.dev/users` and access the `index` method of our controller. That's pretty cool, but our `users` controller's `index` page needs to show us a list of the users in the system. To display a list of users, we're going to need to create a view.

A **view** is a file that contains formatting data (typically HTML). PHP variables, conditionals, and loops are used within the view to display and format dynamic content.

Laravel provides its own templating system called **Blade**. Blade removes PHP tags and provides shortcuts for common tasks so that your views are cleaner and easier to create and maintain.

Let's get started by creating the folder `application/views/users/`. This folder will store all of the views for our `users` controller. It is a standard convention to create a folder under `application/views` for each controller that needs a view. Then, create the view file at `application/views/users/index.blade.php`. The convention is to name the view file after the controller action in which it's used. In this example, we're using Blade. If you do not wish to use Blade simply name the file `index.php`.

Let's fill the view with the following HTML:

```
<h1>Users</h1>

<ul>
  <li>Real Name - Email</li>
</ul>
```

It's not pretty. But, it's easy to understand.

Now, we'll make a modification to the `users` controller's `index` action:

```
public function action_index()
{
  return View::make('users.index');
}
```

Now, where we were previously returning a string, we'll return a View object. Let's take a closer look at what's happening here.

The `View` class' `make()` method is a factory method that is used to generate View objects. In this case, we're passing the argument `users.index`. This is Laravel's internal notation for referring to view files. The notation is made up of the path to the view file relative to the `application/views` directory including the filename without its file extension. For example, `application/views/users/index.php` would be written as `users.index`, and `application/views/admin/users/create.php` would be written as `admin.users.create`.

It's important to note that we're returning the View object instead of using echo to send the rendered contents of the view to the browser. This is an important aspect of the way that Laravel works. We'll never use echo from within a routed closure or a controller action. Instead, we'll always return the result and allow Laravel to handle things appropriately.

Now, when we go to `http://myfirst.dev/users`, we'll see the view that we just created!

Step 8 – Passing data from a controller to a view

The fact that we can go to a URL and see the view file that we created is pretty cool. But, we need to be able to see the list of users from our database. To accomplish this goal, we'll first query the `User` model from our controller action, and then pass that data to the view. Finally, we'll update our view to display the user data received from the controller.

Let's start by updating our `users` controller's `index` action:

```
public function action_index()
{
  $users = User::all();

  return View::make('users.index')->with('users', $users);
}
```

Let's look at this line by line:

```
$users = User::all();
```

First, we request all users as objects from Eloquent. If we have no rows in our `users` table, `$users` will be an empty array. Otherwise, `$users` will be an array of objects. These objects are instantiations of our `User` class.

```
return View::make('users.index')->with('users', $users);
```

Then, we modify the creation of our `View` object a bit. We chained a new method named `with()`. The `with()` method allows us to pass data from the controller into the view. This method accepts two arguments. The first argument is the name of the variable that will be created in the view. The second argument will be the value of that variable.

To recap, we've queried the database for all users and passed them as an array of `User` objects to the view. The array of `User` objects will be available in the view as the variable `$users` due to the fact that `users` was the first argument to the `with()` method.

Step 9 – Adding our dynamic content to the view

Now that our view has access to the user data, let's update the view so that we can actually see those users.

```
<h1>Users</h1>

@if($users)
  <ul>
    @foreach($users as $user)
      <li>{{ $user->real_name }} - {{ $user->email }}</li>
    @endforeach
  </ul>
```

```
@else
  Looks like we haven't added any users, yet!
@endif
```

Blade is easy to understand and results in much more elegant code. The {{ }} tags output the results of the expression within them and replace a typical echo command. Other constructions such as `if()`, `foreach()`, and `for()` are the same, but without PHP tags and with a preceding `@`.

Blade doesn't incur a significant performance penalty since it renders to a raw PHP cache. Parsing of a Blade template is only done when changes have been made.

It's also important to note that you are still free to use PHP tags in the Blade templates.

As you can see we use an if statement to check if the `$users` array contains data. If it does, we loop through the users and display a new list item for each. If the `$users` array doesn't contain any data, we output a message saying so.

And that's it! Save the file and hit reload in your browser, and you will see the test account that we created.

This may be a good time to play around with the HTML or add new users using the `test` route that we made to get a better feel for how everything works.

Step 10 – RESTful controllers

We've already mentioned that Laravel's routing system enables you to route `GET`, `POST`, `PUT`, and `DELETE` requests to closures. But we haven't talked about how to individually route them to controller actions.

The answer is RESTful controllers! RESTful controllers enable you to route to different controller actions based on the request method. Let's configure our application to use RESTful controllers.

Since all of our controller classes are derived from the `Base_Controller` class, we can simply add the `$restful` configuration to it and all of our controllers will be affected.

Update your `Base_Controller` class to look like this:

```
classBase_Controller extends Controller
{
  public $restful = true;

  /**
```

```
    * Catch-all method for requests that can't be matched.
    *
    * @param  string    $method
    * @param  array     $parameters
    * @return Response
    */
    public function __call($method, $parameters)
    {
      return Response::error('404');
    }

}
```

Now, every controller that extends `Base_Controller` (including our own `Users_Controller`) is a RESTful controller!

But, wait. Now, we'll get a 404 error when we go to `http://myfirst.dev/users`. This is because we are not declaring our actions the RESTful way.

Edit your `Users_Controller` class and change the line:

```
public function action_index()
```

To this:

```
public function get_index()
```

Your `Users_Controller` class should now look like this:

```
classUsers_Controller extends Base_Controller
{

  public function get_index()
  {
    $users = User::all();

    return View::make('users.index')->with('users', $users);
  }

}
```

Now, when we save and reload the page in our browser it works again! The `get_ prefix` that we added to our `index` method serves much the same purpose as the `action_` prefix that we were using previously.

Unless a method is prefixed appropriately, Laravel will not route URLs to them. In this way, we can ensure that only controller actions are routable and that our web-application's users can't access other methods that may exist in our controllers by simply typing the names of the methods in their browsers.

When an action is prefixed with `get_`, it will only respond to the `GET` requests. An action prefixed with `post_` will only respond to the `POST` requests. The same is true of `put_` and `delete_`. This gives us more code separation and allows us to really improve the readability and maintainability of our application.

Step 11 – Creating a form for adding users

It's time to give our site's administrators the ability to create users. We're going to need a new form. Let's start off by creating the file `application/views/users/create.php` and populating it with the following form HTML:

```
<h1>Create a User</h1>

<form method="POST">
  Real Name: <input type="text" name="real_name"/><br />
  Email: <input type="text" name="email"/><br />
  Password: <input type="password" name="password" /><br />
  <input type="submit" value="Create User"/>
</form>
```

Then, let's create a controller action for it. In our `Users_Controller`, let's add the following action.

```
public function get_create()
{
  return View::make('users.create');
}
```

Now, we can go to `http://myfirst.dev/users/create` and see our form. Once again, it's not pretty but sometimes simple is best.

Step 12 – Routing POST requests to a controller action

When we submit the form it's going to make a `POST` request to our application. We haven't yet created any actions to handle the `POST` requests, so if we submit it now we're going to get a 404 error. Let's go ahead and create a new action in `Users_Controller`:

```
public function post_create()
{
  return "The form has been posted here.";
}
```

Notice that this method has the same action name as the `get_create()` method that we're using to show the create user form. Only the prefix is different. The `get_create()` method responds to the `GET` requests where the `post_create()` method responds to the `POST` requests. In the case of our create user form, the `post_create()` method receives the contents of the form's submitted input fields.

Go ahead and submit the create user form and you'll see the message, **The form has been posted here**.

Step 13 – Receiving form input and saving to the database

Now that we are receiving the data from the form, we can go ahead and create the user account.

Let's update our `post_create()` function in the `Users_Controller` class to add this functionality:

```
public function post_create()
{
  $user = new User;

  $user->real_name = Input::get('real_name');
  $user->email = Input::get('email');
  $user->password = Input::get('password');

  $user->save();

  return Redirect::to ('users');
}
```

Here we're creating a new user record in the same way that we did in our `test` route. The only difference is that we're using Laravel's `Input` class to retrieve the data from the form. Whether the data comes from a `GET` request's query string or a `POST` request's post data, the `Input::get()` method can be used to retrieve the data.

We populate the `User` object with input data. Then we save the new user.

```
return Redirect::to('users');
```

Here's something new. Instead of returning a string or a `View` object, we're using the `Redirect` class to return a `Response` object. Both routed closures and controller actions are expected to return a response. That response could be a string or a `Response` object. When a `View` object is returned it will be rendered as a string. The `Redirect` class' `to()` method specifically tells Laravel to redirect the user to the page specified in its argument. In this example the user will be redirected to `http://myfirst.dev/users`.

We're redirecting the user here so that they can see the updated list of users, which will include the user that they just created. Go ahead and give it a try!

Step 14 – Creating links with the HTML helper

We need a link from the users index view to the create user form as it's currently inaccessible from the user interface. Go ahead and add the link to the file `application/views/users/index.blade.php` with the following line of code:

```
{{ HTML::link('users/create', 'Create a User') }}
```

Laravel's `HTML` class can be used to create a variety of HTML tags. You might be asking yourself why you wouldn't simply write the HTML for the link yourself. One very good reason to use Laravel's HTML helper class is that it provides a unified interface for creating tags that may need to change dynamically. Let's look at an example to clarify this point.

Let's say that we want that link to look like a button and our designer created a sweet CSS class named `btn`. We need to update the call to `HTML::link()` to include the new class attribute:

```
{{ HTML::link('users/create', 'Create a User', array('class' =>
'btn')) }}
```

Actually, we could include any number of attributes to that class and they'd all be handled appropriately. Any attribute assigned to the HTML elements can be updated dynamically by passing a variable to that method instead of declaring it inline.

```
<?php $create_link_attributes = array('class' => 'btn'); ?>

{{ HTML::link('users/create', 'Create a User', $create_link_
attributes) }}
```

Step 15 – Deleting user records with Eloquent

Now that we can add users, we may want to do a bit of cleanup. Let's add a delete action to our `users` controller.

```
public function get_delete($user_id)
{
  $user = User::find($user_id);

  if(is_null($user))
  {
    return Redirect::to('users');
  }

  $user->delete();

  return Redirect::to('users');
}
```

Now, let's step through this.

```
public function get_delete($user_id)
```

This is the first time that we've declared a parameter in a controller action. In order to delete a user, we need to know which user to delete. Since we have used `Route::controller('users')` to have Laravel automatically handle the routing for our controller, it'll know that when we go to the URL `http://myfirst.dev/users/delete/1` it should route to the delete action and pass additional URI segments as arguments to the method.

If you wanted to receive a second argument from a URL (for example, `http://myfirst.dev/users/delete/happy`), you would add a second parameter to your action as follows:

```
public function get_delete($user_id, $emotion)
```

Next, we need to verify that a user with the specified user ID actually exists.

```
$user = User::find($user_id);
```

This line tells Eloquent to find a user with an ID that matches the argument. If a user is found, the `$user` variable will be populated with an object that is an instance of our `User` class. If not, the `$user` variable will contain a null value.

```
if(is_null($user))
{
  return Redirect::to('users');
}
```

Here, we're checking if our user variable has the null value indicating that the requested user was not found. If so, we'll redirect back to the `users` index.

```
$user->delete();
return Redirect::to('users');
```

Next, we delete the user and redirect back to the `users` index.

Of course, our work here won't be finished until we update our `application/views/users/index.php` file to give us links to delete each user. Replace the list item code with the following:

```
<li>{{ $user->real_name }} - {{ $user->email }} - {{
HTML::link('users/delete/'.$user->id, 'Delete') }}</li>
```

Reload the `users` index page and you'll now see the delete link. Click on it and be horrified that we've irreparably removed data from the database. I hope it wasn't anything important!

Step 16 – Updating a user with Eloquent

So, we can add and delete users, but what if we have made a typo and want to fix it? Let's update our `users` controller with the methods that are necessary for us to display our update form and then to retrieve the data from it in order to update the user record:

```
public function get_update($user_id)
{
  $user = User::find($user_id);

  if(is_null($user))
  {
    return Redirect::to('users');
```

```
    }

    return View::make('users.update')->with('user', $user);
  }
```

Here we have our new `get_update()` method. This method accepts a user ID as an argument. Much like we did with the `get_delete()` method, we need to load the user record from the database to verify that it exists. Then, we'll pass that user to the update form.

```
  public function post_update($user_id)
  {
    $user = User::find($user_id);

    if(is_null($user))
    {
      return Redirect::to('users');
    }

    $user->real_name = Input::get('real_name');
    $user->email = Input::get('email');

    if(Input::has('password'))
    {
      $user->password = Input::get('password');
    }

    $user->save();

    return Redirect::to('users');
  }
```

When a user submits our update form, they'll be routed to `post_update()`.

You may have noticed a common theme with methods that receive a user ID as an argument. Whenever we are going to interact with a user model we need to know for sure that the database record exists and that the model is populated. We must always first load it and validate that it is not null.

Afterwards, we assign new values to the `real_name` and `email` attributes. We don't want to just change the user's password every time we submit a change. So, we'll first verify that the password field wasn't left blank. Laravel's `Input` class' `has()` method will return `false`, if an attribute either wasn't sent in the form post or if it's blank. If it's not blank, we can go ahead and update the attribute in the model.

We then save the changes to the user and redirect back to the `users` index page.

Step 17 – Creating the update form with the form helper

Now, we just need to create the `update` form and we'll have a full administrative system!

Go ahead and create the view at `application/views/users/update.blade.php` and fill it with this lovely form:

```
<h1>Update a User</h1>

{{ Form::open() }}

  Real Name: {{ Form::text('real_name', $user->real_name) }}<br />
  Email: {{ Form::text('email', $user->email) }}<br />
  Change Password: {{ Form::password('password') }}<br />

{{ Form::submit('Update User') }}

{{ Form::close() }}
```

This is almost exactly like the create form except that we have mixed things up a little. First of all, you'll notice that we're using Laravel's `Form` class helper methods. These helper methods, like the `HTML` class' helper methods, are not mandatory. However, they are recommended. They offer many of the same advantages as the `HTML` class' helper methods. The `Form` class' helper methods offer a unified interface for generating the resulting HTML tags. It's much easier to programmatically update HTML tag attributes by passing an array as an argument than to loop through and generate the HTML yourself.

```
Real Name: {{ Form::text('real_name', $user->real_name) }}
```

Text fields can be prepopulated by passing in a second argument. In this example, we're passing the `real_name` attribute from the `user` object that we passed from the controller. We then prepopulate the `email` field in the same way.

```
Change Password: {{ Form::password('password') }}
```

Notice that we're not prepopulating the `password` field. It doesn't make sense to do so as we're not storing a readable version of the password in the database. Not only that, to prevent a developer from making a mistake the `Form::password()` method does not have the functionality to prepopulate this field at all.

And with that we have a fully working update user form!

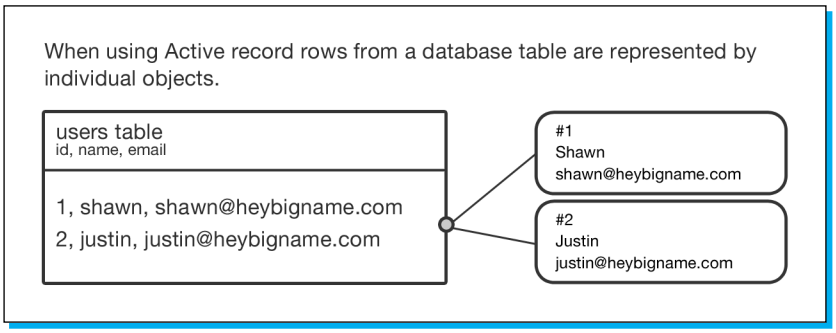
Top 5 features you need to know about

As you start to use Laravel, you will realize that it provides a wide variety of functionality. We've taken the time to describe the five most vital components that we haven't covered in the *Quick start* section. Gaining mastery over these five components gives you the power to make amazing web applications with Laravel.

1 – Eloquent relationships

Eloquent is Laravel's native ActiveRecord implementation. It is built upon Laravel's Fluent Query Builder. Due to the way in which Eloquent operates with Fluent, complex queries and relationships are easy to describe and understand.

ActiveRecord is a design pattern that describes an object-oriented way of interacting with your database. For example, your database's `users` table contains rows and each of these rows represents a single user of your site. Your `User` model is a class that extends the Eloquent Model class. When you query a record from your database, an instantiation of your `User` model class is created and populated with the information from the database.



A distinct advantage of ActiveRecord is that your data and the business logic that is related to the data are housed within the same object. For example, it's typical to store the user's password in your model as a hash, to prevent it from being stored as plaintext. It's also typical to store the method, which creates this password hash within your `User` class.

Another powerful aspect of the ActiveRecord pattern is the ability to define relationships between models. Imagine that you're building a blog site and your users are authors who must be able to post their writings. Using an ActiveRecord implementation, you are able to define the parameters of the relationship. The task of maintaining this relationship is then simplified dramatically. Simple code is the easy code to change. Difficult to understand code is the easy code to break.

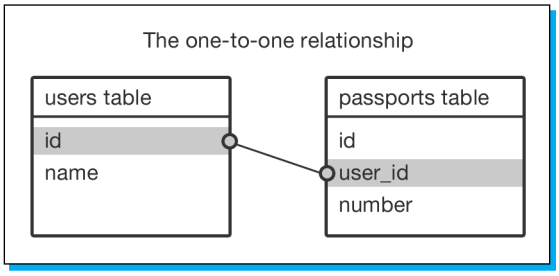
As a PHP developer, you're probably already familiar with the concept of database normalization. If you're not, **normalization** is the process of designing databases so that there is little redundancy in the stored data. For example, you wouldn't want to have both a `users` table which contains the user's name and a table of blog posts which also contains the author's name. Instead, your blog post record would refer to the user using their user ID. In this way we avoid synchronization problems and a lot of extra work!

There are a number of ways in which relationships can be established in normalized database schemas.

One-to-one relationship

When a relationship connects two records in a way that doesn't allow for more records to be related, it is a **one-to-one relationship**. For example, a *user* record might have a one-to-one relationship with a *passport* record. In this example, a *user* record is not permitted to be linked to more than one *passport* record. Similarly, it is not permitted for a *passport* record to relate to more than one user record.

How would the database look? Your `users` table contains information about each user in your database. Your `passports` table contains passport numbers and a link to the user which owns the passport.



In this example, each user has no more than one passport and each passport must have an owner. The `passports` table contains its own `id` column which it uses as a primary key. It also contains the column `user_id`, which contains the ID of the user to whom the passport belongs. Last but not least, the `passports` table contains a column for the passport number.

First, let's model this relationship in the `User` class:

```
class User extends Eloquent
{
  public function passport()
  {
    return $this->has_one('Passport');
  }
}
```

We created a method named `passport()` that returns a relationship. It might seem strange to return relationships at first. But, you'll soon come to love it for the flexibility it offers.

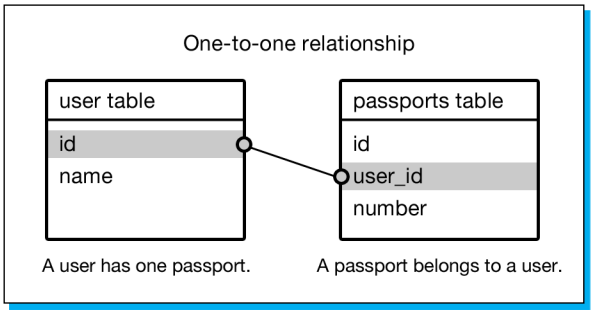
You'll notice that we're using the `has_one()` method and passing the name of the model as a parameter. In this case, a user has one passport. So, the parameter is the name of the passport model class. This is enough information for Eloquent to understand how to acquire the correct passport record for each user.

Now, let's look at the `Passport` class:

```
class Passport extends Eloquent
{
  public function users()
  {
    return $this->belongs_to('User');
  }
}
```

We're defining the passport's relationship differently. In the `User` class, we used the `has_one()` method. In the `Passport` class we used `belongs_to()`.

It's vital to identify the difference early so that understanding the rest of the relationships is more simple. When a database table contains a foreign key, it is said that it belongs to a record in another table. In this example, our `passports` table refers to records in the `users` table through the foreign key `user_id`. Consequently, we would say that a passport belongs to a user. Since this is a one-to-one relationship the user has one (`has_one()`) passport.



Let's say that we want to view the passport number of the user with the `id` of `1`.

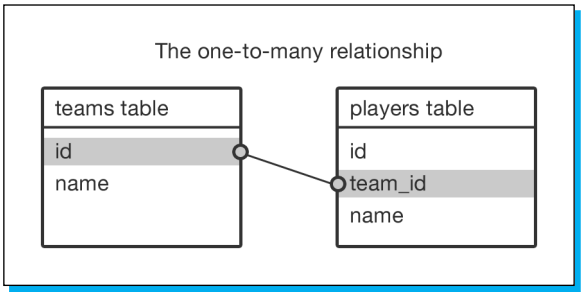
```
$user = User::find(1);

If(is_null($user))
{
  echo "No user found.";
```

```
  return;
}

If($user->passport)
{
  echo "The user's passport number is " . $user->passport->number;
}
else
{
  echo "This user has no passport.";
}
```

In this example, we're dutifully checking to make sure that our `user` object was returned as expected. This is a necessary step that should not be overlooked. Then, we check whether or not the user has a passport record associated with it. If a passport record for this user exists, the related object will be returned. If it doesn't exist, `$user->passport` will return `null`. In the preceding example, we test for the existence of a record and return the appropriate response.

One-to-many relationships

One-to-many relationships are similar to one-to-one relationships. In this relationship type, one model has many of other relationships, which in turn belongs to the former. One example of a one-to-many relationship is a professional sports team's relationship to its players. One team has many players. In this example, each player can only belong to one team. The database tables have the same structure.



Now, let's look at the code which describes this relationship.

```
class Team extends Eloquent
{
  public function players()
  {
    return $this->has_many('Player');
  }
}
```

```
class Player extends Eloquent
{
  public function team()
  {
    return $this->belongs_to('Team');
  }
}
```

This example is almost identical to the one-to-one example. The only difference is that the team's `players()` relationship uses `has_many()` rather than `has_one()`. The `has_one()` relationship returns a model object. The `has_many()` relationship returns an array of model objects.

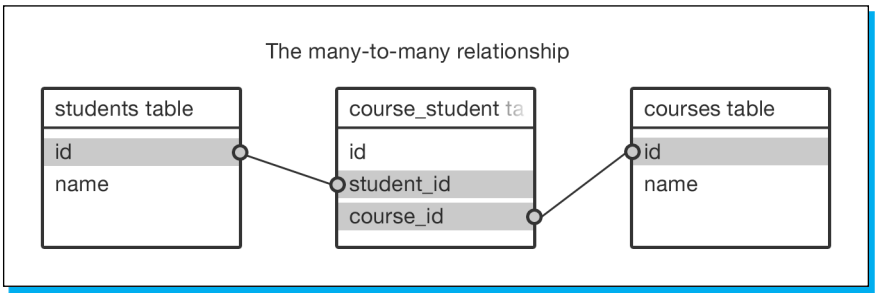
Let's display all of the players on a specific team:

```
$team = Team::find(2);

if(is_null($team))
{
  echo "The team could not be found.";
}

if(!$team->players)
{
  echo "The team has no players.";
}

foreach($team->players as $player)
{
  echo "$player->name is on team $team->name. ";
}
```

Again, we test to make sure that our team could be found. Then, we test to make sure that the team has players. Once we know that for sure, we can loop through those players and echo their names. If we tried to loop through the players without first testing and if the team had players, we'd get an error.

Many-to-many relationships

The last relationship type that we're going to cover is the many-to-many relationship. This relationship is different in that each record from each table could potentially be tied simultaneously to each record in another. We aren't storing foreign keys in either of these tables. Instead, we have a third table that exists solely to store our foreign keys. Let's take a look at the schema.



Here we have a students table and a courses table. A student can be enrolled in many courses and a course can contain many students. The connection between students and courses is stored in a pivot table.

A **pivot table** is a table that exists to connect two tables specifically for many-to-many relationships. Standard convention for naming a pivot table is to combine the names of both of the related tables, singularized, alphabetically ordered, and connected with an underscore. This gives us the table name `course_student`. This convention is not only used by Laravel and it's a good idea to follow the naming conventions covered in this document as strictly as possible as they're widely used in the web-development industry.

It's important to notice that we're not creating a model for the pivot table. Laravel allows us to manage these tables without needing to interact with a model. This is especially nice because it doesn't make sense to model a pivot table with business logic. Only the students and courses are a part of our business. The connection between them is important, but only to the students and to the course. It's not important for its own sake.

Let's define these models, shall we?

```
class Student extends Eloquent
{
  public function courses()
  {
    return $this->has_many_and_belongs_to('Course');
  }
}

class Course extends Eloquent
```

```
{
  public function students()
  {
    return $this->has_many_and_belongs_to('Student');
  }
}
```

We have two models, each with the same type of relationship to each other. `has_many_and_belongs_to` is a long name. But, it's a fairly simple concept. A course has many students. But, it also belongs to (`belongs_to`) student records and vice-versa. In this way, they are considered equal.

Let's look at how we'll interact with these models in practice:

```
$student = Student::find(1);

if(is_null($student))
{
    echo "The student can't be found.";
    exit;
}

if(!$student->courses)
{
    echo "The student $student->name is not enrolled in any
    courses.";
    exit;
}

foreach($student->courses as $course)
{
    echo "The student $student->name is enrolled in the course
    $course->name.";
}
```

Here you can see that we can loop through the courses much the same way we could with the one-to-many relationship. Any time a relationship includes the word *many*, you know that you'll be receiving an array of models. Conversely, let's pull a course and see which students are a part of it.

```
$course = Course::find(1);

if(is_null($course))
{
    echo "The course can't be found.";
    exit;
```

```
}

if(!$course->students)
{
    echo "The course $course->name seems to have no students
    enrolled.";
    exit;
}

foreach($course->students as $student)
{
    echo "The student $student->name is enrolled in the course
    $course->name.";
}
```

The relationship functions exactly the same way from the course side.

Now that we have established this relationship, we can do some fun things with it. Let's look at how we'd enroll a new student into an existing course:

```
$course = Course::find(13);

if(is_null($course))
{
    echo "The course can't be found.";
    exit;
}

$new_student_information = array(
    'name' => 'Danielle'
);

$course->students()->insert($new_student_information);
```

Here we're adding a new student to our course by using the method `insert()`. This method is specific to this relationship type and creates a new student record. It also adds a record to the `course_student` table to link the course and the new student. Very handy!

But, hold on. What's this new syntax?

```
$course->students()->insert($new_student_information);
```

Notice how we're not using `$course->students->insert()`. Our reference to students is a method reference rather than a property reference. That's because Eloquent handles methods that return relationship objects differently from other model methods.

When you access a property of a model that doesn't exist, Eloquent will look to see if you have a function that matches that property's name. For example, if we try to access the property `$course->students`, Eloquent won't be able to find a member variable named `$students`. So it'll look for a function named `students()`. We do have one of those. Eloquent will then receive the relationship object from that method, process it, and return the resulting student records.

If we access a relationship method as a method and not as a property, we directly receive the relationship object back. The relationship's class extends the `Query` class. This means that you can operate on a relationship object in the same way that you can operate on a query object, except that it now has new methods that are specific to the relationship type. The specific implementation details aren't important at this point. It's just important to know that we're calling the `insert()` method on the relationship object returned from `$course->students()`.

Imagine that you have a user model and it has many relationships and belongs to a role model. Roles represent different permission groupings. Example roles might include customer, admin, super admin, and ultra admin.

It's easy to imagine a user form for managing its roles. It would contain a number of checkboxes, one for each potential role. The name of the checkboxes is `role_ids[]` and each value represents the ID of a role in the roles table.

When that form is posted we'll retrieve those values with the `Input::get()` method.

```
$role_ids = Input::get('role_ids');
```

`$role_ids` is now an array that contains the values `1`, `2`, `3`, and `4`.

```
$user->roles()->sync($role_ids);
```

The `sync()` method is specific to this relationship type and is also perfectly suited for our needs. We're telling Eloquent to connect our current `$user` to the roles whose IDs exist within the `$role_ids` array.

Let's look at what's going on here in further detail. `$user->roles()` is returning a `has_many_and_belongs_to relationship` object. We're calling the `sync()` method on that object. Eloquent now looks at the `$role_ids` array and acknowledges it as the authoritative list of roles for this user. It then removes any records that shouldn't exist in the `role_user` pivot table and adds records for any role that should exist in the pivot table.

2– Authentication

Laravel helps you to handle the typical tasks of logging users in and out as well as makes it easy to access the `user` record of the currently authenticated user.

First, we'll want to configure authentication for our site. The authentication configuration file can be found at `application/config/auth.php`.

Here we are presented with a number of configuration options. Primarily, we must choose which `Auth` driver we use. If we choose the Fluent driver, the authentication system will use the table configuration option to find users and will return dumb objects (objects that contain only data) when we request the currently authenticated user. If we use the Eloquent driver, the authentication system will use the model listed in the model option for querying users and Laravel will return an instance of that model when we request the currently authenticated user. Additionally, you can choose which fields Laravel will authenticate against by changing the username and password options. Typically, you'll be using the Eloquent driver.

Let's continue our example from the *Quick start* section. We already have a `User` model, so let's set the driver option to Eloquent. We think that logging in with your e-mail address and password is good, so we'll set the username option to `email` and we'll leave the password option set to `password`. We'll also leave the model set to `User`.

That's it, we're all configured. Let's implement login! First, let's create a new controller for authentication. We'll store it in `application/controllers/auth.php`.

```
<?php

class Auth_Controller extends Base_Controller
{

  public function get_login()
  {
    return View::make('auth.login');
  }

}
```

We'll need to route this and since we typically don't want to go to `http://myfirst.dev/auth/login`, let's manually set up a route. Add this to your `application/routes.php`.

```
Route::any('login', 'auth@login');
```

Finally, create a login form at `application/views/auth/login.blade.php`:

```
{{ Form::open() }}
  Email: {{ Form::text('email', Input::old('email')) }}<br />
  Password: {{ Form::password('password') }}<br />
  {{ Form::submit('Login') }}
{{ Form::close() }}
```

That's it! Now, let's just navigate our browsers to `http://myfirst.dev/login`. We see our nice new login form!

Now, we just need to be able to submit our form. Let's add a new action to our `Auth` controller:

```
public function post_login()
{
  $credentials = array(
    'username' => Input::get('email'),
    'password' => Input::get('password'),
  );

  if(Auth::attempt($credentials))
  {
    return "User has been logged in.";
  }
  else
  {
    return Redirect::back()->with_input();
  }
}
```

With the addition of this method, we now have a functioning login form. Feel free to go ahead and try it out!

Let's look at how we validate a user's e-mail and password. First, we create an array that contains the credentials received from the login form. Notice that we're storing e-mail and password by using the keys `username` and `password`. Despite the fact that we're using e-mail for authentication, Laravel always receives authentication credentials with the keys `username` and `password`. This is because the `username` and `password` fields are configurable in the `auth config` file.

Then, we pass the credentials to the `Auth::attempt()` method. This method takes care of the rest of the process. It'll compare our records in the database against the credentials that we have passed. If the credentials match up, it'll create a cookie in the user's browser and the user will officially be logged in. `Auth::attempt()` returns `true` if a successful login has taken place, and `false` if it failed. If the authentication attempt fails, we redirect the user back to the form and repopulate the `email` field.

Now, let's add `logout` functionality. Add the following method to your `Auth` controller:

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

  return Redirect::to('');
}
```

Finally, add the following line to your `routes.php` file:

```
Route::get('logout', 'auth@logout');
```

That's all! Now, when we go to `http://myfirst.dev/logout`, we'll be logged out and redirected to our site's index page.

So, how can we find out if someone is logged in? The Eloquent and Fluent `Auth` drivers contain two methods for handling this, `check()` and `guest()`. Let's look at each of these in turn:

- `Auth::check()` returns `true` if a user is currently logged in, and `false` otherwise.
- `Auth::guest()` is the opposite of `Auth::check()`. It returns `false` if a user is logged in, and `true` otherwise.

Once you've made sure that a user is logged in, you can use `Auth::user()` to return the user record. If you're using the Fluent driver, `Auth::user()` will return a dumb object containing the appropriate values from the users table. If you're using the Eloquent driver, `Auth::user()` will return an instance of your `User` model. This is very powerful. Let's look at an example:

```
@if(Auth::check())
  <strong>You're logged in as {{ Auth::user()->real_name
  }}</strong>
@endif
```

As you've seen, Laravel's authentication system is driver-based. In this example, we used the Eloquent driver. However, you also have the ability to create custom authentication drivers. This gives you the power to authenticate users with different means and return different types of data with the standard `Auth` class' API. Covering the development of custom drivers is outside the scope of this document. However, it's simple and powerful. Be sure to look into the Laravel documentation for more information.

3 – Filters

Now that we have a user administration site with authentication, we need to restrict some pages on our site to users who have successfully authenticated. We'll do that by using filters.

Filters are functions that can be run before or after routed code. A filter that runs before the routed code is called a **before filter**. Similarly well-named is the **after filter**, which runs after routed code.

Filters are often used for enforcing authentication. We can create a filter that detects if a user is not logged in, then redirect him/her to the login form. Actually, we don't need to make this filter at all. Laravel ships with this filter already written. You can find it in your `application/routes.php`.

Let's take a look:

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

You can see that a filter is registered with the `Route::filter()` method. The typical location to store filter registrations is within your `application/routes.php` file.

The `Route::filter()` method takes two parameters. The first is a string containing the name of the filter. The second is the anonymous function that will be run when the filter is activated.

In this example, the anonymous function will check if a user is logged in using the `Auth::guest()` method. If the user is not logged in, the filter returns a response object that tells Laravel to redirect the user to the login page.

It's important to note that while you can return response objects from before filters, it's not possible to redirect from after filters as at this point it's too late.

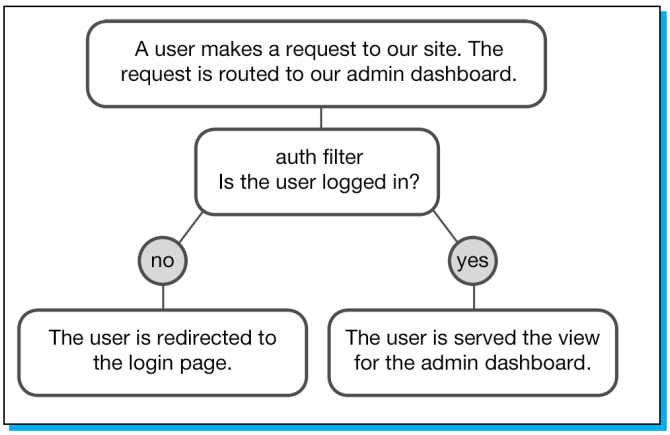
Now that we have the `auth` filter, how do we tell Laravel when to run it? The correct algorithm is situational.

The following is an example of applying a filter to a routed function:

```
Route::get('admin', array('before' => 'auth', function()
{
  return View::make('admin.dashboard');
}));
```

In this example, we want to provide an admin dashboard for users who have successfully authenticated. You may notice that our `Route::get()` declaration has changed. Our first argument is still the route's URI. However, our second argument is no longer an anonymous function, it's now an array. This array provides a method for configuring our route registration. Laravel knows that when you pass a key/value pair, it should be used as configuration and that when you pass an anonymous function, it should be used as the target function for the route.

In this example, we're only using one key/value pair for configuring our route. We use the key `before` to tell Laravel that our route uses a before filter. The value associated with the `before` key is the name of the filter, which should be run before our anonymous function is executed.



Controllers are groups of routable methods that are similar and are therefore uniquely convenient to filter upon. Often the same set of filters that is appropriate for one action in a controller is appropriate for the rest. Filtering at the controller level gives you more flexibility and less redundancy than defining your filters in each route declaration. Let's look at securing our `users` controller.

```
class Users_Controller extends Base_Controller
{

  public function __construct()
  {
    parent::__construct();

    $this->filter('before', 'auth');
  }
```

Here we're looking only at the top-most section of our `users` controller. The rest of the controller is identical to what we created in the *Quick start* section.

As you may have noticed, we've declared a constructor for our `Users_Controller` class. A constructor is a method that is run immediately once our class is instantiated as an object. We use a controller class' constructor to define filters for that controller's actions. It's also important to notice that we first call the `parent::__construct()` method. It is important for Laravel's `Controller` class to have its constructor executed so that it can initialize itself and be ready for action.

Then, we tell our controller that for every request to one of its actions we want to run the before filter `auth`. Now, this controller is completely protected behind your authentication implementation. You will be unable to access the actions within this controller until you've successfully logged in. If you try to access one of the controller's actions, you will be redirected to the login page.

Now, thanks to a combination of Laravel's `Auth` class and its `auth` filter, you now have a properly secured admin site.

Unfortunately, a complete description of Laravel's filter functionality is outside the scope of this book. Thankfully, Laravel's documentation is a great resource for learning more about what you can do with filters.

4 – Validation

Laravel provides a `Validator` class full of functionality to help with validating forms, database models, or anything that you'd like. The `Validator` class allows you to pass any input, declare your own rules, and define your own custom validation messages.

Let's take a look at an example implementation for our create users actions.

```
public function post_create()
{
  // validate input
  $rules = array(
    'real_name' => 'required|max:50',
    'email'     => 'required|email|unique:users',
    'password'  => 'required|min:5',
  );

  $validation = Validator::make(Input::all(), $rules);

  if($validation->fails())
  {
    return Redirect::back()->with_input()
    ->with_errors($validation);
  }

  // create a user
  $user = new User;

  $user->real_name = Input::get('real_name');
  $user->email = Input::get('email');
  $user->password = Input::get('password');
```

```
        $user->save();

        return Redirect::to_action('users@index');
    }
```

First, we create an array that defines our validation rules. Validation rule arrays are key/value pairs. Each key represents the field name that it will be validating and each value is a string that contains the validation rules and their configurations. Validation rules are separated by the pipe character (`|`) and a validation rule's configuration parameters are separated from the name of the rule by a colon (`:`).

The required rule ensures that input has been received for its fields. The `max` and `min` rules can ensure that strings are no longer or shorter than a specific length. The length for `min` and `max` are passed as parameters and are therefore separated from the rule name with a colon. In the `real_name` example, we're ensuring that it's no longer than 50 characters. We also want to ensure that the user's password is no less than five characters in length.

Since we're using e-mail for authentication, we should make sure that it's a unique address in our database. So, for our `email` field, we define the `unique` rule and tell it to compare it against other values in the `users` database table. If it finds another e-mail address that matches the address from our create users form, it will return an error.

Next, we create the validation object by using the `Validator::make()` method. We are providing the form's input as our first argument and our `$rules` array as the second.

We can now check to see if the validation passes by using the `$validation->passes()` method or if the validation fails by using the `$validation->fails()` method.

In this example, if our validation fails, we redirect the user back to the form with the input data for repopulating the form as well as the error data from our `$validation` object. With this error data, we can populate our form with errors so that our user knows why our form didn't validate.

View objects have a special `$errors` variable that is typically empty. When a user is redirected back to another action `with_errors($validation)`, the special `$errors` variable is populated with the errors from the validation object. Let's look at an example of how we can display an error for the `email` field:

```
{{ $errors->first('email', '<span class="help-inline">:message</
span>') }}
```

Here we're displaying the error message from the first validation rule that didn't pass on the `email` field. Our second argument is a formatting string. The error message will replace the `:message` symbol in the string. If the `email` field has no validation errors, none of the formatted string will be returned. This makes this algorithm ideal for creating forms with individual feedback per field.

A complete list of validation rules can be found in the Laravel documentation in your project at `http://myfirst.dev/docs`.

In a typical web application, validation occurs on both forms and data models. Form validation ensures that the data retrieved from the user meets certain criteria. Data model validation ensures that the data that is being inserted into the database is adequate, relationships are maintained, field uniqueness is maintained, and so on.

Our validation example functions. But, for brevity, we coded it directly inside of our controller. A more appropriate place to store form validation rules is within a model specific to that form. Likewise, a more appropriate place to store database model validation rules is within an Eloquent model.

5 – Bundles

A major selling point for the Laravel framework is the way in which it handles modular code. Any functionality that can be written within Laravel can be bundled. Controllers, models, views, libraries, filters, config files, routing, and migrations can all be packaged as a bundle and either re-used by you and your team or distributed to be used by others. It may excite you to know that Laravel's application folder is considered to be its default bundle. That's right, all web-application code written with Laravel runs within a bundle.

As a result of bundles being a first-class citizen within Laravel, they are useful for a wide variety of applications. Bundles can be used to add something as simple as a vendor library or helper functions. Bundles are also often used to package up entire web-app subsystems. For example, it's very reasonable to be able to drop a blog bundle into your application, run migrations to create your blog database tables, and then have the URLs `http://myfirst.dev/blog` and `http://myfirst.dev/admin/blog` automatically start working. Bundles are amazingly powerful.

Your application's bundle configuration can be found in the `bundles.php` file that sits in the root directory of your Laravel installation. Let's look at our web applications' `bundles.php` file right now:

```
return array(

  'docs' => array('handles' => 'docs'),

);
```

Oh look here, we already have a bundle installed. The docs bundle contains the current version of the Laravel documentation. It's because the docs bundle handles the docs route that you can go to `http://myfirst.dev/docs` and view the Laravel documentation. You can comment out or remove the line that configures the docs bundle to prevent your users from accessing the `docs` route on your production site.

Laravel has a public online bundle repository, which can be found at `http://bundles.laravel.com`. Users are free to create and add their own bundles to this repository. We can then install their bundles into our own application.

There are a few ways in which you can install bundles.

Installing bundles with Artisan is generally the preferred method to install bundles from the repository. Simply use Artisan's `bundle:install` task, and the bundle that you request will be downloaded from the Laravel bundle repository and installed into your bundles directory. Let's give this a shot.

```
# php artisan bundle:install swiftmailer
Fetching [swiftmailer]...done! Bundle installed.
```

We told Artisan that we wanted to install the `swiftmailer` bundle from the bundle repository. It downloaded the bundle and now we have a directory `bundles/swiftmailer`, which contains the `swiftmailer` vendor library as well as the bundle's `start.php` file. `start.php` is the file responsible for loading the contents of the bundle and making it ready to be used; it is run when the bundle is first started.

You could accomplish the same without Artisan. The bundle repository functions by using GitHub. Therefore, all bundles in the repository can be found on GitHub. You could easily go to `https://github.com/taylorotwell/swiftmailer` and download the code into your bundles directory. This accomplishes the same as doing it with Artisan with only a little more elbow-grease required.

Once you've installed your bundle, you must add it to the bundle configuration file before it can be used. Let's add configuration for our `swiftmailer` bundle to our `bundles.php` file and look at the results.

```
return array(

  'docs' => array('handles' => 'docs'),
  'swiftmailer',

);
```

It is unnecessary for us to add any additional configuration parameters in order to make our `swiftmailer` bundle work. Within our code we would simply run `Bundle::start('swiftmailer')`, and proceed to use it. Alternatively, if you wish to automatically start a bundle, you can simply add the autoconfiguration to your `bundles.php` file.

```
return array(

  'docs' => array('handles' => 'docs'),
  'swiftmailer' => array('auto' => true),

);
```

Now, it's entirely unnecessary to start the bundle manually. It will be started automatically before your routed code is executed.

The core purpose behind the existence of bundles is code re-use. On the road to mastering Laravel, we recommend implementing new code first without bundles. Then, once you find a need to re-use that code, you may then prefer to bundle the code up and refactor it as necessary. This prevents you from being slowed down by both learning how bundles are put together and by implementing code for the first time in Laravel.

After you have gained the experience of making a few bundles, you'll find that they're very easy to design. Until you gain this experience, you may find that you're burning through precious development hours refactoring your code.

You're now aware of all of the most fundamental components of developing with Laravel. As you continue gaining mastery, you'll discover more advanced features such as the Inversion of Control container, view composers, events, and much more. Laravel provides a unique platform in the world of PHP. You're given the chance to implement your own software architecture design without having to destructively modify the core to support it. We highly recommend that if you continue your education in design patterns as you are now working within a platform that truly supports the implementation of your own unique architectures.

People and places you should get to know

If you need help with Laravel, here are some people and places which will prove invaluable.

- **Official Homepage**: `http://laravel.com`
- **Official documentation**: `http://laravel.com/docs`
- **Official API documentation**: `http://laravel.com/api`
- **GitHub repository**: `https://github.com/laravel/laravel`

Articles and tutorials

There are a number of people writing and recording tutorials and screencast series on using Laravel. Here are a few that will help you improve your game:

- *nettuts* provides a number of Laravel tutorials in both their free and paid sections. They're well-known for the quality of their presentations (`http://net.tutsplus.com/tag/laravel/`).
- Laravel: Ins and Outs is a study group that has recently been started by the Laravel community containing valuable information that can't be found anywhere else. Join us at `http://laravel.io` and follow us at `http://twitter.com/laravelio`.
- Jason Lewis, the author of *Feather Forums* and a long-time contributor to Laravel has created a good series of Laravel tutorials that includes a how-to guide for contributing to a GitHub project (`http://jasonlewis.me/blog/laravel-tutorials`).
- Matthew Machuga, a well-respected multi-disciplined developer, has some one-of-a-kind Laravel screencasts, which highlight test-driven development with Laravel (`http://matthewmachuga.com/screencasts`).
- Dayle Rees has released a popular set of tutorials that cover many of Laravel's basics (`http://daylerees.com/category/laravel-tutorials/`).
- And finally, my own screencast series contains walkthroughs of Laravel's folder structure, explanations of security best-practices, and information about modeling forms (`http://heybigname.com/2012/03/12/a-walk-through-laravel-folder-structure/`).

Community

Laravel has a fantastic community. Professional developers with years of experience contribute to the forums and offer their time to help others in the IRC channel. They're both great places to build familiarity with Laravel and great places to go to if you get stuck.

An important part of being a software development professional is exposing yourself to as many good solutions to as many problems as possible. The only way that this can be reasonably accomplished is by joining a community. By regularly reading forums and participating in an IRC channel, you'll be exposed to many new ideas than you could think of on your own.

- *Laravel Forums*: `http://forums.laravel.com`
- *Laravel IRC* (live chat): `http://laravel.com/irc`

Twitter

Twitter is a great way to keep up with Laravel news—word travels fast over the wire. Here are a few accounts that you'll want to follow.

- *@taylorotwell*: He's the one responsible for Laravel and is a major player in pushing PHP forward as a serious development platform
- *@laravelphp*: The official Twitter account for Laravel
- *@laravelnews*: Catch the retweets of news about all aspects of Laravel from users around the world

Thank you for buying

Laravel Starter

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

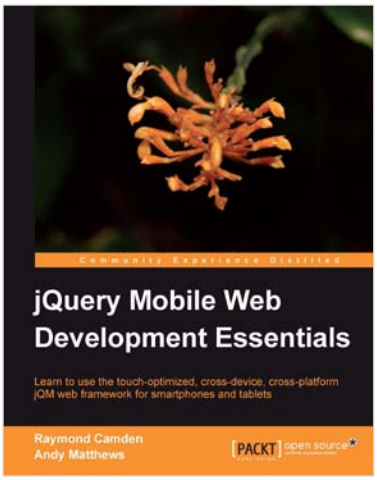
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



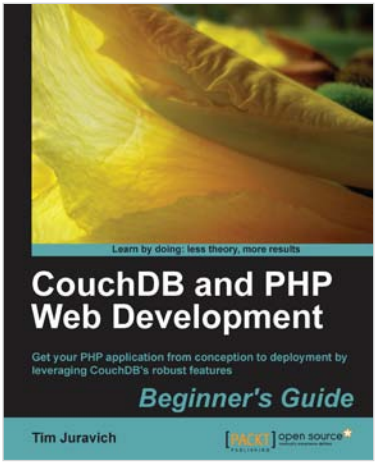
jQuery Mobile Web Development Essentials

ISBN: 978-1-84951-726-3 Paperback: 246 pages

Learn to use the touch-optimised, cross-device, cross-platform jQM web framework for smartphones and tablets

1. Create websites that work beautifully on a wide range of mobile devices with jQuery mobile
2. Learn to prepare your jQuery mobile project by learning through three sample applications
3. Packed with easy to follow examples and clear explanations of how to easily build mobile-optimized websites

---



CouchDB and PHP Web Development Beginner's Guide

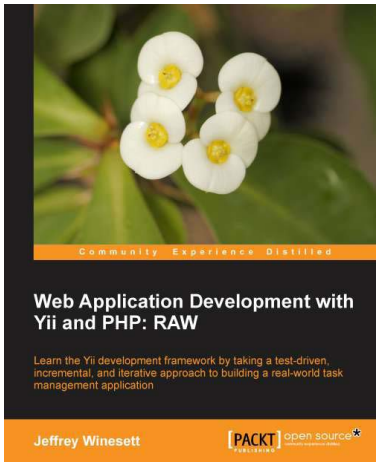
ISBN: 978-1-84951-358-6 Paperback: 304 pages

Get your PHP application from conception to development by leveraging CouchDB's robust features

1. Build and deploy a flexible Social Networking application using PHP and leveraging key features of CouchDB to do the heavy lifting
2. Explore the features and functionality of CouchDB, by taking a deep look into Documents, Views, Replication, and much more.
3. Conceptualize a lightweight PHP framework from scratch and write code that can easily port to other frameworks

Please check www.PacktPub.com for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING



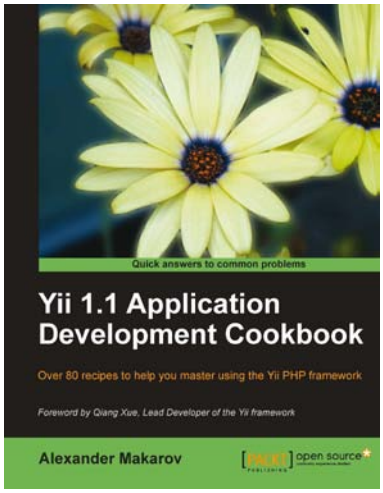
Web Application Development with Yii and PHP

ISBN: 978-1-84951-872-7 Paperback: 350 pages

Learn the Yii development framework by taking a test-driven, incremental, and iterative approach to building a real-world task management application

1. A step-by-step guide to creating a modern Web application using PHP, MySQL, and Yii
2. Build a real-world, user-based, database-driven project task management application using the Yii development framework
3. Start with a general idea, and finish with deploying to production, learning everything about Yii inbetween, from "A"ctive record to "Z"ii component library

---



Yii 1.1 Application Development Cookbook

ISBN: 978-1-84951-548-1 Paperback: 392 pages

Over 80 recipes to help you master using the Yii PHP framework

1. Learn to use Yii more efficiently through plentiful Yii recipes on diverse topics
2. Make the most efficient use of your controller and views and reuse them
3. Automate error tracking and understand the Yii log and stack trace

Please check www.PacktPub.com for information on our titles