{ WRITING APIs WITH LUMEN }

A HANDS-ON GUIDE TO WRITING API SERVICES WITH PHP

BY PAUL REDMOND

Writing APIs with Lumen

A Hands-on Guide to Writing API Services With PHP

Paul Redmond

Tweet This Book!

Please help Paul Redmond by spreading the word about this book on Twitter!

The suggested hashtag for this book is #lumenapibook.

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

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

To Bernadette.

I would be remiss if I didn’t thank Taylor Otwell for creating Lumen and Laravel, and the community contributions that make the Laravel ecosystem amazing.

Contents

Introduction

Lumen is a framework that is designed to write APIs. With the rising popularity of microservices[1], existing patterns like SOA[2] (Service Oriented Architecture), and increased demand for public APIs, Lumen is a perfect fit for writing the service layer in the same language as the web applications you write.

In my experience, it's not uncommon for PHP shops to write web applications with PHP and API services with something like Node.js[3]. I am not suggesting that this is a *bad* idea, but I see Lumen as a chance to improve development workflows for PHP developers and for companies to standarize around a powerful set of complimentary frameworks: Laravel and Lumen.

You can write APIs quickly with Lumen using the built-in packages provided, but Lumen can also get out of your way and be as minimalist as you want it to be. Set aside framework benchmarks and open your mind to increased developer productivity. Lumen is fast but more importantly, **it helps me be more productive**.

The Same Tools to Write APIs and Web Applications

Lumen is a minimal framework that uses a subset of the same components from Laravel[4]. Together Laravel and Lumen give developers a powerful combination of tools: a lightweight framework for writing APIs and a full-fledged web framework for web applications. Lumen also has a subset of console tools available from Laravel. Other powerful features from Laravel are included like database migrations, Eloquent Models (ORM Package), job queues, scheduled jobs, and a test suite focused on testing APIs.

The development experience between Lumen and Laravel is relatively the same, which means developers will see a productivity boost by adopting both frameworks. Together they provide a consistent workflow and can simplify the software stack for developers, release engineers and operations teams.

Who This Book is For

This book is for programmers that want to write APIs in PHP. Familiarity with the HTTP spec, Composer, PHPUnit, and the command line will help, but this book walks you through each step of building an API. You don't need to be an expert on these subjects, and more experienced developers can skip things they understand to focus on the specific code needed to write APIs in Lumen.

[1]http://microservices.io/patterns/microservices.html
[2]https://en.wikipedia.org/wiki/Service-oriented_architecture
[3]https://nodejs.org/en/
[4]https://laravel.com/

Conventions Used in This Book

The book is a hands-on guide to building a working API and so you will see tons of code samples throughout the book. I will point out a few conventions used so that you can understand the console commands and code. The code is meant to provide a fully working API and so you can follow along or copy and paste code samples.

Code Examples

A typical PHP code snippet looks like this:

Example PHP Code Snippet

```php
/**
 * A Hello World Example
 */
$app->get('/', function () {
    return 'Hello World';
});
```

To guide readers, approximate line numbers are used when you will be adding a block of code to an existing class or test file:

Example PHP Code Snippet

```php
10 /**
11  * A Foobar Example
12  */
13 $app->get('/foo', function () {
14     return 'bar';
15 });
```

Longer lines end in a backslash (\) and continue to the next line:

Example of Long Line in PHP

```php
$thisIsALongLine = 'Lorem ipsum dolor sit amet, consectetur adipisicing elit. Qu\
os unde deserunt eos?'
```

When you need to run console commands to execute the test suite or create files, the snippet appears as plain text without line numbers. Lines start with $ which represents the terminal prompt.

Example Console Command

```
$ touch the/file.php
```

Console commands that should be executed in the recommended Homestead[5] environment will be indicated like the following example. The book removes extra output from PHPUnit tests to make examples less verbose.

Console Command in the Homestead Virtual Machine

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (1 test, 4 assertions)
```

Tips, Notes and Warnings

Your Aside title

This is an aside

Hey, Listen!

Tips give you pointers related to concepts in the book.

Danger!

Warnings point out potential issues and security concerns.

Need the Info

Provides additional info related to code and concepts.

Git Commit: Amazing Refactor!

rm3dwe2f

This is an example of a code commit if you are following along and using git to commit your work.

[5] https://laravel.com/docs/homestead

Discussions

Includes deeper discussions around topics in the book. Advanced users can generally skip these.

Tools You Will Need

All tools are recommended but if you know what you're doing you can set up your own coding environment and skip the recommended tools. You might even have these tools already, just make sure they are relatively up-to-date. **All tools listed are free unless otherwise noted**.

VirtualBox

This book uses a virtual machine to run the API application. You will need to download VirtualBox if you plan on using the recommended Homestead environment. VirtualBox works on Windows, Mac, and Linux.

Vagrant

Homestead also requires Vagrant to manage and provision virtual machines. Vagrant works on Windows, Mac, and Linux (Debian and CentOS).

Version Control

If you want to work along in the book and commit your code as you go (recommended) you need to install a version control system. I recommend git but anything you want will do.

Editor / IDE

Most readers will already have a go-to editor. I highly recommend PhpStorm—which is not free—but it pays for itself. Other common IDE options are Eclipse PDT and NetBeans.

If you don't like IDE's, I recommend Sublime Text or Atom. If you are on Mac, TextMate is another great choice. TextMate 2 is marked as “beta” but is reliable.

About Me

Salut, Hoi, Hello

I am a web developer writing highly available applications powered by PHP, JavaScript, and RESTful Web Services. I live in Scottsdale, Arizona with my wife, three boys, and one cat. When I am not wrangling code, kittens, or children, I enjoy writing, movies, golfing, basketball, and (more recently) fishing.

I'd love to hear from you on Twitter @paulredmond. Please send me your feedback (good and bad) about the book so I can make it better. If you find this book useful, please recommend it to others by sending them to https://leanpub.com/lumen-apis. Contact me if you'd like to share discount coupon codes (and maybe even a few freebies) at meetups, conferences, online, and any other way you connect with developers to share ideas.

Chapter 1: Installing Lumen

Before start diving into Lumen we need to make sure PHP is installed as well as a few other tools needed to develop a real application. You can get PHP a number of ways, so I am going to recommend the best option for all platforms: Laravel Homestead[6]. I also include a few different ways to install PHP locally if you are interested, but the book examples will use Homestead. **I highly encourage using Homestead to work through this book**.

To work through the applications in this book, we will basically need:

- PHP >= 5.5.9 as well as a few PHP extensions
- Composer
- MySQL Database

Homestead comes with a modern version of PHP, Composer[7], and a few database options, so you won't need to worry about the requirements if you are using Homestead; if you are not using Homestead you will need >= PHP 5.5.9 as outlined by the Lumen installation instructions[8].

The last thing on our list we need is a database. Lumen can be configured to use different databases including MySQL, SQLite, PostgreSQL, or SQL Server. We will use MySQL (any MySQL variant[9] will do[10]) for this book. MySQL is the default database connection in the Lumen Framework database configuration[11] so we will stick with the convention.

Homestead

Laravel Homestead is the best development environment choice because it provides a complete development environment for *all* your Laravel and Lumen projects. Homestead provides some solid benefits for your development environment as well:

- Isolated environment on a virtual machine
- Works on Windows, Mac, and Linux
- Easily configure all your projects in one place

[6]laravel.com/docs/homestead

[7]https://getcomposer.org/

[8]https://lumen.laravel.com/docs/installation#installation

[9]https://www.percona.com/software/mysql-database/percona-server

[10]https://mariadb.org/

[11]https://github.com/laravel/lumen-framework/blob/5.1/config/database.php

As mentioned in the introduction, Homestead requires Vagrant[12] and VirtualBox[13] so you will need to install both. Follow the installation instructions[14] to finish setting up homestead.

Once you complete the installation instructions you should be able to run `vagrant ssh` from within the homestead project and successfully ssh into your Homestead virtual machine. We will revisit Homestead to set up our sample application in Chapter 2, and then we will set up another application in Chapter 3 that we will work on throughout the remainder of the book.

When the install instructions instruct you to clone the Homestead git repository, I encourage you to clone it to `~/Code/Homestead` to follow along with the book, or you can adapt the examples to match whatever you pick:

Clone the Homestead Project to ~/Code/Homestead

```
$ mkdir -p ~/Code
$ git clone https://github.com/laravel/homestead.git Homestead
```

Once you finish the Homestead installation instructions you should be able to ssh into the virtual machine:

SSH Into Homestead

```
$ cd ~/Code/Homestead
$ vagrant ssh
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 3.19.0-25-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Last login: Tue Feb  2 04:48:52 2016 from 10.0.2.2
vagrant@homestead:~$
```

You can type "exit" or use "Control + D" to exit the virtual machine. The homestead repository will be at `~/Code/Homestead` and that is the path we will use in this book for our applications. I encourage you to review the Homestead.yaml file at `~/.homestead/Homestead.yaml` after you finish installing Homestead. Once you get homestead installed you can skip ahead to Chapter 2. See you in the next section!

Optional Local Instructions

The following sections are optional if you are interested in running PHP locally, but feel free to skip them. I cannot guarantee these instructions, but for the most part they should work for you.

[12] https://www.vagrantup.com/
[13] https://www.virtualbox.org/
[14] http://laravel.com/docs/5.1/homestead#installation-and-setup

Mac OSX

If you want to develop locally on OS X, I recommend using Homebrew[15] to install PHP and MySQL. The PHP installation that ships with OSX will probably suffice, but I will show you how to install PHP with Homebrew instead of dealing with different versions of PHP that ship with different versions of OS X.

To install packages with Homebrew you will need Xcode developer tools and the Xcode command line tools. XCode is a rather large download—I'll be waiting for you right here.

Once you have Xcode, follow the installation instructions[16] on Homebrew's site. Next, you need to tell `brew` about "homebrew-php" so you can install PHP 5.6:

Tap homebrew-php

```
$ brew tap homebrew/dupes
$ brew tap homebrew/versions
$ brew tap homebrew/homebrew-php
$ brew install php56 php56-xdebug
```

Once the installation finishes, verify that you have the right version of PHP in your path:

```
$ php --version
PHP 5.6.16 (cli) (built: Dec  7 2015 10:06:24)
Copyright (c) 1997-2015 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2015 Zend Technologies
```

Next, we need to install the MySQL database server with Homebrew:

```
$ brew install mysql
```

Once the MySQL installation is finished, make sure you can connect to the database server:

[15] http://brew.sh/
[16] http://brew.sh/

Connect to MySQL

```
$ mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3795
Server version: 5.6.26 Homebrew
...
mysql>
```

I highly recommend updating the root password, and adding another user besides root that you will use to connect to MySQL. Although the database is local, securing MySQL is a good habit.

You can configure Apache or Nginx locally if you want to use a webserver (Mac ships with Apache), or you can run the examples with PHP's built-in server using the following command in a Lumen project:

Running the Built-in PHP Server

```
$ php artisan serve
```

I'll leave the rest up to you, but it should be pretty easy to get PHP and a webserver going on Mac by searching Google.

Linux

I am providing simple instructions to install PHP on Unix-like systems; this section includes the most popular distributions like CentOS and Ubuntu. This is not an exhaustive set of setup instructions but it should be enough to work with Lumen.

Red Hat / CentOS

To install a modern version of PHP on Red Hat and CentOS I recommend using the Webtatic[17] yum repository. First add the repository with the Webtatic release RPM; you should use the the repository that matches your specific version:

[17] https://webtatic.com/

Add Webtatic Repository

```
# CentOS/REHL 7
$ yum -y update
$ rpm -Uvh https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
$ rpm -Uvh https://mirror.webtatic.com/yum/el7/webtatic-release.rpm

# CentOS/REHL 6
$ yum -y update
$ rpm -Uvh https://mirror.webtatic.com/yum/el6/latest.rpm
```

Next, install the following PHP packages and verify PHP was installed properly:

Install PHP Packages from Webtatic

```
$ yum install \
    php56w.x86_64 \
    php56w-mysql.x86_64 \
    php56w-mbstring.x86_64 \
    php56w-xml.x86_64 \
    php56w-pecl-xdebug.x86_64

# Verify
$ php --version
PHP 5.6.16 (cli) (built: Nov 27 2015 21:46:01)
Copyright (c) 1997-2015 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2015 Zend Technologies
```

Next, install the MySQL client and server:

Install MySQL on REHL

```
$ yum install mysql-server mysql
```

Once MySQL is installed we should set a root password:

Secure MySQL Installation

```
$ /usr/bin/mysql_secure_installation
```

Follow the prompts and you should be all set!

Debian/Ubuntu

On Debian systems I recommend using the php5-5.6 PPA[18] from Ondřej Surý[19]. Installation of the PPA varies slightly between different versions. Most of the steps will remain the same, but I am providing Ubuntu 14.04 and Ubuntu 12.04.

First, install a couple dependencies needed to add the PPA. If you are using Ubuntu 14.04:

Install Dependencies Needed and the PPA on Ubuntu 14.04

```
$ apt-get install -y language-pack-en-base
$ apt-get install -y software-properties-common --no-install-recommends
$ LC_ALL=en_US.UTF-8 add-apt-repository ppa:ondrej/php5-5.6
```

If you are using Ubuntu 12.04, run this instead:

Install Dependencies and the PPA on Ubuntu 12.04

```
$ apt-get install -y language-pack-en-base
$ apt-get install -y python-software-properties --no-install-recommends
$ LC_ALL=en_US.UTF-8 add-apt-repository ppa:ondrej/php5-5.6
```

Note that non-UTF-8 locales will not work[20] at the time of writing. Next, update and install the required packages and verify; the commands are the same for Ubuntu `14.04` and `12.04`:

Update and Install Packages

```
$ apt-get update
$ apt-get install -y \
    php5 \
    php5-mysql \
    php5-xdebug

# Verify
$ php --version
PHP 5.6.16-2+deb.sury.org~precise+1 (cli)
Copyright (c) 1997-2015 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2015 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2015, by Zend Technologies
```

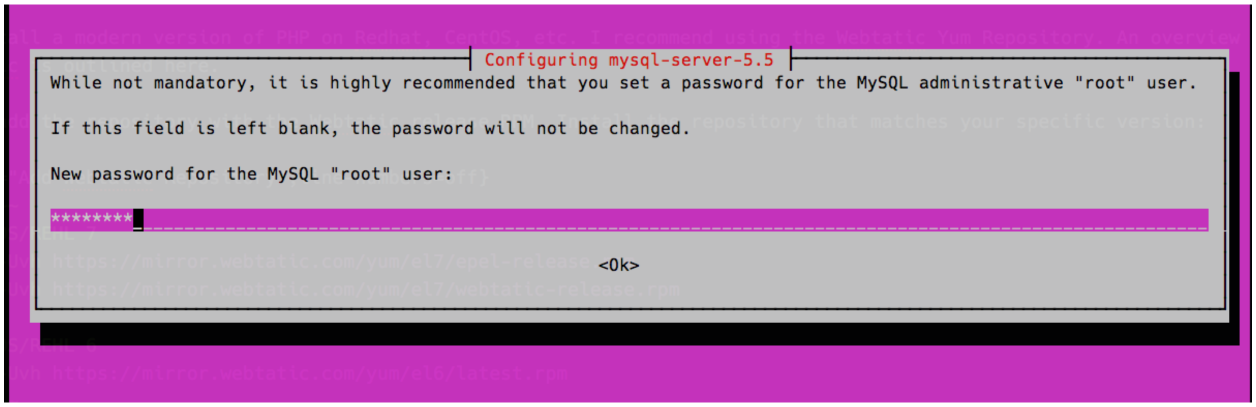
Next, install MySQL server and client packages, make the MySQL service starts on boot, and start the service manually:

[18]https://launchpad.net/~ondrej/+archive/ubuntu/php5-5.6
[19]https://launchpad.net/~ondrej
[20]https://github.com/oerdnj/deb.sury.org/issues/56

Install MySQL Packages on Ubuntu

```
$ apt-get install \
    mysql-server \
    mysql-client
$ sudo update-rc.d mysql defaults
$ sudo service mysql start
```

During the installation of the `mysql-server` package you should be prompted to update the root password, which will look similar to the following:



```
Configuring mysql-server-5.5
While not mandatory, it is highly recommended that you set a password for the MySQL administrative "root" user.

If this field is left blank, the password will not be changed.

New password for the MySQL "root" user:

********

<Ok>
```

Configuring MySQL Root Password

Verify you can connect to MySQL after you finish installing MySQL and setting a root password (you did set one, didn't you?):

Connect to MySQL

```
$ mysql -u root -p
Enter password:
...
mysql>
```

At this point, you should have everything required to get through this book using the built-in PHP server on a local Ubuntu machine.

Windows

I recommend using Homestead[21] to work through this book on Windows.

[21]http://laravel.com/docs/5.1/homestead#installation-and-setup

Conclusion

You should now have a working environment that you can use to write Lumen applications! Let's summarize what we did in this chapter:

- Installed Vagrant and VirtualBox
- Installed the Homestead virtual machine
- Covered alternative ways of installing PHP and MySQL

I want to emphasize how easy Homestead makes getting a solid, portable development environment working with little effort. Now that we have PHP installed, it's time to learn Lumen!

Chapter 2: Hello Lumen

Let's dive right into Lumen. We will go over setting up a new Lumen project and explore some of Lumen's basic features:

- Routing
- Middleware and Responses
- Views

To follow along you should have the recommended Homestead environment from Chapter 1 installed.

2.1: Setting Up a New Project

Before we can get started, we need to create a new Lumen project in Homestead. To create a new project, ssh into Homestead virtual machine and use composer to create a new Lumen project:

Creating a New Lumen Application in Homestead

```
# On your local machine
$ cd ~/Code/Homestead
$ vagrant ssh

# In the virtual machine
vagrant@homestead:~$ cd ~/Code
vagrant@homestead:~/Code$ composer create-project \
  laravel/lumen=~5.1.0 --prefer-dist hello-lumen
vagrant@homestead:~/Code$ cd hello-lumen
```

The book examples assume Homestead was cloned to the suggested path `~/Code/Homestead`. Adjust the commands if you cloned Homestead elsewhere.

In the homestead virtual machine, we change directories to `~/Code` where our application files will live. Next, we used composer's `create-project` command to create a new Lumen project. The last argument in the `create-project` command tells composer to create the project in the path `~/Code/hello-lumen`. Now that you've created a new project on the virtual machine, you should also see a shared local path at `~/Code/hello-lumen` on your own machine.

The next step is telling Homestead about the `hello-lumen` application. On your local machine, open `~/.homestead/Homestead.yaml` and find the default project under the `sites` key:

Default Sites Configuration in Homestead.yaml

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

Replace it with the following and save the file:

Default Sites Configuration in Homestead.yaml

```
sites:
    - map: hello-lumen.app
      to: /home/vagrant/Code/hello-lumen/public
```

We configure the project's hostname and the path to the `public` folder on the virtual machine. Save the file and run `vagrant provision` on your local machine to update Homestead with the new configuration changes:

Provision Vagrant Locally

```
> cd ~/Code/Homestead
> vagrant provision
```

Every time you update `Homestead.yaml` you will need to run the `vagrant provision` command.

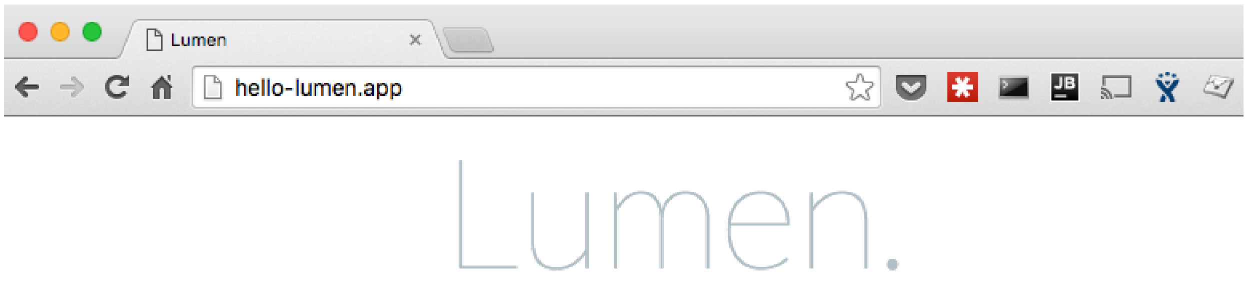
Once vagrant is finished provisioning the machine, the last step is adding an entry to the *hosts* file on your local machine. The hosts file will map the hostname `hello-lumen.app` to our virtual machine's IP address. You can find Homestead's IP address by finding the `ip` key in the ~`/.homestead/Homestead.yaml` file—you should see something like `ip: "192.168.10.10"`. Take note of the IP address so we can add it to the local `hosts` file. To update the hosts file on Mac or Linux, the file path is `/etc/hosts`; if you are on Windows the file path is `C:\Windows\System32\drivers\etc\hosts`. Add the following line to your `hosts` file:

Adding Hostname to Hosts File

```
192.168.10.10 hello-lumen.app
```

Be sure to use the IP address found in your ~/.homestead/Homestead.yaml file, not the IP shown in this book. It might be the same, but make sure.

After updating the hosts file visit http://hello-lumen.app/[22] in your browser and you should see something similar to this:



Lumen Default Route

Running the Built-In Server

If you have the correct requirements on your local machine you can run the application with the built-in server through the artisan console[a]:

Running Lumen Built-In Server

```
hello-lumen> php artisan serve
Lumen development server started on http://localhost:8000/
```

You can also configure the port with something like `php artisan serve --port 9999`.

[a]https://laravel.com/docs/5.1/artisan

You should now have a working hello-lumen project. Let's get to work!

[22]http://hello-lumen.app/

2.2: Routes

Routing[23] is the first feature we will cover. Application routes in Lumen are defined in the `app/Http/routes.php` file. In the most basic form, routing configuration includes an HTTP verb (GET, POST, etc.) which accepts a URI and a `Closure`. We will use the Closure style routes in this chapter, but we will use controllers throughout the book.

Our first routes will be two simple "Hello World" examples to introduce you to routing:

- `/hello/world` which responds with the text "Hello World"
- `/hello/{name}` which responds with a customized greeting

Before we define our own routes, if you open the file `app/Http/routes.php` the default contents looks like this (with comments removed):

The Default Lumen Route in `app/Http/routes.php`

```
<?php

$app->get('/', function () use ($app) {
    return $app->welcome();
});
```

The `$app` variable in the routes file is an instance of `\Laravel\Lumen\Application` which is defined in the `bootstrap/app.php` file. The application routes file is imported near the end of `bootstrap/app.php`:

The Bootstrap File Importing Routes

```
$app->group(['namespace' => 'App\Http\Controllers'], function ($app) {
    require __DIR__.'/../app/Http/routes.php';
});
```

The Hello World Route

Our first route is a simple `/hello/world` route that responds with the text "Hello World". Open up the `app/Http/routes.php` file and add the following route:

[23]https://lumen.laravel.com/docs/routing

The `/hello/world` Route in `app/Http/routes.php`

```
$app->get('/hello/world', function () use ($app) {
    return "Hello world!";
});
```

The `$app->get()` method accepts a URI and a `\Closure` that gets executed to create the response. The route returns a string response. If you visit http://hello-lumen.app/hello/world[24] in your browser you will see the response "Hello world!".

The `$app` instance has HTTP methods like `get`, `put`, `post`, and `delete` which are used to define routes. In our example the defined route will respond to `GET` requests. If you try to send a `POST` request you will get a `405` response:

Trying to POST to the Hello World Route

```
$ curl -I -XPOST http://hello-lumen.app/hello/world
HTTP/1.1 405 Method Not Allowed
Server: nginx/1.9.7
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
allow: GET
Cache-Control: no-cache, private
date: Tue, 29 Dec 2015 06:28:46 GMT
```

Route Parameters

The second route we are going to add has a dynamic route parameter:

Our Second Route

```
$app->get('/hello/{name}', function ($name) use ($app) {
    return "Hello {$name}";
});
```

The route URI has a required route parameter `{name}` which is then passed to the `\Closure`. We then `return` our concatenated `$name` variable, which creates the following HTTP response:

[24]http://hello-lumen.app/hello/world

Example Response from the Router

```
curl -i http://hello-lumen.app/hello/paul
HTTP/1.1 200 OK
Server: nginx/1.9.7
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache
Date: Sat, 26 Dec 2015 21:27:19 GMT

Hello paul
```

You can define multiple route parameters in one route and add constraints to them (ie. only digits). We will go over plenty of route examples as we work through this book together.

2.3: Middleware and Responses

Similar to express.js[25] and many other web frameworks, Lumen has HTTP middleware[26]. Middleware provides a way to filter incoming HTTP requests before a defined route handles the request. You can use middleware to do any number of things, like authentication, validating a signed request, and CORS support, to name a few. Middleware classes are typically created in the `app/Http/Middleware` path by convention; I suggest sticking to the convention unless you plan on writing a standalone package that includes middleware.

Lumen has two types of middleware configuration: global middleware and route middleware. What is the difference between the two types? Global middleware runs on every HTTP request and route middleware runs on specific routes (or groups of routes) configured to run the middleware. I will show you an example of each.

You will also see an example of creating a response object in middleware. We will work with response objects throughout this book, but we only touch on them lightly in this chapter.

Global Middleware

The first middleware example we will write is a simple request logger that logs every incoming request to the `storage/logs/lumen.log` application log file. Configuring the logging middleware to be a *global middleware* makes sense because we want to log all HTTP requests.

Start by creating the file `app/Http/Middleware/RequestLogMiddleware.php` with the following contents:

[25]http://expressjs.com/
[26]http://lumen.laravel.com/docs/middleware

Creating the RequestLogMiddleware

```php
<?php

namespace App\Http\Middleware;

use Log;
use Closure;
use Illuminate\Http\Request;

class RequestLogMiddleware
{
    public function handle(Request $request, Closure $next)
    {
        Log::info("Request Logged\n" .
            sprintf("~~~~\n%s~~~~", (string) $request));

        return $next($request);
    }
}
```

Middleware needs to define a `handle` method which accepts two parameters: the request object and a `Closure` instance. The request object is an instance of `Illuminate\Http\Request` and represents the current request.

Each middleware must call `return $next($request)` at some point in order to continue processing the request.

Now we need to register our new middleware in `bootstrap/app.php`:

Register a Global Middleware

```php
// $app->middleware([
//     // Illuminate\Cookie\Middleware\EncryptCookies::class,
//     // Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
//     // Illuminate\Session\Middleware\StartSession::class,
//     // Illuminate\View\Middleware\ShareErrorsFromSession::class,
//     // Laravel\Lumen\Http\Middleware\VerifyCsrfToken::class,
// ]);

$app->middleware([
    App\Http\Middleware\RequestLogMiddleware::class
]);
```

The `Application::middleware()` method accepts an array of middleware class names. I have included the commented out middleware so you can see other types of middleware that ship with Lumen.

We have one more step to get our middleware working: we need to enable facades[27] so the `Log` class will work as expected.

In `bootstrap/app.php` uncomment the following:

Enabling Facades in Our Application

```
// $app->withFacades();
$app->withFacades();
```

With facades enabled, our new middleware will add a log entry to `storage/logs/lumen.log` for every request:

Partial Log Output from RequestLogMiddleware in lumen.log

```
[2015-12-26 21:47:53] lumen.INFO: Request Logged
GET /hello/paul HTTP/1.1
...
```

Our middleware seems to be working. What happens if we forget to call `$next($request)`? To experiment, you would get the following response by removing `return $next($request)` from the middleware (be sure to put it back):

What Happens When $next($request) is not Returned?

```
curl -i http://hello-lumen.app/hello/paul
HTTP/1.1 200 OK
Server: nginx/1.9.7
Date: Sat, 26 Dec 2015 21:54:00 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
```

Middleware can also control **whether or not** the HTTP request shoud continue being processed. For example, an authentication middleware would deny access to guests trying to access secured parts of the application by sending a `403 Forbidden` response instead of proceeding with the request. Middleware should either allow the request to continue or send a response back.

Route Middleware

Our next middleware will be *route middleware* for the `/hello/{name}` route. Create a new middleware class in `app/Http/Middleware/HelloMiddleware.php` with the following contents:

[27] http://laravel.com/docs/5.1/facades

Creating the HelloMiddleware

```php
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;

class HelloMiddleware
{
    public function handle(Request $request, Closure $next)
    {
        if (preg_match('/balrog$/i', $request->getRequestUri())) {
            return response('YOU SHALL NOT PASS!', 403);
        }

        return $next($request);
    }
}
```

The `HelloMiddleware` checks the request URI against a case-insensitive regex pattern. If the URI matches the regex pattern, the middleware returns a `403 forbidden` response error with the `response()` helper function. If the user is not asking to say hello to a "balrog" the request will proceed as expected.

In order to use the `HelloMiddleware` we need to configure it in `bootstrap/app.php`:

Registering the HelloMiddleware

```php
$app->middleware([
    App\Http\Middleware\RequestLogMiddleware::class
]);

$app->routeMiddleware([
    'hello' => App\Http\Middleware\HelloMiddleware::class
]);
```

The `$app->routeMiddleware()` method takes an associative array. The key `hello` is a short-hand reference to the middleware class; the short-hand key configures routes to use the middleware:

Configuring Our Route to Use the HelloMiddleware

```
$app->get('/hello/{name}', ['middleware' => 'hello', function ($name) {
    return "Hello {$name}";
}]);
```

We have changed the second parameter in `$app->get()` to an `array`. The `middleware` key instructs our route to run the 'hello' middleware we defined in our `bootstrap/app.php` file. Note that the code example also drops `use($app)` in our `Closure` because we are not using `$app` inside the `Closure`.

Now if you try saying hello to a **balrog** the middleware will authoritatively stop the request:

Saying Hello to a Balrog

```
hello-lumen> curl -i http://hello-lumen.app/hello/balrog
HTTP/1.1 403 Forbidden
Server: nginx/1.9.7
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache
Date: Sun, 27 Dec 2015 02:00:27 GMT

YOU SHALL NOT PASS!
```

We are done with our quick tour of middleware. To learn more about middleware read the full documentation[28]. Another good resource is reading the source code of the middleware that ships with Lumen.

2.4: Views

This book is about using Lumen to write **APIs**, but as of Lumen `5.1` views are included as part of the framework. I highly advise you to use Lumen for making APIs. If you need views, use Laravel! Regardless, I will quickly show you the view functionality that ships with Lumen. Maybe this could serve you well if you are writing a single page application (SPA) with one or two server-side views.

To demonstrate views we will convert the `/hello/{name}` route to use an HTML view:

[28]http://lumen.laravel.com/docs/middleware

Convert Our Route to Use a View

```php
$app->get('/hello/{name}', ['middleware' => 'hello', function ($name) {
    return view('hello', compact('name'));
}]);
```

Lumen provides a `view()` helper function that takes `1` or `2` parameters. The first parameter is the name of the file. The filename can include dots '.' to separate folders; for example, calling `view('greetings.hello')` would look for a template in `resources/views/greetings/hello.php`. The second parameter is a key/value pair of variables that will be available in the view.

To get our route working with the `view()` helper, we need to create the view file at `resources/views/hello.php` with the following contents:

The hello.php View

```html
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello <?= ucfirst($name); ?></title>
</head>
<body>
    <p>Hello <?= ucfirst($name); ?>!</p>
</body>
</html>
```

Now when you make a request (that doesn't end in "balrog") you should see something similar to the following:

Requesting Our Route with a View

```
$ curl -i http://hello-lumen.app/hello/paul
HTTP/1.1 200 OK
Server: nginx/1.9.7
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache
Date: Sun, 27 Dec 2015 06:05:03 GMT

<!doctype html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>Hello Paul</title>
</head>
<body>
    <p>Hello Paul!</p>
</body>
</html>
```

Views are simple and straightforward, and that's all the time we are going to spend on them. All of our responses in this book will be JSON responses, but I wanted to introduce you to views anyway since you might need to use them in your own application. Just be careful to not expect the same functionality that Laravel provides. The Lumen framework is designed to be light and fast, focused on delivering APIs.

Onward

We are done with our tour of the basic parts of Lumen. In the next chapter we will create another Lumen application and prepare to write test-driven features as we work through the remainder of the book.

Chapter 3: Creating the Book Application

Unfortunately, the world doesn't have much demand for "Hello World" APIs and working on trivial applications is not going to help you for long. We are ready to start building something of more substance driven by tests. For the remainder of this book, we will create a RESTful (hopefully) book API. Our book API will represent two main RESTful resources, books and authors, and a few other resources. We are writing a RESTful API, and while I try to follow good practices, it might not be perfectly "RESTful". I digress.

You will see some similarities between writing APIs and web applications in this book, like routing, models, database associations, and the like. We will cover specific challenges and needs that differ from traditional web applications. As we work on **book** and **author** resources we will also include validation, ways to structure response data, and error handling, among other API-specific topics.

Before we can start writing our API, we need to define and set up our application to store our virtual library of books. In true web naming fashion, our virtual library will be known to the world as **bookr**.

Bookr will be developed in small, test-driven increments of code. You will get accustomed to running tests often, and I will show you a couple of my favorite workflows around testing. I think you will begin to realize how easily you can write fully-tested APIs with Lumen!

Source Code

If you get stuck or you want to see the source code, you can download it.

Build Something Amazing

If you've used Laravel before, you are probably familiar with the love and attention to detail in the little things that make the Laravel ecosystem amazing! Without further ado, we are ready to start by creating a new project on the Homestead virtual machine:

Creating a New Lumen Application in Homestead

```
# On your local machine
$ cd ~/Code/Homestead
$ vagrant ssh

# In the virtual machine
vagrant@homestead:~$ cd ~/Code
vagrant@homestead:~/Code$ composer create-project \
  laravel/lumen=~5.1.0 --prefer-dist bookr
vagrant@homestead:~/Code$ cd bookr
```

Next, let's put our application under version control on the Homestead virtual machine:

Add the application to version control

```
# vagrant@homestead:~/Code/bookr$
$ git init
$ git add .
$ git commit -m"Initial commit of Bookr"
```

Initial commit of Bookr

6dad943[29]

Lumen ships with sensible `.gitignore` defaults so don't be concerned with running `git add .` for the initial commit.

Keep Private things Private

Avoid committing sensitive information (ie. database passwords) to version control. Lumen uses the phpdotenv[30] library to load environment configuration, making it really easy to keep sensitive data out of your code.

Next, we need to configure our new application locally in ~/`.homestead/Homestead.yaml`. On your local machine add the following configuration:

[29]https://bitbucket.org/paulredmond/bookr/commits/6dad943
[30]https://github.com/vlucas/phpdotenv

Add Bookr Site Configuration in Homestead.yaml

```
sites:
    - map: hello-lumen.app
      to: /home/vagrant/Code/hello-lumen/public
    - map: bookr.app
      to: /home/vagrant/Code/bookr/public

databases:
    - homestead
    - bookr
    - bookr_testing
```

We've added a site configuration and two databases that Homestead will create when we run `vagrant provision`. The `bookr` database is our development database, and the `bookr_testing` database will be our testing database in a later chapter. Save the configuration and provision the virtual machine by running the following locally:

Provision Vagrant Locally

```
$ cd ~/Code/Homestead
$ vagrant provision
```

Once provisioning is complete, we are ready to add `bookr.app` to our local system's **hosts** file. Update the hosts file entry we added in Chapter 2 by editing `/etc/hosts` (or `C:\Windows\System32\drivers\etc\hosts` on Windows) and make it look like the following:

Updating the Hosts File

```
192.168.10.10 hello-lumen.app bookr.app
```

Be sure you use the correct IP address from your ~`/.homestead/Homestead.yaml` file!

We are done setting up our application on Homestead. If you visit http://bookr.app/[31] in your browser you should now see the same "Lumen" text you saw in chapter 2.

Now that we have a working application on Homestead, we should be able to connect to our MySQL databases. If you read connecting to databases[32] in the Homestead documentation you should be able to connect with your favorite MySQL GUI app or the console. I personally like Sequel Pro[33] on OS X. At the time of this writing, you can connect to Homestead's MySQL server with the following from your local terminal (you must have the MySQL client installed):

[31] http://bookr.app/
[32] https://laravel.com/docs/5.2/homestead#connecting-to-databases
[33] http://www.sequelpro.com/

Connect to Homestead Bookr Databases

```
> mysql \
-u homestead \
-h 127.0.0.1 \
-P 33060 \
-psecret

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| bookr              |
| bookr_testing      |
| ...                |
+--------------------+
mysql> exit
Bye
```

Environment Setup

Now that our project is set up and under version control, we need to create a `.env` file. The `.env` file is used to set up environment-specific configuration which is used by Lumen. I will show you how configuration files use environment variables later on. For now, copy the `.env.example` file that ships with Lumen for our development environment:

Copy the example .env file

```
$ cd ~/Code/Homestead
$ vagrant ssh
# ...
vagrant@homestead:~$ cd Code/bookr/
vagrant@homestead:~$ cp .env.example .env
```

Repeatable Environments

I recommend keeping `.env.example` under version control and up to date with any configuration used in your application. The `.env` file is ignored by git to keep sensitive data out of your repo. That means that another developer pulling your changes they will not have any new environment variables you've added in his or her `.env` file.

Keeping track of environment variables can be frustrating when you pull changes and things stop working. Having an accurate starting point in `.env.example` makes it easy for a new developer to get an environment going and for developers to see what has been added or changed.

We will use as many conventions as possible in our environment. Since Lumen defaults to using MySQL we will stick with that convention. Open `.env` (around `Line 8` at the time of writing) and you will see the MySQL credentials:

The .env database connection configuration

```
 8 DB_CONNECTION=mysql
 9 DB_HOST=localhost
10 DB_PORT=3306
11 DB_DATABASE=bookr
12 DB_USERNAME=homestead
13 DB_PASSWORD=secret
```

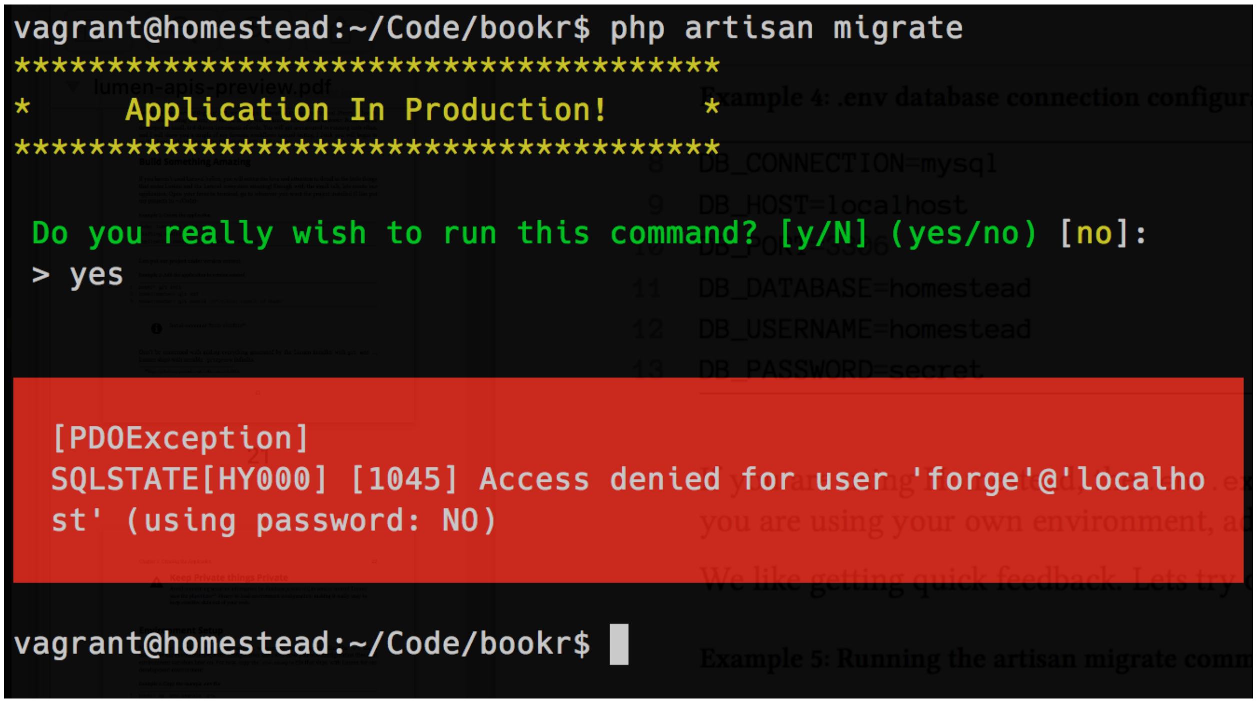
If you are using Homestead, the `.env.example` file already contains the correct MySQL credentials and the only thing you need to change is `DB_DATABASE=bookr`; if you are using your own environment, adjust accordingly.

We like getting quick feedback. Lets try out our new configuration! Run the following on the Homestead server:

Running the artisan migrate command

```
vagrant@homestead:~$ cd ~/Code/bookr
vagrant@homestead:~/Code/bookr$ php artisan migrate
```

You should see something similar to this:



```
vagrant@homestead:~/Code/bookr$ php artisan migrate
**************************************
*     Application In Production!     *
**************************************

 Do you really wish to run this command? [y/N] (yes/no) [no]:
 > yes


  [PDOException]
  SQLSTATE[HY000] [1045] Access denied for user 'forge'@'localho
  st' (using password: NO)


vagrant@homestead:~/Code/bookr$
```

Migration Failure

Why did this happen? Lumen used the default database configuration values that ship with the framework. We need to instruct Lumen to use phpdotenv[34] by opening `bootstrap/app.php` and uncommenting a line:

Uncomment Dotenv

```
Dotenv::load(__DIR__.'/../');
```

Lumen doesn't assume you want or need environment configuration, so you have to enable it. Now the database configuration will use the values from our `.env` file. Okay, lets try our command out again:

Running the artisan migrate command again

```
vagrant@homestead:~/Code/bookr$ php artisan migrate
Migration table created successfully.
Nothing to migrate.
```

Sweet! Our database and environment configurations are ready to go. You also got a taste of iterating change and getting feedback often.

[34]https://github.com/vlucas/phpdotenv

If we want to hit the ground running on our first endpoint in the next chapter, we need to do a little more setup. Lumen doesn't assume that you need or want to use Object-relational mapping (ORM), but Eloquent[35] is a really nice ORM, and I personally think its worth using in Lumen.

How do we enable Eloquent in Lumen? Back to our app bootstrap file `bootstrap/app.php` to uncomment a few lines. We will also enable Facades[36] at the same time, so uncomment the following two lines:

Enable Facades and Eloquent

```
$app->withFacades();

$app->withEloquent();
```

Lets commit our changes:

Commit application configuration

```
# vagrant@homestead:~/Code/bookr$
$ git commit -am"Enable .env, Facades, and Eloquent"
```

Enable .env, Facades, and Eloquent

04fc1e7[37]

Before moving on to our final setup task, I'd like to show you quickly how these environment settings work. Lumen has all of the default php configuration files located in the `vendor/laravel/lumen-framework/config` folder. Feel free to take a peek now.

If you create a `config/` folder in the root of the project, you can copy over config files and the application will read your copied file instead of the vendor `config` file.

The following code example is the default MySQL configuration that ships with Lumen, which may vary slightly from the time this was published. This example will give you an idea of how Lumen uses `env()` for configuration. You can see that by using `env()` we can use the `.env` file to get what we want without copying the `vendor/laravel/lumen-framework/config/database.php` configuration file to the project's `config/` folder:

[35] http://laravel.com/docs/eloquent
[36] http://laravel.com/docs/facades
[37] https://bitbucket.org/paulredmond/bookr/commits/04fc1e7

Default database.php Config (partial source)

```
'mysql' => [
    'driver'    => 'mysql',
    'host'      => env('DB_HOST', 'localhost'),
    'port'      => env('DB_PORT', 3306),
    'database'  => env('DB_DATABASE', 'forge'),
    'username'  => env('DB_USERNAME', 'forge'),
    'password'  => env('DB_PASSWORD', ''),
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => env('DB_PREFIX', ''),
    'timezone'  => env('DB_TIMEZONE', '+00:00'),
    'strict'    => false,
],
```

The `env()` function will get the value for the first argument, and if the configuration doesn't exist the second argument is the default. When we ran our failed migration command earlier the application configuration was using the defaults.

Checking Unit Tests

Lumen uses PHPUnit for tests. When I create new projects I always ensure that PHPUnit is running properly. If you don't test early and often, you will likely fail to commit to testing in a project. Try running the PHPUnit suite that ships with Lumen:

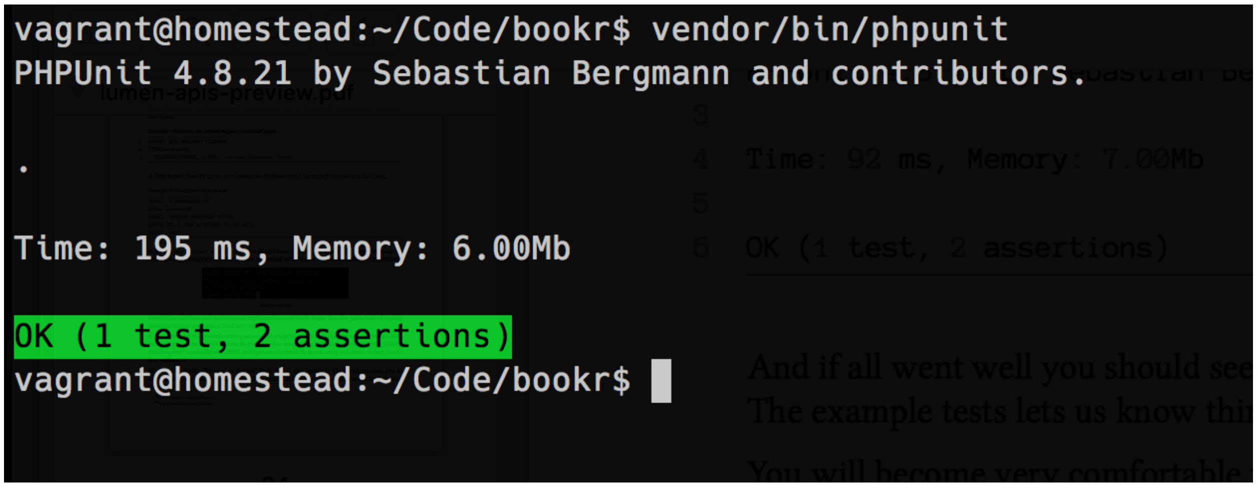
Running phpunit tests

```
vagrant@homestead:~/Code/bookr$ vendor/bin/phpunit

OK (1 test, 2 assertions)
```

PHPUnit is a composer dependency, and we execute tests by referencing `vendor/bin/phpunit`. If all went well you should see green! Lumen ships with an example test class which is passing. The example tests lets us know things are working as expected.

You will become very comfortable writing tests as you work through this book, but for now we just need to know we have everything working so we can focus on writing our application.



```
vagrant@homestead:~/Code/bookr$ vendor/bin/phpunit
PHPUnit 4.8.21 by Sebastian Bergmann and contributors.

.

Time: 195 ms, Memory: 6.00Mb

OK (1 test, 2 assertions)
vagrant@homestead:~/Code/bookr$ 
```

PHPUnit Success

PHPUnit Alias on Homestead

On Homestead you can simply run `phpunit` without referencing the `vendor/bin` path.

Homestead creates an alias for you defined on your local machine in the `~/.homestead/aliases` file. You can also add your own aliases to that file.

I also have `./vendor/bin` to my path when I am not using Homestead:

```
export PATH=./vendor/bin:$PATH
```

You can also install PHPUnit on your system. Refer to the official installation documentation[38].

Setup Complete

With minimal setup we are in good shape to start writing the first API endpoint: books. Setup was simple, but we covered many important steps that I like to do at the beginning of an application. Getting a working database and unit tests will go a long way in helping us focus on writing the API. We are establishing conventions and good practices early.

[38] https://phpunit.de/manual/current/en/installation.html

Chapter 4: Starting the Books API

In this chapter we are going to focus on writing our first API resource: `/books`. We will take this in small chunks and test it as we go. At the end of the next few chapters we will have a fully-tested `/books` resource doing basic CRUD operations. The `/books` resource will look like this:

Basic REST /books resource

GET	/books	Get all the books
POST	/books	Create a new book
GET	/books/{id}	Get a book
PUT	/books/{id}	Update a book
DELETE	/books/{id}	Delete a book

Creating the First Endpoint

The first order of business is creating a `BooksControllerTest` class and writing our first failing test. We will then write the minimum amount of code required to get the test passing. When we get back to green we are free to refactor or add another feature.

I prefer my tests namespace to be organized under the same namespace as the application namespace. In our case the controller namespace will be `App\Http\Controllers` and the tests for controllers will go under `Tests\App\Http\Controllers`. So lets write some code!

Creating our test and controller

```
# vagrant@homestead:~/Code/bookr$
$ mkdir -p tests/app/Http/Controllers
$ git mv tests/ExampleTest.php tests/app/Http/Controllers/BooksControllerTest.php
$ touch app/Http/Controllers/BooksController.php
```

We rename the `ExampleTest.php` file as the `BooksControllerTest.php` with git since we don't want a fake example test. Feel free to create files however you want, but I will do it from the command line throughout the book.

Now to write and execute our first failing test for the `GET /books` route. Note the "use TestCase" import because we are namespacing our tests. You could also reference it with `extends \TestCase` and skip the `use`.

The `BooksControllerTest.php` File

```php
<?php

namespace Tests\App\Http\Controllers;

use TestCase;

class BooksControllerTest extends TestCase
{
    /** @test **/
    public function index_status_code_should_be_200()
    {
        $this->visit('/books')->seeStatusCode(200);
    }
}
```

Our test makes a request to the `/books` route and then expects to see a `200` status code. The `visit` and `seeStatusCode` methods are provided by the `Laravel\Lumen\Testing\CrawlerTrait` trait. You will become more familiar with the methods the `CrawlerTrait` provides as we work through the book.

Let's run the test we just created and see what happens:

Failing Test

```
vagrant@homestead:~/Code/bookr$ phpunit
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

I have omitted a stack trace, but basically our failing test received a `404` status code. Now its time to make our test pass, but we only write the least amount of code to get our test to passing and nothing more. We created the `BooksController` file earlier in the chapter, now add the following:

The `BooksController.php` Class

```php
<?php

namespace App\Http\Controllers;

/**
 * Class BooksController
 * @package App\Http\Controllers
 */
class BooksController
{
    /**
     * GET /books
     * @return array
     */
    public function index()
    {
        return [];
    }
}
```

We define an `index` method and return an empty array. We still haven't defined a route for our `GET /books` endpoint which we'll add below the default route in `app/Http/routes.php`:

Add the BooksController@index Route

```php
$app->get('/', function () use ($app) {
    return $app->welcome();
});

$app->get('/books', 'BooksController@index');
```

This is our first time using a controller for the second argument of a route definition. Lumen assumes the namespace of a controller to be `App\Http\Controllers`, and the second argument is in this format: `<controller_name>@<method>`. The `BooksController@index` string passed to our route references the public index method of the `BooksController`. With the controller and route in place our test should pass now:

Test is passing

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (1 test, 2 assertions)
```

Now that our tests are back to green we are ready to add more features. We know our endpoint will return a collection of books, so lets write a test for that!

Testing the JSON response

```
/** @test **/
public function index_should_return_a_collection_of_records()
{
    $this
        ->get('/books')
        ->seeJson([
            'title' => 'War of the Worlds'
        ])
        ->seeJson([
            'title' => 'A Wrinkle in Time'
        ]);
}
```

The test introduces the `seeJson()` method, which converts the passed array into JSON and assures that the JSON occurs somewhere within the response. I encourage you to read the official Lumen documentation on testing[39]. The Testing JSON APIs[40] section has information on testing JSON responses.

As you would expect, if you run the tests you will see failures. A failed test means we are ready to write implementation code. To reiterate, we will write the smallest amount of code to get tests passing again:

[39]http://lumen.laravel.com/docs/testing
[40]http://lumen.laravel.com/docs/testing#testing-json-apis

Return a collection of books

```php
/**
 * GET /books
 * @return array
 */
public function index()
{
    return [
        ['title' => 'War of the Worlds'],
        ['title' => 'A Wrinkle in Time']
    ];
}
```

Tests are back to green now!

Passing test for a collection of books

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (2 tests, 6 assertions)
```

Git commit: Add /books Index Route

1407b02[41]

Now that we have passing tests that means we are free to refactor our code and continue to ensure our tests still pass. It seems like a good time to introduce some real data from the database.

Setting up Models and Seed Data

Our book API is not very useful right now without dynamic data, but we have a good testing foundation to make sure we can iterate and verify our tests still pass. We can lean on our tests while we convert our controller to use a database. Lumen has some great features that make working with databases very pleasant!

Our first order of business is to define a `Book` model. What should that look like? What columns should we include? We could flesh out the entire data structure up front, but database migrations make it really easy to iterate on the database schema. For now our schema will be very simple.

[41]https://bitbucket.org/paulredmond/bookr/commits/1407b02

I encourage you to spend some time thinking about your database structure up front. Aim for mix of defining good structure up front without it paralyzing you from getting started on writing code.

Migrations allow you to iterate on the schema, but making good data decisions up front can save you some headache too.

We will start by creating a database migration with the artisan console[42]:

The books table migration

```
# vagrant@homestead:~/Code/bookr$
$ php artisan make:migration create_books_table --create=books
```

The `make:migration` command knows we intend to create a new database table from the `--create` flag. Your filename date will differ from my example, but will end with the same `create_books_table.php` file suffix.

migrations

Database migrations[43] are covered in great detail in the Laravel documentation. The Laravel migration documentation applies to Lumen.

In the generated migration you will see two methods: `up()` and `down()`. The `up()` method will be used to apply the migration and the `down()` method will roll back the migration. The `artisan` command takes care of `down` for us automatically because of the `--create` flag, so lets finish writing the `up` method:

The books table migration

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

[42] https://laravel.com/docs/artisan
[43] http://laravel.com/docs/migrations

```
        $table->string('author');
        $table->timestamps();
    });
}
```

The migration is very readable and simple. Take note of `$table->timestamps()`, which will create two datetime columns: `created_at` and `updated_at`. Eloquent will also populate the timestamp columns for us automatically when we create and update records.

Now that we have the migration ready, lets run it:

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate
```

Seems to have worked, lets check the database to be sure:

```
mysql> use bookr;
mysql> show columns from books;
```

The books table structure

Field	Type	Key	Default
id	int(10) unsigned	PRI	NULL
title	varchar(255)		NULL
description	text		NULL
author	varchar(255)		NULL
created_at	timestamp		0000-00-00 00:00:00
updated_at	timestamp		0000-00-00 00:00:00

Success! Now that we have a working database migration we are going to need some data. How should we get data into our development database? We could version a sequel script that we execute when setting up an environment; we could connect to the database and add some rows by hand. At this point either would be fine, but it will get clunky fast. Real fast. Fortunately (and not surprisingly) Lumen provides a better better way with **database seeding**.

The database seeding files are located in the `database/seeds` folder—specifically the `DatabaseSeeder.php` file. This is what the stock `DatabaseSeeder` class looks like:

The default seeder class

```
 1  <?php
 2
 3  use Illuminate\Database\Seeder;
 4  use Illuminate\Database\Eloquent\Model;
 5
 6  class DatabaseSeeder extends Seeder
 7  {
 8      /**
 9       * Run the database seeds.
10       *
11       * @return void
12       */
13      public function run()
14      {
15          Model::unguard();
16
17          // $this->call('UserTableSeeder');
18
19          Model::reguard();
20      }
21  }
```

Line #17 is an example of how the `DatabaseSeeder` can call other seeder classes to keep things tidy. Note the `Model::unguard()` and `Model::reguard()` calls; the former enables mass assignment[44] of models for the purposes of seeding data and the latter re-enables protection from mass assignment of models after the seeders run.

Based on the `DatabaseSeeder` example, it makes sense to make a seeder class for the `books` table. We will call the file `BooksTableSeeder.php` and call it from the `DatabaseSeeder` class. Once we create our seeder we can use it to populate our development database with the `artisan db:seed` command.

[44]https://laravel.com/docs/eloquent#mass-assignment

The BooksTableSeeder

```php
<?php

use Carbon\Carbon;
use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class BooksTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('books')->insert([
            'title' => 'War of the Worlds',
            'description' => 'A science fiction masterpiece about Martians invad\
ing London',
            'author' => 'H. G. Wells',
            'created_at' => Carbon::now(),
            'updated_at' => Carbon::now(),
        ]);

        DB::table('books')->insert([
            'title' => 'A Wrinkle in Time',
            'description' => 'A young girl goes on a mission to save her father \
who has gone missing after working on a mysterious project called a tesseract.',
            'author' => 'Madeleine L\'Engle',
            'created_at' => Carbon::now(),
            'updated_at' => Carbon::now()
        ]);
    }
}
```

The `DB::table()` method returns an instance of the `\Illuminate\Database\Query\Builder` class, which has an `insert` method to insert a record to the database. The `Builder::insert()` method accepts an associative array of data. The code also introduces the Carbon[45] library, a standalone library for working with PHP's `DateTime`.

[45] http://carbon.nesbot.com/

To use the book seeder we need to call the `BooksTableSeeder` within the `database/seeds/DatabaseSeeder.php` class:

Call the BooksTableSeeder in the DatabaseSeeder class

```
public function run()
{
    Model::unguard();

    // $this->call('UserTableSeeder');
    $this->call(BooksTableSeeder::class);

    Model::reguard();
}
```

We are now ready to seed the database with artisan. Since we created a new seeder class we need to dump the composer autoloader. The database classes are autoloaded through composer's class map[46] setting, so each new seeder requires running the `dump-autoload` command:

```
# vagrant@homestead:~/Code/bookr$
$ composer dump-autoload
Generating autoload files

$ php artisan migrate:refresh
Rolled back: 2015_10_17_075310_create_books_table
Migrated: 2015_10_17_075310_create_books_table

$ php artisan db:seed
Seeded: BooksTableSeeder
```

I introduced a new artisan command `migrate:refresh` which will reset and re-run all migrations. The `db:seed` command populates the `bookr` database with the seed data we just defined.

Artisan

To get an overview of what each artisan command does, run `php artisan` to get a list of commands and a short description. You can even write your own commands.

Database migrations, database seeding, and Eloquent, are the great features that set apart Lumen from other PHP micro-frameworks in my opinion. Lumen is lightweight, but features that help developer productivity are not out of reach or difficult to enable.

[46] https://getcomposer.org/doc/04-schema.md#classmap

Eloquent Books

Now that the `books` table has data, lets define a model representing the books table that we can use to query the database. Lumen (like Laravel) has access to Eloquent ORM, which is a fantastic ActiveRecord implementation for querying data and inserting/updating data.

Lumen does not ship with an `artisan` command for creating models, but creating one is not hard:

```
# vagrant@homestead:~/Code/bookr$
$ touch app/Book.php
```

The model is really simple at the moment:

The Book Eloquent Model

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{

}
```

With the seeded data and `Book` model in hand, we are ready to wrap up our refactored `BooksController@index` route:

Putting the Book Model to Work

```php
<?php

namespace App\Http\Controllers;

use App\Book;

/**
 * Class BooksController
 * @package App\Http\Controllers
 */
class BooksController
{
    /**
     * GET /books
     * @return array
     */
    public function index()
    {
        return Book::all();
    }
}
```

To Facade or not to Facade?

You might have noticed `return Book::all();` in the `BooksController@index` method. This is the facade pattern, which provides a static interface to classes available in the service container[47].

Another approach is injecting the service in the Controller's constructor method (or even the controller action method) by type hinting a parameter. See the resolving[48] section of the documentation for this technique.

Time to run our test suite to see if our refactor broke anything:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (2 tests, 6 assertions)
```

Our refactor was simple and passed the first time. Sometimes your refactors will lead to broken tests which we will see later on in this book. The goal of refactoring is that you start refactoring while your tests are green, and then when you are done refactoring you should still be at green.

If you make a request to http://bookr.app/books[49] you should something resembling the following response:

[47]http://lumen.laravel.com/docs/container
[48]http://lumen.laravel.com/docs/container#resolving
[49]http://bookr.app/books

Example Response from `/books`

```
[
    {
        "id":1,
        "title":"War of the Worlds",
        "description":"A science fiction masterpiece about Martians invading Lon\
don",
        "author":"H. G. Wells",
        "created_at":"2015-12-30 03:11:40",
        "updated_at":"2015-12-30 03:11:40"
    },
    {
        "id":2,
        "title":"A Wrinkle in Time",
        "description":"A young girl goes on a mission to save her father who has\
 gone missing after working on a mysterious project called a tesseract.",
        "author":"Madeleine L'Engle",
        "created_at":"2015-12-30 03:11:40",
        "updated_at":"2015-12-30 03:11:40"
    }
]
```

Success

Boom! We've successfully refactored the BooksController to use a database and an Eloquent model. We will see plenty more examples of models throughout the book. Now we can move on to the rest of the `/books` endpoints. See you in the next chapter.

Git commit: Create books table and BooksController

8544f02[50]

[50]https://bitbucket.org/paulredmond/bookr/commits/8544f02

Chapter 5: Creating, Reading, Updating, and Deleting Books

Now that we have a **Books** model, we are well-equipped to handle the management of books. We will continue to follow our test-driven development workflow as we complete the remaining CRUD operations of the `/books` API.

Here is a quick refresher of our `/books` RESTful endpoints:

Basic REST /books resource

GET	/books	Get all the books
POST	/books	Create a new book
GET	/books/{id}	Get a book
PUT	/books/{id}	Update a book
DELETE	/books/{id}	Delete a book

5.1: Requesting an Individual Book

We are going to start off with the `GET /books/{id}` route. Before we start coding lets quickly define our high-level acceptance criteria for this route. We can do this within the `tests/app/Http/Controllers/BooksControllerTest.php` file by defining the skeleton test methods:

The GET books/:id acceptance criteria

```
/** @test **/
public function show_should_return_a_valid_book()
{
    $this->markTestIncomplete('Pending test');
}

/** @test **/
public function show_should_fail_when_the_book_id_does_not_exist()
{
    $this->markTestIncomplete('Pending test');
}

/** @test **/
```

```php
public function show_route_should_not_match_an_invalid_route()
{
    $this->markTestIncomplete('Pending test');
}
```

We mark these tests are incomplete while we work through each criteria. Running our test suite will provide the following output now:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK, but incomplete, skipped, or risky tests!
Tests: 5, Assertions: 4, Incomplete: 3.
```

I want to point out that our tests rely on seed data. This is not ideal, and we will address it in Section 6.1. In the meantime, you need to make sure that the database has seed data before running tests that rely on specific data in the database. You can always ensure that the database is fresh by running:

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh
$ php artisan db:seed
```

Let's get to work on the first acceptance criteria `show_should_return_a_valid_book`. To meet this acceptance criteria we need to:

- Find the book in the database
- Respond with the Book's data
- Make sure it responds with a correct status code
- Assert the JSON data returned is accurate

Test for a valid book

```php
/** @test **/
public function show_should_return_a_valid_book()
{
    $this
        ->get('/books/1')
        ->seeStatusCode(200)
        ->seeJson([
            'id' => 1,
            'title' => 'War of the Worlds',
            'description' => 'A science fiction masterpiece about Martians invad\
ing London',
            'author' => 'H. G. Wells'
        ]);

    $data = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('created_at', $data);
    $this->assertArrayHasKey('updated_at', $data);
}
```

You've already seen `seeStatusCode` and `seeJson`. The only thing new here is the last three lines that get the response body (JSON) and decode it. The test simply checks the existence of `created_at` and `updated_at`, but does not test the values. Our seed data cannot guarantee consistent dates so it's impossible to test the values. In a later chapter I will show you how to test date values once we start using proper test data.

Next we unit test, which will definitely fail. Can you think of why the test will fail?

Run the Failing Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit
There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::show_should_return_a_valid_bo\
ok
Failed asserting that 404 matches expected 200.

bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait.php:412
bookr/tests/app/Http/Controllers/BooksControllerTest.php:33
```

If you inspect the test failure closely you will see that on **line 33** our test asserts a **200** HTTP status code, but a **404** response is given because no route exists yet. We will need to create a new route and controller method to get our tests back to green. Make the following changes in `app/Http/routes.php` and `app/Http/Controllers/BooksController.php`:

Adding show route to routes.php

```
$app->get('/books', 'BooksController@index');
$app->get('/books/{id}', 'BooksController@show');
```

Adding show method to BooksController.php

```php
/**
 * GET /books/{id}
 * @param integer $id
 * @return mixed
 */
public function show($id)
{
    return Book::findOrFail($id);
}
```

The code for `BooksController@show` introduces the `findOrFail` method; the `findOrFail` method either returns a record or thows an exception of type `Illuminate\Database\Eloquent\ModelNotFoundException`.

The new route and the `BooksController::show()` method should be enough to get our tests passing again. We will use PHPUnit's `--filter` flag to only run our latest test:

Run PHPUnit Test for Returning a Valid Book

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=show_should_return_a_valid_book

OK (1 test, 8 assertions)
```

We are back to green and we can move on to our next acceptance criteria:

Test Failure When a Book Does Not Exist

```php
/** @test **/
public function show_should_fail_when_the_book_id_does_not_exist()
{
    $this
        ->get('/books/99999')
        ->seeStatusCode(404)
        ->seeJson([
            'error' => [
                'message' => 'Book not found'
            ]
        ]);
}
```

Run PHPUnit after Writing the Test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=show_should_fail_when_the_book_id_does_not_exist

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::show_should_fail_when_the_boo\
k_id_does_not_exist
Failed asserting that 500 matches expected 404.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:51

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Our latest test brings us back to failure. The `findOrFail` method will throw an exception if the id does not match a record in the database, resulting in a **500** error response. What we really wanted based on our test criteria is a **404** response with a friendly error message. Time to refactor our controller and get tests back to green!

Respond to ‘ModelNotFoundException BooksController::show()

```php
<?php

namespace App\Http\Controllers;

use App\Book;
use Illuminate\Database\Eloquent\ModelNotFoundException;

/**
 * Class BooksController
 * @package App\Http\Controllers
 */
class BooksController
{
    /**
     * GET /books
     * @return array
     */
    public function index()
19     {
20         return Book::all();
21     }
22 
23     /**
24      * GET /books/{id}
25      * @param integer $id
26      * @return mixed
27      */
28     public function show($id)
29     {
30         try {
31             return Book::findOrFail($id);
32         } catch (ModelNotFoundException $e) {
33             return response()->json([
34                 'error' => [
35                     'message' => 'Book not found'
36                 ]
37             ], 404);
38         }
39     }
40 }
```

The entire controller is included to avoid confusion. Let's break down the changes:

- We wrap the `Book::findOrFail()` method in a `try/catch` block so we can catch the exception and respond with a **404**.
- If the id is found, a valid record is returned.
- The `ModelNotFoundException` is imported on line #6.
- We introduce the `response()` helper to create a response object
- We then call `json()` on the response object to send a JSON response.
- The `json()` method accepts our array of data, and a HTTP status code.

Running our test suite will get us back into the green. Getting back to green is a good feeling.

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=show_should_fail_when_the_book_id_does_not_exist

OK (1 test, 3 assertions)
```

We have one final acceptance criteria to tackle and then we can move on to the next endpoint. The final test is ensuring that our `BooksController::show` route only matches integer ids. As of right now, our `show` route will hapily match any parameter. Lets write a test for the route matching:

Test for route matching.

```
/** @test **/
public function show_route_should_not_match_an_invalid_route()
{
    $this->get('/books/this-is-invalid');

    $this->assertNotRegExp(
        '/Book not found/',
        $this->response->getContent(),
        'BooksController@show route matching when it should not.'
    );
}
```

Our test ensures that the response does not contain `Book not found` because that would mean the `BooksController@show` route was executed and the `404` response for the `ModelNotFoundException` was sent. We added a helpful message "BooksController@show route matching when it should not" to help explain that we do not want to match the `BooksController@show` route.

Let's run the test and see if it fails:

Run Test for Invalid Show Route

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=show_route_should_not_match_an_invalid_route

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::show_route_should_not_match_a\
n_invalid_route
BooksController@show route matching when it should not.
Failed asserting that '{"error":{"message":"Book not found"}}' does not match PC\
RE pattern "/Book not found/".

/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:67

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Since we have not changed the `BooksController@show` code yet, we get an expected failure. We need to change the route parameter to be constrained with a regular expression and then the test will pass:

Constrain the Show Route with a Regular Expression (app/Http/routes.php).

```php
$app->get('/books/{id:[\d]+}', 'BooksController@show');
```

Now only requests where the `id` is an integer will match. Let's see if the test is passing:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=show_route_should_not_match_an_invalid_route

OK (1 test, 1 assertion)
```

Our acceptance criteria has been met for the `GET /book/{id}` route! We are ready to move on to creating new books. Commit your changes:

> Git commit Add /books/{id} Route
>
> 6045928[51]

Here is what our `BooksControllerTest` class looks like so far:

BooksControllerTest

```php
<?php

namespace Tests\App\Http\Controllers;

use TestCase;

class BooksControllerTest extends TestCase
{
    /** @test **/
    public function index_status_code_should_be_200()
    {
        $this->visit('/books')->seeStatusCode(200);
    }

    /** @test **/
    public function index_should_return_a_collection_of_records()
    {
        $this
            ->get('/books')
```

[51]https://bitbucket.org/paulredmond/bookr/commits/6045928

```
            ->seeJson([
                'title' => 'War of the Worlds'
            ])
            ->seeJson([
                'title' => 'A Wrinkle in Time'
            ]);
    }

    /** @test **/
    public function show_should_return_a_valid_book()
    {
        $this
            ->get('/books/1')
            ->seeStatusCode(200)
            ->seeJson([
                'id' => 1,
                'title' => 'War of the Worlds',
                'description' => 'A science fiction masterpiece about Martians i\
nvading London',
                'author' => 'H. G. Wells'
            ]);

        $data = json_decode($this->response->getContent(), true);
        $this->assertArrayHasKey('created_at', $data);
        $this->assertArrayHasKey('updated_at', $data);
    }

    /** @test **/
    public function show_should_fail_when_the_book_id_does_not_exist()
    {
        $this
            ->get('/books/99999')
            ->seeStatusCode(404)
            ->seeJson([
                'error' => [
                    'message' => 'Book not found'
                ]
            ]);
    }

    /** @test **/
    public function show_route_should_not_match_an_invalid_route()
    {
        $this->get('/books/this-is-invalid');

        $this->assertNotRegExp(
            '/Book not found/',
            $this->response->getContent(),
            'BooksController@show route matching when it should not.'
        );
    }


}
```

We have been running specific tests. We need to make sure the whole suite is passing before moving to the next feature:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (5 tests, 18 assertions)
```

Running the **whole suite** is important before moving on to make sure the entire test harness is sound before writing new features. You will build a dependable—albiet imperfect—set of tests you can depend on when refactoring and making major changes.

How is the missing route handled?

Now that our `BooksController::show()` route only matches integer ids, how is an invalid route like `/books/this-is-invalid` handled? If Lumen can't find a route, the application will throw a `NotFoundHttpException`. If you investigate the response headers you can see that Lumen responds with a `text/html` content type by default:

```
curl -I -H"Content-Type: application/json" \
-H"Accept: application/json" \
http://localhost:8000/books/this-is-invalid

HTTP/1.0 404 Not Found
Host: localhost:8000
Connection: close
Cache-Control: no-cache, private
date: Sun, 18 Oct 2015 07:25:30 GMT
Content-type: text/html; charset=UTF-8
```

> The HTML response is not something an API consumer expects and we intend on making error responses JSON. For now, be aware of how things are working. We will revisit handling missing routes exceptions with JSON in the next chapter.

5.2: Creating a New Book

The next feature is creating a new book with the `POST /books` route. We will dive more into Eloquent and see how to handle `POST` data in the controller. Here is what our acceptance criteria will be for creating a new book in the `tests/app/Http/Controllers/BooksControllerTest.php` file:

The POST /books acceptance criteria

```
/** @test **/
public function store_should_save_new_book_in_the_database()
{
    $this->markTestIncomplete('pending');
}

/** @test */
public function store_should_respond_with_a_201_and_location_header_when_success\
ful()
{
    $this->markTestIncomplete('pending');
}
```

Let's start with the `store_should_save_new_book_in_the_database` test. When things go as expected, this test will make sure that:

- The response contains a `"created": true` JSON fragment
- The book exists in the database

Test Creating a New Book

```
/** @test **/
public function store_should_save_new_book_in_the_database()
{
    $this->post('/books', [
        'title' => 'The Invisible Man',
        'description' => 'An invisible man is trapped in the terror of his own c\
reation',
        'author' => 'H. G. Wells'
    ]);

    $this
        ->seeJson(['created' => true])
        ->seeInDatabase('books', ['title' => 'The Invisible Man']);
}
```

Our test inherits a `$this->post()` method which accepts a URI and an array of post data. The test contains the `seeInDatabase()` method, which accepts a table name, and an associative array in the format of `column => value` and ensures the record is in the database. You can pass multiple columns to further constrain `$this->seeInDatabase()` assertions.

Running the test suite will result in a failure because we haven't defined the route or the controller method yet:

Run the Full Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::store_should_save_new_book_in\
_the_database
Failed asserting that JSON returned [http://localhost/books].
Failed asserting that '<!DOCTYPE html>\n<html>\n    <h.../html>' is valid JSON (\
Syntax error, malformed JSON).

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:352
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:81

FAILURES!
Tests: 7, Assertions: 19, Failures: 1, Incomplete: 1.
```

We will start coding by adding a route to `app/Http/routes.php`:

Add the POST Route

```php
$app->get('/books', 'BooksController@index');
$app->get('/books/{id}', 'BooksController@show');
$app->post('/books', 'BooksController@store');
```

Our first version of the `BooksController@store` method:

First version of the `BooksController::store()` method.

```php
/**
 * POST /books
 * @param Request $request
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function store(Request $request)
{
    $book = Book::create($request->all());

    return response()->json(['created' => true], 201);
}
```

The store method is our first example using the service container[52] to do method injection in a controller. The store method accepts an `Illuminate\Http\Request` instance which represents the current HTTP request.

Inside the method, the first line tries to create a new book in the database by passing post data from the request (`$request->all()`) to the Book::create() method. If the `Book::create()` method succeeds, it will return an instance of the new book. After creating the book, the method returns a `JsonResponse` object by calling `response()->json()` which accepts an array of data and a status code. The `JsonResponse` object will convert the array into JSON before being sent to the browser.

Your senses might be warning you that we are allowing mass-assignment in the controller. While it is good to always keep mass-assignment in mind, Eloquent does guard against mass assignment[53] as you will see shortly.

Before we run our test we need to also import `Illuminate\Http\Request` at the top of the `BooksController`:

[52]https://lumen.laravel.com/docs/container
[53]https://laravel.com/docs/eloquent#mass-assignment

Import the Illuminate Request Class in the BooksController

```php
<?php

namespace App\Http\Controllers;

use App\Book;
use Illuminate\Http\Request;
// ...
```

You might think we've covered everything and tests should pass, however, our test suite still fails:

The Failing Test Output

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=store_should_save_new_book_in_the_database

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::store_should_save_new_book_in\
_the_database
Failed asserting that JSON returned [http://localhost/books].
Failed asserting that '<!DOCTYPE html>\n<html>\n    <h.../html>' is valid JSON (\
Syntax error, malformed JSON).

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:352
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:81

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

The error message is identical to the last time we ran this test!? We need to dig a little deeper to figure this out. A couple options could include looking at the `storages/logs/lumen.log` file or adding some temporary debugging code to the request. We will choose the second option and add some debugging code to the `BooksController`:

Temporary debugging code.

```php
/**
 * POST /books
 * @param Request $request
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function store(Request $request)
{
    try {
        $book = Book::create($request->all());
    } catch (\Exception $e) {
        dd(get_class($e));
    }

    return response()->json(['created' => true], 201);
}
```

We are attempting to debug the `store` method by catching any exception that happens when calling `Book::create()`. The code introduces the `dd()` function which dumps the passed variable(s) and exits the script.

With the debug code in place, run the test again:

Run Tests with Debug Code in Place

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=store_should_save_new_book_in_the_database

"Illuminate\Database\Eloquent\MassAssignmentException"
```

The exception being caught is a mass-assignment exception. I mentioned earlier that Eloquent guards against mass-assignment out of the box, so we need to configure which fields are mass-assignable using the `protected $fillable = []` array in the `Book` model:

Defining mass assignable fields (app/Http/Book.php).

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * The attributes that are mass assignable
     *
     * @var array
     */
    protected $fillable = ['title', 'description', 'author'];
}
```

You can also provide a `protected $guarded` property with values that should not be mass-assignable. We are taking the route that all columns should be protected, minus the exceptions we add to the `$fillable` array. See the mass assignment[54] section of the Eloquent documentation for more information.

Before you remove the debugging code, try running the test again:

Run the Test after Defining Fillable Columns

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=store_should_save_new_book_in_the_database

OK (1 test, 3 assertions)
```

After defining the fillable fields our test worked as expected! Now we need to revert the debugging code we added:

[54] https://laravel.com/docs/eloquent#mass-assignment

Revert Debugging Code

```
/**
 * POST /books
 * @param Request $request
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function store(Request $request)
{
    $book = Book::create($request->all());

    return response()->json(['created' => true], 201);
}
```

The next test criteria makes sure that successfully creating a new book will respond with a `201` status code and a `Location` header matching the URI of the created resource:

Writing test for 201 status code and Location header.

```
/** @test */
public function store_should_respond_with_a_201_and_location_header_when_success\
ful()
{
    $this->post('/books', [
        'title' => 'The Invisible Man',
        'description' => 'An invisible man is trapped in the terror of his own c\
reation',
        'author' => 'H. G. Wells'
    ]);

    $this
        ->seeStatusCode(201)
        ->seeHeaderWithRegExp('Location', '#/books/[\d]+$#');

}
```

The test will look familiar except for the `seeHeaderWithRegExp` method, because we haven't defined it yet :). To provide some convenience around checking header values, we will define this custom assertion in the base test class `tests/TestCase.php` that our tests inherit:

New test assertions (tests/TestCase.php)

```php
/**
 * See if the response has a header.
 *
 * @param $header
 * @return $this
 */
public function seeHasHeader($header)
{
    $this->assertTrue(
        $this->response->headers->has($header),
        "Response should have the header '{$header}' but does not."
    );

    return $this;
}

/**
 * Asserts that the response header matches a given regular expression
 *
 * @param $header
 * @param $regexp
 * @return $this
 */
public function seeHeaderWithRegExp($header, $regexp)
{
    $this
        ->seeHasHeader($header)
        ->assertRegExp(
            $regexp,
            $this->response->headers->get($header)
        );

    return $this;
}
```

We added two public methods that help assert headers. The `seeHasHeader()` method asserts the mere existence of a header. The `seeHeaderWithRegExp` method chains a call to `seeHasHeader` and then uses PHPUnit's `assertRegExp` to see if the response header matches the passed regular expression. These methods use `Illuminate\Http\Response` which wraps the `Symfony\Component\HttpFoundation\Response`. I encourage you to become familiar with the request and response classes, because you will use them frequently to build APIs.

After defining the new test methods, we will run the test:

Running Tests for the 201 Status Code and Location Header

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=store_should_respond_with_a_201_and_location_header_when_succ\
essful

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::store_should_respond_with_a_2\
01_and_location_header_when_successful
Response should have the header 'Location' but does not.
Failed asserting that false is true.

/home/vagrant/Code/bookr/tests/TestCase.php:25
/home/vagrant/Code/bookr/tests/TestCase.php:41
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:96

FAILURES!
Tests: 1, Assertions: 2, Failures: 1.
```

Implement the `Location` header to get our tests passing:

Add Location Header to BooksController@store

```php
/**
 * POST /books
 * @param Request $request
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function store(Request $request)
{
    $book = Book::create($request->all());

    return response()->json(['created' => true], 201, [
        'Location' => route('books.show', ['id' => $book->id])
    ]);
}
```

The `json()` method accepts an associative array of headers as the third argument, and we use this argument to set the `Location` header. The `route()` helper method takes a named route[55] to create

[55] https://lumen.laravel.com/docs/routing#named-routes

a URI for the value of the `Location` header. We haven't created a named route called "books.show" yet, so we need to define one now in `app/Http/routes.php`:

Add the BooksController@show Named Route

```
$app->get('/books', 'BooksController@index');
$app->get('/books/{id:[\d]+}', [
    'as' => 'books.show',
    'uses' => 'BooksController@show'
]);
$app->post('/books', 'BooksController@store');
```

The `show` route accepts an associative array with the named route `'as' => 'books.show'` and `uses` to define the controller.

> The dot notation used in the name route has no special meaning, but the namespace helps with organization. We could have named the route something like "show_book" if we wanted, but I personally follow the dot notation because it feels more organized.

Let's see if our test is passing now:

Run the Full Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (7 tests, 24 assertions)
```

Tests passing means we are done with our `BooksController@store` route.

> Git commit: Create a new Book
>
> 57761c8[56]

5.3: Updating an Existing Book

Now that we can create a new book the next feature is the ability to update an existing book. You will begin to understand (if you don't already) in this section that using our development database for tests is not a good idea; we will cover using a separate test database in Section 6.1. I hope you are getting a feel for using test-driven development to drive good design. Good design does not happen all at once, but through small increments of test and code cycles.

Here is the acceptance criteria for updating an existing book record the BooksControllerTest class (`tests/app/Http/Controllers/BooksControllerTest.php`):

[56] https://bitbucket.org/paulredmond/bookr/commits/57761c8

Acceptance criteria for updating a book

```php
/** @test **/
public function update_should_only_change_fillable_fields()
{
    $this->markTestIncomplete('pending');
}

/** @test **/
public function update_should_fail_with_an_invalid_id()
{
    $this->markTestIncomplete('pending');
}

/** @test **/
public function update_should_not_match_an_invalid_route()
{
    $this->markTestIncomplete('pending');
}
```

The first test will ensure that only fillable fields can be changed and that the changes are persisted in the database. The response should also return the updated record data and a `200 OK` status code.

Test for a successful book update

```php
/** @test **/
public function update_should_only_change_fillable_fields()
{
    $this->notSeeInDatabase('books', [
        'title' => 'The War of the Worlds'
    ]);

    $this->put('/books/1', [
        'id' => 5,
        'title' => 'The War of the Worlds',
        'description' => 'The book is way better than the movie.',
        'author' => 'Wells, H. G.'
    ]);

    $this
        ->seeStatusCode(200)
        ->seeJson([
            'id' => 1,
            'title' => 'The War of the Worlds',
            'description' => 'The book is way better than the movie.',
            'author' => 'Wells, H. G.'
        ])
        ->seeInDatabase('books', [
            'title' => 'The War of the Worlds'
        ]);
}
```

Our test assumes that a post with an id of `1` exists in the database from our seed data. We make a `PUT` request with changes all the fillable fields so we can assert that they get updated. Notice the test even tries to update the `id` which should not be allowed. Before making the `PUT` request our test makes sure a record doesn't already exist in the database. After updating the record the test verifies the response, the status code, and that a record exists with the new changes in the database.

Our test will fail with the expectation of a `200` response because we haven't defined a route:

Test Updating a Book

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::update_should_only_change_fil\
lable_fields
Failed asserting that 404 matches expected 200.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:115

FAILURES!
Tests: 10, Assertions: 26, Failures: 1, Incomplete: 2.
```

With the failing test written, define the route and the controller implementation:

Books update route (app/Http/routes.php)

```php
$app->get('/books', 'BooksController@index');
$app->get('/books/{id:[\d]+}', [
    'as' => 'books.show',
    'uses' => 'BooksController@show'
]);
$app->post('/books', 'BooksController@store');
$app->put('/books/{id:[\d]+}', 'BooksController@update');
```

First attempt at updating a book (BooksController.php)

```php
/**
 * PUT /books/{id}
 *
 * @param Request $request
 * @param $id
 * @return mixed
 */
public function update(Request $request, $id)
{
    $book = Book::findOrFail($id);

    $book->fill($request->all());
    $book->save();

    return $book;
}
```

The controller uses the `findOrFail` method you've already seen, which returns the book if it exists. The `$book->fill()` method, provided by Eloquent, takes an array of data from the request and only updates the model's `$fillable` fields. Next, the `$book->save()` method is called and the updated book returned.

Run the tests after writing the implementation to see if we can move on:

Run the Update Test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=update_should_only_change_fillable_fields

OK (1 test, 8 assertions)
```

I will spoil it for you—if you run the test again it will fail the second time. Why does it fail? Because we are using the development database to run tests. The second time we run our test, it will see the updated record in the database when it should not:

Our failing test that should pass

```
$this->notSeeInDatabase('books', [
    'title' => 'The War of the Worlds'
]);
```

Until we start using a separate test database we will have to purge and seed the data before each test run:

Refresh the Database Before Running Tests

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh && php artisan db:seed
$ phpunit --filter=update_should_only_change_fillable_fields

OK (1 test, 8 assertions)
```

We are ready to work on the second acceptance criteria: `update_should_fail_with_an_invalid_id`. The final two acceptance criteria don't need to insert a record, they just make sure that our route matches integer digits and that non-existent records return a 404 response. Lets break the rules a little and write tests for both features at the same time:

Writing remaining tests for the update action (BooksController.php)

```
/** @test **/
public function update_should_fail_with_an_invalid_id()
{
    $this
        ->put('/books/999999999999999')
        ->seeStatusCode(404)
        ->seeJsonEquals([
            'error' => [
                'message' => 'Book not found'
            ]
        ]);
}

/** @test **/
public function update_should_not_match_an_invalid_route()
{
    $this->put('/books/this-is-invalid')
        ->seeStatusCode(404);
}
```

Lets run all the `update_should` tests:

Run the Update Tests

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh && php artisan db:seed
$ phpunit --filter=update_should

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::update_should_fail_with_an_in\
valid_id
Failed asserting that 500 matches expected 404.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:132

FAILURES!
Tests: 3, Assertions: 10, Failures: 1.
```

It looks like the only failing test is `update_should_fail_with_an_invalid_id`. The app is responding with a `500` status code when we expect a `404`. We already dealt with this issue in the `GET /books/{id}` route, so see if you can fix this test on your own before you read on.

Respond to missing books with a 404 (BooksController.php)

```
/**
 * PUT /books/{id}
 * @param Request $request
 * @param $id
 * @return mixed
 */
public function update(Request $request, $id)
{
    try {
        $book = Book::findOrFail($id);
    } catch (ModelNotFoundException $e) {
        return response()->json([
            'error' => [
                'message' => 'Book not found'
            ]
        ], 404);
    }

    $book->fill($request->all());
    $book->save();

    return $book;
}
```

Like the `BooksController@show` method, the `@update` method catches the `ModelNotFoundException` and responds with an error message and a `404` status code. This should get our test suite passing again:

Run the Full Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh && php artisan db:seed
$ phpunit

OK (10 tests, 35 assertions)
```

Git Commit: Update a Book

1be1a96[57]

5.4: Deleting books

Deleting books is the last part of CRUD and the end of this really long chapter. Luckily, deleting is easy. Here is the `DELETE /books/{id}` route criteria:

Delete books acceptance criteria (BooksControllerTest.php)

```php
/** @test **/
public function destroy_should_remove_a_valid_book()
{
    $this->markTestIncomplete('pending');
}

/** @test **/
public function destroy_should_return_a_404_with_an_invalid_id()
{
    $this->markTestIncomplete('pending');
}

/** @test **/
public function destroy_should_not_match_an_invalid_route()
{
    $this->markTestIncomplete('pending');
}
```

Our first test will make sure we can successfully delete a book. The test expects a `204 No Content` status code and an empty response when deletion succeeds:

[57]https://bitbucket.org/paulredmond/bookr/commits/1be1a96

Test for successful deletion (BooksControllerTest.php)

```
/** @test **/
public function destroy_should_remove_a_valid_book()
{
    $this
        ->delete('/books/1')
        ->seeStatusCode(204)
        ->isEmpty();

    $this->notSeeInDatabase('books', ['id' => 1]);
}
```

Run the Test for Destroying a Book

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=destroy_should_remove_a_valid_book

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::destroy_should_remove_a_valid\
_book
Failed asserting that 405 matches expected 204.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:152

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

The failed test responds with a `405 Method not allowed` response. Lumen responds with the `405` without any work on our part until we define the `DELETE /books/{id}` route. A failed test means we are ready to implement our first version of the `BooksController@destroy` method and route:

BooksController::destroy() method (BooksController.php)

```
/**
 * DELETE /books/{id}
 * @param $id
 * @return \Illuminate\Http\JsonResponse
 */
public function destroy($id)
{
    $book = Book::findOrFail($id);
    $book->delete();

    return response(null, 204);
}
```

Just like our test outlines, we find the book and call `$book->delete()` on the model and omit a response body. We don't call `response()->json()` because we are not sending back a `204` response indicating that the server successfully fulfilled the request but there is no content to send back.

We need to define the accompanying route and then run the test:

The BooksController@destroy Route

```
$app->get('/books', 'BooksController@index');
$app->get('/books/{id:[\d]+}', [
    'as' => 'books.show',
    'uses' => 'BooksController@show'
]);
$app->put('/books/{id:[\d]+}', 'BooksController@update');
$app->post('/books', 'BooksController@store');
$app->delete('/books/{id:[\d]+}', 'BooksController@destroy');
```

Run the Test for Destroying a Book

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=destroy_should_remove_a_valid_book

OK (1 test, 2 assertions)
```

After we delete the record in the database our test will fail if we run it a second time because the record is removed. We will keep refreshing the migration and seeding data to wrap up this chapter. The final two tests are the same as the `PUT /books/{id}` tests:

Final BooksController::destroy() tests

```
/** @test **/
public function destroy_should_return_a_404_with_an_invalid_id()
{
    $this
        ->delete('/books/99999')
        ->seeStatusCode(404)
        ->seeJsonEquals([
            'error' => [
                'message' => 'Book not found'
            ]
        ]);
}

/** @test **/
public function destroy_should_not_match_an_invalid_route()
{
    $this->delete('/books/this-is-invalid')
        ->seeStatusCode(404);
}
```

Running the Tests for Deleting a Book

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh && php artisan db:seed
$ phpunit --filter=destroy_

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::destroy_should_return_a_404_w\
ith_an_invalid_id
Failed asserting that 500 matches expected 404.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:163

FAILURES!
Tests: 3, Assertions: 4, Failures: 1.
```

We have one final test failing and we will consider our initial BooksController complete. To get the final test passing we must catch the `ModelNotFoundException` like our other controller methods:

Respond with 404 error if missing (BooksController.php)

```php
/**
 * DELETE /books/{id}
 * @param $id
 * @return \Illuminate\Http\JsonResponse
 */
public function destroy($id)
{
    try {
        $book = Book::findOrFail($id);
    } catch (ModelNotFoundException $e) {
        return response()->json([
            'error' => [
                'message' => 'Book not found'
            ]
        ], 404);
    }

    $book->delete();

    return response(null, 204);
}
```

Our code change should get our tests passing:

Run the Full Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh && php artisan db:seed
$ phpunit

OK (13 tests, 40 assertions)
```

Success! We are done with our first version of the BooksController. Here is the full source code of the main files we are working on:

BooksController.php

```php
<?php

namespace App\Http\Controllers;

use App\Book;
use Illuminate\Http\Request;
use Illuminate\Database\Eloquent\ModelNotFoundException;

/**
 * Class BooksController
 * @package App\Http\Controllers
 */
class BooksController
{
    /**
     * GET /books
     * @return array
     */
    public function index()
    {
        return Book::all();
    }

    /**
     * GET /books/{id}
     * @param integer $id
     * @return mixed
     */
    public function show($id)
    {
        try {
            return Book::findOrFail($id);
        } catch (ModelNotFoundException $e) {
            return response()->json([
                'error' => [
                    'message' => 'Book not found'
                ]
            ], 404);
        }
    }

    /**
     * POST /books
     * @param Request $request
     * @return \Symfony\Component\HttpFoundation\Response
     */
    public function store(Request $request)
    {
        $book = Book::create($request->all());

        return response()->json(['created' => true], 201, [
            'Location' => route('books.show', ['id' => $book->id])
        ]);
    }

    /**
     * PUT /books/{id}
     * @param Request $request
     * @param $id
     * @return mixed
     */
    public function update(Request $request, $id)
    {
        try {
            $book = Book::findOrFail($id);
        } catch (ModelNotFoundException $e) {
            return response()->json([
                'error' => [
                    'message' => 'Book not found'
                ]
            ], 404);
        }

        $book->fill($request->all());
        $book->save();

        return $book;
    }

    /**
     * DELETE /books/{id}
     * @param $id
     * @return \Illuminate\Http\JsonResponse
     */
    public function destroy($id)
    {
        try {
            $book = Book::findOrFail($id);
        } catch (ModelNotFoundException $e) {
            return response()->json([
                'error' => [
                    'message' => 'Book not found'
                ]
            ], 404);
        }

        $book->delete();

        return response(null, 204);
    }
}
```

BooksControllerTest.php

```
<?php

namespace Tests\App\Http\Controllers;

use TestCase;

class BooksControllerTest extends TestCase
{
    /** @test **/
    public function index_status_code_should_be_200()
    {
        $this->visit('/books')->seeStatusCode(200);
    }

    /** @test **/
    public function index_should_return_a_collection_of_records()
    {
        $this
            ->get('/books')
            ->seeJson([
                'title' => 'War of the Worlds'
            ])
            ->seeJson([
                'title' => 'A Wrinkle in Time'
            ]);
    }

    /** @test **/
    public function show_should_return_a_valid_book()
    {
        $this
            ->get('/books/1')
            ->seeStatusCode(200)
            ->seeJson([
                'id' => 1,
                'title' => 'War of the Worlds',
                'description' => 'A science fiction masterpiece about Martians i\
nvading London',
                'author' => 'H. G. Wells'
            ]);

        $data = json_decode($this->response->getContent(), true);
        $this->assertArrayHasKey('created_at', $data);
        $this->assertArrayHasKey('updated_at', $data);
    }

    /** @test **/
    public function show_should_fail_when_the_book_id_does_not_exist()
    {
        $this
            ->get('/books/99999')
            ->seeStatusCode(404)
            ->seeJson([
                'error' => [
                    'message' => 'Book not found'
                ]
            ]);
    }

    /** @test **/
    public function show_route_should_not_match_an_invalid_route()
    {
        $this->get('/books/this-is-invalid');

        $this->assertNotRegExp(
            '/Book not found/',
            $this->response->getContent(),
            'BooksController@show route matching when it should not.'
        );
    }

    /** @test **/
    public function store_should_save_new_book_in_the_database()
    {
        $this->post('/books', [
            'title' => 'The Invisible Man',
            'description' => 'An invisible man is trapped in the terror of his o\
wn creation',
            'author' => 'H. G. Wells'
        ]);

        $this
            ->seeJson(['created' => true])
            ->seeInDatabase('books', ['title' => 'The Invisible Man']);
    }

    /** @test */
    public function store_should_respond_with_a_201_and_location_header_when_suc\
cessful()
    {
        $this->post('/books', [
            'title' => 'The Invisible Man',
            'description' => 'An invisible man is trapped in the terror of his o\
wn creation',
            'author' => 'H. G. Wells'
        ]);

        $this
            ->seeStatusCode(201)
            ->seeHeaderWithRegExp('Location', '#/books/[\d]+$#');

    }

    /** @test **/
    public function update_should_only_change_fillable_fields()
    {
        $this->notSeeInDatabase('books', [
            'title' => 'The War of the Worlds'
        ]);

        $this->put('/books/1', [
            'id' => 5,
            'title' => 'The War of the Worlds',
            'description' => 'The book is way better than the movie.',
            'author' => 'Wells, H. G.'
        ]);

        $this
            ->seeStatusCode(200)
            ->seeJson([
                'id' => 1,
                'title' => 'The War of the Worlds',
                'description' => 'The book is way better than the movie.',
                'author' => 'Wells, H. G.'
            ])
            ->seeInDatabase('books', [
                'title' => 'The War of the Worlds'
            ]);
    }

    /** @test **/
    public function update_should_fail_with_an_invalid_id()
    {
        $this
            ->put('/books/999999999999999')
            ->seeStatusCode(404)
            ->seeJsonEquals([
                'error' => [
                    'message' => 'Book not found'
                ]
            ]);
    }

    /** @test **/
    public function update_should_not_match_an_invalid_route()
    {
        $this->put('/books/this-is-invalid')
```

```
            ->seeStatusCode(404);
    }

    /** @test **/
    public function destroy_should_remove_a_valid_book()
    {
        $this
            ->delete('/books/1')
            ->seeStatusCode(204)
            ->isEmpty();

        $this->notSeeInDatabase('books', ['id' => 1]);
    }

    /** @test **/
    public function destroy_should_return_a_404_with_an_invalid_id()
    {
        $this
            ->delete('/books/99999')
            ->seeStatusCode(404)
            ->seeJsonEquals([
                'error' => [
                    'message' => 'Book not found'
                ]
            ]);
    }

    /** @test **/
    public function destroy_should_not_match_an_invalid_route()
    {
        $this->delete('/books/this-is-invalid')
            ->seeStatusCode(404);
    }
}
```

routes.php

```php
<?php

/*
|--------------------------------------------------------------------------
| Application Routes
|--------------------------------------------------------------------------
|
| Here is where you can register all of the routes for an application.
| It is a breeze. Simply tell Lumen the URIs it should respond to
| and give it the Closure to call when that URI is requested.
|
*/

$app->get('/', function () use ($app) {
    return $app->welcome();
});

$app->get('/books', 'BooksController@index');
$app->get('/books/{id:[\d]+}', [
    'as' => 'books.show',
    'uses' => 'BooksController@show'
]);
$app->put('/books/{id:[\d]+}', 'BooksController@update');
$app->post('/books', 'BooksController@store');
$app->delete('/books/{id:[\d]+}', 'BooksController@destroy');
```

If you are following along, commit your changes before moving on to the next chapter.

Git commit: Delete a Book

dfe5932[58]

Conclusion

We covered a lot of ground in this chapter, including:

- Testing first and then writing a passing implementation

[58]https://bitbucket.org/paulredmond/bookr/commits/dfe5932

- Creating new records in a database with Eloquent
- Updating existing records in the database
- Deleting existing records in the database
- Using correct response codes for creating, updating, and deleting resources
- How to define a named route

The `BooksController` is clean and simple, with Eloquent is doing the heavy lifing for our data. Our methods are small and concise which makes code easier to digest and maintain.

Chapter 6: Responding to Errors

During our work on the Book API in Chapter 5 it became evident that we need to start using a separate test database. We also don't have very good error responses for an API consumer yet. Responding to a client with an HTML error that expects JSON is not going to cut it anymore. When an API consumer interacts with our API, we want to respond with the correct `Content-Type` header. For now we will assume all of our consumers want JSON.

6.1: Test Database

The main focus of this chapter will be error responses, but before we work on error responses we will side-step and fix our glaring database testing issue. Lumen provides convenient and clever tools out of the box to support creating test data:

- Model factories
- DatabaseMigrations trait
- Faker[59] data generator

The basic steps needed to start using the test database include:

- Configuring PHPUnit to use the test database
- Create a model factory definition for the `Book` model
- Modify existing tests to use factory test data

Back in Chapter 3 we set up the `bookr_testing` database during our project setup in the `Homestead.yaml` file. If you don't have that database yet, you need to configure it now and re-run `vagrant provision`.

If you recall our project environment setup, the MySQL database is configured using phpdotenv[60]. We take advantage of that to set the test database used within PHPUnit. Open the `phpunit.xml` file in the root of the project and you will see opening and closing `<php></php>` tags that contain environment variables; we need to add the `DB_DATABASE` variable.

[59]https://github.com/fzaninotto/Faker
[60]https://github.com/vlucas/phpdotenv

Configure PHPUnit to use the testing database.

```
<php>
    <env name="APP_ENV" value="testing"/>
    <!-- Test Database -->
    <env name="DB_DATABASE" value="bookr_testing"/>
    <env name="CACHE_DRIVER" value="array"/>
    <env name="SESSION_DRIVER" value="array"/>
    <env name="QUEUE_DRIVER" value="sync"/>
</php>
```

I won't show the output if you run `phpunit` at this point, but you will get Exceptions, lots of them. Switching the database to `bookr_testing` means that we don't have any tables or data yet. To get data in our test database we need to configure a model factory and tweak our tests to take advantage of the `DatabaseMigrations` trait that Lumen provides.

Model Factories

Model factories provide fake test data that can be used to populate the test data before running a test. Factories generate random fake data you can use in a test, making your test data isolated and repeatable. In fact, each test requiring test data starts with a fresh database.

We only have one model right now, so we will define a factory we can use in our `BooksControllerTest.php`.All the model factory definitions should be added in the `database/factories/ModelFactory.php` file.

Add the Book Model Factory.

```
$factory->define(App\Book::class, function ($faker) {
    $title = $faker->sentence(rand(3, 10));

    return [
        'title' => substr($title, 0, strlen($title) - 1),
        'description' => $faker->text,
        'author' => $faker->name
    ];
});
```

We define a Book factory with the first argument `App\Book::class` which references the model. The second argument in the factory definition is a `Closure` which does the following:

- Uses `$faker` to generate a random sentence between 3 and 10 words long
- Return an array of fake data using Faker's various "formatters"
- Remove the period (.) from the sentence formatter with `substr`

Factories in Tests

We will update all our existing tests depending on test data to use our new factory. The only test file we have at this point is the `tests/app/Http/Controllers/BooksControllerTest.php` so open it up in your favorite editor.

The first test is the `index_should_return_a_collection_of_records` which ensures the `@index` method returns a collection of books.

Refactor `index_should_return_a_collection_of_records` test.

```php
/** @test **/
public function index_should_return_a_collection_of_records()
{
    $books = factory('App\Book', 2)->create();

    $this->get('/books');

    foreach ($books as $book) {
        $this->seeJson(['title' => $book->title]);
    }
}
```

The last code snippet is the first example of using the `factory()` helper function. We indicate that we want to use the `App\Book` factory, and that we want it to generate two book models. The second argument is optional, and if you omit it you will get one record back.

The `factory()` helper returns an instance of `Illuminate\Database\Eloquent\FactoryBuilder`, which has the methods `make()` and `create()`. We want to persist data to the database so we use `create()`; the `make()` method will return a model without saving it to the database.

The `factory()->create()` returns the model instances and we loop over them to make sure each model is represented in the `/books` response.

Now that we've see the first example of using a factory, we need a way to migrate and reset our database before each test requiring the database. Enter the `DatabaseMigrations` trait provided by the Lumen framework.

Add the `DatabaseMigrations` trait.

```php
<?php

namespace Tests\App\Http\Controllers;

use TestCase;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class BooksControllerTest extends TestCase
{
    use DatabaseMigrations;
    // ...
}
```

The `DatabaseMigrations` trait uses a `@before` PHPUnit annotation to migrate the database automatically.

Learn More About Model Factories

See the Model Factories[61] section of the official documentation for the full documentation on how you can use model factories.

It is time to run `phpunit` against our first test refactor:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=index_should_return_a_collection_of_records

OK (1 test, 4 assertions)
```

Now that you have a test with a factory working we can knock out the remaining test cases that need model data. We will cheat a little and fix the remaining tests before we run the whole test suite again.

Next up is the `GET /book/{id}` route test:

[61] http://lumen.laravel.com/docs/testing#model-factories

Refactor `show_should_return_a_valid_book` to use factories

```php
/** @test **/
public function show_should_return_a_valid_book()
{
    $book = factory('App\Book')->create();
    $this
        ->get("/books/{$book->id}")
        ->seeStatusCode(200)
        ->seeJson([
            'id' => $book->id,
            'title' => $book->title,
            'description' => $book->description,
            'author' => $book->author
        ]);

    $data = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('created_at', $data);
    $this->assertArrayHasKey('updated_at', $data);
}
```

Note that we use the `$book->id` to build the request URI and then assert the response from the `$book` instance. The remainder of the test stays the same.

Refactor `update_should_only_change_fillable_fields` to use model factories

```php
/** @test **/
public function update_should_only_change_fillable_fields()
{
    $book = factory('App\Book')->create([
        'title' => 'War of the Worlds',
        'description' => 'A science fiction masterpiece about Martians invading \
London',
        'author' => 'H. G. Wells',
    ]);

    $this->put("/books/{$book->id}", [
        'id' => 5,
        'title' => 'The War of the Worlds',
        'description' => 'The book is way better than the movie.',
        'author' => 'Wells, H. G.'
    ]);

    $this
        ->seeStatusCode(200)
        ->seeJson([
            'id' => 1,
            'title' => 'The War of the Worlds',
            'description' => 'The book is way better than the movie.',
            'author' => 'Wells, H. G.'
        ])
        ->seeInDatabase('books', [
            'title' => 'The War of the Worlds'
        ]);
}
```

The test is the first example of passing an array to the `factory()->create()` to override model values. We are not *required* to pass specific data to get this test passing, but I think it makes the test more readable. Note that we removed the `notSeeInDatabase()` assertion at the beginning because each test has a clean database.

Refactor `destroy_should_remove_a_valid_book` to use factories

```
/** @test **/
public function destroy_should_remove_a_valid_book()
{
    $book = factory('App\Book')->create();
    $this
        ->delete("/books/{$book->id}")
        ->seeStatusCode(204)
        ->isEmpty();

    $this->notSeeInDatabase('books', ['id' => $book->id]);
}
```

Our refactor is done, let's see if the tests are passing now:

Test suite after model factory refactoring

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (13 tests, 39 assertions)
```

You can run `phpunit` multiple times and the tests will pass every time, yay!

Git commit: Use model factories and a test database for tests

41d8878[62]

6.2: Better Error Responses

We are ready to improve our APIs error responses. Thus far we get back HTML when we have an application exception like a `ModelNotFoundException`, but an API consumer should get JSON. We could support other content types too, but we will focus on JSON responses.

Framework Exception Handling

So how does lumen deal with exceptions out of the box?

Lumen's Application::run() method at the time of writing (v5.1)

```
/**
 * Run the application and send the response.
 *
 * @param  SymfonyRequest|null  $request
 * @return void
 */
public function run($request = null)
{
    $response = $this->dispatch($request);

    if ($response instanceof SymfonyResponse) {
        $response->send();
    } else {
        echo (string) $response;
    }

    if (count($this->middleware) > 0) {
        $this->callTerminableMiddleware($response);
    }
}
```

The `run()` method is called from the front controller (public/index.php). The first line dispatches the request expecting a response back. If you view the source for the `Application::dispatch()` method you will see a `try/catch` where exceptions are caught and handled:

[62]https://bitbucket.org/paulredmond/bookr/commits/41d8878

Partial source of Lumen's `Application::dispatch()` method

```
try {
    return $this->sendThroughPipeline($this->middleware, function () use ($metho\
d, $pathInfo) {
        if (isset($this->routes[$method.$pathInfo])) {
            return $this->handleFoundRoute([true, $this->routes[$method.$pathInf\
o]['action'], []]);
        }

        return $this->handleDispatcherResponse(
            $this->createDispatcher()->dispatch($method, $pathInfo)
        );
    });
} catch (Exception $e) {
    return $this->sendExceptionToHandler($e);
} catch (Throwable $e) {
    return $this->sendExceptionToHandler($e);
}
```

Application exceptions are caught and then `Application::sendExceptionToHandler()` is called. The `sendExceptionToHandler()` appears as follows at the time of writing (v5.1):

The Application::sendExceptionToHandler() method.

```
/**
 * Send the exception to the handler and return the response.
 *
 * @param  \Throwable  $e
 * @return Response
 */
protected function sendExceptionToHandler($e)
{
    $handler = $this->make('Illuminate\Contracts\Debug\ExceptionHandler');

    if ($e instanceof Error) {
        $e = new FatalThrowableError($e);
    }

    $handler->report($e);

    return $handler->render($this->make('request'), $e);
}
```

The exception handler is resolved from the container on the first line, and is an instance of `app/Exceptions/Handler.php`. The handler is responsible for rendering the exception. Lumen uses a contract[63]—specifically the `Illuminate\Contracts\Debug\ExceptionHandler`—to bind the container to the application's handler. The `bootstrap/app.php` binds the application handler to the `ExceptionHandler` contract:

The Exception Handler contact.

```
$app->singleton(
    Illuminate\Contracts\Debug\ExceptionHandler::class,
    App\Exceptions\Handler::class
);
```

The Application class will resolve the `Handler` class in `app/Exceptions/Handler.php`. Here is the `Handler::render()` method:

The Exception Handler `render` method.

```
/**
 * Render an exception into an HTTP response.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Exception  $e
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $e)
{
    return parent::render($request, $e);
}
```

The application handler defers to the parent class to render out the exception. The parent of the application `Handler` is the Lumen framework Handler, which you can find at `vendor/laravel/lumen-framework/src/Exceptions/Handler.php`. I'll leave it to you to investigate the source if you want, but basically the default Handler uses the Symfony Debug component[64] `ExceptionHandler` class to send a response.

JSON Exceptions

Now that you understand a bit about how Lumen calls the `app/Exception/Handler` we need to update the `Handler::render()` method to respond with JSON instead of HTML when appropriate.

[63]http://laravel.com/docs/master/contracts
[64]http://symfony.com/doc/current/components/debug/introduction.html

Checking to See if the User Wants a JSON response.

```php
/**
 * Render an exception into an HTTP response.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Exception  $e
 * @return \Illuminate\Http\Response
 */
public function render($request, Exception $e)
{
    if ($request->wantsJson()) {
        $response = [
            'message' => (string) $e->getMessage(),
            'status' => 400
        ];

        if ($e instanceof HttpException) {
            $response['message'] = Response::$statusTexts[$e->getStatusCode()];
            $response['status'] = $e->getStatusCode();
        }

        if ($this->isDebugMode()) {
            $response['debug'] = [
                'exception' => get_class($e),
                'trace' => $e->getTrace()
            ];
        }

        return response()->json(['error' => $response], $response['status']);
    }

    return parent::render($request, $e);
}

/**
 * Determine if the application is in debug mode.
 *
 * @return Boolean
 */
public function isDebugMode()
{
    return (Boolean) env('APP_DEBUG');
}
```

We need to import `Response` at the top of our `Handler` class or we will get a fatal error:

Import the Response class

```php
<?php

namespace App\Exceptions;

use Exception;
use Symfony\Component\HttpFoundation\Response;
// ...

class Handler extends ExceptionHandler
```

Lets break down the changes we made to the `render` method:

- $request->wantsJson() checks if the user has “json” in the `Accept` header
- If the user doesn’t want JSON, skip and call `parent::render()` like the original method
- If the user wants JSON, we start building the response array which will be the returned JSON data
- We start out with the default exception message and a status code of `400 Bad Request` for generic errors
- If the exception is an instance of `Symfony\Component\HttpKernel\Exception\HttpException` change the message and status code of the exception (ie. status=404, message=Not Found)
- If debug mode is enabled, add the exception class and the stack trace
- Finally, return a JSON response with the assembled data

Lets try our new handler logic out in the console with `curl`:

Exception Response With Debugging Disabled for Brevity

```
# vagrant@homestead:~/Code/bookr$
$ curl -H"Accept: application/json" http://bookr.app/foo/bar

HTTP/1.0 404 Not Found
Host: localhost:8000
Connection: close
X-Powered-By: PHP/5.6.13
Cache-Control: no-cache
```

```
Date: Sun, 25 Oct 2015 06:37:31 GMT
Content-Type: application/json

{"error":{"message":"Not Found","status":404}}
```

If you have debugging turned on your response will have more data. We should check what happens if we don't send the `Accept` header:

```
# vagrant@homestead:~/Code/bookr$
$ curl -I http://bookr.app/foo/bar

HTTP/1.1 404 Not Found
Server: nginx/1.9.7
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
Cache-Control: no-cache, private
date: Wed, 06 Jan 2016 04:28:54 GMT
```

We have not asked for JSON, so the handler defers to the parent class. You can make the determination to always respond with JSON in your own applications, but for now we will keep our check and fall back to the default. We could have started writing tests before this change, but I thought it was more important to show you how the handler works under the hood and see the results of our modifications first.

6.3: Testing the Exception Handler

We have an exception handler that can respond to JSON, but we need to write tests to make sure behaves as we expect. We will write unit tests for the handler, and our existing API tests will cover integration tests.

A really useful for unit testing is a mocking library called Mockery[65], which will help us mock dependencies with relative ease. Mockery is not a requirement for Lumen so we need to install it with composer:

Require Mockery

```
# vagrant@homestead:~/Code/bookr$
$ composer require --dev mockery/mockery:~0.9.4
```

Now lets create our `Handler` unit test file:

[65] http://docs.mockery.io/en/latest/

```
# vagrant@homestead:~/Code/bookr$
$ mkdir -p tests/app/Exceptions
$ touch tests/app/Exceptions/HandlerTest.php
```

Add the following code to the new `HandlerTest.php` file:

The HandlerTest file (`tests/app/Exceptions/HandlerTest.php`)

```
<?php

namespace Tests\App\Exceptions;

use TestCase;
use \Mockery as m;
use App\Exceptions\Handler;
use Illuminate\Http\Request;
use Illuminate\Http\JsonResponse;

class HandlerTest extends TestCase
{

}
```

The following is a list of things we need to test based on the code we've written in the `Handler::render()` method:

- It responds with HTML when json *is not* requested
- It responds with json when json *is* requested
- It provides a default status code for non-HTTP exceptions
- It provides common http status codes for HTTP exceptions
- It provides debug information when debugging is *enabled*
- It skips debugging information when debugging is *disabled*

Our first test will ensure the application responds with HTML when JSON is not requested. Mockery makes mocking dependencies a breeze:

Test that the API responds with HTML when JSON is not accepted

```php
/** @test **/
public function it_responds_with_html_when_json_is_not_accepted()
{
    // Make the mock a partial, we only want to mock the `isDebugMode` method
    $subject = m::mock(Handler::class)->makePartial();
    $subject->shouldNotReceive('isDebugMode');

    // Mock the interaction with the Request
    $request = m::mock(Request::class);
    $request->shouldReceive('wantsJson')->andReturn(false);

    // Mock the interaction with the exception
    $exception = m::mock(\Exception::class, ['Error!']);
    $exception->shouldNotReceive('getStatusCode');
    $exception->shouldNotReceive('getTrace');
    $exception->shouldNotReceive('getMessage');

    // Call the method under test, this is not a mocked method.
    $result = $subject->render($request, $exception);

    // Assert that `render` does not return a JsonResponse
    $this->assertNotInstanceOf(JsonResponse::class, $result);
}
```

We mock the class under test (`App\Exceptions\Hanler`) as a partial mock[66]. This allows us to mock certain methods, but the other methods respond normally. In our case, we want to control the `return` of the `isDebugMode` method we created. Mocking `isDebugMode` allows us to assert that it was never called! The expectation declaration[67] `shouldNotReceive` means that the mocked method should never receive a call.

We mock `Illuminate\Http\Request` and control the flow of the `render` method by instructing that `wantsJson` returns `false`. If `wantsJson` returns `false` the entire block will be skipped and thus `Handler::isDebugMode()` will not be called. You can see how simple mocks make controlling the flow of the method under test.

The last mock we create before calling `$subject->render()` is an `\Exception`. Note in the `m::mock()` call that we pass an array of constructor arguments (`m::mock('ClassName', [arg1,arg2])`). Like the partial mock, we set up three `shouldNotRecieve` expectations for the exception mock: `getStatusCode`, `getTrace`, and `getMessage`.

[66]http://docs.mockery.io/en/latest/reference/partial_mocks.html
[67]http://docs.mockery.io/en/latest/reference/expectations.html

Lastly, we actually call `$subject->render($request, $exception);`. We use a PHP assertion to make sure that the `render` method did not return an instance of the `JsonResponse` class. Note that because we partially mocked the subject of this test, the `render` method responds and works normally.

Now that we have an understanding of the test, lets run the first test:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=it_responds_with_html_when_json_is_not_accepted

OK (1 test, 1 assertion)
```

Everything passed, but all that work for one assertion!? Fortunately, mockery provides a way to make expectations[68] count as assertions in PHPUnit. Open up the base `TestCase` class found in `tests/TestCase.php` and add the following:

Mockery's PHPUnit Integration Trait (partial source)

```php
<?php

use Mockery\Adapter\Phpunit\MockeryPHPUnitIntegration;

class TestCase extends Laravel\Lumen\Testing\TestCase
{
    use MockeryPHPUnitIntegration;
    // ...
}
```

Now we should get more assertions:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=it_responds_with_html_when_json_is_not_accepted

OK (1 test, 6 assertions)
```

Great, we want our hard work and commitment to writing tests mean something. I personally don't know if I would write all those mocks without the assertion count payoff :).

[68]http://docs.mockery.io/en/latest/reference/expectations.html

Mocking

Mocking is an invaluable tool in your testing arsenal. It forces you to think about good code design. Mocking the interactions a class has with other dependencies helps you detect complicated code and hard-to-test code. A class with many dependencies can become difficult to mock, which might mean that you should refactor.

Mocking is an easy enough concept to understand, but practical use is an art that takes practice. Don't give up on using mocking in your tests, the concepts will eventually click! I encourage you to read through the whole Mockery[a] documentation and practice mocking.

The phpspec[b] library is another good library that uses mocks for tests. In this book we will stick to PHPUnit + Mockery, but phpspec is an excellent choice too!

[a]http://docs.mockery.io/en/latest/index.html
[b]http://phpspec.readthedocs.org/en/latest/

We've covered the test for what happens when our app is rendering an exception response, but the user doesn't want JSON back. Now we are going to test the JSON response when the user *wants* JSON:

Test that the API responds with JSON on an exception

```
/** @test */
public function it_responds_with_json_for_json_consumers()
{
    $subject = m::mock(Handler::class)->makePartial();
    $subject
        ->shouldReceive('isDebugMode')
        ->andReturn(false);

    $request = m::mock(Request::class);
    $request
        ->shouldReceive('wantsJson')
        ->andReturn(true);

    $exception = m::mock(\Exception::class, ['Doh!']);
    $exception
        ->shouldReceive('getMessage')
        ->andReturn('Doh!');

    /** @var JsonResponse $result */
    $result = $subject->render($request, $exception);
    $data = $result->getData();

    $this->assertInstanceOf(JsonResponse::class, $result);
    $this->assertObjectHasAttribute('error', $data);
    $this->assertAttributeEquals('Doh!', 'message', $data->error);
    $this->assertAttributeEquals(400, 'status', $data->error);
}
```

The mocks in our latest test are very similar to the first handler test, but this time we mock our method under test to go into the `if ($request->wantsJson()) {` logic. Inside the `if ($request->wantsJson()) {` statement there are calls that we need to mock that should receive. The `$exception` mock should receive a call to `getMessage` and return `Doh!` since that is the parameter we passed to the `$exception` mock constructor.

The `$exception` is not an instance of `Symfony\Component\HttpKernel\Exception\HttpException` so our test will use the default status code and message from the exception. We then assert the method under test returns an instance of `Illuminate\Http\JsonResponse`. Finally we assert the response body for the correct keys, values, and status code.

A couple more tests and we will call our exception response handler good. The next test needs to import a few classes to the top of our `HandlerTest` class.

Add a few HTTP exception classes (`tests/app/Exceptions/HandlerTest.php`)

```php
<?php

namespace Tests\App\Exceptions;

// ...
use Symfony\Component\HttpKernel\Exception\AccessDeniedHttpException;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class HandlerTest extends TestCase
```

And now the final test for the `Handler.php` class. This test ensures that classes extending from `HttpException` will respond with the matching HTTP request status code and message:

Testing HTTP Exception responses

```php
/** @test */
public function it_provides_json_responses_for_http_exceptions()
{
    $subject = m::mock(Handler::class)->makePartial();
    $subject
        ->shouldReceive('isDebugMode')
        ->andReturn(false);

    $request = m::mock(Request::class);
    $request->shouldReceive('wantsJson')->andReturn(true);

    $examples = [
        [
            'mock' => NotFoundHttpException::class,
            'status' => 404,
            'message' => 'Not Found'
        ],
        [
            'mock' => AccessDeniedHttpException::class,
            'status' => 403,
            'message' => 'Forbidden'
        ]
    ];

    foreach ($examples as $e) {
        $exception = m::mock($e['mock']);
        $exception->shouldReceive('getMessage')->andReturn(null);
        $exception->shouldReceive('getStatusCode')->andReturn($e['status']);

        /** @var JsonResponse $result */
        $result = $subject->render($request, $exception);
        $data = $result->getData();

        $this->assertEquals($e['status'], $result->getStatusCode());
        $this->assertEquals($e['message'], $data->error->message);
        $this->assertEquals($e['status'], $data->error->status);
    }
}
```

The `$subject` and `$request` mocks should look familiar at this point. After mocking and setting expectations, the test creates an array of `$examples` that we will loop over to test a few exceptions that extend from the `HttpException` class. The `foreach` loop mocks each example and sets mockery expectations. Each loop will do the following:

- Set `shouldReceive` expectation for `getMessage` and `getStatusCode`
- The `$subject->render()` call is made on the partially mocked test subject
- Make PHPUnit assertions about the response `status` and `message` keys.

We are at a point where we should run tests before moving on.

Run the Handler Tests and the Full Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=HandlerTest

OK (3 tests, 25 assertions)

$ phpunit

OK (16 tests, 64 assertions)
```

Before we call it good, lets refactor the `BooksController@show` method to not catch exceptions from the `Book::findOrFail()` method call.

This is the current version of the `@show` method:

Current ‘BooksController@show method

```
/**
 * GET /books/{id}
 * @param integer $id
 * @return mixed
 */
public function show($id)
{
    try {
        return Book::findOrFail($id);
    } catch (ModelNotFoundException $e) {
        return response()->json([
            'error' => [
                'message' => 'Book not found'
            ]
        ], 404);
    }
}
```

Lets remove the `try/catch` to see how our `App\Exceptions\Handler::render()` method handles the `findOrFail()` call:

Remove try/catch from the ‘BooksController@show method

```
/**
 * GET /books/{id}
 * @param integer $id
 * @return mixed
 */
public function show($id)
{
    return Book::findOrFail($id);
}
```

Now try making a request to an invalid record:

```
# vagrant@homestead:~/Code/bookr$
$ curl -i -H"Accept: application/json" http://bookr.app/books/5555

HTTP/1.1 400 Bad Request
Server: nginx/1.9.7
Content-Type: application/json
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache
Date: Wed, 06 Jan 2016 05:00:35 GMT

{"error":{"message":"No query results for model [App\\Book].","status":400}}
```

It looks like our Handler.php adjustments are working, but a `400` is not exactly the right response for a `ModelNotFoundException`. If you run `phpunit` now you will get a failure, which helps us avoid bugs. Lets make one last adjustment to our Handler::render() method to account for a `ModelNotFoundException`.

Additional Check for ModelNotFoundException in

```php
if ($e instanceof HttpException) {
    $response['message'] = Response::$statusTexts[$e->getStatusCode()];
    $response['status'] = $e->getStatusCode();
} else if ($e instanceof ModelNotFoundException) {
    $response['message'] = Response::$statusTexts[Response::HTTP_NOT_FOUND];
    $response['status'] = Response::HTTP_NOT_FOUND;
}
```

We need to import the `ModelNotFoundException` class at the top of our `app/Exceptions/Handler.php` file:

Import the ModelNotFoundException class

```php
<?php

namespace App\Exceptions;

use Exception;
use Symfony\Component\HttpFoundation\Response;
use Illuminate\Database\Eloquent\ModelNotFoundException;
```

Lets try our request again and then run our test suite:

```
# vagrant@homestead:~/Code/bookr$
$ curl -i -H"Accept: application/json" http://bookr.app/books/5555

HTTP/1.1 404 Not Found
Server: nginx/1.9.7
Content-Type: application/json
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache
Date: Wed, 06 Jan 2016 05:07:21 GMT

{"error":{"message":"Not Found","status":404}}
```

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::show_should_fail_when_the_boo\
k_id_does_not_exist
Failed asserting that 500 matches expected 404.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:54

FAILURES!
Tests: 16, Assertions: 62, Failures: 1.
```

Weird. Our failing test claims that we expected a 404, but we got a 500 back. Our test is not asking for JSON with `Accept: application/json` and the `return parent::render($request, $e);` part of our handler is returning a `500` HTML response.

Fix broken BooksController@show test

```
/** @test **/
public function show_should_fail_when_the_book_id_does_not_exist()
{
    $this
        ->get('/books/99999', ['Accept' => 'application/json'])
        ->seeStatusCode(404)
        ->seeJson([
            'message' => 'Not Found',
            'status'  => 404
        ]);
}
```

The `$this->get()` method can pass an array of headers; the `Accept` header will trigger our `Handler::response()` JSON logic. Next we changed the `seeJson` check to match the `ModelNot-FoundException` response message and status; we gain more consistent `404` messages by allowing the `Handler` to deal with them.

Run the Full Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (16 tests, 65 assertions)
```

Much better! We added more logic to our `Handler` class to deal with the `ModelNotFoundException` so we need to account for the added code in our test suite. We will add the `ModelNotFoundException` to our array of examples that we loop through by modifying an existing test:

Add the `ModelNotFoundException` to our `$examples` array.

```php
/** @test */
public function it_provides_json_responses_for_http_exceptions()
{
    $subject = m::mock(Handler::class)->makePartial();
    $subject
        ->shouldReceive('isDebugMode')
        ->andReturn(false);

    $request = m::mock(Request::class);
    $request->shouldReceive('wantsJson')->andReturn(true);

    $examples = [
        [
            'mock' => NotFoundHttpException::class,
            'status' => 404,
            'message' => 'Not Found'
        ],
        [
            'mock' => AccessDeniedHttpException::class,
            'status' => 403,
            'message' => 'Forbidden'
        ],
        [
            'mock' => ModelNotFoundException::class,
            'status' => 404,
            'message' => 'Not Found'
        ]
    ];

    foreach ($examples as $e) {
        $exception = m::mock($e['mock']);
        $exception->shouldReceive('getMessage')->andReturn(null);
        $exception->shouldReceive('getStatusCode')->andReturn($e['status']);

        /** @var JsonResponse $result */
        $result = $subject->render($request, $exception);
        $data = $result->getData();

        $this->assertEquals($e['status'], $result->getStatusCode());
        $this->assertEquals($e['message'], $data->error->message);
        $this->assertEquals($e['status'], $data->error->status);
    }
}
```

We need to import the `ModelNotFoundException` class and run the test suite.

Import `ModelNotFoundException` to our HandlerTest class.

```
<?php

namespace Tests\App\Exceptions;

// ...
use Illuminate\Database\Eloquent\ModelNotFoundException;

class HandlerTest extends TestCase
{
    // ...
}
```

Run the Full Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (16 tests, 70 assertions)
```

Adding a check for the `ModelNotFoundException` was fairly easy! Our Handler is in good shape and we will call it good. We now have test database migrations and factories working and a better exception response handler class.

Git commit: Handle Exceptions with a JSON Response

ec12420[69]

[69]https://bitbucket.org/paulredmond/bookr/commits/ec12420

Here are the final `Handler` and `HandlerTest` files in full:

Final Handler.php file

```php
<?php

namespace App\Exceptions;

use Exception;
use Symfony\Component\HttpFoundation\Response;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Symfony\Component\HttpKernel\Exception\HttpException;
use Laravel\Lumen\Exceptions\Handler as ExceptionHandler;

class Handler extends ExceptionHandler
{
    /**
     * A list of the exception types that should not be reported.
     *
     * @var array
     */
    protected $dontReport = [
        HttpException::class,
    ];

    /**
     * Report or log an exception.
     *
     * This is a great spot to send exceptions to Sentry, Bugsnag, etc.
     *
     * @param  \Exception  $e
     * @return void
     */
    public function report(Exception $e)
    {
        return parent::report($e);
    }

    /**
     * Render an exception into an HTTP response.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Exception  $e
     * @return \Illuminate\Http\Response
     */
    public function render($request, Exception $e)
    {
        if ($request->wantsJson()) {
            $response = [
                'message' => (string) $e->getMessage(),
                'status' => 400
            ];

            if ($e instanceof HttpException) {
                $response['message'] = Response::$statusTexts[$e->getStatusCode(\
)];
                $response['status'] = $e->getStatusCode();
            } else if ($e instanceof ModelNotFoundException) {
                $response['message'] = Response::$statusTexts[Response::HTTP_NOT\
_FOUND];
                $response['status'] = Response::HTTP_NOT_FOUND;
            }

            if ($this->isDebugMode()) {
                $response['debug'] = [
                    'exception' => get_class($e),
                    'trace' => $e->getTrace()
                ];
            }

            return response()->json(['error' => $response], $response['status']);
        }

        return parent::render($request, $e);
    }

    /**
     * Determine if the application is in debug mode.
     *
     * @return Boolean
     */
    public function isDebugMode()
    {
        return (Boolean) env('APP_DEBUG');
    }
```

```
}
```

Final HandlerTest.php

```php
<?php

namespace Tests\App\Exceptions;

use TestCase;
use \Mockery as m;
use App\Exceptions\Handler;
use Illuminate\Http\Request;
use Illuminate\Http\JsonResponse;
use Symfony\Component\HttpKernel\Exception\AccessDeniedHttpException;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use Illuminate\Database\Eloquent\ModelNotFoundException;

class HandlerTest extends TestCase
{
    /** @test **/
    public function it_responds_with_html_when_json_is_not_accepted()
    {
        // Make the mock a partial, we only want to mock the `isDebugMode` method
        $subject = m::mock(Handler::class)->makePartial();
        $subject->shouldNotReceive('isDebugMode');

        // Mock the interaction with the Request
        $request = m::mock(Request::class);
        $request->shouldReceive('wantsJson')->andReturn(false);

        // Mock the interaction with the exception
        $exception = m::mock(\Exception::class, ['Error!']);
        $exception->shouldNotReceive('getStatusCode');
        $exception->shouldNotReceive('getTrace');
        $exception->shouldNotReceive('getMessage');

        // Call the method under test, this is not a mocked method.
        $result = $subject->render($request, $exception);

        // Assert that `render` does not return a JsonResponse
        $this->assertNotInstanceOf(JsonResponse::class, $result);
    }

    /** @test */
    public function it_responds_with_json_for_json_consumers()
    {
        $subject = m::mock(Handler::class)->makePartial();
        $subject
            ->shouldReceive('isDebugMode')
            ->andReturn(false);

        $request = m::mock(Request::class);
        $request
            ->shouldReceive('wantsJson')
            ->andReturn(true);

        $exception = m::mock(\Exception::class, ['Doh!']);
        $exception
            ->shouldReceive('getMessage')
            ->andReturn('Doh!');

        /** @var JsonResponse $result */
        $result = $subject->render($request, $exception);
        $data = $result->getData();

        $this->assertInstanceOf(JsonResponse::class, $result);
        $this->assertObjectHasAttribute('error', $data);
        $this->assertAttributeEquals('Doh!', 'message', $data->error);
        $this->assertAttributeEquals(400, 'status', $data->error);
    }

    /** @test */
    public function it_provides_json_responses_for_http_exceptions()
    {
        $subject = m::mock(Handler::class)->makePartial();
        $subject
            ->shouldReceive('isDebugMode')
            ->andReturn(false);

        $request = m::mock(Request::class);
        $request->shouldReceive('wantsJson')->andReturn(true);

        $examples = [
            [
                'mock' => NotFoundHttpException::class,
                'status' => 404,
                'message' => 'Not Found'
            ],
            [
                'mock' => AccessDeniedHttpException::class,
                'status' => 403,
                'message' => 'Forbidden'
            ],
            [
                'mock' => ModelNotFoundException::class,
                'status' => 404,
                'message' => 'Not Found'
            ]
        ];

        foreach ($examples as $e) {
            $exception = m::mock($e['mock']);
            $exception->shouldReceive('getMessage')->andReturn(null);
            $exception->shouldReceive('getStatusCode')->andReturn($e['status']);

            /** @var JsonResponse $result */
            $result = $subject->render($request, $exception);
            $data = $result->getData();

            $this->assertEquals($e['status'], $result->getStatusCode());
            $this->assertEquals($e['message'], $data->error->message);
            $this->assertEquals($e['status'], $data->error->status);
        }
    }
}
```

Conclusion

Our API responds to exceptions more intelligently and API consumers will get better feedback when things go wrong. Along the way, we worked on:

- Defining model factories
- Using a dedicated test database
- Installing and using Mockery
- Customizing the `App\Exceptions\Handler` response

- Understanding how Lumen responds to exceptions

I hope you spend more time on your own playing with Mockery; it’s a wonderful mocking library. Mocking takes time and practice, but well worth the effort.

Chapter 7: Leveling Up Responses

Our API responses work fine, but at the moment they don't scale very well. We need to make them more consistent across all endpoints by using conventions, as well as anticipate adding things like pagination to our response. If you've consumed a few APIs, you have probably notice that response data formats vary between each API. Understanding each API format can be challenging enough, but add inconsistencies between endpoints within the same API and it can be a frustrating user experience.

Our current API response for all books looks something like this:

The GET /books JSON Response

```
[{
    "id": 1,
    "title": "War of the Worlds",
    "description": "A science fiction masterpiece about Martians invading London\
",
    "author": "H. G. Wells",
    "created_at": "2015-10-21 06:54:33",
    "updated_at": "2015-10-21 06:54:33"
}, {
    "id": 2,
    "title": "A Wrinkle in Time",
    "description": "A young girl goes on a mission to save her father who has go\
ne missing after working on a mysterious project called a tesseract.",
    "author": "Madeleine L'Engle",
    "created_at": "2015-10-21 06:54:33",
    "updated_at": "2015-10-21 06:54:33"
}]
```

The data format not *bad* per-say, but what if we want to add metadata like pagination? Where will it go? It's easy to see how quickly our current response will crumble. Plus, right now each controller is responsible for formatting its own response.

7.1: Introducing Fractal

I would argue that the response handling code is one of the most critical parts of an API, and so we need to offload the responsiblity from the controller to a service that the controller can use. We could write the code from scratch on our own, but Fractal[70] is library that provides a good (and flexible) solution for us. Fractal describes itself as follows:

> Fractal provides a presentation and transformation layer for complex data output, the like found in RESTful APIs, and works really well with JSON. Think of this as a view layer for your JSON/YAML/etc.
>
> When building an API it is common for people to just grab stuff from the database and pass it to `json_encode()`. This might be passable for "trivial" APIs but if they are in use by the public, or used by mobile applications then this will quickly lead to inconsistent output.

Sounds exactly like what we need! Out of the box we can quickly get a format like this:

Example JSON Response Using Fractal

```
{
    "data": [{
        "id": 1,
        "title": "War of the Worlds",
        "description": "A science fiction masterpiece about Martians invading Lo\
ndon",
        "author": "H. G. Wells",
        "created_at": "2015-10-21 06:54:33",
        "updated_at": "2015-10-21 06:54:33"
    }, {
        "id": 2,
        "title": "A Wrinkle in Time",
        "description": "A young girl goes on a mission to save her father who ha\
s gone missing after working on a mysterious project called a tesseract.",
        "author": "Madeleine L'Engle",
        "created_at": "2015-10-21 06:54:33",
        "updated_at": "2015-10-21 06:54:33"
    }]
}
```

With the response content in a `"data"` key, we can add other things to the response without mixing it in with the book data. Fractal encourages good data design and responses will be consistent across the API.

We need to install Fractal with composer and then configure it so we can use it in our controllers. Various libraries exist to even integrate Fractal, but but we are going to roll our own. Sometimes

[70] http://fractal.thephpleague.com/

rolling your own is better than adding a dependency. Using Fractal is simple enough to use that we don't need to use a Laravel/Lumen integration.

I try to keep application dependencies to a minimum and invest in dependencies where it counts. You add complexity to your application with each new dependency and you increase the size of your application. Always weigh the value of using a vendor vs. rolling your own. In our case, Fractal makes sense and provides a good value for data transformation; but by rolling our own service provider to integrate Fractal you will learn about (if you haven't used them in Laravel) how to write services.

7.2: First Version of API Response Formatting

Let's get to work on our first version of refactoring response formatting. We could start integrating Fractal immediately, but that's not an iterative approach. First, we will write test specifications and make the minimum change needed to support our response changes. Once we have the tests in place, we will be ready to make the move to the Fractal library.

We will cover the `BooksController@index` route test first:

Update BooksController@index Test

```php
/** @test **/
public function index_should_return_a_collection_of_records()
{
    $books = factory('App\Book', 2)->create();

    $this->get('/books');
    $expected = [
        'data' => $books->toArray()
    ];

    $this->seeJsonEquals($expected);
}
```

We removed the foreach loop, and opted to use the `Collection::toArray()` method to create the expected JSON response. We check the data with `seeJsonEquals()` to make sure our response matches the actual response

Failing phpunit test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

1) Tests\App\Http\Controllers\BooksControllerTest::index_should_return_a_collect\
ion_of_records
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'{"data":[{"author":"Dortha Hodkiewicz"...
+'[{"author":"Dortha Hodkiewicz",...

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:338
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:28

FAILURES!
Tests: 16, Assertions: 67, Failures: 1.
```

The change to get our implementation passing is simply adding a `data` array key:

BooksController@index Data Implementation

```php
/**
 * GET /books
 * @return array
 */
public function index()
{
    return ['data' => Book::all()->toArray()];
}
```

Passing PHPUnit Test for `BooksController@index`

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (16 tests, 67 assertions)
```

Next, we update the `BooksController@show` test:

Update the BooksController@show Test.

```
/** @test **/
public function show_should_return_a_valid_book()
{
    $book = factory('App\Book')->create();
    $expected = [
        'data' => $book->toArray()
    ];
    $this
        ->get("/books/{$book->id}")
        ->seeStatusCode(200)
        ->seeJsonEquals($expected);
}
```

Failing PHPUnit Test for the `BooksController@show` Route

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

1) Tests\App\Http\Controllers\BooksControllerTest::show_should_return_a_valid_bo\
ok
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'{"data":{"author":"Dr. Wyman Brown"...
+'{"author":"Dr. Wyman Brown"...

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:338
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:41

FAILURES!
Tests: 16, Assertions: 61, Failures: 1.
```

Getting the BooksController@show Route Back to Green

```php
/**
 * GET /books/{id}
 * @param integer $id
 * @return mixed
 */
public function show($id)
{
    return ['data' => Book::findOrFail($id)->toArray()];
}
```

Passing Tests After Updating `BooksController@show`

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (16 tests, 61 assertions)
```

The remaining routes are `store` and the `update`. In the original `BooksController@store` method we cheated a little bit and responded with `{"created": true}`, but let's return the new record data instead.

Update the BooksController@store Response

```php
/** @test **/
public function store_should_save_new_book_in_the_database()
{
    $this->post('/books', [
        'title' => 'The Invisible Man',
        'description' => 'An invisible man is trapped in the terror of his own c\
reation',
        'author' => 'H. G. Wells'
    ]);

    $body = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $body);

    $data = $body['data'];
    $this->assertEquals('The Invisible Man', $data['title']);
    $this->assertEquals(
        'An invisible man is trapped in the terror of his own creation',
        $data['description']
    );
    $this->assertEquals('H. G. Wells', $data['author']);
    $this->assertTrue($data['id'] > 0, 'Expected a positive integer, but did not\
 see one.');
    $this->seeInDatabase('books', ['title' => 'The Invisible Man']);
}
```

Failing `BooksController@store` Test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::store_should_save_new_book_in\
_the_database
Failed asserting that an array has the key 'data'.

/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:78

FAILURES!
Tests: 16, Assertions: 59, Failures: 1.
```

The implementation swaps out the `"created": true` JSON with the newly created book data:

Updating the `BooksController@store` Method

```
/**
 * POST /books
 * @param Request $request
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function store(Request $request)
{
    $book = Book::create($request->all());

    return response()->json(['data' => $book->toArray()], 201, [
        'Location' => route('books.show', ['id' => $book->id])
    ]);
}
```

Passing BooksController@store Test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (16 tests, 64 assertions)
```

The `BooksController@update` test is next. We will update the code before writing the test to illustrate a caveat of using `seeJson()`:

Updating the `BooksController@update` Method

```php
/**
 * PUT /books/{id}
 * @param Request $request
 * @param $id
 * @return mixed
 */
public function update(Request $request, $id)
{
    try {
        $book = Book::findOrFail($id);
    } catch (ModelNotFoundException $e) {
        return response()->json([
            'error' => [
                'message' => 'Book not found'
            ]
        ], 404);
    }

    $book->fill($request->all());
    $book->save();

    return ['data' => $book->toArray()];
}
```

Running Tests After Changing BooksController@update. Will it fail?

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (16 tests, 64 assertions)
```

The controller change did not cause our tests to fail—not good. We've introduced a breaking change to the API that our tests failed to catch. Our test does some response checking with `seeJson`, but it doesn't check the integrity of the JSON response. The `seeJson()` method checks *fragments* of JSON but not the location of the data. Be careful when you are writing tests to ensure the response format carefully. The `seeJson()` method is not to blame, we need to test for the `data` key:

Check for the Data Key in the Response

```php
/** @test **/
public function update_should_only_change_fillable_fields()
{
    $book = factory('App\Book')->create([
        'title' => 'War of the Worlds',
        'description' => 'A science fiction masterpiece about Martians invading \
London',
        'author' => 'H. G. Wells',
    ]);

    $this->notSeeInDatabase('books', [
        'title' => 'The War of the Worlds',
        'description' => 'The book is way better than the movie.',
        'author' => 'Wells, H. G.'
    ]);

    $this->put("/books/{$book->id}", [
        'id' => 5,
        'title' => 'The War of the Worlds',
        'description' => 'The book is way better than the movie.',
        'author' => 'Wells, H. G.'
    ]);

    $this
        ->seeStatusCode(200)
        ->seeJson([
            'id' => 1,
            'title' => 'The War of the Worlds',
            'description' => 'The book is way better than the movie.',
            'author' => 'Wells, H. G.',
        ])
        ->seeInDatabase('books', [
            'title' => 'The War of the Worlds'
        ]);

    // Verify the data key in the response
    $body = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $body);
}
```

At the end of the test we check the response body for the existence of the `data` key. If you revert the controller change we did first, you should now get a failing test.

Test Results of Reverting the BooksController@update Route

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::update_should_only_change_fil\
lable_fields
Failed asserting that an array has the key 'data'.

/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:142

FAILURES!
Tests: 16, Assertions: 66, Failures: 1.
```

The test suite should be fully passing again. If you reverted the `BooksController@update` method to see the test fail, put the change back.

Fully Passing Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (16 tests, 66 assertions)
```

Git commit: Book Responses Contained in a "data" Attribute

fc3113f[71]

[71]https://bitbucket.org/paulredmond/bookr/commits/fc3113f

7.2: The FractalResponse Class

Now that our tests are passing again, we are free to start refactoring code. We could end the chapter now, but experience says that changing arrays to include a `data` key like we've done isn't going to scale very well. It does afford us passing tests and gives us the green light to refactor and make sure our tests still pass.

The first step is installing the Fractal[72] library.

Install Fractal with composer

```
# vagrant@homestead:~/Code/bookr$
$ composer require league/fractal=^0.13.0
```

We will take the following steps to refactor responses:

- Create a transformer[73] for the `Book` model
- Create a dedicated Fractal service to transform models
- Create a service provider[74] to define the Fractal service
- Use the service in the `BooksController` to return responses

The Book Transformer

The first step is creating a transformer for the `Book` model. Transformers in fractal are responsible for transforming data into an array format. Our application will pass Eloqent models into transformers and the transformer will be responsible for formatting the model data into an array. You will see shortly how this works.

Creating the `BookTransformerTest.php` Test

```
# vagrant@homestead:~/Code/bookr$
$ mkdir -p tests/app/Transformer/
$ touch tests/app/Transformer/BookTransformerTest.php
```

Here is the skeleton of our class:

[72] http://fractal.thephpleague.com/
[73] http://fractal.thephpleague.com/transformers/
[74] http://lumen.laravel.com/docs/providers

The BookTransformerTest Class Skeleton

```php
<?php

namespace Tests\App\Transformer;

use TestCase;
use App\Book;
use App\Transformer\BookTransformer;
use League\Fractal\TransformerAbstract;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class BookTransformerTest extends TestCase
{
    use DatabaseMigrations;

    /** @test **/
    public function it_can_be_initialized()
    {
        $subject = new BookTransformer();
        $this->assertInstanceOf(TransformerAbstract::class, $subject);
    }
}
```

I like to write an initalization test for unit tests. I find that starting with a simple initialization test gets me past the mental hurdle of starting to write tests for a class.

First Test Run For The BookTransforerTest

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=BookTransformerTest

There was 1 error:

1) Tests\App\Transformer\BookTransformerTest::it_can_be_initialized
Error: Class 'App\Transformer\BookTransformer' not found

/home/vagrant/Code/bookr/tests/app/Transformer/BookTransformerTest.php:18

FAILURES!
Tests: 1, Assertions: 0, Errors: 1.
```

Obviously, we haven't defined the `BookTransformer` yet; to get the initialization test passing we will define the class.

Create the BookTransformer Class

```
# vagrant@homestead:~/Code/bookr$
$ mkdir -p app/Transformer
$ touch app/Transformer/BookTransformer.php
```

Define the BookTransformer Class

```php
<?php

namespace App\Transformer;

use League\Fractal\TransformerAbstract;

class BookTransformer extends TransformerAbstract
{

}
```

Fractal transfomers extend the `League\Fractal\TransformerAbstract` class provided by Fractal. All your transformers should extend this class.

The Initializable Test Passes

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=BookTransformerTest

OK (1 test, 1 assertion)
```

Ok, now we are ready to write our next test describing how the `BookTransformer` should transform a `Book` model. Transforming model data will allow all controllers rendering book data to use a consistent transformation.

Test to Transform a Book Model

```
/** @test **/
public function it_transforms_a_book_model()
{
    $book = factory(Book::class)->create();
    $subject = new BookTransformer();

    $transform = $subject->transform($book);

    $this->assertArrayHasKey('id', $transform);
    $this->assertArrayHasKey('title', $transform);
    $this->assertArrayHasKey('description', $transform);
    $this->assertArrayHasKey('author', $transform);
    $this->assertArrayHasKey('created_at', $transform);
    $this->assertArrayHasKey('updated_at', $transform);
}
```

You can see the missing method we have not defined in the transformer—the `transform()` method. The test passes a model from the factory that exists in the database, and then asserts the model has all the keys we expect a book response to contain. The book must exist in the database because our transformer is transforming existing records, including the `id` attribute.

Testing the `BookTransformer@transform` Method

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=BookTransformerTest::it_transforms_a_book_model

There was 1 error:

1) Tests\App\Transformer\BookTransformerTest::it_transforms_a_book_model
Error: Call to undefined method App\Transformer\BookTransformer::transform()

/home/vagrant/Code/bookr/tests/app/Transformer/BookTransformerTest.php:28

FAILURES!
Tests: 1, Assertions: 0, Errors: 1.
```

We are ready to write the `BookTransformer::transform()` implementation. The `transform` method allows us to define a public data interface of the book model while the internal details are hidden. This means that our internal logic can change but the API consumer continues to get the same data interface.

Writing the `BookTransformer::transform()` method

```php
<?php

namespace App\Transformer;

use App\Book;
use League\Fractal\TransformerAbstract;

class BookTransformer extends TransformerAbstract
{
    /**
     * Transform a Book model into an array
     *
     * @param Book $book
     * @return array
     */
    public function transform(Book $book)
    {
        return [
            'id'          => $book->id,
            'title'       => $book->title,
            'description' => $book->description,
            'author'      => $book->author,
            'created'     => $book->created_at->toIso8601String(),
            'updated'     => $book->updated_at->toIso8601String(),
        ];
    }
}
```

Note the `created` and `updated` keys do not match our database column. What is really cool about using Fractal transformers is abstracting the API format from the database schema when it makes sense. In this case does it matter to change `created_at` into `created`? Probably not, but it serves as an illustration of how you can change the background implementation details and *continue to provide consistent data transformation* to the API user.

As an example, say that when a new book is created in the database we immediately offer it for sale in our online store. Imagine the application needs to display how long ago the title was released in a relative format:

Example Transformation

```
return [
    'id' => $book->id,
    'title' => $book->title,
    'description' => $book->description,
    'author' => $book->author,
    'created' => $book->created_at->toIso8601String(),
    'updated' => $book->updated_at->toIso8601String(),
    'released' => $book->created_at->diffForHumans()
]
```

In the previous example the `released` key is tied to `created_at` field and uses Carbon's[75] `diffForHumans()` method to provide a relative date. In the future we could change `released` to use a `published` datetime from the database and the implementation details would be invisible to consumers. Note that the `created_at` and `updated_at` properties in the example are also Carbon instances.

Date Mutators

Eloquent automatically provides a `Carbon` instance for `created_at` and `updated_at`, but you can configure your model to provide `Carbon` for other fields. See Date Mutators[a] in the Laravel documentation.

[a]http://laravel.com/docs/eloquent-mutators#date-mutators

Next, we need to test our implementation to see if it satisfies our test:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=BookTransformerTest::it_transforms_a_book_model

There was 1 failure:

1) Tests\App\Transformer\BookTransformerTest::it_transforms_a_book_model
Failed asserting that an array has the key 'created_at'.

/home/vagrant/Code/bookr/tests/app/Transformer/BookTransformerTest.php:34

FAILURES!
Tests: 1, Assertions: 5, Failures: 1.
```

[75]http://carbon.nesbot.com/

Woops! Our test was looking for `created_at` but we designed our transformer to use `created`. Lets fix the test to match and run the tests again:

Fix the BookTransformerTest to match `BookTransformer::transform()`

```
$this->assertArrayHasKey('created', $transform);
$this->assertArrayHasKey('updated', $transform);
```

Run PHPUnit Test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=BookTransformerTest::it_transforms_a_book_model

OK (1 test, 6 assertions)
```

The Fractal Response Class

Time to move on to our Fractal response class. If you look at the Fractal documentation (Simple Example[76]), you will see that we need a `League\Fractal\Manager` class. We could use the manager class directly in our controllers, but we will opt to provide a service for the Fractal manager.

We will call our service `App\Http\Response\FractalResponse` and start by writing some tests for it:

Setting up the FractalResponse Files

```
# vagrant@homestead:~/Code/bookr$
$ mkdir -p app/Http/Response
$ touch app/Http/Response/FractalResponse.php
$ mkdir -p tests/app/Http/Response
$ touch tests/app/Http/Response/FractalResponseTest.php
```

We write the class initialization test first:

[76] http://fractal.thephpleague.com/simple-example/

Test FractalResponse Initialization

```php
<?php

namespace Tests\App\Http\Response;

use TestCase;
use App\Http\Response\FractalResponse;

class FractalResponseTest extends TestCase
{
    /** @test **/
    public function it_can_be_initialized()
    {
        $this->assertInstanceOf(FractalResponse::class, new FractalResponse());
    }
}
```

Running the test will fail at this point. Here is the initial class skeleton to get it passing again:

Initial FractalResponse skeleton class

```php
<?php

namespace App\Http\Response;

class FractalResponse
{
    public function __construct()
    {

    }
}
```

The FractalResponse constructor will have a few dependencies: a `League\Fractal\Manager` instance and a DataArraySerializer[77] instance. You can configure a Fractal manager to use a Serializer. Serializers provide structure around your data without affecting transformers—in our case that means wrapping our transformer output in a `"data"` property using the `DataArraySerializer`. The manager sets the serializer by calling `setSerializer()` and passing a serializer instance. Serializers extend from the `League\Fractal\Serializer\SerializerAbstract` class so we can even write our unit tests against the abstract implementation!

[77]http://fractal.thephpleague.com/serializers/#dataarrayserializer

Testing the FractalResponse with a Manager dependency

```php
<?php

namespace Tests\App\Http\Response;

use TestCase;
use Mockery as m;
use League\Fractal\Manager;
use App\Http\Response\FractalResponse;
use League\Fractal\Serializer\SerializerAbstract;

class FractalResponseTest extends TestCase
{
    /** @test **/
    public function it_can_be_initialized()
    {
        $manager = m::mock(Manager::class);
        $serializer = m::mock(SerializerAbstract::class);

        $manager
            ->shouldReceive('setSerializer')
            ->with($serializer)
            ->once()
            ->andReturn($manager);

        $fractal = new FractalResponse($manager, $serializer);
        $this->assertInstanceOf(FractalResponse::class, $fractal);
    }
}
```

The whole class is provided for clarity. Note the use of Mockery's `shouldReceive`, which sets an expectation that `FractalResponse` calls `setSerializer` once. The test mocks `SerializerAbstract` and expects that the manager will receive a call to `setSerializer` with the mock. The last part of test asserts the `FractalResponse` class was successfully initialized.

Test Initialization of FractalResponse

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 error:

1) Tests\App\Http\Response\FractalResponseTest::it_can_be_initialized
Mockery\Exception\InvalidCountException: Method setSerializer(object(Mockery_7_L\
eague_Fractal_Serializer_SerializerAbstract)) from Mockery_6_League_Fractal_Mana\
ger should be called
 exactly 1 times but called 0 times.

/home/vagrant/Code/bookr/vendor/mockery/mockery/library/Mockery/CountValidator/E\
xact.php:37
/home/vagrant/Code/bookr/vendor/mockery/mockery/library/Mockery/Expectation.php:\
271
/home/vagrant/Code/bookr/vendor/mockery/mockery/library/Mockery/ExpectationDirec\
tor.php:120
/home/vagrant/Code/bookr/vendor/mockery/mockery/library/Mockery/Container.php:297
/home/vagrant/Code/bookr/vendor/mockery/mockery/library/Mockery/Container.php:282
/home/vagrant/Code/bookr/vendor/mockery/mockery/library/Mockery.php:142
/home/vagrant/Code/bookr/vendor/mockery/mockery/library/Mockery/Adapter/Phpunit/\
MockeryPHPUnitIntegration.php:24

FAILURES!
Tests: 19, Assertions: 75, Errors: 1.
```

You will notice that Mockery expected `setSerializer()` to be called once, but we haven't introduced the manager and serializer parameters in the constructor. Now we are ready to write out our implementation:

FractalResponse defines the manager and serializer dependencies

```php
<?php

namespace App\Http\Response;

use League\Fractal\Manager;
use League\Fractal\Serializer\SerializerAbstract;

class FractalResponse
{
    /**
     * @var Manager
     */
    private $manager;

    /**
     * @var SerializerAbstract
     */
    private $serializer;

    public function __construct(Manager $manager, SerializerAbstract $serializer)
    {
        $this->manager = $manager;
        $this->serializer = $serializer;
        $this->manager->setSerializer($serializer);
    }
}
```

Our constructor type hints `SerializerAbstract` which allows us to accept any instance that extends the abstract class so we can swap it out whenever we wish.

The `FractalResponse` class is not very useful right now, we are ready to add a few methods that controllers can use to create response data. We know based on our `BooksController` that we need to support serializing collections and individual records. We will define the following public methods: `FractalResponse::item()` and `FractalResponse::collection` which will use Resources[78] to transform the data and convert it into an array.

The following is an example of how we will use our service:

Example FractalResponse Method Usage

```
// BooksController::show()
return $this->fractal->item($book, new BookTransformer());

// BooksController::index()
return $this->fractal->collection($books, new BookTransformer());
```

We will start with the `FractalResponse::item()` method first. The basic usage of Fractal to return an item might look like this:

[78] http://fractal.thephpleague.com/resources/

Example Usage of Fractal Manager Item

```
$data = ['bar' => 'bar'];
$manager = new \League\Fractal\Manager();
$item = new \League\Fractal\Resource\Item($data, function (array $data) {
    return [
        'foo' => $data['bar']
    ];
});

$manager->createData($item)->toArray();
```

In the code example above we create a `Manager` instance and an `Item` resource. The manager calls `createData()` which returns an instance of `League\Fractal\Scope` and then we chain a `toArray()` call on the scope. This simple example illustrates using a `Closure` but we will pass a transformer object in our implementation.

Test for the `FractalResponse::item()` method.

```
/** @test **/
public function it_can_transform_an_item()
{
    // Transformer
    $transformer = m::mock('League\Fractal\TransformerAbstract');

    // Scope
    $scope = m::mock('League\Fractal\Scope');
    $scope
        ->shouldReceive('toArray')
        ->once()
        ->andReturn(['foo' => 'bar']);

    // Serializer
    $serializer = m::mock('League\Fractal\Serializer\SerializerAbstract');

    $manager = m::mock('League\Fractal\Manager');
    $manager
        ->shouldReceive('setSerializer')
        ->with($serializer)
        ->once();

    $manager
        ->shouldReceive('createData')
```

```
            ->once()
            ->andReturn($scope);

        $subject = new FractalResponse($manager, $serializer);
        $this->assertInternalType(
            'array',
            $subject->item(['foo' => 'bar'], $transformer)
        );
    }
```

This is a pretty long test, so we will break it down:

- We mock a transformer which the `item` method will accept (ie. `BookTransformer`)
- We mock a `Scope` object which will be returned from the `Manager::createData()` method call
- We mock the serializer to initialize our class
- We mock the manager class which should receive a call to `createdData` and `setSerializer`
- Finally, we initialize the class under test and assert that the `item` method returns the internal type `array`.

I must say, its pretty neat when tests help design how we write our implementation. Now that we have a failing test (you can run it on your own) we are ready to write some code.

The FractalResponse::item() Implementation

```php
<?php

namespace App\Http\Response;

use League\Fractal\Manager;
use League\Fractal\Resource\Item;
use League\Fractal\TransformerAbstract;
use League\Fractal\Serializer\SerializerAbstract;

class FractalResponse
{
    /**
     * @var Manager
     */
    private $manager;

    /**
```

```
     * @var SerializerAbstract
     */
    private $serializer;

    public function __construct(Manager $manager, SerializerAbstract $serializer)
    {
        $this->manager = $manager;
        $this->serializer = $serializer;
        $this->manager->setSerializer($serializer);
    }

    public function item($data, TransformerAbstract $transformer, $resourceKey =\
 null)
    {
        $resource = new Item($data, $transformer, $resourceKey);

        return $this->manager->createData($resource)->toArray();
    }
}
```

The `item()` method accepts the data to be transformed, a transformer (ie. BookTransformer), and a resource key. We will not be using the resource key, but we want to support the full `League\Fractal\Resource\Item` API. We create a new `League\Fractal\Resource\Item` instance and pass it to the manager's `createData` method. The `createData` method returns a `League\Fractal\Scope` instance and we call `toArray` to return array data from the `League\Fractal\Scope` instance.

You can read about resource keys in the official Fractal serializers[79] documentation

We are ready to run the tests after our first stab at the implementation:

Testing the `FractalResponse::item()` implementation

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (20 tests, 79 assertions)
```

We can move on to our `collection` method test. The `FractalResponse::collection()` method is very similar to the `item` method, so we will not spend much time on it. We will be passing the `collection` method an `Illuminate\Database\Eloquent\Collection` to iterate over the collection and transform each item. Here is the test for the `collection` method:

[79] http://fractal.thephpleague.com/serializers/

Initial test for the `FractalResponse::collection()` method

```php
/** @test **/
public function it_can_transform_a_collection()
{
    $data = [
        ['foo' => 'bar'],
        ['fizz' => 'buzz'],
    ];

    // Transformer
    $transformer = m::mock('League\Fractal\TransformerAbstract');

    // Scope
    $scope = m::mock('League\Fractal\Scope');
    $scope
        ->shouldReceive('toArray')
        ->once()
        ->andReturn($data);

    // Serializer
    $serializer = m::mock('League\Fractal\Serializer\SerializerAbstract');

    $manager = m::mock('League\Fractal\Manager');
    $manager
        ->shouldReceive('setSerializer')
        ->with($serializer)
        ->once();

    $manager
        ->shouldReceive('createData')
        ->once()
        ->andReturn($scope);

    $subject = new FractalResponse($manager, $serializer);
    $this->assertInternalType(
        'array',
        $subject->collection($data, $transformer)
    );
}
```

This test is nearly identical to our `FractalResponseTest::it_can_transform_an_item()` test. The only difference is that we are calling the `collection` method instead. We could merge the `item` and `collection` tests into one, but we will leave them separate for clarity.

Here is the implemenation for `FractalResponse::collection()`:

The `FractalResponse::collection()` implementation

```php
<?php

namespace App\Http\Response;

use League\Fractal\Manager;
use League\Fractal\Resource\Item;
use League\Fractal\TransformerAbstract;
use League\Fractal\Resource\Collection;
use League\Fractal\Serializer\SerializerAbstract;

class FractalResponse
{
    /**
     * @var Manager
     */
    private $manager;

    /**
     * @var SerializerAbstract
     */
    private $serializer;

    public function __construct(Manager $manager, SerializerAbstract $serializer)
    {
        $this->manager = $manager;
        $this->serializer = $serializer;
        $this->manager->setSerializer($serializer);
    }

    public function item($data, TransformerAbstract $transformer, $resourceKey =\
 null)
    {
        $resource = new Item($data, $transformer, $resourceKey);

        return $this->manager->createData($resource)->toArray();
    }

    public function collection($data, TransformerAbstract $transformer, $resourc\
eKey = null)
    {
        $resource = new Collection($data, $transformer, $resourceKey);

        return $this->manager->createData($resource)->toArray();
    }
}
```

The `collection` endpoint initializes a `League\Fractal\Resource\Collection` instance, but everything else is exactly the same. Both the `item` and `collection` methods duplicate creating the data and converting it to an array. All tests should be passing, so we are now able to refactor out the duplication.

Refactor `FractalResponse::item()` and `FractalResponse::collection()`

```
<?php

namespace App\Http\Response;

use League\Fractal\Manager;
use League\Fractal\Resource\Item;
use League\Fractal\TransformerAbstract;
use League\Fractal\Resource\Collection;
use League\Fractal\Resource\ResourceInterface;
use League\Fractal\Serializer\SerializerAbstract;

class FractalResponse
{
    /**
     * @var Manager
     */
    private $manager;

    /**
     * @var SerializerAbstract
     */
    private $serializer;

    public function __construct(Manager $manager, SerializerAbstract $serializer)
    {
        $this->manager = $manager;
        $this->serializer = $serializer;
        $this->manager->setSerializer($serializer);
    }

    public function item($data, TransformerAbstract $transformer, $resourceKey =\
 null)
    {
        return $this->createDataArray(
            new Item($data, $transformer, $resourceKey)
        );
    }

    public function collection($data, TransformerAbstract $transformer, $resourc\
eKey = null)
    {
        return $this->createDataArray(
            new Collection($data, $transformer, $resourceKey)
        );
    }

    private function createDataArray(ResourceInterface $resource)
    {
        return $this->manager->createData($resource)->toArray();
    }
}
```

Make Sure Tests Still Pass After the Refactor

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (21 tests, 83 assertions)
```

We didn't do anything groundbreaking, but we consolidated duplicate code while using our passing tests to refactor. Refactoring is an important part of the test-driven process because the goal is to get tests passing with minimal code, and then improve the minimal code while keeping tests passing.

Git commit: Add FractalResponse Class

a6cc22d[80]

[80]https://bitbucket.org/paulredmond/bookr/commits/a6cc22d

7.3: FractalResponse Service

The `FractalResponse` class is ready to be used in our `BooksController`. What is really cool is that we wrote the implementation in isolation, but tests provide a good degree of confidence that it will work. Writing it in isolation also has the benefit of helping us resit the temptation to start integrating it before it is fully tested.

How should we go about integrating the `FractalResponse` class in our controllers?

One option would be to create an instance of `FractalResponse` in the controller:

```php
public function __construct() {
    // ...
    $this->fractal = new FractalResponse($manager, $serializer);
}
```

Constructing a new instance of the `FractalResponse` in the controller would mean that we would also need to create an instance of the manager and serializer and pass them to the `FractalResponse` constructor. Doing it by hand each time we need it sounds like a bad time.

A facade would work, and that is definitely a viable option:

```php
// An individual book
return FractalResponse::item($book, new BookTransformer());

// A collection of books
return FractalResponse::collection($books, new BookTransformer());
```

In this book we will define a Service Provider[81] to resolve a fully initialized `FractalResponse` instance anywhere we need it. Service providers are used to bootstrap the application by registering service container bindings (among other things) which can be resolved from the service container. This means that we need to define a service for the `FractalResponse` class that we can resolve out of the container.

In Lumen, providers are configured in the `bootstrap/app.php` file. At the time of writing this is what the section looks like by default:

[81]http://lumen.laravel.com/docs/providers

Boilerplate Service Provider Configuration

```
/*
|--------------------------------------------------------------------------
| Register Service Providers
|--------------------------------------------------------------------------
|
| Here we will register all of the application's service providers which
| are used to bind services into the container. Service providers are
| totally optional, so you are not required to uncomment this line.
|
*/

// $app->register(App\Providers\AppServiceProvider::class);
// $app->register(App\Providers\EventServiceProvider::class);
```

As you can see the framework provides an `AppServiceProvider` class that we could use to register application services. We will create our own service provider class because you will benefit from learning how to create one.

Create the FractalServiceProvider class

```
# vagrant@homestead:~/Code/bookr$
$ touch app/Providers/FractalServiceProvider.php
```

Add the following code to the newly created file:

A Skeleton FractalServiceProvider Class

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class FractalServiceProvider extends ServiceProvider
{
    public function register()
    {

    }

    public function boot()
    {

    }
}
```

Service providers can be defined anywhere you want, however, use the convention provided in lumen: under the `App\Providers` namespace. Service providers must define a `register` method. The sole purpose of the `register` method is to bind things to the service container.

The `boot` method is called after all other services have been registered, and is optional. One example of using the `boot` method could be loading configuration that your provider needs. We don't need to use `boot` in the `FractalServiceProvider`, but it was included for awareness. We will not delve into the full ServiceProvider API; read the documentation[82] and browse the source code to learn more.

With a bit of background behind us, let's register our service:

Define the `FractalResponse` Service

```php
<?php

namespace App\Providers;

use League\Fractal\Manager;
use App\Http\Response\FractalResponse;
use Illuminate\Support\ServiceProvider;
use League\Fractal\Serializer\DataArraySerializer;

class FractalServiceProvider extends ServiceProvider
{
    public function register()
    {
        // Bind the DataArraySerializer to an interface contract
        $this->app->bind(
            'League\Fractal\Serializer\SerializerAbstract',
            'League\Fractal\Serializer\DataArraySerializer'
        );

        $this->app->bind(FractalResponse::class, function ($app) {
            $manager = new Manager();
            $serializer = $app['League\Fractal\Serializer\SerializerAbstract'];

```

[82] http://lumen.laravel.com/docs/providers

```
            return new FractalResponse($manager, $serializer);
        });

        $this->app->alias(FractalResponse::class, 'fractal');
    }
}
```

The first thing the `register` method does is bind the `DataArraySerializer` implementation to the abstract serializer. When we type hint `League\Fractal\Serializer\SerializerAbstract` the container will resolve `League\Fractal\Serializer\DataArraySerializer` for us. Binding is powerful because when we code to an interface we can change the implementation in the provider and classes consuming the service will not know the difference.

Secondly we define our `FractalResponse` service. The `$this->app->bind()` call accepts the abstract name, and the second argument can be a concrete implementation (like our DataArraySerializer class) or a `Closure`. When you pass a `Closure` it accepts our application instance (`$app`) as an argument. In the closure we construct a `League\Fractal\Manager` instance and reference the `SerializerAbstract` from the container which was the contract we bound to the `DataArraySerializer`. Lastly, we create and return the `FractalResponse` instance.

The last thing we do in the `register` method is call `$this->app->alias()` which will alias our service to a shorter name when we ask for it from the container. The alias is a convenient short name to the service but is not required.

We can now access our service in a few ways.

Different Ways to Resolve the `FractalResponse` Service

```
$fractal = app('App\Http\Response\FractalResponse');
$fractal = \App::make('App\Http\Response\FractalResponse');

// Using the alias we defined:
$fractal = app('fractal');

// Or via constructor hinting for the classes the container resolves
class MyController extends Controller
{
    public function __construct(App\Http\Response\FractalResponse $fractal)
    {
        $this->fractal = $fractal;
    }
}
```

For our particular use case we will use the constructor to easily provide all of our controllers with the `FractalResponse` service.

The next step in finishing the service provider is registering the provider in our application. Open the `bootstrap/app.php` file and add the following:

Register the FractalServiceProvider

```
// $app->register(App\Providers\AppServiceProvider::class);
// $app->register(App\Providers\EventServiceProvider::class);

$app->register(\App\Providers\FractalServiceProvider::class);
```

You can now resolve the `FractalResponse` class from the service container, and we are ready to integrate the service into our `BooksController` class in the next section. Onward!

7.4: Integrating the FractalResponse Service

Our final task is integrating the `FractalResponse` service. We could make each controller resolve the service out of the container, but we want to provide consistency across all our responses. The `App\Http\Controllers\Controller` class sounds like a good place to resolve the service because all controllers extend from it. We will edit `app/Http/Controllers/Controller.php` and add the following code that our controllers can use:

Integrate the `FractalResponse` Service in the Base Controller

```
<?php

namespace App\Http\Controllers;

use App\Http\Response\FractalResponse;
use Laravel\Lumen\Routing\Controller as BaseController;
use League\Fractal\TransformerAbstract;

class Controller extends BaseController
{
    /**
     * @var FractalResponse
     */
    private $fractal;

    public function __construct(FractalResponse $fractal)
    {
        $this->fractal = $fractal;
    }

    /**
     * @param $data
     * @param TransformerAbstract $transformer
     * @param null $resourceKey
     * @return array
     */
    public function item($data, TransformerAbstract $transformer, $resourceKey =\
 null)
    {
        return $this->fractal->item($data, $transformer, $resourceKey);
    }

    /**
     * @param $data
     * @param TransformerAbstract $transformer
     * @param null $resourceKey
     * @return array
     */
    public function collection($data, TransformerAbstract $transformer, $resourc\
eKey = null)
    {
        return $this->fractal->collection($data, $transformer, $resourceKey);
    }
}
```

The constructor method type hints the `FractalResponse` argument which the service container will resolve automatically when initializing the controller. The `item` and `collection` methods are pass-through methods to the `FractalResponse` service. We could make the methods more generic like `respondWithItem` and take care of the response, but what we have is good enough for now. We can refactor when we sense duplication and brittle code.

Let's try our first integration with the `BooksController`. Be sure to add "`use App\Transformer\BookTransformer;`" to the top of `app/Http/Controllers/BooksController.php`.

Use the FractalResponse Service in the BooksController

```php
<?php

namespace App\Http\Controllers;

use App\Book;
use Illuminate\Http\Request;
use App\Transformer\BookTransformer;
use Illuminate\Database\Eloquent\ModelNotFoundException;

/**
 * Class BooksController
 * @package App\Http\Controllers
 */
class BooksController extends Controller
{
    /**
     * GET /books
     * @return array
     */
    public function index()
    {
        return $this->collection(Book::all(), new BookTransformer());
    }
    // ...
}
```

Importantly, the `BooksController` is now extending the base `Controller` class found in the same folder. Don't forget to extend this controller now, because we need it to inherit the `collection` and `item` methods we just wrote! The `BooksController@index` method calls the `App\Http\Controller::collection()` method and passes the Eloquent collection and the `BookTransformer` as parameters. The method returns an `array` which will resemble the following JSON response:

```
{
    "data": [
        {
            "author": "H. G. Wells",
            "created": "2015-10-21T06:54:33+0000",
            "description": "A science fiction masterpiece about Martians invadin\
g London",
            "id": 1,
            "title": "War of the Worlds",
            "updated": "2015-10-21T06:54:33+0000"
        },
        {
            "author": "Madeleine L'Engle",
            "created": "2015-10-21T06:54:33+0000",
            "description": "A young girl goes on a mission to save her father wh\
o has gone missing after working on a mysterious project called a tesseract.",
            "id": 2,
            "title": "A Wrinkle in Time",
            "updated": "2015-10-21T06:54:33+0000"
        }
    ]
}
```

Let's run tests to see if our integration is working:

Test Failure After Integrating Fractal

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::index_should_return_a_collect\
ion_of_records
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'{"data":[{"author":"Asa Hintz","created_at":"2016-01-13 05:45:25"...
+'{"data":[{"author":"Asa Hintz","created":"2016-01-13T05:45:25+0000"...

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:338
```

```
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:28

FAILURES!
Tests: 21, Assertions: 83, Failures: 1.
```

The test error looks like our response testing doesn't match the controller. If you recall, the `BooksControllerTest` tests were updated earlier when we added the `data` key:

```php
/** @test **/
public function index_should_return_a_collection_of_records()
{
    $books = factory('App\Book', 2)->create();

    $this->get('/books');

    $expected = [
        'data' => $books->toArray()
    ];

    $this->seeJsonEquals($expected);
}
```

The `$books->toArray()` method will do the equivalent of `(string) $book->created_at` to the date, but our new transformer is using `$book->created_at->toIso8601String()`. The transformer is also defining a "created" property but our test is expecting a "created_at" property. Lets fix the test to pass again:

Change the Test to Assert the `BookTransformer` Correctly

```php
/** @test **/
public function index_should_return_a_collection_of_records()
{
    $books = factory('App\Book', 2)->create();

    $this->get('/books');

    $content = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $content);

    foreach ($books as $book) {
        $this->seeJson([
            'id' => $book->id,
            'title' => $book->title,
            'description' => $book->description,
            'author' => $book->author,
            'created' => $book->created_at->toIso8601String(),
            'updated' => $book->updated_at->toIso8601String(),
        ]);
    }
}
```

After getting the `/books` response, we check for the `data` key and then loop through the factory models to ensure they are all in the response correctly. With our change in place, we will see if the tests pass now.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (21 tests, 97 assertions)
```

The `BooksController@show` method is next:

Updating `BooksController@show` to Use Fractal

```
/**
 * GET /books/{id}
 * @param integer $id
 * @return mixed
 */
public function show($id)
{
    return $this->item(Book::findOrFail($id), new BookTransformer());
}
```

The `@show` method is using the inherited `Controller::item()` method we built earlier. We are refactoring at this point so we should expect our tests to pass. If not, we need to fix them before moving on:

Running the Test Suite after Refactoring `BooksController@show`

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::show_should_return_a_valid_bo\
ok
Failed asserting that two strings are equal.
--- Expected
+++ Actual
@@ @@
-'{"data":{"author":"Miss Pasquale Jaskolski DDS","created_at":"2016-01-13 05:52\
:25"...
+'{"data":{"author":"Miss Pasquale Jaskolski DDS","created":"2016-01-13T05:52:25\
+0000",...

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:338
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:50

FAILURES!
Tests: 21, Assertions: 97, Failures: 1.
```

Darn! If you have a keen eye, you can see that we have the same issue in the `BooksController@show` test that we just fixed for the `@index` test.

Updating the `BooksController@show` Test

```
/** @test **/
public function show_should_return_a_valid_book()
{
    $book = factory('App\Book')->create();

    $this
        ->get("/books/{$book->id}")
        ->seeStatusCode(200);

    // Get the response and assert the data key exists
    $content = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $content);
    $data = $content['data'];

    // Assert the Book Properties match
    $this->assertEquals($book->id, $data['id']);
    $this->assertEquals($book->title, $data['title']);
    $this->assertEquals($book->description, $data['description']);
    $this->assertEquals($book->author, $data['author']);
    $this->assertEquals($book->created_at->toIso8601String(), $data['created']);
    $this->assertEquals($book->updated_at->toIso8601String(), $data['created']);
}
```

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (21 tests, 103 assertions)
```

Our next refactor is the `BooksController@store` method:

Refactor `BooksController@store` to Use Fractal

```
/**
 * POST /books
 * @param Request $request
 * @return \Symfony\Component\HttpFoundation\Response
 */
public function store(Request $request)
{
    $book = Book::create($request->all());
    $data = $this->item($book, new BookTransformer());

    return response()->json($data, 201, [
        'Location' => route('books.show', ['id' => $book->id])
    ]);
}
```

Nothing new in the example that we haven’t seen before, let’s run the tests:

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (21 tests, 103 assertions)
```

Our test passed, but we should review it to make sure it still covers everything. If you look at the "`store_should_save_new_book_in_the_database`" test, notice that it doesn't check for the `created` or `updated` field. You can either provide extra assertions to check, or you can rely on the `BookTransformer::it_transforms_a_book_model()` test which ensures this works correctly. Our test is an integration test, so we will choose to add the additional checks. We cannot guarantee at this point that the `BookTransformer` is being used.

We have one problem though: if we assert the value of `created` and `updated` how do we know what to expect? These fields are populated dynamically when the book is created. Lucky for us, Lumen uses the Carbon[83] library to work with dates in Eloquent. If you browse the Carbon documentation or the API you will find the `Carbon::setTestNow()` method; we need to mock the time so we know what to expect! Mocking `Carbon` makes testing dates really easy!

In preparation to updating our test, we need to add a `setUp` and `tearDown` method so we can test dates in each test and then reset Carbon afterwards:

Mocking Carbon in the `BooksControllerTest` Class (partial source)

```php
<?php

namespace Tests\App\Http\Controllers;

use TestCase;
use Carbon\Carbon;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class BooksControllerTest extends TestCase
{
    use DatabaseMigrations;

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

        Carbon::setTestNow(Carbon::now('UTC'));
```

[83]http://carbon.nesbot.com/

```
    }

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

        Carbon::setTestNow();
    }
    // ...
}
```

At the top of our `BooksControllerTest` we import Carbon with `use Carbon\Carbon;`. We override the `setUp` and `tearDown` methods[84] to mock Carbon.

Calling `Carbon::setTestNow(Carbon::now('UTC'));` will mock our tests to use the current time. Once the test is over we call `Carbon::setTestNow();` in the `tearDown` method to reset Carbon to its normal behavior.

Now that we've mocked Carbon we can write our test expectations:

Update the `BooksController@store` test to assert `created` and `updated` values.

```
/** @test **/
public function store_should_save_new_book_in_the_database()
{
    $this->post('/books', [
        'title' => 'The Invisible Man',
        'description' => 'An invisible man is trapped in the terror of his own c\
reation',
        'author' => 'H. G. Wells'
    ]);

    $body = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $body);

    $data = $body['data'];
    $this->assertEquals('The Invisible Man', $data['title']);
    $this->assertEquals(
        'An invisible man is trapped in the terror of his own creation',
        $data['description']
    );
    $this->assertEquals('H. G. Wells', $data['author']);
```

[84] https://phpunit.de/manual/current/en/fixtures.html

```
    $this->assertTrue($data['id'] > 0, 'Expected a positive integer, but did not\
 see one.');

    $this->assertArrayHasKey('created', $data);
    $this->assertEquals(Carbon::now()->toIso8601String(), $data['created']);
    $this->assertArrayHasKey('updated', $data);
    $this->assertEquals(Carbon::now()->toIso8601String(), $data['updated']);

    $this->seeInDatabase('books', ['title' => 'The Invisible Man']);
}
```

Near the end of the test we assert that the response has the `created` and `updated` keys, and we use carbon to make sure the dates are in ISO 8601[85] format.

Run the Entire Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (21 tests, 107 assertions)
```

And our tests all pass! We are on the final method refactor and then we done. The `BooksController@update` method will be similar to the `store` method:

Refactor `BooksController@update`

```
/**
 * PUT /books/{id}
 * @param Request $request
 * @param $id
 * @return mixed
 */
public function update(Request $request, $id)
{
    try {
        $book = Book::findOrFail($id);
    } catch (ModelNotFoundException $e) {
        return response()->json([
            'error' => [
                'message' => 'Book not found'
            ]
```

[85]http://www.iso.org/iso/home/standards/iso8601.htm

```
        ], 404);
    }

    $book->fill($request->all());
    $book->save();

    return $this->item($book, new BookTransformer());
}
```

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (21 tests, 107 assertions)
```

And we are still green!

Next, we will add response checks and assert the dates in the "`update_should_only_change_fillable_fields`" test:

Adding Additional Checks for BooksController@update

```
/** @test **/
public function update_should_only_change_fillable_fields()
{
    $book = factory('App\Book')->create([
        'title' => 'War of the Worlds',
        'description' => 'A science fiction masterpiece about Martians invading \
London',
        'author' => 'H. G. Wells',
    ]);

    $this->notSeeInDatabase('books', [
        'title' => 'The War of the Worlds',
        'description' => 'The book is way better than the movie.',
        'author' => 'Wells, H. G.'
    ]);

    $this->put("/books/{$book->id}", [
        'id' => 5,
        'title' => 'The War of the Worlds',
        'description' => 'The book is way better than the movie.',
        'author' => 'Wells, H. G.'
    ]);

    $this
        ->seeStatusCode(200)
        ->seeJson([
            'id' => 1,
            'title' => 'The War of the Worlds',
            'description' => 'The book is way better than the movie.',
            'author' => 'Wells, H. G.',
        ])
        ->seeInDatabase('books', [
            'title' => 'The War of the Worlds'
        ]);

    $body = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $body);

    $data = $body['data'];
    $this->assertArrayHasKey('created', $data);
    $this->assertEquals(Carbon::now()->toIso8601String(), $data['created']);
    $this->assertArrayHasKey('updated', $data);
    $this->assertEquals(Carbon::now()->toIso8601String(), $data['updated']);
}
```

At the end of the test method we added assertions for the `created` and `updated` values just like our last test.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (21 tests, 111 assertions)
```

Git commit: Integrate the FractalResponse Service

8ddd2ef[86]

[86]https://bitbucket.org/paulredmond/bookr/commits/8ddd2ef

Conclusion

We've had a long journey in this chapter, but now we have a decent way to generate consistent responses in our app. Along the way you've learned:

- Writing the minimum amount of code to get tests passing
- Refactor *only* after getting tests passing
- How to create and install a service provider
- How to use the service container
- The basics of the Fractal[87] library

At this point I'd encourage you to read more about the service container[88] and service providers[89]. The container and providers are related concepts that you will use frequently when you develop Lumen applications.

[87]http://fractal.thephpleague.com/
[88]http://lumen.laravel.com/docs/container
[89]http://lumen.laravel.com/docs/providers

Chapter 8: Validation

Our `/book` API is going smoothly up to this point, but our tests and controllers assume that *good* data is being submitted. Our application doesn't protect against bad data (and empty data) being saved to the database; specifically, the `BooksController@store` and `BooksController@update` methods happily save bad data.

We will focus our efforts on validating data submitted to the `/books` API using the validation[90] tools provided by Lumen. We will write tests for our validation logic, providing you with a good foundation of how write validation tests in your own applications.

8.1: First Attempt at Validation

We will start out with basic validation and then iteratively add features as we go. Like always, we create tests first.

What validation rules do we need? Validation is a combination of business logic and security practices. In our books API all books should be required to have a title. For our purposes the database schema doesn't allow null values and we intend to make the `title`, `description`, and `author` fields required. If these fields are present, we will consider them valid.

Let's start a new integration test class with a few basic tests for validation:

Creating the Test Class

```
# vagrant@homestead:~/Code/bookr$
$ touch tests/app/Http/Controllers/BooksControllerValidationTest.php
```

The Skeleton BooksControllerValidationTest Class

```php
<?php

namespace Tests\App\Http\Controllers;

use TestCase;
use Illuminate\Http\Response;
use Illuminate\Foundation\Testing\DatabaseMigrations;

```

[90] https://lumen.laravel.com/docs/5.1/validation

```php
class BooksControllerValidationTest extends TestCase
{
    use DatabaseMigrations;

    /** @test **/
    public function it_validates_required_fields_when_creating_a_new_book()
    {

    }

    /** @test **/
    public function it_validates_requied_fields_when_updating_a_book()
    {

    }
}
```

Take note of the imported `Illuminate\Http\Response` class that we will use to test expected response codes. We also import the `DatabaseMigrations` trait to test existing database records and insert new ones.

Writing Validation Assertions for the BooksController@store Method.

```php
/** @test **/
public function it_validates_required_fields_when_creating_a_new_book()
{
    $this->post('/books', [], ['Accept' => 'application/json']);

    $this->assertEquals(Response::HTTP_UNPROCESSABLE_ENTITY, $this->response->ge\
tStatusCode());

    $body = json_decode($this->response->getContent(), true);

    $this->assertArrayHasKey('title', $body);
    $this->assertArrayHasKey('description', $body);
    $this->assertArrayHasKey('author', $body);

    $this->assertEquals(["The title field is required."], $body['title']);
    $this->assertEquals(["The description field is required."], $body['descripti\
on']);
    $this->assertEquals(["The author field is required."], $body['author']);
}
```

Our first test tries to create a new book without sending any data; we expect to receive a `422 Unprocessable Entity` status code from the failed validation. Next we decode the JSON response and assert that certain validation errors are present in the response. The validator formats error messages into an array for each field.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=it_validates_required_fields_when_creating_a_new_book

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerValidationTest::it_validates_requir\
ed_fields_when_creating_a_new_book
Failed asserting that 201 matches expected 422.

bookr/tests/app/Http/Controllers/BooksControllerValidationTest.php:18

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

The API responds with a `201 Created` status code which we expect when valid data is submitted, but we expect failure because the data submitted is invalid. With the failed test we are ready to integrate our validation logic in the controller:

The Validation Implementation for Creating a Book

```
/**
 * POST /books
 *
 * @param Request $request
 * @return \Illuminate\Http\JsonResponse
 */
public function store(Request $request)
{
    $this->validate($request, [
        'title' => 'required',
        'description' => 'required',
        'author' => 'required'
    ]);

    $book = Book::create($request->all());
    $data = $this->item($book, new BookTransformer());

    return response()->json($data, 201, [
        'Location' => route('books.show', ['id' => $book->id])
    ]);
}
```

The first line of the controller calls the `validate()` method which accepts an `Illuminate\Http\Request` instance and an array of validation rules mapped to the request data. We validate the request using the `required` rule which means the field must be present. If the `validate()` method fails it will throw and `HttpResponseException` with the `422 Unprocessable Entity` status code, which is what our test expects.

The ValidatesRequest Trait

The `validate()` method comes from the `Laravel\Lumen\Routing\ValidatesRequest` trait provided by the base `Laravel\Lumen\Routing\Controller` class.

Read the validation documentation[a] to find a complete list of all the validation rules Lumen supports out of the box, as well as how to create your own custom rules.

[a] https://lumen.laravel.com/docs/5.1/validation#available-validation-rules

Next, we test our implementation to see if it succeeds:

Test the BooksController@store Validation Implementation

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=it_validates_required_fields_when_creating_a_new_book

OK (1 test, 7 assertions)
```

We will add the same basic validation to the `BooksController@update` method before doing some refactoring and improvements:

Writing the Validation Test for the `BooksController@update` Method

```
/** @test **/
public function it_validates_validates_passed_fields_when_updating_a_book()
{
    $book = factory(\App\Book::class)->create();

    $this->put("/books/{$book->id}", [], ['Accept' => 'application/json']);

    $this->assertEquals(Response::HTTP_UNPROCESSABLE_ENTITY, $this->response->ge\
tStatusCode());

    $body = json_decode($this->response->getContent(), true);

    $this->assertArrayHasKey('title', $body);
    $this->assertArrayHasKey('description', $body);
    $this->assertArrayHasKey('author', $body);

    $this->assertEquals(["The title field is required."], $body['title']);
    $this->assertEquals(["The description field is required."], $body['descripti\
on']);
    $this->assertEquals(["The author field is required."], $body['author']);
}
```

This test creates a valid record in the database and attempts a `PUT` request with no data. Running the tests will give an error similar to the first validation test:

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerValidationTest::it_validates_valida\
tes_passed_fields_when_updating_a_book
Failed asserting that 200 matches expected 422.

/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerValidationTes\
t.php:38

FAILURES!
Tests: 23, Assertions: 119, Failures: 1.
```

The `BooksController@update` implementation is identical to the `BooksController@create` method:

Implement Validation in the `BooksController@update` Method

```
/**
 * PUT /books/{id}
 *
 * @param Request $request
 * @param $id
 * @return mixed
 */
public function update(Request $request, $id)
{
    try {
        $book = Book::findOrFail($id);
    } catch (ModelNotFoundException $e) {
        return response()->json([
            'error' => [
                'message' => 'Book not found'
            ]
        ], 404);
    }

    $this->validate($request, [
        'title' => 'required',
        'description' => 'required',
        'author' => 'required'
    ]);

    $book->fill($request->all());
    $book->save();

    return $this->item($book, new BookTransformer());
}
```

Running the BooksController@update Validation Test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (23 tests, 125 assertions)
```

With passing tests we've provided basic validation for books. Our validation code is ready to be expanded to include more rules and tests to ensure our validation continues to work as expected.

8.2: More Validation Constraints

The second validation rule is limiting the length of the book title. The database is constrained to 255 characters so we will write our application to match. We will test a few scenarios, including just long enough and just too long. Add the following tests to the `BooksControllerValidationTest` file:

Testing Title is 'too long' Scenario

```
/** @test **/
public function title_fails_create_validation_when_just_too_long()
{
    // Creating a book
    $book = factory(\App\Book::class)->make();
    $book->title = str_repeat('a', 256);

    $this->post("/books", [
        'title' => $book->title,
        'description' => $book->description,
        'author' => $book->author,
    ], ['Accept' => 'application/json']);

    $this
        ->seeStatusCode(Response::HTTP_UNPROCESSABLE_ENTITY)
        ->seeJson([
            'title' => ["The title may not be greater than 255 characters."]
        ])
        ->notSeeInDatabase('books', ['title' => $book->title]);
}

/** @test **/
public function title_fails_update_validation_when_just_too_long()
{
    // Updating a book
    $book = factory(\App\Book::class)->create();
    $book->title = str_repeat('a', 256);

    $this->put("/books/{$book->id}", [
        'title' => $book->title,
        'description' => $book->description,
        'author' => $book->author
    ], ['Accept' => 'application/json']);

    $this
```

```
        ->seeStatusCode(Response::HTTP_UNPROCESSABLE_ENTITY)
        ->seeJson([
            'title' => ["The title may not be greater than 255 characters."]
        ])
        ->notSeeInDatabase('books', ['title' => $book->title]);
}
```

In the first test the `factory(\App\Book::class)->make()` method was introduced. When you call `->make()` from the factory instead of `->create()` it gives you an unsaved model that is not persisted in the database. Both tests make the title just too long to pass validation in order to test the threshold of failure. The error message is asserted and we also assert that a record with the invalid title is not in the database.

We need to wire up the controller changes to match these tests. At this point you should run tests, which will fail because the validation will still pass. We will fix that:

Adding Max Validation to BooksController@create

```
$this->validate($request, [
    'title' => 'required|max:255',
    'description' => 'required',
    'author' => 'required'
]);
```

Adding Max Validation to BooksController@update

```
$this->validate($request, [
    'title' => 'required|max:255',
    'description' => 'required',
    'author' => 'required'
]);
```

Validation rules are separated by the pipe `|` character, thus, title is `required` and `max:255` length.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (25 tests, 133 assertions)
```

The next two tests will assert that validation passes when we have exactly `255` characters:

Test that Validation Passes with a Title Exactly 255 Characters

```
/** @test **/
public function title_passes_create_validation_when_exactly_max()
{
    // Creating a new Book
    $book = factory(\App\Book::class)->make();
    $book->title = str_repeat('a', 255);

    $this->post("/books", [
        'title' => $book->title,
        'description' => $book->description,
        'author' => $book->author,
    ], ['Accept' => 'application/json']);

    $this
        ->seeStatusCode(Response::HTTP_CREATED)
        ->seeInDatabase('books', ['title' => $book->title]);
}

/** @test **/
public function title_passes_update_validation_when_exactly_max()
{
    // Updating a book
    $book = factory(\App\Book::class)->create();
    $book->title = str_repeat('a', 255);

    $this->put("/books/{$book->id}", [
        'title' => $book->title,
        'description' => $book->description,
        'author' => $book->author
    ], ['Accept' => 'application/json']);

    $this
        ->seeStatusCode(Response::HTTP_OK)
        ->seeInDatabase('books', ['title' => $book->title]);
}
```

The `BooksControllerTest` class already covers creating a valid book, so we only assert the correct status code which means validation passed. Last, we check that a new record exists in the database.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (27 tests, 137 assertions)
```

We now have additional `max:255` validation for the title that's working. As you can see, validation is really easy in Lumen, but it can also handle more complex rules. Our data is not complicated at the moment so a few built-in validation rules will suffice.

8.3: Custom Validation Messages

Lumen has good validation messages out of the box, but I will show you how we can customize them if needed. We will use the description field as an example.

The first thing we will do is update our tests to match the new error message we want to use. The validation test for `BooksController@create` should look like this now:

Test Custom Description Validation Message for BooksController@create

```
/** @test **/
public function it_validates_required_fields_when_creating_a_new_book()
{
    $this->post('/books', [], ['Accept' => 'application/json']);

    $this->assertEquals(Response::HTTP_UNPROCESSABLE_ENTITY, $this->response->ge\
tStatusCode());

    $body = json_decode($this->response->getContent(), true);

    $this->assertArrayHasKey('title', $body);
    $this->assertArrayHasKey('description', $body);
    $this->assertArrayHasKey('author', $body);

    $this->assertEquals(["The title field is required."], $body['title']);
    $this->assertEquals(
        ["Please fill out the description."],
        $body['description']
    );
    $this->assertEquals(["The author field is required."], $body['author']);
}
```

Test Custom Description Validation Message for BooksController@update

```php
/** @test **/
public function it_validates_validates_passed_fields_when_updating_a_book()
{
    $book = factory(\App\Book::class)->create();

    $this->put("/books/{$book->id}", [], ['Accept' => 'application/json']);

    $this->assertEquals(Response::HTTP_UNPROCESSABLE_ENTITY, $this->response->ge\
tStatusCode());

    $body = json_decode($this->response->getContent(), true);

    $this->assertArrayHasKey('title', $body);
    $this->assertArrayHasKey('description', $body);
    $this->assertArrayHasKey('author', $body);

    $this->assertEquals(["The title field is required."], $body['title']);
    $this->assertEquals(
        ["Please fill out the description."],
        $body['description']
    );
    $this->assertEquals(["The author field is required."], $body['author']);
}
```

The only thing we needed to change was the error message for the description field.

At this point the tests are failing and then we are ready to implement our custom message. The `Controller::validate()` method accepts an optional third argument, which is an associative array of custom validation messages. You can define them with the pattern `'attribute.<rule>' => <message>` to override a specific field:

Adding the Custom Validation Method for the Description

```php
$this->validate($request, [
    'title' => 'required|max:255',
    'description' => 'required',
    'author' => 'required'
], [
    'description.required' => 'Please fill out the :attribute.'
]);
```

Adding the Custom Description Message for BooksController@update

```
$this->validate($request, [
    'title' => 'required|max:255',
    'description' => 'required',
    'author' => 'required'
], [
    'description.required' => 'Please fill out the :attribute.'
]);
```

We added a custom message for the description's `required` validation rule. The validator understands certain placeholders like `:attribute` which will be resolved to the word "description". If we were to pass `'required' => 'Please fill out the :attribute.' the custom message would apply to all fields being validated with the` required' rule.

Time to run our test suite one last time and see if the new custom message worked:

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (27 tests, 137 assertions)
```

Conclusion

We breezed through adding basic API validation. At this point, I encourage you to read through the validation documentation[91] to become more familiar with the available built-in validation rules and making your own custom rules. You should have a good foundation for writing validation rules and tests to ensure your APIs respond to invalid data in a useful way. Validation not only helps keep bad data out of the database, it provides API consumers helpful information when bad data is submitted to the application.

Git commit: Add Book API Validation

74cb5fa[92]

[91]https://lumen.laravel.com/docs/5.1/validation
[92]https://bitbucket.org/paulredmond/bookr/commits/74cb5fa

Chapter 9: Authors

So far our `/books` endpoint provides an `author` column on the `books` database table. Making authors a string field on the books table was intentional so we could focus on other primary concepts of building our API, but now we are ready to create proper author data and provide API endpoints for author data.

The high-level overview of this chapter will be:

- Creating the authors database schema
- Creating a relationship between book data and authors
- Create API endpoints for author information
- Modify our `/books` endpoints to use new author data

9.1: The Authors database schema

Creating the `authors` database table involves a few migrations:

1. Create a new authors database table
2. Associate authors and books
3. Remove the deprecated `author` column on the `books` table

We will start with creating the new `authors` database table. If you remember when we created the `books` table we have an artisan command to create migration scripts:

Creating the Authors Database Migration

```
# vagrant@homestead:~/Code/bookr$
$ php artisan make:migration create_authors_table --create=authors
```

Add the following columns to the `up()` method:

The Authors Table Migration

```php
/**
 * Run the migrations.
 *
 * @return void
 */
public function up()
{
    Schema::create('authors', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->enum('gender', ['male', 'female']);
        $table->text('biography');
        $table->timestamps();
    });
}
```

This time around we will move at a quicker pace so you might want to review our original migration of the `books` table.

The `up` migration might be the first time you've seen the `$table->text()` method and `$table->enum()` in a migration:

- The `text()` method is the TEXT equivalent for the database
- The `enum()` method is equivalent to a relational database ENUM field

We can go ahead and run the migration to try it out:

Run the Database Migration

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh
Rolled back: 2015_12_29_234835_create_books_table
Migrated: 2015_12_29_234835_create_books_table
Migrated: 2016_01_20_050526_create_authors_table
```

Keep in mind that commands like `migrate:refresh` will remove data from the database, but this is a good thing in development because we can depend on seed data to quickly get a repeatable development environment.

The next migration will define the database association between the `books` table and the `authors` table. Because we are using Eloquent we have access to the following types of relationships:

- One to One
- One to Many
- Many to Many
- Has Many Through
- Polymorphic Relationships
- Many to Many Polymorphic Relationships

You can read more about relationships[93] in the Laravel documentation. We will pick "One to Many" although a "Many to Many" relationship might be better suited if you are allowing multiple authors. For the purposes of our API a book will only have *one* author.

Make the Migration to Associate Books and Authors

```
# vagrant@homestead:~/Code/bookr$
$ php artisan make:migration \
associate_books_with_authors --table=books
```

This migration is a little trickier than our previous experience with migrations:

The Migration to Associate Authors and Books

```
<?php

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

class AssociateBooksWithAuthors extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('books', function (Blueprint $table) {

            // Create the author_id column as an unsigned integer
            $table->integer('author_id')->after('id')->unsigned();

            // Create a basic index for the author_id column
```

[93] http://laravel.com/docs/5.1/eloquent-relationships

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

            // Create a foreign key constraint and cascade on delete.
            $table
                ->foreign('author_id')
                ->references('id')
                ->on('authors')
                ->onDelete('cascade');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('books', function (Blueprint $table) {
            // Drop the foreign key first
            $table->dropForeign('books_author_id_foreign');

            // Now drop the basic index
            $table->dropIndex('books_author_id_index');

            // Lastly, now it's safe to drop the column
            $table->dropColumn('author_id');
        });
    }
}
```

I will start by explaining the `up` method:

- Create an unsigned integer column `author_id` *after* the `id` column
- Add a basic index to the `author_id` column
- Add a foreign key associated with the `authors.id` column
- The foreign key should cascade on delete

The `down` method looks simple but is actually a little trickier:

- You *must* drop the foreign key first

- You then *must* drop the index
- Finally, you are safe to drop the `author_id` column

A migration should be able to be applied and rolled back without error. We will try out applying and rolling back to make sure our migration is working as expected:

Making Sure Migrations can be Applied and Rolled Back

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh
Rolled back: 2016_01_20_050526_create_authors_table
Rolled back: 2015_12_29_234835_create_books_table
Migrated: 2015_12_29_234835_create_books_table
Migrated: 2016_01_20_050526_create_authors_table
Migrated: 2016_01_20_051118_associate_books_with_authors
```

Now that we have a working migration we need to update our seed data for the `books` table and create a new factory for authors. First we need an eloquent model for `authors`:

Create the Author Model File

```
# vagrant@homestead:~/Code/bookr$
$ touch app/Author.php
```

Our `Author` model will look like the following:

The Author Model Class

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Author extends Model
{
    /**
     * The attributes that are mass assignable
     *
     * @var array
     */
    protected $fillable = ['name', 'biography', 'gender'];

    public function books()
    {
        return $this->hasMany(Book::class);
    }
}
```

The `$fillable` property was introduced in the `Book` model and defines mass-assignable columns. Next, we define a `books()` method that has the one-to-many relationship with the `Book` model. The `hasMany` method takes the `Book` model class name. By convention the `Author` model will use `author_id` on the `books` table to look up associations.

Next we need to adjust our `Book` model to include the `authors` association:

The Book Model Association with Authors

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * The attributes that are mass assignable
     *
     * @var array
     */
    protected $fillable = ['title', 'description', 'author'];

    public function author()
    {
        return $this->belongsTo(Author::class);
    }
}
```

By convention the `belongsTo` method sets up the association with the `Author` model and Eloquent will take the `author_id` and find the author record with the `Author.id` field. You can tweak the foreign key name as the second argument in `belongsTo`, but we will follow the naming conventions that automatically take care of it.

The eloquent relationships[94] documentation explains in detail how you can customize model associations and how to use them.

[94]http://laravel.com/docs/5.1/eloquent-relationships

With the one-to-many association in place Eloquent can load data about the associated models, for example:

Example of Using Eloquent Associations

```
$book = Book::find(1);
echo $book->author->name;

$author = Author::find(1);
foreach ($author->books as $book) {
    echo $book->title;
}
```

The models are ready to go and we have our schema migration working! Now we need to add a new factory definition for authors and update seed data. We will start by adding the factory definition to `database/factories/ModelFactory.php`.

The Author Model Factory

```
$factory->define(App\Author::class, function ($faker) {
    return [
        'name' => $faker->name,
        'biography' => join(" ", $faker->sentences(rand(3, 5))),
        'gender' => rand(1, 6) % 2 === 0 ? 'male' : 'female'
    ];
});
```

The `biography` key uses the `$faker->sentences()` method from the `Faker\Provider\Lorem` provider. The `$faker->sentences()` method takes an integer for the first parameter which determines how many sentences will be generated. We use a random number between 3 and 5 sentences and join the array with an empty space (" "). The `gender` key can either be `male` or `female` and we use the modulus operator to randomly pick male or female. An even random number will be male, and an odd number will be female.

Next we will put our new factory to work by using it to seed book and author data. We will modify the `database/seeds/BooksTableSeeder.php` to use our model association:

Seed Author and Book Data

```php
<?php

use Carbon\Carbon;
use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class BooksTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        factory(App\Author::class, 10)->create()->each(function ($author) {
            $booksCount = rand(1, 5);

            while ($booksCount > 0) {
                $author->books()->save(factory(App\Book::class)->make());
                $booksCount--;
            }
        });
    }
}
```

The modified `BooksTableSeeder` is not complicated; it creates 10 authors and then iterates over each record with a `Closure` callback. Inside the `each` callback we get a random `$booksCount` integer to determine how many books an author will have. The `while` loop is used to keep creating new books for the author until the `$bookCount` is 0.

Feel free to purge your database and run the seeder again to try it out:

Purge the Database and Apply the Modified `BooksTableSeeder`

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh
$ php artisan db:seed
```

Keep in mind that if our application were in production, we would need to migrate the data *and* the schema. Since we are in early development we will not worry about losing data in the database.

We have one final migration that will remove the `authors` column in the database.

Migration to Remove the Authors Column from the Books Table

```
# vagrant@homestead:~/Code/bookr$
$ php artisan make:migration \
remove_books_authors_column --table=books
```

The migration will drop the `author` column on `up` and add it back on `down`. Like I mentioned, if this application were in production we might have to find all the authors and migrate them over to the new `authors` table inside of the `up` method. The `down` callback would revers the effects of migrating the authors column to a new table.

Here is the simple migration for removing the `author` column:

Cleaning Up the Author Column on the `books` Table.

```php
<?php

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

class RemoveBooksAuthorsColumn extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('books', function (Blueprint $table) {
            $table->dropColumn('author');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('books', function (Blueprint $table) {
            $table->string('author');
        });
    }
}
```

Time to run the migration and see what happens:

Refresh the Database Migrations

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh
Rolled back: 2016_01_20_051118_associate_books_with_authors
Rolled back: 2016_01_20_050526_create_authors_table
Rolled back: 2015_12_29_234835_create_books_table
Migrated: 2015_12_29_234835_create_books_table
Migrated: 2016_01_20_050526_create_authors_table
Migrated: 2016_01_20_051118_associate_books_with_authors
Migrated: 2016_01_20_052512_remove_books_authors_column
```

You will get errors now if you try to run `php artisan db:seed` after running the migration to remove the `authors` column from the `books` table. We need to fix the db seeder which is now broken due to the database migration.

```
# vagrant@homestead:~/Code/bookr$
$ php artisan db:seed
...
[PDOException]
  SQLSTATE[42S22]: Column not found: 1054 Unknown column 'author' in 'field list'
...
```

To fix the seeder we need to remove the `author` key in the `Book` factory. Open the `database/factories/ModelFactory.php` file and locate the `Book` factory:

Remove the Author Key from the Factory

```
$factory->define(App\Book::class, function ($faker) {
    $title = $faker->sentence(rand(3, 10));

    return [
        'title' => substr($title, 0, strlen($title) - 1),
        'description' => $faker->text,
    ];
});
```

Now you can refresh seed data successfully.

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh
...
$ php artisan db:seed
```

That wraps up our database schema and migration. We have a new authors table and we've cleaned up migrations and database seeding. At this point the tests will blow up, so we intend to find out what broke and fix them.

9.2: Fixing Broken Tests

We need to fix the tests that are failing as the result of schema changes. Although fixing broken tests isn't the most glamorous part of programming, it's nice to know that our test suite catches broken code before we ship it.

Since we just changed our database schema it makes sense that various tests adding database records would break since we removed the `authors` column from the `books` database. We have a requirement of an author name in our `/books` endpoints too, which doesn't exist in our schema.

If you inspect the `BooksControllerTest` you will see this factory call a few times:

```
$books = factory('App\Book', 2)->create();
```

Similar to the `BooksTableSeeder` change we've made in this chapter, various tests will need to create an author and then associate that author with a book. It makes sense to create a method to avoid boilerplate code everywhere. Add the following to the `tests/TestCase.php` from which our tests extend from:

A Book Factory Method in `tests/TestCase.php`

```
/**
 * Convenience method for creating a book with an author
 *
 * @param int $count
 * @return mixed
 */
protected function bookFactory($count = 1)
{
    $author = factory(\App\Author::class)->create();
    $books  = factory(\App\Book::class, $count)->make();

    if ($count === 1) {
        $books->author()->associate($author);
        $books->save();
    } else {
        foreach ($books as $book) {
            $book->author()->associate($author);
            $book->save();
        }
    }

    return $books;
}
```

The `bookFactory` method is creating an author in the database. Next, the method calls `make()` to populate a `Book` model instance that is not yet saved in the database. The `author_id` column is required by a foreign key constraint so we need to call `make()` so we can attach the `author_id` before inserting the record in the database. The "if" statement checks to see how many books should be created. If only one book needs created we attach the author to the book, otherwise we loop through each book and attach the author to all books.

Now that we have a convenient way of creating books and authors in the test database, we need to determine which tests need fixing. The easiest way to do that is to run the whole `phpunit` suite and go through each error one-by-one. As we find an error we update it and run the test in isolation until it passes. We then move on to the next error, and so on, until we get back to green.

I will spare you the output from all of the failures, but the first one we will work on is this error:

```
1) Tests\App\Http\Controllers\BooksControllerTest::store_should_save_new_book_in\
_the_database
PHPUnit_Framework_Exception: Argument #2 (No Value) of PHPUnit_Framework_Assert:\
:assertArrayHasKey() must be a array or ArrayAccess

bookr/tests/app/Http/Controllers/BooksControllerTest.php:108
```

The `BooksController@store` method is obviously breaking. Lets find out what is happening:

Temporarily Dump the `$body` Variable

```
/** @test **/
public function store_should_save_new_book_in_the_database()
{
    $this->post('/books', [
        'title' => 'The Invisible Man',
        'description' => 'An invisible man is trapped in the terror of his own c\
reation',
        'author' => 'H. G. Wells'
    ]);

    $body = json_decode($this->response->getContent(), true);
    dd($body, $this->response->getStatusCode());
    // ....
}
```

The `dd()` function is a convenience function for dumping the variable and exiting the program. This is the output you will receive when you run the test now:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=store_should_save_new_book_in_the_database

null
500
```

The response body is `null` and the status code is `500`. Weird! We should comb the logs and see what we can find. Clear out the `storage/logs/lumen.log` file and run the test again. You should see something like the following:

Test Error in `storage/logs/lumen.log`

```
[2015-11-21 15:45:50] lumen.ERROR: exception 'PDOException' with message 'SQLSTA\
TE[42S22]: Column not found: 1054 Unknown column 'author' in 'field list'' in bo\
okr/vendor/illuminate/database/Connection.php:390
```

The `BooksController@store` method is trying to insert an `author` column that no longer exists in the `books` table. We could remove it from our test to get the next error, but I want to show you another issue here: the `Book` model has defined `author` as a `fillable` field. We need to remove that field which is invalid now:

Update the `app/Book.php` Fillable Fields

```php
/**
 * The attributes that are mass assignable
 *
 * @var array
 */
protected $fillable = ['title', 'description'];
```

Now if we run our test again we will get a different error:

A PDOException Error in `storage/logs/lumen.log`

```
[2015-11-21 15:57:56] lumen.ERROR: exception 'PDOException' with message 'SQLSTA\
TE[23000]: Integrity constraint violation: 1452 Cannot add or update a child row\
: a foreign key constraint fails (`bookr_testing`.`books`, CONSTRAINT `books_aut\
hor_id_foreign` FOREIGN KEY (`author_id`) REFERENCES `authors` (`id`) ON DELETE \
CASCADE)' in bookr/vendor/illuminate/database/Connection.php:390
```

Now the `BooksController@store` method is failing because of a missing foreign key constraint; we are not passing a proper `author_id` in the request. We are creating a book so we just need to generate a valid author in our test.

Update Test to Pass an Author ID

```php
/** @test **/
public function store_should_save_new_book_in_the_database()
{
    $author = factory(\App\Author::class)->create([
        'name' => 'H. G. Wells'
    ]);

    $this->post('/books', [
        'title' => 'The Invisible Man',
        'description' => 'An invisible man is trapped in the terror of his own c\
reation',
        'author_id' => $author->id
    ], ['Accept' => 'application/json']);

    $body = json_decode($this->response->getContent(), true);

    $this->assertArrayHasKey('data', $body);

    $data = $body['data'];
    $this->assertEquals('The Invisible Man', $data['title']);
    $this->assertEquals(
        'An invisible man is trapped in the terror of his own creation',
        $data['description']
    );
    $this->assertEquals('H. G. Wells', $data['author']);
    $this->assertTrue($data['id'] > 0, 'Expected a positive integer, but did not\
 see one.');

    $this->assertArrayHasKey('created', $data);
    $this->assertEquals(Carbon::now()->toIso8601String(), $data['created']);
    $this->assertArrayHasKey('updated', $data);
    $this->assertEquals(Carbon::now()->toIso8601String(), $data['updated']);

    $this->seeInDatabase('books', ['title' => 'The Invisible Man']);
}
```

Lets run the test again:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=store_should_save_new_book_in_the_database

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::store_should_save_new_book_in\
_the_database
Failed asserting that an array has the key 'data'.

/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:117

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

We're still getting errors, let's investigate the response:

Debug the Failing Test (partial code)

```
/** @test **/
public function store_should_save_new_book_in_the_database()
{
    $author = factory(\App\Author::class)->create([
        'name' => 'H. G. Wells'
    ]);

    $this->post('/books', [
        'title' => 'The Invisible Man',
        'description' => 'An invisible man is trapped in the terror of his own c\
reation',
        'author_id' => $author->id
    ], ['Accept' => 'application/json']);

    $body = json_decode($this->response->getContent(), true);
    dd($body);
    // ...
```

The output should be something like this:

Output from Debugging the failing test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=store_should_save_new_book_in_the_database

array:1 [
  "author" => array:1 [
    0 => "The author field is required."
  ]
]
```

We have failed validation constraint for the `author` column that is no longer valid. We need to replace it with a validation rule for a valid `author_id`:

Update Validation in `BooksController@store`

```
$this->validate($request, [
    'title' => 'required|max:255',
    'description' => 'required',
    'author_id' => 'required|exists:authors,id'
], [
    'description.required' => 'Please fill out the :attribute.'
]);
```

We've replaced `author` with `author_id` and use the `exists` validation rule that ensures an author record exist in the `authors` table. The `exists` validation in this case reads like this: `exists:<table>,<column>`. If you omit the column it will use the field associated with the validation rule, in this case `author_id`.

At this point running the test again will still fail (see if you can figure out why on your own). We haven't changed the controller to set the `author_id` property on the model, although we are passing it at this point in the failing test. We need to allow this field to be fillable:

Making the `author_id` Field Mass-assignable

```
/**
 * The attributes that are mass assignable
 *
 * @var array
 */
protected $fillable = ['title', 'description', 'author_id'];
```

Now our controller should be able to populate the `author_id` field when it calls `$book = Book::create($request->all());`. Lets run our test again and see:

Running the Test after making `author_id` fillable

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=store_should_save_new_book_in_the_database

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::store_should_save_new_book_in\
_the_database
Array (...) does not match expected type "string".

bookr/tests/app/Http/Controllers/BooksControllerTest.php:121
```

It looks like our save is now succeeding, but our assertions are not matching up anymore. The response looks like this now if you call `dd()`:

Dumped Response from the Test

```
array:1 [
  "data" => array:6 [
    "id" => 1
    "title" => "The Invisible Man"
    "description" => "An invisible man is trapped in the terror of his own creat\
ion"
    "author" => array:6 [
      "id" => 1
      "name" => "H. G. Wells"
      "gender" => "female"
      "biography" => "Ut quis doloremque dolorem eaque. Repellendus et dolor eos\
 doloribus. Velit omnis alias ut fugiat molestias ab velit."
      "created_at" => "2015-11-21 16:32:34"
      "updated_at" => "2015-11-21 16:32:34"
    ]
    "created" => "2015-11-21T16:32:34+0000"
    "updated" => "2015-11-21T16:32:34+0000"
  ]
]
```

The `ArticleTransformer` is now outputting the entire author record instead of just the author name. Later on in the book we will change the way responses include author data, but for now lets just keep our API the same. We can now use the associated author model to pass the author name.

Update the `BookTransformer::transform()` Method

```php
/**
 * Transform a Book model into an array
 *
 * @param Book $book
 * @return array
 */
public function transform(Book $book)
{
    return [
        'id'          => $book->id,
        'title'       => $book->title,
        'description' => $book->description,
        'author'      => $book->author->name,
        'created'     => $book->created_at->toIso8601String(),
        'updated'     => $book->updated_at->toIso8601String(),
    ];
}
```

Let's run our test again after the transformer change. If you have any `dd()` calls in your test don't forget to remove them!

Running the Test After Updating the `BookTransformer`

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=store_should_save_new_book_in_the_database

OK (1 test, 10 assertions)
```

We had to do considerable amount of investigating for a simple change, but it was a good exercise to get more familiar with debugging failing tests.

We are ready to move on to the `update_should_only_change_fillable_fields` failing test failure:

Failing Test for Updating Fillable Fields

```
1) Tests\App\Http\Controllers\BooksControllerTest::update_should_only_change_fil\
lable_fields
Illuminate\Database\QueryException: SQLSTATE[42S22]: Column not found: 1054 Unkn\
own column 'author' in 'field list'...
```

This error looks familiar. The `BooksController@update` and accompanying tests are still using the `author` parameter. We will start by updating the test:

Fixes for the `update_should_only_change_fillable_fields` Test

```
/** @test **/
public function update_should_only_change_fillable_fields()
{
    $book = $this->bookFactory();

    $this->notSeeInDatabase('books', [
        'title' => 'The War of the Worlds',
        'description' => 'The book is way better than the movie.',
    ]);

    $this->put("/books/{$book->id}", [
        'id' => 5,
        'title' => 'The War of the Worlds',
        'description' => 'The book is way better than the movie.'
    ], ['Accept' => 'application/json']);

    $this
        ->seeStatusCode(200)
        ->seeJson([
            'id' => 1,
            'title' => 'The War of the Worlds',
            'description' => 'The book is way better than the movie.'
        ])
        ->seeInDatabase('books', [
            'title' => 'The War of the Worlds'
        ]);

    $body = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $body);

    $data = $body['data'];
    $this->assertArrayHasKey('created', $data);
    $this->assertEquals(Carbon::now()->toIso8601String(), $data['created']);
    $this->assertArrayHasKey('updated', $data);
    $this->assertEquals(Carbon::now()->toIso8601String(), $data['updated']);
}
```

It is a good idea to remove changing the author from the test and focus on updating the book. Now that we have a separate database table for authors we should make a separate test for changing the author. We also remove the `author` column from the `notSeeInDatabase` and the `$this->seeJson()` assertion.

Run the Test After Updating

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=update_should_only_change_fillable_fields

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::update_should_only_change_fil\
lable_fields
Failed asserting that 422 matches expected 200.
```

It looks like we are getting a validation error from our missing `author` parameter. Let's change the validation rule to match the `BooksController@update` change we just made:

Update Validation for `BooksController@update`

```
$this->validate($request, [
    'title' => 'required|max:255',
    'description' => 'required',
    'author_id' => 'exists:authors,id'
], [
    'description.required' => 'Please fill out the :attribute.'
]);
```

An important difference between `BooksController@update` and `BooksController@store` validation is that we don't use the `require` rule on update. We only check that the records exists in the `authors` table if the field is present and allow the endpoint to optionally change the author.

Run the Test After Updating Validation

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=update_should_only_change_fillable_fields

OK (1 test, 12 assertions)
```

Moving on, the next error from the suite is from the "`destroy_should_remove_a_valid_book`" test.

The Next Error from Our Test Suite

```
1) Tests\App\Http\Controllers\BooksControllerTest::destroy_should_remove_a_valid\
_book
Illuminate\Database\QueryException: SQLSTATE[23000]: Integrity constraint violat\
ion: 1452 Cannot add or update a child row: a foreign key constraint fails
```

The fail test's `factory()` call fails because of the foreign key constraint. Fixing this test is a one-line change using our new `bookFactory()` method:

Fixing the Failing `BooksController@destroy` Test

```
/** @test **/
public function destroy_should_remove_a_valid_book()
{
    $book = $this->bookFactory();

    $this
        ->delete("/books/{$book->id}")
        ->seeStatusCode(204)
        ->isEmpty();

    $this->notSeeInDatabase('books', ['id' => $book->id]);
}
```

Running the Test After Updating `BooksController@destroy`

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=destroy_should_remove_a_valid_book

OK (1 test, 2 assertions)
```

We've fixed a few individual tests, let's run the entire suite and see what remains.

The Next Error from the Test Suite

```
1) Tests\App\Http\Controllers\BooksControllerValidationTest::it_validates_requir\
ed_fields_when_updating_a_book
Illuminate\Database\QueryException: SQLSTATE[23000]: Integrity constraint violat\
ion: 1452 Cannot add or update a child row: a foreign key constraint fails
```

The error above comes from `tests/app/Http/Controllers/BooksControllerValidationTest.php` and looks like another factory failure:

Updating the Test to Use the `bookFactory()` Method

```
/** @test **/
public function it_validates_required_fields_when_updating_a_book()
{
    $book = $this->bookFactory();

    $this->put("/books/{$book->id}", [], ['Accept' => 'application/json']);

    $this->assertEquals(Response::HTTP_UNPROCESSABLE_ENTITY, $this->response->ge\
tStatusCode());

    $body = json_decode($this->response->getContent(), true);

    $this->assertArrayHasKey('title', $body);
    $this->assertArrayHasKey('description', $body);

    $this->assertEquals(["The title field is required."], $body['title']);
    $this->assertEquals(["Please fill out the description."], $body['description\
']);
}
```

We replace the `factory()` call with the `bookFactory()` method and remove a few assertions for the `authors` validation.

Run the Validation Test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=it_validates_required_fields_when_updating_a_book

OK (1 test, 5 assertions)
```

We are getting the same foreign key errors in various places so let's bulk-update tests that are using the original `factory()` call and see what remains afterwards. Let's start with the `tests/app/Http/Controllers/BooksControllerValidationTest.php` file:

Replace All `factory()` Calls in the Validation Test File

```
/** @test **/
public function title_fails_create_validation_when_just_too_long()
{
    // Creating a book
    $book = $this->bookFactory();
    //...
}

/** @test **/
public function title_fails_update_validation_when_just_too_long()
{
    // Updating a book
    $book = $this->bookFactory();
    // ...
}

/** @test **/
public function title_passes_create_validation_when_exactly_max()
{
    // Creating a new Book
    $book = $this->bookFactory();
    // ...
}

/** @test **/
public function title_passes_update_validation_when_exactly_max()
{
    // Updating a book
    $book = $this->bookFactory();
    // ...
}
```

Next, we will update the `tests/app/Http/Controllers/BooksControllerTest.php` file:

Replace All `factory()` Calls in BooksControllerTest

```
/** @test **/
public function index_should_return_a_collection_of_records()
{
    $books = $this->bookFactory(2);
    // ...
}

/** @test **/
public function show_should_return_a_valid_book()
{
    $book = $this->bookFactory();
    // ...
}
```

The `BooksController@index` test creates two books since we are testing for a collection of records.

Hopefully the foreign key errors are behind us. Let's see what failures we have remaining in the `BooksControllerValidationTest` tests:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit tests/app/Http/Controllers/BooksControllerValidationTest.php

There were 2 failures:

1) Tests\App\Http\Controllers\BooksControllerValidationTest::it_validates_requir\
ed_fields_when_creating_a_new_book
Failed asserting that an array has the key 'author'.

/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerValidationTes\
t.php:24

2) Tests\App\Http\Controllers\BooksControllerValidationTest::title_passes_create\
_validation_when_exactly_max
Failed asserting that 422 matches expected 201.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerValidationTes\
t.php:107

FAILURES!
Tests: 6, Assertions: 20, Failures: 2.
```

It looks like we have a failed assertion for a key we are no longer providing and a `422` response code which means we have some validation errors when we don't expect them.

The first fix is simply removing an assertion for the `author` key and removing the author validation message check. This is what the test should look like in the `BooksControllerValidationTest` after you remove the author checks:

Remove Author Test Assertions

```php
/** @test **/
public function it_validates_required_fields_when_creating_a_new_book()
{
    $this->post('/books', [], ['Accept' => 'application/json']);

    $this->assertEquals(Response::HTTP_UNPROCESSABLE_ENTITY, $this->response->ge\
tStatusCode());

    $body = json_decode($this->response->getContent(), true);

    $this->assertArrayHasKey('title', $body);
    $this->assertArrayHasKey('description', $body);

    $this->assertEquals(["The title field is required."], $body['title']);
    $this->assertEquals(["Please fill out the description."], $body['description\
']);
}
```

The second failure in `BooksControllerValidationTest` happens because the `author_id` is a required field:

Fixing the `BooksController@create` Exactly Max Validation Test

```php
/** @test **/
public function title_passes_create_validation_when_exactly_max()
{
    // Creating a new Book
    $book = $this->bookFactory();
    $book->title = str_repeat('a', 255);
    $this->post("/books", [
        'title' => $book->title,
        'description' => $book->description,
        'author_id' => $book->author->id, // Pass a valid author
    ], ['Accept' => 'application/json']);

    $this
        ->seeStatusCode(Response::HTTP_CREATED)
        ->seeInDatabase('books', ['title' => $book->title]);
}
```

Next, there are some instances of `'author' => $book->author` we need to replace with a valid `author_id` in `BooksControllerValidationTest`.

Add the `author_id` Field to the POST Request

```php
/** @test **/
public function title_fails_create_validation_when_just_too_long()
{
    // Creating a book
    $book = $this->bookFactory();
    $book->title = str_repeat('a', 256);
    $this->post("/books", [
        'title' => $book->title,
        'description' => $book->description,
        'author_id' => $book->author->id,
    ], ['Accept' => 'application/json']);

    $this
        ->seeStatusCode(Response::HTTP_UNPROCESSABLE_ENTITY)
        ->seeJson([
            'title' => ["The title may not be greater than 255 characters."]
        ])
        ->notSeeInDatabase('books', ['title' => $book->title]);
}
```

Add the `author_id` Field to the PUT Request

```
/** @test **/
public function title_fails_update_validation_when_just_too_long()
{
    // Updating a book
    $book = $this->bookFactory();
    $book->title = str_repeat('a', 256);
    $this->put("/books/{$book->id}", [
        'title' => $book->title,
        'description' => $book->description,
        'author_id' => $book->author->id,
    ], ['Accept' => 'application/json']);

    $this
        ->seeStatusCode(Response::HTTP_UNPROCESSABLE_ENTITY)
        ->seeJson([
            'title' => ["The title may not be greater than 255 characters."]
        ])
        ->notSeeInDatabase('books', ['title' => $book->title]);
}
```

Add the `author_id` Field to the PUT Request

```
/** @test **/
public function title_passes_update_validation_when_exactly_max()
{
    // Updating a book
    $book = $this->bookFactory();
    $book->title = str_repeat('a', 255);

    $this->put("/books/{$book->id}", [
        'title' => $book->title,
        'description' => $book->description,
        'author_id' => $book->author->id,
    ], ['Accept' => 'application/json']);

    $this
        ->seeStatusCode(Response::HTTP_OK)
        ->seeInDatabase('books', ['title' => $book->title]);
}
```

After those author changes we are ready to run all the tests in the `BooksControllerValidationTest` class:

Run the Tests for `BooksControllerValidationTest`

```
# vagrant@homestead:~/Code/bookr$
$ phpunit tests/app/Http/Controllers/BooksControllerValidationTest.php

OK (6 tests, 22 assertions)
```

We are starting to see the light at the end of the refactor tunnel! Let's run our test suite and see what remains.

Running the Full Test Suite After Our Fixes

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There were 3 failures:

1) Tests\App\Http\Controllers\BooksControllerTest::index_should_return_a_collect\
ion_of_records
Unable to find JSON fragment...

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:375
/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:355
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:50

FAILURES!
Tests: 27, Assertions: 110, Errors: 1, Failures: 3.
```

I've only provided partial error output, but if you inspect the "Unable to find JSON fragment" error closely you will notice that the response includes all author data from the `Author` model. We need to update our test to check the author's name:

Use Author Model in the Assertion of the `author` Property

```
/** @test **/
public function index_should_return_a_collection_of_records()
{
    $books = $this->bookFactory(2);

    $this->get('/books');

    $content = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $content);

    foreach ($books as $book) {
        $this->seeJson([
            'id' => $book->id,
            'title' => $book->title,
            'description' => $book->description,
            'author' => $book->author->name, // Check the author's name
            'created' => $book->created_at->toIso8601String(),
            'updated' => $book->updated_at->toIso8601String(),
        ]);
    }
}
```

Test the `BooksController@index` Test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=index_should_return_a_collection_of_records

OK (1 test, 15 assertions)
```

We run `phpunit` again to get the last two failures:

Getting the Next PHPUnit Failure

```
# vagrant@homestead:~/Code/bookr$
$ phpunit
...
There were 2 failures:

1) Tests\App\Http\Controllers\BooksControllerTest::show_should_return_a_valid_bo\
ok
Failed asserting that 'Miss Anahi Tromp DDS' matches expected App\Author Object

FAILURES!
Tests: 27, Assertions: 122, Errors: 1, Failures: 2.
```

We're getting closer! The next error looks like an invalid comparison of the `Author` model. Looking at the `show_should_return_a_valid_book` test closely, you need to update the following `assertEquals` check:

```
/** @test **/
public function show_should_return_a_valid_book()
{
    $book = $this->bookFactory();

    $this
        ->get("/books/{$book->id}")
        ->seeStatusCode(200);

    // Get the response and assert the data key exists
    $content = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $content);
    $data = $content['data'];

    // Assert the Book Properties match
    $this->assertEquals($book->id, $data['id']);
    $this->assertEquals($book->title, $data['title']);
    $this->assertEquals($book->description, $data['description']);
    $this->assertEquals($book->author->name, $data['author']);
    $this->assertEquals($book->created_at->toIso8601String(), $data['created']);
    $this->assertEquals($book->updated_at->toIso8601String(), $data['created']);
}
```

Run the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\BooksControllerTest::store_should_respond_with_a_2\
01_and_location_header_when_successful
Failed asserting that 302 matches expected 201.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/BooksControllerTest.php:146

FAILURES!
Tests: 27, Assertions: 124, Errors: 1, Failures: 1.
```

The "`Failed asserting that 302 matches expected 201`" message means that the controller is sending a redirect response. We are communicating with JSON clients so we need to send the request with the `Accept: application/json` header so we get back a JSON response instead of the default failed validation redirect. While looking at this test, we can see that validation will fail because we need to send an `author_id` which is required in `BooksController@create`, so let's update both now:

Use the Author Model to fix the failing Test

```php
/** @test */
public function store_should_respond_with_a_201_and_location_header_when_success\
ful()
{
    $author = factory(\App\Author::class)->create();
    $this->post('/books', [
        'title' => 'The Invisible Man',
        'description' => 'An invisible man is trapped in the terror of his own c\
reation',
        'author_id' => $author->id
    ], ['Accept' => 'application/json']);

    $this
        ->seeStatusCode(201)
        ->seeHeaderWithRegExp('Location', '#/books/[\d]+$#');
}
```

Running the individual test pass now:

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=store_should_respond_with_a_201_and_location_header_when_succ\
essful

OK (1 test, 3 assertions)
```

Let's try to run our entire suite again and see if we are back to green yet:

Run the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 error:

1) Tests\App\Transformer\BookTransformerTest::it_transforms_a_book_model
Illuminate\Database\QueryException: SQLSTATE[23000]: Integrity constraint violat\
ion...

FAILURES!
Tests: 27, Assertions: 126, Errors: 1.
```

It looks like we have one more error and we should be back to green. The error is another foreign key constraint violation in the `BooksTransformerTest` file. We just need to use the `bookFactory()` method:

Update the Failing `BookTransformerTest`

```php
/** @test **/
public function it_transforms_a_book_model()
{
    $book = $this->bookFactory();
    $subject = new BookTransformer();

    $transform = $subject->transform($book);

    $this->assertArrayHasKey('id', $transform);
    $this->assertArrayHasKey('title', $transform);
    $this->assertArrayHasKey('description', $transform);
    $this->assertArrayHasKey('author', $transform);
    $this->assertArrayHasKey('created', $transform);
    $this->assertArrayHasKey('updated', $transform);
}
```

We should be completely done and have full passing tests again.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (27 tests, 132 assertions)
```

Finally!

Git Commit: Add Author Model Data to Books

3bbff2c[95]

Conclusion

We've successfully refactored our code to use the new `authors` model. For a minor change you can see that our test suite took a bit of work to fix. While annoying, good test coverage is a huge time saver in the long run.

The small investment in fixing tests will pay divedens. You can rip out the guts of your code and have confidence that test specifications will lead you to less bugs and keeps business and code requirements documented.

We could have designed our schema with separate tables up front in a real project. Thinking about schema design early is important, but do not over-design your schema up front if it blocks you from developing iteratively. Our string column for the author name was fine to get us going, and we were able to refactor things (somewhat) painlessly. You don't have to finish the application in one sitting and you may get a few things wrong at first.

[95]https://bitbucket.org/paulredmond/bookr/commits/3bbff2c

Chapter 10: The /authors API Resource

In this chapter we are going to build on the last chapter's introduction to the `authors` table. We will start working on API endpoints for our dedicated `Author` resource! At this point you should be familiar with the basics of defining routes and working with controllers. Reference Chapter 4 and Chapter 5 to get a refresher as needed.

The author endpoint will provide API endpoints for getting author details and books they've authored. Along the way I will show you a really cool feature of Fractal that allows you to include associated transformer data in a response; this means we can easily include book data in author responses if desired.

Our basic API endpoints will look like the following:

Basic REST `/authors` resource

```
GET     /authors        Get all the authors
POST    /authors        Create a new author
GET     /authors/{id}   Get an author's details
PUT     /authors/{id}   Update an author
DELETE  /authors/{id}   Delete an author
```

Before we work on the individual endpoints lets define all our routes in `app/Http/routes.php`:

Defining the Author Routes

```php
$app->group([
    'prefix' => '/authors',
    'namespace' => 'App\Http\Controllers'
], function (\Laravel\Lumen\Application $app) {
    $app->get('/', 'AuthorsController@index');
    $app->post('/', 'AuthorsController@store');
    $app->get('/{id:[\d]+}', 'AuthorsController@show');
    $app->put('/{id:[\d]+}', 'AuthorsController@update');
    $app->delete('/{id:[\d]+}', 'AuthorsController@destroy');
});
```

The code snippet introduces the `$app->group()` method which accepts the following keys: `prefix`, `namespace`, and `middleware`. We need to define the `namespace` key so the group knows how to locate the `AuthorsController` and we define a prefix of `/authors` that all routes in the group will use.

10.1: The GET /authors Endpoint

Our first endpoint will get all authors. We will create an `AuthorTransformer` that we can use on all author responses. If you recall in the last chapter, we created a factory and database seeder for authors that we can use to write our tests for the `/authors` endpoints. Time to code.

First up, let's create the files necessary for this whole section:

Create the AuthorTransformer Class

```
# vagrant@homestead:~/Code/bookr$
$ touch app/Transformer/AuthorTransformer.php
$ touch tests/app/Transformer/AuthorTransformerTest.php
$ touch app/Http/Controllers/AuthorsController.php
$ touch tests/app/Http/Controllers/AuthorsControllerTest.php
```

The AuthorsTransformer

Let's start with the `AuthorTransformerTest` class:

The `AuthorTransformerTest` Skeleton Class

```php
<?php

namespace Tests\App\Transformer;

use TestCase;
use App\Transformer\AuthorTransformer;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class AuthorTransformerTest extends TestCase
{
    use DatabaseMigrations;

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

        $this->subject = new AuthorTransformer();
    }
    /** @test **/
    public function it_can_be_initialized()
    {
        $this->assertInstanceOf(AuthorTransformer::class, $this->subject);
    }
}
```

Our test file is fairly familiar. As a convenience I've opted to construct the transformer before each test in the `setUp()` method and assign it to `$this->subject`.

Next, we create the `AuthorTransformer` class to get the test passing:

Create the `AuthorTransformer` Class

```php
<?php

namespace App\Transformer;

use App\Author;
use League\Fractal\TransformerAbstract;

class AuthorTransformer extends TransformerAbstract
{
}
```

Now that we have the boilerplate code, we need to write tests for our specification. First, the `AuthorTransformer` should simply need to be able to transform an `Author` model in a consistent way that we expect:

The Test Specification for the transform() method

```php
/** @test **/
public function it_can_transform_an_author()
{
    $author = factory(\App\Author::class)->create();

    $actual = $this->subject->transform($author);

    $this->assertEquals($author->id, $actual['id']);
    $this->assertEquals($author->name, $actual['name']);
    $this->assertEquals($author->gender, $actual['gender']);
    $this->assertEquals($author->biography, $actual['biography']);
    $this->assertEquals(
        $author->created_at->toIso8601String(),
        $actual['created']
    );
    $this->assertEquals(
        $author->updated_at->toIso8601String(),
        $actual['created']
    );
}
```

Our assertions are straightforward and make sure that the transformer includes all the expected author properties. Now write the `AuthorTransformer::transform()` implementation to get the newly written test passing:

Implement the `AuthorTransformer::transform()` Method

```php
/**
 * Transform an author model
 *
 * @param Author $author
 * @return array
 */
public function transform(Author $author)
{
    return [
        'id'        => $author->id,
        'name'      => $author->name,
        'gender'    => $author->gender,
        'biography' => $author->biography,
        'created'   => $author->created_at->toIso8601String(),
        'updated'   => $author->created_at->toIso8601String(),
    ];
}
```

You should have been running tests after writing each test and then after implementing it. Let's make sure we are passing now.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (29 tests, 139 assertions)
```

The Author Controller

Now that we have a basic `AuthorTransformer` we will change focus to the `AuthorsController@index` route. Let's write our first test in the `tests/app/Http/Controllers/AuthorsControllerTest.php` file:

Testing the `AuthorsController@index` for a `200` Status Code

```php
<?php

namespace Tests\App\Http\Controllers;

use TestCase;
use Illuminate\Http\Response;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class AuthorsControllerTest extends TestCase
{
    use DatabaseMigrations;

    /** @test **/
    public function index_responds_with_200_status_code()
    {
        $this->get('/authors')->seeStatusCode(Response::HTTP_OK);
    }
}
```

Running the test at this point results in a failing test with a `500` status code because we haven't defined the controller. Add the following controller code to get the test back to green:

Defining the AuthorsController

```php
<?php

namespace App\Http\Controllers;

use App\Author;

class AuthorsController extends Controller
{
    public function index()
    {
    }
}
```

Just defining the route and controller method results in a `200` status code. Next, we will write assertions to test that the `AuthorsController@index` returns a collection of records in the `AuthorsControllerTest` file:

Test that the `AuthorsController@index` Returns Multiple Records

```php
/** @test **/
public function index_should_return_a_collection_of_records()
{
    $authors = factory(\App\Author::class, 2)->create();

    $this->get('/authors', ['Accept' => 'application/json']);

    $body = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $body);
    $this->assertCount(2, $body['data']);

    foreach ($authors as $author) {
        $this->seeJson([
            'id' => $author->id,
            'name' => $author->name,
            'gender' => $author->gender,
            'biography' => $author->biography,
            'created' => $author->created_at->toIso8601String(),
            'updated' => $author->updated_at->toIso8601String(),
        ]);
    }
}
```

The test is nearly identical to the same test for the `/books` route. Next, write the integration and get the test passing:

Returning a Collection of Author Records

```php
<?php

namespace App\Http\Controllers;

use App\Author;
use App\Transformer\AuthorTransformer;

class AuthorsController extends Controller
{
    public function index()
    {
        return $this->collection(
            Author::all(),
            new AuthorTransformer()
        );
    }
}
```

We are starting to reap the benefits of our Fractal integration and returning a consistent response takes no effort!

Run Tests for the AuthorsController

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (31 tests, 156 assertions)
```

10.2: The GET /authors/{id} Endpoint

We will continue with our quick pace and knock out the first version of the `GET /authors/{id}` route. In this route, we want to allow including an author's books, so I will show you a way to easily accomplish that with Fractal.

A Basic Response

Let's start with testing for a basic response you are familiar with from the `/books/{id}` route:

Test the `AuthorsController@show` Method

```
/** @test **/
public function show_should_return_a_valid_author()
{
    $book = $this->bookFactory();
    $author = $book->author;

    $this->get("/authors/{$author->id}", ['Accept' => 'application/json']);
    $body = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('data', $body);

    $this->seeJson([
        'id' => $author->id,
        'name' => $author->name,
        'gender' => $author->gender,
        'biography' => $author->biography,
        'created' => $author->created_at->toIso8601String(),
        'updated' => $author->updated_at->toIso8601String(),
    ]);
}

/** @test **/
public function show_should_fail_on_an_invalid_author()
{
    $this->get('/authors/1234', ['Accept' => 'application/json']);
    $this->seeStatusCode(Response::HTTP_NOT_FOUND);

    $this->seeJson([
        'message' => 'Not Found',
        'status' => Response::HTTP_NOT_FOUND
    ]);

    $body = json_decode($this->response->getContent(), true);
    $this->assertArrayHasKey('error', $body);
    $error = $body['error'];

    $this->assertEquals('Not Found', $error['message']);
    $this->assertEquals(Response::HTTP_NOT_FOUND, $error['status']);
}
```

We cheated a little and added two tests; but everything in these tests should be familiar to you already. The implementation for the `AuthorsController@show` method should cover both tests and get back to green:

Implement a Passing `AuthorsController@show` Method

```
public function show($id)
{
    return $this->item(
        Author::findOrFail($id),
        new AuthorTransformer()
    );
}
```

Including Other Models in the Response

Fractal provides a way to build responses for relationships between transformers. This will allow us to load the `Book` data for an author without much effort on our part, and provide consistent behavior across our API. You can provide these associations by default, or optionally include them.

How should we go about implementing the inclusion of optional data in our API? A common strategy that we are going to use is including a query string parameter to instruct the API to provide optional extra data:

Our Query String Used to Include Extra Data

```
http://bookr.app/authors/1?include=books
```

In order for Fractal to include `Book` data we need to add a few things to `app/Http/Response/FractalResponse.php`. We need a way to get the request query string param `include` and pass its value to fractal. One way to accomplish this is to pass an instance of `Illuminate\Http\Request` as a dependency of `FractalResponse`.

First, let's write our tests for updating the `FractalResponse` to include the `Request` dependency. We will start by adding the `Illuminate\Http\Request` dependency throughout the `tests/app/Http/Response/FractalResponseTest.php` file. The `FractalResponseTest` class should look like this when you are done:

Full FractalResponseTest Source

```php
<?php

namespace Tests\App\Http\Response;

use TestCase;
use Mockery as m;
use League\Fractal\Manager;
use Illuminate\Http\Request;
use App\Http\Response\FractalResponse;
use League\Fractal\Serializer\SerializerAbstract;

class FractalResponseTest extends TestCase
{
    /** @test **/
    public function it_can_be_initialized()
    {
        $manager = m::mock(Manager::class);
        $serializer = m::mock(SerializerAbstract::class);
        $request = m::mock(Request::class);

        $manager
            ->shouldReceive('setSerializer')
            ->with($serializer)
            ->once()
            ->andReturn($manager);
        $fractal = new FractalResponse($manager, $serializer, $request);
        $this->assertInstanceOf(FractalResponse::class, $fractal);
    }

    /** @test **/
    public function it_can_transform_an_item()
    {
        // Request
        $request = m::mock(Request::class);

        // Transformer
        $transformer = m::mock('League\Fractal\TransformerAbstract');

        // Scope
        $scope = m::mock('League\Fractal\Scope');
        $scope
            ->shouldReceive('toArray')
            ->once()
            ->andReturn(['foo' => 'bar']);

        // Serializer
        $serializer = m::mock('League\Fractal\Serializer\SerializerAbstract');

        $manager = m::mock('League\Fractal\Manager');
        $manager
            ->shouldReceive('setSerializer')
            ->with($serializer)
            ->once();

        $manager
            ->shouldReceive('createData')
            ->once()
            ->andReturn($scope);

        $subject = new FractalResponse($manager, $serializer, $request);
        $this->assertInternalType(
            'array',
            $subject->item(['foo' => 'bar'], $transformer)
        );
    }

    /** @test **/
    public function it_can_transform_a_collection()
    {
        $data = [
            ['foo' => 'bar'],
            ['fizz' => 'buzz'],
        ];

        // Request
        $request = m::mock(Request::class);

        // Transformer
        $transformer = m::mock('League\Fractal\TransformerAbstract');

        // Scope
        $scope = m::mock('League\Fractal\Scope');
        $scope
            ->shouldReceive('toArray')
            ->once()
            ->andReturn($data);

        // Serializer
        $serializer = m::mock('League\Fractal\Serializer\SerializerAbstract');

        $manager = m::mock('League\Fractal\Manager');
        $manager
            ->shouldReceive('setSerializer')
            ->with($serializer)
            ->once();

        $manager
            ->shouldReceive('createData')
            ->once()
            ->andReturn($scope);

        $subject = new FractalResponse($manager, $serializer, $request);
        $this->assertInternalType(
            'array',
            $subject->collection($data, $transformer)
        );
    }
}
```

We've imported the `Illuminate\Http\Response` class and included it throughout the file in order to initialize an instance with the response. Now we need to update our `FractalResponse` class to accept the Illuminate `Request` instance in the constructor:

Update FractalResponse to Accept a Request Instance (partial source)

```php
<?php

namespace App\Http\Response;

use League\Fractal\Manager;
use League\Fractal\Resource\Item;
use League\Fractal\TransformerAbstract;
use League\Fractal\Resource\Collection;
use League\Fractal\Resource\ResourceInterface;
use League\Fractal\Serializer\SerializerAbstract;
use Illuminate\Http\Request;

class FractalResponse
{
    /**
     * @var Manager
     */
    private $manager;

    /**
     * @var SerializerAbstract
     */
    private $serializer;

    /**
     * @var Request
     */
    private $request;

    public function __construct(
        Manager $manager,
        SerializerAbstract $serializer,
        Request $request
    ) {
        $this->manager = $manager;
        $this->serializer = $serializer;
        $this->manager->setSerializer($serializer);
        $this->request = $request;
    }
    // ...
}
```

The code snippet looks like a lot of code; all that is happening here is importing the `Illuminate\Http\Response` class, type hinting the constructor argument, and assigning the request. The `$request` parameter will come from the Service Container.

If you run the test suite now you will get lots of failures because we've updated our FractalResponse constructor with the `Response` instance. In order to get back to green we need to update the `app/Providers/FractalServiceProvider.php` file to pass the response instance from the container into our `FractalResponse` instance:

Adding the Request Dependency to the FractalResponse Service

```
public function register()
{
    // Bind the DataArraySerializer to an interface contract
    $this->app->bind(
        'League\Fractal\Serializer\SerializerAbstract',
        'League\Fractal\Serializer\DataArraySerializer'
    );

    $this->app->bind(FractalResponse::class, function ($app) {
        $manager = new Manager();
        $serializer = $app['League\Fractal\Serializer\SerializerAbstract'];

        return new FractalResponse($manager, $serializer, $app['request']);
    });

    $this->app->alias(FractalResponse::class, 'fractal');
}
```

The service container has the service `$app['request']`, which is a service that represents the current request. We simply pass the service into the `FractalResponse` constructor and now our tests should be passing again.

Run the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (33 tests, 171 assertions)
```

Now that we have the `$app['request']` instance in our `FractalResponse` service we are ready to write a method in the `FractalResponse` class that parses request includes with the following requirements:

- If an "include" string is passed, use it to parse includes
- If a parameter is *not* passed, use the URL `?include=` query param

The requirements allow us to manually pass the includes, but we will commonly rely on the `?include=` query string parameter.

The tests should look something like the following:

Tests for the `FractalResponse::parseIncludes()` Method

```php
/** @test **/
public function it_should_parse_passed_includes_when_passed()
{
    $serializer = m::mock(SerializerAbstract::class);

    $manager = m::mock(Manager::class);
    $manager->shouldReceive('setSerializer')->with($serializer);
    $manager
        ->shouldReceive('parseIncludes')
        ->with('books');

    $request = m::mock(Request::class);
    $request->shouldNotReceive('query');

    $subject = new FractalResponse($manager, $serializer, $request);
    $subject->parseIncludes('books');
}

/** @test **/
public function it_should_parse_request_query_includes_with_no_arguments()
{
    $serializer = m::mock(SerializerAbstract::class);
    $manager = m::mock(Manager::class);
    $manager->shouldReceive('setSerializer')->with($serializer);
    $manager
        ->shouldReceive('parseIncludes')
        ->with('books');

    $request = m::mock(Request::class);
    $request
        ->shouldReceive('query')
        ->with('include', '')
        ->andReturn('books');

    (new FractalResponse($manager, $serializer, $request))->parseIncludes();
}
```

We have tested both outlined scenarios with mockery. The first test ensures that the passed parameter is used and that the request instance `query()` method is not called. In the second example we call `parseIncludes()` with no arguments and assert that the request object's `query` method is called and returns a value.

The `AuthorsTransformer` will now optionally include the books associated with an author when the request query string contains `/authors/{id}?include=books`. This is where our eager-loading call in the controller is a good idea: instead of each individual book requiring an extra query, all the author's books are retrieved in a single query.

We are now ready to write the implementation in our `FractalResponse` service:

Implement the FractalResponse::parseIncludes()‘ Method

```
/**
 * Get the includes from the request if none are passed.
 *
 * @param null $includes
 */
public function parseIncludes($includes = null)
{
    if (empty($includes)) {
        $includes = $this->request->query('include', '');
    }

    $this->manager->parseIncludes($includes);
}
```

We've defined the `parseIncludes()` method right below the `__construct()` The method checks to see if the `$includes` parameter has a non-empty value. If it *is* empty we assign `$includes` to the request query param `?include`. The second parameter in `$this->request->query('include', '')` is the default if `include` doesn't exist.

Let's run the `FractalResponseTest` after adding tests and the implementation and see where things are at:

Running the FractalResponse Tests

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (35 tests, 177 assertions)
```

The `FractalResponse` class can now accept and `?include=` query parameter and allows the Fractal manager class to parse the passed includes.

The other side of the includes functionality is the `AuthorTransformer` class and making sure it can transform the books associated with the author.

If you look at the transformers[96] documentation (see "Including Data") you will see two class properties that transformers can define: `$availableIncludes` and `$defaultIncludes`. The `$availableIncludes` is used to *optionally* include other transformers, and the `$defaultIncludes` array automatically includes other transformers. We will use opt to use `$availableIncludes` to include related `Book` records on demand.

Test for Transforming an Author's Books

```
/** @test **/
public function it_can_transform_related_books()
{
    $book = $this->bookFactory();
    $author = $book->author;

    $data = $this->subject->includeBooks($author);
    $this->assertInstanceOf(\League\Fractal\Resource\Collection::class, $data);
}
```

In order to get our new failing test passing we need to define the `AuthorTransformer::includeBooks()` in the file `app/Transformers/AuthorTransformer.php`. Define the following at the top of the `AuthorTransformer` class:

[96] http://fractal.thephpleague.com/transformers/

The AuthorsTransformer::includeBooks() method

```php
protected $availableIncludes = [
    'books'
];

public function includeBooks(Author $author)
{
    return $this->collection($author->books, new BookTransformer());
}
```

Our transformer should be passing now:

Running All Tests for AuthorTransformer

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (36 tests, 178 assertions)
```

We have one task remaining for this feature—need to wire it all up in the controller. First let's create a test to optionally include books in the `/author/{id}` response in our `tests/app/Http/Controllers/AuthorsController.php` file:

Test Optionally Including Books

```php
/** @test **/
public function show_optionally_includes_books()
{
    $book = $this->bookFactory();
    $author = $book->author;

    $this->get(
        "/authors/{$author->id}?include=books",
        ['Accept' => 'application/json']
    );

    $body = json_decode($this->response->getContent(), true);

    $this->assertArrayHasKey('data', $body);
    $data = $body['data'];
    $this->assertArrayHasKey('books', $data);
    $this->assertArrayHasKey('data', $data['books']);
    $this->assertCount(1, $data['books']['data']);

    // See Author Data
    $this->seeJson([
        'id' => $author->id,
        'name' => $author->name,
    ]);

    // Test included book Data (the first record)
    $actual = $data['books']['data'][0];
    $this->assertEquals($book->id, $actual['id']);
    $this->assertEquals($book->title, $actual['title']);
    $this->assertEquals($book->description, $actual['description']);
    $this->assertEquals(
        $book->created_at->toIso8601String(),
        $actual['created']
    );
    $this->assertEquals(
        $book->updated_at->toIso8601String(),
        $actual['updated']
    );
}
```

Our test is long, but easy to understand. The test looks similar to the `show_should_return_a_valid_author` test, but we don't need to assert as many author keys in the response because it's already covered. We only do enough to know the author is represented in the response and focus on testing that the books associated with the author are included.

The test will fail if you run it because we haven't told fractal about the `?include=books` part of our test yet, so fractal doesn't know we want to include books.

Optionally Including Books Test Failure

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\AuthorsControllerTest::show_optionally_includes_bo\
oks
Failed asserting that an array has the key 'books'.
```

```
/home/vagrant/Code/bookr/tests/app/Http/Controllers/AuthorsControllerTest.php:96

FAILURES!
Tests: 37, Assertions: 180, Failures: 1.
```

It would be nice if our controllers just automatically called the `parseIncludes()` method we build earlier because we intend to use the `?includes` query string parameter to include related entities. To do this, we can call `parseIncludes()` in the base controller `App\Http\Controllers\Controller.php` where we assign `$this->fractal` in the constructor.

Calling `parseIncludes()` in the Base Controller

```php
public function __construct(FractalResponse $fractal)
{
    $this->fractal = $fractal;
    $this->fractal->parseIncludes();
}
```

The base controller is calling `FractalManager::parseIncludes()` without passing any arguments and will assign any values in the `?include=` query params. You can include multiple with `?include=books,another`. Our feature should be passing at this point.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (37 tests, 190 assertions)
```

After we put in some hard work our feature is working! An example response might look like this:

Sample Response for `AuthorsController@show` With Books

```json
{
    "data":{
        "id":1,
        "name":"Jane Doe",
        "gender":"female",
        "biography":"Hello World",
        "created":"2015-11-25T00:13:00+0000",
        "updated":"2015-11-25T00:13:00+0000",
        "books":{
            "data":[
                {
                    "id":1,
                    "title":"Ab beatae dignissimos laudantium aut quod beatae",
                    "description":"Non reprehenderit ut pariatur. Voluptate magn\
i nam ea modi dolores rerum. Molestiae eaque et sunt et.",
                    "author":"Jane Doe",
                    "created":"2015-11-25T00:13:00+0000",
                    "updated":"2015-11-25T00:13:00+0000"
                }
            ]
        }
    }
}
```

With minimal effort we were able to load related model data into our response in a consistent fashion. Any time we include related entities consumers can expect the same data patterns and keys.

10.3: The POST /authors Endpoint

The last section had many moving parts, time to slow it down and work on a more focused `POST /authors` endpoint. We've covered creating a new resource in the `POST /books` endpoint so most of this will be good review and practice. If you recall the steps taken in the `BooksController::store()` method they were as follows:

- Validate the POST data
- Create a new resource, in this case a new Author
- Respond with the new resource and a `201` created status code
- Provide a `Location` header with the new location of the resource

We will start by creating a new resource and responding with a `201`, followed by adding on validation and the `Location` header.

First we are ready to define a successful POST request test in `tests/app/Http/Controllers/AuthorsControllerTest.php`:

Test Creating a New Author

```php
/** @test **/
public function store_can_create_a_new_author()
{
    $postData = [
        'name' => 'H. G. Wells',
        'gender' => 'male',
        'biography' => 'Prolific Science-Fiction Writer',
    ];

    $this->post('/authors', $postData, ['Accept' => 'application/json']);

    $this->seeStatusCode(201);
    $data = $this->response->getData(true);
    $this->assertArrayHasKey('data', $data);
    $this->seeJson($postData);

    $this->seeInDatabase('authors', $postData);
}
```

Run the `store_can_create_a_new_author` Test

```
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\AuthorsControllerTest::store_can_create_a_new_auth\
or
Failed asserting that 404 matches expected 201.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/AuthorsControllerTest.php:132

FAILURES!
Tests: 38, Assertions: 191, Failures: 1.
```

Here is the initial version of the `AuthorsController@store` method:

AuthorsController with @store Method Added

```php
<?php

namespace App\Http\Controllers;

use App\Author;
use App\Transformer\AuthorTransformer;
use Illuminate\Http\Request;

class AuthorsController extends Controller
{
    public function index()
    {
        return $this->collection(Author::all(), new AuthorTransformer());
    }

    public function show($id)
    {
        return $this->item(
            Author::findorFail($id),
            new AuthorTransformer()
        );
    }

    public function store(Request $request)
    {
        $author = Author::create($request->all());
        $data = $this->item($author, new AuthorTransformer());

        return response()->json($data, 201);
    }
}
```

Running Tests After Defining the `@store` Method

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (38 tests, 197 assertions)
```

The next feature we are going to add is validation. For the BooksController we separated our controller validation tests into a separate file. Feel free to do that for the AuthorsController validation tests, the code will still work, but we are going to add these tests in the `tests/app/Http/Controllers/AuthorsControllerTest.php` file:

Validation Test for the AuthorsController@store Method

```php
/** @test **/
public function store_method_validates_required_fields()
{
    $this->post('/authors', [],
        ['Accept' => 'application/json']);

    $data = $this->response->getData(true);

    $fields = ['name', 'gender', 'biography'];

    foreach ($fields as $field) {
        $this->assertArrayHasKey($field, $data);
        $this->assertEquals(["The {$field} field is required."], $data[$field]);
    }
}
```

Run the tests to make sure it fails and then implement the following to get the test passing:

Add Validation to `AuthorsController@store`

```php
public function store(Request $request)
{
    $this->validate($request, [
        'name' => 'required',
        'gender' => 'required',
        'biography' => 'required'
    ]);

    $author = Author::create($request->all());
    $data = $this->item($author, new AuthorTransformer());

    return response()->json($data, 201);
}
```

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (39 tests, 203 assertions)
```

Next, let's experiment and see what happens if we try to post a value for gender that does not match our enum field of `male` or `female`:

Test for Invalid Gender Data

```php
/** @test **/
public function store_invalidates_incorrect_gender_data()
{
    $postData = [
        'name' => 'John Doe',
        'gender' => 'unknown',
        'biography' => 'An anonymous author'
    ];

    $this->post('/authors', $postData, ['Accept' => 'application/json']);

    $this->seeStatusCode(422);

    $data = $this->response->getData(true);
    $this->assertCount(1, $data);
    $this->assertArrayHasKey('gender', $data);
    $this->assertEquals(
        ["Gender format is invalid: must equal 'male' or 'female'"],
        $data['gender']
    );
}
```

In our test we've specified a gender value not allowed in our enum field. We assert that we should only have one validation failure by counting the `$data` array and then assert the validation message for failed gender validation. Even though we are using an enum field, we get a `201` when we shouldn't:

Running the Gender Validation Test

```
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\AuthorsControllerTest::store_invalidates_incorrect\
_gender_data
Failed asserting that 201 matches expected 422.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/AuthorsControllerTest.php:167

FAILURES!
Tests: 40, Assertions: 204, Failures: 1.
```

We will use a `regex` rule to guarantee a valid gender to get the requirement passing:

Validating Gender in the AuthorsController@store Method

```php
public function store(Request $request)
{
    $this->validate($request, [
        'name' => 'required',
        'gender' => [
            'required',
            'regex:/^(male|female)$/i',
        ],
        'biography' => 'required'
    ], [
        'gender.regex' => "Gender format is invalid: must equal 'male' or 'femal\
e'"
    ]);

    $author = Author::create($request->all());
    $data = $this->item($author, new AuthorTransformer());

    return response()->json($data, 201);
}
```

We've added a regular expression validation rule for `gender` and changed the `gender` rules to use an array. This is because our regex contains a pipe character '|' which is what the validator uses to separate rules. Lastly, we add a custom validation message so users of the API will know what values are allowed.

Testing Gender Validation Again

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (40 tests, 207 assertions)
```

The last validation rule we will add before we complete the `POST /authors` endpoint is length validation for the `name` field:

Add Length Validation Tests for AuthorsController@store

```php
/** @test **/
public function store_invalidates_name_when_name_is_just_too_long()
{
    $postData = [
        'name' => str_repeat('a', 256),
        'gender' => 'male',
        'biography' => 'A Valid Biography'
    ];

    $this->post('/authors', $postData, ['Accept' => 'application/json']);

    $this->seeStatusCode(422);

    $data = $this->response->getData(true);
    $this->assertCount(1, $data);
    $this->assertArrayHasKey('name', $data);
    $this->assertEquals(
        ["The name may not be greater than 255 characters."],
        $data['name']
    );
}

/** @test **/
public function store_is_valid_when_name_is_just_long_enough()
{
    $postData = [
        'name' => str_repeat('a', 255),
        'gender' => 'male',
        'biography' => 'A Valid Biography'
    ];

    $this->post('/authors', $postData,
        ['Accept' => 'application/json']);

    $this->seeStatusCode(201);
    $this->seeInDatabase('authors', $postData);
}
```

The two tests check the same max validation we added for the Book's `title` column. To get these tests passing we need one-line change:

Adding Max Name Validation Rule

```
public function store(Request $request)
{
    $this->validate($request, [
        'name' => 'required|max:255',
        'gender' => [
            'required',
            'regex:/^(male|female)$/i',
        ],
        'biography' => 'required'
    ], [
        'gender.regex' => "Gender format is invalid: must equal 'male' or 'femal\
e'"
    ]);

    $author = Author::create($request->all());
    $data = $this->item($author, new AuthorTransformer());

    return response()->json($data, 201);
}
```

Running the Test Suite After Adding Max Validation

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (42 tests, 213 assertions)
```

We now have a solid endpoint for creating authors with valid data. Now we need to make sure that creating the author adds a `Location` header:

Test for the Location Header When an Author is Created

```php
/** @test **/
public function store_returns_a_valid_location_header()
{
    $postData = [
        'name' => 'H. G. Wells',
        'gender' => 'male',
        'biography' => 'Prolific Science-Fiction Writer'
    ];

    $this
        ->post('/authors', $postData,
            ['Accept' => 'application/json'])
        ->seeStatusCode(201);

    $data = $this->response->getData(true);
    $this->assertArrayHasKey('data', $data);
    $this->assertArrayHasKey('id', $data['data']);

    // Check the Location header
    $id = $data['data']['id'];
    $this->seeHeaderWithRegExp('Location', "#/authors/{$id}$#");
}
```

This test doesn't need to check every detail since another test already covers checking the response data. The test focuses on asserting the value of the `Location` header and checks for a valid id. We use our custom `seeHeaderWithRegExp` assertion to make sure the location header is correct.

Running Test Suite After Adding Location Header Test

```
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\AuthorsControllerTest::store_returns_a_valid_locat\
ion_header
Response should have the header 'Location' but does not.
Failed asserting that false is true.

/home/vagrant/Code/bookr/tests/TestCase.php:29
/home/vagrant/Code/bookr/tests/TestCase.php:45
/home/vagrant/Code/bookr/tests/app/Http/Controllers/AuthorsControllerTest.php:235

FAILURES!
Tests: 43, Assertions: 217, Failures: 1.
```

The first code update we will make is defining the `GET /authors/id` route as a named route so we can more easily generate a route URL. Modify the route in `app/Http/routes.php`:

Making the Authors Route a Named Route

```
$app->group([
    'prefix' => '/authors',
    'namespace' => 'App\Http\Controllers'
], function (\Laravel\Lumen\Application $app) {
    // ...
    $app->get('/{id:[\d]+}', [
        'as' => 'authors.show',
        'uses' => 'AuthorsController@show'
    ]);
    // ...
});
```

Now we can add the `Location` header in the `AuthorsController@store` route:

Add the Location Header in the AuthorsController

```php
public function store(Request $request)
{
    $this->validate($request, [
        'name' => 'required|max:255',
        'gender' => [
            'required',
            'regex:/^(male|female)$/i',
        ],
        'biography' => 'required'
    ], [
        'gender.regex' => "Gender format is invalid: must equal 'male' or 'femal\
e'"
    ]);

    $author = Author::create($request->all());
    $data = $this->item($author, new AuthorTransformer());

    return response()->json($data, 201, [
        'Location' => route('authors.show', ['id' => $author->id])
    ]);
}
```

Let's see if our suite is passing now:

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (43 tests, 218 assertions)
```

We have all features passing for the `POST /authors` endpoint!

10.4: The PUT /authors/{id} Endpoint

Next up is the ability to update an author. We will add the following features:

- Ability to change author name, gender, and biography
- Ability to validate name, gender, and biography
- It should not match an invalid route, ie. `/authors/foobar`
- It should 404 an invalid author

Test for Successfully Updating an Author

```php
/** @test **/
public function update_can_update_an_existing_author()
{
    $author = factory(\App\Author::class)->create();

    $requestData = [
        'name' => 'New Author Name',
        'gender' => $author->gender === 'male' ? 'female' : 'male',
        'biography' => 'An updated biography',
    ];

    $this
        ->put(
            "/authors/{$author->id}",
            $requestData,
            ['Accept' => 'application/json']
        )
        ->seeStatusCode(200)
        ->seeJson($requestData)
        ->seeInDatabase('authors', [
            'name' => 'New Author Name'
        ])
        ->notSeeInDatabase('authors', [
            'name' => $author->name
        ]);

    $this->assertArrayHasKey('data', $this->response->getData(true));
}
```

Our test is updating all the author fields and then chaining assertions together to ensure the proper status code and that the response has correct author data. Next, the test asserts that the updated record is found in the database and that the old author data is not found. Lastly, we continue to ensure our API contains a `data` key.

The minimum code needed to get the controller passing is as follows:

Write the `AuthorsController@update` Code

```php
public function update(Request $request, $id)
{
    $author = Author::findOrFail($id);

    $author->fill($request->all());
    $author->save();

    $data = $this->item($author, new AuthorTransformer());

    return response()->json($data, 200);
}
```

The code in `AuthorsController@update` should be familiar, let's see if it passes:

Running the Test Suite

```
$ phpunit

OK (44 tests, 226 assertions)
```

Next up, is validation testing. The validation rules will be the same as the `AuthorsController@store` method, so let's refactor both methods to use the same rules. With our tests passing, let's refactor the validation out of the `@store` method. We will write a `private` method at the end of the controller to deal with validation:

Custom Method for Author Validation

```php
/**
 * Validate author updates from the request.
 *
 * @param Request $request
 */
private function validateAuthor(Request $request)
{
    $this->validate($request, [
        'name' => 'required|max:255',
        'gender' => [
            'required',
            'regex:/^(male|female)$/i',
        ],
        'biography' => 'required'
    ], [
        'gender.regex' => "Gender format is invalid: must equal 'male' or 'femal\
e'"
    ]);
}
```

Nothing new in this method; we just copied the original `AuthorsController@store` validation code into a private method. Let's use the `validateAuthor()` method in the `@store` controller method:

Refactor the `AuthorsController@store` Method

```
public function store(Request $request)
{
    $this->validateAuthor($request);

    $author = Author::create($request->all());
    $data = $this->item($author, new AuthorTransformer());

    return response()->json($data, 201, [
        'Location' => route('authors.show', ['id' => $author->id])
    ]);
}
```

Let's see if our tests are still passing:

See if Tests Pass After the Refactor

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (44 tests, 226 assertions)
```

Our tests are still passing, so we can write the next validation tests for the `AuthorsController@update` method.

Write a Test for `AuthorsController@update` Validation

```php
/** @test **/
public function update_method_validates_required_fields()
{
    $author = factory(\App\Author::class)->create();
    $this->put("/authors/{$author->id}", [], ['Accept' => 'application/json']);
    $this->seeStatusCode(422);
    $data = $this->response->getData(true);

    $fields = ['name', 'gender', 'biography'];

    foreach ($fields as $field) {
        $this->assertArrayHasKey($field, $data);
        $this->assertEquals(["The {$field} field is required."], $data[$field]);
    }
}
```

Testing the Update Validation Method

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\AuthorsControllerTest::update_method_validates_req\
uired_fields
Failed asserting that 200 matches expected 422.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/AuthorsControllerTest.php:273

FAILURES!
Tests: 45, Assertions: 227, Failures: 1.
```

We can now use our refactored validation to make the new test pass:

Use the New `validateAuthor()` Method

```php
public function update(Request $request, $id)
{
    $this->validateAuthor($request);
    $author = Author::findOrFail($id);

    $author->fill($request->all());
    $author->save();

    $data = $this->item($author, new AuthorTransformer());

    return response()->json($data, 200);
}
```

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (45 tests, 233 assertions)
```

Another thing we can clean up is the duplicated test code for validating an author. Let's refactor our tests to test both the `@update` and `@store` methods at the same time. We can start by refactoring the `store_method_validates_required_fields` test in the `AuthorsControllerTest`:

Test Both Create and Update Validation Fields

```php
/** @test **/
public function validation_validates_required_fields()
{
    $author = factory(\App\Author::class)->create();
    $tests = [
        ['method' => 'post', 'url' => '/authors'],
        ['method' => 'put', 'url' => "/authors/{$author->id}"],
    ];

    foreach ($tests as $test) {
        $method = $test['method'];
        $this->{$method}($test['url'], [], ['Accept' => 'application/json']);
        $this->seeStatusCode(422);
        $data = $this->response->getData(true);

        $fields = ['name', 'gender', 'biography'];

        foreach ($fields as $field) {
            $this->assertArrayHasKey($field, $data);
            $this->assertEquals(["The {$field} field is required."], $data[$fiel\
d]);
        }
    }
}
```

Note that the method name has been updated to reflect the scope of the test after refactoring. The `$tests` array defines the outline for testing both @create and @update validation. The `foreach` is basically the same as the original test except that now the tests use the dynamic values from the `$tests` array.

All of our tests should still be passing at this point if you want to run the test suite before we start refactoring the next test. The next test we will refactor is the `store_invalidates_incorrect_-gender_data` test in the same file:

Refactor Invalid Gender Test

```
/** @test **/
public function validation_invalidates_incorrect_gender_data()
{
    $author = factory(\App\Author::class)->create();
    $tests = [
        // Create
        [
            'method' => 'post',
            'url' => '/authors',
            'data' => [
                'name' => 'John Doe',
                'biography' => 'An anonymous author'
            ]
        ],

        // Update
        [
            'method' => 'put',
            'url' => "/authors/{$author->id}",
            'data' => [
                'name' => $author->name,
                'biography' => $author->biography
            ]
        ]
    ];

    foreach ($tests as $test) {
        $method = $test['method'];
        $test['data']['gender'] = 'unknown';
        $this->{$method}($test['url'], $test['data'], ['Accept' => 'application/\
json']);

        $this->seeStatusCode(422);

        $data = $this->response->getData(true);
        $this->assertCount(1, $data);
        $this->assertArrayHasKey('gender', $data);
        $this->assertEquals(
            ["Gender format is invalid: must equal 'male' or 'female'"],
            $data['gender']
        );
    }
}
```

Let’s run our test suite to see if our tests are still passing:

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (45 tests, 245 assertions)
```

Onto refactoring the next test: the `store_invalidates_name_when_name_is_just_too_long` test which we will change the name of since it’s testing create and update.

Refactor Test When the Name is Just Too Long

```
/** @test **/
public function validation_invalidates_name_when_name_is_just_too_long()
{
    $author = factory(\App\Author::class)->create();
    $tests = [
        // Create
        [
            'method' => 'post',
            'url' => '/authors',
            'data' => [
                'name' => 'John Doe',
                'gender' => 'male',
                'biography' => 'An anonymous author'
            ]
        ],

        // Update
        [
            'method' => 'put',
            'url' => "/authors/{$author->id}",
            'data' => [
                'name' => $author->name,
                'gender' => $author->gender,
                'biography' => $author->biography
            ]
        ]
    ];

    foreach ($tests as $test) {
        $method = $test['method'];
        $test['data']['name'] = str_repeat('a', 256);

        $this->{$method}($test['url'], $test['data'], ['Accept' => 'application/\
json']);

        $this->seeStatusCode(422);

        $data = $this->response->getData(true);
        $this->assertCount(1, $data);
        $this->assertArrayHasKey('name', $data);
        $this->assertEquals(["The name may not be greater than 255 characters."]\
, $data['name']);
    }
}
```

Our tests are still passing at this point and we have one more validation-related test to refactor. Before we finish the final test, it's noticeable how much our refactor data is exactly the same in each validation test. Let's extract a method for the data at the end of the `AuthorsControllerTest`:

Extract Boilerplate Validation Data

```
/**
 * Provides boilerplate test instructions for validation.
 * @return array
 */
private function getValidationTestData()
{
    $author = factory(\App\Author::class)->create();
    return [
        // Create
        [
            'method' => 'post',
            'url' => '/authors',
            'status' => 201,
            'data' => [
                'name' => 'John Doe',
                'gender' => 'male',
                'biography' => 'An anonymous author'
            ]
        ],

        // Update
        [
            'method' => 'put',
            'url' => "/authors/{$author->id}",
            'status' => 200,
            'data' => [
                'name' => $author->name,
                'gender' => $author->gender,
                'biography' => $author->biography
            ]
        ]
    ];
}
```

The only thing to note in the new method is the `status` key which allows validation test that succeeds to test the status code for each type of operation.

Now that we have the new `getValidationTestData()` method let's drop it in to our test:

Update Gender Validation Test

```php
/** @test **/
public function validation_invalidates_incorrect_gender_data()
{
    foreach ($this->getValidationTestData() as $test) {
        $method = $test['method'];
        $test['data']['gender'] = 'unknown';
        $this->{$method}($test['url'], $test['data'], ['Accept' => 'application/\
json']);

        $this->seeStatusCode(422);

        $data = $this->response->getData(true);
        $this->assertCount(1, $data);
        $this->assertArrayHasKey('gender', $data);
        $this->assertEquals(
            ["Gender format is invalid: must equal 'male' or 'female'"],
            $data['gender']
        );
    }
}
```

Now let's try using our new method in the `store_is_valid_when_name_is_just_long_enough` test and rename it to match it's new purpose:

Refactor Test When the Name is Just Long Enough

```php
/** @test **/
public function validation_is_valid_when_name_is_just_long_enough()
{
    foreach ($this->getValidationTestData() as $test) {
        $method = $test['method'];
        $test['data']['name'] = str_repeat('a', 255);

        $this->{$method}($test['url'], $test['data'], ['Accept' => 'application/\
json']);

        $this->seeStatusCode($test['status']);
        $this->seeInDatabase('authors', $test['data']);
    }
}
```

With our refactored test using the new `getValidationTestData()` our test is straightforward. Now let's update the `validation_invalidates_name_when_name_is_just_too_long` to use our new method:

Update Test When Name is Just Too Long

```
/** @test **/
public function validation_invalidates_name_when_name_is_just_too_long()
{
    foreach ($this->getValidationTestData() as $test) {
        $method = $test['method'];
        $test['data']['name'] = str_repeat('a', 256);

        $this->{$method}($test['url'], $test['data'], ['Accept' => 'application/\
json']);

        $this->seeStatusCode(422);

        $data = $this->response->getData(true);
        $this->assertCount(1, $data);
        $this->assertArrayHasKey('name', $data);
        $this->assertEquals(["The name may not be greater than 255 characters."]\
, $data['name']);
    }
}
```

Now that validation covers both creating and updating we can remove the `update_method_validates_required_fields` test we originally started writing and then ensure the full test suite is still passing.

Run the Test Suite After Refactor

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (45 tests, 251 assertions)
```

The refactoring of the `AuthorsController` and `AuthorsControllerTest` have the code looking really clean. Our test coverage allowed us to rip out duplicate code and have confidence that things are still working.

10.5: The DELETE /authors/{id} Endpoint

The last author endpoint we will work on in this chapter is the `DELETE /authors/{id}` route. If you recall the schema design of the books table we added a foreign key constraint to the `author_id` field that will `CASCADE` when an author is deleted. This means that when our author is deleted all the books associated with that author will be deleted.

We have simple criteria for our first test:

- The author should not exist in the authors table after delete
- No books with the author's id should exist in the books table
- A 204 status code should be returned with no content when the delete succeeds

We will add our remaining tests above the private `private function getValidationTestdata()` method to keep the test organized.

Successful Delete Test

```
/** @test **/
public function delete_can_remove_an_author_and_his_or_her_books()
{
    $author = factory(\App\Author::class)->create();

    $this
        ->delete("/authors/{$author->id}")
        ->seeStatusCode(204)
        ->notSeeInDatabase('authors', ['id' => $author->id])
        ->notSeeInDatabase('books', ['author_id' => $author->id]);
}
```

And the `@destroy` implementation above the private `validateAuthor()` method:

Implement the `AuthorsController@destroy` Method

```
public function destroy($id)
{
    Author::findOrFail($id)->delete();

    return response(null, 204);
}
```

The final test ensures that a 404 response is returned when an invalid author id is passed:

Test Trying to Delete an Invalid ID

```
/** @test **/
public function deleting_an_invalid_author_should_return_a_404()
{
    $this
        ->delete('/authors/99999', [],
            ['Accept' => 'application/json'])
        ->seeStatusCode(404);
}
```

This test already passes because we used `findOurFail($id)` but we will keep it anyway just in case things change in the future. You could replace the `findOrFail()` with `find()` to see the failure if you want to experiment before replacing it with `findOrFail()`.

Let's run the test suite before concluding the chapter:

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (47 tests, 255 assertions)
```

Git Commit: Add the Authors Resource API

c32273f[97]

Conclusion

We are done with the basics of the `/authors` routes and this chapter! The biggest thing introduced in this chapter was how to include associated entity data in responses with fractal. You also got lots of practice refactoring methods and tests and making sure things still pass afterwards.

Now that we have two working entities, in the next chapter we will use existing entities to build out a book bundles feature. The chapter will also introduce how to add multiple entity associations in a many to many relationship through the API. We will also cover embedded relationships.

[97]https://bitbucket.org/paulredmond/bookr/commits/c32273f

Chapter 11: Book Bundles

In this chapter we will be building a simple book bundle implementation. A book bundle is basically a collection of books bundled together under a common theme. For example, a "Database Primer" book bundle might include books about database design, database theory, and database administration.

The outline for the API endpoints we will cover this chapter will be as follows:

Basic REST /bundles endpoints

GET	/bundles/{id}	Get individual bundle details
PUT	/bundles/{id}/books/{bookId}	Add a Book to a Bundle
DELETE	/bundles/{id}/books/{bookId}	Remove a Book from a Bundle

Our book routes are *nested REST resources*, meaning they represent books in the context of the `/bundles` functionality. We will use these nested resources to manage adding and removing books from bundles.

We won't code basic CRUD endpoints for `/bundles` in this chapter—you should be equipped with enough knowledge to complete CRUD endpoints for `/bundles` on your own. The focus of this chapter will be on new functionality we haven't covered yet.

11.1: Defining the Relationship Between Books and Bundles

The `Bundle` and `Book` model relationship will be a Many to Many[98] relationship—you may also know this relationship as "Has and Belongs to Many" (HABTM). It is reasonable that one book could be included in multiple bundles, and a bundle (as the name implies) will have many books.

In this section we will define the tables, relationships, and data needed for our tests and seed data. Once we have our schema and seed data we can start coding features.

Let's start defining the data and the relationships needed. First, we need a `bundles` table that contains data such as the bundle name.

[98]http://laravel.com/docs/eloquent-relationships#many-to-many

Create the 'bundles' Table

```
# vagrant@homestead:~/Code/bookr$
$ php artisan make:migration \
create_bundles_table --create=bundles
```

If you recall, the `--create` flag will generate the migration with `Schema::create()` in the `CreateBundlesTable::up()` method and `Schema::drop()` in the `CreateBundlesTable::down()` method.

The bundle will simply have `title` and `description` fields with the usual timestamps:

The `CreateBundlesTable` Migration

```php
<?php

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

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

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

Next our *Many to Many* relationship needs a pivot table. The convention for a Many to Many relationship pivot table will be “derived from the alphabetical order of the related model names”. In our case the convention for “Books” and “Bundles” will be “book_bundle”. We will use the convention, but you can also customize things—you might now expect—in eloquent if needed. The next migration will look like this:

Create the Migration for the ‘book_bundle’ Table

```
# vagrant@homestead:~/Code/bookr$
$ php artisan make:migration \
create_book_bundle_table --create=book_bundle
```

And the migration will look like this:

The CreateBookBundleTable Migration Class

```php
<?php

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

class CreateBookBundleTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('book_bundle', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('book_id')->unsigned();
            $table->integer('bundle_id')->unsigned();
            $table->timestamps();
        });
    }

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

The `book_id` and `bundle_id` fields correlate with the id column on the `books` and `bundles` tables, including being an `unsigned` integer. Other than that, nothing new here.

Now we can run our migrations and make sure they work as expected:

Running the Database Migrations

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh
Rolled back: 2016_01_20_052512_remove_books_authors_column
Rolled back: 2016_01_20_051118_associate_books_with_authors
Rolled back: 2016_01_20_050526_create_authors_table
Rolled back: 2015_12_29_234835_create_books_table
Migrated: 2015_12_29_234835_create_books_table
Migrated: 2016_01_20_050526_create_authors_table
Migrated: 2016_01_20_051118_associate_books_with_authors
Migrated: 2016_01_20_052512_remove_books_authors_column
Migrated: 2016_01_24_041900_create_bundles_table
Migrated: 2016_01_30_165357_create_book_bundle_table
```

Now that we have our migrations working the next step is creating a `Bundle` model so that we can define the associations:

Create the `app/Bundle.php` File

```
# vagrant@homestead:~/Code/bookr$
$ touch app/Bundle.php
```

Open the `app/Bundle.php` and define the Bundle model with the following:

The Bundle Eloquent Model

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Bundle extends Model
{
    protected $fillable = ['title', 'description'];

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

The `Bundle::books()` method defines the `Book` model as a "belongs to many" relationship. You have seen the `belongsTo()` and `hasMany` associations when we associated `Book` and `Author`, so this should already look familiar. For more information on Eloquent relationships see the Eloquent Relationships[99] documentation.

We need to update the `Book` model in `app/Book.php` to define the inverse of this relationship, which is identical:

Defining the Bundle Relationship in the Book Model

```php
public function bundles()
{
    return $this->belongsToMany(\App\Bundle::class);
}
```

The last thing we will do is define the factory so that we can create model seed data. Add the following to the end of the `database/factories/ModelFactory.php` file:

[99]http://laravel.com/docs/eloquent-relationships

Model Factory for the Bundle Model

```
$factory->define(\App\Bundle::class, function ($faker) {

    $title = $faker->sentence(rand(3, 10));

    return [
        'title' => substr($title, 0, strlen($title) - 1),
        'description' => $faker->text
    ];
});
```

Next, create a new database seeder class and seed code:

Create the `BundlesTableSeeder` Class

```
# vagrant@homestead:~/Code/bookr$
$ touch database/seeds/BundlesTableSeeder.php
```

The BundlesTableSeeder Class

```
<?php

use Illuminate\Database\Seeder;

class BundlesTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        factory(\App\Bundle::class, 5)->create()->each(function ($bundle) {
            $booksCount = rand(2, 5);
            $bookIds = [];

            while ($booksCount > 0) {
                $book = \App\Book::whereNotIn('id', $bookIds)
                    ->orderByRaw("RAND()")
                    ->first();

                $bundle->books()->attach($book);
                $bookIds[] = $book->id;
                $booksCount--;
            }
        });
    }
}
```

The Bundles seeder class creates 5 bundles and then loops through each one. In the callback, a random amount of books (between 2 and 5) will be defined with `rand(2, 5)`. The `while` loop finds a random book in the database that has not been used yet with the `whereNotIn('id', $bookIds)` call and returns the first result. Next, the book is attached to the current bundle, added to the ignore array so the book will not be in the same bundle twice.

We need to add the new `BundlesTableSeeder` class to the `database/seeds/DatabaseSeeder.php` file:

Add the BundlesTableSeeder to the DatabaseSeeder

```
/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    Model::unguard();

    $this->call(BooksTableSeeder::class);
    $this->call(BundlesTableSeeder::class);

    Model::reguard();
}
```

We are ready to run the migrations again and seed the new bundle data.

Migrate and Seed Application Data

```
# vagrant@homestead:~/Code/bookr$
$ composer dump-autoload
$ php artisan migrate:refresh
...
$ php artisan db:seed
Seeded: BooksTableSeeder
Seeded: BundlesTableSeeder
```

Remember that when you add new seeder classes you need to update the composer autoloader so artisan can find the seeder class. If all went well you should have some seed data in the `bundles` table and `book_bundle` table.

We have the correct data model and associations to move onto our next endpoint, which is returning an individual bundle with related books.

11.2: The GET /bundles/{id} Endpoint

With our data and model relationships in place, we are ready to start working on the individual bundle response. Our API endpoint (`/bundles/{id}`) response resembles the following JSON:

Example Bundle Response

```
{
    "data":{
        "id":1,
        "title":"Eveniet numquam quos doloribus nemo dolore culpa voluptatem nob\
is in omnis aut",
        "description":"Perferendis minus omnis accusantium reiciendis totam. Quo\
 autem ratione amet facere quam. Iure sunt qui odio sint.",
        "created":"2015-12-06T03:14:58+0000",
        "updated":"2015-12-06T03:14:58+0000",
        "books":{
            "data":[
                {
                    "id":6,
                    "title":"Et ipsam facilis rerum expedita doloribus quasi ali\
quam numquam provident rerum mollitia",
                    "description":"Itaque cumque est et vitae voluptatibus ad. R\
erum repellendus consequatur labore eum nostrum. Ut et ut voluptate maiores.",
                    "author":"Angelina Doyle",
                    "created":"2015-12-06T03:14:58+0000",
                    "updated":"2015-12-06T03:14:58+0000"
                },
                {
                    "id":33,
                    "title":"Dolorem at cum esse",
                    "description":"Commodi et consequuntur quia culpa. Totam ear\
um ad tempore cumque dolor. In ratione voluptas quisquam et est quaerat rem.",
                    "author":"Rylan Lind II",
                    "created":"2015-12-06T03:14:58+0000",
                    "updated":"2015-12-06T03:14:58+0000"
                }
            ]
        }
    }
}
```

We are ready to create our `BundleTransformer` class and tests based on the example response.

Create the BundleTransformer Class and Test

```
# vagrant@homestead:~/Code/bookr$
$ touch tests/app/Transformer/BundleTransformerTest.php
$ touch app/Transformer/BundleTransformer.php
```

Test the Bundle Transformer

```
<?php

namespace Tests\App\Transformer;

use TestCase;
use App\Transformer\BundleTransformer;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class BundleTransformerTest extends TestCase
{
    use DatabaseMigrations;

    /**
     * @var BundleTransformer
     */
    private $subject;

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

        $this->subject = new BundleTransformer();
    }

    /** @test **/
    public function it_can_be_initialized()
    {
        $this->assertInstanceOf(
            BundleTransformer::class,
            $this->subject
        );
    }

    /** @test **/
    public function it_can_transform_a_bundle()
    {
        $bundle = factory(\App\Bundle::class)->create();

        $actual = $this->subject->transform($bundle);

        $this->assertEquals($bundle->id, $actual['id']);
        $this->assertEquals($bundle->title, $actual['title']);
        $this->assertEquals(
            $bundle->description,
            $actual['description']
        );
        $this->assertEquals(
            $bundle->created_at->toIso8601String(),
            $actual['created']
        );
        $this->assertEquals(
            $bundle->updated_at->toIso8601String(),
            $actual['updated']
        );
    }

    /** @test **/
```

```php
    public function it_can_transform_related_books()
    {
        $bundle = $this->bundleFactory();

        $data = $this->subject->includeBooks($bundle);
        $this->assertInstanceOf(
            \League\Fractal\Resource\Collection::class,
            $data
        );
        $this->assertInstanceOf(
            \App\Book::class,
            $data->getData()[0]
        );
        $this->assertCount(2, $data->getData());
    }
}
```

We have included all the tests needed to cover the `BundleTransformer`; this test is just like the other transformer tests we've already covered. If might have noticed the call to `$this->bundleFactory()` in the `it_can_transform_related_books` test, which we haven't written yet. You might have also noticed the `setUp()` method used to initialize the subject of our tests as `$this->subject` which is a convenient way to set up the test subject before of each test.

We are ready to write the actual transformer class, write the `bundleFactory()` method, and get tests back to green. Next we will write the `BundleTransformer` implementation in `app/Transformer/BundleTransformer.php`:

Implement the BundleTransformer Class

```php
<?php

namespace App\Transformer;

use App\Bundle;
use League\Fractal\TransformerAbstract;

/**
 * Class BundleTransformer
 * @package App\Transformer
 */
class BundleTransformer extends TransformerAbstract
{
    protected $defaultIncludes = ['books'];

    /**
     * Include a bundle's books
     * @param Bundle $bundle
     * @return \League\Fractal\Resource\Collection
     */
    public function includeBooks(Bundle $bundle)
    {
        return $this->collection($bundle->books, new BookTransformer());
    }

    /**
     * Transform a bundle
     *
     * @param Bundle $bundle
     * @return array
     */
    public function transform(Bundle $bundle)
    {
        return [
            'id' => $bundle->id,
            'title' => $bundle->title,
            'description' => $bundle->description,
            'created' => $bundle->created_at->toIso8601String(),
            'updated' => $bundle->updated_at->toIso8601String(),
        ];
    }
}
```

The `BundleTransformer` will always include books, therefore this transformer has `books` in the `$defaultIncludes` array.

Next, let's define the `bundleFactory()` method in the base `tests/TestCase.php` file right after the `bookFactory()` method in order to easily create bundle records in the database for testing purposes.

Define the `bundleFactory()` Method in the `TestCase` Class

```php
/**
 * Convenience method for creating a book bundle
 *
 * @param int $count
 * @return mixed
 */
protected function bundleFactory($bookCount = 2)
{
    if ($bookCount <= 1) {
        throw new \RuntimeException('A bundle must have two or more books!');
    }

    $bundle = factory(\App\Bundle::class)->create();
    $books = $this->bookFactory($bookCount);

    $books->each(function ($book) use ($bundle) {
        $bundle->books()->attach($book);
    });

    return $bundle;
}
```

The `bundleFactory()` method first makes sure that a `$bookCount` of at least 2 is passed, otherwise it will throw a `\RuntimeException`. Next, a bundle is created in the test database. We use the `TestCase::bookFactory()` method to create multiple books, and then we loop through each book and attach it to our bundle. Finally we return the bundle.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (48 tests, 257 assertions)
```

With the `BundleTransformer` class in place we can create test file for the Bundle controller at `tests/app/Http/Controllers/BundlesControllerTest.php` with the first failing test for the `BundlesController@show` method.

Create the BundlesController and Test File

```
# vagrant@homestead:~/Code/bookr$
$ touch app/Http/Controllers/BundlesController.php
$ touch tests/app/Http/Controllers/BundlesControllerTest.php
```

Writing the Test for the `BundlesController@show` Method

```php
<?php

namespace Tests\App\Http\Controllers;

use TestCase;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class BundlesControllerTest extends TestCase
{
    use DatabaseMigrations;

    /** @test **/
    public function show_should_return_a_valid_bundle()
    {
        $bundle = $this->bundleFactory();

        $this->get("/bundles/{$bundle->id}", ['Accept' => 'application/json']);
        $this->seeStatusCode(200);
        $body = $this->response->getData(true);

        $this->assertArrayHasKey('data', $body);
        $data = $body['data'];

        // Check bundle properties exist in the response
        $this->assertEquals($bundle->id, $data['id']);
        $this->assertEquals($bundle->title, $data['title']);
        $this->assertEquals($bundle->title, $data['title']);
        $this->assertEquals(
            $bundle->description,
            $data['description']
        );
        $this->assertEquals(
            $bundle->created_at->toIso8601String(),
            $data['created']
        );
        $this->assertEquals(
            $bundle->updated_at->toIso8601String(),
            $data['updated']
        );

        // Check that book data is in the response
        $this->assertArrayHasKey('books', $data);
        $books = $data['books'];

        // Check that two books exist in the response
        $this->assertArrayHasKey('data', $books);
        $this->assertCount(2, $books['data']);

        // Verify keys for one book...
        $this->assertEquals(
            $bundle->books[0]->title,
            $books['data'][0]['title']
        );
        $this->assertEquals(
            $bundle->books[0]->description,
            $books['data'][0]['description']
        );
        $this->assertEquals(
            $bundle->books[0]->author->name,
            $books['data'][0]['author']
        );
        $this->assertEquals(
            $bundle->books[0]->created_at->toIso8601String(),
            $books['data'][0]['created']
        );
        $this->assertEquals(
            $bundle->books[0]->updated_at->toIso8601String(),
            $books['data'][0]['updated']
        );
    }
}
```

The test is long but simple. We get the bundle response and go through all the data to make sure our bundle endpoint contains all the response data. The controller isn't complicated but we want to make sure we test all the expected data. In fact, the implementation is only a few lines.

The Initial Version of the BundlesController

```php
<?php

namespace App\Http\Controllers;

use App\Bundle;
use App\Transformer\BundleTransformer;

/**
 * Class BundlesController
 * @package App\Http\Controllers
 */
class BundlesController extends Controller
{
    public function show($id)
    {
        $bundle = Bundle::findOrFail($id);
        $data = $this->item($bundle, new BundleTransformer());

        return response()->json($data);
    }
}
```

You've seen the `@show` method a few times. All we need to do at the moment is find the bundle and run the response through Fractal. The `BundleTransformer` automatically includes the bundle's books in the response. We still need to define a `BundlesController@show` route in the `app/Http/routes.php` file before our tests will pass again; might as well take the opportunity to write all the routes for this chapter.

Add the `/bundles` Routes

```php
$app->group([
    'prefix' => '/bundles',
    'namespace' => 'App\Http\Controllers'
], function (\Laravel\Lumen\Application $app) {

    $app->get('/{id:[\d]+}', [
        'as' => 'bundles.show',
        'uses' => 'BundlesController@show'
    ]);

    $app->put(
        '/{bundleId:[\d]+}/books/{bookId:[\d]+}',
        'BundlesController@addBook'
    );

    $app->delete(
        '/{bundleId:[\d]+}/books/{bookId:[\d]+}',
        'BundlesController@removeBook'
    );
});
```

With the routes defined we are ready to see if our test suite is fully passing again:

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (49 tests, 273 assertions)
```

11.3: Adding a Book to a Bundle

Up to this point we have relied on seed data and factories to associate bundles and books. We need an API endpoint to define the relationship for books contained within a bundle. We will write the API to allow adding and removing a single book per request. Adding a book will use the `PUT /bundles/{bundleId:[\d]+}/books/{bookId:[\d]+}` nested route.

Lets start by writing the failing test for adding a book to an existing bundle in the `tests/app/Http/Controllers/BundlesControllerTest.php` file:

Test Adding a Book to a Bundle

```
/** @test **/
public function addBook_should_add_a_book_to_a_bundle()
{
    $bundle = factory(\App\Bundle::class)->create();
    $book = $this->bookFactory();

    // Bundle should not have any associated books yet
    $this->notSeeInDatabase('book_bundle', ['bundle_id' => $bundle->id]);

    $this->put("/bundles/{$bundle->id}/books/{$book->id}", [],
        ['Accept' => 'application/json']);

    $this->seeStatusCode(200);

    $dbBundle = \App\Bundle::with('books')->find($bundle->id);
    $this->assertCount(1, $dbBundle->books,
        'The bundle should have 1 associated book');

    $this->assertEquals(
        $dbBundle->books()->first()->id,
        $book->id
    );

    $body = $this->response->getData(true);

    $this->assertArrayHasKey('data', $body);
    // Ensure the book id is in the response.
    $this->assertArrayHasKey('books', $body['data']);
    $this->assertArrayHasKey('data', $body['data']['books']);

    // Make sure the book is in the response
    $books = $body['data']['books'];
    $this->assertEquals($book->id, $books['data'][0]['id']);
}
```

Our test creates a bundle and a book separately because we don't want them associated and we assert that the new bundle does not have any books. We associate a book with the bundle with a `PUT` request and then get the bundle from the database to ensure the book is now associated with the bundle. Lastly, the test ensures the response contains the newly associated book and has the expected response.

The Initial Method for Adding a Book

```
/**
 * @param int $bundleId
 * @param int $bookId
 * @return \Illuminate\Http\JsonResponse
 */
public function addBook($bundleId, $bookId)
{
    $bundle = \App\Bundle::findOrFail($bundleId);
    $book = \App\Book::findOrFail($bookId);

    $bundle->books()->attach($book);
    $data = $this->item($bundle, new BundleTransformer());

    return response()->json($data);
}
```

The implementation is really simple; the controller makes sure that both the bundle and book are valid records and attaches the book to the bundle with the `attach()` method.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (50 tests, 281 assertions)
```

11.4: Remove a Book from a Bundle

Now that we have a way to add a book to a bundle, we need a way to remove a book too. Removing a book from a bundle resembles adding a book and will not take much effort to test and implement.

Test for Removing a Book from a Bundle

```
/** @test **/
public function removeBook_should_remove_a_book_from_a_bundle()
{
    $bundle = $this->bundleFactory(3);
    $book = $bundle->books()->first();

    $this->seeInDatabase('book_bundle', [
        'book_id' => $book->id,
        'bundle_id' => $bundle->id
    ]);

    $this->assertCount(3, $bundle->books);

    $this
        ->delete("/bundles/{$bundle->id}/books/{$book->id}")
        ->seeStatusCode(204)
        ->notSeeInDatabase('book_bundle', [
            'book_id' => $book->id,
            'bundle_id' => $bundle->id
        ]);

    $dbBundle = \App\Bundle::find($bundle->id);
    $this->assertCount(2, $dbBundle->books);
}
```

Let's break down the code in this test:

- Creates a bundle with three associated books
- Gets the first associated book and ensures the database association
- Asserts the bundle has 3 associated books
- Makes a `DELETE` request and checks for a `204` response
- Verifies the removed book is not associated with the bundle anymore
- Verifies the bundle only has two associated books in the database after deleting the association

The controller method is just like the `addBook()` method, except that we call `detach()` instead of `attach()` to remove the book from the bundle.

Removing a Book in the BundlesController

```
public function removeBook($bundleId, $bookId)
{
    $bundle = \App\Bundle::findOrFail($bundleId);
    $book = \App\Book::findOrFail($bookId);

    $bundle->books()->detach($book);

    return response(null, 204);
}
```

The `removeBook` method finds the bundle and book records and then detaches the association between the bundle and book. The response sends back a null response body with a `204 No Content` on success.

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (51 tests, 286 assertions)
```

Git Commit: Add the Bundles API Resource

e720e5c[100]

[100]https://bitbucket.org/paulredmond/bookr/commits/e720e5c

Conclusion

We covered how to deal with adding and remove many to many associations through API requests, in addition to basic `GET` responses for all bundles and individual bundles. You have been equipped with enough experience in this book to write tests for the remaining `/bundle` CRUD operations and then implement each one. I demonstrated one way to represent many to many associations through an API and you can now expand upon that with things like bulk adding and removing of books from a bundle.

In the next and final chapter of this book we will explore polymorphic relationships, as well as some performance optimizations to our queries for associated records.

Chapter 12: Ratings

The final chapter will focus on adding the ability to rate things. When you think of ratings you might think of book ratings, but actually, we can apply ratings to multiple things. Specifically, we will add ratings to *authors*. Using Eloquent makes it easy to apply data like ratings to multiple models (as you will see shortly) using polymorphic relationships[101].

Our ratings will be a five star rating system based on an integer value between `1` and `5`. We will be laying all the groundwork in this chapter so that adding ratings to multiple models will be almost effortless.

12.1: Database Design

Eloquent provides polymorphic associations out of the box and it's really easy to get something working quickly. Our database design will start by defining a migration for the `ratings` table. Based on the polymorphic documentation we should come up with something like this:

Create the Ratings Table Migration

```
# vagrant@homestead:~/Code/bookr$
$ php artisan make:migration \
create_ratings_table --create=ratings
Created Migration: 2015_12_12_061605_create_ratings_table
```

Ratings Table Database Migration Code

```php
<?php

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

class CreateRatingsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
```

[101]http://laravel.com/docs/eloquent-relationships#polymorphic-relations

```
     */
    public function up()
    {
        Schema::create('ratings', function (Blueprint $table) {
            $table->increments('id');
            $table->integer('value')->unsigned();
            $table->integer('rateable_id')->unsigned();
            $table->string('rateable_type');
            $table->timestamps();
        });
    }

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

Our migration defines an unsigned integer field `value` to hold the rating. The polymorphic association needs two other fields: `rateable_id` and `rateable_type`. The former is the id of the "owning" model (ie. Author) and the latter is the class name of the "owning" model (ie. `App\Author`). The rest of the migration should be familiar at this point.

With the ratings table defined, our next step is creating a new `Rating` model and update our `Author` model to support ratings. Create the `app/Rating.php` model with the following code:

The Rating Model

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Rating extends Model
{
    /**
     * @inheritdoc
```

```php
     */
    protected $fillable = ['value'];

    public function rateable()
    {
        return $this->morphTo();
    }
}
```

Notice the correlation between the method name `rateable()` and the database field name prefix `rateable_`. The `return $this->morphTo();` call is defining the polymorphic inverse relationship. We also define the `value` field as the only fillable field since the other fields will be managed by Eloquent automatically.

Next we are going to create a PHP Trait[102] that models can use to add the other end of the polymorphic relationship. Create a file at `app/Rateable.php` with the following:

The Rateable Trait

```php
<?php

namespace App;

/**
 * Trait to enable polymorphic ratings on a model.
 *
 * @package App
 */
trait Rateable
{
    public function ratings()
    {
        return $this->morphMany(Rating::class, 'rateable');
    }
}
```

When a model uses the `Rateable` trait the `ratings()` method will define a one-to-many polymorphic relationship. Meaning that the `Author` model will have many ratings and a single `Rating` will *belongTo* an `Author`. Other models can add this trait to implement ratings and this makes it easy at a glance to see that a model is "rateable".

Next let's add the new trait to the `Author` and `Book` models:

[102] http://php.net/manual/en/language.oop5.traits.php

Add the Rateable Trait

```
class Book extends Model
{
    use Rateable;

    // ...
}

class Author extends Model
{
    use Rateable;

    // ...
}
```

I’ve provided one code sample for both models since adding the trait is such a small amount of code. Now our models can store ratings in the database! We are ready to create a factory and seed data now for ratings.

First, append the following factory to the `database/factories/ModelFactory.php` file:

The Rating factory

```
$factory->define(\App\Rating::class, function ($faker) {
    return [
        'value' => rand(1, 5)
    ];
});
```

The factory randomly assigns 1-5 stars for a rating. Note that we will not use `factory()->create()` on the ratings factory. We will mostly use `factory()->make()` and then save the ratings through other models.

To use our new factory, we will refactor the `database/seeds/BooksTableSeeder.php` file to insert ratings into author and book seed data:

Add Ratings to the Books Database Seeder

```php
/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    $authors = factory(App\Author::class, 10)->create();
    $authors->each(function ($author) {
        $author->ratings()->saveMany(
            factory(App\Rating::class, rand(20, 50))->make()
        );

        $booksCount = rand(1, 5);

        while ($booksCount > 0) {
            $book = factory(App\Book::class)->make();
            $author->books()->save($book);
            $book->ratings()->saveMany(
                factory(App\Rating::class, rand(20, 50))->make()
            );
            $booksCount--;
        }
    });
}
```

Now our database seeder is creating ratings for each author, and then ratings for each book. We use `rand(20, 50)` to add between 20 and 50 ratings with `ratings()->saveMany()`. As noted earlier, we call `factory(App\Rating::class, rand(20, 50))->make()` and use the associated models to persist ratings to the database. You should be able to seed the database now with the artisan command.

Migrate and Seed the Database

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh
...
$ php artisan db:seed
Seeded: BooksTableSeeder
Seeded: BundlesTableSeeder
```

If you inspect the database you should see ratings in the `ratings` table for both authors and books. Eloquent and Laravel make it easy to model data quickly and get stuff done!

12.2: Rating an Author

The design of the author ratings API will be a nested resource specific to author ratings. The following are examples of the author rating endpoints we will be writing:

- `POST /authors/1/ratings` for adding a new rating
- `DELETE /authors/1/ratings/1` for deleting an existing author rating

Define the new author rating routes in `app/Http/routes.php` to get started:

Author Ratings Routes in `app/Http/routes.php`

```
$app->group([
    'prefix' => '/authors',
    'namespace' => 'App\Http\Controllers'
], function (\Laravel\Lumen\Application $app) {
    $app->get('/', 'AuthorsController@index');
    $app->post('/', 'AuthorsController@store');
    $app->get('/{id:[\d]+}', [
        'as' => 'authors.show',
        'uses' => 'AuthorsController@show'
    ]);
    $app->put('/{id:[\d]+}', 'AuthorsController@update');
    $app->delete('/{id:[\d]+}', 'AuthorsController@destroy');


    // Author ratings
    $app->post('/{id:[\d]+}/ratings', 'AuthorsRatingsController@store');
    $app->delete(
        '/{authorId:[\d]+}/ratings/{ratingId:[\d]+}',
        'AuthorsRatingsController@destroy'
    );
});
```

Note the `{authorId:[\d]+}` route param which is needed to differentiate with the `ratingId`—route params must be unique. We will organize author rating management in a new controller but nest the routes within the `/authors` resource routes.

Add an Author Rating

The first route we will test is adding a new rating to an author. When the Controller endpoint is complete you will expect a response similar to the following:

Example Response from `AuthorsRatingsController@store`

```
{
    "data":{
        "id":1409,
        "value":"5",
        "type":"App\\Author",
        "links":[
            {
                "rel":"author",
                "href":"http:\/\/localhost:8000\/authors\/1"
            }
        ],
        "created":"2015-12-12T16:42:24+0000",
        "updated":"2015-12-12T16:42:24+0000"
    }
}
```

Let's get to work on writing tests and a controller for our work.

Create the AuthorRatingsController and Test File

```
# vagrant@homestead:~/Code/bookr$
$ touch tests/app/Http/Controllers/AuthorsRatingsControllerTest.php
$ touch app/Http/Controllers/AuthorsRatingsController.php
```

Our first test in the `tests/app/Http/Controllers/AuthorsRatingsControllerTest.php` file will test adding a new rating to an author:

Test for Adding Author Ratings

```php
<?php

namespace Tests\App\Http\Controllers;

use TestCase;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class AuthorsRatingsControllerTest extends TestCase
{
    use DatabaseMigrations;

    /** @test **/
    public function store_can_add_a_rating_to_an_author()
    {
        $author = factory(\App\Author::class)->create();

        $this->post(
            "/authors/{$author->id}/ratings",
            ['value' => 5],
            ['Accept' => 'application/json']
        );

        $this
            ->seeStatusCode(201)
            ->seeJson([
                'value' => 5
            ])
            ->seeJson([
                'rel' => 'author',
                'href' => route('authors.show', ['id' => $author->id])
            ]);

        $body = $this->response->getData(true);
        $this->assertArrayHasKey('data', $body);

        $data = $body['data'];
        $this->assertArrayHasKey('links', $data);
    }
}
```

This test submits a rating of “5” and makes sure that the value is returned. The test also checks for a `links` key which will contain an href to the created rating resource, and an href to the author associated with this rating.

HATEOAS

For those keeping track this book hasn't talked about or addressed HATEOAS[a] (Hypermedia as the Engine of Application State) much at all, but I have provided a few examples how how you can use the `route()` helper function to make generating links between data simple.

You can leverage Fractal transformers to take care of HATEOAS data and I encourage you to learn more of the theory and good practices around writing RESTful services, including HATEOAS, if you are not familiar.

[a] https://en.wikipedia.org/wiki/HATEOAS

Before we start working on the controller, we will need another Fractal transformer for ratings. The transformer will need to be a little smarter than our previous transformers since it will transform multiple types and we want to provide additional data about a rating and how it relates to other models.

Create the new rating transformer file and acompanying test:

Create the RatingTransformer and Test Files

```
$ touch tests/app/Transformer/RatingTransformerTest.php
$ touch app/Transformer/RatingTransformer.php
```

The initial `RatingTransformerTest` class will include the basic initialization test we add for each transformer and a test to transform an author rating:

Initial RatingTransformerTest Class

```php
<?php

namespace Tests\App\Transformer;

use TestCase;
use App\Transformer\RatingTransformer;
use Illuminate\Foundation\Testing\DatabaseMigrations;

class RatingTransformerTest extends TestCase
{
    use DatabaseMigrations;

    /**
     * @var RatingTransformer
     */
    private $subject;

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

        $this->subject = new RatingTransformer();
    }

    /** @test **/
    public function it_can_be_initialized()
    {
        $this->assertInstanceOf(RatingTransformer::class, $this->subject);
    }

    /** @test **/
    public function it_can_transform_a_rating_for_an_author()
    {
        $author = factory(\App\Author::class)->create();
        $rating = $author->ratings()->save(
            factory(\App\Rating::class)->make()
        );


        $actual = $this->subject->transform($rating);

        $this->assertEquals($rating->id, $actual['id']);
        $this->assertEquals($rating->value, $actual['value']);
        $this->assertEquals($rating->rateable_type, $actual['type']);
        $this->assertEquals(
            $rating->created_at->toIso8601String(),
            $actual['created']
        );
        $this->assertEquals(
            $rating->updated_at->toIso8601String(),
            $actual['created']
        );

```

```php
        $this->assertArrayHasKey('links', $actual);
        $links = $actual['links'];
        $this->assertCount(1, $links);
        $authorLink = $links[0];

        $this->assertArrayHasKey('rel', $authorLink);
        $this->assertEquals('author', $authorLink['rel']);
        $this->assertArrayHasKey('href', $authorLink);
        $this->assertEquals(
            route('authors.show', ['id' => $author->id]),
            $authorLink['href']
        );
    }
}
```

Like all the other transformer tests we've covered, we test that the transformer can be initialized. The `it_can_transform_a_rating_for_an_author` test checks for keys and values to make sure the transformer formats data as we expect. The test also verifies the `links` property which should contain an author resource with an href to the author API entity.

Here is what our first `RatingTransformer` implementation looks like:

First Version of the RatingTransformer Implementation

```php
<?php

namespace App\Transformer;

use App\Rating;
use League\Fractal\TransformerAbstract;

/**
 * Class RatingTransformer
 * @package App\Transformer
 */
class RatingTransformer extends TransformerAbstract
{
    /**
     * Transform a Rating
     *
     * @param Rating $rating
     * @return array
     */
    public function transform(Rating $rating)
    {
        return [
            'id' => $rating->id,
            'value' => $rating->value,
            'type'  => $rating->rateable_type,
            'links' => [
                [
                    'rel' => 'author',
                    'href' => route('authors.show', ['id' => $rating->rateable_i\
d])
                ]
            ],
            'created' => $rating->created_at->toIso8601String(),
            'updated' => $rating->updated_at->toIso8601String(),
        ];
    }
}
```

We've side-stepped the failing test for `RatingsAuthorsController` that we originally wrote in this section to work on the `RatingTransformer`. We cannot run the full test suite, but the initial passing version of the `RatingTransfomer` hard-codes the author link data and our `RatingTransformer` tests should be passing now:

Run Tests for the RatingTransformer

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=RatingTransformerTest

OK (2 tests, 12 assertions)
```

Before we write the passing implementation for the `AuthorsRatingsController` let's refactor the transformer to be more dynamic and capable of transforming a rating for other entities:

Refactor the RatingTransformer to Support Multiple Models

```php
<?php

namespace App\Transformer;

use App\Rating;
use League\Fractal\TransformerAbstract;

/**
 * Class RatingTransformer
 * @package App\Transformer
 */
class RatingTransformer extends TransformerAbstract
{
    /**
     * Transform a Rating
     *
     * @param Rating $rating
     * @return array
     */
    public function transform(Rating $rating)
    {
        return [
            'id' => $rating->id,
            'value' => $rating->value,
            'type'  => $rating->rateable_type,
            'links' => [
                [
                    'rel' => $this->getModelName($rating->rateable_type),
                    'href' => $this->getModelUrl($rating)
                ]
            ],
            'created' => $rating->created_at->toIso8601String(),
            'updated' => $rating->updated_at->toIso8601String(),
        ];
    }

    /**
     * Get a human-friendly model name
     *
     * @param $rateable_type
     * @return string
     */
    private function getModelName($rateable_type)
    {
        return strtolower(preg_replace("/^App\\\/", '', $rateable_type));
    }

    /**
     * Generate a URL to the rated model resource
     *
     * @param Rating $rating
     * @return string
     */
    private function getModelUrl(Rating $rating)
    {
        $author = \App\Author::class;
        $book = \App\Author::class;

        switch ($rating->rateable_type) {
            case $author:
                $named = 'authors.show';
                break;
            case $book:
                $named = 'books.show';
                break;
            default:
                throw new \RuntimeException(sprintf(
                    'Rateable model type for %s is not defined',
                    $rating->rateable_type
                ));
        }

        return route($named, ['id' => $rating->rateable_id]);
    }
}
```

We've added two `private` methods for dealing with dynamic model data `getModelName()` and `getModelUrl()`. These methods are not *perfect* but they get the job done by providing a more human-readable `rel` type. We will have to keep adding more models to the `switch` statement in `getModelUrl()` method when we want to make something "rateable", but for now it works.

Our tests should still be passing after the refactor. In real life, you might have to deal with failures along the way, but through the power of books you get the passing code immediately ;):

Run `RatingTransformer` Tests After Refactoring

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=RatingTransformerTest

OK (2 tests, 12 assertions)
```

The `getModelUrl()` will throw an exception if it doesn't recognize the `$rating->rateable_type` so we need to write a test for that:

Test for a Thrown Exception in `RatingTranformerTest`

```
/**
 * @test
 * @expectedException \RuntimeException
 * @expectedExceptionMessage Rateable model type for Foo\Bar is not defined
 */
public function it_throws_an_exception_when_a_model_is_not_defined()
{
    $rating = factory(\App\Rating::class)->create([
        'value' => 5,
        'rateable_type' => 'Foo\Bar',
        'rateable_id' => 1
    ]);

    $this->subject->transform($rating);
}
```

We directly create a rating with a fake `rateable_type` to trigger the exception. This test also uses PHPUnit test annotations[103] to ensure that `\RuntimeException` is thrown with the correct message.

Our transformer is fully passing and ready to use. We can now starting writing the first version of our controller at `app/Http/Controllers/AuthorsRatingsController.php` and get our tests passing:

[103]https://phpunit.de/manual/current/en/appendixes.annotations.html#appendixes.annotations.expectedException

The Ratings Controller

```
<?php

namespace App\Http\Controllers;

use App\Author;
use Illuminate\Http\Request;
use App\Transformer\RatingTransformer;

/**
 * Manage an Author's Ratings
 */
class AuthorsRatingsController extends Controller
{
    public function store(Request $request, $authorId)
    {
        $author = Author::findOrFail($authorId);

        $rating = $author->ratings()->create(['value' => $request->get('value')]\
);
        $data = $this->item($rating, new RatingTransformer());

        return response()->json($data, 201);
    }
}
```

The controller simply checks for a valid author and creates a new rating associated with the author. The controller returns the new rating data with a `201 created` response. The full test suite should be passing now that we've implemented the controller:

Run the Full Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (55 tests, 308 assertions)
```

For your own satisfaction, you can post a rating via the command line on ‘nix systems like so:

Add an Author Rating with cURL

```
# vagrant@homestead:~/Code/bookr$
$ php artisan migrate:refresh
$ php artisan db:seed
$ curl --data "value=5" -X POST http://bookr.app/authors/1/ratings
{
    "data":{
        "id":1188,
        "value":"5",
        "type":"App\\Author",
        "links":[
            {
                "rel":"author",
                "href":"http:\/\/bookr.app\/authors\/1"
            }
        ],
        "created":"2016-02-02T05:55:42+0000",
        "updated":"2016-02-02T05:55:42+0000"
    }
}
```

Now that tests are passing, we will write a test to ensure we get a `404` response back when the author id is invalid in `AuthorsRatingsControllerTest`:

Test Trying to Add a Rating to an Invalid Author

```php
/** @test **/
public function store_fails_when_the_author_is_invalid()
{
    $this->post('/authors/1/ratings', [], ['Accept' => 'application/json']);
    $this->seeStatusCode(404);
}
```

Since we already wrote `Author::findOrFail($authorId);` this test will pass. Be careful of tests that pass without writing additional code. To verify that our test is indeed valid, temporarily update the controller to:

Revert `AuthorsRatingsController` to Make Test Fail

```
public function store(Request $request, $authorId)
{
    $author = Author::find($authorId);

    $rating = $author->ratings()->create(['value' => $request->get('value')]);
    $data = $this->item($rating, new RatingTransformer());

    return response()->json($data, 201);
}
```

After changing the `AuthorsRatingsController::store()` method, you should get this test failure:

Test Failure After Reverting the `AuthorsRatingsController`

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\AuthorsRatingsControllerTest::store_fails_when_the\
_author_is_invalid
Failed asserting that 400 matches expected 404.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/AuthorsRatingsControllerTest\
.php:44

FAILURES!
Tests: 56, Assertions: 309, Failures: 1.
```

Our test will guard against to responding with a 404 when the database lookup fails. Revert the `AuthorsRatingsController.php` file to the following to get back to green:

Restore the Correct Store Method

```
public function store(Request $request, $authorId)
{
    $author = Author::findOrFail($authorId);

    $rating = $author->ratings()->create(['value' => $request->get('value')]);
    $data = $this->item($rating, new RatingTransformer());

    return response()->json($data, 201);
}
```

Check tests to make sure everything is working as expected:

Run the Full Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (56 tests, 309 assertions)
```

Delete an Author Rating

Our next feature will be the ability to *delete* an existing author rating. We already defined the application delete route earlier in this chapter:

The Delete Route for Author Ratings

```
$app->delete(
    '/{authorId:[\d]+}/ratings/{ratingId:[\d]+}',
    'AuthorsRatingsController@destroy'
);
```

The outline for our acceptance criteria:

- Delete should remove an existing rating from an author
- The ratings table no longer associate the rating to the author
- The rating should no longer exist in the database

Let's write the first failing test for deleting a rating from an author in the `AuthorsRatingsControllerTest`:

Test to Delete a Rating from an Author

```php
/** @test **/
public function destroy_can_delete_an_author_rating()
{
    $author  = factory(\App\Author::class)->create();
    $ratings = $author->ratings()->saveMany(
        factory(\App\Rating::class, 2)->make()
    );

    $this->assertCount(2, $ratings);

    $ratings->each(function (\App\Rating $rating) use ($author) {
        $this->seeInDatabase('ratings', [
            'rateable_id' => $author->id,
            'id' => $rating->id
        ]);
    });

    $ratingToDelete = $ratings->first();
    $this
        ->delete(
            "/authors/{$author->id}/ratings/{$ratingToDelete->id}"
        )
        ->seeStatusCode(204);

    $dbAuthor = \App\Author::find($author->id);
    $this->assertCount(1, $dbAuthor->ratings);
    $this->notSeeInDatabase(
        'ratings',
        ['id' => $ratingToDelete->id]
    );
}
```

First, we seed data for the test; data factory code should look familiar because they are basically the same as the database seeding we covered earlier in this chapter. We then check to make sure that we save 2 ratings in the database with `ratings()->saveMany()` so we can verify the rating count decreases by one after deleting one rating. Next, we loop through each rating and just double check that each rating is properly associated with the author.

With the data seeded and verifying associations, our test gets the rating to be deleted and makes the `DELETE` request and we expect a `204` status code in return. Last, we verify the rating was removed by getting the author from the database and asserting that the author only has 1 rating and the rating we deleted is no longer in the `ratings` table.

Technically, the framework takes care of the database associations checked in this test, but extra checking does not hurt and makes me feel safer that my test is accurate.

Run the Failing Test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

There was 1 failure:

1) Tests\App\Http\Controllers\AuthorsRatingsControllerTest::destroy_can_delete_a\
n_author_rating
Failed asserting that 404 matches expected 204.

/home/vagrant/Code/bookr/vendor/laravel/lumen-framework/src/Testing/CrawlerTrait\
.php:412
/home/vagrant/Code/bookr/tests/app/Http/Controllers/AuthorsRatingsControllerTest\
.php:69

FAILURES!
Tests: 57, Assertions: 313, Failures: 1.
```

With the failing test in place, let's write the first implementation of the `AuthorsRatingsController@destroy` method:

Implementation for Deleting an Author Rating

```
/**
 * @param $authorId
 * @param $ratingId
 * @return \Laravel\Lumen\Http\ResponseFactory
 */
public function destroy($authorId, $ratingId)
{
    /** @var \App\Author $author */
    $author = Author::findOrFail($authorId);
    $author
        ->ratings()
        ->findOrFail($ratingId)
        ->delete();

    return response(null, 204);
}
```

Our controller ensures the author exists and then uses the author to find the `$ratingId`. The request can fail if the `$authorId` is invalid *or* the `$ratingId` is invalid. We should write some additional tests in the `AuthorsRatingsControllerTest` class just to ensure that this method fails in the way we expect.

Test API Cannot Delete Another Author's Rating

```
/** @test **/
public function destroy_should_not_delete_ratings_from_another_author()
{
    $authors  = factory(\App\Author::class, 2)->create();
    $authors->each(function (\App\Author $author) {
        $author->ratings()->saveMany(
            factory(\App\Rating::class, 2)->make()
        );
    });

    $firstAuthor = $authors->first();
    $rating = $authors
        ->last()
        ->ratings()
        ->first();

    $this->delete(
        "/authors/{$firstAuthor->id}/ratings/{$rating->id}",
        [],
        ['Accept' => 'application/json']
    )->seeStatusCode(404);
}
```

The test creates factory data for two authors. We then grab the first author and a rating from the second author. Our delete request expects a `404` response because the rating id is invalid in the context of the author from which we try to delete a rating. This test will pass because we've already added `$author->ratings()->findOrFail($ratingId)` to the controller's `destroy` method. You can swap out the code to get the test failing.

We should also expect a `404` if the author id is not valid in the `AuthorsRatingsControllerTest`. You have already seen variations of this test multiple times in this book:

Test Expecting a 404 When the Author is Invalid

```
/** @test **/
public function destroy_fails_when_the_author_is_invalid()
{
    $this->delete(
        '/authors/1/ratings/1',
        [],
        ['Accept' => 'application/json']
    )->seeStatusCode(404);
}
```

We should be fully passing, but we'll run the test suite once more before moving on to the next topic.

Run the Full Test Suite

```
$ phpunit

OK (59 tests, 317 assertions)
```

We are done with managing author ratings, although, we did not cover all the API endpoints you might make to manage ratings in a real application. We could also provide an endpoint to do bulk operations, like removing multiple ratings with one request when it makes sense. You should be equipped with enough knowledge now to develop and test these concepts now.

12.3: Ratings in the Author API

Now that we have the database schema and basic rating management, we are going to add rating data to the `/author` API. Our feature will be to provide an API that includes an author's rating average and rating count. When building this feature we need to keep in mind *how* the ratings might be used. In the simplest form, perhaps an author page will show a five star graphical scale. The API needs to provide enough information to allow the UI to display the ratings. The consumer might need to know things like the maximum rating possible, how many people rated the author, the average rating, and the average rating as a percentage.

With that data in mind our first attempt might look like the following:

Example Author Response With Ratings

```
{
    "data":{
        "id":1,
        "name":"Roslyn Medhurst",
        "gender":"female",
        "biography":"Nemo accusantium et blanditiis.",
        "rating":{
            "average":3.32,
            "max":5,
            "percent":66.4,
            "count":56
        },
        "created":"2015-12-12T14:36:50+0000",
        "updated":"2015-12-12T14:36:50+0000"
    }
}
```

Now that we have an idea of what our API might respond with, our test plan will include:

- Test typical rating values when an author has been rated
- Test the rating data when an author has not been rated yet

We'll start by writing new tests in the `tests/app/Transformer/AuthorTransformerTest.php` class. The first test we will cover is modifying the existing `it_can_transform_an_author` test to add rating data:

Testing the AuthorTransformer Rating Data

```
/** @test **/
public function it_can_transform_an_author()
{
    $author = factory(\App\Author::class)->create();

    $author->ratings()->save(
        factory(\App\Rating::class)->make(['value' => 5])
    );

    $author->ratings()->save(
        factory(\App\Rating::class)->make(['value' => 3])
    );

    $actual = $this->subject->transform($author);

    $this->assertEquals($author->id, $actual['id']);
    $this->assertEquals($author->name, $actual['name']);
    $this->assertEquals($author->gender, $actual['gender']);
    $this->assertEquals($author->biography, $actual['biography']);
    $this->assertEquals($author->created_at->toIso8601String(), $actual['created\
']);
    $this->assertEquals($author->updated_at->toIso8601String(), $actual['created\
']);

    // Rating
    $this->assertArrayHasKey('rating', $actual);
    $this->assertInternalType('array', $actual['rating']);
    $this->assertEquals(4, $actual['rating']['average']);
    $this->assertEquals(5, $actual['rating']['max']);
    $this->assertEquals(80, $actual['rating']['percent']);
    $this->assertEquals(2, $actual['rating']['count']);
}
```

We start by adding ratings to the author data under test. To make it easy to calculate averages we add two ratings individually, and `factory()->make()` allows us to override the rating value. Next, we add assertions that the rating key exists and is an array. Last, we verify the value of each individual rating key we expect.

Our test will fail with the following error:

Run the Modified Test

```
# vagrant@homestead:~/Code/bookr$
$ phpunit --filter=it_can_transform_an_author

There was 1 failure:

1) Tests\App\Transformer\AuthorTransformerTest::it_can_transform_an_author
Failed asserting that an array has the key 'rating'.

/home/vagrant/Code/bookr/tests/app/Transformer/AuthorTransformerTest.php:48

FAILURES!
Tests: 1, Assertions: 7, Failures: 1.
```

Our implementation of this feature will update the `app/Transformer/AuthorTransformer.php` file to include the “ratings” key and do all the ratings calculations:

Implement Ratings in the `AuthorTransformer`

```php
/**
 * Transform an author model
 *
 * @param Author $author
 * @return array
 */
public function transform(Author $author)
{
    return [
        'id'        => $author->id,
        'name'      => $author->name,
        'gender'    => $author->gender,
        'biography' => $author->biography,
        'rating' => [
            'average' => (float) sprintf(
                "%.2f",
                $author->ratings->avg('value')
            ),
            'max' => (float) sprintf("%.2f", 5),
            'percent' => (float) sprintf(
                "%.2f",
                ($author->ratings->avg('value') / 5) * 100
            ),
            'count' => $author->ratings->count(),
        ],
        'created'   => $author->created_at->toIso8601String(),
        'updated'   => $author->created_at->toIso8601String(),
    ];
}
```

If you run the test suite things should be passing again:

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (59 tests, 323 assertions)
```

Our change will affect all of our responses containing an author. Unfortunately, the rating data is lazy-loaded right now, meaning that each `Author` record in the `/authors` request will result in a new query. If we are returning `100` authors in our response, that will result in `100` queries to the `ratings` table. Here is an example from the `app/storage/logs/lumen.log` file where I am outputting queries from the ORM:

Example of a Lazy-Loaded Query

```
[2015-12-17 21:20:56] lumen.INFO: select * from `ratings` where `ratings`.`ratea\
ble_id` = ? and `ratings`.`rateable_id` is not null and `ratings`.`rateable_type\
` = ? [1,"App\\Author"]
```

To get this type of logging working in your app, we are going to use the `app/Providers/AppServiceProvider.php` class to add some database logging so you can visualize the actual queries generated by Eloquent.

Add Database Logging to the AppServiceProvider

```php
<?php

namespace App\Providers;

use DB;
use Log;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    public function boot()
    {
        if (env('DB_LOGGING', false) === true) {
            DB::listen(function($sql, $bindings, $time) {
                Log::info($sql, $bindings, $time);
            });
        }
    }
}
```

We've added a listener that will log out the SQL query and bindings when the environment variable `DB_LOGGING=true` is set. To start using this listener we need to enable the `AppServiceProvider` and configure the environment variable.

To enable the `AppServiceProvider` open up `bootstrap/app.php` and look for the "Register Service Providers" section and uncomment the `AppServiceProvider` class:

Enable the AppServiceProvider

```
$app->register(App\Providers\AppServiceProvider::class);
```

Add the following to your `.env` file in the root of the project (I would also recommend adding it to the .env.example file for other developers to grab):

Enable DB Logging in .env

```
DB_LOGGING=true
```

Disable DB Logging in .env.example

```
DB_LOGGING=false
```

Chatty Logs

It's probably not a good idea to use `DB::listen()` to log database queries in production; use the DB logging feature as a development convenience to see SQL queries. You can just as easily enable MySQL's logging capabilities to get the same effect.

I prefer to toggle it on/off in development as I don't always need (or want) to see database logs while I develop.

Now that you have database logging in place, make a request to `/authors` to visualize the queries. In the next section we will work on preventing excess queries to get author rating data.

12.4: Eager Loading Ratings

After taking an aside and understanding that our transformer can create additional unnecessary (and unintentional) queries, what can we do about it? Enter eager loading[104].

The official documentation for eager loading does a great job of explaining the (potential) problem and how to avoid it. Let's update our `AuthorsController@index` method to use eager loading and check how our database logging changes compared to doing a query for each individual author. Open the `app/Http/Controllers/AuthorsController.php` file and update the index route:

Use Eager Loading on the Authors Index Route

```php
public function index()
{
    $authors = Author::with('ratings')->get();

    return $this->collection($authors, new AuthorTransformer());
}
```

If you make a request to `/authors` with database logging turned on, the request should only generate two queries:

Queries Logged with Eager Loading

```
[2015-12-18 05:33:07] lumen.INFO: select * from `authors`
[2015-12-18 05:33:07] lumen.INFO: select * from `ratings` where `ratings`.`ratea\
ble_id` in (?, ?, ?, ?, ?, ?, ?, ?, ?, ?) and `ratings`.`rateable_type` = ? [1,2\
,3,4,5,6,7,8,9,10,"App\\Author"]
```

Much better! Now our transformer is not creating unnecessary queries.

You Can Still Generate Extra Queries!

In the `AuthorTransformer` you can still create extra queries by calling the `$author->ratings()` method. For example, `$author->ratings()->avg('value')` makes an additional query even if you use eager loading to get the author and ratings. You should use `$author->ratings->avg('value')` as seen in our transformer to avoid extra queries.

At this point we should run the test suite again since we changed our code to use eager loading:

[104] http://laravel.com/docs/5.1/eloquent-relationships#eager-loading

Running the Test Suite

```
# vagrant@homestead:~/Code/bookr$
$ phpunit

OK (59 tests, 323 assertions)
```

The remaining author routes in the `AuthorsController` don't really need to use eager loading because only one record is being requested and we will not generate additional queries. Eager loading is most important when you are getting a collection of records and looping over them.

Git Commit: Add Author Ratings

ea70ca3[105]

Conclusion

In this chapter we demonstrated how to use polymorphic relationships in our API and then expose the data from our Fractal transformer. You are well-equipped to add ratings to books and then use the API to create your own frontend too! Along the way we learned about eager loading[106] and just brushed on the subject of query optimization.

[105]https://bitbucket.org/paulredmond/bookr/commits/ea70ca3
[106]http://laravel.com/docs/5.1/eloquent-relationships#eager-loading

Where To Go From Here

Congratulations! Thanks for reading and working through the whole book. The main objective of this book is two-fold: any PHP developer can pick up this book and write Lumen APIs with no previous Laravel Experience, and practicing test-driven development in an API context.

A wonderful artifact of writing this book in a test-driven manner is that I have high confidence that the code samples in this book work. I am not claiming that the book is *100% free from bugs* or full test coverage, but the *code feels solid*.

There are many things not covered in the scope of this book that may one day become a follow-up or more advanced book. For instance, we did not cover writing our APIs with multi-tenancy in mind and we did not cover authentication. This book was about building a foundation. I will try to direct you to a few recommended resources.

If you found this book useful, **pass it on by giving others a link** to https://leanpub.com/lumen-apis[107]. If you want to share this book with others in your company, meetups, newsletters, and conferences reach out to me for discount codes on Twitter @paulredmond[108].

Laravel

If you have a solid understanding of Laravel[109], you pretty much know Lumen (apart from a few API differences). My hope is that if you have not worked with Laravel, that you read through the documentation[110]. Chances are that most readers have have already at least experimented with Laravel.

Laravel is the other half of my current development toolkit. Together, Laravel and Lumen provide all the core developer tools I need to write web applications and APIs. I have the same basic workflow between writing Laravel and Lumen apps. Having the same workflow makes me feel very productive and the APIs are familiar.

Lumen has other features you can read about in the Laravel documentation, such as a queue system and scheduled jobs. Knowing that you don't have to bring in 3rd party libraries to get a queue going in Lumen or Laravel is a huge win. Lumen also benefits from some more advanced Eloquent features we did not cover in this book that you can learn about in the documentation.

[107]https://leanpub.com/lumen-apis
[108]https://twitter.com/paulredmond
[109]https://laravel.com/
[110]https://laravel.com/docs/

Laracasts

Laracasts[111] is the best resource for learning Laravel (and thus Lumen) period. At the time of writing they have a free series Laravel from Scratch[112]. The paid subscription is valuable and will give you hours of videos on Laravel, general programming, development tools, and even JavaScript.

Mockery

We used Mockery[113] to unit test certain things in this book but I can't emphasize enough to become very familiar with this library. Mockery is a must in my own unit testing toolbelt.

Guzzle

Guzzle[114] is my favorite PHP HTTP client library. When you write APIs, you need a way for other application to communicate with your API. If you haven't used Guzzle I'd recommend writing an HTTP client for the application we wrote in this book. It would be a good exercise. Sometimes your own APIs will need to communicate with other internal APIs.

[111]https://laracasts.com/
[112]https://laracasts.com/series/laravel-5-from-scratch
[113]http://docs.mockery.io/en/latest/
[114]http://docs.guzzlephp.org/