

Community Experience Distilled

Learning Laravel's Eloquent

Develop amazing data-based applications with Eloquent, the Laravel framework ORM

Francesco Malatesta

[PACKT] open source*
PUBLISHING

Table of Contents

Learning Laravel's Eloquent
Credits
About the Author
Acknowledgments
About the Reviewers
www.PacktPub.com
    Support files, eBooks, discount offers, and more
        Why subscribe?
        Free access for Packt account holders
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Errata
        Piracy
        Questions
1. Setting Up Our First Project
    Your Swiss Army Knife – Composer
        Installing Composer
        The composer.json and autoload files
        The most used commands
    Your safe place – Homestead
        Installing Homestead
            The Composer and PHP tool way
            The Git way
        Configuring Homestead
    The new hideout: Homestead improved
    A bonus tool – Adminer
    Your best friend: Laravel
        Installing Laravel
            Using the Laravel installer tool
            Using the Composer create-project command
    The first project – EloquentJourney
        The configuration system
        Setting up the database connection
    Summary
2. Building the Database with the Schema Builder Class
    The Schema Builder class
        Working with tables
        Working with columns
        Columns' methods reference

        Other $table object methods
        Updating tables and columns
        Indexes and foreign keys
    Database versioning with the migrations system
        Creating migrations
        Rolling back migrations
        More examples, more migrations
    Summary
3. The Most Important Element – the Model!
    Creating a Model
    Create, read, update, and delete operations basics
        Creating operations
        Reading operations
        Updating operations
        Deleting operations
    where, aggregates, and other utilities
        where and orWhere
            Magic wheres
        Aggregates
        Utility methods
    Mass assignment... for the masses
    Timestamps and soft deletes
        Timestamps
        Soft deleting
    Query scopes
    Attributes casting, accessors, and mutators
        Attributes casting
        Accessors and mutators
    Descending in the code
        A big file
        Quick conversion to array or JSON
        Imaginary attributes
        Route model binding
        Records chunking for memory optimization
    Summary
4. Exploring the World of Relationships
    The trinity – one to one, one to many, many to many
        One to one
            What exactly happened?
        One to many
        Many to many
        A question of inverses
    Querying-related models
        Accessing a pivot table
        Querying a relationship
    Eager loading (and the N + 1 problem)
        Basic eager loading
        Advanced eager loading

Lazy eager loading
Inserting and updating related models
The save() and associate() methods
What about many to many?
The sync() method
Accessing distant relationships
More power – polymorphic relationships
A simple polymorphic relationship
A many-to-many polymorphic relationship
Summary
5. Using Collections to Enhance Results
Basic collection operations
Transforming collections
Iterating and filtering
Iterating
Filtering
Sorting
Summary
6. Everything under Control with Events and Observers
When should I use events in my models?
Model events
An example of model events
Events observers
An example of model observers
Summary
7. Eloquent… without Laravel!
Exploring the directory structure
Installing and configuring the database package
Installing the package
Configuring the package
Using the ORM
Using the Query and Schema Builder
The Query Builder
The Schema Builder
Summary
8. It's Not Enough! Extending Eloquent, Advanced Concepts
Extending the Model: Aweloquent!
The Aweloquent Model
Auto Hydrate
Model self-validation
Smart password hashing
The autopurge of confirmation fields
Extending the class
The Auto Hydrate feature
The Aweloquent Model self-validation feature – the basic version
The Aweloquent Model self-validation feature – the operation-based version

Smart password hashing and the confirmation fields autopurge method
Fixing the save() Model method
Diving into the repository pattern
Hello, repository pattern!
Introducing repositories – a concrete implementation
Coding on Abstractions
Repositories – a complete implementation
Adding the new repository
Summary
Index

Learning Laravel's Eloquent

Learning Laravel's Eloquent

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: July 2015

Production reference: 1210715

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-158-4

www.packtpub.com

Credits

Author

Francesco Malatesta

Reviewers

Jeff Madsen

Ângelo Marcos Rigo

Commissioning Editor

Ashwin Nair

Acquisition Editors

Nikhil Karkal

Rebecca Youe

Content Development Editor

Ritika Singh

Technical Editors

Edwin Moses

Tejaswita Karvir

Copy Editor

Dipti Mankame

Project Coordinator

Judie Jose

Proofreader

Safis Editing

Indexer

Hemangini Bari

Graphics

Sheetal Aute

Production Coordinator

Nitesh Thakur

Cover Work

Nitesh Thakur

About the Author

Francesco Malatesta, born in 1990, is a web developer and a curious enthusiast from Vasto, a wonderful city on the Italian eastern coast. He actually lives in Rome, where he is studying computer engineering. He received his first PC at the age of six, and since then, they have never separated. It's not all about computers, however; he also likes to travel, add new items to his movie collection, and create new awesome experiences. This book is a perfect example.

"Never stop" is his key.

He is the founder of Laravel-Italia, the Italian Laravel community. He works as a freelancer developer and consultant, but he also writes for Sitepoint, in the PHP section, and HTML.IT, the first portal about information technology in Italy. He started his first job in web development when he was 15 years old.

In the past, he has translated three books, which are *Laravel: Code Happy* (by Dayle Rees), *Code Bright* (by Dayle Rees), and *Laravel Testing: Decoded* (by Jeffrey Way).

Acknowledgments

There are many people that I would like to thank. Without them, what you are actually reading would not have been possible.

First of all, I would like to thank my parents, Paolo and Cinzia. Their efforts, teachings, and support are absolutely precious and invaluable.

Also, I would like to thank the awesome Rebecca Youe and the adorable Ritika Singh. They followed me over the entire process with much advice and feedback. I never felt alone and learned many things about writing. If you end up reading something cool, you have to know that these two fantastic ladies have given a fantastic contribution.

Finally, last but not least, I would like to thank a really important person. Words can make a good book, sometimes a perfect one. However, in some cases, they are not enough. Someday, maybe, I will find the right ones.

Until then… thank you, Lavinia. This book, my first one, is dedicated to you.

About the Reviewers

Jeff Madsen has been programming for over 15 years as both a database programmer and web application developer. He is especially active in the Laravel and Codeigniter communities. You can read some of his tutorials on a variety of topics at http://codebyjeff.com or reach him on Twitter at `@codebyjeff`.

Ângelo Marcos Rigo has a strong background in web development, focusing on content management systems. In the last 7 years, he has also worked on managing, customizing, and developing extensions for Moodle LMS. He can be reached on his website http://www.u4w.com.br for consulting.

He has previously worked on a Moodle Security book.

I would like to thank my wife, Janaina de Souza, and my daugther, Lorena Rigo, for their support while I was away reviewing this book.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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 at www.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.



https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Preface

If you are associated with the field of web development, you know how important data is. The web runs on data, so it's essential for developers to think of quick and effective ways to deal with it. Eloquent is an awesome ORM that comes with the Laravel PHP framework. It is unique and is very beneficial to developers as it allows them to define models, relationships, and many complex operations with a really easy and intuitive syntax, without sacrificing performance. Performing an interesting number of operations on multiple tables without writing long queries with objects will be a bed of roses.

This book will take you through developing brilliant data-based applications with Eloquent, the Laravel framework ORM.

You will do the following:

- Build highly efficient applications with the Eloquent ORM using an expressive syntax
- Get to grips with the power of relationships and how Eloquent handles them
- Go beyond simple theory with various step-by-step code examples

So, let's get started!

What this book covers

Chapter 1, *Setting Up Our First Project*, will discuss how to deal with Composer and Homestead. We will also cover the installation process of our very first Laravel project.

Chapter 2, *Building the Database with the Schema Builder Class*, will discuss the Schema Builder Class. We will analyze everything you can do with the class, look at different types of indexing, and learn about the methods that the Schema class provides.

Chapter 3, *The Most Important Element – the Model!*, will help us implement some "create," "read," "update," and "delete" logic for our items. We will also explore some useful methods and features of the model class.

Chapter 4, *Exploring the World of Relationships*, will help us discover how to work with different types of relationships and how to query and use them in a comfortable and clean way. Also, we will learn how to insert and delete related models in our database, or update existing ones.

Chapter 5, *Using Collections to Enhance Results*, will talk about collections. We will work with some results transformation methods and with the elements that make up a collection.

Chapter 6, *Everything under Control with Events and Observers*, will allow us to learn everything about the events in the context of Eloquent models. Right after, we will cover model events and model observers.

Chapter 7, *Eloquent… without Laravel!*, will explore the structure of the database package and see what is inside it. After that, we will learn how to install the "illuminate/database" package separately for your project and how to configure it for its first use. Yes, exactly: Eloquent without Laravel!

Chapter 8, *It's Not Enough! Extending Eloquent, Advanced Concepts*, will explore two different ways to extend Eloquent, and move on to learn about the Repository pattern.

What you need for this book

This book has no specific requests about hardware or software; we will use Vagrant to set up a standard virtual machine as a foundation for the example projects and code snippets. So, don't worry, my friend! No long afternoons or nights will be passed on the Apache or Nginx configuration files.

Who this book is for

This book is perfect for developers with a basic knowledge of PHP development, but who are new to the Eloquent ORM. However, developers with previous Laravel and Eloquent experience will also benefit from the in-depth analysis of specific classes and methodologies in the book.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "In the `EventServiceProvider` class, you can add a special event listener and bind it to a certain closure."

A block of code is set as follows:

```
    User::saved(function($user)
      {
          // doing something here, after User save
operation (both create and update)...
      });
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "All you have to do is to go on the **Download** page of the Composer website and find the right method for your operating system."

> ### Note
>
> Warnings or important notes appear in a box like this.

> ### Tip
>
> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Setting Up Our First Project

"Chi ben comincia è a metà dell'opera."

(Italian for "Well begun is half done.")

Every journey has a beginning, and only a hero with the right equipment can attain a victory. Of course, there are no exceptions for the 21st century hero: the developer!

In order to avoid problems and fight the bad (and malfunctioning) code monster, the good code artisan will prepare everything necessary before the start.

A developer has to be comfortable with the tools they are going to use, and a good development environment can awesomely improve the process. So, before getting our hands dirty, in this chapter, we will discover how to deal with Composer and Homestead.

Composer is an awesome dependency management tool, which is used by many PHP projects around the world. Homestead is the official Laravel Vagrant box that lets you create a fully functional development environment on a dedicated virtual machine in a matter of minutes. Finally, we will cover the installation process of our very first Laravel project.

I know what you are thinking: you just want to write code, code, and more code.

Be patient for a little while: if you know the tools we are going to analyze, at the end of this chapter, you will feel an enormous difference.

Trust me.

- Your Swiss Army Knife: Composer
- Your safe place: Homestead
- The new hideout: Homestead improved
- A bonus tool: Adminer
- Your best friend: Laravel
- Your first project: EloquentJourney
- Summary

Your Swiss Army Knife – Composer

The very first thing you will need to work with Laravel (and then Eloquent) is Composer. Composer is a dependency management tool for PHP. With this tool, you can easily include every dependency that is needed in your project. This is done in seconds, using a JSON configuration file named `composer.json`.

Usually, dependencies in a PHP project were managed with PEAR or other methods. Composer has a different policy: everything works on a per-project basis. This means that you can have two projects on the same server with different versions of the same dependency package.

Installing Composer

The installation procedure is ridiculously easy. All you have to do, is to go on the **Download** page of the Composer website and find the right method for your operating system.

- If you have Linux or Mac, just use this:

```
curl -sS https://getcomposer.org/installer |
php
```

  Or, if you don't have cURL, then use this:

```
php -r
"readfile('https://getcomposer.org/installer');"
| php
```

  Also, the `composer.phar` file will be downloaded in your current directory.

- On Windows, you can simply download the dedicated installer.

Once Composer is installed, I suggest putting its path in the `PATH` variable of your system, in order to use it wherever you want. There are many ways to do it, which depend on your operating system. Let's look at each.

- On Linux, you can move Composer to the right directory simply with the following command:

```
mv composer.phar /usr/local/bin/composer
```

- The same goes for OS X, but sometimes, the `usr` directory doesn't exist. You must create `usr/local/bin` manually.
- Finally, on Windows, you must open the control panel and type `environment variable` or something similar. The search utility will do the rest for you. Once in the right window, you will get a list of all environment variables. Find `PATH` and add the composer installation path to it.

The composer.json and autoload files

Before we go deep into our project, let's take a look at how Composer works.

In the `composer.json` file, the developer specifies every single dependency for its project. You can also create your packages, but we are not going to look at how to create them in this book.

So, let's say that you want to create a project that uses Monolog for logging purposes.

1. Create a folder for the project, then create an empty text file, and name it `composer.json`.
2. Open it and all you will have to do is to include your dependency as shown:

```
{
    "require": {
        "monolog/monolog": "1.12.0"
    }
}
```

3. After that, save the file and type the following in your project directory:

```
composer update
```

Wait a minute to download everything, and then you are done!

What? OK, here is how it works: Composer downloads every package you may need and automatically creates a loader for all your packages. So, to use your dependencies in your project, you will just need to include `vendor/autoload.php`, and you are good to go.

Let's say that you have an `index.php` file as a start file for your application. You will have to perform something like the following:

```
<?php // index.php file

  require('vendor/autoload.php');
```

```
// your code here...
```

Nothing more!

Why am I showing this to you? Well, Laravel and Eloquent are Composer packages. So, in order to use it and create a Laravel application, you have to know how the mechanism works!

The most used commands

Composer is a command-line tool. Every good CLI tool has some important commands, and in this little section, I will show you what we are going to use the most.

- First of all, we have the following:

```
composer create-project
```

  With this command, you can create a new project using a specific package as a base. You will use this command to create a new Laravel project using the following syntax:

```
composer create-project laravel/laravel
my_project_folder
```

- Then, you can find:

```
composer install
composer update
```

  These are two similar commands; they are similar, but not the same. When you specify your dependencies in the `composer.json` file, you can use `install` to install them. If you already installed them but you want to update your dependencies to a newer version, use `update`.

> **Note**
>
> In order to know what must and must not be updated, Composer uses the `composer.lock` file, which you can see in the root of your project. Actually, you will never have to work with it, but it's important to know that Composer uses it as a *log* of what it does.

- Sometimes, you will also see this:

```
composer require
```

You can use `require` to include dependencies in your project on the fly. Here's an example of Monolog inclusion using `require`:

```
composer.phar require monolog/monolog:1.12.0
```

- Another often used command is:

```
composer dump-autoload
```

This command regenerates the `autoload.php` file. It can be useful if you add some classes into your projects without using namespaces or PSR conventions and rules.

- Sometimes, you will have to use (after a warning):

```
composer self-update
```

This command updates Composer itself. Just a few seconds, and you are up and running again!

- Finally, you can use the following special command:

```
composer global COMMAND_HERE
```

Use it to execute a specific command in the Composer home directory. As I mentioned before, Composer works on a per-project basis, but sometimes you will need to install some tools globally. With the global command, you can do it easily.

That's all you need to know about Composer right now, and yes, there are many other commands, but we don't need them now.

Let's take a step forward: it's time to learn about Homestead!

Your safe place – Homestead

When we start a new project, we might also stumble upon many compatibility and environment issues. The first one to think about is the PHP version. Maybe you are using XAMPP or some preconfigured stack on your local machine. For your new project, you want to use PHP 5.6, but the installed version is 5.3 (as you used it for some older projects). Fine, no problem; you can just install 5.6, and you are good to go.

Yes, but after two days, the phone rings. It's your *old* customer; finally, it's time to make some improvements and add new features! So, you start your stack services, browse your old project index, and BOOM! Compatibility issues, compatibility issues everywhere! Not exactly the best way to start your day.

This is not a code problem, but an environment problem.

Actually, the best solution is to start using Vagrant. Vagrant is a fantastic tool that lets you create a virtual machine with a headless operating system in order to configure the virtual machine on a per-project basis. Also, you can share some folders from your local machine with that machine, so you can work on an isolated environment while working with your favorite IDE and operating system.

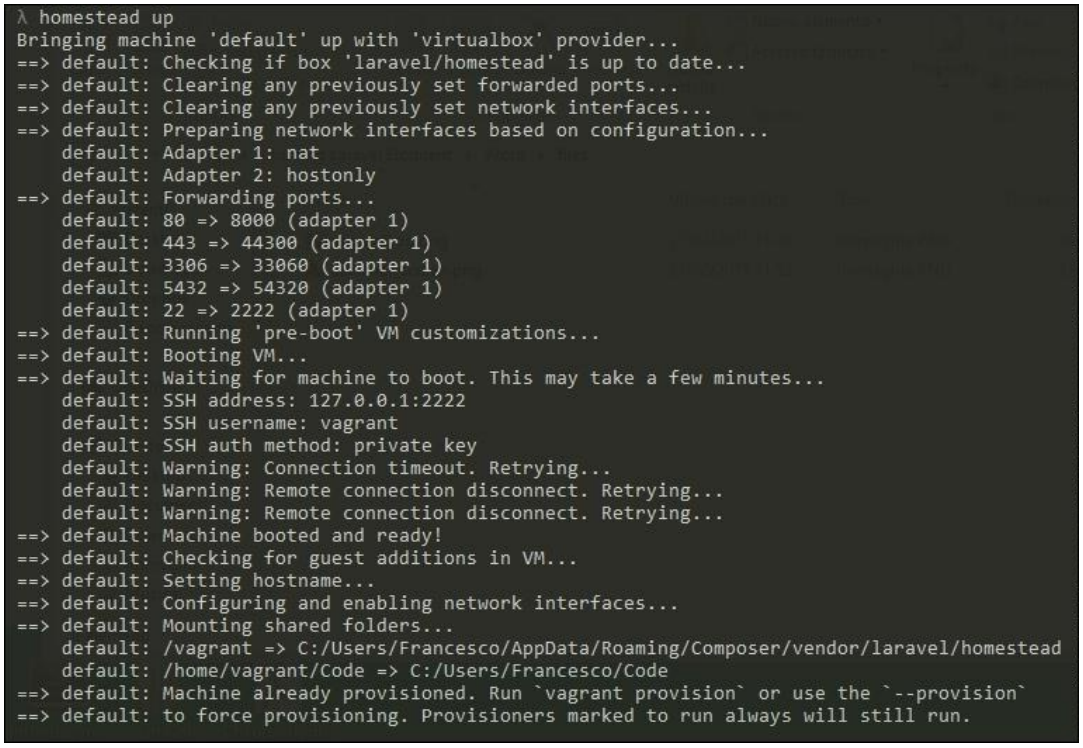
> **Note**
>
> Note that the per-project basis is the most important part of the entire thing. If you configure a separate machine for a single project, you can tweak everything you want to reach the perfect environment. Also, with Vagrant, you will be able to set your local environment in the same way your production machine is configured. So, no more *local to production* bugs and issues!

Last but not least, the fun (and useful) thing about Vagrant is that you can put a specific box under version control. So, for every new team member, all you have to do is to clone the repository and start the machine.

This looks complicated, but it is not. With Vagrant you can easily download a box (a ready-to-use virtual machine with all the tools and applications you need) and start it with a simple command from the shell as shown:

```
$ vagrant up
```

The Laravel community knows a couple of things about Vagrant and makes up a Vagrant Box to help you in your job.



Homestead is the official Vagrant Box for Laravel and already has everything you need to get started. You will find it (already installed and working), by default:

Ubuntu 14.04	**Node (with Bower, Grunt, and Gulp)**
PHP 5.6	Redis
HHVM	Memcached
nginx	Beanstalkd
MySQL	Laravel Envoy

Ubuntu 14.04	**Node (with Bower, Grunt, and Gulp)**
PostgreSQL	Fabric + Hipchat Extension

Not too bad for a tool box that you can prepare in a matter of minutes!

Now let's stop discussing and install Homestead.

Installing Homestead

First of all, ensure that you have already installed VirtualBox (https://www.virtualbox.org/) and Vagrant (https://www.vagrantup.com/). You can install them on every operating system, so feel free to use whichever you want.

> **Tip**
>
> If you want to work with a good shell on Windows, I suggest you use Cmder (http://bliker.github.io/cmder/). While writing this book, I referred to the same link.

Next, we can add Homestead to our local boxes. This means that Vagrant will download the Homestead box in order to be used locally.

You can do it with a simple command:

```
vagrant box add laravel/homestead
```

You will have to wait a couple of minutes to download the box. So, if you want to have a coffee, this is the perfect moment.

> **Note**
>
> Here, you don't have to worry about where Vagrant is placing the box, as it is going to save it locally in the Vagrant folder. In the future, every time you will need a specific box, Vagrant will clone and use it.

Alright, your box is now on your local machine and ready to be started. However, accordingly to your local machine settings, you can install

Homestead in two different ways. They are both present in the official Laravel documentation, so they are both *official*.

The Composer and PHP tool way

Let's start with the first one: it is a perfect choice if you already have Composer and PHP on your local machine. Note that you are only going to do these steps the first time.

Use this command to install the Homestead CLI tool.

```
composer global require "laravel/homestead=~2.0"
```

Then, be sure to put the `~/.composer/vendor/bin` directory in the `PATH` environment variable, in order to use the tool wherever you want.

After that, you can initialize your machine. Use the `init` command:

```
homestead init
```

This will create a `~/.homestead` folder with a `Homestead.yaml` inside of it. This file will be used by Vagrant at the virtual machine start.



The Git way

If you don't have PHP and Composer installed on your local machine (or maybe you just don't want to use them), no problem. You can simply use Git.

Choose a folder where you want to save your virtual machine. Then, clone the repository with:

```
git clone
https://github.com/laravel/homestead.git
HomesteadFolder
```

Here, `HomesteadFolder` is the place you chose for your VM files. After the clone process, use `cd` to get into the folder and start the `init` script using the following command:

```
bash init.sh
```

This script will create a `Homestead.yaml` file in a `~/.homestead` directory, and that's it!

The following steps for installation are the same for both the methods you just saw.

Configuring Homestead

Before we go forward, let's take a look at the default `Homestead.yaml` file.

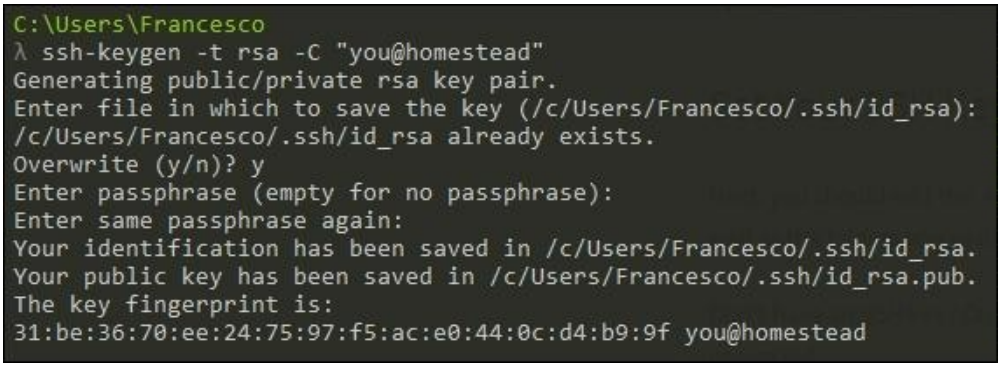
```
---
ip: "192.168.10.10"
memory: 2048
cpus: 1

authorize: ~/.ssh/id_rsa.pub

keys:
    - ~/.ssh/id_rsa

folders:
    - map: ~/Code
      to: /home/vagrant/Code

sites:
    - map: homestead.app
      to: /home/vagrant/Code/Laravel/public

databases:
    - homestead

variables:
    - key: APP_ENV
      value: local
```

If you are unfamiliar with this syntax, no problem; it's a simple YAML (YAML *ain't* a markup language) markup file. It is a very readable way to specify settings, and Homestead uses it. Here, you can choose the IP address for your virtual machine and other settings. Tweak the configuration file accordingly to your needs.

1. Do you see the `authorize` property in the `Homestead.yaml` file? Well, we are going to set up our SSH key and put its path there. If it scares you, don't worry; it is just a command.

```
ssh-keygen -t rsa -C "you@homestead"
```

   If you are using Windows, the Laravel documentation recommends Git Bash. Personally, as I mentioned before, I prefer to use Cmder. However, you can also use PuTTY or whatever you want. Use `ssh-keygen -t rsa -C "you@homestead"` to generate your ssh key. This is shown in the following screenshot:



2. Put the generated SSH key path in the `authorize` property of `Homestead.yaml`, as shown in the following:

```
authorize: ~/.ssh/id_rsa.pub
```

   Done? Good. Now, you can see a `folders` property as well.

> **Note**
>
> As I mentioned before, Vagrant lets the developer share some folders between the local and virtual machines.
>
> What is the point of that? Well, it is really important because with this system we can work on our project on a separate machine, while being able to use whatever IDE or tool we want from our local machine. For example, even if the VM has Ubuntu, I can easily use Windows 8.1 and PHPStorm. The best of both worlds!

3. By default, Homestead suggests this structure:

```
folders:
      - map: ~/Code
```

```
    to: /home/vagrant/Code
```

Also, this means that you will have to create a `Code` folder in your user folder. This local folder will be mapped to a `/home/vagrant/Code` folder on the VM; every change that you make there will be reflected on the virtual machine and vice versa.

> **Note**
>
> You can customize this mapping to your needs.

4. Next, let's take a look at the `sites` property. Here's what you can see in a default setup:

```
sites:
    - map: homestead.app
      to: /home/vagrant/Code/Laravel/public
```

You can define a custom domain for every project, which is a really comfortable way to work with your projects, as you will no longer need to test your project with an IP (like 192.168.10.10), only a simple local domain, such as `myproject.dev`.

5. This is a good point to define a separate site for our project. So, feel free to add these lines to your file:

```
- map: eloquent.dev

to: /home/vagrant/Code/EloquentJourney/public
```

6. Next, go to your host's file (on the host machine) and add this record:

```
192.168.10.10 eloquent.dev
```

You can see how you need to add it in the following screenshot:

```
# be placed in the first column followed by the corresponding host name.
# The IP address and the host name should be separated by at least one
# space.
#
# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#      102.54.94.97     rhino.acme.com          # source server
#       38.25.63.10     x.acme.com              # x client host

# localhost name resolution is handled within DNS itself.
#       127.0.0.1       localhost
#       ::1             localhost

192.168.10.15 eloquent.dev
```

> **Note**
>
> Of course, you have to insert the same IP you specified in the `Homestead.yaml` file.

7. The last thing we are going to see here is the `database` property. For every name you add here, Homestead will automatically create a database to work with. So, edit the property to something like this:

```
databases:
    - homestead
    - eloquent_journey
```

This is because we are going to use a separate `eloquent_journey` MySQL database for our test application.

> **Note**
>
> The default username for the MySQL server is `homestead`, and the default password is `secret`.

We have nothing more to do here; our setup is complete, and now we are ready to boot up our virtual machine and use it.

The new hideout: Homestead improved

Even if Homestead is a fantastic box, many people complain about some of its structural *choices*. As I mentioned previously, Vagrant is used to create virtual machines on a per-project basis. This means that, in an ideal situation, every project must have its own VM. Now, with Homestead, you can create a single VM and manage all your projects on it. Some people like this idea, and it is more familiar to the classical XAMPP approach. Quite familiar!

However, other people like a more *pure* approach to Vagrant. While doing some researches on this concept, I stumbled upon *Homestead Improved* (https://github.com/Swader/homestead_improved) by *Swader*, on GitHub.

It is an improved version of Homestead that you can install and run without saving files all around your user folder. A really good approach! Also, you won't have to configure any SSH keys or execute `apt-get update` and `composer auto-update`. Everything will be done automatically.

If you want to use Homestead Improved, just open your terminal and type the following:

```
git clone
https://github.com/Swader/homestead_improved.git
MyHomesteadImprovedVM
```

Here, `MyHomesteadImprovedVM` will be the containing folder of all the VM's files.

After the clone procedure, just type the following:

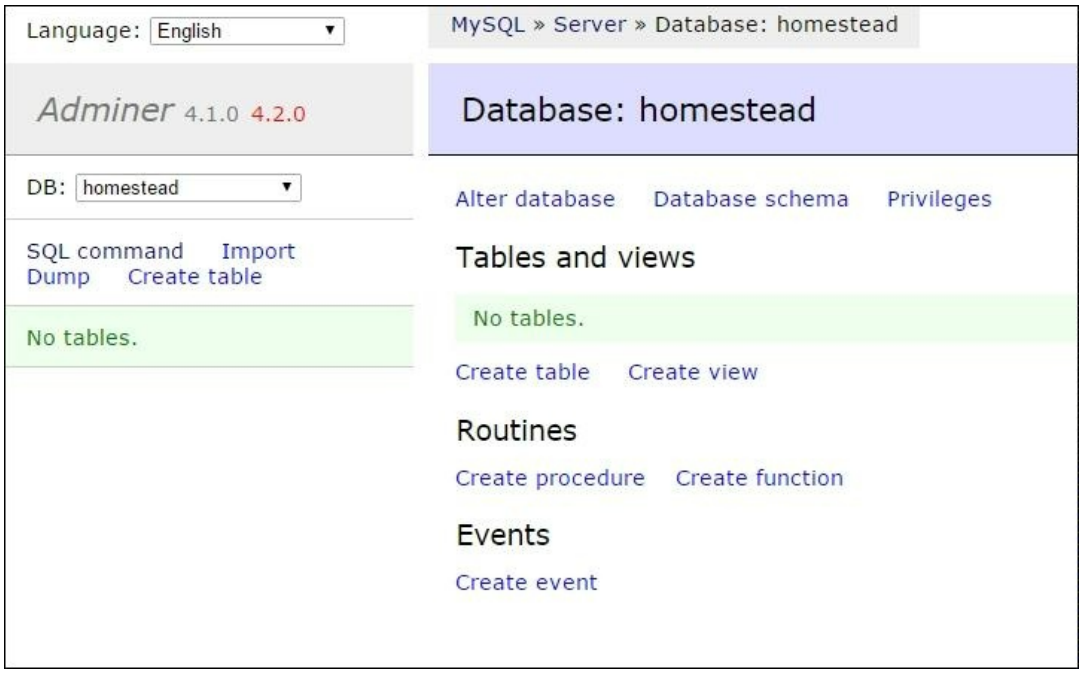
```
vagrant up
```

So, you're done! Easier than before, isn't it?

A bonus tool – Adminer

Before going deeper along our journey, there is another really useful tool that I want to show you. I am talking about Adminer, a Database Management tool entirely contained in a single `.php` file. You can download it at http://www.adminer.org/.

Maybe you will find the Adminer interface very similar to the phpMyAdmin interface. It's true, but Adminer has more features. Just to make a simple example, phpMyAdmin only supports MySQL. Instead, Adminer supports MySQL, PostgreSQL, SQLite, Oracle, and MS SQL.



Obviously, you can use whatever you want to deal with your database. However, I wanted to show you Adminer because it is what I am going to use to show, from time to time, some query results or various examples. So, it would be good if you get more familiar with this tool.

Your best friend: Laravel

We are close to the end. You have a weapon (Composer) and a safe place to do everything you want without worrying about issues (Homestead). What about an ally? Laravel could be a good one, don't you think? Also, Laravel is the Eloquent container: we are going to create a new project with it to fully embrace its power.

Installing Laravel

Before going further, remember that Laravel has some prerequisites. You will need the following:

- PHP 5.4 (or more recent versions)
- The PHP Mcrypt extension
- The PHP OpenSSL extension
- The PHP Mbstring extension

> **Note**
>
> If you are using PHP 5.5, you may need to install the JSON PHP extension. If this is the case, just type this:
>
> ```
> apt-get install php5-json
> ```
>
> So, you are good to go.

Obviously, if you have installed Homestead, everything is already in its right place.

1. All you have to do is to boot up the VM with the following command:

   ```
   homestead up
   ```

2. And when the bootstrap procedure is done, use the following command to get in the machine via SSH:

   ```
   homestead ssh
   ```

Having said that, as you may have experienced from Homestead, Laravel also gives you two different ways to install it and create a new project.

- The first one is done using a specific tool, the Laravel installer tool. It's a CLI tool that you can install as a global Composer package.
- The second one is a simple `composer create-project` command. Of course, we will now see both ways.

Using the Laravel installer tool

The Laravel installer tool is a nice utility that lets you create a new Laravel project with a very simple syntax. Imagine that you want to create a new project in a folder called `my_project`. All you have to do, if you have the tool installed, is to type this and nothing more:

```
laravel new my_project
```

Installing the tool is easy. Just open the terminal and type the following:

```
composer global require "laravel/installer=~1.1"
```

As you saw before, we are executing the `require` command with the `global` keyword. This means that the installer tool package will be saved in the Composer's `global` folder and the tool will be available everywhere.

> **Note**
>
> If you have any problems running the tool, just be sure to put `~/.composer/vendor/bin` in the `PATH` environment variable. Otherwise, it won't work!

Using the Composer create-project command

If you don't want to install the Laravel installer tool, you can simply use the `create-project` command of Composer.

All you have to do, in this case, is use this command:

```
composer create-project laravel/laravel
ProjectName
```

Here, `ProjectName` stands for the folder name that you want to use as the root of your new Laravel project.

Nothing more to do here! Your Laravel project is now fully installed in your specified folder.

Note

Be sure to configure the right permissions on your folders and ensure you take a good look at the URL rewriting rules. If you take a look at the Laravel-dedicated documentation page (http://laravel.com/docs/5.0/installation#pretty-urls), you can learn how to do it on Apache or nginx.

The first project – EloquentJourney

A new project will be the perfect metaphor for our new, fantastic journey! While studying Eloquent, we will build a simple project. More specifically, we will analyze a hypothetical library management system's data-related part and its components.

What are you waiting for? Let's start! First of all, create a new project (using your favorite method). We will call our new project `EloquentJourney`. Type the following in your server folder:

```
laravel new EloquentJourney
```

Otherwise, type the following if you prefer:

```
composer create-project laravel/laravel
EloquentJourney
```

Wait a few seconds to build the project, and after the installation procedure, you are done! You can use `cd` to get into your new folder and see what's there.

Cool! All right, but what are we going to do now? There are thousands of files here and in other subfolders! Don't worry. Take a breath and follow me. First of all, we need to do some practice with the Laravel configuration system in order to set up an appropriate database connection.

Without it, we could not use Eloquent!

The configuration system

Everything you could need on configuration is stored in the `config` directory. Every file here has quite a descriptive name: `app.php`, `database.php`, `filesystems.php`, `cache.php`, and so on. Actually, we are going to use two of these files: `app.php`, for some basic settings, and `database.php`, for obvious reasons.

First of all, let's open the `app.php` file and see what you can find inside.

```
<?php

return [
```

```
    'debug' => true,

    'url' => 'http://localhost',

    'timezone' => 'Europe/Rome',

    'locale' => 'en',

    'fallback_locale' => 'en',

    'key' => env('APP_KEY', 'SomeRandomString'),

    'cipher' => MCRYPT_RIJNDAEL_128,

    'log' => 'daily',

    // other items here...

];
```

A Laravel config file contains a return instruction. The returned value is an associative array. As you can easily imagine, the key-value system represents the configuration item name and its value. For example, let's examine the first item:

```
    'debug' => true,
```

This means that the `app.debug` configuration item is set on the Boolean value true.

Laravel uses these values all around the framework code, and you can use them too with the `\Config` class.

Specifically, if you want to retrieve a specific item value, you have to call the `get()` method, as follows:

```
$myItem = \Config::get('item.name');
var_dump($myItem);

// true
```

You can also set a specific `Config` value at runtime, this time using the `set()` method, as follows:

```
\Config::set('item.name', 'my value!');

$myItem = \Config::get('item.name');
var_dump($myItem);
```

```
// "my value!"
```

Setting up the database connection

Yes, we finally arrived at the end of this chapter. The last thing we need to do here, is to set up the database connection.

Let's open the `database.php` file under `config`. You should see something like this:

```
<?php

return [

  'fetch' => PDO::FETCH_CLASS,

  'default' => 'mysql',

  'connections' => [

    'sqlite' => [
      'driver'   => 'sqlite',
      'database' =>
storage_path().'/database.sqlite',
      'prefix'   => '',
    ],

    'mysql' => [
      'driver'    => 'mysql',
      'host'      => 'localhost',
      'database'  => 'homestead',
      'username'  => 'homestead',
      'password'  => 'secret',
      'charset'   => 'utf8',
      'collation' => 'utf8_unicode_ci',
      'prefix'    => '',
      'strict'    => false,
    ],

    'pgsql' => [
      'driver'   => 'pgsql',
      'host'     => env('DB_HOST', 'localhost'),
      'database' => env('DB_DATABASE', 'forge'),
      'username' => env('DB_USERNAME', 'forge'),
      'password' => env('DB_PASSWORD', ''),
      'charset'  => 'utf8',
      'prefix'   => '',
      'schema'   => 'public',
    ],

    'sqlsrv' => [
      'driver'   => 'sqlsrv',
      'host'     => env('DB_HOST', 'localhost'),
```

```
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'prefix'   => '',
        ],

    ],

    'migrations' => 'migrations',

    'redis' => [

        'cluster' => false,

        'default' => [
            'host'     => '127.0.0.1',
            'port'     => 6379,
            'database' => 0,
        ],

    ],

];
```

The two most important items are `default` and `connections`. In this second item, `connections`, we are storing all the information we need to connect to our databases. By default, you will find many examples. In fact, here you can see the `sqlite`, then `mysql`, and also `sqlsrv` connections.

Every connection has a driver. The `driver` element indicates the used database for that connection. If you want, you can specify more than one connection, when necessary. The `default` element represents the chosen connection.

Let's delete everything and replace the `default` and `connections` elements with the following:

```
    'default' => 'eloquentJourney',

    'connections' => [

        'eloquentJourney' => [
            'driver'    => 'mysql',
            'host'      => 'localhost',
            'database'  => 'eloquent_journey',
            'username'  => 'homestead',
            'password'  => 'secret',
            'charset'   => 'utf8',
            'collation' => 'utf8_unicode_ci',
            'prefix'    => '',
            'strict'    => false,
        ],
```

```
        ],
```

What did we just do?

Quite simple! We have defined an `eloquentJourney` connection. This connection will use the `mysql` driver. So, we are going to connect Laravel to a MySQL server. I am not going to explain the other properties, as it is really easy to understand their meanings.

After that, we specified the connection name as the default option. This means that, for every future call to a database-related operation, Laravel will connect to the server specified in the `eloquentJourney` connection with the given credentials.

Summary

We did it!

We prepared everything that is needed to work with Laravel and Eloquent. We set up a local development server, learned the basics of Composer to correctly manage our dependencies, installed a couple of more useful tools, and, finally, successfully configured our database connection. Not bad for the first chapter, huh?

However, we have just begun, and our journey inside Eloquent is at the very beginning. We are ready to leave our safe house, go into the darkest corner of the Eloquent ORM to explore it, and understand all of its secrets.

It will be a great ride. And now, let's explore our first topic on our way: the Schema Builder and the migrations system, to build the perfect database!

Chapter 2. Building the Database with the Schema Builder Class

Hey hero!

Our development environment is now ready. No more worries, and a whole new world awaits: the hero left his home. Little by little, step by step, he is going forward toward the goal. The hero knows what is really important: to stay grounded. Having a good foundation helps. No big differences for an application: a well-designed database for your project is always the best start.

Starting from this important assumption, the question naturally arises: is there a way, with Laravel, to deal with the design of a database before using it, maybe in a smart way that you can easily manage? The answer is, as you can imagine, yes.

In this chapter, we will look at the Schema Builder class. A very important class that lets you design your entire database without writing a single SQL line! Quite impressive if you think about it, considering that it is quite possible that you will build your entire application on a SQL database without using SQL! We will analyze everything you can do with the class: creating, dropping, and updating a table; adding, removing, and renaming columns. Also, we will look at indexing in many ways: not only simple indexes but also unique indexes and foreign keys. Also, we are going to look at some methods that the Schema class provides in order to have a real and total control over everything. Sometimes, you will need to be sure about what exists in your database!

After all this stuff, it is not be over yet. In fact, we will explore the world of **migrations**, which is something Laravel uses to do versioning of your database. Combining the power of the Schema class with the **migrations system** will give you great power over your data design. Also, it will be very easy to share your application with another new member of your team! You know that versioning is always the best choice.

An important note before going forward: even if the Schema Builder class and migrations system are not tightly related to Eloquent, they easily create a database with a structure that fits perfectly for the Eloquent standards and conventions. Also, the Schema Builder and migrations system are a part of the `illuminate/database` (https://github.com/illuminate/database) package; yes, the same package from Eloquent! Let's go.

- The Schema Builder class
- Database versioning with the migrations system

The Schema Builder class

The Schema Builder class is a tool that provides you with a *database agnostic* way to deal with your database design. The term *database agnostic* means that you will never have to worry about what database you are using: the only important thing is to use the right driver, as we saw before in the configuration part of the first chapter. So, if you think of switching your database system from SQLite to MySQL or SQL Server, don't worry: you will always have your structure ready to be used.

In the first part of this chapter, we will use some simple routes. All you have to do is to add them to the `app/routes.php` file.

Working with tables

Let's start with the very, very basic phenomena. The following is the `Schema::create()` method that you can use to create a new table in your database:



The `create()` method accepts two parameters: the first method is the name of the table you want to create, and the second method is a closure where we will specify all the table fields. To be more precise, the closure parameter is a Blueprint object.

> **Tip**
>
> When possible, I like to use type-hinting for a better code readability.

So, you may read something like this:

```
Schema::create('users', function(Blueprint $table)
{
$table->increments('id');
});
```

Don't worry, it's the same.

You will also be able to rename a table: in this case, just use the `rename()` method.

```
Schema::rename($previousName, $newName);
```

Finally, what about dropping a table? The method `drop()` is here for you.

```
Schema::drop($tableName);
```

There is also a similar method, `dropIfExists()`, with the same syntax.

```
Schema::dropIfExists($tableName);
```

As the awesomely expressive name suggests, the method drops the table only if it exists in the database.

> **Tip**
>
> I often use this method instead of the simple `drop()` method.

Remember that if you want to check if a table exists, you can use the `hasTable()` method, as follows:

```
if (Schema::hasTable('books'))
{
  // the table "books" exists...
}
```

Now, let's take a look at how you can work with table columns.

Working with columns

Working with columns is quite easy. First of all, let's see how to create new columns while adding a new table to the database.

As I told you before, we will use the `$table` parameter of `Schema::create()`.

```
  Schema::create('books', function(Blueprint
$table)
  {
      $table->increments('id');
```

```
            $table->string('title');

            $table->integer('pages_count');
            $table->decimal('price', 5, 2);

            $table->text('description');

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

The `Blueprint $table` object has many methods, which are related to the creation of a single new column. You can easily see that all the methods' names represent a specific type of column. For example, `string` is the equivalent of `VARCHAR` on MySQL, and `integer` is the equivalent of `INT`.

There are also some utility methods: if you take a look at the very first code line of the closure, you will read an `increments()` method. Nothing complex, however, it just creates an `id` integer field (also setting it as a primary key) with `autoincrement`.

Finally, you can see the `timestamps()` method in the last line. The Schema Builder class also has methods that are related to some Eloquent functionalities. In this case, the `timestamps()` method automatically creates two `DATETIME` equivalent fields, named `created_at` and `updated_at`. They can be quite useful if you want to track the time of insert and last update operations.

Now, why don't we make some tests, before going further?

It's a matter of seconds: just create a new `GET` route in the `app/routes.php` file and add that `Schema::create()` method we saw like this:

```
    // remember this "use" directive!
    use Illuminate\Database\Schema\Blueprint;

    Route::get('create_books_table', function(){

      Schema::create('books', function(Blueprint
$table)
        {
            $table->increments('id');

            $table->string('title', 30);

            $table->integer('pages_count');
            $table->decimal('price', 5, 2);

            $table->text('description');

            $table->timestamps();
```

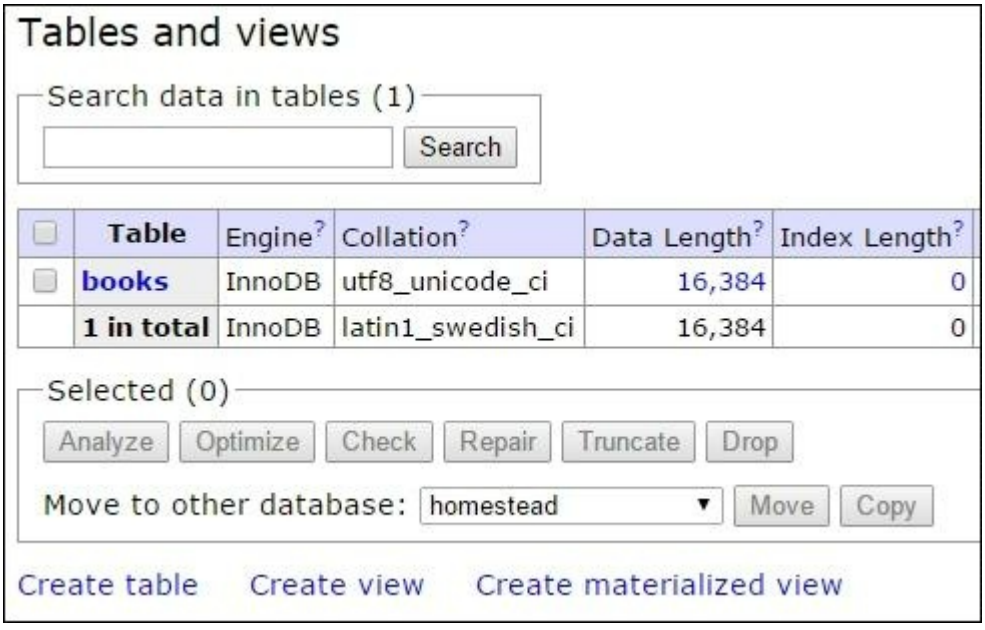
```
    });
});
```

We will create a table named `books` on our database. This table will have seven columns:

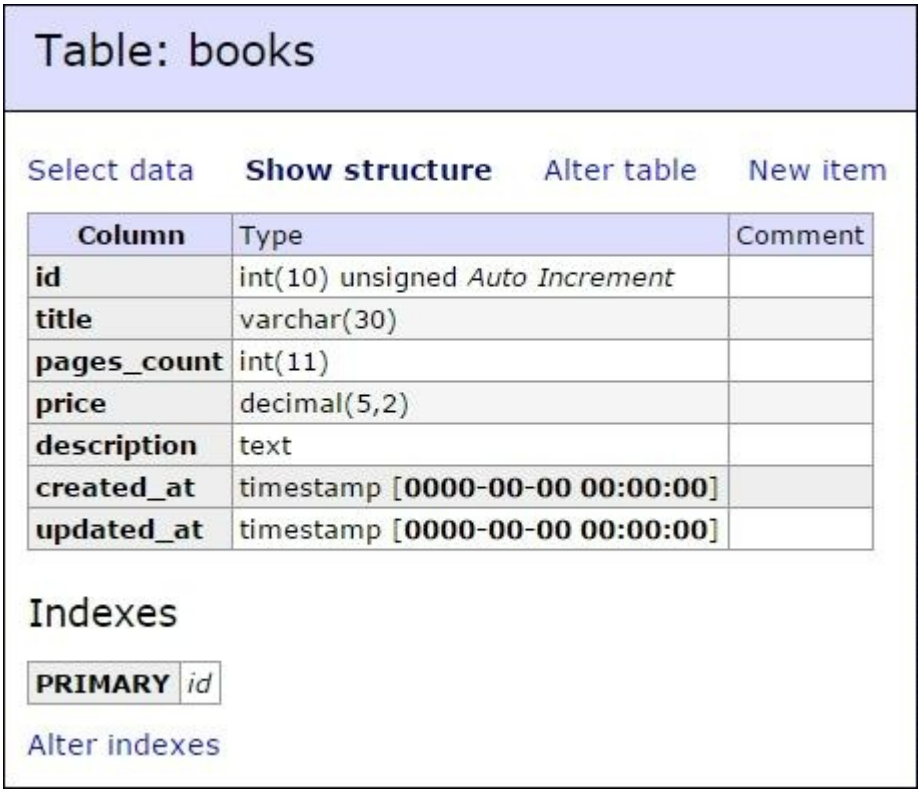
- The unique, primary key `id` as an integer
- The `title` column as a string (with 30 characters maximum)
- The `pages_count` integer column
- The `price` decimal column
- The `description` text column
- The `created_at` and `updated_at` columns, created by the `timestamps()` method

Let's navigate to the `/create_books_table` URL and then open your Adminer (or whatever tool you like to deal with your database) to verify if everything went well.

This is the result!



Good job! Let's take a look at the structure.



Perfect. Exactly what we wanted—without a single SQL line!

Note that as we had done before for tables, we can also check if a specific column exists with the `hasColumn()` method:

```
  if (Schema::hasColumn('books', 'title'))
  {
    // the column "title" in the "books" table
exists...
  }
```

Columns' methods reference

Here's a quick reference for all the methods you can use on the Blueprint `$table` instance:

Method	Description
`$table->bigIncrements('id');`	Incrementing ID using a *big integer* equivalent

Method	Description
`$table->bigInteger('votes');`	`BIGINT` equivalent to the table
`$table->binary('data');`	`BLOB` equivalent to the table
`$table->boolean('confirmed');`	`BOOLEAN` equivalent to the table
`$table->char('name', 4);`	`CHAR` equivalent with the length
`$table->date('created_at');`	`DATE` equivalent to the table
`$table->dateTime('created_at');`	`DATETIME` equivalent to the table
`$table->decimal('amount', 5, 2);`	`DECIMAL` equivalent with a precision and scale
`$table->double('column', 15, 8);`	`DOUBLE` equivalent with precision, 15 digits in total and 8 after the decimal point
`$table->enum('choices', ['foo', 'bar']);`	`ENUM` equivalent to the table
`$table->float('amount');`	`FLOAT` equivalent to the table
`$table->increments('id');`	Incrementing `ID` to the table (primary key)
`$table->integer('votes');`	`INTEGER` equivalent to the table

Method	Description
`$table->json('options');`	`JSON` equivalent to the table
`$table->longText('description');`	`LONGTEXT` equivalent to the table
`$table->mediumInteger('numbers');`	`MEDIUMINT` equivalent to the table
`$table->mediumText('description');`	`MEDIUMTEXT` equivalent to the table
`$table->nullableTimestamps();`	Same as `timestamps()`, except the fact that this allows NULLs
`$table->smallInteger('votes');`	`SMALLINT` equivalent to the table
`$table->tinyInteger('numbers');`	`TINYINT` equivalent to the table
`$table->string('email');`	`VARCHAR` equivalent column
`$table->string('name', 100);`	`VARCHAR` equivalent with the length of a string
`$table->text('description');`	`TEXT` equivalent to the table

Note

You can also find the complete reference in the Laravel official documentation, at the http://laravel.com/docs/5.0/schema#adding-columns page.

Other $table object methods

If you take a look at the methods reference in the Laravel documentation, you will probably see some more than in the list you just saw.

The first methods reference is:

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

I already used this in the example. It creates the table timestamps `created_at` and `updated_at` columns, which are really useful if you want to track chronological information about the creation or update of a specific record.

Eloquent also manages timestamps columns automatically, so you don't have to worry about them in your code.

Another important (and really similar) method is as follows:

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

It is used to add a `deleted_at` column to the desired table.

Sometimes, in your applications, it is really useful to keep some information even if you don't want to show it to the final user. Think about an e-commerce orders' history: the customer can choose to clean his history but you can't allow him/her to physically delete records!

The soft delete system solves this problem by adding a `deleted_at` column to a table in order to keep track of the delete date of a record.

Eloquent handles this system automatically, so you will be able to show a clean history to your customer and a complete list of orders to the shop administrator.

Next, you can see this:

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

This method adds a `remember_token` column.

The Laravel authentication system uses this token (a simple `VARCHAR 100`) to track the user status. It is used when you click on a **Remember Me** checkbox in the login page of your application.

There are also some methods you can chain after some table columns' definitions.

If you want to make a numeric column unsigned, use the following:

```
$table->integer('my_column')->unsigned();
```

Of course, you can use it with every numeric field (float, decimal, and so on).

You can also specify if a column is nullable with `nullable()`:

```
$table->string('my_column')->nullable();
```

If you want to specify a default value, you can use the following:

```
 $table->string('my_column')-
>default('my_default_value');
```

> **Tip**
>
> Sometimes, you will need to use `BIGINT` or equivalent for your primary keys. If this is the case, use `bigIncrements()` instead of `increments()` (that uses `INT` or equivalent).

Updating tables and columns

What? Yeah, I know what you are thinking—what if I have to add a book named The Awesome Super Life of the Marvelous Francesco Malatesta? It's a 60-character-long name!

Things change, so our database will have to change too. In a simple way, though. You will use the `table` method to update an existing one.

Here you can see an example:

```
  Schema::table('table_name', function(Blueprint
$table)
  {
    // update operations here...
```

```
});
```

So, let's make a little update. Create a new route in our routes file and name it `update_books_table`:

```
Route::get('update_books_table', function(){

    Schema::table('books', function(Blueprint
$table)
    {
        $table->string('title', 250)->change();
    });

});
```

Note that we used the `change()` method right after the `string()` method. You will use `change()` on changing a column. If you simply want to add a new column to an existing table, just use this as usual:

```
$table->string('title', 250);
```

> **Note**
>
> You will need the `doctrine/dbal` package to change columns. To obtain it, just add the `doctrine/dbal` to your `composer.json` file: 2.5.0 dependency.
>
> Then, type `composer update` in your terminal and you are good to go.

Browse to the new URL and update the table. Now you will be able to add my awesome biography to your library in the future. Be proud of it.

Indexes and foreign keys

Let's take another step forward. Let's imagine that we are upgrading our application and we want to add the possibility to store book authors and related data. Also, we will give the possibility to the user to search a book by its title.

We would have to do a couple of the following things:

- Create the `authors` table to store authors' data
- Update the `books` table to insert an external foreign key and an index to the `title` column

You already know how to update an existing table but what about indexing columns and creating foreign keys?

Adding indexes is quite easy with the Schema Builder. There are some expressive methods you can use.

First of all, you can create a primary key with the `primary()` method. The parameter is the field you want to index.

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

If you need, you can specify an array of columns as a single index.

```
$table->primary(['first_name', 'last_name',
'document_number']);
```

The method `unique()` is used to create a unique key:

```
$table->unique('email');
```

Finally, if you want to add a simple index, you can use the `index()` method!

```
$table->index('title');
```

You can also add a foreign key, if you need. In the following example, we are referencing the `author_id` column (on a hypothetical books table, maybe) to the `id` column in the `authors` table.

```
$table->foreign('author_id')->references('id')-
>on('authors');
```

Specifying `onDelete` and `onUpdate` constraint operations is not a problem.

```
$table->foreign('author_id')
    ->references('id')
    ->on('authors')
    ->onDelete('cascade');
```

If you don't need your indexes anymore, you can easily drop them:

```
$table->dropPrimary('authors_id_primary');
$table->dropUnique('authors_email_unique');
$table->dropIndex('books_title_index');
$table->dropForeign('books_author_id_foreign');
```

> **Note**
>
> The convention used to create an index name is `table-name_column-name_index-type`.

Alright, now that you have a good overview of indexes with the Schema Builder class, let's add the new functionality to our example.

Create a new `GET` route. This time it is named `update_books_table_2`.

```
  Route::get('update_books_table_2', function(){

      // creating the authors table...
      Schema::create('authors',
function(Blueprint $table)
      {
          $table->increments('id');

          $table->string('first_name');
          $table->string('last_name');

          $table->timestamps();
      });

      Schema::table('books', function(Blueprint
$table)
      {
          // creating the index on the title
column...
          $table->index('title');

          // creating the foreign key...

 $table->integer('author_id')->unsigned();

 $table->foreign('author_id')->references('id')-
>on('authors');
      });

  });
```

That's all! Let's verify all with Adminer, as we did before:

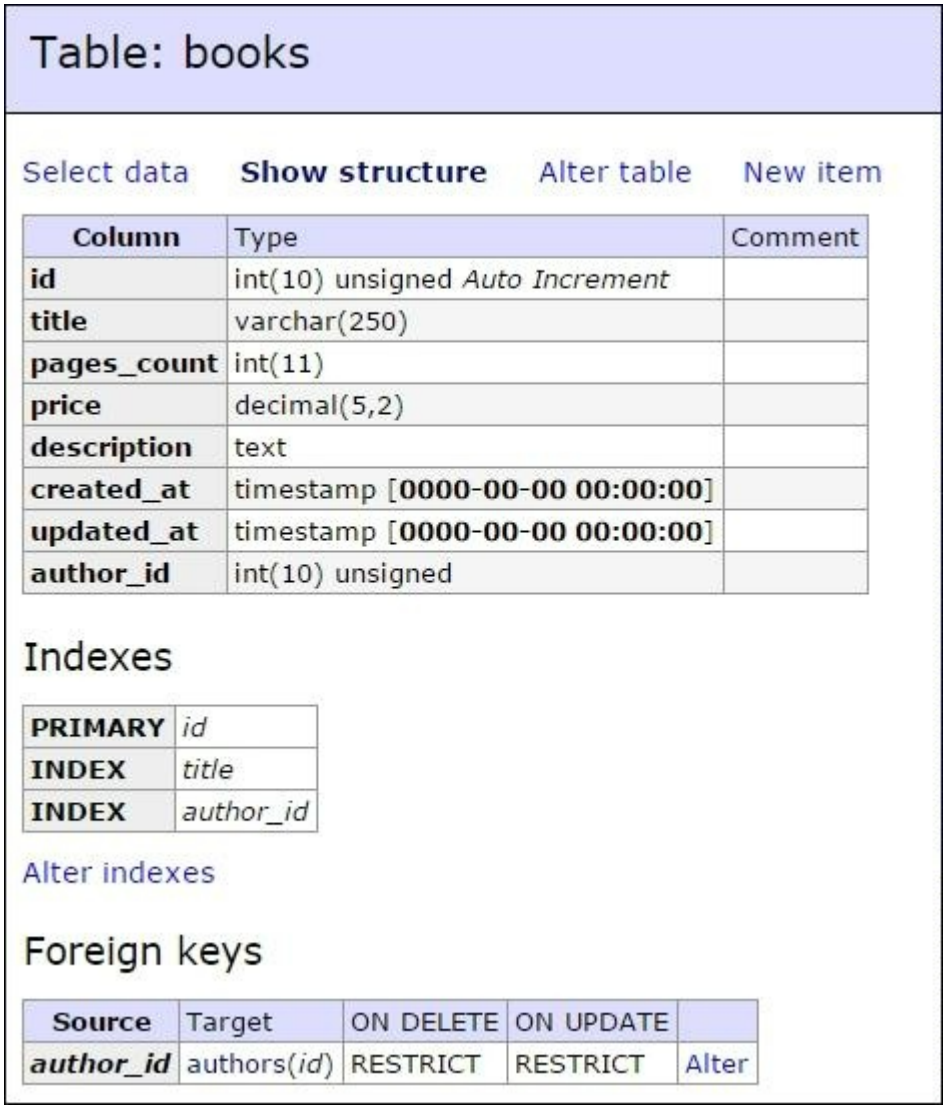


Table: books

Select data **Show structure** Alter table New item

Column	Type	Comment
id	int(10) unsigned *Auto Increment*	
title	varchar(250)	
pages_count	int(11)	
price	decimal(5,2)	
description	text	
created_at	timestamp [**0000-00-00 00:00:00**]	
updated_at	timestamp [**0000-00-00 00:00:00**]	
author_id	int(10) unsigned	

Indexes

PRIMARY	*id*
INDEX	*title*
INDEX	*author_id*

Alter indexes

Foreign keys

Source	Target	ON DELETE	ON UPDATE	
author_id	authors(*id*)	RESTRICT	RESTRICT	Alter

Yeah! It worked.

Database versioning with the migrations system

Until this point, we worked with the Schema class using some simple routes. Quite easy, but it's not a very good practice. You know: the single responsibility principle is not just a bedtime story.

Also, it's not just about the code: the database too should have a fully functional version control system in order to keep track of all the updates and facilitate a new member in your team.

Well, the migrations system is here to help. You can see it as a version control system for your database, which is made up of many files. Every single one of these files is a class with the two methods: `up()` and `down()`. In the `up()` method, you will put all of your database construction logic. In the `down()` method, instead, you will put anything related to *rolling back* what you did in the `up()` method.

Creating migrations

Let's make the first basic example. We already have the `books` and `authors` tables. Imagine that we want to gather data about our books' publishers.

We will have to:

- Create a new table called `publishers`, with at least the name of the publisher
- Create a new foreign key `publisher_id` in `books`

Let's make it with a migration file! Open your terminal and type the following:

```
php artisan make:migration publishers_update
```

Wait for a few seconds and then go to the `database/migrations` folder. You will find a file named like this:

```
2015_03_06_105131_publishers_update.php
```

Open it. This class already has empty `up()` and `down()` methods:

```
<?php

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

```
    class PublishersUpdate extends Migration {

      public function up()
      {
      //
    }

      public function down()
      {
        //
      }

    }
```

Now, fill them with some Schema Builder instructions, like this.

```
  <?php

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

    class PublishersUpdate extends Migration {

      public function up()
      {
        Schema::create('publishers',
function(Blueprint $table)
        {
          $table->increments('id');

          $table->string('name');

          $table->timestamps();
        });

        Schema::table('books', function(Blueprint
$table)
        {

 $table->integer('publisher_id')->unsigned();

 $table->foreign('publisher_id')->references('id')-
>on('publishers');
        });
      }

      public function down()
      {
        Schema::table('books', function(Blueprint
$table)
        {

 $table->dropForeign('books_publisher_id_foreign');
```

```
            $table->dropColumn('publisher_id');
        });

        Schema::drop('publishers');
    }

}
```

Quite linear: what we do in `up()` we undo in `down()`. Of course, in the reverse order. Remember, it's a *mirror* operation!

Now, it's time to run your migrations. All you have to do is run `php artisan migrate`; it loads all the migration classes and executes their `up()` method. In this way, you can build your database accordingly to your files. The execution order is decided by the file name. The file we just created is named `2015_03_06_105131_publishers_update.php`.

This means that this file was created on March 6, 2015 at 10:51:31. As you can easily imagine, all the migrations will be executed in a chronological order. To make things clearer, delete the two `2014_10_12_000000_create_users_table.php` and `2014_10_12_100000_create_password_resets_table.php`. We won't need them.

Return to your terminal and type:

```
php artisan migrate
```

Wait for a few seconds and you will get an output really similar to the following:

```
Migration table created successfully.
Migrated: 2015_03_06_105131_publishers_update
```

Now, return to your database. The update was done successfully, but we have another new table: the `migrations` table. Laravel uses this table to keep track of every update to the database. So, if you make another update to your application (with another new migration) and you run again the `php artisan migrate` command, you will execute only the new migrations and not the old ones. Exactly, version control, as I mentioned before.

Rolling back migrations

Let's try this *rollback* thing right now. In your terminal, type

```
php artisan migrate:rollback
```

Then, go back to your database: no more `publishers` table and no more related foreign key! Cool!

Got it? Good. Now, type this again:

```
php artisan migrate
```

Let's go forward.

> **Tip**
>
> You could get a `class not found` error while executing the `php artisan migrate` command.
>
> Don't worry, just type `composer dump-autoload` in your terminal and try again! Even the best, sometimes, make mistakes.

More examples, more migrations

Now, what about another example?

Let's imagine another feature. Alright, I have an idea: imagine that you want to specify some tags for every book you store with your application. This time you will have two things to do:

- Create a new `tags` table to store tags.
- Create a new `book_tag` table, as we are going to represent a many-to-many relationship. In this table, you will find the `book_id` and `tag_id`.

First of all, create a new migration. Open your terminal and type:

```
php artisan make:migration tags_update
```

Also, update the file as the following:

```
<?php

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

  class TagsUpdate extends Migration {

    /**
     * Run the migrations.
```

```
         *
         * @return void
         */
        public function up()
        {
          Schema::create('tags', function(Blueprint
$table)
          {
            $table->increments('id');

            $table->string('name');

            $table->timestamps();
          });

          Schema::create('book_tag',
function(Blueprint $table)
          {
            $table->increments('id');

            $table->integer('book_id')->unsigned();
            $table->integer('tag_id')->unsigned();


 $table->foreign('book_id')->references('id')-
>on('books');

 $table->foreign('tag_id')->references('id')-
>on('tags');
          });
        }

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

    }
```

Note

I like to write in the `up()` and `down()` methods at the *same time*. In that example you just saw, I put in `up()` the `create` method for `tags`. Then, I passed to the `down()` method and added the `drop('tags')` method call. Then, I went back to `up()` and added the `book_tag` table, and I finally went to `down()` again to drop the `book_tag` table.

> Working in this way helps me reduce distraction errors.

Now, open the terminal and type:

```
php artisan migrate
```

Go to your database. Open the `migrations` table to see what happened:



We have two different batches here. The first batch (*1*) is related to the publisher's update. The second batch (*2*) is related to the tags system update.

Why am I telling this to you? Who cares?

Well, when you type `php artisan migrate:rollback` in your terminal, the migrations system searches for the last batch and rolls it back. Not every batch, just the last batch. This means that the first `rollback` command will undo everything related to the tags system. If you type it again, the migrations system will also undo everything about the publisher's update.

With the batch number, you can know how many iterations you did on your database. However, there is another important thing you need to know: regarding our example, if you roll back two times and then migrate again, both migration files will be grouped under batch 1.

Also, you can roll back everything with this:

```
php artisan migrate:reset
```

You can roll back and migrate everything again with this:

```
php artisan migrate:refresh
```

Finally, you will get a list of all the migrated/rolled back migrations with this:

```
php artisan migrate:status
```

That's all! With this last command, we are also done with migrations.

Summary

Finally, the hero leaves his home. After the training and obtaining the right foundations, he is really ready to go.

Remember that the Schema Builder class and migrations system are not just about building your DB. In this chapter, you learned how to improve your database design method in a smarter way without the need for a single SQL line. Also, by using migrations, you are not going to need anymore to download database dumps or, in some *extreme* way, install any extra tool to deal with it.

Everything is done with the `artisan migrate` commands. However, as I already told you, this was just a *sample*. The real thing is arriving.

In the next chapter, we will go deeply and straight into the most important and *atomic* Eloquent component: the model.

Are you ready, hero?

Chapter 3. The Most Important Element – the Model!

Finally, I can say it: things are getting really, really serious from this moment. In the previous chapters, you studied everything you needed to make a start. Maybe it was a little annoying, but you knew that you would need it. Now, stop talking about the past; it's over. In this chapter, many amazing things await.

As you can imagine by the title, this part is going to be about the most important and atomic Eloquent element: **the model**. We will analyze the *M* in **MVC**. Considering that you are reading a book about creating a data-based application, exactly, really important! However, I don't want to bore you anymore. Let's talk about what we are going to see in the next sections.

Usually, there is a *standard* way to look at the Eloquent documentation. It's the same as you can see on the Laravel site. In my initial Laravel days, I used those pages but I felt a little *incomplete*. So, I changed things a little bit but in a more simplified way. If you have ever developed a web application in your life, I'm quite sure that you made a data-based application with tables and records. Right? Well...

If you think about it, you can do four basic operations on them. Whenever you develop an application, there's a very high possibility that you are going to implement some *create*, *read*, *update*, and *delete* logic for your items. This is exactly what we are going to see in this chapter.

First of all, we will introduce the simple, basic model. It will be a single line of code! Then, we will talk about CRUD (**create, read, update, and delete**) operations with Eloquent. After that point, you will have all the *basics* and an overview of the mechanism. We will also deal with the `where()` method and everything related, unscrambling everything about conditions and selections.

After that, we will go deep into the **Model** class, studying the mass assignment as another way to store and update our data. Then, we will discuss **timestamps** and **soft deletes**, and find out how Laravel and Eloquent deal with dates. After that, you will learn about query scopes and how to use them to improve your development process.

Also, we will take a look at many cool methods to transform our data in order to be shown (or stored) correctly: attributes casting, mutators, date mutators, and accessors. If that isn't enough, we will explore all the Model-related events that you can use to introduce new behaviors in your code

without breaking it. Also, Model observers will be analyzed when an event isn't enough.

In the last part of this chapter, we will explore some useful methods and features of the Model class, diving directly into the basic Model class code! Not bad, huh? Let's start! Here are the topics:

- Creating a Model
- Create, read, update, and delete operations basics
- Where, aggregates, and other utilities
- Mass assignment – for the masses
- Query scopes
- Attributes casting, accessors, and mutators
- Model events and observers
- Descending in the code

Creating a Model

First of all, let's see how you can create a Model and how its basic structure is made.

The fastest way to create a Model is to use the following command, with a parameter that you can use to specify the model name:

```
php artisan make:model
```

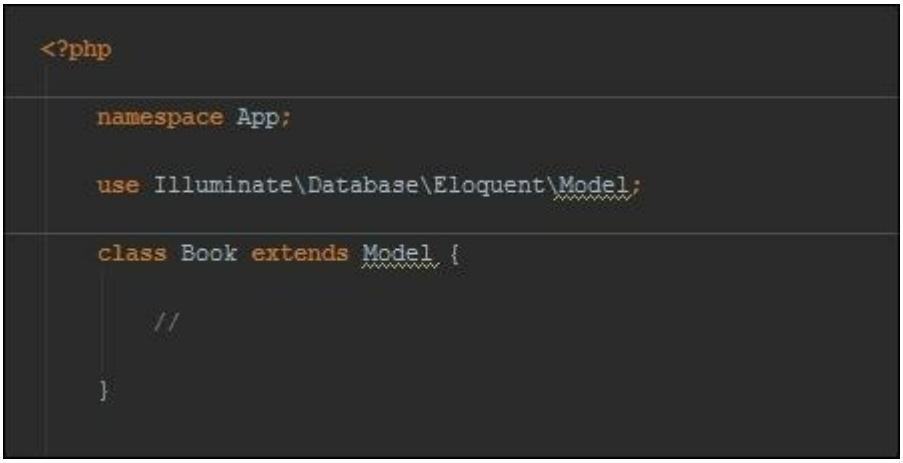
So, let's imagine that you want to create a `Book` model. The steps are as follows:

1. All you have to do is use the following command:

   ```
   php artisan make:model Book
   ```

   Of course, you can also create it manually; usually, Laravel puts Models in the `app` folder.

2. Once you have done this, let's open the `Book.php` file under `app` to see what's inside.



Wait. What? An empty class? Seriously?

Yes.

Laravel is designed to create a great web application in very little time. Eloquent (and its models) are no exceptions. The class you can see here is ready to be used in your application; using it, you will be able to do everything related to your books.

Talking about SQL databases, you can think of a connection between every model and table. If you respect a certain *convention*, Laravel automatically guesses the table name starting from the model name. So, if I have a model by the name of `Book`, Laravel will search for a `books` table on the database, without the need for specifying it explicitly.

If you need to bind a certain model with another table, you can specify the name adding it as a `$table` property, like this:

```
  <?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Book extends Model {
    protected $table = 'my_books';
}
```

Alright, nothing more to say here. Now, let's play with our new model.

Create, read, update, and delete operations basics

Every single article I read about Eloquent usually starts with some reading operations. I don't like it. I will teach you how to create and insert new records, then we will fetch them with some reading operations.

No boring test inserts with some external administration tool.

Creating operations

Let's create our first book! As a reference for the record *structure*, we will use the `books` table that we created in the previous chapter. The table has a very simple structure: a title, page count (`pages_count`), price, and description.

The procedure is as easy as creating an object.

Well, actually it is exactly the same. Create a new `GET` route in `routes.php` file under `app/Http/`, named `book_create` and type this:

```
Route::get('book_create', function(){

  $book = new \App\Book;

  $book->title = 'My First Book!';
  $book->pages_count = 230;
  $book->price = 10.5;
  $book->description = 'A very original lorem
ipsum dolor sit amet...';


});
```

If you think about it, there is something strange here! After checking the Model file, you can see that there are no `title` or `pages_count` properties declared. Among other things, Eloquent heavily uses magic methods. When the final query is built Laravel will use the names of the properties as the table columns to fill. Big deal!

Now, if you run this code and then check your table, you will not find any record yet. You must add a single final instruction: the `save()` method call.

```
Route::get('book_create', function() {

    $book = new \App\Book;
```

```
        $book->title = 'My First Book!';
        $book->pages_count = 230;
        $book->price = 10.5;
        $book->description = 'A very original lorem
ipsum dolor sit amet...';

        $book->save();

    });
```

Execute it. Now, your book is saved on the database.

If you want, you can access a specific record field even after you've saved it. Let's make another example.

```
    Route::get('book_create', function() {

        $book = new \App\Book;

        $book->title = 'My First Book!';
        $book->pages_count = 230;
        $book->price = 10.5;
        $book->description = 'A very original lorem
ipsum dolor sit amet...';

        $book->save();

        echo 'Book: ' . $book->id;

    });
```

> **Note**
>
> There is another little thing about Eloquent conventions; that is, every single table has an ID, autoincrementing a primary key.

Reading operations

Now that we've created some example records, why don't we try to read them? An example is better than a thousand words.

```
    Route::get('book_get_all', function(){

      return \App\Book::all();

    });
```

With a single instruction, here we are returning all the table records. The output is going to be very similar to this:

```
[
  {
    id: 1,
    title: "My First Book!",
    pages_count: 230,
    price: "10.50",
    description: "A very original lorem ipsum
dolor sit amet...",
    created_at: "2015-03-24 16:45:59",
    updated_at: "2015-03-24 16:45:59"
  }
]
```

> **Note**
>
> If you feel strange about this, don't worry. If you return, in a route (or in a controller method) the results of an Eloquent model query, the results will be automatically transformed in JSON. It is a very useful shortcut if you are thinking about building a RESTful API.

Let's create another book, to give our tests more elements.

```
    $book = new \App\Book;

    $book->title = 'My Second Book!';
    $book->pages_count = 122;
    $book->price = 9.5;
    $book->description = 'Another very original
lorem ipsum dolor sit amet...';

    $book->save();
```

After this, again execute your code in the `book_get_all` route. The result will be like this:

```
[
  {
    id: 1,
    title: "My First Book!",
    pages_count: 230,
    price: "10.50",
    description: "A very original lorem ipsum
dolor sit amet...",
    created_at: "2015-03-24 16:45:59",
    updated_at: "2015-03-24 16:45:59"
  },
```

```
    {
      id: 2,
      title: "My Second Book!",
      pages_count: 122,
      price: "9.50",
      description: "Another very original lorem
ipsum dolor sit amet...",
      created_at: "2015-03-24 16:57:15",
      updated_at: "2015-03-24 16:57:15"
    }
  ]
```

However, we can do more. In fact, another great method is the `find()` method. You can use it like this:

```
Route::get('book_get_2', function(){

    return \App\Book::find(2);

});
```

This method takes the primary ID as a parameter and returns the single record as an instance of the model.

Have a look at its output:

```
  {
    id: 2,
    title: "My Second Book!",
    pages_count: 122,
    price: "9.50",
    description: "Another very original lorem
ipsum dolor sit amet...",
    created_at: "2015-03-24 16:57:15",
    updated_at: "2015-03-24 16:57:15"
  }
```

Note that this time you don't have an array but a single object. These *static* methods, of course, aren't all Eloquent has to offer. The cool part starts here.

Take a look at this:

```
Route::get('book_get_where', function(){

    $result = \App\Book::where('pages_count',
'<', 1000)->get();
    return $result;

});
```

You can use the `where()` method to filter your results. Then, after specifying your criteria, the `get()` method retrieves the results from the database. For a better understanding, imagine that the `where()` method is building a query. The `get()` method executes it. This last one is a *trigger* method.

If you just want to retrieve the first result instead of all of them, you can use the `first()` method instead of `get()`.

```
Route::get('book_get_where', function(){

    $result = \App\Book::where('pages_count',
'<', 1000)->first();
    return $result;

});
```

Tip

Before we go any further, here's a little reminder that can save you a lot of time. Actually, when you use *trigger* methods such as `get()` or `first()`, you can get two different kinds of results.

When you use `first()`, you are selecting a single instance. So, you will receive as a result (if present) a single instance of a certain model. Otherwise, if you are using `all()`, or `get()`, you will get a collection of instances.

Getting back to our `where()`, you can chain as many calls as you wish.

```
Route::get('book_get_where_chained', function(){

    $result = \App\Book::where('pages_count',
'<', 1000)
            ->where('title', '=', 'My First
Book!')
            ->get();

    return $result;

});
```

You can iterate through results in a very simple way: simple for each is enough.

```
Route::get('book_get_where_iterate', function(){
```

```
        $results = \App\Book::where('pages_count',
'<', 1000)->get();

        if(count($results) > 0)
        {
          foreach($results as $book){

              echo 'Book: ' . $book->title . ' -
Pages: ' . $book->pages_count . ' <br/>';

          }
        }
        else
          echo 'No Results!';

        return '';
    });
```

The `$results` object is countable, so you can also check if you have results or not.

Have a look at this line:

```
      echo 'Book: ' . $book->title . ' - Pages: ' .
$book->pages_count . ' <br/>';
```

You probably already noted that you can access a single record field in the same way as you did for setting them: magic methods.

> **Note**
>
> If you like it, you can access record fields like an array.
>
> Try to switch `$book->title` to `$book['title']` and see what happens.

Updating operations

Updating a record is as easy as creating it. To be honest, it is exactly the same thing with a single change in the first instruction.

```
    Route::get('book_update', function() {

        $book = \App\Book::find(1);

        $book->title = 'My Updated First Book!';
        $book->pages_count = 150;

        $book->save();
```

```
});
```

Instead of creating it, we are retrieving from the database the instance of the Model we desired. After that, using magic methods, we modified and then saved it. As happened for the insert procedure, the change you make becomes persistent after the `save()` call.

Deleting operations

Deleting a record is the simplest thing.

```
Route::get('book_delete_1', function() {

    \App\Book::find(1)->delete();

});
```

Time to die, `Book`!

> **Note**
>
> Here's a key concept. When you use the `update()` and `delete()` methods, you are working on a Model instance, just like you did before while creating it.
>
> So, if you run a `\App\Book::find(1)` instruction, you will get a `Book` class instance as a result. For some, it will be obvious, but many newcomers often have problems with this.

where, aggregates, and other utilities

No doubt that the `where()` method will be one of your best friends while building queries and selecting records in your work with Eloquent. Don't you think it would be worth taking a look at it in detail?

Let's recap what we already know.

You can use the `where()` method to filter results. The correct syntax you have to use is this:

```
where('field_name', 'operator', 'term')
```

So, for example, you can filter all the books with less than 100 pages with

```
where('pages_count', '<', 100)
```

Also, you can chain more `where` methods, one after another, to build more complex queries. Let's select all those books that have less than 100 pages, with a title that starts with an *M*.

```
  where('pages_count', '<', 100)->where('title',
'LIKE', 'M%')
```

> **Note**
>
> Two chained conditions are equivalent to `condition1 AND condition2`.

Great!

where and orWhere

However, you know that this is not enough. Usually, in a real-world application, you may have more complex conditions. First of all, you may need to get results that respect a condition OR another condition. The `AND` condition is not a standard; so, here's the solution: `orWhere()`.

```
  Route::get('book_get_where_complex', function(){

      $results = \App\Book::where('title',
'LIKE', '%Second%')
            ->orWhere('pages_count', '>', 140)
```

```
            ->get();

        return $results;

    });
```

In this code, we are telling Eloquent to take all the books that have the word *Second* in the title or all the books with more than 140 pages.

Yeah, this is a little better than before.

Now let's try to imagine another more complex condition: we want to find all the books that have more than 120 pages and the word *Book* in the title, or all the books that have less than 200 pages and an empty description.

```
    Route::get('book_get_where_more_complex',
function(){

        $results =
\App\Book::where(function($query){

            $query
                ->where('pages_count', '>', 120)
                ->where('title', 'LIKE', '%Book%');

        })->orWhere(function($query){

            $query
                ->where('pages_count', '<', 200)
                ->orWhere('description', '=', '');

        })->get();

        return $results;

    });
```

Instead of specifying three parameters for `where()` and `orWhere()` methods, you can use a single closure parameter that takes a `$query` argument. Starting from that `$query` object, you will be able to do every selection and filtering in the way you prefer.

Of course, you can nest many of them:

```
    Route::get('...', function(){

        $results =
\App\Book::where(function($query){

            $query
                ->where(function($query){
```

```
                // other conditions here...

                $query->where(function($query){

                  // deeper and deeper in the
seas of conditions...

                });

              })
              ->orWhere('field', 'operator',
'condition');

      })->orWhere(function($query){

          $query
              ->where('field', 'operator',
'condition')
              ->orWhere(function($query){

          // other conditions here...

              });

      })->get();

      return $results;

  });
```

Alright, stop. I think that the concept is quite clear now. Let's see some other forms of `where`.

First of all, here's `whereBetween` that you can use to filter some fields using a range, not only a single value.

```
  $results =
\App
\Book::whereBetween('pages_count', [100,
200])->get();
```

With that, we just got all the books that have a page count between 100 and 200.

You can also use `whereIn` to check if a specific field is in an array of other values.

```
  $results = \App\Book::whereIn('title', ['My
First Book!', 'My Second Book!'])->get();
```

Finally, you can use `whereNull` if you want to get all the records with a certain column equal to null.

```
  $booksThatDontExist =
\App\Book::whereNull('title')->get();
```

Magic wheres

Another great and cool feature is the *magic* `where`. You know that everything in Laravel has a little bit of magic here and there. Obviously, Eloquent isn't an exception.

In fact, you can use an alternative syntax for your `where` clause, a magic syntax that lets you define the interested field as the method name.

You already know this syntax:

```
Route::get('book_get_where', function(){

      $result = \App\Book::where('pages_count',
'=', 1000)->first();
      return $result;

  });
```

Now, you can get the same result with this code:

```
Route::get('book_get_where', function(){

      $result =
\App\Book::wherePagesCount(1000)->first();
      return $result;

  });
```

Obviously, the `wherePagesCount` method doesn't exist but Laravel automatically creates a quick `where` clause using PHP magic methods. As you can see from the example (and as the syntax suggests), you cannot use this technique every time as it works only with the equal sign.

However, it's good to know a similar shortcut, right?

Aggregates

Sometimes, you will need to use the aggregates functions. No problem! Here is how you can do it with Eloquent.

```
\App\Book::count();
```

This is an example of its practical use:

```
Route::get('book_get_books_count', function(){

    $booksCount = \App\Book::count();
    return $booksCount;

});
```

Exactly as you saw before for `get()` and `first()` methods, you can use aggregate methods with `where` methods. Here, we are counting the number of the books with more than 140 pages.

```
Route::get('book_get_books_count', function(){

    $booksCount =
\App
\Book::where('pages_count', '>', 140)->count();
    return $booksCount;

});
```

Obviously, `count()` is not the only aggregate. Let's see them with some other `where()` examples to do more practice. This time, we are searching for the minimum number of pages (but the books with more than 120 pages, at least).

```
Route::get('book_get_books_min_pages_count',
function(){

    $minPagesCount =
\App
\Book::where('pages_count', '>',
120)->min('pages_count');
    return $minPagesCount;

});
```

You can also do the same thing with `max()`:

```
Route::get('book_get_books_max_pages_count',
function(){

    $maxPagesCount =
\App
\Book::where('pages_count', '>',
180)->max('pages_count');
```

```
    return $maxPagesCount;

});
```

Now, let's find the average price for all the books that have the *Book* word in the name:

```
  Route::get('book_get_books_avg_price',
function(){

    $avgPrice = \App\Book::where('title', 'LIKE',
'%Book%')->avg('price');
      return $avgPrice;

  });
```

Finally, let's get the sum of all the page counts for all the books with more than 100 pages.

```
  Route::get('book_get_books_avg_price',
function(){

    $countTotal = \App\Book::where('pages_count',
'>', 100)->avg('price');
      return $countTotal;

  });
```

Utility methods

As you probably imagined, Eloquent has a lot of utilities and methods that can improve your life as a developer. Trying to cover every single method would be hard but besides the most common methods, there are a few other methods that deserve to be here.

First of all, the `skip()` and `take()` methods that are used to implement some pagination in your queries.

```
$books = \App\Book::skip(10)->take(10)->get();
```

We are telling Eloquent to take 10 records and skip 10 from the start. So, if `skip(0)->take(10)` will take records from 1 to 10, `skip(10)->take(10)` will take from 11 to 20, and so on.

Of course, you can't also miss the `orderBy`, `groupBy`, and `having` methods. If you know a little about SQL databases (and if you are here, I think you do) you won't have any problem understanding this code:

```
// orderBy
```

```
\App\Book::orderBy('title', 'asc')->get();

// groupBy
\App\Book::groupBy('price')->get();

// having
\App\Book::having('count', '<', 20)->get();
```

Mass assignment... for the masses

You got the basics. Cool!

Now, you have to know about another couple of ways to insert a record with Eloquent.

The first is the Model constructor. You can pass an associative array as a parameter of the constructor call, something like this:

```
$book = new \App\Book([
  'title' => $title,
  'pages_count' => $pagesCount,
  'price' => $price
]);
```

Another is the `create()` method that you can call *statically* from the model itself.

```
$book = \App\Book::create([
  'title' => 'My First Book!',
  'price' => 10.50,
  'pages_count' => 150,
  'description' => 'My lorem ipsum dolor
description here...'
  ]);
```

Similar to the constructor, this `create()` method takes an associative array as a parameter and every key in this array corresponds to a column on the table (and a `magic` property on the model). The returned value (the one that is stored in `$book`) is an instance of the `Book` class.

This is called **mass assignment** and, until this point, everything is fine except for a serious security issue. In fact, sometimes you could pass the entire request input array as a parameter. Trust me: sooner or later, you will do it.

> **Note**
>
> The `$request` object you are going to see in the following example is a request instance that you can use to deal with the current request data. I will use it to retrieve some hypothetical `POST` data from a form (with the `all()` method) we will use to create a new book.

Something like this can be used:

```
$book = new \App\Book($request->all());

// or...

$book = \App\Book:create($request->all());
```

Technically, you don't know exactly what's in `$request->all()` so it is not the best practice, right? However, you can resolve this issue quite easily by adding a single property to your model—`fillable` or `guarded`.

Let's take an example for a better understanding of the concept. Imagine that for some reason you specify other fields in the form you already use for a new book insert, maybe the current user ID for logging purposes.

However, you just want to insert four single fields: `title`, `pages_count`, `price`, and `description`. Nothing else!

The key property is `fillable`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model {

    protected $fillable = [
        'title',
        'price',
        'pages_count',
        'description'
    ];

}
```

You can execute the code like this:

```
$book = \App\Book:create($request->all());
```

From the moment you execute the code, the model will search for `title`, `pages_count`, `price`, and `description` items. Nothing more! If you also have a `user_id` field in your request data array, it will be ignored by the model.

Of course, be careful and check twice what you put in the `$fillable` array of your models. Sometimes, developers easily forget the rights fields, and they pass hours and hours after an empty database record.

So, if you actually get a problem with an empty database record, remember: check your `fillable`.

The `guarded` property has a similar behavior but does the opposite thing: if `fillable` is a whitelist, then `guarded` is a blacklist. A mechanism like a blacklist can be extremely useful if you store some important and sensible information that you don't want to get updated by the user in any way.

Also, sometimes you will need to guard every single attribute of your model. No problem, use `guarded` like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model {

    protected $guarded = ['*'];

}
```

Note

If you put a field both in the `fillable` and `guarded` arrays, the model behavior will give the precedence to `fillable` and the field will be filled in.

Also, use `fillable` and `guarded` wisely. Remember also, that if you are using the `guarded` array and you are passing a classic full request data array to your model, you could update some unwanted fields and get some really unwanted errors.

Boring errors! Avoid them.

Timestamps and soft deletes

Time to cover two awesome features of the Model class: timestamps and soft deletes. How many times have you had to manually handle the creation date of a record and its last update time? Model timestamps are here to help.

Also, how many times have you had to create a delete feature while maintaining some information about your data, if not at all? Yeah, the soft deleting feature is here to help too.

Timestamps

Do you remember the last method we called in our `books` table migrations? No?

Don't worry, here it is:

```
  Schema::create('books', function(Blueprint
$table)
  {
    // other fields...

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

Exactly, it is the `timestamps()` method. This special Schema Builder method is used to create two separate fields: `created_at` and `updated_at`, both MySQL `DATETIME` or equivalent. Eloquent automatically handles these two fields when you create or update a record.

It can be extremely useful: how many times did you have to deal with some *last edit* data on a specific table? Also, imagine how much easier it could be with these two fields to handle some scheduled article posting.

However, sometimes they are not so useful: you can disable them just by setting the `timestamps` property model to false.

```
<?php

namespace App;

class Book extends Model {

    public $timestamps = false;

}
```

> **Note**
>
> Obviously, if you plan to disable timestamps you can delete the corresponding `timestamps()` Schema Builder call in your migration. You are not going to need it anymore.

You can eventually *specify the format of your timestamps*, if needed. Here's an example:

```
<?php

namespace App;

class Book extends Model {

    protected $table = 'books'';

    protected function getDateFormat()
    {
        // returining a different timestamp
format!
        return 'd/m/Y';
    }

}
```

All you have to do is to implement the `getDateFormat` method in your model and let it return a string that describes the desired format.

> **Note**
>
> In the returned string, you can insert anything that is a valid format for the `date()` PHP function (you can find a complete reference at http://php.net/manual/en/function.date.php).

Soft deleting

The soft deleting feature is a really interesting feature that can be useful on many occasions.

If you decide to activate soft deleting, you will never really delete a record: instead, a `deleted_at` column will be updated with the date of the operation, but nothing more. Eloquent will work exactly like before but you will never lose anything.

It could be a perfect fit for an e-commerce database that handles orders. The customer could decide to *clear* his order history. However, in order to keep accounts perfect, the shop owner would continuously need every single order detail.

As you can easily imagine, from the previous chapter, in order to activate the soft deleting feature, you will have to make a call to `softDeletes()` Schema Builder method.

```
  Schema::create('books', function(Blueprint
$table)
  {
    // other fields...

      $table->softDeletes();
  });
```

After this, you will have to include the `SoftDeletes` trait in the model.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\SoftDeletes;

class Book extends Model {

    use SoftDeletes;

    protected $dates = ['deleted_at'];

}
```

Nothing more!

Also, working with soft deleted data is really easy. Let's implement a simple example based on the situation we described before (the customer order's details and shop owner's need).

First of all, here's an example model:

```
<?php // Order.php

namespace App;

use Illuminate\Database\Eloquent\SoftDeletes;

class Order extends Model {

    use SoftDeletes;
```

```
    protected $dates = ['deleted_at'];

}
```

In his/her order history, a customer should see only his/her *existent* orders and not *deleted* orders. I used quotation marks because we are talking about records that always exist.

```
// getting all the orders

$orders = \App\Order::orderBy('created_at',
'desc')->get();
```

Nothing really special!

What about the shop owner? The magical word is `withTrashed`:

```
// getting all the orders, including the
"deleted" ones...

$orders =
\App\Order::withTrashed()-
>orderBy('created_at', 'desc')->get();
```

The `withTrashed` method automatically includes every result that is actually present in the table, regardless of the `deleted_at` field value.

Also, if you need to see only the soft deleted fields, change `withTrashed` to `onlyTrashed`.

```
$trashedOrders =
\App\Order::onlyTrashed()-
>orderBy('created_at', 'desc')->get();
```

Finally, you can restore a record that has been deleted with the `restore` method.

```
$trashedOrder =
\App\Order::find($trashedOrderId);
$trashedOrder->restore();

// $trashedOrder is not so trashed anymore...
```

If you prefer, you can execute the restore operation using a query as a `filter`.

```
    \App\Order::where('customer_id', '=',
$customerId)->restore();
```

Alright, I know what you are thinking: this feature is cool but what if I want to really delete a field?

No problem; just use `forceDelete`.

```
order = \App\Order::find($orderId);

// bye bye... forever :'(
$order->forceDelete();
```

Query scopes

Query scopes are very funny and also powerful. I really like them because, like many programmers out there, I am absolutely and overwhelmingly lazy. I also have a great justification for my laziness: the **Don't Repeat Yourself** (**DRY**) principle.

In a few words, they let you reuse some logic in your queries. It is useful if you have similar queries in your application and you don't want to write them again and again every single time.

Let's take an example.

```
<?php // Book.php

namespace App;

class Book extends Model {

    public function scopeCheapButBig($query)
    {
        return $query->where('price', '<',
10)->where('pages_count', '>', 300);
    }

}
```

What happened? I declared a `scopeCheaperButBig` method. The `scope` prefix is used to specify that this is going to be used as a scope.

Now, how can I use a scope?

Here it is:

```
<?php

$bigAndCheaperBooks =
\App\Book::cheapButBig()->get();
```

It is a cool feature. If you also think that you can split your logic in a more intelligent way, you can do it as follows:

```
<?php // Book.php

namespace App;

class Book extends Model {

    public function scopeCheap($query)
```

```
        {
            return $query->where('price', '<', 10);
        }

        public function scopeExpensive($query)
        {
        return $query->where('price', '>', 100);
        }

        public function scopeLong($query)
        {
          return $query->where('pages_count', '>',
700);
        }

        public function scopeShort($query)
        {
        return $query->where('pages_count', '<',
100);
        }

    }
```

Use it accordingly, reducing repeated code.

```
    <?php

      // getting cheaper and longer books;
      $cheapAndLongBooks =
\App\Book::cheap()->long()->get();

      // getting most expensive and longer books;
      $expensiveAndLongBooks =
\App\Book::expensive()->long()->get();

      // getting cheaper and shorter books;
      $cheapAndShortBooks =
\App\Book::cheap()->short()->get();

      // getting expensive and shorter books;
      $expensiveAndShortBooks =
\App\Book::expensive()->short()->get();
```

If you need it, you can also define dynamic scopes in order to pass parameters to your scopes. If you like it, calling a scope inside another is not a problem.

```
    <?php // Book.php

    namespace App;

    class Book extends Model {
```

```
    public function scopeLong($query)
      {
        return $query->where('pages_count', '>',
700);
      }

      public function
scopeLongAndCheaperThan($query, $amount)
      {
          return $query->long()->where('price',
'<', $amount);
      }

  }
```

Attributes casting, accessors, and mutators

Eloquent has many ways to transform the model data into something more readable or usable (and vice versa). In this chapter, we are going to analyze three of them: attributes casting, accessors, and mutators.

Attributes casting

The easier way to transform your model attributes while accessing them is the attributes casting. In a few words, it lets you define what attribute you want to cast and what is going to be the *destination* type.

Suppose that you have another integer field in your `books` table: `is_rare`. If the book is rare, this will be equal to `1`, `0` otherwise. However, when you work with it, the best thing would logically be as follows:

```
if($book->is_rare)
{
  // do wow things here...
}
else
{
  // do common things here...
}
```

It will not be something like this:

```
if($book->is_rare === 1)
{
  // ...
}
```

Right? Good.

So, all you have to do to fix this problem is to specify, in your model, the `casts` array:

```
<?php // Book.php

namespace App;

class Book extends Model {

  protected $casts = [
      'is_rare' => 'boolean',
  ];
```

```
}
```

From this moment, every time you call the `is_rare` attribute, it will be automatically converted to the Boolean corresponding value and returned.

The supported types for casting are: integer, real, float, double, string, Boolean, object, and array.

> **Note**
>
> As also the documentation suggests, the *array* casting type is useful when you have a JSON array stored in a specific table column and you want to work with it quickly.

Accessors and mutators

Attributes casting is very useful but has some limitations. Sometimes, starting from a simple value stored in the database, you need to do more complex work on it. Accessors and mutators are here to help.

To be more specific, an accessor is a method that is executed when the user reads a specific attribute. The accessor works on the attribute stored in the database and returns it. A mutator works in the opposite way: when you store a value, the mutator works, does its job and then saves it in the table. Let's say they are sorts of getters and setters.

Defining an accessor (or a mutator) is not so difficult: all you have to do is to follow a naming convention.

Let's start with something simple. Imagine that every time that you access the price of your book, you want to put the dollar symbol *$* at the beginning of the string.

```
<?php // Book.php

  namespace App;

  class Book extends Model {

    public function getPriceAttribute()
      {
          return '$ ' . $value;
      }

  }
```

The naming convention for an accessor is simple as shown:

- The method name starts with `get`
- The middle part of the name is the attribute name, which is camel cased
- The method name ends with `Attribute`

Nothing more.

Now, have a look at this:

```
$book = \App\Book::find(1);

echo $book->price;
// output: $ 10.50
```

Every time you have the preceding code, the mutator is very similar. This time, we want to store a lowercase version of the title.

```
<?php // Book.php

  namespace App;

  class Book extends Model {

    public function setTitleAttribute($value)
      {
          $this->attributes['title'] =
strtolower($value);
      }

  }
```

The convention isn't changed so much: the only difference is that the method name now starts with `set` and not `get`. It's all about getters and setters, my friend.

Another real common use of mutators is when the application stores the user's password. A mutator can be used to hash the chosen password and the result is then stored.

```
<?php // User.php

  namespace App;

  class User extends Model {

    public function setPasswordAttribute($value)
      {
```

```
            $this->attributes['password'] =
\Hash::make($value);
        }

    }
```

Descending in the code

In the previous text, we analyzed many aspects of the Eloquent model. If you think about your path, in a few pages you learned everything you needed to do many operations with your data. Also, you saw many interesting ways to add behaviors (from accessors and mutators to model observers) in order to improve your code usability, readability, and maintainability.

In this section you are going to read, I will go through the Model class and analyze it a little deeper than the usual. Nothing so advanced, so don't worry: I will just use the class code to show you what you can do with your models.

Take it as a *list* of useful tips and tricks.

A big file

At the moment I am writing this chapter, the `Model` class under `Illuminate\Database\Eloquent` counts 3361 lines of code, which is a big class with many methods and features. However, this length is acceptable considering that we can do the same thing in many different ways (just think about the `save()` method, the `::create()` method, and the possibility to specify an associative array in the model constructor: three ways to insert a new record).

In the first part of the class you can see all the properties we saw before, set to their default values.

```
...

protected $fillable = array();

...

protected $guarded = array('*');

...

protected $casts = array();

...
```

Yes, by default every attribute is guarded. A *total blacklist* for maximum security!

Quick conversion to array or JSON

Do you remember at the beginning of this chapter what I told you about the automatic conversion to JSON of your model attributes? If you take a look at the code into the model you will see two methods: `toArray` and `toJson`. These two methods are the ones that make the magic happen.

```
  // from the Illuminate\Database\Eloquent\Model
class

  ...

  /**
   * Convert the model instance to an array.
   *
   * @return array
   */
  public function toArray()
  {
    $attributes = $this->attributesToArray();

    return array_merge($attributes,
$this->relationsToArray());
  }

  ...

  /**
   * Convert the model instance to JSON.
   *
   * @param  int  $options
   * @return string
   */
  public function toJson($options = 0)
  {
    return json_encode($this->toArray(),
$options);
  }
```

> **Note**
>
> You can also see that the second method uses the first. If you don't fear big files, I suggest you to take a deeper look at all the methods in this class. You will see some awesome examples of code reusing and function writing. In this chapter, we are just scratching the surface.

You can convert both single models and model collections to arrays or JSON:

```
  return \App\Book::all()->toJson();
```

```
    // or ...

    return \App\Book::where('title', 'LIKE',
'My%')->get()->toArray();

    // or

    return \App\Book::find(1)->toJson();
```

Obviously, for security reasons, you can choose to mark some specific fields as `hidden`. They will not be shown when the record is converted to array or JSON.

```
    <?php

    namespace App;

    use Illuminate\Database\Eloquent\Model;

    class User extends Model {

        protected $hidden = ['password',
'credit_card_number'];

    }
```

In this case, with `hidden`, we specified a blacklist. You can also define a whitelist with the `visible` property.

```
    <?php

    namespace App;

    use Illuminate\Database\Eloquent\Model;

    class User extends Model {

        protected $visible = ['first_name',
'last_name'];

    }
```

Imaginary attributes

Using accessors and a specific property of the model, `appends`, you can create some attributes even if they don't exist as table columns.

For example purposes, let's assume that we want to create a `complete_name` attribute, made with the two attributes `first_name` and `last_name`.

First of all, let's create the appropriate accessor in the model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model {

    public function getCompleteNameAttribute()
  {
      return $this->attributes['first_name']. '
' . $this->attributes['last_name'];
  }

}
```

Then as the last step, we can include it in the `appends` array property.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model {

  protected $appends = ['complete_name'];

    public function getCompleteNameAttribute()
  {
      return $this->attributes['first_name']. '
' . $this->attributes['last_name'];
  }

}
```

Nothing else!

Now we can use our attribute like this:

```
echo $user->complete_name;
// outputs: Francesco Malatesta
```

> **Note**
>
> Every attribute that you create with the `appends` array respects the rules you specify eventually in the `visible` and `hidden` arrays.

Route model binding

Another little cool shortcut is **model binding**. In a few words, it lets you link a certain route parameter to an instance of a defined model.

Let's test it: first of all, go to `RouteServiceProvider` in `app/Providers` folder. Now follow these steps:

1. Add this line to the `boot` method:

```
public function boot(Router $router)
{
    parent::boot($router);

    $router->model('book', 'App\Book');
}
```

2. Then go to your routes file and create a new route that uses this parameter:

```
  Route::get('books/{book}',
function(App\Book $book)
  {
      return $book->title;
  });
```

3. So, calling the `books/1` URL will output something like this:

```
My First Book!
```

What happened? Laravel resolved and automatically instantiated in the `$book` variable the desired object, using the given parameter (`1`) as the search term based on the primary ID. Then, the instance was used to output the title.

It's a little trick but sometimes it could be useful.

You can also customize the resolution logic: here's another example using the e-mail address instead.

```
Route::bind('user', function($value)
{
```

```
        return User::where('email', '=',
$value)->first();
    });
```

If the record with the specified primary ID doesn't exist, the application will return a `404 not found` error. Obviously, you can change this behavior by specifying a third parameter for the `model()` binding.

```
Route::model('user', 'User', function()
{
    throw new MyCustomNotFoundHttpException;
});
```

Records chunking for memory optimization

Sometimes, you will need to process thousand and thousand of records. You know that those operations are very heavy for your RAM, but Eloquent has a useful method to chunk query results in *blocks* to optimize your load.

```
\App\Book::chunk(200, function($books)
{
    foreach ($books as $book)
    {
        // heavy operations on the book here...
    }
});
```

The first parameter defines the size of the block you want to use. In this case, we will load 200 results, process them, unload them, and repeat the same thing with the next 200.

The second parameter is a closure that defines what to do with that chunk: the `books` closure parameter is the returned collection of records.

Summary

Hey, we made it again! This first overview of Eloquent is completed and now you are able to make your first experiments on your own! Well actually you are able to do many experiments, not just the basics. We learned all about the CRUD, mass assignment, and many other interesting things.

If you feel overwhelmed by all this information, don't worry. Take your time to do many tests and get familiarized with all the mechanisms, including the conventions you found on your road. We are only at the third chapter!

In the next chapter, we will go deeper in one of the most amazing aspects of Eloquent: relations. You will experience the power of Eloquent and how it handles every kind of relationship between models, from the one-to-one to many-to-many, in all its beauty.

Go, go, go, hero!

Chapter 4. Exploring the World of Relationships

United (and related) we stand, divided we fall.

In a real-world context, everything is connected; for example, a car has an owner, a book has an author (or maybe more than one), or an e-commerce order is related to one or more products that a customer (another relation!) has ordered.

Everything, actually, is related!

There are no differences in the application development world; usually, you create software to solve a real-world problem. The real world is made from related things, so you will probably have to define many relationships between your entities.

However, let's be clear: I am not saying anything new. Just go to Wikipedia and search for `entity-relationship model`.

Usually, in your school books, you can find three fundamental types of relationships:

- **One-to-one**: This is used to relate a single entity with another single entity (for example, a person and an identity document)
- **One-to-many**: This is used to define a connection between an entity with more entities of the same type (for example, all the books of the same author)
- **Many-to-many**: This is used to relate multiple entities with many other entities (for example, a book can be a part of more than one category, and one category can include more than one book)

Of course, web development makes no exceptions. Eloquent makes no exceptions.

Following the *convention* used in what you have seen until now, Eloquent has a great way to deal with relationships, the methods used to define them for your models, and the techniques you can adopt to work with them.

So, let's explore what we are going to do in this chapter.

First of all, we will deal with the basic relationship types we just saw. How does Eloquent handle them? You will discover the beauty of powerful methods such as `hasMany` or `belongsTo`. This time, there are no more

snippets; we will follow the creation of our library management tool classes, defining every entity and every relationship.

After the basics, you will discover how to work with these relationships: how to query and use them in a comfortable and clean way. Also, we will see how to insert and delete related models in your database, or update the existing models.

Sometimes, working with many-to-many relationships will mean storing some data specific to that relationship. Eloquent has a very useful property named **pivot** that you can use to query the desired pivot table.

So, many things to look at this time! However, it is not over yet! Eloquent offers two other *relationships* that you can use: the *has many through* and *polymorphic many to many* relationships.

Alright, enough chit-chat! I am not going to spoil anything now. Follow the chapter, and you will fall in love with it.

Obviously, I will show you a real-world example for every concept. Come on, hero!

- The trinity: one to one, one to many, and many to many
- Querying-related models
- Eager loading (and the N + 1 problem)
- Inserting and updating related models
- Accessing *distant* relationships
- More power! Polymorphic relationships

The trinity – one to one, one to many, many to many

As I mentioned earlier, we will start from the basics. So, the first thing we will see is how you can define relationships between entities in Eloquent. This is really simple, and usually you will just have to add a single line of code for every relationship.

One to one

Our library is very concerned about tracking people who borrow books. For this reason, every new user has to give to the library some identity document data.

Now, every user has a single identity document, and every document is absolutely unique. If you think about it, this is a perfect one-to-one relationship. When you build your database, the most followed rule tells you

that you have to add the necessary columns in the *first* table. In this specific example, we would add columns to the user table.

However, someone could say "yes, but this is a completely different entity!"

Also, we could have to store many details about the identity document of every user: number, type, due date, city, and so on.

In this case, you would add four or five columns to an existing table. Many people don't like this solution, so imagine that you have the `User` and `IdentityDocument` models.

Here's our default `User` model:

```
  <?php namespace App;

  use Illuminate\Auth\Authenticatable;
  use Illuminate\Database\Eloquent\Model;
  use Illuminate\Auth\Passwords\CanResetPassword;
  use Illuminate\Contracts\Auth\Authenticatable
as AuthenticatableContract;
  use Illuminate\Contracts\Auth\CanResetPassword
as CanResetPasswordContract;

  class User extends Model implements
AuthenticatableContract, CanResetPasswordContract
{

    use Authenticatable, CanResetPassword;

    /**
     * The database table used by the model.
     *
     * @var string
     */
    protected $table = 'users';

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name', 'email',
'password'];

    /**
     * The attributes excluded from the model's
JSON form.
     *
     * @var array
     */
    protected $hidden = ['password',
'remember_token'];
```

```
}
```

The following is our `IdentityDocument` model:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class IdentityDocument extends Model {

  //

}
```

Now, let's see how to define the connection between them. For one-to-one relationships, the methods used are `hasOne()` and `belongsTo()`. The `hasOne()` method is used in the following way:

```
$this->hasOne('App\IdentityDocument');
```

Also, the `belongsTo()` method is used as follows:

```
$this->belongsTo('App\User');
```

The syntax makes sense, right? A user has a document, and a document belongs to a certain user! I know what you are thinking: you can't just place this method call wherever you want in the class.

In fact, you must define a method that returns that method call like this:

```
public function identityDocument()
{
  return $this->hasOne('App\IdentityDocument');
}
```

Similarly, `belongsTo()` must be defined as follows:

```
public function user()
{
  return $this->belongsTo(' App\User');
}
```

So, the final model's code will be like this:

```
<?php namespace App;

use Illuminate\Auth\Authenticatable;
```

```
  use Illuminate\Database\Eloquent\Model;
  use Illuminate\Auth\Passwords\CanResetPassword;
  use Illuminate\Contracts\Auth\Authenticatable
as AuthenticatableContract;
  use Illuminate\Contracts\Auth\CanResetPassword
as CanResetPasswordContract;

  class User extends Model implements
AuthenticatableContract, CanResetPasswordContract
{

    use Authenticatable, CanResetPassword;

    /**
     * The database table used by the model.
     *
     * @var string
     */
    protected $table = 'users';

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name', 'email',
'password'];

    /**
     * The attributes excluded from the model's
JSON form.
     *
     * @var array
     */
    protected $hidden = ['password',
'remember_token'];

    public function identityDocument()
    {
      return $this->hasOne('
App\IdentityDocument');
    }

  }
```

The code for the `User` class will be:

```
  <?php namespace App;

  use Illuminate\Database\Eloquent\Model;

  class IdentityDocument extends Model {

    public function user()
```

```
    {
      return $this->belongsTo(' App\User');
    }

}
```

Et voilà!

What exactly happened?

At a database level, we created an `identitydocuments` table with some data columns and a `user_id` external key. This external key is important because it is used automatically by Eloquent to resolve the relationship.

If you want, you can specify a different foreign key as a second parameter in both methods:

```
  $this->hasOne('App\IdentityDocument',
'another_user_external_id');
```

Otherwise, you can use this:

```
  $this->belongsTo(' App\User',
'another_user_external_id');
```

The field we specify is the same. Of course, the `hasOne` method (the `User` model) will see it as an external key, and the `belongsTo` method (the `IdentityDocument` model) will see it as a local one.

There's also a third parameter that you can use in both methods. In `hasOne()`, it is used to specify the local key (the default is the `id` field). In the `belongsTo()` method, it is used to define the parent key on the parent table (again, the default is the `id` field).

Let's take another example with the same models. Imagine that we have our `IdentityDocuments` table with a primary key named `documentidentifier`. Also, we need to follow a certain standard, and we can't use `user_id` as the external foreign key name. We must use `documentowner_id`.

No problem. First, you will define your `hasOne` like this:

```
  $this->hasOne('App\IdentityDocument',
'owner_id');
```

We don't need to define the third argument, as our `users` primary key is `id`.

Then, you define `belongsTo`:

```
        $this->belongsTo('App\IdentityDocument',
    'documentidentifier', 'owner_id');
```

Now you're done! Note that this concept is applied in the same way for other relationship methods that we will see.

One to many

This time it's easier than before: every book has an author, right? Sometimes, a book may have more than one author, but let's assume a basic case. The second relationship type we will analyze here is one to many. Every author can have more than one book. Let's consider the involved models: `Author` and `Book`.

Here's the `Author` class:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Author extends Model {

  //

}
```

The `Book` class is as follows:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model {

  //

}
```

These are nothing more than simple Eloquent models. Now, in order to define a one-to-many relationship, we must use the `hasMany()` method.

So, the `Author` class will look like this:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class Author extends Model {

  public function books()
  {
    return $this->hasMany('App\Book');
  }

}
```

You can then use the `belongsTo()` method, as before, to define the `inverse` of this relationship:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model {

  public function author()
  {
    return $this->belongsTo('App\Author');
  }

}
```

So, we are done! We used `belongsTo()` again as the concept of *belonging* is exactly the same; no differences.

At a database structure level, I just added an external `author_id` key in the `books` table.

So yeah, finished! I mention it in a note before, but I will repeat: remember that you can change your foreign key just by specifying it as a second parameter in the `hasMany()` and `belongsTo()` methods.

Many to many

Let's imagine a good example for many-to-many relationships. Well, the books/categories relationship is perfect. In fact, imagine *Twenty Thousand Leagues Under the Sea*, *Jules Verne*.

It is not only an adventure novel but also a classic. So, you will need to classify it under two separate categories: *Classics* and *Adventure*. Our library could also contain *Journey to the Center of the Earth*, another classic, but also an other adventure novel. Same thing!

As you can see, this time a many-to-many relationship is absolutely necessary. Let's discover how Eloquent handles many-to-many relationships and how you can define them on models.

Here's the code for our `Book` model:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model {

  public function author()
  {
    return $this->belongsTo('App\Author');
  }

}
```

The code for `Category` model is:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Category extends Model {

  //

}
```

This time, we don't have any *directions* or a possible *inverse* of a relationship.

Specifically, there are many books for many categories. So, the only method you need to use in this case is `belongsToMany()`. Use the method like this:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model {

  public function author()
  {
    return $this->belongsTo('App\Author');
  }

  public function categories()
```

```
    {
      return $this->belongsToMany('App\Category');
    }

}
```

The other method is used as follows:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Category extends Model {

  public function books()
  {
    return $this->belongsToMany('App\Book');
  }

}
```

Nothing else!

Let's see what we need at a database level to handle this relation. As you can easily imagine, you will have to work with a pivot table in this case.

So, you will need to specify an appropriate extra table in your migration file, using the `up()` method, as follows:

```
  Schema::create('book_category',
function(Blueprint $table)
  {
    $this->increments('id');

    $this->integer('book_id')->unsigned();
    $this->integer('category_id')->unsigned();

    $this->text('notes');

    $this->timestamps();
  });
```

You also have some conventions to follow. These are given here:

- The table name is comprised of the name of the entities, which is singular, separated by an underscore
- The table will contain two columns named after the interested entities (`author_id` and `book_id`)

Note

When you specify a relation, remember to use `return` before the appropriate method call. I know that's a little obvious, but very often newbies forget it.

It's really, really important that you follow the defined conventions. Laravel and Eloquent can change your workflow timeline significantly, but to get the result, you must follow conventions. The sooner you do it, the better you will feel.

A question of inverses

A little extra before we go forward. We just saw how to define a relationship and its inverse. However, what are the consequences you would stumble upon if you don't define an inverse? Or what if I define an inverse and not the *inverse of the inverse*?

Nothing special, actually!

The best rule is to define the relationship you need. Let's imagine a situation like this: in your software, you will need to know the category of a book, but not all the books in a certain category.

It is a strange situation, but this happens. In this case, you can just define the `categories()` relationship in `Book` and nothing else.

Vice versa, if in your application you will just need to get a list of books from the category and nothing else, you will define only the `books()` relationship in the `Category` model. That's all.

Done! The three basic relationship types are covered! You don't have to do anything more; actually, Eloquent handles everything automatically, so all you have to do is to write your code, use models, and raise your queries.

Oh, about queries...

Querying-related models

Now that you have learned how to define your relationships, I think you are ready to learn how to query them. Let's start with a very basic example.

Let's suppose that we are searching for the document number of a specific user. We will use the `User` and `IdentityDocument` entities we just saw. For the purpose of this example, imagine that you have a table `identitydocuments` with the following columns:

- `Number`: This indicates the document number
- `Type`: This indicates the document type
- `due_date`: This indicates the due date of the document
- `City`: This indicates the city where the document was released

Here's the code to get the document identity number, starting from a `User` instance:

```
  $user = \App\User::where('first_name', '=',
'Francesco')-
>where('last_name', '=', 'Malatesta')->first();

  $identityDocumentNumber =
$user->identityDocument->number;
```

If you echo `$identityDocumentNumber`, you will read the desired information. Nice, huh?

Well, this is the way Laravel and Eloquent deal with querying your relationships. Once you have defined it, all you have to do is to access it as a simple property or a method.

All the other queries will be executed automatically by Laravel. In fact, follow these simple instructions:

```
  $user = \App\User::where('first_name', '=',
'Francesco')-
>where('last_name', '=', 'Malatesta')->first();

  $identityDocumentNumber =
$user->identityDocument->number;
```

You just executed these queries:

```
  // the user Francesco Malatesta as an ID = 1...
  select * from users where first_name =
'Francesco' AND last_name = 'Malatesta';
```

```
  select * from identitydocuments where user_id =
1
```

Now put the result in to the `$identityDocumentNumber`. It is quite obvious for a one-to-one relationship; however, the same applies for a one-to-many relationship.

Let's consider another example right now: good old Jules will be a perfect fit. Let's suppose that we want to get a list of every Jules Verne books we have, and the code is as follows:

```
  $author = \App\Author::where('first_name', '=',
'Jules')-
>where('last_name', '=', 'Verne')->first();

  foreach($author->books as $book)
  {
    echo $book->title . <br/>;
  }

  // outputs:
  //
  // Journey to the Center of the Earth
  // Twenty Thousand Leagues Under the Sea
  // Around the World in Eighty Days
  // Michel Strogoff
```

Note

As I told you before, you can have access to your relationship using a simple attribute or a method call. What's the difference? Well, with the method call, you can do some filtering, and everything you saw before, in order to get the desired result. In fact, you can raise a query on a relationship.

Imagine that we want to get all the books with a *the* in the title. Here's the code:

```
  $author = \App\Author::where('first_name', '=',
'Jules')-
>where('last_name', '=', 'Verne')->first();

  $theBooks = $author->books()->where('title',
'LIKE', '%the%')->get();

  foreach($theBooks as $book)
  {
    echo $book->title . <br/>;
  }
```

```
// outputs:
//
// Journey to the Center of the Earth
// Twenty Thousand Leagues Under the Sea
// Around the World in Eighty Days
```

Cool, right? It's not over yet, this is just scratching the surface!

Accessing a pivot table

Working with many-to-many relationships, is not always about just defining a couple of external keys. You can choose to put extra data in to your pivot table in order to store some information for a specific connection between entities.

You already know how to create a pivot table, but how to access it? It's nothing complex: all you have to do is to use the `pivot` attribute of your relationship. Let's take an example using the previous books/categories relationship we created.

1. First, you must define which attribute you want to take from the table, modifying the `belongsToMany()` call in your model:

```
  return
$this->belongsToMany('App\Category')-
>withPivot('created_at', 'notes');
```

2. Then your code should be as follows:

```
  $book = App\Book::find(23);

  foreach($book->categories as $category)
  {
    echo 'Association Date: ' .
$category->pivot->created_at;
    echo 'Association Notes: ' .
$category->pivot->notes;
  }
```

In this little example, we just have printed all the dates that we *attached* a specific category to the book with `id = 23`. As an extra, we also printed some extra notes. This means that on the pivot table, we have the `created_at` and `notes` fields.

As a shortcut, you can also use:

```
  return
$this->belongsToMany('App\Category')-
>withTimestamps();
```

This is used if you just want to import timestamps data from the pivot table.

Querying a relationship

Eloquent allows you to query a relationship. In other words, you can get some results based on the existence of a certain relationship. Imagine that we want to get all the authors who have at least one book in the database.

With Eloquent, we can do something like this:

```
  $authorsWithABook =
\App\Author::has('books')->get();
```

In this case, you have to use the `has` method, specifying the name of the desired method for the relationship you want to check. The author will be added to `$authorsWithABook` only if at least one related book is be found.

If you don't like this *Boolean* approach, don't worry; let's see how to find every author with at least five books in the database.

```
  $authorsWithAtLeastFiveBooks =
\App\Author::has('books', '>=', 5)->get();
```

Yeah, you can specify the second and third arguments as operator and comparison term, respectively, for this *count check*.

I know, I know, cool, but not enough. Alright, what about getting all the authors with at least one book dated 1864?

Here we go, this time with the `whereHas` method:

```
  $authorsWithABookFromThe1864 =
\App\Author::whereHas('books', function($q)
  {
      $q->where('year', '=', 1864);

  })->get();
```

As you can see, you can do it in a quite elegant way. The first parameter specified is the name of the relationship you want to query. The second argument is a closure that takes a `$q` query parameter, which you can use to define conditions.

Note

The same concept you just saw here for a one-to-many relationship is also used for a many-to-many relationship.

Eager loading (and the N + 1 problem)

Every powerful tool must be used wisely. Relationships in Eloquent are no exception to the rule. Actually, one of the most common problems in using Eloquent is the *N + 1* problem. To explain it, I will use an example as usual.

Let's suppose that I am showing some data for the first 100 books. Starting with this data, I also want to print the author name for every single book.

Using what we've learned before now, here's the code:

```
$books = \App\Book::take(100)->get();

foreach($books as $book)
{
  $author = $book->author;

  echo $author->first_name . ' ' .
$author->last_name;
}
```

Even if the syntax is simple, under the hood, Eloquent is performing 101 queries! The first one is to get the list of the 100 books, then a query for every book to get the author. This is not exactly performance friendly, right?

Don't worry, there is a solution!

Basic eager loading

Eager loading solves your problem. This time using the `with()` method of the `Book` model, like this:

```
$books =
\App\Book::with('author')->take(100)->get();

foreach($books as $book)
{
  $author = $book->author;

  echo $author->first_name . ' ' .
$author->last_name;
}
```

Now, the number of executed queries will precipitate down to two, using a `where in`!

```
  select * from books;
  select * from authors where id in (1, 2,
3, ...);
```

If you want, you can also include multiple relationships in your final result. Let's also include categories data for every book!

```
  $books = \App\Book::with('author',
'categories')->take(100)->get();

  foreach($books as $book)
  {
    $author = $book->author;

    echo 'Author: ' . $author->first_name . ' ' .
$author->last_name;

    echo 'Categories:';

    foreach($book->categories as $category)
    {
      echo $category->name . ', ';
    }
  }
```

As if it's not enough, you can also include data from *nested relationships*.

Let's imagine that you are getting a list of categories in your application. Then, you want to include book data for every category, and for every related book you want to include the author's data.

All you have to do is this:

```
  $categories =
\App\Categories::with('books.author')->get();

  foreach($categories as $category)
  {
    echo $category->name;

    foreach($category->books as $book)
    {
      echo 'Title: ' . $book->title;
      echo 'Author: ' .
$book->author-
>first_name . ' ' . $book->author->last_name;
    }
  }
```

Advanced eager loading

If you want better control over your eager loading, you can define some constraints or conditions. Let's assume that you are getting a list of authors. From this list, you want to get every published book, from the oldest to the most recent.

You can do it like this:

```
  $authors = \App\Author::with(['books' =>
function($query)
  {
    $query->orderBy('year', 'asc');

  }])->get();
```

All you have to do is to specify the desired eager-loaded relationships as elements of an associative array with this syntax:

```
['relationship' => function($query){

  // conditions here, using the $query object

}]
```

Lazy eager loading

Sometimes you will use eager loading, but not every time. Sometimes you will need it, sometimes not.

If you want, you can manually eager load a specific relationship in the next moment.

How? It can be done with the `load()` method:

```
$books = Book::all();

// some operations here...

$books->load('author', 'categories');
```

The syntax is really similar to what you saw before; all you have to do is to specify the desired relationships as arguments.

Of course, you can also define conditions with the following:

```
$books->load(['categories' => function($query)
{
    $query->orderBy('name', 'asc');
```

```
        }]);
```

The same way, nothing less, nothing more!

> **Note**
>
> Remember that eager loading is a great solution to many performance issues. Especially in my initial experiments, it helped me a lot by letting me reach a lower number of queries. Just to make an example, on the Laravel-Italia forum, I showed a list of threads, with some replies, information, and thread author data, with just three queries.

Inserting and updating related models

Until now, you have learned how to define relationships and query them, in order to get data from related models. However, you can also insert and update related models in an easy way.

Let's pick a really basic example to get started: imagine that we are adding a new book (*Michael Strogoff*, *Jules Verne*). We have to specify some details as follows:

```
  $book = new Book;

  $book->title = 'Michael Strogoff';
```

Then, we specify the correct author ID:

```
$author = Author::where('first_name', '=',
'Jules')-
>where('last_name', '=', 'Verne')->first();

  $book = new Book;

  $book->title = 'Michael Strogoff';
  // other data...

  $book->author_id = $author->id;

  // and finally...
  $book->save();
```

The save() and associate() methods

Let's be clear; this works really well. However, it is not the best way you can do it. In fact, with Eloquent, you can work with related models using some other specific methods.

Let's rewrite this example using the method `save()` on a relationship. Also, we will use an associative array as a constructor argument in order to assign attributes.

```
  $author = Author::where('first_name', '=',
'Jules')-
>where('last_name', '=', 'Verne')->first();

  $author->books->save(new Book([
    'title' => 'Michael Strogoff',
```

```
    // other attributes...
  ]));
```

Done! It only took a couple of instructions. The `save()` method will automatically set the `author_id` key for the book you just passed as a parameter.

However, this is not just a time-saving trick; if you read the code carefully, you will note that we are actually creating very readable code. Instead of setting external keys, treating the relationship in a more *physical* way, we are saving the book as an element of the collection of books for a certain author. Quite different!

You can also do the same thing with arrays of objects, using the `saveMany()` method:

```
  $author = Author::where('first_name', '=',
'Jules')-
>where('last_name', '=', 'Verne')->first();

  $author->books->saveMany([
    new Book(['title' => 'Michael Strogoff']),
    new Book(['title' => 'The Mysterious
Island']),
    new Book(['title' => 'Off on a Comet'])
  ]);
```

As mentioned before, the `saveMany` method will set the external `author_id` key for every `Book` instance passed in the array. You can also update an existing relationship, changing its association with another model.

In this case, you must use the `associate()` method. Take a look at this example:

```
  $wrongAuthor = Author::where('first_name', '=',
'Jules')-
>where('last_name', '=', 'Verne')->first();

  $wrongAuthor->books->save(new Book([
    'title' => 'The Alchemist'
  ]));

  // oops! wrong author!

  $book = Book::where('title', '=', 'The
Alchemist')->first();
  $rightAuthor = Author::where('first_name', '=',
'Paulo')-
>where('last_name', '=', 'Coelho')->first();

  // done!
```

```
$rightAuthor->books->associate($book);
```

What happened? In the first part of the example, I assigned *The Alchemist* to Jules Verne. After one year of penitence in the desert, I came back to get the right author (by using the `$rightAuthor` variable), and then using the `associate()` method on the books relationship.

The first passed parameter is the model instance you want to work with.

> **Tip**
>
> So, the rule is quite simple: use the `save()` method when inserting a new record or the `associate()` method in order to update an existing record.

Remember that *The Alchemist* was not written by Jules Verne but by Paulo Coelho!

What about many to many?

Everything we saw is great for one-to-one or one-to-many relationships. However, what about many-to-many relationships? The mechanism, this time, is a little bit different; not in terms of complexity, but in terms of syntax, more than anything else. However, let's use an example as we did before.

- We have a few books: *The Alchemist* and *Journey to the Center of the Earth*.
- We also have some categories: *Science Fiction*, *Adventure*, and *Classic*.

As I mentioned in the first part of this chapter, this is a many-to-many relationship. What can we do to register a relationship between *The Alchemist* and *Adventure*, or *Journey to the Center of the Earth* and *Classic*?

In order to access our data in separate pivot tables, we will need other dedicated methods. I know that it seems stupid and boring, but use your database management tool to see what happens in your database when you interact with your data. It's always a good idea to familiarize with the process.

However, the first method you will use is `attach()`:

```
$book = new Book();

$book->title = "The Alchemist";
```

```
// other attributes...

$book->save();

// after save() call, an id is created.

$category = new Category();

$category->name = "Adventure";

$category->save();

// and now...

$category->books->attach($book->id);
```

As you can easily see, the `attach()` method is another one you can call on a relationship. Of course, only on a many-to-many one! It takes a single parameter in this case: the primary ID of the book I want to associate with the category.

You can also add an associative array as a second parameter if you are storing some extra data in your pivot table:

```
  $category = Category::where('name', '=',
'Opera')->first();

  $book = Book::where('title', '=', 'Journey to
the Center of the Earth')->first();

  $category->books->attach(
    $book->id,
    ['notes' => "Well, I'm not so sure about
this..."]
  );
```

The associative array follows the `attribute_name => attribute_value` format, as usual.

In the same way you *attach*, you can *detach*. If you want to remove a many-to-many relationship between two model instances, use the `detach()` method, as follows:

```
  $category = Category::where('name', '=',
'Opera')->first();

  $book = Book::where('title', '=', 'Journey to
the Center of the Earth')->first();

  // oh, come on...
  $category->books->detach($book->id);
```

So, you are done!

Also, both `attach()` and `detach()` methods support an array as a parameter instead of a simple integer.

```
$category->books->attach(
  [4, 8, 15, 17, 22, 42]
);

$category->books->detach([17, 22]);

$category->books->attach([16, 23 => ['notes' =>
'be careful next time...']]);
```

The sync() method

This is a really cool way to deal with many-to-many relationships. However, attaching and detaching things could be a little boring sometimes. Let's try the `sync()` method:

```
$category->books->sync(
  [4, 8, 15, 17, 22, 42]
);

$category->books->sync(
  [4, 8, 15, 16, 23, 42]
);
```

Confused? Let me explain everything. The `sync` method automatically *synchronizes* relationship data, taking an array of IDs. Element after element, it checks if the relationship was previously created or not, and sets (or unsets) them when necessary. Let's imagine that the pivot table `book_category` is empty; here's the first instruction:

```
$category->books->sync(
  [4, 8, 15, 17, 22, 42]
);
```

This instruction will create a connection between the chosen category and the books with the IDs 4, 8, 15, 17, 22, and 42. However, here's the second method call:

```
$category->books->sync(
  [4, 8, 15, 16, 23, 42]
);
```

It checks everything, and calculates a positive difference and a negative one. Books 17 and 22 are no longer in the array. The relationship will be

automatically *detached*. Instead, books 16 and 23 will be added with *attach*, another really cool utility method that saves you a lot of time!

You can obviously add data to the pivot table with the same method used earlier:

```
  $category->books->sync(
    [4, 8, 15, 16, 23, 42 => ['notes' => 'We
either live together.... or die alone.']]
  );
```

Accessing distant relationships

Another really interesting Eloquent feature is the possibility to define (and then access) a distant relation, using the `hasManyThrough()` method.

What? I see you are a little confused. No problem: let's take another example, which is a little different from our actual context.

Imagine that you are writing an application for some kind of research management for a research team. In this software, every user will be able to create a new research entity, and then add some sections to that research something like this:

- A `User` entity that has a one-to-many relationship with a `Research` entity
- The `Research` entity has a one-to-many relationship with a `Section` entity

Good. First of all, for models, you could write something like this:

```
// file: app/User.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model {

  public function researches()
  {
    return $this->hasMany('App\Research');
  }

}

// file: app/Research.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Research extends Model {

  public function user()
  {
    return $this->belongsTo('App\User');
  }

  public function sections()
  {
    return $this->hasMany('App\Section');
  }
```

```
}

// file: app/Section.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Section extends Model {

  public function research()
  {
    return $this->belongsTo('App\Research');
  }

}
```

As a basic setup, it could work. Now, what if we want access to every section added by a certain user? Probably, you would use something like this:

```
// getting $author with id = 1...
$author = Author::find(1);

// getting all $author researches...
$researches = $author->researches;

$allSections = [];

// iterating to get all sections
foreach($researches as $research)
{
  $allSections[] = $research->sections;
}
```

The `$allSections` array would contain every section added by the user. With Eloquent, if you want, you can create a shortcut using the `hasManyThrough()` method I mentioned earlier.

All you have to do is to put it in the User model as follows:

```
// file: app/User.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model {

  public function researches()
  {
    return $this->hasMany('App\Research');
  }
```

```
    public function sections()
    {
      return $this->hasManyThrough('App\Section',
'App\Research');
    }

  }
```

If you want, you can specify external keys (for the current and *middle* entities) as the third and fourth parameters:

```
  return $this->hasManyThrough('App\Section',
'App\Research', 'user_id', 'research_id');
```

In some such specific cases, it is a very useful shortcut. Enjoy it!

More power – polymorphic relationships

Probably, you are thinking that Eloquent is cool and very powerful.

Well, yes, it is. However, sometimes, `hasMany()` or `belongsToMany()` isn't enough. In some situations during your development flow, you will have to deal with more complex relationships that could involve more than two entities.

So, as a last part of this chapter, I will talk about polymorphic relationships. As usual, even if they aren't complex to learn, I will cover them with many detailed examples, in order to let you fully understand the entire concept.

Let's start from the simple polymorphic relationship.

A simple polymorphic relationship

A simple polymorphic relationship can be used when you have an entity that can belong to either one entity or to another.

So, here's our first example. Imagine that you are creating an e-commerce application. You will be able to upload some photos: either for a product, a category, or a blog post.

This means that we will have, first of all, four separate entities:

- Photo
- Product
- Category
- Post

Now, let's prepare some code skeletons as follows:

```
// file: app/Photo.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Photo extends Model {

  //

}

// file: app/Product.php
<?php namespace App;
```

```
use Illuminate\Database\Eloquent\Model;

class Product extends Model {

  //

}

// file: app/Category.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Category extends Model {

  //

}

// file: app/Post.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model {

  //

}
```

Now, you can define this polymorphic relationship using the `morphTo()` and `morphMany()` methods.

- The `morphTo()` method is used by the class that is related to all the other classes.
- The `morphMany()` method is called by the `owner` classes.

So, let's edit our models like this:

```
// file: app/Photo.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Photo extends Model {

  public function imageable()
  {
    return $this->morphTo();
  }

}
```

```
// file: app/Product.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Product extends Model {

  public function photos()
  {
    return $this->morphMany('Photo',
'imageable');
  }

}

// file: app/Category.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Category extends Model {

  public function photos()
  {
    return $this->morphMany('Photo',
'imageable');
  }

}

// file: app/Post.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model {

  public function photos()
  {
    return $this->morphMany('Photo',
'imageable');
  }

}
```

Done! Wait, wait. What's that `imageable` that is used both as a name for the method and as a string parameter? It is a name you can choose for yourself: however, take a look at the table structure I have used, to understand.

```
products
    id - integer
    name - string
```

```
categories
    id - integer
    name - string

posts
    id - integer
    name - string

photos
    id - integer
    path - string
    imageable_id - integer
    imageable_type - string
```

The photos table has two special fields: `imageable_id` and `imageable_type`. A simple external key, the only difference is that, for the elements in this photos table, you can count *owners* of different types.

So, in `imageable_id`, you will put the owner ID and in `imageable_type`, the owner class name!

If a photo belongs to a product, you will see `Product` in the `imageable_type` column, then `Category` if the photo belongs to a category, and so on.

Obviously, working with this relationship is very simple. Here's an example:

```
// getting a sample product...
$product = App\Product::find(3);

foreach($product->photos as $photo)
{
  // working with photos here...
}
```

This applies for every other entity!

```
// getting a sample category
$category = App\Category::find(42);

foreach($category->photos as $photo)
{
  // working with category photos here...
}
```

Finally, you can also *reverse* things. If you have a photo and want to know *who* the owner is, all you have to do is:

```
$photo = App\Photo::find(23);

// getting the owner...
```

```
var_dump($photo->imageable);
```

No matter what the owner's class is, Eloquent will automatically resolve the instance and return it to you. If the *owner* is a blog post, you'll get the blog post. Easy!

A many-to-many polymorphic relationship

If a simple polymorphic relationship can be defined as a *special* one to many, the many-to-many relationship finds equivalence in many-to-many polymorphic relationships.

As you saw in the earlier text, it works exactly like a many-to-many relationship. The only difference is that you can *connect* a certain entity with more entities.

For our example, this time, let's return to our library management system. At the beginning of this chapter, you saw three main entities:

- Author
- Book
- Category

Between books and categories, there is a many-to-many relationship. A book can belong to more than one category. Similarly, a category can include more than one book. Now, imagine that you want to *extend* this concept to authors.

Let's pick good old Jules as a perfect example. He wrote adventure books, so he could be easily classified as an adventure author.

The many-to-many polymorphic relationship is the best way to deal with the situation. This time, you will have to use the `morphMany()` and `morphedByMany()` methods:

```
// file: app/Author.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Author extends Model {

  public function categories()
  {
    return $this->morphToMany('App\Category',
'categorizable');
  }

}
```

```
// file: app/Book.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model {

  public function categories()
  {
    return $this->morphToMany('App\Category',
'categorizable');
  }

}

// file: app/Category.php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Category extends Model {

  public function authors()
  {
    return $this->morphedByMany('App\Author',
'categorizable');
  }

  public function books()
  {
    return $this->morphedByMany('App\Book',
'categorizable');
  }

}
```

Of course, as for every many-to-many relationship, you will need a pivot table with a structure similar to the following:

```
categorizables
    tag_id - integer
    categorizable_id - integer
    categorizable_type - string
```

Pay maximum attention to names and conventions, as usual. The second parameter of `categorizable` for the `morphToMany()` and `morphedByMany()` methods is the same as you specify in the pivot table, for `categorizable_id` and `categorizable_type`.

Note

Also, the table name used is the plural for the term (`categorizable` and `categorizables`).

After this setup, you are able to use the relationship in your code, as follows:

```
// getting a sample author
$author = App\Author::find(30);


// accessing categories...
var_dump($author->categories);

// getting a sample book
$book = App\Book::find(60);

// accessing categories... in the same way!
var_dump($book->categories);
```

While creating tables, you will need to add the specific `_id` and `_type` columns, such as `categorizable_id` and `categorizable_type`.

Note

In the Schema Builder, you can use the `$table->morphs('categorizable')`, if you want. It will automatically add the columns you need, just specify the `-able` name you desire.

Summary

Alright, I think that's enough. Well, actually that's all; time to rest!

You have learned everything related to relationships in Eloquent, and now you can even build complex applications in no time. You know everything related to Eloquent basics, so my suggestion is: take your time to recap everything, do some tests, write good code, and enjoy models and relationships.

When ready, turn the page and dive into something more advanced!

Chapter 5. Using Collections to Enhance Results

Until now, I told you everything about models and how to create relationships between them. I explained you how to query your data and relationships and even to specify complex conditions and constraints. However, I have never told you anything about what Eloquent's outputs are. Yes, sometimes, I mentioned an *array* or just the word *results*. Don't worry; it wasn't wrong, but there is something more under the hood.

Well, in this chapter, I will discuss collections. When retrieving your results from a query (using, for instance, `get()` or `all()`), you are getting a collection. That's the right term to use.

Basically, you can think of a collection as an array of results but with some extra utility methods. In fact, when you use a collection, you are using an instance of the `Collection` class under `Illuminate\Database\Eloquent`.

This class implements the `AggregateIterator` interface that lets you treat a collection like an array. You can use collections to perform many operations, sometimes complex operations as well. First of all, you will see how to perform some basic research operations and checks with collections.

Then, we will see some results transformation methods. Do you remember, in Chapter 3, *The Most Important Element – the Model!*, I had told you about the automatic transformation in JSON of a model result? Great! It's one of these methods.

Straight after, we will go a little deeper; after all, a collection is made up of elements. We will work with these elements. With a collection, obviously, you can iterate through its elements. There are some dedicated methods for iterations. Also, you will learn how to filter a collection in an easy way, just as many things in Eloquent are easy. Finally, we will talk about sorting operations on collections and how to deal with them. So, nothing of this really is essential, but it will help you to better understand how Eloquent works in every single way.

Are you ready? Here are the topics to cover:

- Basic collection operations
- Transforming collections
- Iterating and filtering
- Sorting

Basic collection operations

Let's start with some really basic methods. For a better understanding of what you are going to do, I *strongly suggest that you try on your project* every single method from the following list:

- The first is `contains()`. It returns true or false if a record with a certain ID is included in the collection.

  Here's an example:

  ```
  $books = \App\Book::all();

  if($books->contains(3))
  {
    return 'yeah, book 3 is here!';
  }
  ```

  All you have to do, here, is to specify the ID as a parameter.

- As I told you earlier, you can use a collection as an array. So, if you want to get the third element in a collection, you can use this:

  ```
  $book = $books[2];
  ```

- However, if you don't like this syntax for some reason, you can use `get()` in an alternative way, like this:

  ```
  $book = $books->get(2);
  ```

- With this *powered* syntax, you can also specify a default value if the desired index doesn't exist:

  ```
  $book = $books->get(2, "Not Found!");
  ```

- Obviously, you can check the existence of a specific element, if you want, with `has()`:

  ```
  if($books->has(3))
  {
    $book = $books->get(3);
  }
  ```

- As the opposite of `get()`, you can add an element with a specific index using `put()`:

  ```
  $book = new Book;
  // other attributes assignment here...

  $books->put(2, $book);
  ```

As you may imagine, the first argument is the desired index, and the second argument is the value.

- Another cool method is `prepend()`, which you can use to prepend an element to a certain collection. Here's the syntax:

```
$firstBook = new Book;
// other attributes assignment here...

$books->prepend($book);
```

- If you want to get an array with all the primary keys, you can use a dedicated `modelKeys()` method!

```
$books = \App\Book::all();

$primaryKeys = $books->modelKeys();
```

It's not over yet; actually, there are many methods you can use for many different things.

- An example is `random()`, which extracts a single random item from the specified collection:

```
$books = \App\Book::all();

$randomBook = $books->random();
```

- Also, you can use `keys()` or `values()` to get arrays for only keys or values, respectively.

```
$books = \App\Book::all();

$keysArray = $books->keys();
$valuesArray = $books->values();
```

- What? You want to treat your collection as a stack? No problem, `pop()` and `push()` are here to help!

```
// let's get the last item!
$books = \App\Books::all();
$lastBook = $books->pop();

// lets' add a new item ad the end!
$book = new Book;
// other attributes assignment here...

$books->push($book);
```

- Now, what about searching for an item in a collection using syntax similar to the `here()` you called on models? Take a look at this:

```
$books = \App\Book::all();
```

```
    $book = $books->where('title', 'Michael
Strogoff');
```

Note that this method returns another category. This means that you can use many chained calls of `where()`. Here's another example that gives a better idea:

```
  $books = \App\Book::all();

  $book = $categories->where('year',
1876)->where('page_count', 254);
```

- Let's close this first part of our chapter with `perPage`, which is a really intuitive method that gets a certain number of items, and all you have to do is to specify the page and number of items you want per page.

  The syntax is something like this:

```
  $books = \App\Book::all();

  $secondPageBooks = $books->perPage(2, 10);
```

  With this simple call, you are getting 10 books, starting from the second page. I think this is a great example of expressive syntax of methods that Eloquent (and Laravel) offers.

Note

If you want to know more about the `Collection` class and what it offers, take a look at http://laravel.com/api/5.0/Illuminate/Database/Eloquent/Collection.html or look directly at the code in `Illuminate\Database\Eloquent\Collection` and `Illuminate\Support\Collection` classes.

Transforming collections

Quite often, Eloquent automatically takes a collection to transform it into something that you can output in a better way. For example, here is the code that I am using to show a list of a magazine website's news categories:

```
$categories = \App\Category::all();
 return $categories;
```

This is the corresponding output:


```
[
  {
    id: 1,
    name: "Editorial",
    slug: "editorial",
    description: "Qui est quo asperiores
aliquid vitae possimus. Dolor consequuntur
similique voluptatem a laborum dolorem ea
repellendus. Aspernatur ducimus quis dolorum
consequatur vel nam at. Aut omnis rem laborum.",
    created_at: "2015-04-21 10:14:37",
    updated_at: "2015-04-21 10:14:37"
  },
  {
    id: 2,
    name: "Interview",
    slug: "interview",
    description: "Rerum deleniti rerum aliquid
laudantium id non voluptatum. Aut quia distinctio
consequatur velit natus inventore sunt iusto. Non
totam quis quam sint et.",
    created_at: "2015-04-21 10:14:37",
    updated_at: "2015-04-21 10:14:37"
  },
  {
    id: 3,
    name: "Reportage",
    slug: "reportage",
    description: "Adipisci et veritatis
excepturi ullam explicabo. Eos dolore quas a
vero. Optio voluptatem accusamus ex optio. Rerum
rem quaerat qui maiores.",
    created_at: "2015-04-21 10:14:37",
    updated_at: "2015-04-21 10:14:37"
  }
]
```


However, now consider the following code:

```
$categories = \App\Category::all();
 return $categories;
```

Then, let's change the code to this:

```
$categories = \App\Category::all();
dd($categories);
```

This is what happens:

```
Collection {#152
  #items: array:3 [
    0 => Category {#155
      #connection: null
      #table: null
      #primaryKey: "id"
      #perPage: 15
      +incrementing: true
      +timestamps: true
      #attributes: array:6 [
        "id" => 1
        "name" => "News"
        "slug" => "news"
        "description" => "Qui est quo
asperiores aliquid vitae possimus. Dolor
consequuntur similique voluptatem a laborum
dolorem ea repellendus. Aspernatur ducimus quis
dolorum consequatur vel nam at. Aut omnis rem
laborum."
        "created_at" => "2015-04-21 10:14:37"
        "updated_at" => "2015-04-21 10:14:37"
      ]
      #original: array:6 [
        "id" => 1
        "name" => "News"
        "slug" => "news"
        "description" => "Qui est quo
asperiores aliquid vitae possimus. Dolor
consequuntur similique voluptatem a laborum
dolorem ea repellendus. Aspernatur ducimus quis
dolorum consequatur vel nam at. Aut omnis rem
laborum."
        "created_at" => "2015-04-21 10:14:37"
        "updated_at" => "2015-04-21 10:14:37"
      ]
      #relations: []
      #hidden: []
      #visible: []
      #appends: []
      #fillable: []
      #guarded: array:1 [
        0 => "*"
```

```
        ]
        #dates: []
        #casts: []
        #touches: []
        #observables: []
        #with: []
        #morphClass: null
        +exists: true
      }
      1 => Category {#156 ...}
      2 => Category {#157 ...}
    ]
}
```

Wait, wait; what? Just changing a `dd()` call with a return? Well, you can see this magic using two special methods: `toArray` and `toJSON`. You can also use them manually, if you need, just like this:

```
$books = \App\Book::all();

$toArray = $books->toArray();
$toJson = $books->toJson();
```

Cool, right?

> **Note**
>
> The `dd()` function I used before is a Laravel utility. It's a mix of the native PHP `var_dump()` and `die()`. To be more precise, it shows the value of a certain object or variable, and then stops the script.

Iterating and filtering

Sometimes you will need something more than passing a collection in to a view, or a simple `toArray()` call. An Eloquent collection has many methods that you can use to filter and iterate through its elements. Let's see something in action!

Iterating

First of all, let's begin with simple iteration. You can call the `each()` method to iterate the elements of a certain collection:

```
$books = \App\Book::all();

$books->each(function($book)
{
  echo $book->title;
});
```

All you have to do is to pass as the first (and only) argument a closure with a single parameter: the single item to be used. In this example, I just printed all the titles.

Filtering

If you want to filter your collection in a more complex way, you can use `filter()`. Let's take an example: I want to select every book that was printed after 1840.

```
$books = \App\Book:all();

$books->filter(function($book)
{
  if($book->year > 1840)
    return true;
  else
    return false;
});
```

The syntax is really similar to the previous example. You have a closure as a parameter, with a single argument passed; that is, the collection item.

However, this time you will have to check your conditions and return true or false if you want to include (or not) this item in the `result` collection.

So, in this specific case, the current `$book` value was printed after 1840? Great, come in. Not printed after 1840? Bye bye!

Sorting

Finally, you can sort data using a certain field. The methods you must use, this time, are `sortBy` and `sortByDesc`. I think you are quite smart enough to understand what they do, right?

However, here are some examples:

```
// ordering books by title, ascending
$books = $books->sortBy(function($book)
{
    return $book->title;
});

// ordering books by creation date, descending;
$books = $books->sortByDesc(function($book)
{
    return $book->created_at;
});
```

Also, you can use a shortcut if your closure logic is really simple, such as the earlier examples:

```
$books = $books->sortBy('title');
$books = $books->sortByDesc('created_at');
```

Summary

Let's be clear; in my opinion, knowing every single method of a collection isn't really indispensable. However, it can be really useful in some situations where you need a certain method to do something very specific. How can I say it? The more things you know, the better you are!

Now, let's move on! After this little break, it's time to go down in to the world of events!

Chapter 6. Everything under Control with Events and Observers

Have you ever heard anything about the single responsibility principle? I hope so. It is one of the SOLID principles in programming, and it basically says that a class has one and only one responsibility. In other words, every class has to do *one* single thing and not anything else.

Usually, when you build the first version of software, everything goes fine. Then, it happens. Your boss calls: time to introduce a new feature, developer! Especially if *update* means *insert this little extra behavior here*, your code base easily becomes heavy and sloppy.

Terribly sloppy! Then, you fight against deadlines, tests, Q&A, and so on, literally an odyssey. Not a very good practice, right?

Now, in the software development world, you can find many techniques and methods to add new features to your software in an elegant way. You have probably heard about *events* in programming.

In a few words, let's say it consists of logic such as this: *when X does this, then Y must do that*.

Imagine a similar situation in your application: you just finished your application and then you say, "Oh, I just forgot to send an e-mail to the newbie user!"

With Eloquent, you can handle this situation in two ways. The first way is using the very interesting concept of model events. The second way is based on a more advanced concept: model observers.

In this chapter, in the first place, you will learn everything about events in the context of Eloquent models. Then, I will cover model events: what they are and when you would use them.

Then, I will do the same for model observers. You will learn all the differences, and the pros and cons. Obviously, for both of the concepts, I will use a practical example to show how to use them in a real-world situation.

Are you ready, hero?

- When should I use events in my models?
- Model events
- An example of model events
- Model observers

- An example of model observers

When should I use events in my models?

What is an event? If you search the term on Google, you will get multiple results.

For example, it will be defined as *something that happens or is regarded as happening; an occurrence, especially one of some importance*. It may also be defined as *something that occurs in a certain place during a particular interval of time*.

I like these two definitions because they fit perfectly in this context. In fact, you can see the *particular interval of time* as the model lifecycle, in a certain sense.

You can create a new instance, update an existing instance, or delete it.

Every operation you can do is related to two events.

Starting from the basics: I have just created that record, I deleted that record, or I am updating that record, sounds natural, right?

Good. Now, Eloquent triggers some events when something happens in the model lifecycle. To be more precise, they are as follows:

creating	saved
created	deleting
updating	deleted
updated	restoring
saving	restored

For every operation, you have two separate events. As you may imagine, they refer to separate moments. Let's pick the *create* operation for our example.

You have the *creating* event, that you can read as "the create operation is going to happen." Then, you have *created* that means "the create operation just happened".

As a scientist would say:

Operation	**Description**
creating	is about the *t - 1* moment
created	is related to the *t + 1* moment

So, you have two events for the three basic operations: create, update, and delete.

Also, you can see two more operations: save and restore. However, don't worry, they are not complex, in fact:

- **Save**: All you have to know is that the save operation is related both to *create* and *update*. Let's assume that you want to add a behavior, whether or not the application is creating a record or saving an existing record. Why bother declaring the same thing twice? Just use the "save" generic operation and you're done.
- **Restore**: The restore operation is used when you have the soft deletes feature enabled for a certain model, and you undo a delete operation on it.

Ok, I know what you are thinking: what about going deeper into the concepts?

Model events

The first technique we are going to look at for events is called **model events**. The basic concept is really simple:

- In the `EventServiceProvider` class, you can add a special event listener and bind it to a certain closure
- In this closure, you will be able to specify your new behavior without touching the model code
- This binding must be placed in the `boot()` method of the class

Here's a simple example of a binding between the *created* user event and a closure, passed as a parameter of the called method.

The `$user` parameter of the closure contains the instance of the concerned user:

```
    public function boot(DispatcherContract $events)
    {
        parent::boot($events);

        User::created(function($user)
        {
            // doing something here, after User
creation...
        });
    }
```

As you can imagine, every model has these methods, named after the events we saw before. So, if you want to bind a certain operation to a *saved* event, for instance, you must use the following:

```
    User::saved(function($user)
      {
          // doing something here, after User save
operation (both create and update)...
      });
```

Another interesting feature is the possibility to stop the current operation with the *pre* methods. In fact, if you are using any of the following:

- `creating`
- `updating`
- `saving`
- `restoring`
- `deleting`

You can decide to return the Boolean value `false` if you want to abort the operation.

Let's say you want to abort the create operation if the user e-mail ends with the `@deniedprovider.com` string. You could do something like this:

```
User::creating(function($user)
  {
    if(ends_with($user->email,
'@deniedprovider.com'))
    {
      return false;
    }
  });
```

Obviously, you can't do the same on the created, updated, saved, restored, and deleted events; the event just happened and you can't go back in time!

An example of model events

We can now look at some examples of model events in action.

Let's start with the great classic. A new user joins us, and we want to greet them with a welcome e-mail. Nothing is easier! Let's open `EventServiceProvider` and add this code just after the `parent::boot()` call:

```
  User::created(function($user){

    Mail::send('emails.welcome', ['user' =>
$user], function($message) use ($user)
    {
        $message->to($user->email,
$user-
>first_name . ' ' .
$user->last_name)-
>subject('Welcome to My Awesome App,
'.$user->first_name.'!');
    });

  });
```

Done! Wasn't it easy?

> **Note**
>
> I am assuming that you have a `welcome.blade.php` view in the `emails` folder under `resources/views`. I am also assuming that you know the basics of e-mail sending in Laravel. If you need some more info, go to http://laravel.com/docs/5.0/mail#basic-usage.

Here is another example of model events in action:

Let's assume that we have a class (for the purpose of this example) that is delegated to send an e-mail to every user who wants to know when a new book by a certain author is added. The class is named `NewBookNotifier`, and the method is called `forAuthor($authorId)`, where the `$authorId` is the primary key of the desired author.

We could do something like this:

```
Book::created(function($book){

  $newBookNotifier = new NewBookNotifier();
```

```
        $newBookNotifier->forAuthor($book->author->id);

         });
```

Done! The main point is that this is very simple. As I mentioned earlier, the most important part is that every model remains untouched. You can even add a very complex behavior, but you will not touch anything in the model. This is a great advantage because if you test that model and you don't touch it, you can be sure that it will never break.

Now, I am going to talk about *more complex things*.

Events observers

Model events are cool, I agree. However, sometimes, you could need something more advanced.

When you use Laravel, you are working mostly with object-oriented programming and probably want to do the same with your model events. The answer to your questions is model observers, which is an advanced version of model events.

To use them, all you have to do is to declare a new class like the following (maybe in a dedicated folder called `observers`):

```
class BookObserver {

  public function creating($book)
  {
    // I want to create the $book book, but
first...
  }

    public function saving($book)
    {
        // I want to save the $book book, but
first...
    }

    public function saved($book)
    {
        // I just saved the $book book, so....
    }

}
```

Then register it with:

```
Book::observe(new BookObserver);
```

In the `boot()` method of the `EventServiceProvider` class.

There is nothing more to it, the concept is exactly the same. With observers you can also use every single notion you learned before with model events. You can declare every method you want, and to bind a specific event, just use the event identifier for the method name. So, the *creating* event will be related to the `creating()` method and so on.

Obviously, you can also abort an operation if you are using the *pre* methods, such as *creating* or *updating*:

```
class BookObserver {

  public function creating($book)
  {
    $somethingGoesWrong = true;

    if($somethingGoesWrong)
    {
      return false;
    }
  }

}
```

Alright, now let's look at a couple of examples using model observers!

An example of model observers

First of all, here's how you can do the same thing you did in the first model events example, using observers.

Create a new file named `WelcomeUserObserver.php` under `app/Observers`. Now, type in the following:

```
<?php

namespace App\Observers;

class WelcomeUserObserver {

  public function created($user){

    Mail::send('emails.welcome', ['user' =>
$user], function($message) use ($user)
    {
        $message->to($user->email,
$user-
>first_name . ' ' .
$user->last_name)-
>subject('Welcome to My Awesome App,
'.$user->first_name.'!');
    });

  }

}
```

Then, you can register the observer in the `boot()` method of `EventServiceProvider`:

```
/**
 * Register any other events for your
application.
 *
 * @param
\Illuminate\Contracts\Events\Dispatcher  $events
 * @return void
 */
public function boot(DispatcherContract $events)
{
    parent::boot($events);

    User::observe(new WelcomeUserObserver);
}
```

Ta-dah! You're done. Your observer is now attached to your model.

Let's imagine another situation now. After a meeting with developers, it comes out that the librarian needs some small introduced in to the code base:

- A notification for every user when a new author is added to the system
- An e-mail sent every time an author is added or deleted

Finally, every time a book is deleted, the librarian must know how many authors are stored in the database without a related book.

Good. Let's start. We will build three separate classes: remember the *Single Responsibility Principle*, my friend.

We will have:

- `CustomerNewAuthorObserver`
- `LibrarianAuthorObserver`
- `AuthorsWithoutBooksObservers`

> **Note**
>
> You can name your classes as you prefer. I used this convention just as an example to easily link the behavior to the chosen name.

Then, let's create the three separate classes:

```
<?php

// file: app/Observers/CustomerNewAuthorObserver

namespace App\Observers;

class CustomerNewAuthorObserver {

  public function created($author)
  {

  }

}

<?php

// file: app/Observers/LibrarianAuthorObserver
```

```
namespace App\Observers;

class LibrarianAuthorObserver {

  public function created($author)
  {

  }

  public function deleted($author)
  {

  }

}

<?php

// file:
app/Observers/AuthorsWithoutBooksObservers

namespace App\Observers;

class AuthorsWithoutBooksObservers {

  public function deleted($author)
  {

  }

}
```

Good. Now, it's time to add some logic.

First of all, let's add `CustomerNewAuthorObserver`:

```
<?php

// file: app/Observers/CustomerNewAuthorObserver

namespace App\Observers;

class CustomerNewAuthorObserver {

  public function created($author)
  {
    // getting all users...
    $users = \App\User::all();

    foreach($users as $user)
    {

Mail::send('emails.created_author_customer',
```

```
['author' => $author], function($message) use
($user)
        {
            $message->to($user->email,
$user-
>first_name . ' ' .
$user->last_name)->subject('New Author Added!');
        });
      }
    }

  }
```

> **Note**
>
> A very rude approach, I know. As usual, it's just for teaching purposes. Don't try this at home!

Then, our `LibrarianAuthorObserver` class is as follows:

```
  <?php

  // file: app/Observers/LibrarianAuthorObserver

  namespace App\Observers;

  class LibrarianAuthorObserver {

    public function created($author) {

 Mail::send('emails.created_author_librarian',
['author' => $author], function($message) use
($author)
      {

 $message-
>to('librarian@awesomelibrary.com', 'The
Librarian')-
>subject('New Author: ' . $author->first_name . '
' . $author->last_name);
      });
    }

    public function deleted($author) {

 Mail::send('emails.deleted_author_librarian',
['author' => $author], function($message) use
($author)
      {

 $message-
>to('librarian@awesomelibrary.com', 'The
```

```
Librarian')-
>subject('New Author: ' . $author->first_name . '
' . $author->last_name);
      });
    }

  }
```

Finally, we have the following:

```
  <?php

  // file:
app/Observers/AuthorsWithoutBooksObservers

  namespace App\Observers;

  class AuthorsWithoutBooksObservers {

    public function deleted($author) {
      $authorsWithoutBooks =
\App\Author::has('books', '=', 0)->get();

      if(count($authorsWithoutBooks) > 0){

 Mail::send('emails.author_without_books_librarian',
['authorsWithoutBooks' => $authorsWithoutBooks],
function($message)
        {

 $message-
>to('librarian@awesomelibrary.com', 'The
Librarian')-
>subject('Authors without Books! A check is
required!');
        });
      }
    }

  }
```

> **Note**
>
> As mentioned earlier, I am assuming that you have all the needed views and know how to deal with e-mails. If not, take a look at the http://laravel.com/docs/5.0/mail#basic-usage page.

It doesn't end here. You can use observers and events for a vast number of cases and scenarios. Just to take an example, imagine that you are writing a blog and you want to regenerate your sitemap every time you create or

edit an article. Observers are the answer, or maybe you want to log something while you add new books—use event observers, again!

Summary

Great! You are now able to deal with events in every form, starting from the very basic to the more advanced concepts of observers. You just added another little piece to your Eloquent knowledge: the further you go, the more you will learn about how to make sophisticated applications. Also, we're respecting some of the SOLID principles!

Not bad, huh? However, don't use observers and events for everything. Sometimes, they are not the best choice, and you have to use other tools. So, be careful and analyze the individual problem you want to solve. A good technique never applies well to everything.

Well, time to take another step forward. If you want, take some rest; the *intermediate* part of our work is done. In the next two chapters, in fact, you will learn some advanced things.

Are you ready? Great!

Turn the page and learn how to use Eloquent without Laravel!

Chapter 7. Eloquent… without Laravel!

Our journey is close to the end, hero. You learned everything about Eloquent, starting from the very basics and going through models, relationships, and other topics. You probably started to like it and think about implementing it in your next project.

In fact, creating an application without a single SQL query is tempting. Maybe you also showed it to your boss and convinced him/her to use it in your next production project.

I am so proud of you, hero!

However, there is a little problem. Yeah, the next project isn't so new. It already exists, and, despite everything, it doesn't use Laravel! You start to shiver. This is so sad because you passed the last week studying this new ORM, a really cool one, and then moving forward.

Ok, stop complaining! There is always a solution! You are a developer! Also, the solution is not so hard to find. If you want, you can use Eloquent without Laravel. Yes, seriously!

Actually, Laravel is not a *monolithic* framework. It is made up of several, separate parts, which are combined together to build something greater. However, nothing prevents you from using only selected packages in another application.

A really cool idea!

So, what are we going to see in this chapter?

First of all, we will explore the structure of the database package and see what is inside it. Then, you will learn how to install the `illuminate/database` package separately for your project and how to configure it for the first use.

Then, you will encounter some examples. First of all, we will look at the **Eloquent ORM**. You will learn how to define models and use them.

Having done this, as a little extra, I will show you how to use the `Query Builder` (remember that the `"illuminate/database"` package isn't just Eloquent). Maybe you would also enjoy the `Schema Builder` class. I will cover it, don't worry!

À la charge!

We will cover the following:

- Exploring the directory structure
- Installing and configuring the database package
- Using the ORM
- Using the Query and Schema Builders
- Summary

Exploring the directory structure

As I mentioned before, the key step in order to use Eloquent in your application without Laravel is to use the `"illuminate/database"` package.

So, before we install it, let's examine it a little.

You can see the package contents here: https://github.com/illuminate/database.

So, this is what you will probably see:

Folder	Description
`Capsule`	The **capsule manager** is a fundamental component. It instantiates the service container and loads some dependencies.
`Connectors`	The database package can communicate with various DB systems. For instance, SQLite, MySQL, or PostgreSQL. Every type of database has its own connector. This is the folder in which you will find them.
`Console`	The database package isn't just Eloquent with a bunch of connectors. In this specific folder, you will find everything related to console commands, such as `artisan db:seed` or `artisan migrate`.
`Eloquent`	Every single Eloquent class is placed here.

Folder	Description
`Migrations`	Don't confuse this with the `Console` folder. Every class related to migrations is stored here. When you type `artisan migrate` in your terminal, you are calling a class that is placed here.
`Query`	The Query Builder is placed here.
`Schema`	Everything related to the Schema Builder is placed here.

In the main folder, you will also find some other files. However, don't worry as you don't need to know what they are.

If you open the `composer.json` file, take a look at the following `"require"` section:

```
"require": {
  "php": ">=5.4.0",
  "illuminate/container": "5.1.*",
  "illuminate/contracts": "5.1.*",
  "illuminate/support": "5.1.*",
  "nesbot/carbon": "~1.0"
},
```

As you can see, the database package has some prerequisites that you can't avoid. However, the container is quite small, and it is the same for `contracts` (just some interfaces) and `"illuminate/support"`.

> **Note**
>
> Eloquent uses `Carbon` (https://github.com/briannesbitt/Carbon) to deal with dates in a smarter way. So, if you are seeing this for the first time and you are confused, don't worry! Everything is all right.

Now that you know what you can find in this package, let's see how to install it and configure it for the first time.

Installing and configuring the database package

Let's start with the setup. First of all, we will install the package using composer as usual. After that, we will configure the capsule manager in order to get started.

Installing the package

Installing the `"illuminate/database"` package is really easy.

All you have to do is to add `"illuminate/database"` to the `"require"` section of your `composer.json` file, like this:

```
"require": {


  "illuminate/database": "5.0.*",

},
```

Then type `composer update` in to your terminal, and wait for a few seconds.

Another way is to include it with the shortcut in your project folder, obviously from the terminal:

```
composer require illuminate/database
```

No matter which method you chose, you just installed the package.

Configuring the package

Time to use the capsule manager! In your project, you will use something like this to get started:

```
use Illuminate\Database\Capsule\Manager as
Capsule;

$capsule = new Capsule;

$capsule->addConnection([
  'driver'    => 'mysql',
  'host'      => 'localhost',
  'database'  => 'database',
  'username'  => 'root',
  'password'  => 'password',
```

```
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
]);


// Set the event dispatcher used by Eloquent
models... (optional)
use Illuminate\Events\Dispatcher;
use Illuminate\Container\Container;
$capsule-
>setEventDispatcher(new Dispatcher(new
Container));
```

The config syntax I used is exactly the same you can find in the `config/database.php` configuration file. The only difference is that this time you are explicitly using an instance of the capsule manager in order to do everything.

In the second part of the code, I am setting up the event dispatcher. You must do this if events are required from your project.

However, events are not included by default in this package, so you will have to manually add the `"illuminate/events"` dependency to your `composer.json` file.

Now, the final step!

Add this code to your setup file:

```
// Make this Capsule instance available globally
via static methods... (optional)
$capsule->setAsGlobal();

// Setup the Eloquent ORM... (optional; unless
you've used setEventDispatcher())
$capsule->bootEloquent();
```

With `setAsGlobal()` called on the capsule manager, you can set it as a global component in order to use it with static methods. You may like it or not; the choice is yours. The final line starts up Eloquent, so you will need it.

However, this is also an optional instruction. In some situations you may need the Query Builder only.

Then there is nothing else to do! Your application is now configured with the database package (and Eloquent)!

Using the ORM

Using the Eloquent ORM in a non-Laravel application is not a big change. All you have to do is to declare your model as you are used to doing. Then, you need to call it and use it as you are used to.

Here is a perfect example of what I am talking about:

```
use Illuminate\Database\Eloquent\Model;

class Book extends Model {

  ...

  // some attributes here…
  protected $table = 'my_books_table';

  // some scopes here...
  public function scopeNewest()
  {
    // query here...
  }

  ...

}
```

Exactly as you did with Laravel, the package you are using is the same. So, no worries! If you want to use the model you just created, then use the following:

```
$books = Book::newest()->take(5)->get();
```

This also applies for relationships, observers, and so on. Everything is the same.

> **Note**
>
> In order to use the database package and ORM exactly, you would do the same thing you did in Laravel; remember to set up the project structure in a way that follows the PSR-4 autoloading convention.

Using the Query and Schema Builder

It's not just about the ORM; with the database package, you can also use the Query and the Schema Builders. Let's discover how!

The Query Builder

The Query Builder is also very easy to use. The only difference, this time, is that you are passing through the capsule manager object, like this:

```
$books = Capsule::table('books')
            ->where('title', '=', "Michael
Strogoff")
            ->first();
```

However, the result is still the same.

Also, if you like the DB facade in Laravel, you can use the capsule manager class in the same way:

```
$book = Capsule::select('select title,
pages_count from books where id = ?', array(12));
```

The Schema Builder

At the beginning of this book, I showed you the Schema Builder. You learned how to use it with migrations, but now, without Laravel, you don't have migrations.

However, you can still use the Schema Builder. Like this:

```
Capsule::schema()-
>create('books', function($table)
{
    $table->increments(''id');
    $table->string(''title'', 30);
    $table->integer(''pages_count'');
    $table->decimal(''price'', 5, 2);.
    $table->text(''description'');
    $table->timestamps();
});
```

Previously, you used to call the `create()` method of the `Schema facade`. This time is a little different: you will use the `create()` method, chained to the `schema()` method of the `Capsule` class.

Obviously, you can use any Schema class method in this way. For instance, you could call something like following:

```
Capsule::schema()->table('books', function($table)
{
    $table->string('title', 50)->change();
    $table->decimal('special_price', 5, 2);
});
```

And you are good to go!

> **Note**
>
> Remember that if you want to unlock some Schema Builder-specific features you will need to install other dependencies.
>
> For example, do you want to rename a column? You will need the `doctrine/dbal` dependency package.

Summary

Yeah, this time, it was quite quick.

I decided to add this chapter because many people ask me how to use Eloquent without Laravel. Mostly because they like the framework, but they can't migrate an already started project in its entirety.

Also, I think that it's cool to know, in a certain sense, what you can find under the hood.

It is always just about curiosity. Curiosity opens new paths, and you have a choice to solve a problem in a new and more elegant way.

In these few pages, I just scratched the surface. I want to give you some advice: explore the code. The best way to write good code is to read good code.

And now, to the final chapter!

Chapter 8. It's Not Enough! Extending Eloquent, Advanced Concepts

Through this book, you learned that you can do many amazing things with Eloquent. It is a great active record implementation; it's easy to use, really flexible, and it offers many tools out of the box to improve your code base quality.

A developer usually must face two types of projects: *Applications* and *applications*, as follows:

- For *applications*, I intend something that you can do, maybe in a quick way, with some workarounds and some hacks, here and there. Also, I know that you know what I am talking about. That site you made for a friend, a little blog, and so on.
- Let's be clear and serious, you can't make every application with the same, perfect care. Being honest, neither me nor probably any one does. Then, you have to deal with *Applications*. Things can get really serious there, and you must be able to build a maintainable, awesome application structure.

It's not just about something that works; in this case, I am thinking about something that can scale easily, with a good quality code base. It's not just about calling your models from controllers. That's not enough. You could stumble upon many issues: testability, maintainability, and also in following some principles.

In this chapter, we will explore two different ways to extend Eloquent more seriously.

In the very first part, you will learn how to extend the Eloquent Model class. Actually, the Model does many things, but what if we need something more? No problem: extending the existing Model class will be a bed of roses if standing on the shoulders of a giant.

Have you ever heard about the Ardent Project? It's a package for Laravel 4 that extends the Model class, adding some super powers: self-validating models and auto-hydrating from the request input data.

You'll make something similar for the Laravel 5 Eloquent Model, and I will show you how to do it step by step. Also, it is inspired by the work of Philip Brown in his blog (http://culttt.com/2013/08/05/extending-eloquent-in-laravel-4/).

However, as mentioned earlier in this book, Laravel is mostly about freedom, especially when it comes to freedom in organizing your projects. Now, a really interesting trend is the **repository pattern**. It's a way to abstract your code in a better way and separate responsibilities. For a project bigger than the bakery's blog, it's a must-have in your knowledge base.

Also, the repository pattern is not something related to Laravel only. This means that you will learn something new that you will be able to reuse in the future, with other languages and products.

Alright, no more chitchat. It is time to get our hands dirty for the last time.

Come on, hero! We will cover the following topics:

- Extending the Model: Aweloquent !
- Diving into the repository pattern
- The summary

Extending the Model: Aweloquent!

The Eloquent Model can actually do tons of things in a very smart and easy way. However, something can be improved in terms of code to write every time you want to do a specific operation.

Usually, when creating a new model instance, you are probably using some data that the user previously typed in to a form.

Adding a new author to our database can be the perfect example. All you have to do is to insert the first and last names in to a form and then press **save**.

Then, in the dedicated post route (or relative controller method), you will do something similar to the following:

```
<?php


public function postAdd(Request $request)
{
  $author = new Author;

  $author->first_name =
$request->input('first_name');
  $author->last_name =
$request->input('last_name');

  $author->save();
```

```
}
```

That's quite fine. However, you will probably also have to validate the user input.

So, assuming that you are still in a controller, you could add a controller validator call, just like this one:

```
<?php


public function postAdd(Request $request)
{
  $this->validate($request, [
    'first_name' => 'required',
    'last_name' => 'required'
  ], [
    'first_name.required' => 'You forgot the
first name!',
    'last_name.required' => 'You forgot the last
name!'
  ]);

  $author = new Author;

  $author->first_name =
$request->input('first_name');
  $author->last_name =
$request->input('last_name');

  $author->save();
}
```

Once again, saved the day!

Now, very often, developers debate the responsibilities of a single class in terms of what that class has to do and what not. Everyone has a thought about the topic.

The **Single Responsibility Principle**, a part of the SOLID principles, is very clear about that—put simply, the principle says that a class should do one and only one thing.

On the other hand, however, you will often find very big classes. The Eloquent Model is one of them. At the time of writing, the Illuminate \Database\Eloquent\Model counts 3,399 lines of code. Not exactly something small!

Obviously, the Model doesn't perform a single operation; it fills its own attributes, deals with relationships, and serializes its own attributes. Yes, it goes far beyond the principle you just read.

So, what's the deal with it?

Well, even if it is very big, a Model like this allows you to perform many operations using a single class.

A perfect example is how you can use a Model as a model, like this:

```
<?php

$user = new User;

// using magic methods...
$user->first_name = 'Francesco';
$user->last_name = 'Malatesta';

...
```

You can also use it as a factory (a class that is used to create instances in a more elegant and better way) using the `create()` method:

```
<?php

$user = User::create([
  'first_name' => 'Francesco',
  'last_name' => 'Malatesta',
]);
```

If this isn't enough, the Model also handles everything related to the persistence of the instance:

```
<?php

$user = new User;

// some assignments...
$user->first_name = 'Francesco';
// ...

// and then save!
$user->save();
```

All using a single class—that's the main advantage.

You are probably asking yourself what he is trying to say with all this stuff. The answer is really simple: there isn't always a single correct solution. Some people hate the Eloquent Model, some people love it.

So, in this specific situation, I will add new features to the existing Model class creating a new Eloquent Model… **the Aweloquent**!

> **Note**
>
> Before we go any further, here's a clarification. I will repeat it again, but I also want to say it now. In the following part of this chapter, we will extend the Model class adding a feature that, in Laravel, is handled by the `Validator` class. I am not teaching you this because I want you to do this, but because I want to show you how to extend the Model class.
>
> For instance, you will probably find the **smart password hashing** feature stupid, but it's just an example. Extending Models and using repositories are two different techniques, which are totally separated. I am just giving you the knowledge, then you can choose what to do, and I am sure you will do the right thing, hero!

The Aweloquent Model

As I mentioned earlier, our powered-up Eloquent Model will be really similar to the Ardent Laravel 4 package-improved Model. I will also borrow some ideas from Philip Brown's Magniloquent project.

To be more precise, our improved Model will feature the following:

- Auto Hydrate
- Model self-validation
- Smart password hashing
- The autopurge of confirmation fields

Auto Hydrate

Instead of assigning attributes individually, or passing them in an array, the Aweloquent Model will be able to read the current request and autopopulate its attributes without any other code lines.

This means that you will be able to use the following:

```
<?php

$user = new User;
```

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

Instead of the more classic:

```
<?php

$user = new User;

$user->first_name = 'Francesco';
$user->last_name = 'Malatesta';
// other assignments here...

$user->save();
```

Model self-validation

You will be able to specify validation rules and messages as static properties of a Model. Then, the Model will automatically perform the validation operation you need, without using any external classes or controller validators. You will be able, also, to assign a certain rule to a certain operation (the `'create'` or the `'update'` operation, or both).

So, in your Model, you will have something like this:

```
<?php namespace App;

use App\Aweloquent\AweloquentModel;

class Author extends AweloquentModel {

  protected $fillable = [
    'first_name', 'last_name', 'bio'
  ];

  protected static $rules = [
    'everytime' => [
      'first_name' => 'required'
    ],

     'create' => [
      'last_name' => 'required'
    ],

    'update' => [
      'bio' => 'required'
    ],
  ];

  protected static $messages = [
    'first_name.required' => 'You forgot the
first name!',
```

```
        'last_name.required' => 'You forgot the last
    name!',
        'bio.required' => 'You forgot the biography!'
      ];

    }
```

Smart password hashing

Another thing you have to do frequently is to **hash a password**. Usually, you take the value from a `password` attribute. So, the Aweloquent Model automatically performs the hash operation on a `password` field, if present.

The autopurge of confirmation fields

The Laravel validator has the confirmed rule, based on an `x_confirmation` attribute (where `x` is the name of the field). You have probably used it used it for a password confirmation field. The auto purge feature of the Aweloquent Model automatically removes (after validation, of course) every `_confirmation` field.

However, it's not over yet! The Aweloquent Model will automatically exclude the `'_token'` field, used by the **Cross Site Request Forgery** (**CSRF**) protection middleware.

Alright, that's all! Now you can write some code.

Extending the class

The first thing you have to do is to create a new class, the `AweloquentModel` class that extends the existing Eloquent Model one.

In my specific case, I made something really simple: I created a new folder called `Aweloquent` in the `app` one and then created an `AweloquentModel.php` file inside.

Here's the code you have to put into this file:

```
<?php

namespace App\Aweloquent;

use Illuminate\Database\Eloquent\Model;

class AweloquentModel extends Model {}
```

Great! As a start, we have our new `AweloquentModel` class.

If you want, you can use it as a base for your future models. There are no changes here, just a simple extension.

Let's add the first feature: Auto Hydrate.

The Auto Hydrate feature

Before we implement this first feature, let's think about what we want as a result.

Actually, when you create a new model, you can quickly pass its attributes using the constructor:

```
<?php

$user = new User([
  'first_name' => 'Francesco',
  'last_name' => 'Malatesta'
]);
```

These parameters are passed from the constructor to another method called `fill()`:

```
/**
   * Create a new Eloquent model instance.
   *
   * @param  array  $attributes
   * @return void
   */

  public function __construct(array $attributes =
array())
  {
    $this->bootIfNotBooted();

    $this->syncOriginal();

    $this->fill($attributes);
  }
```

As a logical consequence, if we want to implement this Auto Hydrate feature, we will have to write a new constructor to deal with the auto hydrating there and then to call the parent class. So, let's go back to our `AweloquentModel` class. Here's the first implementation:

```
<?php

namespace App\Aweloquent;
```

```
use Illuminate\Database\Eloquent\Model;

class AweloquentModel extends Model {

  public function __construct(array $attributes =
[])
  {
    $attributes = $this->autoHydrate($attributes);

    parent::__construct($attributes);
  }

  private function autoHydrate(array $attributes)
  {
    // getting the request instance using the
service container
    $request = app('Illuminate\Http\Request');

    // getting the request form data, except the
token
    $requestData = $request->except('_token');

    foreach($requestData as $name => $value)
    {
      // manually specified attribute has
priority over auto-
   hydrated one.
      if(!isset($attributes[$name]))
      $attributes[$name] = $value;
    }

    return $attributes;
  }
}
```

The `autoHydrate` method creates the following:

- An instance of the current request to get the required data
- After that, and for each cycle, it adds to the attributes array every element in the request data array (excluding the CSRF `'_token'`)

Note that the explicit specified attribute (the one you can put in the Model constructor) has the priority over the request data array. So, you are still free to deal with the Model and decide what to define and what not, maybe to add some extra data that you are not getting from the form.

If you try to create a new user by setting up a basic form, the Auto Hydrate feature is already working.

Let's move forward!

The Aweloquent Model self-validation feature – the basic version

It is time to implement the self-validation feature of our Aweloquent Model. The idea is quite simple: for every model, you will be able to declare (as properties) rules and related messages. So, it should look something like this:

```
<?php namespace App;

use App\Aweloquent\AweloquentModel;

class Author extends AweloquentModel {

  protected $fillable = [
    'first_name', 'last_name', 'bio'
  ];

  protected static $rules = [
    'first_name' => 'required',
    'last_name' => 'required'
  ];

  protected static $messages = [
    'first_name.required' => 'You forgot the
first name!',
    'last_name.required' => 'You forgot the last
name!'
  ];

}
```

These rules and message will be used automatically by a `validate()` dedicated method. What I want to achieve is something like this:

```
<?php

$user = new User;

if(!$user->validate())
{
  dd($user->errors);
}
```

So, let's open the `AweloquentModel.php` file and add some code:

```
<?php

namespace App\Aweloquent;
```

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\Validator;

class AweloquentModel extends Model {

  protected static $rules = [];
  protected static $messages = [];

  public $errors;

  public function __construct(array $attributes =
[])
  {
    $attributes = $this->autoHydrate($attributes);

    parent::__construct($attributes);
  }

  public function validate()
  {
    $validator =
Validator::make($this-
>attributes, static::$rules, static::$messages);

    if($validator->fails())
    {
      $this->errors = $validator->messages();
      return false;
    }

    return true;
  }

  private function autoHydrate(array $attributes)
  {
    // auto hydrate method here...
  }

}
```

Great! The `Validator Facadec` is used to instantiate a new validator. The static `$rules` and `$message` attributes are used for the `make()` method.

Then, the `$validator->fails()` call determines if the given model is valid or not. If not, the `$errors` property is populated using the validation errors `MessageBag` object.

Obviously, this is a very basic validation system. However, we could do something more. For instance, an implementation of an operation-based validation would be great.

Go ahead and try it, if you want! It already works!

The Aweloquent Model self-validation feature – the operation-based version

In order to implement the advanced version of the self-validation system, we have to define the rule format for every model.

In this specific one, I have chosen something like this:

```
<?php namespace App;

use App\Aweloquent\AweloquentModel;

class Author extends AweloquentModel {

  protected $fillable = [
    'first_name', 'last_name', 'bio'
  ];

  protected static $rules = [
    'everytime' => [
      'first_name' => 'required'
    ],

    'create' => [
      'last_name' => 'required'
    ],

    'update' => [
      'bio' => 'required'
    ],
  ];

  protected static $messages = [
    'first_name.required' => 'You forgot the
first name!',
    'last_name.required' => 'You forgot the last
name!',
    'bio.required' => 'You forgot the biography!'
  ];

}
```

The `$message` property is left untouched. The only one that has to be modified is `$rules`, as you may easily imagine.

In this new version of `$rules`, you can define rules for a single operation, `'create'`, or both. If you want to use a rule in both of them, there is a dedicated `'everytime'` item to avoid the duplication of rules.

Of course, we have to edit our `AweloquentModel` once again. This time, we have to define a method that has to work with the existing validate one and understand if we are creating or updating it.

Then, merge the right rules in a single array and validate the model against those rules.

Let's see what we can do! Consider the following code:

```
<?php



class AweloquentModel extends Model {

  ...

  public function __construct(array $attributes =
[])
  {
    // constructor remains the same…
  }

  public function validate()
  {
    static::$rules =
$this->mergeValidationRules();

    $validator =
Validator::make($this-
>attributes, static::$rules, static::$messages);

    if($validator->fails())
    {
      $this->errors = $validator->messages();
      return false;
    }

    return true;
  }

  private function mergeValidationRules()
  {
    // if updating, use "update" rules, "create"
otherwise.
    if($this->exists)
      $mergedRules =
array_merge_recursive(static::
$rules['everytime'], static::$rules
      ['update']);
  else
```

```
        $mergedRules =
    array_merge_recursive(static::
    $rules['everytime'], static::$rules

        ['create']);

      $finalRules = [];

      foreach($mergedRules as $field => $rules){
        if(is_array($rules))
          $finalRules[$field] = implode("|", $rules);
        else
          $finalRules[$field] = $rules;
        }

        return $finalRules;
      }

    }
```

Great, we have it!

The `validate()` method doesn't change too much. The only big difference stands in the newly line:

```
static::$rules = $this->mergeValidationRules();
```

Basically, we are saying, "Ok, now assign to the `$rules` property the result of this `mergeValidationRules()` method."

Then, in the `mergeValidationRules()` method, the first instruction to be used is:

```
if($this->exists)
```

That is used to determine if the current operation is an insert or an update. Starting from this value, we can get the right rules array, merging them with the `everytime` rules.

Your new complex self-validating model is almost ready to be used.

Smart password hashing and the confirmation fields autopurge method

The last features we have to implement are the smart password hashing and confirmation fields auto purge methods.

The first is very easy and intuitive:

```
private function smartPasswordHashing()
{
  if($this->attributes['password'])
    $this->attributes['password'] =
Hash::make($this- >attributes['password']);
}
```

If a `'password'` field is present, hash it. Nothing more!

Even if, it's a little longer, the `purgeConfirmationFields()` isn't so hard to understand:

```
private function purgeConfirmationFields()
{
  foreach($this->attributes as $name => $value)
  {
    if(Str::endsWith($name, '_confirmation'))
      unset($this->attributes[$name]);
  }
}
```

This time, I used the `Str` string utility class in order to use the `endsWith()` method, which is used to determine if a strings ends with a certain sequence of characters. Every `'_confirmation'` field is removed.

Fixing the save() Model method

Now, the last thing that we need to fix is the `save()` method. Actually, the `save()` method totally ignores the validation procedure, and this is no good. So, this is my final version of the `AweloquentModel` class:

```
<?php

namespace App\Aweloquent;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Validator;
use Illuminate\Support\Str;

class AweloquentModel extends Model {

  protected static $rules = [];
  protected static $messages = [];

  public $errors;

  public function __construct(array $attributes =
[])
  {
```

```
    $attributes = $this->autoHydrate($attributes);

    parent::__construct($attributes);
  }

  public function save(array $options = [])
  {
    if($this->validate())
    {
      $this->smartPasswordHashing();
      $this->purgeConfirmationFields();

      return parent::save($options);
    }
    else
      return false;
  }

  public function validate()
  {
    static::$rules =
$this->mergeValidationRules();

    $validator =
Validator::make($this-
>attributes, static::$rules, static::$messages);

    if($validator->fails())
    {
      $this->errors = $validator->messages();
      return false;
    }

    return true;
  }

  private function autoHydrate(array $attributes)
  {
    // getting the request instance using the
service container
    $request = app('Illuminate\Http\Request');

    // getting the request form data, except the
token
    $requestData = $request->except('_token');

    foreach($requestData as $name => $value)
    {
      // manually specified attribute has
priority over auto-
   hydrated one.
      if(!isset($attributes[$name]))
        $attributes[$name] = $value;
    }
```

```
    return $attributes;
  }

  private function mergeValidationRules()
  {
    // if updating, use "update" rules, "create"
otherwise.
    if($this->exists)
      $mergedRules =
array_merge_recursive(static::
$rules['everytime'], static::$rules

      ['update']);
    else
      $mergedRules =
array_merge_recursive(static::
$rules['everytime'], static::$rules

    ['create']);

    $finalRules = [];

    foreach($mergedRules as $field => $rules){
      if(is_array($rules))
        $finalRules[$field] = implode("|",
$rules);
      else
        $finalRules[$field] = $rules;
    }

    return $finalRules;
  }

  private function smartPasswordHashing()
  {
    if($this->attributes['password'])
      $this->attributes['password'] =
Hash::make($this- >attributes['password']);
  }

  private function purgeConfirmationFields()
  {
    foreach($this->attributes as $name => $value)
    {
      if(Str::endsWith($name, '_confirmation'))
        unset($this->attributes[$name]);
    }
  }
}
```

Let's analyze the `save()` method in detail:

```
public function save(array $options = [])
{
```

```
    if($this->validate())
    {
      $this->smartPasswordHashing();
      $this->purgeConfirmationFields();

      return parent::save($options);
    }
    else
      return false;
  }
```

The first thing to do is to validate the entire input. After that, if everything is fine, passwords are hashed and the confirmation fields are purged, as we don't need them anymore.

Finally, the `parent::save()` method is called, and the operation is complete.

> ## Note
>
> In order to create a perfect continuity with the parent class, I declared the `save()` method with the same signature as its parent (including the `$options` array parameter).

That's it! The `AweloquentModel` class is finished, and you can use it as you want in your projects. You also learned how to go deep inside the `Model` class and extend it in order to add new methods, behaviors, and features.

Diving into the repository pattern

If you know a couple of things about good development and best practices, you have probably heard about software design patterns.

You can define them as useful solution templates for a certain kind of problem, or to be more precise:

"In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system."

> **Note**
>
> This extract is from the software design patterns page (http://en.wikipedia.org/wiki/Software_design_pattern) on Wikipedia.

Now, let's focus on the second sentence.

A design pattern is not something that can be transformed directly into source code.

That's the most important part because it explains many things. It is not something you learn specifically for Laravel or maybe for a certain language.

Absolutely, once you learned about design patterns, it is for life!

In the previous part of this chapter, you learned how to create an improved version of the Eloquent Model. You reached the objective by adding something to the existing model.

However, many people don't like that kind of approach. They strongly believe in the Single Responsibility Principle, so every class must do one and only one thing. Nothing more!

Let's be clear; I don't want to bore you by adding my useless opinion in this long-time debate.

In the final part of this chapter, I will introduce a specific design pattern that can be really useful for improving a Laravel application—the repository pattern.

Hello, repository pattern!

You know that I like to explain a concepts with an example as a start. This is no exception.

Imagine that you are building an application for a warehouse. In this warehouse, you can store whatever you want and you probably have a model like this:

```
<?php namespace Warehouse;

use Illuminate\Database\Eloquent\Model;

class Item extends Model {

  // properties and methods here...

}
You are probably using this model in a
controller, like this one:
<?php namespace Warehouse\Http\Controllers;

class ItemsController extends Controller {

  public function getIndex()
  {
    $items =
\Warehouse
\Item::orderBy('created_at', 'DESC')-
>paginate(30);

    return view('item.list', compact('items'));
  }

}
```

There's nothing to say; it works, and you know it.

This is a cool solution for a simple project, the kind of project that you can call an *application*. However, what happens if we don't have an *Application*, instead?

Imagine that the business you're helping grows and the management decides to create a mobile application that must be used internally. You will probably need to implement an API for the other developers.

If you don't know how to deal with something like this, you will soon start to write duplicated code. In your Rest API, for sure, there will be an endpoint `\items` that does the same thing you did in the controller method such as:

```
$items = \Warehouse\Item::orderBy('created_at',
'DESC')->paginate(30);
```

Repeating the same code many times isn't safe. You know it, right?

But no fear, hero! The solution is called a **repository pattern**.

The best definition of this concept is the one you can find on Martin Fowler's website (http://martinfowler.com/eaaCatalog/repository.html):

"A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction.

Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes."

So, imagine a repository as something in between that abstracts all the necessary.

Going back to the previous example, imagine that we have a repository with a dedicated method `getRecent($perPage, $pageNumber)`.

We will use the same method in the controller:

```
 <?php namespace Warehouse\Http\Controllers;

class ItemsController extends Controller {

  public function getIndex(ItemsRepository
$itemsRepository)
  {
    // this is an example...
    $items = $itemsRepository->getRecent(30, 1);

    return view('item.list', compact('items'));
  }

}
```

In the Rest API, the following method is used:

```
<?php

Route::get('api/v1/items',
function(ItemsRepository $repo){

  return $repo->getRecent(30, 1);

});
```

The same code is used twice and written once. However, there's more than this; let's see how to implement repositories in a Laravel project.

Introducing repositories – a concrete implementation

The best way to start with repositories is to implement a concrete implementation. As I mentioned earlier, a repository stands between the controller and model. It is something in the middle.

When you build a repository, you must do it thinking about what you will need from the repository. Let's imagine something for our `Author` model.

I will probably need the following methods:

- `getAll($perPage, $pageNumber)`: This returns a paginated list of every author in my data source
- `find($authorId)`: This returns a specific author with a certain primary key
- `search($firstName, $lastName)`: This returns an array of results starting from the first name to the last name

Enough of searching for and getting records! However, we will also need other methods, which are dedicated to data persistence:

- `create($authorData)`: This will save a new author in the data source
- `save($authorData, $authorId)`: This will update an existing author in the data source with a certain primary key

Let's start!

First of all, create a new directory in the `app` folder. Name it `Repositories`. Inside it, create a new file called `DbAuthorsRepository.php`.

Here's the content:

```php
<?php

namespace App\Repositories;

use App\Author;

class DbAuthorsRepository {

  private $model;

  public function __construct(Author $model)
  {
    $this->model = $model;
  }

  public function getAll($perPage, $pageNumber)
  {
    $authors = $this->model->skip(($pageNumber -
1) * $perPage)- >take($perPage)->get();
    return $authors->toArray();
  }

  public function find($authorId)
  {
    return
$this->model->find($authorId)->toArray();
  }

  public function search($firstName, $lastName)
  {
    return $this->model
    ->where('first_name', 'LIKE',
'%'.$firstName.'%')
    ->where('last_name', 'LIKE',
'%'.$lastName.'%')
    ->get()
    ->toArray();
  }

  public function create($authorData)
  {
    return $this->model->create($authorData);
  }

  public function update($authorData, $authorId)
  {
    return
$this->model->find($authorId)-
>update($authorData);
  }

}
```

This is how you can use it in some test routes:

```
<?php

Route::get('authors',
function(\App\Repositories
\DbAuthorsRepository $repository){

  return $repository->getAll(10, 1);

});

Route::get('create_author',
function(\App\Repositories
\DbAuthorsRepository $repository){

  $repository->create([
    'first_name' => 'Francesco',
    'last_name' => 'Malatesta',
    'bio' => 'Lorem ipsum...'
  ]);

});

Route::get('update_author',
function(\App\Repositories
\DbAuthorsRepository $repository){

  $repository->update([
    'first_name' => 'Frank',
    'last_name' => 'Smith',
    'bio' => 'Other ipsum...'
  ], 6);

});
```

Nothing more!

By creating a repository, you learned how to improve your software architecture and the abstraction level of your solution. Also, instead of what you saw earlier in this chapter with Aweloquent, this time you can feel a great separation of responsibilities.

However, this is not the end yet; the repository pattern hasn't yet shown all of its power.

Coding on Abstractions

I already introduced you to the SOLID principles earlier in this chapter. I mentioned the Single Responsibility Principle, the *S* of SOLID. Now that we are close to the end, I will introduce the *D*.

The dependency inversion principle is one of my favorites because it really highlights the importance of abstracting your code base the best you can.

Its definition is:

> *"A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
>
> *B. Abstractions should not depend on details. Details should depend on abstractions."*
>
> *-Wikipedia (http://en.wikipedia.org/wiki/Dependency_inversion_principle)*

In a few words, the concept is that you must code abstractions and should not depend on concrete classes. Else, a better way to say it: you must depend on abstractions and not on concrete classes.

In PHP, talking about abstraction refers to working with interfaces and contracts.

In a certain sense, Laravel itself widely uses this concept. The basic Laravel package is `Contracts`, that is made up of several interfaces that specify whatevery component must do and how it does it.

However, it's not just about a framework big thing. You can apply this principle in your everyday development. More specifically, you can apply the concept to repositories.

I will show you how!

Repositories – a complete implementation

Before we dive in, let's introduce a little problem to our software.

Actually, our situation is the following: our controllers and routes use the `DbAuthorsRepository` class to get data, like this:

```
Route::get('authors',
function(\App\Repositories
\DbAuthorsRepository $repository){

  return $repository->getAll(10, 1);
```

```
});
```

Then, the `DbAuthorsRepository` class uses the Author model in order to get the desired data from the physical storage:

```
<?php

namespace App\Repositories;

use App\Author;

class DbAuthorsRepository {

  private $model;

  public function __construct(Author $model)
  {
    $this->model = $model;
  }

  public function getAll($perPage, $pageNumber)
  {
    $authors = $this->model->skip(($pageNumber -
1) * $perPage)- >take($perPage)->get();
    return $authors->toArray();
  }



}
```

Now, let's imagine that our data sources change. For a certain reason (I know, that is quite paradoxical), the management wants to switch to a file-based storage.

You have two ways to deal with this problem:

- You scream and are paralyzed by fear
- Decide to organize your code base in a better way, introducing interfaces in to your repository workflow

Here's the plan:

Using the Laravel service container, you can decide to bind a certain interface to a specific implementation.

So, if you create an interface for every repository, you will be able to write your code once and then write every concrete repository you need and,

finally, to switch from one repository to another, you will have to change a single line of code.

However, let's make it step by step:

1. First of all, let's define a standard behavior for our author's repository defining an `AuthorRepository` interface. Create a new `AuthorRepository.php` file in `app/Repositories/Contracts`. I will use the `Contracts` folder for interfaces.

   Here're the contents of the fresh file:

   ```
   <?php

   namespace App\Repositories\Contracts;
   interface AuthorsRepository {

     public function getAll($perPage,
   $pageNumber);

     public function find($authorId);

     public function search($firstName,
   $lastName);

     public function create($authorData);

     public function update($authorData,
   $authorId);

   }
   ```

   Interfaces are pure abstraction. All we are saying here is, "When I build a new author repository, I don't care about the underlying implementation. I don't care if I am working with a NoSQL database or a flat file driver. All I want is that every repository implements all these methods."

   Working in this manner means that we can define a standard format for every component (or repository, in this case) that we will use in the future.

2. Now we can update our `DbAuthorsRepository` class in order to implement our interface. Consider the following line:

   ```
   class DbAuthorsRepository {
   It now becomes:
   class DbAuthorsRepository implements
   AuthorsRepository {
   ```

Ok. Now, let's see the power of the entire mechanism in action.

3. First of all, open the `app/Providers/AppServiceProvider.php` file and add this binding to the `register()` method:

```
public function register()
{
  $this->app->bind(

 'App\Repositories\Contracts
\AuthorsRepository',
    'App\Repositories\DbAuthorsRepository'
  );
}
```

   Laravel now knows that every time you will request an instance of `AuthorsRepository`, it will have to create an instance of the `DbAuthorsRepository` using the service container.

4. To test our assumptions, open the routes file and add this:

```
<?php

Route::get('authors',
function(\App\Repositories\Contracts
\AuthorsRepository $repository){

    return $repository->getAll(10, 1);

});
```

   The result will be exactly what we are expecting. Also, using the method injection technique, we don't have to explicitly call the service container.

5. In fact, an alternative to this syntax would be the following:

```
Route::get('authors', function(){

    $repository =
app('App\Repositories\Contracts
\AuthorsRepository');

    return $repository->getAll(10, 1);

});
```

Adding the new repository

Finally, we are close to the end. Let's return to our main problem: we have to implement a new file-based author's repository.

At this point, it is quite easy:

1. First, create a new file in `app/Repositories`, called `FileAuthorsRepository`. It will be a new class, of course.
2. It will implement the AuthorsRepository interface, obviously.

   Here are the class contents:

```
<?php

namespace App\Repositories;

use
App\Repositories\Contracts\AuthorsRepository;

class FileAuthorsRepository implements
AuthorsRepository {

  public function getAll($perPage,
$pageNumber)
  {
    dd('getting all records from flat file
driver...');
  }

  public function find($authorId)
  {
    dd('searching by id: ' . $authorId);
  }

  public function search($firstName,
$lastName)
  {
    dd('searching by first and last name...',
$firstName, $lastName);
  }

  public function create($authorData)
  {
    dd('creating new author ', $authorData);
  }

  public function update($authorData,
$authorId)
  {
    dd('updating author ' . $authorId,
$authorData);
  }
```

```
}
```

As you can easily see, I have implemented all the required methods from the interface. This means that our application will be able to use the FileAuthorsRepository in the same way as the DbAuthorsRepository.

3. I filled method bodies with some `dd` instructions, just to show you how the concept works. For our final step, go to the `AppServiceProvider` class and update the previous binding to the following:

```
public function register()
{
  $this->app->bind(

 'App\Repositories\Contracts
\AuthorsRepository',
    'App\Repositories\FileAuthorsRepository'
  );
}
```

4. Now, browse to the `/authors` route. Yes, the output now is:

"getting all records from flat file driver..."

Yes, it worked!

Our route file will never know what repository it is using: the interface defines all the methods you need.

This is fantastic because, for instance, if you want to add a NoSQL repository to your application in the future, all you will have to do is to create a new `NoSQLAuthorsRepository` class that implements the `AuthorsRepository` interface. Then, in the `AppServiceProvider`, you will switch the binding with the one you need.

Easy, cool, and also testable!

Maybe I am a little repetitive, but focus your attention on this specific point: with the mentioned structure, you can abstract the way you work with your data from the way you access to it. I know that reading the same thing again and again is boring, but I need you to understand this concept.

It is probable that the first time you read about repositories you will think *what am I doing here? What the hell?* I have done the same thing, so I totally understand your doubts. However, when you work on a more complex project, you will totally feel the difference.

That's the magic of repositories!

Summary

We are done. In this final chapter, you learned about empowering your application in two different and separate ways: on one hand, adding features on an existing entity. In this case, the Eloquent Model.

On the other one, you learned how to structure your application in a different way using repositories in order to get a better code testability, maintainability, and a separation of purposes instead of delegating everything to a single class.

Now you have all the tools you need to build great applications using Eloquent and Laravel.

What are you waiting for? Go on and make me proud!

Index

A

- accessors
  - about / Accessors and mutators
  - naming convention / Accessors and mutators
- Adminer
  - about / A bonus tool – Adminer
  - URL / A bonus tool – Adminer
- aggregates functions
  - about / Aggregates
- attributes casting
  - about / Attributes casting, accessors, and mutators, Attributes casting
- Aweloquent Model
  - extending / Extending the Model: Aweloquent!
  - about / The Aweloquent Model
  - features / The Aweloquent Model
  - auto hydrate / Auto Hydrate, The Auto Hydrate feature
  - self-validation / Model self-validation
  - smart password hashing / Smart password hashing, Smart password hashing and the confirmation fields autopurge method
  - auto purge feature / The autopurge of confirmation fields, Smart password hashing and the confirmation fields autopurge method
  - self-validation, basic version / The Aweloquent Model self-validation feature – the basic version
  - self-validation, operation-based version / The Aweloquent Model self-validation feature – the operation-based version
  - save() method, fixing / Fixing the save() Model method
- AweloquentModel class
  - extending / Extending the class

B

- Blueprint $table object / Working with columns

C

- capsule manager
    - about / Exploring the directory structure
- Cmder
    - URL / Installing Homestead
- Collection class
    - reference / Basic collection operations
- collections
    - basic operations / Basic collection operations
    - transforming / Transforming collections
    - iterating / Iterating
    - filtering / Filtering
    - sorting / Sorting
- columns, Schema Builder class
    - id column / Working with columns
    - title column / Working with columns
    - pages_count integer column / Working with columns
    - price decimal column / Working with columns
    - description text column / Working with columns
    - created_at column / Working with columns
    - updated_at column / Working with columns
- Composer
    - about / Your Swiss Army Knife – Composer
    - installing / Installing Composer
    - composer.json file / The composer.json and autoload files
    - autoload file / The composer.json and autoload files
    - commands / The most used commands
- create($authorData) method / Introducing repositories – a concrete implementation
- created event / When should I use events in my models?
- create operation / When should I use events in my models?
- creating event / When should I use events in my models?
- Cross Site Request Forgery (CSRF) / The autopurge of confirmation fields

- CRUD (create, read, update, and delete) operations
    - about / Create, read, update, and delete operations basics
    - operations, creating / Creating operations
    - operations, reading / Reading operations
    - operations, updating / Updating operations
    - operations, deleting / Deleting operations

D

- database package
    - installing / Installing the package
    - configuring / Configuring the package
- database versioning, with migrations system
    - about / Database versioning with the migrations system
    - migrations, creating / Creating migrations
    - migrations, rolling back / Rolling back migrations
    - examples / More examples, more migrations
- design pattern
    - about / Diving into the repository pattern
- distant relationships
    - accessing / Accessing distant relationships

E

- eager loading
    - about / Eager loading (and the N + 1 problem)
    - basic eager loading / Basic eager loading
    - advanced eager loading / Advanced eager loading
    - lazy eager loading / Lazy eager loading
- Eloquent, without Laravel
    - directory structure / Exploring the directory structure
    - database package, installing / Installing the package
    - database package, configuring / Configuring the package
- EloquentJourney
    - about / The first project – EloquentJourney

    - configuration system / The configuration system
    - the database connection, setting up / Setting up the database connection

- Eloquent Model
    - about / Extending the Model: Aweloquent!
- Eloquent ORM
    - using, in non-Laravel application / Using the ORM
- events
    - about / When should I use events in my models?
    - in model / When should I use events in my models?
- events observers
    - about / Events observers

F

- filtering
    - about / Filtering
- find($authorId) method / Introducing repositories – a concrete implementation

G

- getAll($perPage, $pageNumber) method / Introducing repositories – a concrete implementation

H

- Homestead
    - about / Your safe place – Homestead
    - installing / Installing Homestead
    - configuring / Configuring Homestead
    - improved version / The new hideout: Homestead improved
- Homestead installation
    - about / Installing Homestead
    - Composer and PHP tool way / The Composer and PHP tool way

    - Git way / The Git way

I

- iterating
    - about / Iterating

L

- Laravel
    - about / Your best friend: Laravel
    - installing / Installing Laravel
    - URL, for documentation page / Using the Composer create-project command
- Laravel installation
    - about / Installing Laravel
    - Laravel installer tool, used / Using the Laravel installer tool
    - Composer create-project command, used / Using the Composer create-project command
- Laravel official documentation
    - URL / Columns' methods reference

M

- magic where feature / Magic wheres
- many-to-many relationship
    - about / Many to many
- mass assignment
    - about / Mass assignment... for the masses
- model
    - creating / Creating a Model
- model class
    - code, descending / Descending in the code
    - length / A big file

    - conversion, to array or JSON / Quick conversion to array or JSON
    - toArray method / Quick conversion to array or JSON
    - toJson method / Quick conversion to array or JSON
    - imaginary attributes / Imaginary attributes
    - route model binding / Route model binding
    - records, chunking for memory optimization / Records chunking for memory optimization

- model events
    - about / Model events
    - example / An example of model events

- model observers
    - about / Events observers
    - example / An example of model observers

- mutators
    - about / Accessors and mutators

O

- one-to-many relationship
    - about / One to many

- one-to-one relationship
    - about / One to one, What exactly happened?

- orWhere() method / where and orWhere

P

- polymorphic relationships
    - about / More power – polymorphic relationships
    - simple polymorphic relationship / A simple polymorphic relationship
    - many-to-many polymorphic relationship / A many-to-many polymorphic relationship

Q

- Query Builder
  - using / The Query Builder
- querying-related models
  - about / Querying-related models
  - pivot table, accessing / Accessing a pivot table
  - relationship, querying / Querying a relationship
- query scopes
  - about / Query scopes

R

- related models
  - inserting / Inserting and updating related models
  - updating / Inserting and updating related models
  - save() method / The save() and associate() methods
  - associate() method / The save() and associate() methods
  - many-to-many relationship / What about many to many?
  - sync() method / The sync() method
- relationships
  - inverses / A question of inverses
- repository pattern
  - about / Diving into the repository pattern, Hello, repository pattern!
  - reference / Hello, repository pattern!
  - concrete implementation / Introducing repositories – a concrete implementation
  - coding on abstractions / Coding on Abstractions
  - complete implementation / Repositories – a complete implementation
  - new repository, adding / Adding the new repository

S

- save($authorData, $authorId) method / Introducing repositories – a concrete implementation
- save operation / When should I use events in my models?

- Schema Builder
    - using / The Schema Builder
- Schema Builder class
    - about / The Schema Builder class
    - tables, working with / Working with tables
    - create() method / Working with tables
    - columns, working with / Working with columns
    - columns methods reference / Columns' methods reference
    - $table object methods / Other $table object methods
    - tables and columns, updating / Updating tables and columns
    - foreign keys / Indexes and foreign keys
    - indexes / Indexes and foreign keys
- search($firstName, $lastName) method / Introducing repositories – a concrete implementation
- Single Responsibility Principle / Extending the Model: Aweloquent!
- smart password hashing feature
    - about / Extending the Model: Aweloquent!
- soft deleting feature
    - about / Soft deleting
- software design patterns page
    - URL / Diving into the repository pattern
- sorting
    - about / Sorting

T

- timestamps
    - about / Timestamps and soft deletes, Timestamps

U

- utility methods
    - about / Utility methods

V

- Vagrant
    - URL / Installing Homestead
- VirtualBox
    - URL / Installing Homestead

W

- where() method / where, aggregates, and other utilities