Laravel: My First Framework

by Maksim Surguy

Laravel - my first framework

Companion for developers discovering Laravel PHP framework

Maksim Surguy

This book is for sale at http://leanpub.com/laravel-first-framework

This version was published on 2014-09-05

Tweet This Book!

Please help Maksim Surguy by spreading the word about this book on Twitter!

The suggested hashtag for this book is #laravelfirst.

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

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

Also By Maksim Surguy

Integrating Front end Components with Web Applications

Contents

Introduction

Over the years PHP has grown up into a powerful and popular language for web applications. Without an underlying framework PHP-powered web applications can become messy and nearly impossible to extend or maintain. A framework that has session management, ORM, built-in authentication, and templating saves developers' time by providing the components that are necesary in most web applications. Open source PHP framework called Laravel provides all these components and many more, making you a very efficient developer capable of building better applications a whole lot faster. Laravel allows you to focus on the code that matters instead of constantly reinventing the wheel.

Using well-designed diagrams and source code, this book will serve a great introduction to Laravel PHP framework. It will guide you through usage of Laravel's major components to create powerful web applications. The book begins with an overview of a Laravel-powered web application and its parts, introduces you to concepts about routing, responses and views, then moves onto understanding controllers, dependency injection and database operations. You'll explore Laravel's powerful ORM called Eloquent, simple templating language called Blade and session management. You'll also learn how to implement authentication and protect your applications, learn how to build your own API's applying the concepts learned in the book.

About the author

My name is Maksim Surguy. I am a full time web developer, part time writer and former breakdancer[1]. If you use Laravel PHP framework or Bootstrap, you might have seen some of the websites I created with Laravel:

- Bootsnipp[2]
- Laravel-tricks, open source[3]
- Built With Laravel[4]
- Bookpag.es[5]
- Panopanda[6]
- MyMapList[7]
- Cheatsheetr[8]

I love creating new products and in the process I try to share as much as I possibly can. You can read free web development tutorials on my blog at http://maxoffsky.com, my other book (http://maxoffsky.com/frontend), and you can follow me on Twitter for various web development tips and tricks at http://twitter.com/msurguy

[1] https://www.youtube.com/watch?v=wEF_RHL1NFU
[2] http://bootsnipp.com
[3] http://laravel-tricks.com
[4] http://builtwithlaravel.com
[5] http://bookpag.es
[6] http://panopanda.co
[7] http://mymaplist.com
[8] http://cheatsheetr.com

Prerequisites

In order to use the book correctly, the reader should:

- Have some experience with command line (like creating and deleting files and folders, listing all files in a directory)
- Have heard of MVC but not necessarily have much experience using it
- Have basic understanding of OOP PHP
- Know how to create a basic database with PHPMyAdmin, other GUI tools or with a command line

If you don't have some of the skills necessary to keep up with the book's material, please feel free to ask me[9] for pointers to some great resources.

Source Code

The source code for this book is available for each chapter and is located on Github at https://github.com/msurguy/laravel-first-framework[10]. Feel free to explore it, comment on it and improve it on Github.

[9] http://twitter.com/msurguy
[10] https://github.com/msurguy/laravel-first-framework

1. Meeting Laravel

This chapter covers:

- An introduction to Laravel 4 PHP framework
- History of Laravel framework
- Advantages of using Laravel

Developing a web application from scratch can be a tedious and complicated task. Over time, too many opportunities for bugs and too many revisions can make maintenance of a web application very frustrating for any developer. Because of this, a recently developed framework written in PHP, Laravel, has quickly won the hearts of developers around the world for its clean code and ease of use.

Laravel is a free PHP framework that transforms creation of web applications into a fun and enjoyable process by providing expressive and eloquent code syntax that you can use to build fast, stable, easy to maintain and easy to extend web applications. Laravel consists of many components commonly needed in the process of building web applications. PHP developers use Laravel for a broad range of web applications, from simple blogs and CMSs to online stores and large-scale business applications. It's no wonder Laravel has become a strong contender in the family of package-based web frameworks and is ready for prime time after only a few years of development.

In this chapter you will meet Laravel PHP framework, you will learn about its short but interesting history and its evolution over its lifetime, how by using established conventions and approaches Laravel can speed up development significantly and find out why you, too, will find Laravel a great choice for your applications. Are you ready? Let's meet Laravel!

1.1 Introducing Laravel 4 PHP framework

Laravel 4 is an open source PHP web application framework created by Taylor Otwell and maintained by him and a community of core developers. For over two and a half years web developers around the world have been using Laravel (version 1 to version 4) to build stable and maintainable web applications by using Laravel's powerful features such as built-in authentication, session management, database wrapper and more. Laravel is considered to be a full stack web development framework, meaning it can handle every aspect of a web application architecture – from storing and managing data using the database wrapper to displaying user interfaces using its own templating engine. Majority of the mundane work that a web developer encounters in the process of creating a web application has been thought of and alleviated by Laravel's components. Because of this, an application can be built a lot faster than it would be if the developer made the whole application from scratch.

What's a PHP Web application framework?

A PHP Web application framework is a set of classes, libraries or components written in PHP server-side scripting language that aim to solve common web development problems and promote code reuse. Such components as authentication, session management, caching, routing, database operations wrappers and more are included in Laravel and most popular PHP web application frameworks. Using a web application framework allows developers to save significant amount of time developing a web application.

While being a great time saver, Laravel isn't a magic cure for all your web development problems. Although it provides a solid foundation for the backend of your application, you still need to come up with the application logic, database structure and the HTML, CSS and Javascript for you application. Also, Laravel (and PHP scripting language in general) isn't well suited for real time applications like web chats, video streaming, real time gaming, and instant messaging so you would have to use something else for those kinds of applications.

The architecture of Laravel makes it fitting for building many kinds of web applications. From simple blogs, guestbooks, and single-purpose websites to complex CMSs, forums, APIs, e-Commerce websites and even social networks, Laravel will help you get web application development done faster.

It's not just the features that make Laravel so special. There are many reasons to use Laravel in your next web development project. Web developers choose Laravel for the expressive and clean code that it provides as a basis for web applications. The expressiveness is what distinguishes Laravel from other similar PHP frameworks. The application structure that Laravel provides makes it easy to start working even for a beginner developer that has never used a framework before. The community around Laravel, including the creator of the framework himself, has been very welcoming and supportive. Let's explore these reasons a bit more below. How does Laravel's expressive code help you, the developer, to keep the code of your application maintainable, easy to change and easy to read?

1.1.1 Laravel's Expressive code

The most prominent candidates for code re-use in web development with PHP are managing sessions, authentication and operating with data stored in databases. Laravel has tackled those areas quite gracefully and offers beautiful syntax to help you develop applications. Listing 1.1 shows a handful of examples of Laravel's expressive syntax:

Listing 1.1 Examples of Laravel's expressive syntax

```
$order = array('customer' => 'Prince Caspian', 'product' => 'Laravel: my first framework');
// Store the order details in the session under name "order"
Session::put('order',$order);
...
// Retrieve the order details from the session using the session element called "order"
// and create a new record in the database
$orderInDB = Order::create(Session::get('order'));

// Remove the session element "order"
Session::forget('order');

// Retrieve all orders for customer "Prince Caspian"
$ordersForCaspian = Order::where('customer', 'Prince Caspian')->get();
...
// Count all orders of "Laravel: my first framework"
$ordersQuantity = Order::where('product', 'Laravel: my first framework')->count();
```

Laravel promotes usage of clean and expressive code throughout your web application while it still remains PHP code. All of Laravel's method names are logical, sometimes to an extent that you can even guess them without looking up the documentation. For example to detect if there is an item called 'order'

in the session already you would use `Session::has('order')` which might come naturally to your mind. The same could be said about the sensible defaults for all methods that need to return some value: session items form input elements, cache items, and more. As another minimal example, you could return a default value for an input submitted with a form if nothing was provided by the user: `Input::get('name', 'Guest Customer')`.

These conventions might not immediately appeal to every developer, but in the long run they make the framework consistent and unified. Laravel makes many assumptions that accelerate web development tremendously. While developing applications with Laravel you will enjoy those moments when the framework appears to know exactly what you intend to do. Aside from the expressive and clean code, Laravel makes your applications easier to maintain by adopting separation of the application code into files that work with the data, files that control the flow of the application and files that deal with the output that the end user will see.

1.1.2 Laravel applications use Model-View-Controller pattern

When you install Laravel you will notice that inside of the "app" folder there are folders called "models", "views", "controllers" amongst other folders. This is because Laravel is a framework for MVC applications. Such applications use the Model-View-Controller software architecture pattern, which enforces separation between the information (model), user's interaction with information (controller) and visual representation of information (view). As you develop your Laravel applications you will mainly work with files inside of those three folders. Don't worry if you are not familiar with MVC pattern yet, Laravel is a great framework to learn MVC and there is a section of this book that will give you an introduction to this software pattern and its usage in Laravel applications.

Consider having your whole web application in one PHP script. Mixing SQL statements, business logic and output all inside one file will be very confusing for any developer to work on. Especially if it has been quite some time since you have looked at the code of such application. As the application grows it will become harder or impossible to maintain. The main benefit of using MVC architecture in Laravel applications is that it helps you have the code of your application separated and organized in many small parts that are easier to change and extend. You will come to appreciate this separation when you will want to add a new feature into your application or when you will need to quickly modify existing functionality. Laravel has been around for more than two years. Even though that doesn't seem like a lot, it has matured and has been road tested by thousands of developers worldwide. The developers that use Laravel communicate to each other through many different online and offline channels, sharing tips, code examples, and helping maintain the framework.

1.1.3 Laravel was built by a great community

Contrary to popular belief, Laravel is not one-man's product. While Taylor Otwell is the main developer behind the framework – the visionary and the person in charge of saying "yes" or "no" to features, the open source nature of Laravel made it possible to get code contributions and bug fixes from over a hundred developers worldwide, Laravel's public forums (with over 20,000 registered users) and a thriving IRC channel helped steer the development and testing of the framework. Laravel's vibrant and welcoming community at http://laravel.com is something that makes Laravel special and distinctive from other similar projects.

Laravel would not be in the place where it is now if not for contributions, ideas, suggestions and passion of so many developers and lovers of clean code all around the world. In fact, many Laravel features were inspired by other open source projects like Sinatra, Ruby on Rails, CodeIgniter and others. How did Laravel come such a long way in just over two years? Let's explore Laravel's history and see how it evolved over time.

1.2 History of Laravel framework

Rome wasn't built in a day, and so weren't any of the established frameworks. Usually building a solid web development framework that is well tested and ready for production deployment takes a long time. Though interestingly, in case with Laravel its development was progressing a bit differently and at a faster pace than other frameworks.

1.2.1 State of PHP frameworks world before Laravel 4

In August of 2009 PHP 5.3 was released. It featured introduction of namespaces and anonymous functions called closures amongst other updates. The new features allowed developers write better Object-Oriented PHP applications. Even though it provided many benefits and a way to get to a brighter development future, not all frameworks were looking into that future but instead focused on supporting the older versions of PHP. The framework landscape mainly consisted of Symfony, Zend, Slim micro framework, Kohana, Lithium and CodeIgniter.

CodeIgniter was probably the most well known PHP framework at the time. Developers preferred it for its comprehensive documentation and simplicity. Any PHP programmer could quickly start making applications with it. It had large community and great support from its creators. But back in 2011 CodeIgniter was lacking some functionality that Taylor Otwell, the creator of Laravel, considered to be essential in building web applications. For example out of the box authentication (logging users in and out) and closure routing were absent in CodeIgniter. Therefore, Laravel version 1 beta was released on June 9, 2011 to fill in the missing functionality. According to Laravel's creator, Taylor Otwell, Laravel version 1 was released in June 2011 simply to solve the growing pains of using CodeIgniter PHP framework.

1.2.2 Evolution of Laravel framework

Starting with the first release, Laravel featured built-in authentication, Eloquent ORM for database operations, localization, models and relationships, simple routing mechanism, caching, sessions, views, extendibility through modules and libraries, form and HTML helpers and more. Even on the very first release the framework already had some impressive functionality. Laravel went from version 1 to version 2 in less than six months.

The second major release of the framework got some solid upgrades from the creator and the community. Such features were implemented: controller support, "Blade" templating engine, usage of inversion of control container, replacement of modules with "bundles" and more. With the additions of controllers, the framework became a fully qualified MVC framework. Less than two months later new major point release was announced, Laravel 3.

The third release was focused on unit test integration, the Artisan command line interface, database migrations, events, extended session drivers and database drivers, integration for "bundles" and more. Laravel 3 was quickly catching up to the big boys of PHP frameworks such as CodeIgniter and Kohana, a lot of developers started switching from other frameworks to Laravel for its power and expressiveness. Laravel 3 stayed in a stable release for quite some time but about 5 months after it was released the creator of the framework decided to re-write the whole framework from scratch as a set of packages distributed through "Composer" PHP dependency manager. Thus Laravel 4 codenamed "Illuminate" was in the works.

Laravel 4 was re-written from the ground up as a collection of components (or packages) that integrate with each other to make up a framework. The management of these components is done through the best PHP dependency manager available called "Composer". Laravel 4 has an extended set of features that no other

version of Laravel (and even no other PHP framework) has had before: database seeding, message queues, built in mailer, even more powerful Eloquent ORM featuring scopes, soft deletes and more.

Embracing the new features of PHP 5.3 Laravel has come a long way in just over two years appealing to more and more developers worldwide. The visionary behind the framework – Taylor Otwell and the community surrounding Laravel have made enormous progress creating a future-friendly established architecture for PHP web applications in a very short period of time. Holding this book testifies on Laravel's success and constantly growing community of users and contributors means that Laravel is here to stay. But what makes Laravel so beneficial for developers, and why should you care to use it in your projects instead of hand made code or instead of other available frameworks?

1.3 Advantages of Using Laravel

There are many advantages to using Laravel instead of hand-made code or even instead of other similar frameworks. Laravel is a winner because it incorporates established and proven web development patterns: convention over configuration so it is ready for use out of the box, Model-View-Controller (MVC) application structure and Active Record powering its database wrapper. By using these conventions and patterns Laravel helps you as a developer to build a maintainable web application with easy to understand code separation and in a short time frame.

Without a framework like Laravel maintenance or adding new features to a web application could become a nightmare. Lack of structure will create problems when another developer will want to fix bugs or extend the application. Absence of standards will make it hard to understand application's functionality even for most experienced developer. Unconventional method and function names might make the code of the application unreadable and low quality. Laravel helps you avoid all of these problems. Using commonly accepted conventions and established software development patterns Laravel helps your applications stay alive and relevant. Over the next few pages I will take you on a tour exploring the advantages of using Laravel framework over hand made code or other frameworks. First destination: "Convention over configuration".

1.3.1 Convention over configuration

Imagine if you could do a fresh install of the framework and in the next five minutes already be working on how your application behaves when a user visits a certain page and not worry about configuring the things that are common to most applications? Laravel makes many assumptions that make this kind of scenario possible, this is known as "convention over configuration" pattern. When you start writing a new web application you would like to have as minimal initial set up as possible. Such configuration settings as the location where the application will save your sessions and cache, what database provider and what tables in the database it will use, locale of the application and more settings are usually needed for web applications. Thankfully all these configurations are already pre-set for you in a fresh installation of Laravel, but, if needed, any changes to these configurations could be easily made.

Convention over configuration

Convention over configuration is a software design pattern that minimizes the number of decisions that a developer needs to make by implying reasonable configuration defaults (conventions) requiring the developer to provide the configuration for only those parts that deviate from the conventions.

The configuration of new Laravel applications is very minimal and it is all separated for you inside of

one folder: "app/config". You need to only change the settings that are different from pre-set conventions. For example provide a database name if you are using one, your time zone, and you are good to go. If you are not using a database or don't care about time zone settings, then you don't even have to do any configuration and can start coding your application right away! All the other settings are set to reasonable defaults and you need to change them only if your application departs from pre-set conventions. We will explore Laravel's configuration more in detail in the next chapter. This smart way of minimizing the amount of decisions that a developer has to make to start developing an application makes Laravel very easy to get started with and appealing to developers of all proficiency levels.

1.3.2 Ready out of the box

With some frameworks it could take hours just to get to a point where the developer can start using the framework. Laravel is unique in this sense because you can install and start using Laravel in just few minutes. When you install Laravel for the first time you will notice that you don't need to do anything else to have your application operational. After starting a server using built in command you can right away visit the root URL of the application and get a response from Laravel signifying that your Laravel application is ready (figure 1.1):

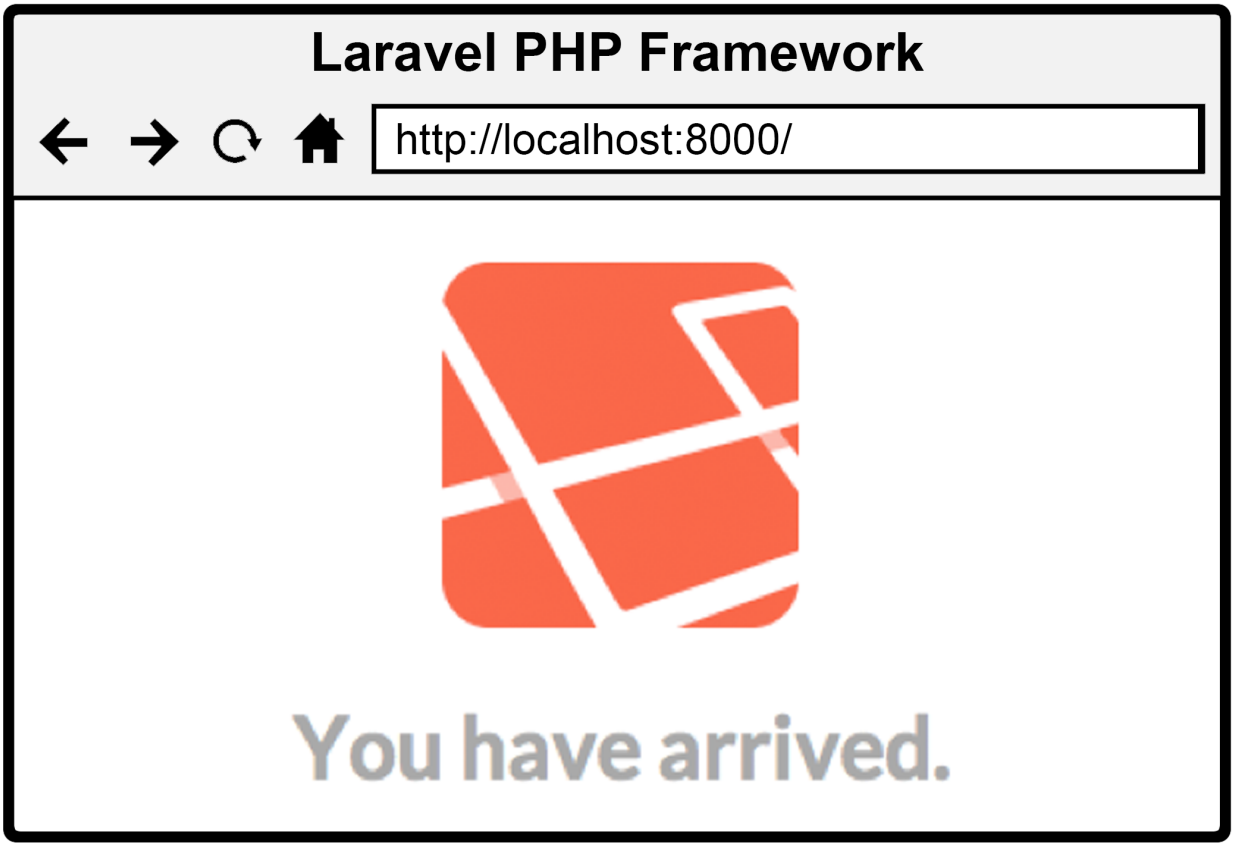

Figure 1.1 Laravel is operational out of the box with zero configuration

Amazingly when this page shows up Laravel already assumes that you are developing locally, it knows

where to save the cached pages, it knows that going to the application's root URL is supposed to display a page with some pre-set content in it. This is great because you can get to the development of your application's logic, models and responses in no time. Not only your application is ready for your coding attention, it already has some great structure to it.

1.3.3 Clear organization of all parts of the application

Everything has its own place in a Laravel application. Because applications built with Laravel follow MVC pattern, the data models live in `app/models` folder, the views reside in `app/views` folder. Can you guess where the controllers are? You got it right, in `app/controllers` folder! This separation makes your code cleaner and easier to find. For example if you have a database table called "orders" you will most likely have a model for it called "order" in your models folder. If you have created a nice login screen for your application it will be somewhere in the views folder. Take a look at figure 1.2 to see which folders your Laravel applications will consist of.

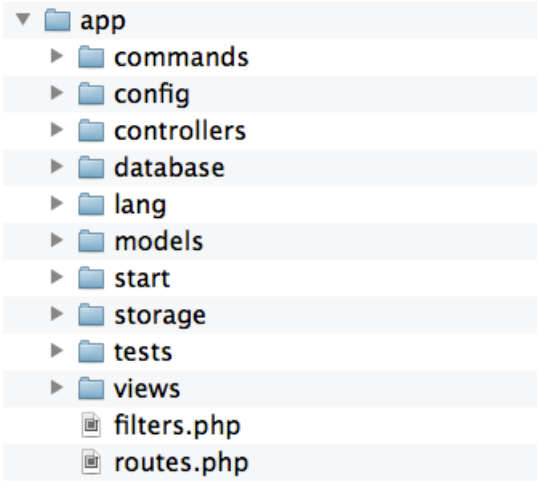

Figure 1.2 Folder structure of a Laravel application

There are a few more folders other than models, views and controllers but don't be afraid, there will be a whole section of the next chapter devoted to all of these folders because together they are the heart and mind of your applications (while Laravel itself is the soul). Each file and folder has its purpose but by conventions they are already pre-filled with something so you can get started quicker and change only the parts that need to break out of the conventions. There are many things that made Laravel so successful. Clean organization is certainly one of those things. Another one is built-in Authentication.

1.3.4 Built-in Authentication

An important feature that has been missing in some other big name frameworks is built-in authentication for web applications.

Authentication

Authentication is the process of identifying a user, usually by some unique user attribute (username or email) and accompanying password, in order to grant access to some specific information.

Since its first release, Laravel featured authentication methods to log users in and log them out and also making it easy for a developer to access properties of the currently logged-in user. Laravel 4 only continues the trend of authentication being pre-baked into the framework and greatly simplifies the process of authentication. In listing 1.2 we will explore some authentication methods that Laravel provides. By default authentication uses a table called "users" in your database and the user model is already in the app/models folder. Please note that the following code is just to demonstrate the simplicity of authentication methods so it is not a part of any particular file (we will get to that in detail later):

Listing 1.2 Demonstration of Laravel's built-in authentication methods

```
// Provide username and password to check against (could be from input)
$user = array('username' => 'caspian356', 'password' => 'P@s5w0rD');

// Check user's credentials against user's details in the DB table called "users"
// and log the user in
if (Auth::attempt($user))
{
  return 'User has been logged in successfully';
}
else
{
        return 'Login failed';
}
...
// Check if the user's browser has a cookie matching Laravel's session contents
if (Auth::check())
{
  return 'User is logged in';
}
...
// Retrieve some attributes of the currently logged in user
$username = Auth::user()->username;

echo $username;
// Outputs 'caspian356'
...
// Log the user out, by erasing the login cookie from the browser and destroying user's se\
ssion
Auth::logout();
```

These are just a few examples of Laravel's incredible authentication functionality. The list above is not a comprehensive list of all authentication methods Laravel 4 has up its sleeve. There are also password resets through a link sent in the email, HTTP Basic authentication (great for APIs) and of course built in methods of protecting areas of your web application. In later chapters we will get to know them better but for now let's take a look at another reason why Laravel is a great framework to use for your next web development project – methods of working with contents of databases.

1.3.5 Eloquent ORM – Laravel's Active Record implementation

Working with data is an essential part of all web applications. Usually PHP web applications store data in rows of database tables. Starting with version 5.0 the PHP language introduced a unified interface to query databases and to fetch the results. While those new features were helpful and enough for some basic applications, they made application's code look like a mess of SQL statements interwoven with PHP code. Also they did not add much convenience to working with data that is related with other data by means of foreign keys. To alleviate that imagine if you could present the data stored in a database table as a class with each row being an object. That would simplify access to the data in database and also would make it possible to relate the data by means of assigning objects to other objects' properties. This technique of wrapping DB data into objects is called Active Record pattern. Laravel implements Active Record pattern in its "Eloquent ORM".

ORM

Object Relational Mapping (ORM) enables access to and manipulation of data within a database as though it was a PHP object.

Laravel's Eloquent ORM allows developers to use Active Record pattern in full extent making application's code that deal with database look like clean PHP and not like messy SQL. Creation of new entries in database, manipulating and deleting entries, querying the database using a range of different parameters and much more are made easy and readable with Eloquent ORM. Eloquent makes quite a few assumptions about your databases but at the same time all of those assumptions could be overridden and customized (again, convention over configuration). For example it expects the primary key of each table to be called "id" and it expects the name of the tables to be plural of the model's name.

Let's say you have a table called "shops" in a database. A model that corresponds to that table would have to be called "Shop". If you wanted to use Eloquent ORM to create a new row in "shops" table containing the details of a new shop, you would do that by creating a new object of class Shop and operating on that object. This is done with code in listing 1.3:

Listing 1.3 Demonstration of Eloquent ORM creating a row in database

```
// Define the Eloquent model for "shop" in app/models/shop.php file
class Shop extends Eloquent {}
...
// Create a new instance of the model "shop"
$shop = new Shop;

// Assign content to the columns of the database row that will be created
$shop->name = 'Best Coffee Co';
$shop->city = 'Seattle';

// Perform a save of the new row into the database
$shop->save();
```

With Eloquent ORM you are not limited to operating on just one table in the database. Built-in relationships allow you to manipulate related models and in turn related tables. Eloquent assumes that you have a foreign key set up in format "model_id" on the tables that need to be related to a particular model. For

example if you have a table called “customers” containing foreign key called “shop_id” and wanted to relate the newly created shop to a new customer, you would use code that follows in listing 1.4:

Listing 1.4 Using Eloquent relationship to create related model

```
// Define Eloquent model for "customer" in app/models/customer.php file
class Customer extends Eloquent {}

...
// Eloquent model of "shop"
class Shop extends Eloquent {

  // One-to-many Eloquent relationship that relates model "shop" to model "customer"
  public function customers()
  {
    return $this->hasMany('Customer');
  }
}

...
// Create new object of "customer" Eloquent model
$customer = new Customer(array('name' => 'Prince Caspian'));

// Create a new row in table "customers" while setting "shop_id" column
// to the "id" of the shop that was created previously
$customer = $shop->customers()->save($customer);
```

The examples above are just a tiny tip of the iceberg of what’s possible with Eloquent. As you can see, Eloquent makes manipulating database data and related data a breeze while keeping the code of the application clean and easy to read. This book offers a whole chapter devoted to this powerful and unique ORM because being able to work with it will save you countless amounts of time. As many other developers did, you will find it a joy working with Laravel’s Eloquent ORM.

1.4 Summary

You have been introduced to Laravel PHP framework and have learned about its history and importance. Laravel has matured very quickly in just over two years. While being a young and fresh framework, it uses well established software design patterns:

- MVC application structure to keep the code separated while maintaining flexibility
- Active Record in its Eloquent ORM to simplify working with data inside of a database
- Convention over configuration to keep the setup minimal

You have learned that Laravel is a full stack framework. It features built in authentication, it has had a unique and powerful ORM since the first release, it is ready to be used out of the box with minimal configuration. From templating to working with data, Laravel has you covered. Applications built with Laravel feature code that is both expressive and clean at the same time, which makes them easier to maintain.

In the chapters that follow you will become a Laravel expert. Let's begin this journey by installing Laravel and making our first application with it!

2. Database operations

This chapter covers

- Configuring Laravel to use a database
- Introduction to Query Builder and Eloquent ORM
- Using Query Builder operators for managing data
- Performing Join queries

Working with data stored in a database is an essential feature of any modern web application. Saving user's orders and preferences, storing information about the products, creating data about uploaded files, all of these are just a few examples of application's actions that could use a database for storing data.

Managing data in the database used to be a painful process for PHP developers. Absence of built-in security, lack of common standards and consistency, messy database operations intermixed with the rest of the application's code have plagued PHP applications for many years. All of these and more problems concerning database management and database operations have mostly been solved by modern PHP frameworks like Laravel.

With Laravel's powerful database operators creating and managing records in a relational database like MySQL or PostgreSQL becomes much simpler and much more secure. Laravel interprets the data stored in the database into objects that you can easily manipulate like you would manipulate any other PHP object. This allows the developer to focus on other parts of the application instead of figuring out complex Structured Query Language (SQL) queries necessary to work with the data.

In this chapter you will learn how to set up Laravel to work with one of the many database engines that it supports. You will be then introduced to Laravel's Query Builder that makes creating and managing data in the database a breeze. After that you will get an introduction to Laravel's powerful Object Relational Mapping (ORM) called Eloquent and see how it could help manage related data. Then you will learn how you can use Laravel's query chaining and Query Builder operators to filter, sort, group and combine retrieved data for further operations.

2.1 Introducing Database Operations in Laravel

Laravel makes connecting to a database and managing data in it extremely simple and secure. Instead of requiring the developer to write complicated SQL queries it provides a convenient way of telling the application how you want to manipulate the data and Laravel would automatically translate those commands into SQL queries behind the scenes. As you will see throughout this chapter, from simple operations like insertion and updating of records to complex filtering and sorting to working with interrelated data, Laravel has the right tool for the job.

Laravel doesn't lock you into using a single database engine. Out of the box, it supports the following relational database systems:

- MySQL
- Postgres
- SQLite

- SQL Server

You can use any of these engines in your application after a simple configuration of one of the application's configuration files. When the database is configured, you can immediately use it in your application by using clean and elegant syntax of either Eloquent ORM or Query Builder.

Please note

Laravel provides two ways of operating with data in the database, by using Query Builder or Laravel's Object-Relational Mapping (ORM) called Eloquent. This chapter will focus on the Query Builder, while the whole next chapter will be devoted to Eloquent

When the application's code contains any DB operations, behind the scenes Laravel converts this code into proper SQL statements that are then executed on any of the DB engines supported by Laravel. The result of the SQL execution is then returned all the way back to the application code and the application could use the result to either show data to the user, or process the data according to application's requirements (figure 6.1):

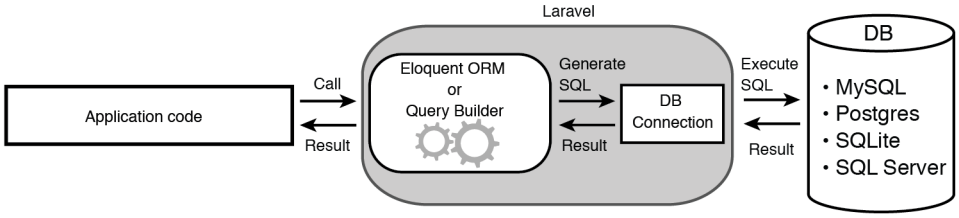

Figure 6.1 Database operations in Laravel

Laravel's flexibility and abundance of ways to work with relational data separate it from other frameworks. What other frameworks achieve with extensions and packages, Laravel has already built-in to help you manage data stored in a database efficiently. Some of the advanced database operations that Laravel provides out of the box are:

- Managing database tables and fields through special files called "migrations"
- Populating database with sample data (also called "seeding")
- Working with separate read/write databases
- Pagination
- Many different relationship types and advanced methods of querying relations
- Database transactions
- Query logging

When combined together, these features make Laravel one of the best PHP frameworks to work with complex databases while keeping things simple for the developer.

Also, an important feature of database operations in Laravel is that all queries are run through PDO (PHP Data Objects) extension. PDO is a powerful interface for database operations and it comes with all versions of PHP following version 5.1. Laravel makes use of PDO parameter binding that provides additional security.

Please note

Laravel uses PDO extension to execute any database operations which adds an extra layer of protection against SQL injection attacks and allows the use of many database engines beside MySQL

You will get to experience Laravel's powerful database operations in action as you read through this chapter. First, let's learn what options Laravel provides for configuring the database settings.

2.1.1 Configuring database settings

Laravel makes it extremely easy to configure the settings for interacting with a database. All configuration settings of a Laravel application are located in "app/config" folder. The file that stores the configuration settings for the database is named "database.php".

To use the database in your application you only need to open up "database.php" file and specify connection credentials for an existing database. Laravel will then use those credentials to do any further database-related operations.

Please note

Note Laravel cannot create new databases. It can only manage existing databases

Out of the box, the database configuration file has almost everything ready for you to be able to use the database. Sometimes the developer's needs go beyond the conventions. The database configuration file provides you with the following options for database operations:

- Preferred method of object retrieval (`PDO::FETCH_CLASS`, `PDO::FETCH_ASSOC`, `PDO::FETCH_OBJ` or any other method available in PDO)
- Which database connection the application should use by default
- Connection settings for MySQL, Postgres, SQLite or SQL Server database
- Connection settings for Redis
- Name of the table that will be used for "migrations" if migrations are used

By default, the database connection is set to use MySQL engine, PDO is set to fetch objects through "`PDO::FETCH_CLASS`" and most of the connection settings are already in place. Listing 6.1 shows the contents of database configuration file that Laravel comes with:

Listing 6.1 Default database configuration (app/config/database.php)

```
<?php

return array(
  // PDO fetch style (PDO::FETCH_CLASS, PDO::FETCH_ASSOC, PDO::FETCH_OBJ etc.)
  'fetch' => PDO::FETCH_CLASS,

  //The DB connection that will be used by the application for all DB operations
  'default' => 'mysql',
```

```
  'connections' => array(
    // SQLite connection settings
    'sqlite' => array(
      'driver'   => 'sqlite',
      'database' => __DIR__.'/../database/production.sqlite',
      'prefix'   => '',
    ),

    // MySQL connection settings
    'mysql' => array(
      'driver'    => 'mysql',
      'host'      => 'localhost',
      'database'  => 'database',
      'username'  => 'root',
      'password'  => '',
      'charset'   => 'utf8',
      'collation' => 'utf8_unicode_ci',
      'prefix'    => '',
    ),

    // Postgres connection settings
    'pgsql' => array(
      'driver'   => 'pgsql',
      'host'     => 'localhost',
      'database' => 'database',
      'username' => 'root',
      'password' => '',
      'charset'  => 'utf8',
      'prefix'   => '',
      'schema'   => 'public',
    ),

    // SQL Server connection settings
    'sqlsrv' => array(
      'driver'   => 'sqlsrv',
      'host'     => 'localhost',
      'database' => 'database',
      'username' => 'root',
      'password' => '',
      'prefix'   => '',
    ),
  ),
  'migrations' => 'migrations', // Name of the table used for migrations

  // Redis connection settings
```

```
  'redis' => array(
    'cluster' => false,
    'default' => array(
      'host'     => '127.0.0.1',
      'port'     => 6379,
      'database' => 0,
    ),
  ),
);
```

If you use a MySQL database on your machine, as all examples in this book are using, you will need to provide the name of an existing database. The default connection settings will most likely match your database for the exception of the username and password fields. Updating those configuration options is enough to start using a MySQL database on your local machine from Laravel!

Swapping database engines

One of the major advantages of Laravel using PDO for its database operations is that if you wanted to use a different database engine like Postgres instead of MySQL, you wouldn't change a single line of application's code and only database settings would need to be updated. This makes your applications much more future-proof and allows for easy swap out of database engines without rewriting any of application's code.

When you have the database settings configured, you are ready to start using the database in your application. Let's explore the two ways that you can use to work with databases in Laravel: Query Builder and Eloquent ORM.

2.1.2 Introducing Query Builder & Eloquent ORM

Laravel comes with two powerful sets of features to execute database operations, by using its own Query Builder or by using concept of "models" in Eloquent ORM. While you can use both in the same application (and as you will see later, even combine both to get the most flexibility out of DB operations) it helps to know the difference between the two. Let's first look at the Query Builder and then you'll meet Eloquent ORM.

2.1.2.1 Query Builder

Laravel's Query Builder provides a clean and simple interface for executing database queries. It is a powerful tool that can be used for various database operations such as:

- Retrieving records
- Inserting new records
- Deleting records
- Updating records
- Performing "Join" queries
- Executing raw SQL queries

- Filtering, grouping and sorting of records

Database commands that use the Query Builder use Laravel's "DB" class and look like the following (listing 6.2):

Listing 6.2 Examples of using Query Builder to create, filter, calculate & update data

```
// Create three new records in the "orders" table
DB::table('orders')->insert(array(
  array('price' => 400, 'product' => 'Laptop'),
  array('price' => 200, 'product' => 'Smartphone'),
  array('price' => 50,  'product' => 'Accessory'),
));

// Retrieve all records from "orders" table that have price greater than 100
$orders = DB::table('orders')
  ->where('price', '>', 100)
  ->get();

// Get the average of the "price" column from the "orders" table
$averagePrice = DB::table('orders')->avg('price');

// Find all records in "orders" table with price of 50 and update those records
// to have column "product" set as "Laptop" and column "price" set as 400
DB::table('orders')
  ->where('price', 50)
  ->update(array('product' => 'Laptop', 'price' => 400));
```

Query Builder is very easy to use yet powerful way to interact with the data in your database. As noted before, in this chapter we will mainly focus on using the Query Builder and you will get to see most of its methods of working with database in action. Now let's take a look at Eloquent ORM before diving deeper into the Query Builder and other concepts.

2.1.2.2 Eloquent ORM

Since its very first version Laravel featured an intelligent ORM called "Eloquent ORM". Eloquent is Laravel's implementation of Active Record pattern that uses the concept of "models" to represent data and it makes working with interrelated data easy. Eloquent is very good at working with complicated data relationships while remaining efficient, easy to manage and fast.

ORM, Active Record and Eloquent

Object-Relational Mapping (ORM) is a technique of representing data stored in database tables as objects that make database operations as easy as working with any other objects in PHP.

Active Record is an architectural pattern that enables representing database tables as special classes (also called "models") and brings the concepts found in Object Oriented Programming into database operations.

Popular web development frameworks like CodeIgniter, Ruby on Rails and Symfony use Active Record

> pattern to simplify database operations. Eloquent ORM uses Laravel's own powerful implementation of Active Record. When a database table is represented as a model, creation of a new row in that table is as easy as creating a new instance of the model. Working with tables and with the data stored in them becomes similar to working with any other classes and objects in PHP and even complex data relationships are easy to manage with Eloquent.

To get started using Eloquent you only need to configure the database connection settings in "app/config/-database.php" file and create "models" that correspond to tables of your database. The Eloquent syntax looks not much different from plain OOP PHP code so it is very easy to read. Eloquent provides many advanced features to manage and operate on data, the most prominent of them are:

- Working with data by the use of "models"
- Creating relationships between data
- Querying related data
- Conversion of data to JSON and arrays
- Query optimization
- Automatic timestamps

From simple data insertion to association of data between tables to complex filtering within related data, Eloquent has you covered. The clean and logical syntax is something that separates Eloquent from other existing ORMs. Listing 6.3 shows an example of Eloquent syntax in action to create a new user, new order and then associate the new order to the newly created user:

Listing 6.3 Example of using Eloquent ORM to create and associate related data

```
// Create a model definition for the table "users" indicating its
// relationship to the "orders" table

// app/models/User.php
class User extends Eloquent {
  public function orders()
  {
    return $this->hasMany('Order');
  }
}

// Create a model definition for the table "orders"
// app/models/Order.php
class Order extends Eloquent {}

// application code

// Create a new row in "users" table
$user = new User;
$user->name = "John Doe";
```

```
$user->save();

// Create a new row in “orders” table and link it to the user
$order = new Order;
$order->price   = 100;
$order->product = "TV";

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

The example above is just barely scratching the surface of Eloquent's features. There is just so much about Eloquent that it deserves its own chapter and you will get to experience this mighty ORM at a much deeper level in the next chapter. Right now let's take a closer look at using the Query Builder.

2.2 Using Query Builder

From inserting and managing records to sorting and filtering, Query Builder provides you with convenient operators to work with the data in an existing database. Most of these operators could be combined together to get the most out of a single query.

Application code using Query Builder uses Laravel's “DB” class. When the application executes any command of class “DB” (a command could also be called an “operator”), Laravel's Query Builder will build an SQL query that will be executed against a table specified as an argument to “table” method (figure 6.2). That query will be executed on a database using the database connection settings specified in the “app/config/database.php” file. The result of the query execution will then be passed all the way back to the caller changing its state: by returning retrieved records, a Boolean value or returning an empty result set.

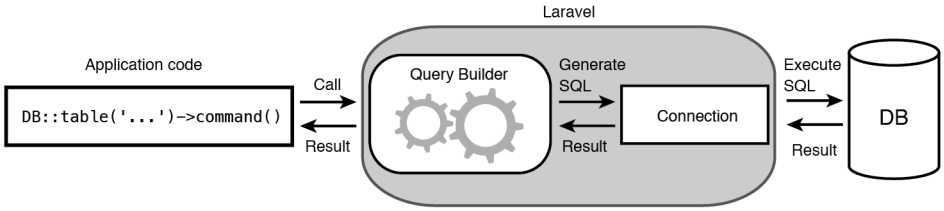

Figure 6.2 Executing database operations using Query Builder

The operators that Query Builder provides for some of the most common operations are listed in table 6.1 along with descriptions of these operators:

Table 6.1 Most commonly used operators of the Query Builder

Operator	Description
`insert(array(...))`	insert a new record using the data provided as an array of key – value pairs
`find($id)`	retrieve a record that has a primary key “id” equal to the provided argument
`update(array(...))`	update a record using the data provide as an array of key – value pairs
`delete()`	delete a record
`get()`	retrieve the record as an object containing the column names and the data stored in the record
`take($number)`	retrieve only a limited number of records

Throughout the next few pages you will get to see the Query Builder in action and meet a few of its powerful operators listed in the table 6.1. Let's start with the basics, inserting new records into an existing table in a database.

2.2.1 Inserting records

The "insert" operator inserts a new row (record) into an existing table. You can specify what data to insert into the table by providing an array of data as a parameter to "insert" operator. Let's take a look at a simple example. Imagine that you have a table called "orders" that has 3 fields (columns) in it: auto incrementing ID, price and a name of the product (table 6.2).

Table 6.2 Table structure for the "orders" table

Key	Column	Type
primary	id	int (11), auto-incrementing
	price	int (11)
	product	varchar(255)

2.2.1.1 Inserting a single record

Let's use the "insert" operator of Query Builder to insert a new record into this table. By passing an array formatted as column-value pair you can tell the Query Builder to insert a new record using the data passed as a parameter to the "insert" operator (listing 6.4):

Listing 6.4 Inserting a new record using Query Builder

```
// Tell the query builder to do an insertion on the "orders" table
DB::table('orders')->insert(
  array(
    'price'   => 200, // Set a value for the column "price"
    'product' => 'Console' // Set a value for the column "product"
    )
);
```

When this code is executed the following process will take place behind the scenes. Laravel's Query Builder will translate the "insert" command into SQL query specific to the database that is currently in use by the application (specified in "database.php" configuration file). The data passed as an argument to the "insert" command will be placed into the SQL query in form of parameters. The SQL query will then be executed on the specified table ("orders") and the result of query execution will be returned to the caller. Figure 6.3 illustrates this whole process:

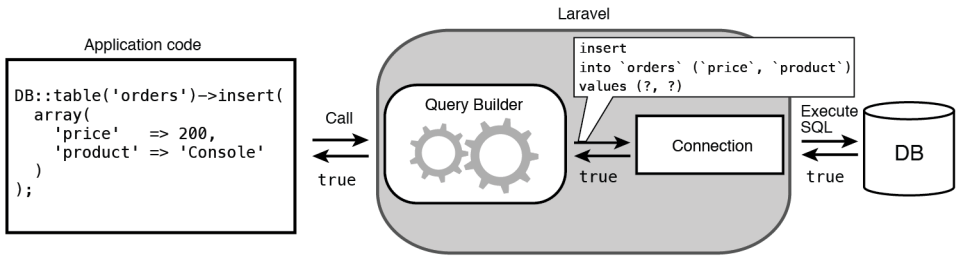

Figure 6.3 Behind the scenes process of running "insert" operator

You might have noticed the question marks instead of actual values passed to the SQL query above. As noted in the introduction section of this chapter, Laravel uses PDO to execute SQL statements. Behind the scenes PDO will attach the data (in this case "200" and "Console") to the unnamed placeholders seen as question marks. Using a prepared statement by including placeholders for data adds protection from SQL injection and increases security of insertions and updates of data. As a developer you don't need to enable this in any way, Laravel does this for you out of the box. Now that you know how to insert a single record, let's also take a look at inserting many records.

2.2.1.2 Inserting multiple records

The same "insert" operator of the Query Builder could be used to insert more than one row of data. Passing an array containing arrays of data allows you to insert as many rows as you would like to, two, three or a thousand. Listing 6.5 shows the use of "insert" operator to create three new rows of data:

Listing 6.5 Inserting multiple records using Query Builder

```
// Create three new records in the "orders" table
DB::table('orders')->insert(array(
  array('price' => 400, 'product' => 'Laptop'),
  array('price' => 200, 'product' => 'Smartphone'),
  array('price' => 50,  'product' => 'Accessory'),
));
```

When executed, the "insert" statement in listing above will create three new records, and the SQL query that Laravel will build behind the scenes to do that is:

```
1  insert into `orders`(`price`,`product`) values (?, ?),(?, ?),(?, ?)
```

As you can see Laravel is smart enough to do the insertion of three rows of data in one query instead of running three separate queries. Now that you can easily insert data using Query Builder, let's learn how to retrieve data.

2.2.2 Retrieving records

Query Builder provides a few convenient operators of getting the data from the database. To accommodate many different situations it is very flexible, allowing you to do the following operations:

- Retrieve a single record from a table
- Retrieve all records in the table
- Retrieve only specific columns of all records in the table
- Retrieve a limited number of records in the table

Over the course of the next few pages we will look at these operators of retrieving data. Let's start with the basics, getting a single record from the database.

2.2.2.1 Retrieving a single record

You can retrieve a single record from the table by using operator "find". Just provide the value of the primary key of the record that you'd like to retrieve as a parameter to operator "find" and Laravel will return that record as an object or return NULL if the record is not found. The example in listing 6.6 illustrates using "find" to retrieve a record with the primary key set to "3":

Listing 6.6 Retrieving a single record using operator 'find'

```
// Retrieve a record with primary key (id) equal to "3"
$order = DB::table('orders')->find(3);

// If there is a record with ID "3", The $order variable will contain:
object(stdClass)#157 (3) {
  ["id"]=>
  string(1) "3"
  ["price"]=>
  string(3) "200"
  ["product"]=>
  string(10) "Smartphone"
}
```

Please note

Operator "find" expects the column containing record's primary key to be called "id". You will need to use other operators of retrieving records if your table doesn't use "id" as the primary key

Running the code in listing 6.6 would execute the following SQL, binding the value passed into "find" operator as a parameter and limiting the number of returned records to a maximum of 1:

```
1  select * from `orders` where `id` = ? limit 1
```

As you can see, retrieving a single record is easy with the use of operator "find". What if you wanted to retrieve all records from a table?

2.2.2.2 Retrieving all records in the table

To retrieve all records from a table you can use operator "get" without any parameters. Running "get" on a specified table without any prior operators would return all records from that table as an array of objects. Imagine that you wanted to retrieve all records from the "orders" table. Listing 6.7 below shows how you would get all data from that table as an array of objects:

Listing 6.7 Retrieving all records using 'get'

```
// Using "get" without any prior operators will grab all records in the
// specified "orders" table
$orders = DB::table('orders')->get();
```

Running operator "get" without any parameters would run the following SQL query behind the scenes and return its result as an array of objects:

```
1  select * from `orders`
```

The result returned from execution of code in listing 6.7 would have an array of objects containing data of all rows in the table. Each object would have array's keys storing the names of the columns of the table and array's values storing the row's values. Listing 6.8 shows the resulting array of objects:

Listing 6.8 Result of retrieving all records (contents of orders table)

```
array(4) {
  [0]=>
  object(stdClass)#157 (3) {
    ["id"]=>
    string(1) "1"
    ["price"]=>
    string(3) "200"
    ["product"]=>
    string(7) "Console"
  }
  /* ... 3 more rows returned as objects ... */
}
```

What if you wanted to retrieve just a few specific columns of all records of the table instead of all columns? Let's take a look how you could do that with Query Builder.

2.2.2.3 Retrieving specific columns of all records in the table

To get only specific columns of all records in the table you can still use the "get" operator, but this time pass the desired column names as an array of parameters to the "get" operator. For example if you wanted to retrieve columns "id" and "price" of all records, while omitting "product" column, you would pass "id" and "price" as a parameter to the "get" operator like in listing 6.9:

Listing 6.9 Retrieving specific columns of all records in a table

```
// Tell the Query Builder to only retrieve "id" and "price" columns
$orders = DB::table('orders')->get(array('id','price'));
```

Running this statement would produce the following SQL:

```
1  select `id`, `price` from `orders`
```

The returned data in this case would contain an array of objects as follows in listing 6.10:

Listing 6.10 Result of retrieving specific columns of all records (contents of $orders)

```
array(4) {
  [0]=>
  object(stdClass)#157 (2) {
    ["id"]=>
    string(1) "1"
    ["price"]=>
    string(3) "200"
  }
  ... 3 more rows returned as objects ...
}
```

Laravel's Query Builder tries to make retrieval of data extremely simple and efficient. The flexibility of specifying which columns of the table you'd like to get simply by passing the names of the columns to "get" operator could be very useful. It could come handy when your application works with big amounts of data. Retrieving only specific columns from the table instead of the whole table cuts down the amount of RAM your application uses.

What if you wanted to limit the maximum number of records that a Query Builder statement would return? Let's take a look at how you can tell Query Builder to only retrieve a specific number of records.

2.2.2.4 Retrieving a limited number of records

To specify a maximum number of records that you'd like to get from a table you can use operator "take" with a number of records passed as a parameter to it and appending operator "get" to the query. For example imagine if you had 1000 records in the "orders" table and you wanted to only retrieve a maximum of 50 records from it. Using the operator "take" with 50 passed as a parameter, you can. Listing 6.11 shows this in action:

Listing 6.11 Retrieving a limited number of records from a table

```
$orders = DB::table('orders')->take(50)->get();
```

Executing this query would result in $orders being an array of objects with maximum of 50 objects that store the data retrieved from the table. Now that you know how to create and retrieve data in various ways, you are ready to learn how to update existing data in the database.

2.2.3 Updating records

Updating records using Query Builder is very similar to creating new records. To update the data of an existing record or a set of records you can append operator "update" to the query and pass an array of new data as a parameter to it. You can use query chaining, a concept that we will explore later in this chapter, to target specific records that would be updated.

2.2.3.1 Updating specific records

To update only specific records in the table you would need to somehow tell Laravel which records you want updated. Fortunately you can use Query Builder's filtering operators and append the "update" operator to the end of the query targeting only specific records.

One of the filtering operators is operator "where" that allows you to choose records by specifying a column, comparison operator and a value to compare the data to. For example if you wanted to update all records in table "orders" that have price set to more than 50, you would construct the following query (listing 6.12):

Listing 6.12 Updating a column of all records in a table that match a criterion

```
// Tell the Query Builder to target all rows with column 'price' set to more than 50 in
// table "orders"
DB::table('orders')
  ->where('price','>','50')
  ->update(array('price' => 100)); // Update column 'price' with a new value
```

Executing this query would generate the following SQL query:

```
update `orders` set `price` = ? where `price` > ?
```

You are not limited to updating just one column of the records. Passing more key-value pairs in the array supplied to "update" operator as a parameter would update all specified columns. What if you wanted to update all records in the table at once?

2.2.3.2 Updating all records in the table

Even though it is a rare case to update all records in a table, you can update columns of all records in the table by appending "update" operator to the beginning of the DB query. Listing 6.13 shows an example of updating the column "product" of all records in the "orders" table:

Listing 6.13 Updating a column of all records in a table

```
// Tell the Query Builder to update column 'product' of all records to be 'Headphones'
DB::table('orders')->update(array('product'=>'Headphones'));
```

Executing this query would generate the following SQL query:

```
update `orders` set `product` = ?
```

As you can see, Query Builder allows updating the records in a table in multiple ways that are easy to write. In the next section we will take a look at deleting existing records.

2.2.4 Deleting records

Deleting records from a table using Query Builder follows the same patterns as updating records. You can delete specific records that match some criteria or delete all records from the table by using operator "delete".

2.2.5 Deleting specific records

To delete specific records in the table you can also use filtering operators like "where" to target records matching some sort of filtering criteria. For example if you wanted to delete the records that have "product" column set to "Smartphone", you would append "delete" operator to the query specifying which records to target (listing 6.14):

Listing 6.14 Deleting records that match a criterion

```
// Tell the Query Builder to target all records that have column "product"
// set to "Smartphone"
DB::table('orders')
  ->where('product','=','Smartphone')
  ->delete(); // Delete records matching the criterion
```

Executing this query would generate the following SQL query:

```
delete from `orders` where `product` = ?
```

The query in listing 6.14 would affect only records that were targeted by the "where" operator. If you wanted to delete all records from a table, you would do it in a way similar to updating all records, by only having operator "delete" following "DB::table()" statement.

Now that you know how to insert, retrieve, update and delete records, let's learn how to achieve precise control when executing any of these actions by filtering, sorting and grouping data.

2.3 Filtering, sorting and grouping data

When managing data in the database applications often need to have precise control over which records will be affected. This might be either getting exactly the set of data required by application's specifications or deleting just a few records that match some criteria. Some of these operations could get very complicated if you were to use plain SQL. Laravel's Query Builder allows you to filter, sort and group data while maintaining clean consistent syntax that is easy to understand. You can see a list of most commonly used operators to filter, sort, and group data in table 6.3 below:

Table 6.3 Query Builder operators for filtering, sorting and grouping data

Operator	Description
`where('column','comparator','value')`	retrieve records that match a criterion
`orderBy('column','order')`	sort records by specified column and order (ascending or descending)
`groupBy('column')`	group records by a column

Over the next few pages you will get deeper understanding of how to operate on very specific records from the database using the operators from table 6.3. While getting to know the various ways of operating with data you will get to meet and use the concept of "query chaining".

2.3.1 Query Chaining

Query chaining allows you to run many database operations in a single query. Imagine that you had a table "users" and wanted to retrieve an object containing all users with first name "John", located in California and sorted by their birthdate. How would you do that in an efficient way?

One way would be to first retrieve an object containing all users with first name "John", then do further filtering using PHP's "foreach" loops to get to the result that you want. While this sounds acceptable in theory, it is not efficient and could take up a lot of server's RAM. In addition, if the requirements for the application change, there would be no way to quickly update code like that to accommodate new functionality. Of course like with many other common web development problems, Laravel provides a better option.

Instead of creating separate queries to filter and sort data, you could put many of them together, effectively "chaining" them. A query chain combines a sequence of various actions that can be performed with data one after another to get a specific result that can be operated on. Filtering data by various parameters, sorting data and more could be represented as a series of actions performed upon data in a table (figure 6.4):

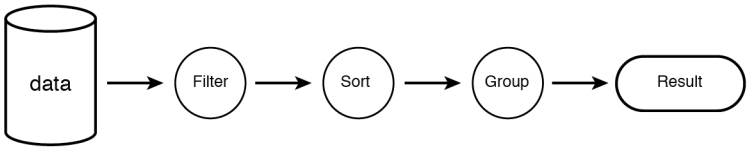

Figure 6.4 Concept of chaining actions together to get specific data from the database

Laravel allows you to put as many queries together as you want. From running two operations to ten and more, query chaining can significantly cut down on the amount of code you need to write to execute complicated database operations. For example to execute aforementioned actions on the "users" table you could put the filtering and sorting together into a single query chain like on the left side of figure 6.5:

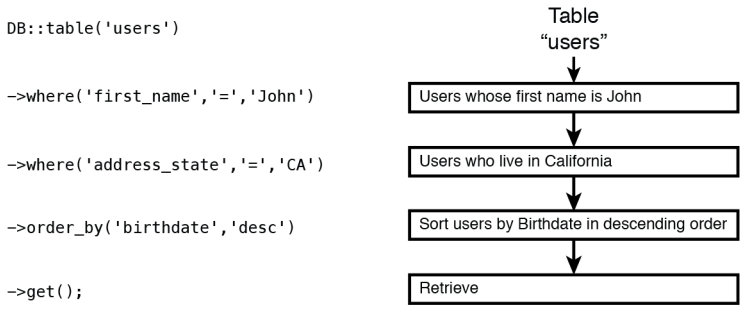

Figure 6.5 Example of query chaining in action

Note

You can use query chaining to execute multiple operations like sorting, filtering, grouping to precisely target a set of data that can be further retrieved, updated or deleted. You cannot mix insert/get/update/delete operations together in a single query.

While query chaining is not a new concept in the world of web development frameworks, Laravel provides one of the most elegant ways to do it. Let’s look at the first operator that demonstrates the use of query chaining, operator “where”.

2.3.2 Using “where” operator to filter data

Query Builder’s operator “where” makes filtering records easy. It provides a clean interface for executing SQL “WHERE” clause and has syntax that is very similar to it. Some of the operations that you could do with this operator include:

- Selecting records that match a certain criterion
- Selecting records that match either one criterion
- Selecting records that have a column that lies in a certain range of values
- Selecting records that have a column that is out of range of values

After the records are selected using “where” you can do any of the operations discussed previously: retrieval, updating or deleting. Alternatively you could apply more “where” operators by the use of query chaining to achieve more precise matching. The flexibility that Laravel provides for filtering data is truly superb. Let’s start exploring the powerful operator “where” by looking at using it in a simple query.

2.3.2.1 Simple “where” query

A query using “where” consists of supplying three parameters that will be used to filter the data:

- The name of the column that will be used for comparison
- The operator that will be used to compare data
- The value that the columns’ contents will be compared to

The diagram in figure 6.6 shows the syntax of using operator “where”:

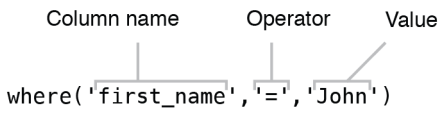

Figure 6.6 Syntax of using operator “where”

Operator “where” can be used not only for finding entries that exactly match the desired value. In fact you can use any of the operators supported by the database to specify what kind of matching you need. For numerical comparison you might want to use “>” or “<” operators to get data that has values of a certain numerical column greater or less than provided value. For string comparison you might use “like” or “not like” comparison operator.

Note

If you pass only two parameters to operator “where”, Laravel will assume that you want to do strict checking (using “=” to compare) which in certain cases can cut down on the amount of code even more

A list of allowed comparison operators that work in all databases supported by Laravel is presented in table 6.4 below:

Table 6.4 Allowed comparison operators for “where” operator

Operator	Description
”=”	Equal
”<”	Less than
”>”	Greater than
”<=”	Less than or equal
”>=”	Greater than or equal
“<>” or “!=”	Not equal
“like”	Simple pattern matching (you can use the “%” sign with this operator to define wildcards both before and after the pattern)
“not like”	Negation of simple pattern matching

Besides using a single “where” operator you can chain more than one “where” to filter the results even further. Under the covers Laravel will automatically put an “AND” in the SQL statement between the chained “where” operators. For example if you wanted to get all users that have last name starting with letters “Sm” and whose age is less than 50, you would create a query chain as shown in listing 6.15:

Listing 6.15 Chaining multiple "where" operators

```
// Use table "users"
$users = DB::table('users')
  // Match users whose last_name column starts with "Sm"
  ->where('last_name', 'like','Sm%')
  // Match users whose age is less than 50
  ->where('age','<' ,50)
  // Retrieve the records as an array of objects
  ->get();
```

Running this query would produce the following SQL behind the scenes:
select * from `users` where `last_name` like ? and `age` < ?

Laravel allows chaining as many "where" operators as you need to allow very precise control over which records you would like to operate with. Let's now look at using another operator, "orWhere" that complements "where" operator nicely.

2.3.2.2 Using "orWhere" operator

To select data that matches at least one criterion out of several could be achieved by using "orWhere" operator. This operator has exactly the same syntax as "where" operator and it has to be appended to an existing "where" operator in order for it to work.

Let's imagine that you wanted to delete all records from the "orders" table that have been either marked with "1" in "processed" column or whose "price" column is less than or equal to 250. Using "where" and "orWhere" operators together and appending "delete" operator at the end will do just that. Listing 6.16 shows using "orWhere" operator to filter records and subsequently deleting the records that match at least one criterion:

Listing 6.16 Combining "where" and "orWhere" operators

```
// Use table "orders"
$orders  = DB::table('orders')
  // Match orders that have been marked as processed
  ->where('processed', 1)
  // Match orders that have price lower than or equal to 250
  ->orWhere('price','<=' ,250)
  // Delete records that match either criterion
  ->delete();
```

This query would generate the following SQL:
delete from `orders` where `processed` = 1 or `price` <= 250

Note

You can chain more than one "orWhere" operator together for most flexibility We looked at using "where" and "orWhere" operators and you now know how to use them together. Now let's look at "whereBetwen" operator.

2.3.2.3 Using “whereBetween” operator

“whereBetween” operator comes handy when you want to match records that have a numerical column set to a value that falls within a certain range. The syntax of “whereBetween” is a bit simpler than that of “where” or “orWhere” operators (figure 6.7). This operator expects only two parameters, a column that will be used for matching and an array containing two numerical values indicating a range.

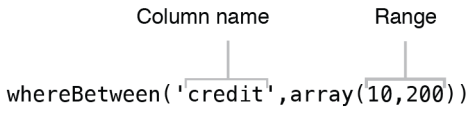

Figure 6.7 Syntax of “whereBetween” operator

Adding this operator will match the records that have the value of the specified column fall in the provided range. For example if you wanted to retrieve all records in the “users” table that have the value of the “credit” column set to anything with 100-300 range, you would use the following query:

Listing 6.17 Using “whereBetween” operator

```
// Use table “users”
$users  = DB::table('users')
  // Match users that have the value of “credit” column between 100 and 300
  ->whereBetween('credit', array(100,300))
  // Retrieve records as an array of objects
  ->get();
```

The query in the listing above would execute the following SQL:

select * from `users` where `credit` between ? and ?

You have now learned many different ways to get the desired data. As you have seen, filtering data with the operator “where” and its derivatives doesn’t involve complex SQL statements when using Query Builder. The Query Builder has a lot more features that allow more control for the way the data is retrieved. Let’s take a look at sorting and grouping data.

2.3.3 Using “orderBy” to sort data

“orderBy” operator of the Query Builder provides an easy way to sort the data that is retrieved from the database. Just like “where” operator is similar to “WHERE” clause in SQL, “orderBy” operator is very similar to the “ORDER BY” clause found in SQL. Query Builder’s syntax for this operator is practically the same as it is in SQL. To sort a set of data by some column, two parameters need to be passed to the “orderBy” operator: the column by which to sort the data and the direction of sorting (ascending or descending). Figure 6.8 shows syntax of “orderBy” operator:

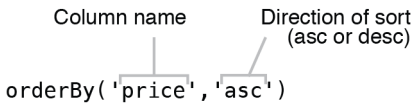

Figure 6.8 Syntax of “orderBy” operator used to sort data

Applying “orderBy” operator to a set of data retrieved by one of the operators of Query Builder will sort the data according to the column name and the direction specified through the use of parameters. For example imagine that you had a table called “products” and wanted to sort the products by their price, in ascending order. You could use the query from listing 6.18 to do that:

Listing 6.18 Using “orderBy” operator to sort products by price

```
// Use table “products”
$products  = DB::table('products')
  // Sort the products by their price in ascending order
  ->orderBy('price', 'asc')
  // Retrieve records as an array of objects
  ->get();
```

You could also use query chaining to accommodate more complex filtering and sorting scenarios. For example if you wanted to get all records from the “products” table whose “name” column contains letters “so” and whose “price” column is greater than 100 and sort the result by the “price” column in ascending order, you would construct the following query as in listing 6.19:

Listing 6.19 Using “orderBy” operator with other operators

```
// Use table “products”
$products  = DB::table('products')
  // Get products whose name contains letters “so”
  ->where('name','like','%so%')
  // Get products whose price is greater than 100
  ->where('price','>', 100)
  // Sort products by their price in ascending order
  ->orderBy('price', 'asc')
  // Retrieve products from the table as an array of objects
  ->get();
```

“orderBy” operator is chainable just like “where” operator. You can combine multiple “orderBy” operators to get a sorted result that you need to achieve. If you also want to group records together, you can use the “groupBy” operator.

2.3.4 Using “groupBy” to group data

You can group records together using “groupBy” operator that is similar to SQL “GROUP BY” clause. It accepts only one parameter – the column by which to group the records by. For example if you had a table “products” and wanted to retrieve all records from it grouped by the product name, you would create the following query (listing 6.20):

Listing 6.20 Using "groupBy" operator to group data

```
// Use table "products"
$products  = DB::table('products')
  // Group products by the "name" column
  ->groupBy('name')
  // Retrieve products from the table as an array of objects
  ->get();
```

Now that you know how to use filtering, sorting and grouping of records in various ways and are familiar with the basics of query chaining, we can go on to a bit more complicated topic and learn how to use Laravel's advanced query operators to do "join" statements.

2.4 Using Join Statements

Laravel's Query Builder supports all types of Join statements supported by the database you are using. Join statements are used to combine records from multiple tables that have values common to those tables. Consider two tables, "Users" and "Orders", with contents as shown in figure 6.9:

users

id	name
5	John
6	Mark
7	Sam

orders

id	user_id	item
100	5	TV
120	7	Laptop
122	5	Phone

Figure 6.9 Contents of two sample tables

While you can use Join statements to combine more than two tables, we will use only the two small tables from figure 6.9 to demonstrate some of the types of Join statements that you can use in Query Builder.

Note

Caution should be taken if the joined tables have columns with the same names. You can use "select()" operator to alias duplicate columns

2.4.1 Using Inner Join

Inner Join is a simple and most common type of Join. It is used to return all of the records in one table that have a matching record in the other table. For example if you wanted to know which users have placed an order you could use an Inner Join statement that would return a list of all users with orders in the "orders" table and data about each of their order.

Note

You can chain more than one Join statement together to join more than two tables

Inner Join in Query Builder follows the syntax in figure 6.10 below:

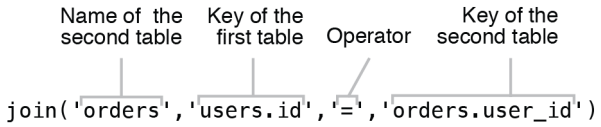

Figure 6.10 Syntax of Inner Join

Let's apply this syntax to the "users" and "orders" tables, to see which records are matching between them. In our example tables the primary key in each table is the "id" column. Listing 6.21 shows the Join query applied using the "users" table as first table and "orders" table as the second table:

Listing 6.21 Using Inner Join

```
// Use table "users" as first table
$usersOrders  = DB::table('users')
  // Perform a Join with the "orders" table, checking for the presence of matching
  // "user_id" column in "orders" table and "id" column of the "user" table.
  ->join('orders', 'users.id', '=', 'orders.user_id')
  // Retrieve users from the table as an array of objects containing users and
  // products that each user has purchased
  ->get();
```

When we run this query, the following SQL would be executed behind the scenes:

```
1  select * from `users` inner join `orders` on `users`.`id` = `orders`.`user_id`
```

This query would look for values of "user_id" column in the "orders" table that have a matching value in the "id" column in the "users" table. A Venn diagram on the left side of figure 6.11 shows the result of Inner Join as the shaded area between the two sets of data. If represented as a table, the resulting array of objects assigned to $usersOrders after running the Join statement in listing 6.21 would look like the table on the right side of figure 6.11:

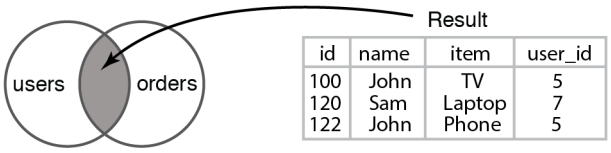

Figure 6.11 Result of running an Inner Join query between the "users" and "orders" tables

As you can see, running an inner Join could be useful and simple with the Query Builder. Now let's explore another type of join, a Left Join.

2.4.2 Using Left Join

Left Join is a bit more inclusive than the Inner Join and has syntax similar to it. It produces a set of records that match between two tables and in addition it returns all records from the first table that don't have a match. When creating this type of join the only difference in syntax is using "leftJoin" operator instead of "join" as shown in figure 6.12 below:

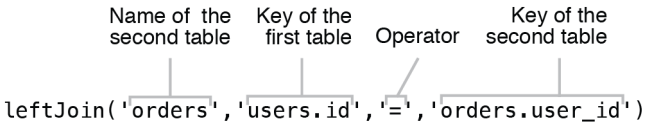

Figure 6.12 Syntax of Left Join

Let's use this type of Join on our two sets of data, the "users" and "orders" table to see what the result would look like. Listing 6.22 shows usage of the "leftJoin" operator to join the two tables:

Listing 6.22 Using Left Join

```
// Use table "users" as first table
$usersOrders  = DB::table('users')
  // Perform a Left Join with the "orders" table, checking for the presence of
  // matching "user_id" column in "orders" table and "id" column of the "user" table.
  ->leftJoin('orders', 'users.id', '=', 'orders.user_id')
  // Retrieve an array of objects containing records of "users" table that have
  // a corresponding record in the "orders" table and also all records in "users"
  // table that don't have a match in the "orders" table
  ->get();
```

When we run this query, the following SQL would be executed behind the scenes:

```
1  select * from `users` left join `orders` on `users`.`id` = `orders`.`user_id`
```

This query would contain all rows from "users" regardless of having a matching entry in the "orders" table. The values of columns of the result would consist of the same values as if you had an Inner Join but those rows from the "users" table that don't have a match in the "order" would return NULL for "id", "item" and "user_id" columns (figure 6.13). If represented as a table, the resulting array of objects assigned to $usersOrders after running the Left Join statement in listing 6.22 would look like the table on the right side of figure 6.13:

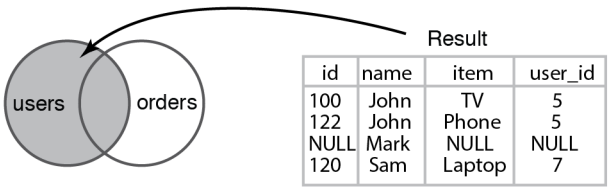

Figure 6.13 Result of running a Left Join query between the "users" and "orders" tables

Left Joins are useful in those cases when you need to get all records from the first table even if there are none matching records in the second table. There might be a need to use another type of Joins. Fortunately Laravel doesn't limit you by just Inner Joins and Left Joins. In fact, you can use any type of join supported by your database and we will learn how to do that in the next section.

2.4.3 Using other types of Joins

Laravel's Query Builder is so flexible that it takes into account special cases of Join queries and allows you to execute all types of Join queries supported by your database. While most SQL database engines (like

MySQL and SQLite) support Inner Right and Left Joins/Outer Right and Left Joins, other SQL engines like Postgres and SQL Server support Full Joins in addition to other types of Join queries. You can execute any type of Join that your database supports by supplying a fifth argument into "join" operator specifying which type of Join query you want to execute. Figure 6.14 shows the syntax of Join operator with a custom type of the Join query:

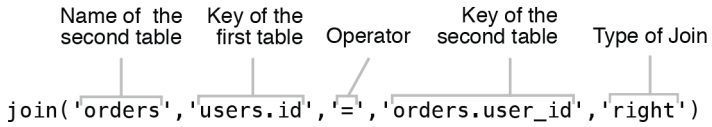

Figure 6.14 Using fifth argument of "join" operator for custom type of Join query

From cross joins to outer joins and full joins, the flexibility of the "join" operator would fit any possible Join query scenario. Listing 6.23 shows some of the custom types of Joins that work with all database engines supported by Laravel:

Listing 6.23 Using other types of Join queries

```
// Right Join
join('orders', 'users.id', '=', 'orders.user_id','right')

// Right Outer Join
join('orders', 'users.id', '=', 'orders.user_id','right outer')

// Excluding Right Outer Join
join('orders', 'users.id', '=', 'orders.user_id','right outer')
     ->where('orders.user_id',NULL)

// Left Join
join('orders', 'users.id', '=', 'orders.user_id','left')

// Left Outer Join
join('orders', 'users.id', '=', 'orders.user_id','left outer')

// Excluding Left Outer Join
join('orders', 'users.id', '=', 'orders.user_id','left outer')
     ->where('orders.user_id',NULL)

// Cross join
join('orders', 'users.id', '=', 'orders.user_id','cross')
```

Multiple types of chainable Join queries could make almost any complex database operation possible. Great support of Join queries along with other database operators you have seen up to this point makes working with database in Laravel a pleasant process instead of being a mundane operation.

2.5 Summary

In this chapter you have looked at Laravel's powerful methods of working with databases. From simple connection configuration to creation of complicated filtering and sorting scenarios Laravel provides convenient database operators at every step.

You have met Query Builder and have used its easy-to-follow syntax to insert, retrieve, update and delete records. You have used its "where" operators to filter records, "orderBy" to sort them and "join" to build various types of Join queries. Besides, on the way to getting a close look at the features of Query Builder you have learned an important concept of database operations in Laravel – query chaining that makes multi-stage database operations efficient and easy to work with.

The concepts you have learned so far in this chapter are fundamental to the next chapter where you will get to know one of the most elegant ORMs of modern web development frameworks, Eloquent. In the next chapter we will take a closer look at Eloquent ORM and become familiar with its smart ways to manage data and relationships between the data in Laravel-powered web applications.