by Wacky Studio

BUILD an API with LARAVEL

THOMAS GAMBORG NØRGAARD

Build an API with Laravel - Sample

First published by Wacky Studio 2019

First edition

Proofreading by Sille Justesen Krogh
Editing by Christian Nørrelund

This book was professionally typeset on Reedsy.
Find out more at reedsy.com

Contents

Errata

Since this is our first book, errors will most certainly have snuck in.

You can help support the book by sending an email with any errors your might have found to hello@wackystudio.com along with the chapter and section title.

As soon as we are informed about an error, we will fix it and release an update to the book. If you have feedback, we would love to hear from you as well.

* * *

Code samples and conventions

In this book, we will reference a bunch of technical things, especially when it comes to API endpoints and code samples.

The conventions used to show an API endpoint follows the protocol below:

```
VERB: /api/endpoint?with=possible&query=params
```

The VERB part references the HTTP verb, we will touch upon this later in the book, but for now these are: **GET**, **POST**, **PUT**, **PATCH** and **DELETE**. Next, you have the endpoint itself that indicates an intention. Lastly, you have the query parameters, which are often used for sorting and such.

To give a more precise example, here is an example of a more common endpoint for querying books from a bookstore backend:

```
GET:/books?include=author
```

As you might also have noticed by now, we use bold text to emphasize

certain technical terms.

In this book, we will also show examples of a response payload in JSON. Let's take a look at how that will be shown:

```
{
 "data": [{
   "type": "books",
   "id": 1,
   "attributes": {
     "title": "Build an API with Laravel",
     "body": "Lorem ipsum",
     "created": "2019-02-01 00:00:00",
     "updated": "2019-02-01 00:00:00"
   },
   "relationships": {
     "author": {
       "data": {
         "id": 1,
         "type": "authors"
       }
     }
   }
 }],
 "included": [
   {
     "type": "authors",
     "id": 1,
     "attributes": {
       "name": "Wacky Studio"
     }
   }
 ]
}
```

Let's cover the code samples you will find in this book first. Since this

book is centered around building an API using Laravel and Laravel is developed in PHP, most code will be PHP code.

The code provided should not be considered applicable in production — it's written with the intention of teaching in mind. We strive to write code following best practices, as much as possible. You might find areas where this is not the case, but keep in mind that we also strive to write code, where it is as easy to get a technical points across as possible.

This book has been written with syntax highlighting enabled. How this is interpreted is different depending on the platform, e-readers and chosen theme. The following code is an example to see how your e-reader highlights the syntax. If you see beautiful colors you are in luck, otherwise we hope you can still follow along.

```
<?php
class Object {

   private $variable;

   public function __construct($parameter)
   {
       $this->variable = $parameter;
   }

   public function doSomething()
   {
       return 'did something';
   }
}
?>
```

* * *

Why Laravel?

There are a lot of reasons why we chose to write this book around Laravel. The biggest reason is that we use the framework in almost all of our applications and solutions. At the time of writing, we have been using Laravel for almost six years and have written over a dozen applications, varying from small to rather large sizes.

Since our primary income comes from developing applications, we want the development time to be as short and cheap as possible. We don't want to reinvent the wheel every time we start on a new project, we don't want something that is extremely hard to deploy to a server, and we also want to make something that is not a nightmare to maintain later on.

Laravel helps us deal with precisely those problems, which makes development a joy.

* * *

Prerequisites

This book is written with Laravel developers in mind. You do not have to be a super advanced and skilled developer to follow along, but bear in mind that this is not a book about the Laravel framework itself, but instead about how to write an API using Laravel.

Therefore, you will have to have a basic understanding of Laravel to keep up. We certainly recommend that you have tried writing an application in the framework before reading, and that you know what we talk about when we mention: Client, Server, Request, Response, Routes, Controllers, Eloquent or Models, Migrations, Factories, Authentication, Authorization, and Validation.

If most of these words are foreign to you, we recommend that you read up on Laravel before continuing.

We will also be using Laravel Collections heavily, so an understanding of these – especially the **map**, **filter**, **each**, **flatten**, **flatMap**, **merge**, **pluck**, **sort**, **unique** and **values** methods is necessary.

Also, we expect that you know the basics around PHP, especially the basics around PSR-4 namespacing, how to import classes from other namespaces and so forth. In many IDE's and editors, all this functionality can be installed with a simple plugin or will already built into the IDE or editor.

* * *

1

Introduction

Welcome to Build an API with Laravel, where we, as the title reveals, will take a look at how to build an API using Laravel. First, we will be looking at an API from a more theoretical point of view. Don't worry, we won't bore you to death with small details, but rather give you a fundamental understanding, which we can build on from there.

We will be looking at why we are using PHP and Laravel, and what makes it a great candidate for writing APIs.

We will go through the JSON:API Specification and learn about the protocols and conventions, and how these can help us build a more consistent API that is easier for us to consume.

We will look at how to plan an API, what to be aware of and what decisions you'll have to make, depending on whether your API is public or private.

Next, we will be looking at authentication, where we take a closer look at Laravel Passport and OAuth 2, which Laravel Passport is built upon. Here, we will go over the different grant types and what they are used for, to give you a clearer image of what you should choose for your applications.

Then, we'll get to the heart of the matter, where we will be writing the actual API. We will start out with a rather simple case to give you a better picture of how to apply the knowledge you'll get from this book and to have a common ground to build on.

We will be looking at Test-driven Development, especially the parts that are relevant to API development, and through this use the great testing tools from Laravel to test our API and also get an excellent workflow on top of it. We won't go into every detail about Test-driven Development, since it's a huge topic in itself and that's not what this book is about. We will, however, use Test-driven Development to show you how we can drive out our implementations, how we can refactor code to not having to repeat the same code over and over again, and get code that is easier to reuse.

Lastly, we will be looking at authorization and how you can authorize different parts of your API.

We will end the book with a bonus chapter, where we will go over the client side of things and show you how you can consume your API using client implementations, which is a huge advantage to using a set of strict protocols which the JSON:API specification will give us.

We hope you will enjoy reading this book as much as we have enjoyed writing it. We find that knowing how to build APIs has helped us a lot during our projects, since you can separate the concerns between frontend and backend, and thereby adapt to changes or new platforms, which might consume your API much easier. Let's get to it!

What is an API?

To best describe what an API is, let's imagine that our application or service is like a restaurant. The frontend of the application is where you sit at the table and eat, and the backend is the kitchen, where they prepare food for you. Here, an API plays the role of the menu, where you can pick what you want the kitchen to cook for you. In programming terms, an API makes it possible for a frontend developer to request a specific task or resource from the backend.

If you look at a menu, you might notice that it consists of a finite predetermined set of dishes: a collection of dishes that the kitchen knows how to cook and prepare for you. The same goes for an API, but instead of having dishes to choose from, you instead select between endpoints. Through endpoints, we can order our backend to prepare and send back some data for us. It could be a list of books for a bookstore or the latest comments for a blog post, depending on the service or application that serves a solution to a problem.

Like a menu being divided into starters, main course and desserts, an API is divided into resources. We'll be touching upon resources later — right now you just have to know that they exist.

Much like a menu can have various designs, the same goes for APIs and the architecture behind. At the time of writing this book, there are many types of API architectures. Some architectures like SOAP are not used that much more, while others like GraphQL are new and exciting. Then there is the common technology like REST, which we are going to cover in this book.

What is REST?

REST stands for **Representational State Transfer** and is an architectural style used for communication between a server and a client. REST uses the HTTP or **H**ypertext **T**ransfer **P**rotocol, as a base for the communication. We already use HTTP for transferring HTML and other media from servers to our clients. This is, for instance, what happens when you visit a website, so by building on an already familiar technology, REST is easily adaptable.

REST sends data using either XML or JSON. Both of these languages are meant for transferring data, but also meant to be readable by humans, which also makes error tracking in REST a lot easier. REST is platform and language independent, as long as you adapt to HTTP, you adapt to REST. Already established features of HTTP, like SSL encryption, make it possible to transfer encrypted data across from server to client, so there are a lot of advantages we get for free.

A disadvantage is that REST is not stateful, meaning that the state is not carried along from one request to the other. Therefore, you always have to send some kind of context to the server for it to know what to deliver to you. This limitation stems from HTTP itself, so this is most likely something you are already familiar with.

Like HTTP, REST works through requests, where you "pull" data from the server. There is no way of "pushing" data through REST, although it is possible for the server to do server pushing, using the HTTP 2 standard, but even that is only initiated by a request.

Since REST relies so much on HTTP, there are some things we have to examine to understand REST and the way communication takes place. For instance, HTTP methods, also called verbs, play a significant role in the intention for a REST request, as much as the HTTP Status code

plays a significant role in the answers. These are the HTTP building blocks that make up most of the base of REST communication. Let's take a closer look at HTTP Verbs.

HTTP verbs

As mentioned earlier, HTTP verbs play a significant role in the intention of a request. The HTTP verb tells the server about how we, on the client side, intend to handle our data. How to handle data is often looked at from a CRUD perspective, which stands for: Create, Read, Update, Delete. As an example, imagine an application for a bookstore. The CRUD part here will be responsible for: Adding, Listing, Updating or Removing books from the bookstore.

GET

A request with a GET verb is for reading data. That's the only thing this verb tells our server. The API endpoint will determine whether we are reading a collection of resources or just a single resource. We will go much further into detail about collections and resources later in this book, but for now, to put it into context, let's revisit the bookstore. Here, the resource is a book and a collection could be a stack of books.

POST

A request with a POST verb is for creating a resource. In other words, we use POST whenever we want to transfer new data from the client to the server.

PUT and PATCH

Both the PUT and the PATCH request is for updating or modifying data, but the way they are intended is a bit different.

PUT verbs are used when all data for a resource is completely replaced with the data given by the client.

PATCH verbs are used for a partial update or modification, instead of replacing everything in the resource.

Whether to use **PUT** or **PATCH** is all up to you and the needs of your application. The differences might be subtle, but they are there to make it clear to the client making the request, what is actually happening.

DELETE

A request with a **DELETE** verb is, much as the name implies, for deleting a resource

Now that we know a bit more about how we tell the server about our intentions using HTTP verbs, let's look at how the server tells us about the response through status codes.

Status Codes

Status codes are used by the server to tell the client whether a request has been successfully completed. The areas that status codes cover are divided into 5 groups. Let's take a look at a few from each group that we will be using the most:

2XX as Success

The 2XX range are statuses that tell the client a request was successful. You would think that only one status was needed here, but in some cases where, for example, you create something, it would be nice to know if the resource has been created.

Let's take a look at a few of the statuses we'll be using later.

200 OK

The **200** status code, which tells the client that the entire request was successful, is the most common. You might think that it's sufficient to use this one status and then add more details about the request in the response payload, but remember that these statuses were created for HTTP and to solve a problem. Since REST is built upon HTTP, and HTTP uses its status codes to communicate how a request has been fulfilled, why reinvent the wheel? Better to use what is already a well-known standard.

Which leads us to the next status.

201 Created

This status code tells the client that one or more resources have been created. As mentioned in the **200** status code, this status code makes it possible to only look at the **201** status, and we know, without having to look at the response payload, that the resource was created and we can move on much faster.

204 No Content

This status code tells the client that the request has been fulfilled, but also that there is no payload in the response. To give an example, this could be used when updating a resource. Here you don't really need any data back since you are telling the backend what to update and therefore know what to expect.

3XX as Redirection

The 3XX range are all statuses that tell the client about redirections. Most applications are continually being updated, and it is not uncommon that endpoints are being updated or removed. Then what do you do if someone out there is using an old endpoint where a sudden change could break their entire site or application?

Here, redirects play an important role.

301 Moved Permanently

The **301** status code tells the client that an endpoint has been moved and should give a new location in its payload for the client to save for future reference. A thing to note here is that the **301** status code makes it possible to change the HTTP verb for the request.

This example is a little silly, but let's imagine we have the following endpoint:

```
POST: /v1/book
```

We know you wouldn't use **POST** for this, but imagine that this endpoint returns a collection of books. With the **301** status code, you are allowed to change the HTTP verb when redirecting to the new endpoint, which then could be like this:

```
GET: /v2/book
```

This is not something we recommend doing, unless you are using a wrong HTTP verb beforehand, where a change makes sense. Let's take a look at a status code that allows you to redirect, but does not allow you to change the HTTP verb. A HTTP verb that can ensure the redirect is more consistent than in this example.

307 Temporary Redirect

The **307** status code is used for a temporary redirect. Here, the server should give a new location in its payload, for the client to redirect to.The client will not save any information about the redirect and will merely follow the location given by the server and gladly hit the old endpoint time after time, since the redirect is only temporary. A thing to note, as mentioned in the **301** status code, is that the **307** status code does not let you change the HTTP verb in the redirect. When redirecting the user, the endpoint to which you are redirected must match the HTTP verb from which you were redirected.

308 Permanent Redirect

The **308** status code is used for a permanent redirect, much like the **301** redirects. The only difference is that, like **307**, it does not allow for the HTTP verb to change from the original endpoint to the endpoint redirected to.

4XX as Client errors

The 4XX range are all statuses that deal with client error, which could be a request that the client is not authenticated to do or even a request to a misspelled endpoint, which the server cannot fulfill.

400 Bad Request

The **400**, much like the **200**, is a broad status code. All it does is tell the client that the server could not or would not process the request. It does not specify any reasons, and the client would have to look at the payload for further information.

401 Unauthorized

The **401** status tells the client that the request could not be fulfilled due to lacking authentication credentials.

403 Forbidden

The **403** status tells the client that the request could not be fulfilled due to lacking authorization. For example, this status code could be sent back if one user tries to access or update another user's data. The user might be authenticated to access the endpoint, but not have authorization to access the data.

404 Not Found

The **404** status tells the client that the requested resource could not be found. This is a pretty common status that most people, even non-developers, have been met by.

405 Method Not Allowed

As we have explained earlier, HTTP methods are also called HTTP verbs in REST. The **405** status tells the client that the request has been made with an HTTP verb that is not allowed. This could, for instance, be a request made using a **GET** verb to an endpoint that only supports the POST verb. In this case, a **405** status should be sent to the client.

422 Unprocessable Entity

The official RFC4918 states that this status is to be used when the server understands the content of the request and the syntax of the request is correct, but was unable to process the request. An RFC stands for **R**equest **F**or **C**omments, which can be viewed as the rules for standardizing the internet. The number references the document in which the request has been documented. In Laravel, the **422** status code is used for validation errors when using REST and JSON. The JSON:API documentation says that this status is to be used when creating or updating a resource where an attribute is invalid. Based on all these examples, we can safely say that the **422** status code tells the client about invalid data sent in the request.

5XX as Server errors

The 5XX range are all statuses that deal with the server, where it is aware that an error has occurred or might otherwise be unable to handle the request.

500 Internal Server Error

This status is used as a generic error message given when it is not possible to provide a more specific status code.

501 Not Implemented

This status is used when the server does not know how to fulfill the request, but implies that it might be available in the future. This could be a new feature that is under development.

502 Bad Gateway

This status is used when the server is acting as a gateway and has received an invalid response. If you have ever used nginx, you have probably seen this status code. Since nginx sits as the man in the middle and intercepts all incoming requests and then proxies these forwards, if nginx receives an error from the party it tries to proxy to, it gives you a **502** status code.

503 Service Unavailable

This status code is used if the server is down for maintenance

504 Gateway Timeout

This status is used when the server is acting as a gateway and did not receive a response within a given period. As with the **502** status code, this is something you see with nginx, where you configure a timeout limit, in which a request should be fulfilled or else a timeout will be sent back to the client. This is used to prevent the server from working forever on a job that might not be solvable. This could, for instance, be an error in PHP, where something loops forever. We don't want our users to wait forever and they want their data, so let's use a time limit and move on.

Summary

We have made a good start already and covered some of the basics for both this book but also APIs in general.

We have looked at what an API is and how it can be seen as a menu at a restaurant, where users can see what you can order from the backend.

We have taken a look at REST, how it builds on top of HTTP and thereby inherits all the abilities that HTTP already has. We have looked at HTTP verbs and how they play a significant role in the intention of a request. We have looked at the more common status codes and the ones we will cover in this book, how these are used to respond back to the client about how the request has been fulfilled or not. With this knowledge in mind, let's dig a little deeper into APIs and how to plan your work before you sit down and write your API.

* * *

2

The JSON:API specification

To read this chapter buy the full book at http://buildanapi.com

3

Planning

To read this chapter buy the full book at http://buildanapi.com

4

Build your API

To read this chapter buy the full book at http://buildanapi.com

5

Test-driven Workflow

To read this chapter buy the full book at http://buildanapi.com

6

Books

To read this chapter buy the full book at http://buildanapi.com

7

Don’t repeat yourself

To read this chapter buy the full book at http://buildanapi.com

8

Finishing up

To read this chapter buy the full book at http://buildanapi.com