{ 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

This book is for sale at http://leanpub.com/lumen-apis

This version was published on 2016-02-10

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; 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)
```

Code Errata and Feedback

Submit errata to lumenapibook@gmail.com. Feel free to send me typos, inaccurate descriptions, code issues, praise, feedback, and code suggestions on better ways of doing something. Please don't be shy, **these things make my book better**!

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.

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

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.

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
```

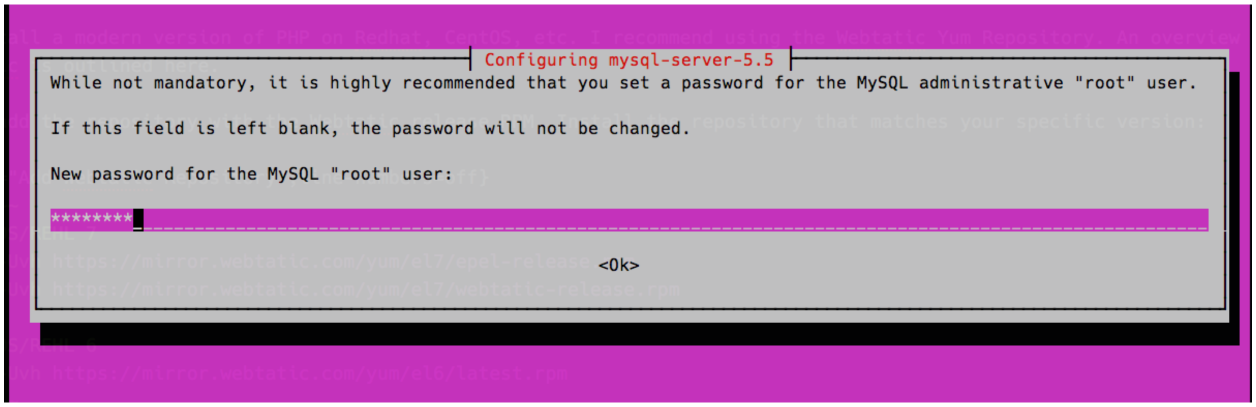
Next, install MySQL server and client packages, make the MySQL service starts on boot, and start the service manually:

[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 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[22] 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[23]. 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.

[22]https://github.com/fzaninotto/Faker
[23]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[24] 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:

[24]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[25]

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)

```php
/**
 * 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:

[25] 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[26]—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 Contract.

```php
$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.

```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 $e)
{
    return parent::render($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[27] `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.

[26]http://laravel.com/docs/master/contracts
[27]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

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[28], 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:

[28] 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[29]. 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[30] `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`.

[29]http://docs.mockery.io/en/latest/reference/partial_mocks.html
[30]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[31] 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 :).

[31] 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

```php
/** @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

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

```
/** @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

```
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

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 `ModelNotFoundException` 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.

```
/** @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[32]

[32] 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.