

by Wacky Studio

**BUILD** an

**API**

with **LARAVEL**



THOMAS GAMBORG NØRGAARD

## Build an API with Laravel



*First published by Wacky Studio 2019*

*Copyright © 2019 by Thomas Gamborg Nørgaard*

*All rights reserved. No part of this publication may be reproduced, stored or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise without written permission from the publisher. It is illegal to copy this book, post it to a website, or distribute it by any other means without permission.*

*Thomas Gamborg Nørgaard asserts the moral right to be identified as the author of this work.*

*Thomas Gamborg Nørgaard has no responsibility for the persistence or accuracy of URLs for external or third-party Internet Websites referred to in this publication and does not guarantee that any content on such Websites is, or will remain, accurate or appropriate.*

*Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book and on its cover are trade names, service marks, trademarks and registered trademarks of their respective owners. The publishers and the book are not associated with any product or vendor mentioned in this book. None of the companies referenced within the book have endorsed the book.*

*First edition*

*Proofreading by Sille Justesen Krogh*

*Editing by Christian Nørrelund*

*This book was professionally typeset on Reedsy.*

*Find out more at [reedsy.com](https://reedsy.com)*



# Contents

|                                     |      |
|-------------------------------------|------|
| <i>Errata</i>                       | iv   |
| <i>Code samples and conventions</i> | v    |
| <i>Why Laravel?</i>                 | viii |
| <i>Prerequisites</i>                | ix   |
| <i>The Github Repository</i>        | x    |
| Introduction                        | 1    |
| The JSON:API specification          | 13   |
| Planning                            | 74   |
| Build your API                      | 137  |
| Test-driven Workflow                | 217  |
| Books                               | 335  |
| Don't repeat yourself               | 476  |
| Finishing up                        | 531  |
| <i>About the Author</i>             | 746  |

# 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 you might have found to [hello@wackystudio.com](mailto: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 be built into the IDE or editor.

\* \* \*

# The Github Repository

Throughout this book, we will not only be building an API, but also touching upon various concepts where we will need code examples. All of these can be found in our Github repository at this address:

```
https://github.com/WackyStudio/build-an-api-with-laravel
```

For the code to the API we will be building, we have the entire application as well as every step of the process separated into small folders in our Github repository, which you can use if you need help or need to see how we have implemented a certain feature:

---

 [01\\_install\\_passport](#)

---

 [02\\_make\\_auth](#)

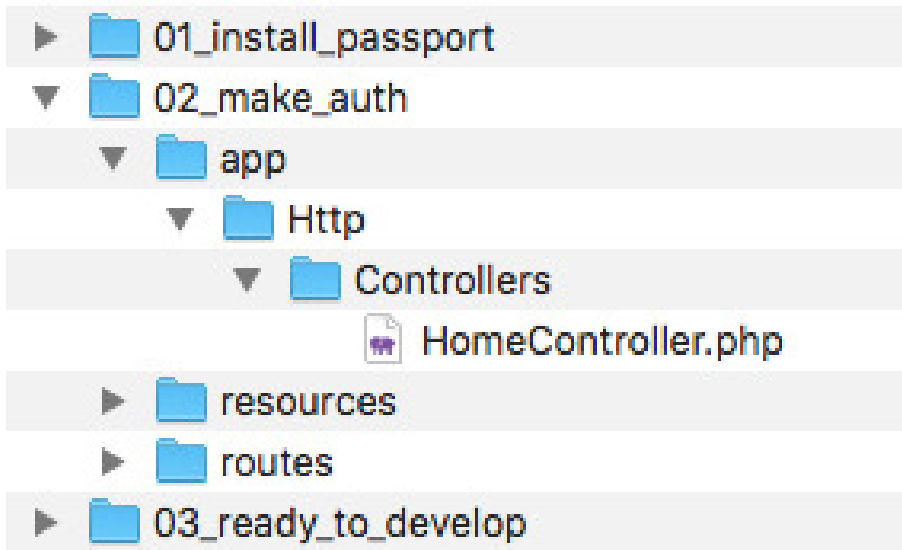
---

 [03\\_ready\\_to\\_develop](#)

---

Each step will add onto the next one and only contain the files that have been added or changed following the structure of our main Laravel application, so you can easily follow along:





You don't have to clone this repository from the beginning, but if you want to follow along in your code editor instead of the code examples in the book, you can do so with the following commands in your terminal. You will have to be in the desired folder on your local system where the repository should be cloned down to, before running any of the commands. When you are ready, type the following command and make sure you are in the right directory before cloning it:

```
git clone git@github.com:WackyStudio/build-an-api-with-laravel.git
```

If you are not using SSH you can clone through HTTPS like this:

```
git clone https://github.com/WackyStudio/build-an-api-with-laravel.git
```

If you are not familiar with git, you can also download a zip file with the contents by visiting the repository with the link given previously in this section.

\* \* \*

# 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 **Hypertext Transfer Protocol**, 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 **Request For Comments**, 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

Let's have a look at the JSON:API specification — a specification for building APIs in JSON. A specification like this can help us build our API, following a set of rules that makes it easier for ourselves, and especially others, who will be consuming our API in the future.

We will be looking at the specification and how it fits into the features Laravel offers, and how it can be a great tool to master, and make it possible for you to actually plan out your API with more confidence.

Firstly, we will be looking at a case for this book to have something to work from: a common ground which makes it a bit easier to put things into perspective. This way we can give examples when looking at the JSON:API specification with context and in later chapters show you how we go from planning all the way to implementation.

Let's have a look at the case.

## The case

As mentioned earlier, we will be using a case to establish a common ground. It makes it easier for us to give examples in a context you understand, and lets us reuse this context for planning and later implementation. By having a case that simulates a real life example, it might be easier for you to understand the concepts we will present to you and apply them into your own projects in the future. We all learn differently, but it is our experience that having a case and a context makes it easier to see the concepts applied and learn how to apply them yourself.

So without further ado, let's get into the case:

*Anna's Bookstore has existed since the early 90s where it opened as a small corner shop in the city. It isn't a fancy place — the floor, bookshelves and furniture make it seem a bit like your grandma's place, but it's cosy, welcoming and the perfect combination of a bookstore and coffee shop, where you will feel right at home. Customers come back for Anna's personality, which is reflected in every little thing in the store, and especially the selection of books. It's not the selection of books you'll find in every store; these are hand-picked by Anna herself. She'll often offer customers a cup of coffee and discuss a book, or the work of an author, when there isn't anything to do at the cash register. You can feel that both literature and people are her passion.*

*As the years go by, new stores start emerging and Anna gets a lot more competition, especially from modern bookstores, where people can find and buy books faster. This isn't really Anna's cup of tea, and to compete, she would have to change everything about her store. She's beginning to think about closing the store, until a day when her nephew visits her. When they talk about the store, the nephew asks: "Why don't you go online, so you can keep your store as it is and sell to people at home or in a hurry?". Anna must admit that it is a good idea. Over the years,*



*she has started to order books online herself and is quite pleased with the experience. The nephew then continues: “You don’t have to have a huge selection of the books, why not cater to a niche and sell a curated selection of books, like you do in your store now? You know people always praise you for having so many hidden gems.” Anna is convinced. This is something she can see herself in, she can still use her expertise, and people can still buy a bit of Anna’s personality in the selection of her books.*

*Other than being able to sell books and being able to keep stock of her books, Anna wants to be able to present these books through authors. Anna has a bunch of authors that she can vouch for, who always publish good work, a key part of her having so many hidden gems.*

*Sometimes, Anna finds books by reading comments that other people have left on the online bookstores. She loves the helpful tips people give in the comments and it reminds her of the discussions she has been a part of in her own bookstore. She would like something like this in her online bookstore as well.*

There are a few things to take note of here. To some, the case might sound really simple and to others, it might seem daunting. But fear not, we won’t be building the entire solution for Anna. Instead, we will pick out parts of the online bookstore that makes sense in terms of an API, and not the entire purchase flow etc. Keep in mind that we have written this case to communicate the concepts about building an API, as easy as possible, and there is more than enough in the part about handling books and authors to do that.

Now that we know more about the case and what we need to build, we could go straight to the planning, but let’s hold that thought and take a closer look at the JSON:API specification, and learn a bit more about the protocols that will be the foundation of how we communicate with our API.

## JSON:API specification

When developing software, there are a lot of decisions to make about how to design the software in the best way — from how to design your code to the application architecture, and all the way up to the visual representation of your application on screen. With APIs this isn't any different.

Up until 2013, where the first draft of the JSON:API specification was made, there weren't any approaches to standardization of JSON API interactions.

Before adhering to the JSON:API, we at Wacky Studio even made our APIs in our own way, the way we thought was best. We drew inspiration from APIs of the services we used and made our decisions based on how they were implementing their APIs.

That was a very bad idea because:

- They were also in a phase of learning how to write a good API for their services
- None of them were following the same conventions
- Most of them had an API design that meant you had to make a lot of requests for data

The worst of it all was when we had to consume our own API on the frontend. Because of the lack of conventions, we had to constantly go back and forth to get the correct endpoints, to see what data each request should contain and to see what would be returned from the server.

And every time we started a new project, we ended up writing all the code for communication with the server, again and again, because of inconsistency.

You see, it's not the development of the API on the backend that takes time and causes pain. The pain is mostly felt when you have to work with the API on the

frontend, or even worse when other people or companies are working with your API. Without strict conventions, you not only have to write documentation that shows what your API can do, but you also end up teaching how to use the API.

This is where following a specification like the JSON:API specification shows its worth. How to use it is already documented and as long as you follow the specification, anybody who knows how to work with the JSON:API specification, knows how to work with your API. Of course, they won't know what your API can do — you will still have to tell them that — but how to communicate with your server through your API, will never be a problem for them.

Even better, when following the conventions of the JSON:API specification, you have a strict protocol that never changes from application to application. This means that you can extract the whole client-server communication part of your frontend application and reuse it from application to application. You won't have to write the same tedious boilerplate code over and over again, and you can focus on building the functionality of your frontend app instead. The JSON:API specification even lists available implementations, which includes implementations made for a lot of languages like Javascript as well as frameworks like VueJS and React.

The JSON:API specification was drafted in 2013 and had the first final v1.0 ready in 2015, where it has been used ever since. At the time of writing, a new version is about to be released, but as stated in the specification, the new version will always be backward compatible, using a “never remove - only add” strategy. So you won't end up like us, where your inspiration is suddenly drastically changed. When implementing the JSON:API specification, you are ensured that people will always be able to use it, no matter which version of the specification they have learned and/or are using.

As we see it, the JSON:API specification is a great approach to standardization of APIs and throughout this book, we will dive further into it and show you

how to build an API in Laravel using it. Before we get ahead of ourselves, let's commence by looking at the fundamentals of the specification.

## Client / Server Responsibilities

When you adopt the JSON:API specification, the first thing you have to look at are the headers sent in your request and responses. As a client, you have to send your request with

```
Accept: application/vnd.api+json
```

And

```
Content-Type: application/vnd.api+json
```

These headers tell the server that what you're sending lives up to the protocol given in the JSON:API specification, and also that you expect to receive data in the response that adhere to that same protocol.

As a server, you **have** to deliver your response with

```
Content-Type: application/vnd.api+json
```

to tell the client that what you're sending lives up to the protocol given in the JSON:API specification.

## Endpoints

Though the specification provides a strict protocol for how requests and responses are structured, it only gives a few recommendations about how to form your endpoints. In this section, we will have a look at the recommendations from the JSON:API specification and give some of our recommendations as well.

It isn't hard to define endpoints, since you are already used to it by the routes you have defined in your existing Laravel applications. The thing that can be hard is making sure that you are consistent.

### *Naming conventions*

In Laravel, we are used to working with models through the Eloquent ORM. Models define the tables in our database that hold the data for our entire application.

Our API should give the client the data being requested and these data are most likely to be fetched from our database. Therefore, it is also very convenient to be thinking of our models as resources.

We are used to naming our tables after what they represent in the real world and often with nouns. As an example, the data for the users of our application will most likely be stored in a **users** table with a **User** model. The books of our bookstore will be placed in a **books** table with a **Book** model. The resource in this example would be "books".

The naming convention may confuse since you are used to working with model names in a singular naming convention, but the table names are made in a plural naming convention. When it comes to resource naming, there have been a lot of discussions whether to use singular or plural names.

The JSON:API specification doesn't provide a clear answer here, but gives the following example of a plural naming when having URL for a collection of resources.

```
GET: /photos
```

Later on, they do give an example where they use singular naming, but here it seems like it is done when there is a **one-to-one** relationship between resources.

We have been down both roads and have felt both the upsides and downsides to plural and singular naming conventions and especially with a combination of both. We started out with a singular naming convention because it made a lot of sense, since it's so close to how you work with models in Laravel. If we want to get a book we write:

```
<?php  
  
$book = Book::first();  
// or  
$book = Book::find(1);
```

Here, our endpoint would reflect this as:

```
GET: /book/1
```

And it makes sense — if you want a single book, you write it out in singular. But what about the situation when you want a collection of books?

Because of the conventions in the Laravel framework where model names are using a singular naming convention, it is not unusual to get a collection of Books when you write the following:

```
<?php  
  
$books = Book::all();
```

For us, when it came to reflecting this in endpoints, we had a little trouble. It felt wrong getting a collection of books by:

```
GET: /book
```

Because of this, we came up with a solution that borrowed a bit more of how Laravel's Eloquent ORM work, with an endpoint like this:

```
GET: /book/all
```

It isn't pretty and it would force our consumers to always remember the all when wanting to fetch collections of resource, which we also occasionally forgot, whenever we had been away from the project in some time.

The next API we wrote, we wanted to change that ugly reference to all and write something that made more sense. We opted to change the resource naming for requests for collections to plural. In this way, we would write the following to get a collection of books:

```
GET: /books
```

And we could then write the following to get a single book:

```
GET: /book/1
```

We thought this was a great and consistent way, until we stumbled upon words with irregular plurals. As a trivial example, let's look at the word leaf. It has one spelling in singular, but the plural spelling leaves is different.

Somebody whose first language isn't English, could end up writing leafs and get an error. This could lead to confusion since you write the following to get a single leaf:

```
GET: /leaf/1
```

To get a collection of leaves, you write the following:

```
GET: /leaves
```

There is suddenly an introduction of inconsistency and that is not what we want. We want to have predictable behavior and one word only for a resource.

The way we do it now, and the way we would recommend, is to use plural **only**. This might sound strange and like we would hit the same obstacles, but that is not the case. Let's look back at the conventions by Laravel's Eloquent ORM.



Model names are in singular, but tables names are in plural. Models are in singular, because you are only interacting with a single model at a time, since a single model corresponds to a single row in the database. Here, the singular naming convention fits perfectly.

If you have multiple models, these are placed in collections. It's not an instance of a Model you get. No, here you'll get an instance of a Collection, which contains an array of many models.

A database table will contain one or more rows. It is suddenly a collection of data for the thing it represents, therefore the plural naming convention of Laravel makes sense for these. If we want a certain row in that table, we use a Primary Key, most often with the name `id`, to access that single row.

We like to take that same approach to our endpoint naming convention. We use a plural naming convention, like our table names, because we expect a collection of resources. We won't be able to avoid irregular plurals, but by sticking to plural only, our consumers only need to remember one name and don't have to care if that name is an irregular plural. They just have to remember the name **leaves** and that's it.

A request to the following will give us a resource collection of all books:

```
GET: /books
```

If we want a single book, we fetch it, by giving an `id` to the books collection, like this:

```
GET: /books/1
```

This also matches the way the JSON:API specification wants us to treat collections of resources, namely as arrays keyed by a resource ID.

There is also a naming convention when it comes to relations between resources. These are a part of the protocols given by the JSON:API specification, which we will look further into in the upcoming sections.

### *Before we continue*

In the upcoming sections, we will take a deeper look at the JSON:API specification. We will touch upon conventions that the JSON:API specification states as conventions that **MUST** be followed and conventions the specification states as conventions that **MAY** be followed. However, we recommend that you follow the conventions we have picked out, whether the specification states that they **MUST** or **MAY** be implemented. We will touch upon most of the conventions given in the specification, but there are a few that we haven't had any use for and feel that they cover more edge cases.

### *Document structure*

Let's look at the document structure of the data for both JSON:API request and responses. The document describes how your JSON data should be formed, how members should be named, where these should be placed, and so forth.

### *Top-level*

Here, the JSON:API specification states that there must be a JSON object at the root of the document, representing the top-level.

In the top-level of the document, there must be at least one of the following members:

- **data** – which is the most important member that contains the primary

data of the document.

- **errors** – which is a member that contains all error objects.
- **included** – which is a member that contains all resource objects that are related to the primary data and/or related to each other. We will touch more on this when we get to the section about resource objects and relationships.
- **jsonapi** – which is a member that contains the server’s implementation of the JSON:API specification
- **meta** – which is a member that contains all non-standard meta information.

Note that it is very important that the data and errors member never coexist in the same document. The data member should only be used in successful request and responses, where the errors member should only be used whenever there is an unsuccessful request or response. By separating these, you have a clear convention that states where to look for either data or the errors that might occur.

Now that we know what the top-level structure should be, let’s take a look at what our primary data will be and also how to structure that.

### *Primary data and Resource objects*

In the section about naming conventions, we talked about how convenient it is to be thinking of our Laravel models as resources since these represent the rows of data in our database and data that we most likely will share across our APIs.

In this section, we will be looking at how to structure these resources according to the JSON:API specification. Resources or **resources objects**, as they are called in the JSON:API specification, will be placed in the **data** member and therefore serve as the **primary data** in the JSON:API.

We know that Laravel Models can be returned in a response, where Laravel will handle the whole conversion of the Model data into JSON, without you having to lift a finger. That's great and a very convenient feature — we have certainly used it a lot in our earlier API days. But the problem is that the data returned is not consistent.

There is no strict document layout so you know where to look for the data you need. Instead, everything is just exposed in the top-level of the returned document. One endpoint exposes a new resource with members different than the next one and you'll quickly have to look at the documentation to find out where to look for the data. Moreover, you actually can't see what type of model you are receiving, so you'll have to rely on the naming of the endpoints to tell that part of the story.

As a solution to the aforementioned problem, the JSON:API specification tells us to structure our resource object in this way:

```
{
  "id": 1,
  "type": "books",
  "attributes": {
  },
  "relationships": {
  }
}
```

In the example, you can see a clear structure. In the root of the resource object you'll find:

- **id** – which is the id of the resource as a string
- **type** – which is the type of the resource as a string
- **attributes** – which contains all of the attributes of our resource
- **relationships** – which contains all of the relationships of our resource

This structure mitigates the problem of not knowing where to look for your data, not knowing the type of the resource, and it gives us a predictable and consistent way of accessing the data of a resource.

It is ok for the **attributes** and **relationship** members to be empty. In fact, these can be removed if not used. But, as an absolute minimum, you should always have the **id** and **type** members in your resource objects, and the value of both should always be a **string**.

Ok, so we know how to structure our resource objects, but as we talked about in the naming convention section, there is a difference between requesting a collection of resources versus requesting a single resource.

The difference here is not that big, but it is important to be aware of it.

When requesting a single resource like this:

```
GET: /books/1
```

the **data** member of the returned document should be structured like this: (***note** that we are omitting the attributes and relationships for the sake of simplicity*)

```
{
  "data": {
    "id": "1",
    "type": "books"
  }
}
```

Here, the **data** member is the resource object itself. When requesting a collection of resources like this :

```
GET: /books
```

the **data** member of the returned document should be structured like this: (***note** that we are omitting the attributes and relationships once again*)

```
{
  "data": [
    {
      "id": "1",
      "type": "books"
    }
  ]
}
```

Here, the **data** member is an array containing the requested resource objects. As you can see here, it should be an array even if there is only one resource in the collection. If there weren't any resources in the collection, an empty array should be returned.

We got the basics down and it's time to look at those attributes and relationships we have omitted in the examples. Here, we open up for the ability to create our own member names, therefore it is important to look at the naming convention for these as well to ensure consistency.

### *Member names*

The JSON:API has a clear naming convention when it comes to member names, where all member names must be treated as case sensitive by both client and servers. Other than that, there are some conditions that the member names must also follow:

1. Member names **must** contain at least one character

2. Member names **must** contain only allowed characters
3. Member names **must** start and end with globally allowed characters

The globally allowed characters are:

- a-z
- A-Z
- 0-9

The characters that are allowed except for the beginning and end of a member name are:

- -
- \_

As an example, it is ok with a member name like this:

```
{
  "member_name": "content"
}
```

But it is not ok with a member name like this:

```
{
  "_member_name_": "content"
}
```

The above example is pretty trivial. Who would do that, right? The important thing to remember here is to have a letter in the beginning and the end and

you're home safe.

We strongly recommend that you keep all your member names in lowercase and stick to a convention when picking characters like spaces, like these examples:

Using underscores as space, also known as snake case:

```
{
  "member_name": "content"
}
```

Using hyphens as space, also known as kebab case:

```
{
  "member-name": "content"
}
```

Using a capital letter on the next word to indicate a space, also known as camel case:

```
{
  "memberName": "content"
}
```

We have adopted the camel case as a naming convention when coding in PHP, but in our APIs, it's a bit of a different story. Here, we use snake casing, mostly because that's a convention we have used from the start, when looking at other companies' APIs. At the time of writing, both Google, Dropbox, and Facebook use snake cases in their APIs. Also, when calling the **toJson()** method on your



model, Laravel converts your model attributes spaces into snake case.

Our recommendation is to use snake cases, especially because we will be using these in this book, but also since it comes for free with Laravel. The choice, however, is entirely yours — just make sure you are consistent and don't suddenly change in the middle of working with this book or in your own APIs.

## *Attributes*

Now that we have a naming convention for our member names, we can continue to attributes. Attributes on a resource object are just like the attributes on your model in a Laravel Application. These are data like the title of a book, the name of an author, and so forth.

The JSON:API specification specifies that on a resource object, the **attributes** member should be an object. Any member inside this object can be whatever data that represents the object, but must never be a relationship. For relationships, we use a dedicated member in the resource object, which we will look at shortly.

To make an example, let's look at a single book again:

```
{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Build an API with Laravel",
      "publication_year": "2019"
    }
  }
}
```

Here, you see the definition of the **attributes** member as an object containing two members, namely **title** and **publication\_year**. You also see how the naming convention of snake casing takes effect.

As mentioned earlier, the **attributes** member can contain any information about the resource object, but cannot contain relationships or a member called **relationships**.

Another rule is that the **attributes** member can never contain an **id** or **type** member, since these are reserved on the root of the resource object.

But how do we define relationships then? Let's take a look at that next.

### *Relationships*

When it comes to data in an application, these are often related to one another. When building applications in Laravel, we are used to defining models as objects in real life and as such, these have different relations to one another:

A car belongs to a brand, a bus can have many passengers, a book can have many authors, and an author can have written many books.

Do you see how everything connects? Chances are that you have already written something like the sentences we just presented, since Laravel uses most of the wording in the relationships you define in models.

```
<?php

namespace App;

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

```

class Car extends Model
{
    /**
     * Get the brand of the car
     */
    public function brand()
    {
        return $this->belongsTo('App\Brand');
    }
}

```

By declaring a relationship method on our model, we can fetch the brand of our car very easily, now that we have told Laravel about the relationship.

But how do you tell about relationships in your APIs and how do you convey enough information, so that your consumers can easily get the data they want? Relationships in the JSON:API specification is defined as the **relationships** member.

Like the **id**, **type** and **attributes** members, it should be placed at the root of the resources object and be defined as an object like this:

```

{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Build an API with Laravel",
      "publication_year": "2019"
    },
    "relationships": {
    }
  }
}

```

Unlike the **attributes** member, where you are in charge of the members, the **relationships** have a more strict set of rules. A **relationship** member must contain at least one of the following members:

- **links**
- - **self**
- - **related**
- **data**
- **meta**

Let's take a look at the **links** member. This member contains two types of links. The link for the **self** member is a link for the relationship itself. With this link, it is possible to manipulate the relationship between two resources without having to delete one of them. A good example here would be tagging. If a book contains one or more tags, this link can be used to remove the tag from the book without having to delete the tag or the book.

The link for the **related** member is a link for the relation between resources. When making a request to this link, the related resources will be queried and returned as primary data. This is very much like calling a relationship method on your Laravel models, where Laravel will make a query for the related models of that model for you.

The **data** member is something we have seen before, yet this one is a little different. It's called the **resource linkage** and instead of holding **resource objects**, it holds resource identifier **objects**. In contrast to resource objects, which hold **id**, **type**, **attributes** and relationships members, resource identifier objects only contain the **id** and **type** members of the related resource object.

The **meta** member is a meta object that can contain non-standard metadata about the relationship. We haven't had the need for this yet and thus won't include it in the examples. Now that we're talking about it, let's take a look at the relationship between a book and an author:

```

{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Build an API with Laravel",
      "publication_year": "2019"
    },
    "relationships": {
      "authors": {
        "links": {
          "self": "http://example.com/books/1/relationships/authors",
          "related": "http://example.com/books/1/authors"
        },
        "data": {
          "id": "5",
          "type": "authors"
        }
      }
    }
  }
}

```

If you take a look at the JSON example, you can see the **author** member inside the relationships object. In this example, we have the links that make it possible to easily fetch the related resource, which in this case is the author of the book.

In the example above, we have a single author, as in a **one-to-one** relationship. In the case of a **one-to-many** or **many-to-many** relationship, an array should be used instead, just like this:

```

{
  "data": {

```

```

    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Build an API with Laravel",
      "publication_year": "2019"
    },
    "relationships": {
      "comments": {
        "links": {
          "self": "http://example.com/books/1/relationships/comments",
          "related": "http://example.com/books/1/comments"
        },
        "data": [
          {
            "id": "16",
            "type": "comments"
          },
          {
            "id": "28",
            "type": "comments"
          }
        ]
      }
    }
  }
}

```

The way the **data** attribute contains resource identifier objects is just like the primary data's **data** member, which holds either an object for a single resource or an array for a collection of resources.

Ok, that was a bunch of rules at once. Let's recap on how to define a relationship. We make an object as the **relationships** member. Each member inside relationships defines each related resource.

In the examples given above, the relationship is between books, authors and comments. All of those combined would look like this:

```

{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Build an API with Laravel",
      "publication_year": "2019"
    },
    "relationships": {
      "author": {
        "links": {
          "self": "http://example.com/books/1/relationships/authors",
          "related": "http://example.com/books/1/authors"
        },
        "data": {
          "id": "5",
          "type": "authors"
        }
      },
      "comments": {
        "links": {
          "self": "http://example.com/books/1/relationships/comments",
          "related": "http://example.com/books/1/comments"
        },
        "data": [
          {
            "id": "16",
            "type": "comments"
          },
          {
            "id": "28",
            "type": "comments"
          }
        ]
      }
    }
  }
}

```

Inside each relationship, we have at least one of the members:

- **links**
- **data**
- **meta**

In our case, we have both **links** and **data**, which we recommend that you do as well.

The **links** members give us a consistent way of accessing either the relationship between the resources or the related resource object. The **data** members give us either a resource identifier object or a collection of resources.

We now know how to define relationships, we even know how to provide links for manipulating relationships and how to fetch related resources. If we want the comments for the book, we can simply make a request to the link given in the **related** member and a response containing all the resource objects will be returned.

Right now, the **data** member of the **relationships** seems a bit redundant, since it only contains **id** and **type** member instead of an entire resource object, but let's look further into this in the next section and it will make more sense.

## *Compound Documents*

We have just looked at the **relationships** member and what kind of members this should contain. We left with some confusion about the **data** member inside the **relationships** object. To understand this part, we need to revisit the top-level of our document, more specifically the **included** member. We only touched upon this briefly in the section about Top-level, so let's have a better look at this.

When building an API or an application for that matter, you have to make



some thoughts about optimization and make sure your application performs as intended. One optimization could be to reduce the number of HTTP requests as much as possible.

One way to do this is to use the **included** member. The reason for this is that it makes it possible for you to include the related resources of the fetched resource, which will then be the resources defined in the **data** member in the **relationships** object.

Instead of having to make a new request for the related resources, they can just be included in the current response.

In this case, the resource objects sent in the **included** member will correspond to the resource identifier objects given in the relationships' **data** member.

Let's build upon the previous examples to give a better idea of this:

```
{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Build an API with Laravel",
      "publication_year": "2019"
    },
    "relationships": {
      "author": {
        "links": {
          "self": "http://example.com/books/1/relationships/authors",
          "related": "http://example.com/books/1/authors"
        },
        "data": {
          "id": "5",
```

```

        "type": "authors"
    }
},
"comments": {
    "links": {
        "self": "http://example.com/books/1/relationships/comments",
        "related": "http://example.com/books/1/comments"
    },
    "data": [
        {
            "id": "16",
            "type": "comments"
        },
        {
            "id": "28",
            "type": "comments"
        }
    ]
}
},
"included": [
    {
        "id": "5",
        "type": "authors",
        "attributes": {
            "name": "Wacky Studio"
        }
    },
    {
        "id": "16",
        "type": "comments",
        "attributes": {
            "body": "Hello world"
        }
    },
    {
        "id": "28",
        "type": "comments",

```

```
    "attributes": {  
      "body": "Foo bar"  
    }  
  }  
]  
}
```

In the example, you can see how the **included** member, includes all of the resource objects, for the resource identifier objects given in the **data** member.

Here, it's important to note that the **included** member will always be an array that contains all of the related resource objects mixed together in a flat array.

The **included** member can be included by default, or by an **include** query parameter like this:

```
GET: /books/1?include=comments
```

Here, there will not be anything included before the query parameters are in the URL and if the relationship does not exist, a **400 Bad Request** should be used. If you choose to support using the **include** query parameter, there are a few more things you should implement.

First, it should be possible to specify which relationships that should be included in the response using a comma separated list:

```
GET: /books/1?include=authors,comments
```

It should be possible to request resources related to other resources using a

dot-separated path for each relationship name. In our example, a book has a relationship with authors and comments, but each comment also has an author, in the form of a user who created the comment. If we wanted to include the users for each comment, it should be possible to do this by adding the related resource like this:

```
GET: /books/1?include=authors,comments.users
```

Again, if it isn't possible to fetch the related resource, you should return with a **400 Bad Request**.

Whether you want to use the **include** query param is all up to you, and the same goes for the **included** member in your response documents, but if you choose to do so, you must have the **data** members in the **relationships** object, for each of your relationships. If not, you can omit the **data** member, but you then have to have the **links** member, so that the related resources can be requested through the **related** link.

Now, let's take a step back and think about what we have been through.

We now know how to structure the document for our data. We know that we must have a top-level object and that we must have a data member representing our primary **data**, and that the **data** member can be either an object or array, whether it's a single resource or a collection we have requested. We know that a resource is represented in our document as a resource object that must contain an **id** and **type** member, both with a string datatype.

We know that we can use an **attributes** member to give information about our resource object, which in this case would be the attributes of your Laravel models.

We know how to use the proper member name convention in our **attributes**

object and how it's important to stick to a naming convention strategy as snake case to keep consistency.

We know how to represent a relationship between our resources through a **relationship** object.

We know how to define a relationship as yet another object, which contains a **links** member with links to the relationship itself or the related resources and a **data** member holding resource identifier objects for use in the **included** top-level member.

We know how to use the included top-level member to save HTTP request by sending related resources in the response.

That was quite a lot and we are almost done with the JSON:API specification. Before we move on though, we just have to look at how we make requests and responses using this new document structure and also how we handle errors.

### *Request and responses*

It's time to look at how we should make our request and responses according to the JSON:API specification. We know what to send and what we can expect to receive, but we don't know how to request it or how these should be sent with a response yet. Of course, there are conventions and we will adhere to them. Let's take a closer look.

### *Requests*

All requests to get data from our API must be done with a GET request. Remember the previous chapter in the section about REST and HTTP verbs, the GET request is for reading data and the same goes for the JSON:API specification. Nothing has changed here.

Some interesting conventions the JSON:API specification brings along for requests is the ability for sorting and pagination of collection data. These are only optional conventions, but we are mentioning them because we have had great use of these. There are more conventions, but these are outside of the scope of this book.

Now, let's first take a closer look at sorting.

## *Sorting*

Sorting data is a great feature to have in an API. Think about sorting just like ORDER BY in your database. You get the ability to sort your data based on member names in a more dynamic way, instead of being limited by the way the API developer may have thought was the best way to sort the data.

Sorting data is done via a query parameter. If you are unsure what a query parameter is, it is a convention in HTTP you use to send along parameters for a request, as a part of the URL like this:

```
GET: http://example.com/cars?color=blue
```

Here, the parameter we are sending along is color with a value of blue. The query parameter used by the sort feature is the sort parameter.

The value of the parameter is the member name of the attribute you want to sort by. It would look something like this:

```
GET: /books?sort=title
```

If you want to support multiple sort fields, these should be separated by a comma like this:

```
GET: /books?sort=title, publication_date
```

When ordering a database query by a column name, we are able to tell if the ordering should be done in ascending order or descending order. The same thing goes for sorting a collection, according to the JSON:API specification. Here, a sorting is always done in ascending order unless you prefix a sort field with a minus, in which case it will be sorted in descending order. It would look something like this:

```
GET: /authors?sort=-age
```

Here, you will get the oldest authors first, descending until the youngest author in the collection.

## *Pagination*

Pagination is another feature that can have great benefits, especially if you have large sets of data that can be quite a strain on the system to query. You can paginate the results and do the queries in smaller chunks, letting the API consumer do the work of progressing through the pagination.

The way pagination is done in the JSON:API specification is through a links object in the root of the response document. The links object must have the following members used for pagination links:

- **first** – which is the first page of data
- **last** – which is the last page of data
- **prev** – which is the previous page of data

- **next** – which is the next page of data

The links object in the document would look something like this:

```
{
  "data": [
    {
      "id": "4",
      "type": "books",
      "attributes": {
        "title": "Lorem Ipsum"
      }
    },
    {
      "id": "5",
      "type": "books",
      "attributes": {
        "title": "Lorem Ipsum"
      }
    },
    {
      "id": "6",
      "type": "books",
      "attributes": {
        "title": "Lorem Ipsum"
      }
    },
  ],
  "links": {
    "first": "http://example.com/books?page=1",
    "last": "http://example.com/books?page=5",
    "prev": "http://example.com/books?page=1",
    "next": "http://example.com/books?page=3"
  }
}
```

In the example, you can see a collection of books and in the bottom of the response document you see the **links** member with the four members of a



pagination link object. Can you guess which page we are on? Correct! We are on page two! Had we been on the first page, the JSON:API specification actually requires us to omit or set a null value for the links that are unavailable, which in that case would be the **prev** link, since there is no previous page, when being on the first page. Just to demonstrate, here is an example of that scenario:

```
{
  "data": [
    {
      "id": "1",
      "type": "books",
      "attributes": {
        "title": "Lorem Ipsum"
      }
    },
    {
      "id": "2",
      "type": "books",
      "attributes": {
        "title": "Lorem Ipsum"
      }
    },
    {
      "id": "3",
      "type": "books",
      "attributes": {
        "title": "Lorem Ipsum"
      }
    }
  ],
  "links": {
    "first": "http://example.com/books?page=1",
    "last": "http://example.com/books?page=5",
    "prev": null,
    "next": "http://example.com/books?page=2"
  }
}
```

You see how a **null** value is provided for the **prev** member to indicate that it is unavailable.

The JSON:API specification does not have any conventions when it comes to query parameters signifying which page in the pagination we are currently on. However, they state that the page query parameter can be used for this and we will recommend that as well. The good thing about this is that it's then possible to deep link into a **page** of data using the **page** query parameter. We will support this in our API as well.

Now, we know how to make a request for data and even how we can sort or paginate data provided in collections. It is time to look at responses and the conventions from the JSON:API specification we must follow.

## *Responses*

It's time to look at the server side and which convention it must adhere to when sending responses to the client. Here, we will revisit HTTP verbs and status codes as well as look at conventions that must be followed to keep your API consistent. The first rule we will look at is making GET requests for data or fetching data, as the JSON:API specification calls it.

## *Response guarantees*

Here, the server must always support getting resource data and or relationship data for all URLs that are provided in a response. The URLs we are talking about are the URLs provided in a relationship for a resource object. It is the links given in the **links** object of the relationship object, more specifically the **self** and **related** links. It should always be possible to get data through these links, otherwise we are breaking the conventions from the specification. Also, it would not make a lot of sense if links we provide from the API does not work. It would lead to a lot of frustration for the consumers of the API and that's not what we want.

## *Resource Responses*

When we make requests to a server for a specific resource or collection, it must always return a status code **200 OK**.

Referring back to the former chapter, we talked about how the **200** status code is the most common, which tells the client that the entire request was successful. When making such a response, the server must also ensure to send the requested resource or resources in the response documents primary data. As covered in the section about Primary data and Resource objects, the server must ensure that the primary data for a response document is either a single resource or a collection of resources.

Here's an example of a response document for a request for a single resource:

```
{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Lorem ipsum"
    }
  }
}
```

Here's an example of a response document for a request for a collection of resources:

```
{
  "data": [
    {
      "id": "1",
```

```
    "type": "books",
    "attributes": {
      "title": "Lorem ipsum"
    }
  },
  {
    "id": "2",
    "type": "books",
    "attributes": {
      "title": "Lorem ipsum"
    }
  }
]
```

If the single resource cannot be found, the primary data would have to be **null**. Take a look at this example with a request to the following resource:

```
GET: /books/1
```

```
{
  "data": null
}
```

Here, the resource with an **id** of the value **1** does not exist. This, however, is only a response you should use if the request warrants a **200 OK** response. The JSON:API specification unfortunately doesn't give a concrete example of this. We're using a **404 Not Found** response for this scenario since it gives a clear message about the resource not being found.

If a collection of resources cannot be found, the primary data would still have to be an array, but an empty array though like this:

```
{  
  "data": []  
}
```

Where a request for a single, not existing resource, should trigger an error since you are trying to access something that does not exist, a collection will always exist. A collection is allowed to be empty since this can be filled with resources later on when they are being created.

### *Relationship Responses*

Relationship responses follow some of the same conventions as Resource responses. Remember though that there are two types of links in a relationship:

- **self** – Which is a link to the relationship itself
- **related** – Which is a link to the related resource

Here, the link for the **related** member would give a response document of a resource, since you are requesting the related resource or collection of resources, depending on the relationship.

For the **self** member, it's a little different. Here, a request to the **self** link will respond with a response document, where the primary data would be the resource identifier object. Here is an example of a book resource with a relationship to an author. The relationship in this context is **one-to-one**, meaning that a book can only have **one** author.

```
{  
  "data": {  
    "id": "1",  
    "type": "books",  
    "attributes": {
```

```

    "title": "Lorem ipsum"
  },
  "relationships": {
    "author": {
      "links": {
        "self": "/books/1/relationships/authors",
        "related": "/books/1/authors"
      },
      "data": {
        "id": "1",
        "type": "authors"
      }
    }
  }
}

```

The URL for the **self** member is the following, and a request for this endpoint would return a response document with the primary data being the resource identifier object.

```
GET: /books/1/relationships/authors
```

So a request to the URL would be something like this:

```

{
  "data": {
    "id": "1",
    "type": "authors"
  }
}

```

Notice how the primary data has the same structure as the data of the relationship of the previous example? This is because a request to relationship endpoint returns resource identifier objects instead of resource objects.

For a **one-to-many** scenario, where a book can have many authors, the response document would contain the following:

```
{
  "data": [
    {
      "id": "1",
      "type": "authors"
    },
    {
      "id": "2",
      "type": "authors"
    }
  ]
}
```

Even though two resources can have a defined relationship, it's not always a guarantee that they have a relation at the moment. In this case, we follow the same conventions as with resources.

In the case of a **one-to-one** relation, a request to the **self** link would give a response document like this:

```
{
  "data": null
}
```

In the case of a one-to-many relation, a request to the self link would give a

response document like this:

```
{  
  "data": []  
}
```

And the pattern continues.

We know the conventions for requests and responses to get data for both resources and relationships. Now, it's time to look at how to create or modify data.

### *Creating or modifying data*

As an interface for your application, an API can give your consumers the ability to create or modify data too. APIs for services like Dropbox or Google give the consumer the ability to add data into their services. This can be data like files, documents and much more. Unless you are writing an API for a service that only provides readable data, like a weather service, you most likely would like a consistent way to add data to your application through your API.

Once more, we will be looking at HTTP verbs covered earlier in this book, as well as status codes required by the JSON:API specification.

### *Creating*

As mentioned earlier in this book, we use the POST HTTP verb to post data to our APIs. This is not any different with the JSON:API specification. The thing that does matter is the request document sent to the server.

In this regard, we should follow the convention we've been through earlier about resource objects and relationships. A request to create a new book would



look something like this with a request to the following endpoint

```
POST: /books
```

```
{
  "data": {
    "type": "books",
    "attributes": {
      "title": "Build an API with Laravel",
      "summary": "Learn how to build an API using the Laravel
        Framework",
      "publication_year": "2019"
    }
  }
}
```

The primary data in this request document is a single resource object that describes the new book to be created. Note that in contrast to the response document, we do not include the **id** member. The JSON:API specification states that you can do that, but we recommend you don't, unless you have a very specific scenario where it's needed. It would be better to let the server and backend take care of this, especially if you are using incremental IDs, but even with UUIDs, it's better to let the server take care of the generation of these. The way we see it, it is a concern that should not be placed on the client side.

When creating a resource, you can also define a relationship. Let's see how that looks, using that same example from before with this request:

```
POST: /books
```

```
{
  "data": {
    "type": "books",
    "attributes": {
      "title": "Build an API with Laravel",
      "summary": "Learn how to build an API using the Laravel
        Framework",
      "publication_year": "2019"
    }
  },
  "relationships": {
    "authors": {
      "data": {
        "type": "authors",
        "id": "1"
      }
    }
  }
}
```

Above, you can see how we reuse the convention about relationships. We give a resource identifier object through the **data** member and the server will take care of the rest.

The example given here is of a **one-to-one** relationship but of course, you can also create a relationship in the case of a **to-many** relationship.

That would look like this, again with this request:

```
POST: /books
```

```

{
  "data": {
    "type": "books",
    "attributes": {
      "title": "Build an API with Laravel",
      "summary": "Learn how to build an API using the Laravel
        Framework",
      "publication_year": "2019"
    }
  },
  "relationships": {
    "authors": {
      "data": [
        {
          "type": "authors",
          "id": "1"
        },
        {
          "type": "authors",
          "id": "2"
        },
        {
          "type": "authors",
          "id": "3"
        }
      ]
    }
  }
}

```

As shown in the example, you give a collection of resource identifier objects in the data member instead.

### *Creating Relationships*

If you recall the relationship links we covered earlier when talking about relationships between resource objects, the endpoint for such a relationship link could look like this:

```
GET: /books/1/relationships/authors
```

You can also create, modify or delete relationships by sending a request to these endpoints, where the only difference is that the primary data of your request document becomes the new resource identifier object instead.

As an example, we can create a new relationship between a book and an author like this with a request to the relationship endpoint:

```
POST: /books/1/relationships/authors
```

```
{
  "data": {
    "type": "authors",
    "id": "1"
  }
}
```

As you can see, it is far less data we need to provide as the request document's primary data. In fact, it's only the **resource identifier object** we need to give. The rest of the information about which book and so forth is given in the endpoint.

The same concept works for **to-many** relationships with a request to the same endpoint:

```
POST: /books/1/relationships/authors
```

```
{
  "data": [
    {
      "type": "authors",
      "id": "4"
    },
    {
      "type": "authors",
      "id": "7"
    },
    {
      "type": "authors",
      "id": "18"
    }
  ]
}
```

The endpoint is reused since it's still a relationship between a book and authors.

If a request to create a relationship that already exists occurs, the server should ignore the new request and keep the relationship as it was before.

## *Status Codes*

When creating a resource, the server must respond with a **201 Created** status code. Remember from the Status Codes section in the first chapter, how this status code is used to tell the client that one or more resources have been created. The JSON:API specification follows that same convention. The primary data of the response document must also contain the newly created resource objects as well as a **Location** header that provides the location of the new resource.

If a request to create a new resource is unsupported, the server should respond with a **403 Forbidden** status code.

Now that we know how to create resources according to the JSON:API specification, how do we then modify these data? We will cover that in the next section.

## *Updating*

As mentioned in the section about HTTP verbs, there are two methods for updating resources. PUT will replace all data in the resource whereas PATCH is only used for updating specific attributes.

The JSON:API specification specifies that the PATCH verb should be used to updating resources, even if all the attributes are updated. Though a request doesn't have to include all the attributes of a resource, the server would have to interpret the missing attributes.

Let's take a look at how to make a request to the update endpoint:

```
PATCH: /books/1
```

```
{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Hello world",
      "summary": "This book is about hello world."
    }
  }
}
```

If you take a look at this example and the examples of how to create a book, do

we send the exact same attributes? No, we are missing the **publication\_year** attribute in the resource object. The server would then have the responsibility to interpret this resource object, find the existing one, and update only the attributes given here.

When updating relationships, some of the same concepts repeat.

Just like with creation of resources, updating a **relationship** can be done through the **relationships** member on the root of the request document. Like with attributes, where the server has to interpret which attributes you wish to update, the same goes for the **relationships** given in the **relationships** object. Only the relationships mentioned are the ones getting updated. Let's look at an example again with a request to the same endpoint:

```
PATCH: /books/1
```

```
{
  "data": {
    "id": "1",
    "type": "books"
  },
  "relationships": {
    "authors": {
      "data": {
        "type": "authors",
        "id": "2"
      }
    }
  }
}
```

Here, the relationship to authors is being updated. In this case, it's a **one-to-**

**one** so the relationship to the author before is being changed to a new author with the **id** of 2. Just like when creating relationships, you give a resource identifier object and the server will take care of the rest.

And as with creating a resource, the way to update relationships with a **to-many** relation you give a collection of resource identifier objects, like this with a request to the same endpoint:

```
PATCH: /books/1
```

```
{
  "data": {
    "id": "1",
    "type": "books"
  },
  "relationships": {
    "authors": {
      "data": [
        {
          "type": "authors",
          "id": "2"
        },
        {
          "type": "authors",
          "id": "4"
        },
        {
          "type": "authors",
          "id": "7"
        }
      ]
    }
  }
}
```



```
}
```

In the example above, we give an array of resource identifier objects that we want to update as the new related resource. It is important to note that in this case the relationship to any earlier resources will be removed and relations to the given resources will be made instead.

You can think of this kind of update as the **sync** method on Laravel Eloquent's **many-to-many** relationships. Here, you give the **sync** method an array of **IDs** that you want to associate, and all the **IDs** not in that array that may already be associated, will be removed.

As mentioned in the creation of resources, relationship links can also be used to create, modify or delete relationship between resources. Here's an example with a request to update the author of a book, using the following relationship endpoint:

```
PATCH: /books/1/relationships/authors
```

```
{
  "data": {
    "type": "authors",
    "id": "1"
  }
}
```

As with relationships defined through resource objects, updating a relationship through relationship links also replaces the relationship before. You could go and delete a relationship by making the following request to the same endpoint as before:

```
PATCH: /books/1/relationships/author
```

```
{
  "data": null
}
```

Now, the old relationship will be removed and since an empty relationship has been given, there is nothing new to save instead.

Again, the examples have been on a **one-to-one** relationship, but the concepts are the same for **to-many** relationships. A request to update authors as a **many-to-many** relationship would be like this, with a request to that same endpoint:

```
PATCH: /books/1/relationships/authors
```

```
{
  "data": [
    {
      "type": "authors",
      "id": "2"
    },
    {
      "type": "authors",
      "id": "4"
    },
    {
      "type": "authors",
      "id": "7"
    }
  ]
}
```

```
]
}
```

And just like with the single resource object, where a request document with the primary data set as null clears a relationship, an empty array clears a **to-many** relationship like this:

```
{
  "data": []
}
```

## Status Codes

When updating a resource, the server must send back either a **200 OK** status code or a **204 No Content** status code in case of an update where nothing else than the attributes provided in the request document was updated.

In an attempt to update a resource that does not exist, the server should send back a response with a **404 Not Found** status code to indicate that the resource is not found.

## Deleting

Things are moving fast. We now know how to both create and update resources and relationships. We only need to know about deletion, so we are almost done.

Just like requests for creating or modifying a resource, there is a dedicated HTTP verb for deletion of a resource. If you look back to the section about HTTP verbs, we discussed the DELETE verb and that it's used to tell the server that we want to delete a resource. The JSON:API specification uses the same

HTTP verb for deletion of resources.

To delete a resource, you don't need a request document since the HTTP verb says it all — at least when it comes to deletion of a resource. Here, it is enough to send a request to an endpoint for a specific resource. For example, imagine that we want to delete the book of **ID 1**. This can be done with a request like this:

```
DELETE: /books/1
```

The same goes for relationships. If you make a DELETE request to a relationship endpoint, the server must delete the relationship between the specified resources. Say, for instance, we want to remove an author from a book. We can just send a request to the endpoint like this:

```
DELETE: /books/1/relationships/authors
```

In case of a **to-many** relationship, you can specify which resources you no longer want to have a relation through a request document, where the primary data is a collection of resource identifier objects. Just like when we want to create or modify a relationship through relationship links. Say we have a book with five authors and we want to remove three of them. This can be done so with a request to the following endpoint:

```
DELETE: /books/1/relationships/authors
```

```
{
  "data": [
    {
      "type": "authors",
      "id": "2"
    },
    {
      "type": "authors",
      "id": "3"
    },
    {
      "type": "authors",
      "id": "5"
    }
  ]
}
```

Now, the relationship between the book and the three authors mentioned in the request document will be deleted, while the rest will remain.

This is actually all there is to requesting, creating, modifying and deleting data, which is enough for us to work with our server with conventions that ensure a consistent communication and data exchange. There is one thing we are missing though. What do we do when an error sneaks in and how do we respond to that?

## *Errors*

It's time to take a look at errors. Let's face it, no matter how great of an application you build, errors will always be a part of it. Some are intentional, like validation rules that are not met, and some are not. In both cases, it's crucial that they are handled in a consistent way, so both you and the consumer of your API know what went wrong.

To start this section, we will take a look back at the top-level of our response

document. If you recall, a document must contain at least one of the following members:

- **data**
- **errors**
- **meta**

A rule we covered earlier was that the **data** and **error** members must never coexist in the same document. So from now on, whenever we talk about errors, we will never include the data member in our response document.

The **errors** member, however, will be an array containing error objects that describe errors of the request made to the server like this:

```
{  
  "errors": []  
}
```

We will take a look at error objects in a moment, but one thing we must cover is how the JSON:API specification states which error status codes should be used.

When an error occurs in your application, you as the developer can decide whether or not the application should continue to try to fulfill the request or if it should just fail and stop.

In the case of a fail and stop solution, you should use the appropriate status code. When we looked at how to get, create, modify or delete resources and relationships, we talked about the status codes that should be used.

If you choose to let your application continue trying to fulfill the request, there may be multiple errors that can happen in that single request. In that case, a

more general status code should be used.

- In the case of a 4XX category, error use a **400 Bad Request** status code
- In the case of a 5XX category, error use a **500 Internal Server Error** status code

## *Error Objects*

Error objects are used to further specify what went wrong when trying to fulfill the request. The JSON:API specification does not have a strict protocol for how this object should be constructed, but a couple of optional members you can include. We have concluded that the following are useful to have in an error document and would recommend that you use these.

- **title** – which should contain a short human-readable summary of the problem.
- **detail** – which should contain a human-readable explanation of the problem.
- **source** – which should contain an object containing references to the source of the error.

The **source** member is a bit special here, since the JSON:API specification recommends that you use an object containing a JSON pointer member. A JSON pointer is a string syntax that can point to another value in a JSON document.

If you have the following JSON document:

```
{
  "foo": "bar",
  "baz": [10,20]
}
```

A JSON pointer described like this, **/foo** would then point to the value **bar**, where a JSON pointer like this, **/baz/0** would point to the value **10** and a JSON pointer like this, **/baz/1** would point to **20**.

You can think of JSON pointer like the dot notations Laravel provides when you, for instance, are accessing values in your config. Here, you are able to write like this:

```
<?php

// Get the application name from config
$name = config('app.name');
```

This will give you the application name defined in the **app.php** config file. Instead of a **dot** notation for separating children, a **slash** notation is used in JSON pointers.

Now that we know about JSON pointers, let's look at how an error object could look like:

```
{
  "errors": [
    {
      "title": "Validation error",
      "source": {
        "pointer": "/data/attributes/name"
      },
      "detail": "The name field is required."
    }
  ]
}
```

In this example, we are getting a validation error in our application like the **title**



member describes. The **source** object contains a JSON pointer to an attribute in the response document sent to the server. Here, it's the **name** attribute that might be empty, based on the description given in the **detail** member.

How would it look if we had more than one error? Right out of the box, Laravel won't continue to process a request, whenever an error occurs. It will just throw an exception and stop the execution. When handling validation though, it is possible that more than one field would fail and here you would get multiple errors returned. That would look something like this:

```
{
  "errors": [
    {
      "title": "Validation error",
      "source": {
        "pointer": "/data/attributes/name"
      },
      "detail": "The name field is required."
    },
    {
      "title": "Validation error",
      "source": {
        "pointer": "/data/attributes/email"
      },
      "detail": "The email must be a valid email address."
    },
    {
      "title": "Validation error",
      "source": {
        "pointer": "/data/attributes/age"
      },
      "detail": "The :attribute must be a number."
    }
  ]
}
```

As you can see, there's an error for each field that did not pass the validation. Although the JSON:API specification states that you should give a more general status code when having multiple errors, in this case, it's ok to give a **422 Unprocessable Entity** status code.

## *Summary*

We're at the end of this chapter and we have covered quite a lot. Not only did we introduce the case of Anna's Bookstore, which we will use as a common ground to easily put things into perspective in the rest of this book. We also covered naming conventions for API endpoints, the JSON:API specification, and especially all of the important rules that are specified.

We looked at how to structure our data or documents for both requests and responses, how to structure our primary data of a document, both when handling single resources and collections of resources.

Then we covered resource objects and which members we must include like **id** and **type**, but also learned how to name our member names for a more consistent structure when adding attributes to a resources object.

We then moved on to relationships and how to define relationship links that make it possible to access both relationships themselves and related resources objects. We covered how to define resource identifier objects and how these could be used together with the **includes** top-level member, to contain related resources in a response in order to save HTTP requests.

Afterward, it was time to look at how to use our new knowledge about data structures to properly communicate between client and servers. We covered how to get resources, and how we could use request documents to fetch the exact resource or collections of resources needed. We looked at how we could sort and paginate resource collections, through both query parameters and a **links** top-level member.

We then looked at how to create resources and how to structure our request documents, how we could create a relationship through request documents for resource creation, but also how relationship links could be used to create relationships, using relationship linkages instead of resource objects.

We covered how to update resources and relationships using the same principles when creating these, and how an update to a relationship would always remove the current relationship to a resource and then update with the newly given resource instead.

We then learned how to delete both resources and relationships.

Finally, we concluded the chapter by looking at errors and how to convey these through error objects.

We now have a good set of conventions and a knowledge to use when we are building our APIs. We have rules that give us a more consistent way to not only structure data, but also how to communicate and what to communicate. Better yet, when implementing this specification, clients and consumers can use client implementations that will work right out of the box, and save them a lot of time when consuming your API, since these client implementations follow the same conventions and consistency.

It's now time to look at how to plan out an API. Even though we have a lot of knowledge about convention and rules to build a consistent API, we still need to plan out our project and documentation for our API.

Let's take a look at that in the next chapter.

\* \* \*

# 3

## Planning

Now that we have the basics covered and have been through the JSON:API specification, it's time to talk about planning.

For us, this is the most important part of a project. Whether it's a small, medium or large project, it is always beneficial to do some planning ahead of time to identify possible problems and roadblocks you can steer around. For us, time equals money so we need to be as fast and efficient as possible.

The same goes for APIs. The more you can plan out ahead of time, the more it benefits you later, when you actually have to implement the API.

If you are making a public API, we especially recommend planning ahead and documenting your API as early as possible. In this way, you know everything about the data and what needs to be developed ahead of time, as well as getting a good grasp of where the more complex areas of your application lie.

The planning tools and methods we will be going through in this chapter are those that we use when planning out our projects. These are the methods and tools we have picked out over time, that fit most of our projects.

## A different way of planning

If you have never written an API before, you need to think of your applications in a new way.

If you are used to building Laravel Applications in the more “old-fashioned” way, where Laravel primarily render all the HTML and returns this to the user’s browsers, thinking in APIs is completely different.

You can stop worrying about user interfaces and how the application should look and feel. You still have features, but it’s in the form of tasks that the backend should perform. The actual worries about UI / UX can be handed over to the frontend developers on the client side and you get to worry about data instead. That can still seem like a daunting task, but remember that the difficulty of projects vary, and in some cases, your backend doesn’t even have to be that smart. In some cases, its only purpose is to ensure that data is stored and retrieved from databases correctly and that’s it. It is quite a contrast to what we are used to, with applications that do all the work on the server.

In most of our newer projects, the client does most of the hard work. Through single page applications, which are Javascript powered web applications, we can build a much more intuitive user experience than ever before. It almost seems like you’re interacting with a real desktop application and not a web application. In fact, it is possible to build desktop application this way too, but it is beyond the scope of this book.

The HTML responsibilities have been removed from the backend so it can focus on tasks like saving/retrieving data, whether it’s from a database, external services or APIs, to sending emails, encoding audio or video and so on.

And of course, this has changed the way we plan our projects. We are still doing UI / UX but it is now separated from the planning and development of the backend. When planning our backend, we can focus on data and how

these data need to be delivered, retrieved and related to each other. Instead of templates, views and actions, we are thinking in resources and relationships.

This makes it possible for us to plan our backend development with the API in mind, since this will be the new interface. By defining our resources and starting to document these, we are forced to think about our applications and the requirements of the backend. You see by documenting the API, we need to think about which data that needs to go into our application and also how we want the application to respond. This isn't easy if you are doing everything from scratch, but remember that we have the JSON:API specification to lean on, so we know exactly how our request and response documents should be formed. We only need to think about the attributes of our documents.

With the documentation done in the planning phase, we not only have everything planned, we get the documentation out of the way. An even bigger plus is that it makes it possible for the frontend developers to mock our API and that makes it possible for the development to be done almost in parallel.

So let's start thinking in these new ways and begin to plan out our API.

## Requirement Specification

The first step in planning a project is to identify all the requirements for the project and put them in a requirement specification. This is especially important since it describes all the features that should be implemented and also how these features are expected to work. It's a document that gives a consensus between the customer and the developer.

Through an interview of the customer, you should take notes of what they tell you about the project they would like you to develop. When interviewing, it's important to consider how we as developers know all the terms in the business, but our customers might not — especially not someone like Anna from the hypothetical coffee shop of our case example. Because of this, it is

our responsibility to take good notes and talk to the customer in a language they understand, and afterward transform those notes into clear requirement specifications that both parties can understand and agree on. Of course, it's also our responsibility to cover as much as possible, especially if the customer does not know a lot about software development. In that case, small sketches can be a huge help, particularly when trying to convey functionality between you and the customer, but also when you have to write the requirement specification later on.

It always pays off to be thorough and leave no stone unturned, so take good notes and be sure to write down all the important parts of the interview. If you are not that good at note taking, then ask if you can record the interview with an audio recorder. The customer is just as interested in a good product in the end as you are, and oftentimes they don't mind being recorded.

When you have your notes after the interview, it's time to write the actual requirement specification.

We recommend that you start out by identifying the main areas of the application you're going to build and divide your specification into these areas. In terms of a bookstore these could be:

- Public store
- User Profile
- Administration

In a case like this, we would recommend starting out with something easy that you are familiar with to get the ball rolling. You should be familiar with how authorization and login work, so start out with that. In the areas above, there are two that need access restrictions and the authorization methods might not be the same. Start by making requirements for the authorization for one of them.

Write each requirement in bullet form and make indented bullets if something needs to be explained further like this:

- The administrator should be able to enter an email address
  - – The email should be of the bookstore’s own domain to be allowed as administrator
  - – The email cannot also be used for a regular user profile.
- The administrator should be able to enter a password
  - – Passwords should be over 6 characters
  - – Passwords should contain at least one capital letter
  - – Passwords should contain at least one number
- The administrator should be able to tick a “remember me” checkbox
  - – When ticking the remember me checkbox, a cookie should be saved for easier logins in the future
  - – A “remember me” cookie should only be valid for 48 hours
- The administrator should be able to click the login button to proceed
- The administrator should be able to see if he/she has entered the email or password incorrectly.

The language in each requirement should most often contain “**should**”, “**must**” or “**cannot**” to clearly signify that it is a requirement for the project.

In the example we gave, some of the requirements relate to the flow of a login, whereas others are business rules, like “The email cannot also be used for a regular user profile”. In some cases, there can be more requirements that describe business rules, rather than the flow. Here, we recommend writing a little scenario to accompany the requirements of the area you are describing. This would look something like:

*The administrator has arrived to the login screen.*

*She types in her email in the email section and pushes the tab button on her keyboard to get down into the password space, where she types*



*in her password as well. Before clicking the login button, she ticks off the “remember me” checkbox so she doesn’t have to perform the same procedure, if she comes back within the next 48 hours.*

Now both the requirements and the scenario describes the authorization precisely. We know what should be developed and the customer knows what to expect.

As you can see, a requirement specification describes the entire project, not just the API, but there’s a lot we can get from the requirement specification as well as the other developers working on the project.

Some of the logic described in the requirements can be on either the frontend or backend and in that case it might be something our API should support.

From a broader perspective, let’s start to focus more on our API. To do this we first want to introduce you to Postman, which we will be using throughout the most of this book.

## Postman

Postman started out as a small in-browser application for Google Chrome in 2012. Later, the application expanded to native applications running on desktop machines, and now it’s an entire SaaS platform.

We have used it for years, first as a tool to test out our own, but also especially, third-party APIs. Sure, you can do the same with CURL, but often a nice UI and the ability to remember both requests and responses beat that.

The expansion of the application has also included features like:

- **Multiple Workspaces** - which gives a great separation of projects with the ability to invite others to share the same workspace as well. A workspace

embraces the following features mentioned in this list.

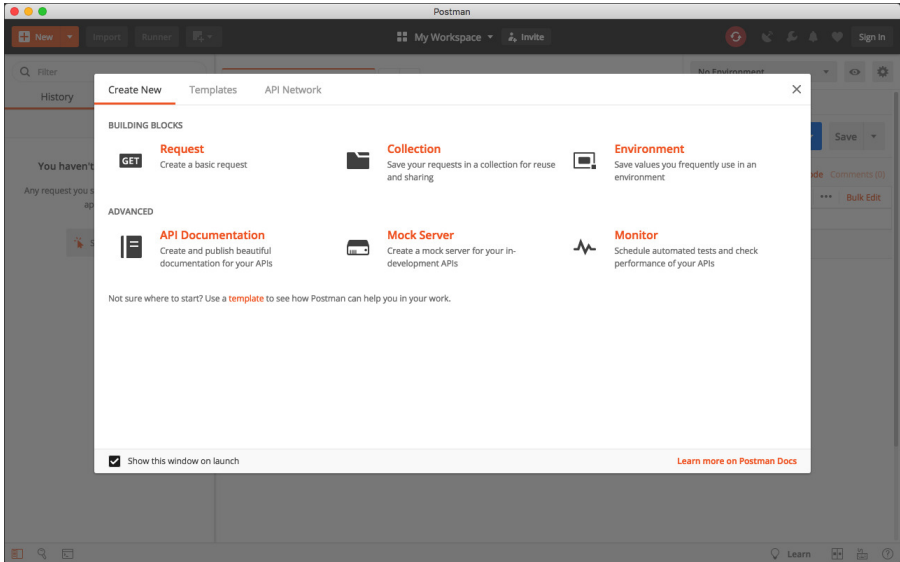
- **Collections** – Which is a grouping of requests, a feature we will be using while planning and developing our API.
- **Mock Servers** – Which is a feature that makes you mock out your API when all endpoints and responses have been planned out. A great feature for frontend developers, which makes it possible to interact with a mocked server, making it possible to work in parallel with the backend developers.
- **Monitoring** – Which is a way to schedule automated tests and thereby monitor how your API performs.
- **API Documentation** – Which makes it possible to create and publish documentation for your API using your collections.

It is especially the ability to test our APIs, the collections, and API documentation we are interested in and will be going through in this book. First, we will be looking at how to plan out our API by setting up a new workspace and identifying resources to determine our endpoints, the attributes of our resources, and later identifying relationships and adding these in as well.

## Download & Installation

To download the application for your platform, follow this link <https://www.get-postman.com> and click on the “**Get Started**” button. This will lead you to the download page, where you can click on the “**Download**” button to get the version for your platform. When it’s downloaded, install the application and when the installation is done, open the application.

## PLANNING

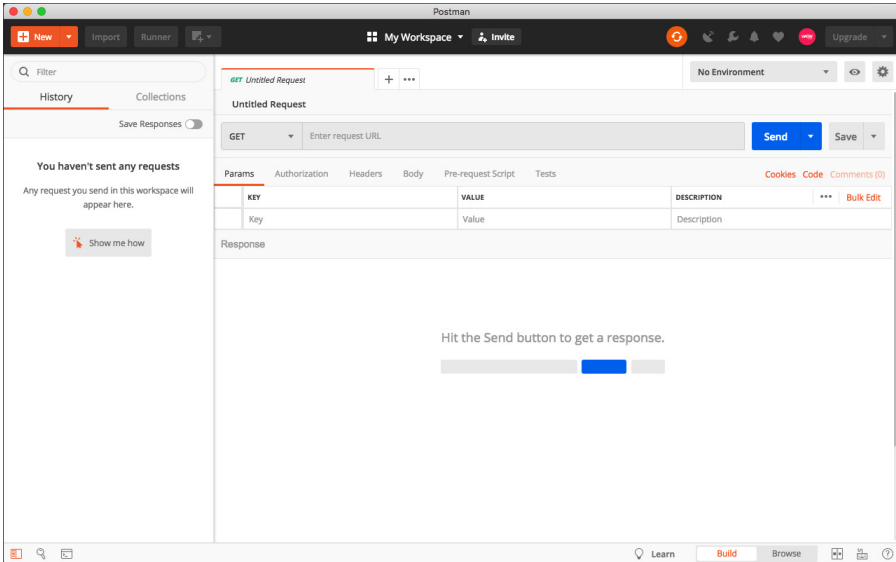


When opening the application, you should be greeted with an interface that looks like the image above. It might prompt you to login and if so, create your account and log in. If you did not get a login, you'll hit it as soon as we start working with Postman and you can then create your account.

If you are at the screen you see above, close the window by hitting the X in the corner.

### *Creating a new Workspace*

## BUILD AN API WITH LARAVEL

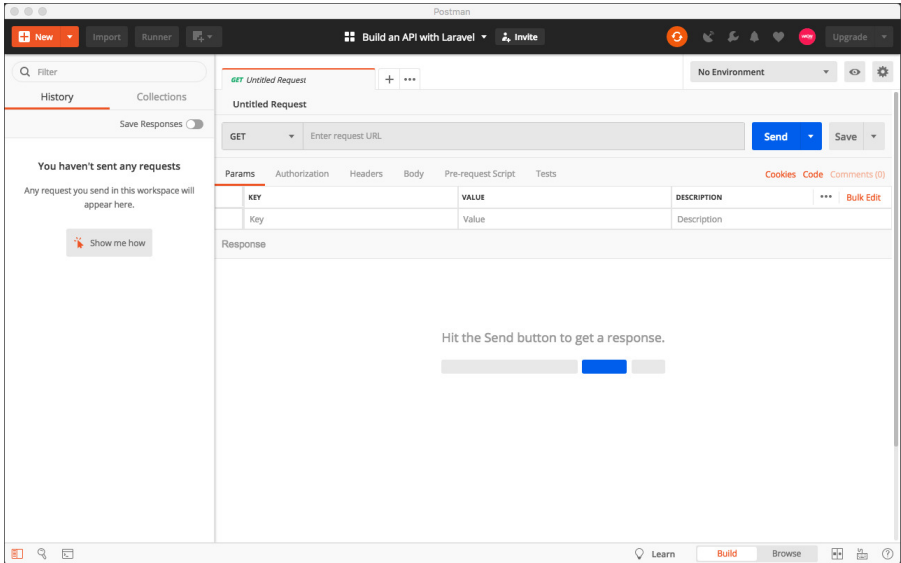


After you have installed Postman, created your account and logged in, you should be seeing an interface like the one in the image above.

It's time to create our workspace for the API we will be building. Click the **"My Workspace"** dropdown at the top of the window. In the dropdown, click **"Create New"** and you will see a form that needs to be filled out to create the Workspace.

Name the workspace **"Build an API with Laravel"** and leave the summary blank. Under **"Type"** choose **"Personal"** and click **"Create this workspace"**.

## PLANNING

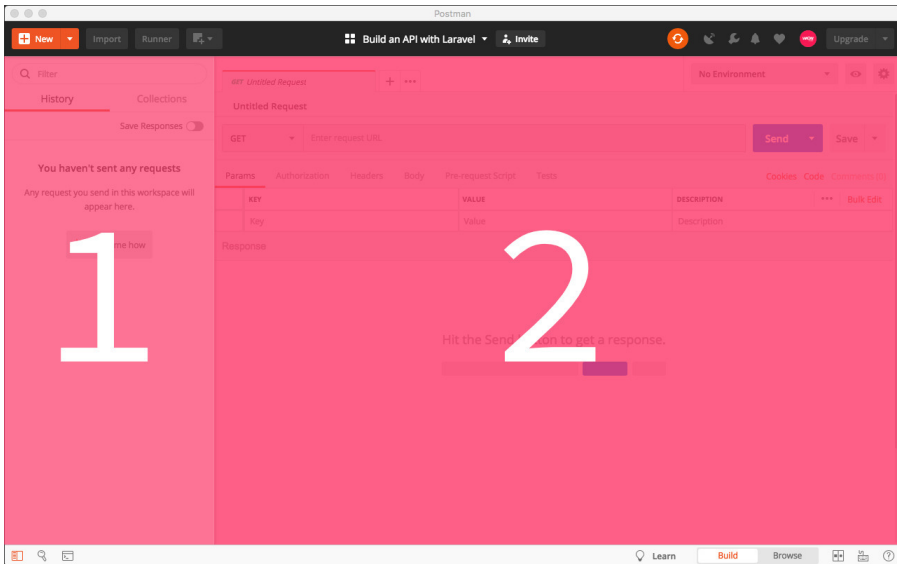


You should then be seeing an interface like the image above.

Great! We now have a workspace for our API, so let's take a closer look at the Postman interface.

### *User Interface*

## BUILD AN API WITH LARAVEL



On the image above, you can see the division of the two main areas of the Postman application.

- Is the side panel where we will create our collection of endpoints for our API.
- Is the main area where we will do our main work, like create requests, define the body of our requests, and later create examples for the documentation of our API.

At first, we will mostly work in the side panel and do some minor work in the main area. When we get deeper into planning our resources, we will do most of our work in the main area.

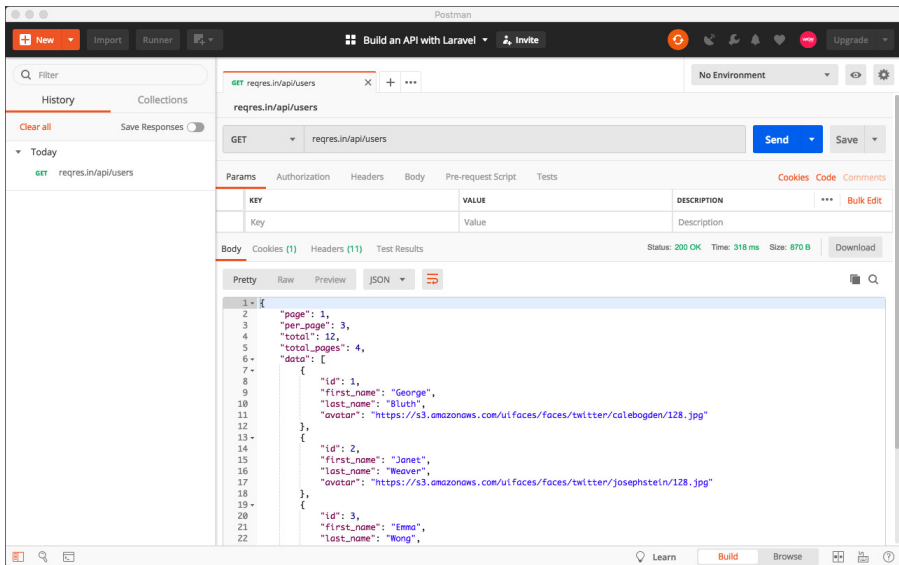
Just to quickly show you the features of the main area, let's try to make a request to the following URL:

## PLANNING

GET: reqres.in/api/users

In the top of the main area, you see a gray input with the placeholder text “**Enter request URL**” and then a selector to the left of this, that has all the HTTP verbs. Directly underneath, there are some tabs and then a small table, where you can fill out params.

This is the request area of the main area. Set the HTTP verb to GET and input the **reqres.in/api/users** URL and hit the “**Send**” button.



Now you can see the response document from the server underneath the request area formatted nicely, so you can see every bit of data that are sent from the server. This is the response area. Here you can also look at the cookies and headers sent from the server.

If you feel like the request and response areas blend too much, you can switch

to “**Two-pane view**” by clicking icon down in the bar in the bottom. This will make the request and response areas sit side-by-side instead, except for the actual endpoint and HTTP verb of the request. For clarity, we will be using this mode for the rest of this book, since it makes it easier

to see the difference, in which data that are being sent in a request, and which data that have been received in the response.

The **reqres.in** is a fake REST API you can use to test your frontend or programs like Postman against a live API. If you want to explore Postman’s capabilities more in depth, we recommend that you spend some time going through the various requests you can make to the API using Postman. We will, of course, return and go through more of the features when we start building our own API.

Before we move on though, take a look at the side panel. It should be on the “**History**” tab where you can see the latest requests you have made. Next, we will be looking at the “**Collections**” tab in the side panel.

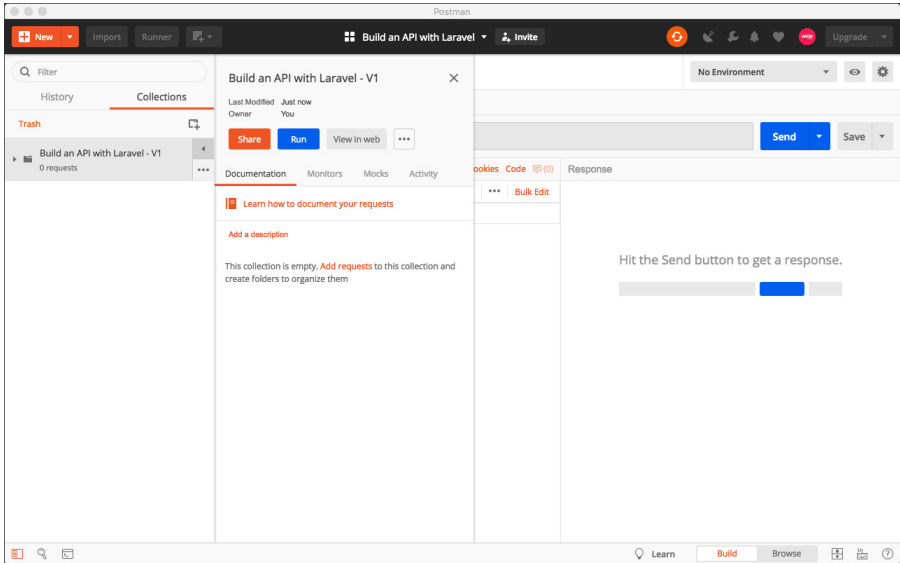
## *Collections*

Click on the “**Collections**” tab in the side panel. Here, you will be greeted by a message saying “**You don’t have any collections**”.

Click on “**Create a collection**” to create a collection. In the overlay that opens, name it “**Build an API with Laravel - V1**”. In a workspace like this, it is possible to have multiple collections where each collection will produce its own documentation. If we were to update our API in the future, it would be possible to produce new documentation to keep the two versions separate.



## PLANNING



Leave everything else as it is, click the “**Create**” button, and you should get a new collection like in the image above.

Click the small arrow to the left of the folder icon and you will see a message telling you that your collection is empty, and you can add request and create folders to organize your request.

A request in Postman is a request to a single endpoint. This could, for example, be a request to the following endpoint to get alle books:

```
GET: /books
```

Another request could be for the following endpoint to get the book with an ID of 1

```
GET: /books/1
```

Another could be to the following to create a book:

```
POST: /books
```

Hopefully, you see a pattern here and see that all requests are for the book resource. Instead of having these in the root of our collection, we can make folders and give these the name of a resource. This will group all requests for that resource and later get a nice separation of requests in our documentation.

Before we do that, let's revisit how we identify resources.

## Identifying Resources

It might seem strange that we use Postman as much in our planning phase, but since it's the platform in which we will create our documentation, you can get ahead by starting to document your findings in your planning phase right away. Nothing is written in stone, we can still make changes as we go, but instead of putting things on paper, why not put it into something useful that can save us time later on.

In chapter 2 when we covered naming conventions for endpoints, we talked about the convenience of thinking about our models in a Laravel application as resources.

If we take another look at the case presented in chapter two, we have been given the following objects of real life:

- Books

- Authors
- Comments on a single book

As Laravel developers, we would quickly identify that we need a model for each of these things and that's absolutely right. But we are here to plan and here to think ahead and find roadblocks that we might have to steer around. If you ask us, that specification up there is only telling half of the story.

In an application like this, there must be someone who creates new books and updates books, or deletes the books that are no longer on sale and the same goes for authors. The persons we are referring to here are administrators. In order to service the bookstore, we need administrators to perform these tasks.

What about comments, who leaves those? The common thing here would be a registered bookstore user. They can leave comments on books and possibly in a future implementation, also leave ratings of these.

If we step back a little and actually look at the things we just discovered about administrators and users, we can see that we have a multi-tenant application on our hands.

A tenant is a group of users that share the same access levels in an application. Since we have administrators, who are allowed to administer books and authors, but also users that are allowed to comment on books, these are two different tenants and therefore it's a multi-tenant application.

There is not a standardized way of making multi-tenant applications in Laravel, but a common pattern is to use a “**Role**” or “**Type**” attribute on the User model to differentiate the users and thereby only using one model for handling users and administrators. We will be using this strategy in our API, where we will be using a **role** attribute to not confuse anything with the already existing **type** member coming from the JSON:API Specification.

## Identifying Attributes

When identifying attributes, you can rely on the same methods you use when thinking about columns in your database or attributes on your models in Laravel.

If a book is an object we are trying to model into our database, the columns convey more about the book. That could be things like:

- A title
- A summary
- Number of pages
- Publication year

Laravel can help you send all these data in the response of your API, but also filter those parts you do not want to give. In the end, you decide how much information you want to give. In terms of planning, we often use the columns we identify when planning our database.

This gives us the ability to start documenting a lot of our APIs early on and also provides the ability to start mocking the API so that the frontend development can begin early on as well.

## Identifying Relationships

As important as it is to identify resources, it is just as important to identify relationships. These are not only to be defined in your API, but your models as well, and the type of the relationship defines the complexity.

It is also the first step where you really have to think about your data and how you are solving the given problems, as well as whether or not decisions you make have an impact when the end-users start working with the application.

## Identifying the right relationship

Let's examine the first and most obvious relationship between **Books** and **Authors**. Here a **one-to-many** relationship would be the easiest to implement. From a database perspective, the book could have a foreign key pointing to the author who has written the book, and from a Laravel Eloquent perspective, a book would belong to an author and so forth.

If you didn't stop and think about it, a **one-to-many** relationship might be a constraint on the applications. Because of that decision, there can only be one author for each book in the bookstore, or at least only one author would get the credit.

This is a roadblock we want to identify early, so that we can steer around it. Here, it would be much better to have a **many-to-many** relationship since a book can be written by many authors, but an author can also have written many books. By identifying this early and not having to change this later when the code is already written, we have potentially saved hours.

In terms of an API, the relationship here would be the same. Since we do not have a **one-to-one** relationship, we know that the relationship will involve collections.

## Identifying the remaining relationships

Let's move on and identify the remaining relationships. Since we are already in the vain of books, why don't we look at the relationship between books and comments next?

The relationship here is not that complex, a comment will only ever be for one book and a book can have many comments, so a **one-to-many** relationship is pretty easy to identify here. In terms of our API, it's not a **one-to-one** relationship, which means that we will still work with collections. There's

one thing here that we might have to note and that is about the users and the relationship between a comment and a user. In the case where you request all comments for a book, it would be nice to also have the users for those comments. Remembering back to the Compound Documents section of chapter 2, this is a good candidate for data to be included in a response, since you would not have to have yet another request just to get the users that the comments belong to. But we are getting a little ahead of ourselves. Let's take a look at the relationship between comments and users then.

Again, it's not that complex. A comment can only really belong to one user so this must be a **one-to-one** relationship, and in terms of our API, we will be handling a single resource identifier object instead of collections.

Let's see the relationships we have identified then:

- **Books and Authors** – A many-to-many relationship with endpoints:
- – **Self**

```
GET: /books/1/relationships/authors
```

- – **Related**

```
GET: /books/1/authors
```

And the inverse relationship from authors:

- – **Self**

```
GET: /authors/1/relationships/books
```

- - **Related**

```
GET: /authors/1/books
```

- **Books and Comments** - A one-to-many relationship with endpoints:
- - **Self**

```
GET: /books/1/relationships/comments
```

- - **Related**

```
GET: /books/1/comments
```

And the inverse relationship from comments:

- - **Self**

```
GET: /comments/1/relationships/books
```

- - **Related**

```
GET: /comments/1/books
```

- **Comments and Users** - A one-to-one relationship with endpoints:
- - **Self**

```
GET: /comments/1/relationships/books
```

- - **Related**

```
GET: /comments/1/books
```

And the inverse relationship from users:

- - **Self**

```
GET: /users/1/relationships/comments
```

- - **Related**

```
GET: /users/1/comments
```

Now we have both identified our resources and relationships, and are ready to begin our documentation. Before we do though, and we promise this is the last thing, let's just quickly take a look at **IDs** and **UUIDs**, since there's a bit to consider here — again to avoid hitting those roadblocks later.

## IDs and UUIDs

When planning out an API, you have to think about what kind of information you are giving away. The things we are talking about here are the **IDs** of your resources. Of course, it's convenient to reuse the IDs given by the database, but is that really a good idea? It depends on the application behind, especially if it's a multi-tenant application. If you take **IDs** from a database, these are most often primary keys in the form of integers that increment chronologically for



each row in that database. If you were building a multi-tenant application, one user could potentially try to access another user's data, by editing the **ID** of the resource being accessed. And given the chronological order of IDs, there's a good possibility that the data of the given **ID** exists in the database.

In this case, it would be a better idea to use a **UUID**.

A **UUID** or **Universal Unique Identifier** is a 128-bit number we can use to identify entities in our applications. These entities could be users in your application: they could be books, they could be virtually any table in your database, but the key here is that they are used to identify information in our systems. **UUIDs** can be generated by anyone and does not require some kind of central registration authority to keep track of each **UUID** created. This does mean that there is a probability that a **UUID** could be duplicated, but the chance is very close to zero and would not happen in the same system at least, which makes **UUID** eligible to use for primary keys in relational databases, which we use for Laravel.

The fact that we get a 128-bit unique number has another advantage, namely that an **UUID** is near impossible to guess.

There are 5 different versions of the **UUID** standard where each version generates a **UUID** in different ways. As an example, the first version uses a date-time and the **MAC** address of the machine in which it is being generated to generate a unique **ID**.

The version we use the most is version 4, which does not use **MAC** addresses and the like, but generates a random number.

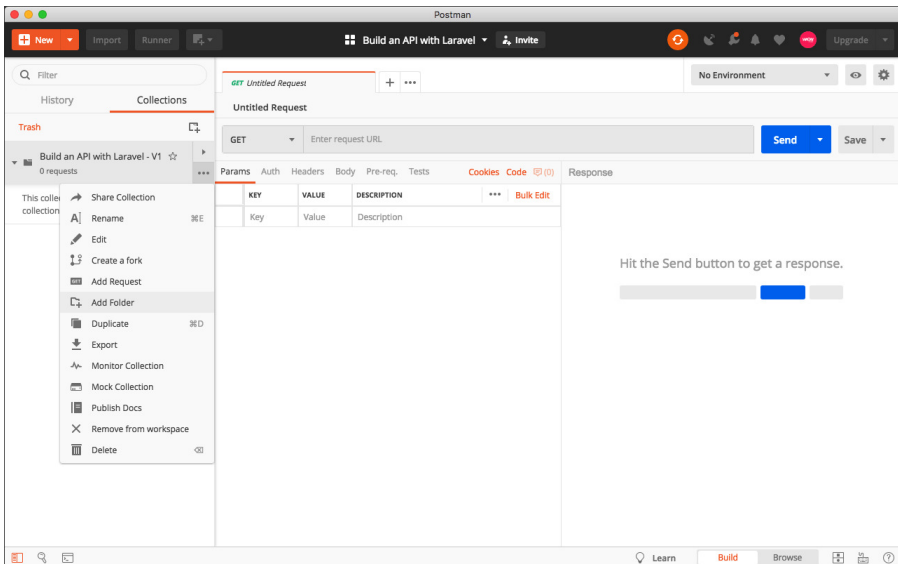
It doesn't mean that you always have to use **UUIDs**. Sometimes, you also have to consider ease of use for your API, especially if it's public. And if the **IDs** don't present a security risk, you could keep the **IDs** from the database.

## Beginning our documentation

We are finally ready to begin our documentation since we now know which resources and relationship we need in our application. Let's start by documenting resources and endpoints, and afterward we'll go more into detail with attributes and relationships.

## Folders for Resources

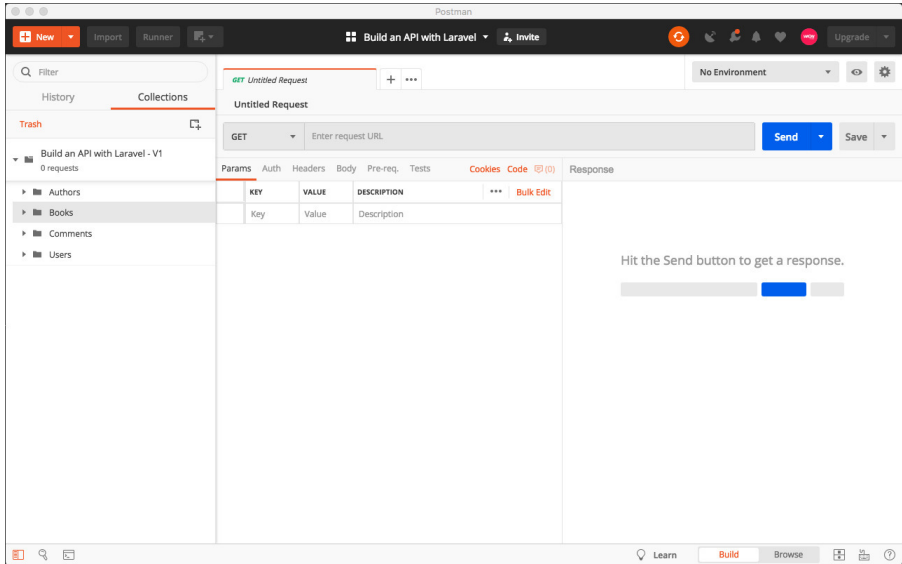
Firstly, to have a nice segregation of resources in our collection and later our documentation, we want to create **Folders** for each resource. By clicking on the three dots, we get a menu in which you will click on **"Add Folder"**, as shown in the image below.



In the overlay, give the folder the name **"Books"** and click on the **"Create"** button. Repeat the same procedure for the rest of the resources which are:

## PLANNING

- **Authors**
- **Comments**
- **Users**



When you're done, you should have something that looks like the image above.

### *Resources Requests*

In the next sections, we will be going through the entire documentation process for the **Books** resource only. But don't worry, with the methods used here, you will be able to document the rest of the resource yourself, plus it will give you hands-on experience with both Postman and documenting your API.

To make a request in a resource folder, you need to click the three dots that appear when you hover over the folder name.

Select "**Add Request**" and name the first request: All Books and click "**Save**

**to Books”**.

In the main window set the method to GET, enter the URL **/books**, and click the **“Save”** button next to the **“Send”** button.

Create another request and name it: Single Book and click **“Save to Books”**. Give it the method GET and URL of **/books/1** and click the **“Save”** button next to the **“Send”** button.

Create yet another request and name it: Create Book and click **“Save to Books”**. Give it the method POST and URL of **/books** and click the **“Save”** button next to the **“Send”** button.

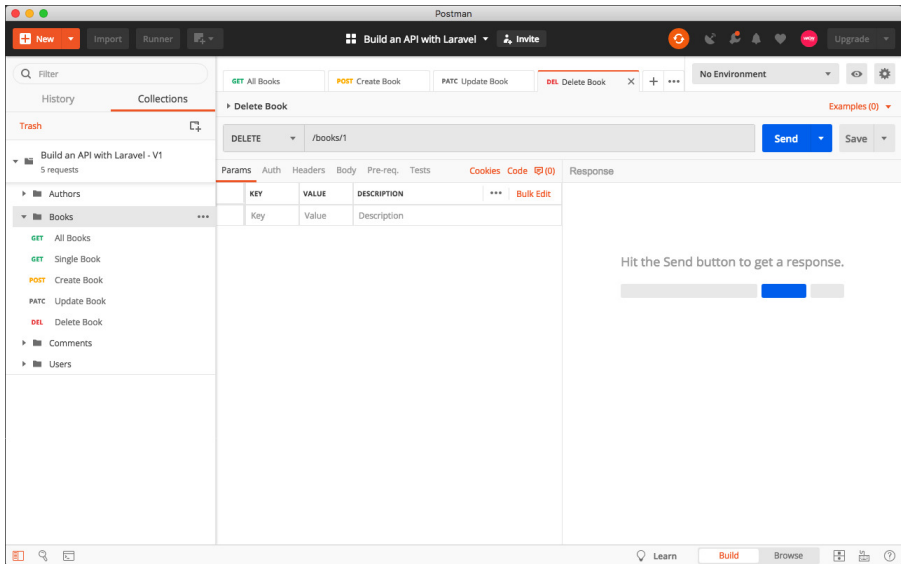
Create the last requests for:

- Update Book
- Delete Book

The Update Book needs a PATCH method and the Delete Book needs a DELETE method. Both have the same URL:

```
/books/1
```

## PLANNING



You should have something that looks like the image above.

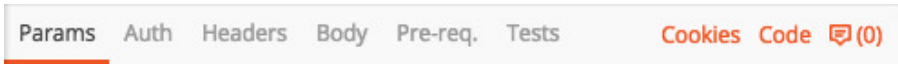
Now, just for the fun of it, let's see how our documentation is taking form. If you look at the status line in the bottom, there are two buttons in the lower right named "**Build**", which is active at the moment, and "**Browse**". Click the "**Browse**" button and click the name of your workspace. You will now be presented with a somewhat raw version of our documentation, but this gives you an idea of what we are slowly building as we give Postman more information. Look how our folders provide a nice structure to the menu. We still need all of our attributes and also the responses that can be expected. Let's look at the attributes next. Go back to Postman and click on the "**Build**" button.

### *Resource Attributes and Request Document*

When thinking in attributes, do you then know when we need to document these? Right, it's when creating and updating our resources. Let's take a look at the **Create Book** request in Postman to set up our request document and

add the attributes.

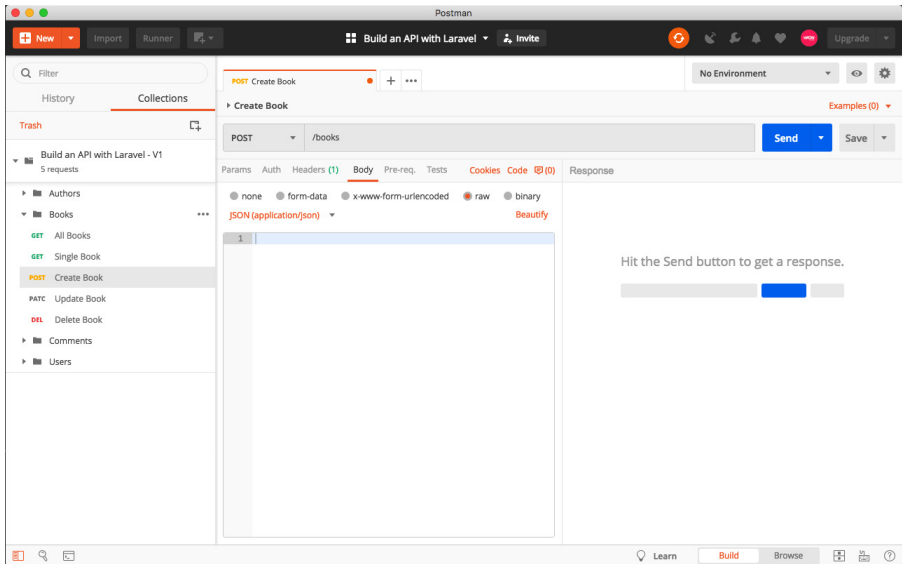
Remember that we are using **Two Pane View** in Postman, so the bottom part of our main area is divided into request on the left side and response on the right.



On the left side, there's a menu with the tabs shown in the image above. Here, we will be working in the **Body** tab to be able to define our request document, so you should click on this.

Under the **Body** tab, you'll see a bunch of choices for the format of our data. You should choose **Raw**, which makes us able to define our request document in the JSON format so that it adheres to the JSON:API specification. Right under the choices, there's a dropdown list that currently is on **Text**. Fold this out and choose **JSON (application/json)**. You'll see a change in the header's tabs with a **1** indicating that there's one header defined in the request. We will return to that shortly.

## PLANNING



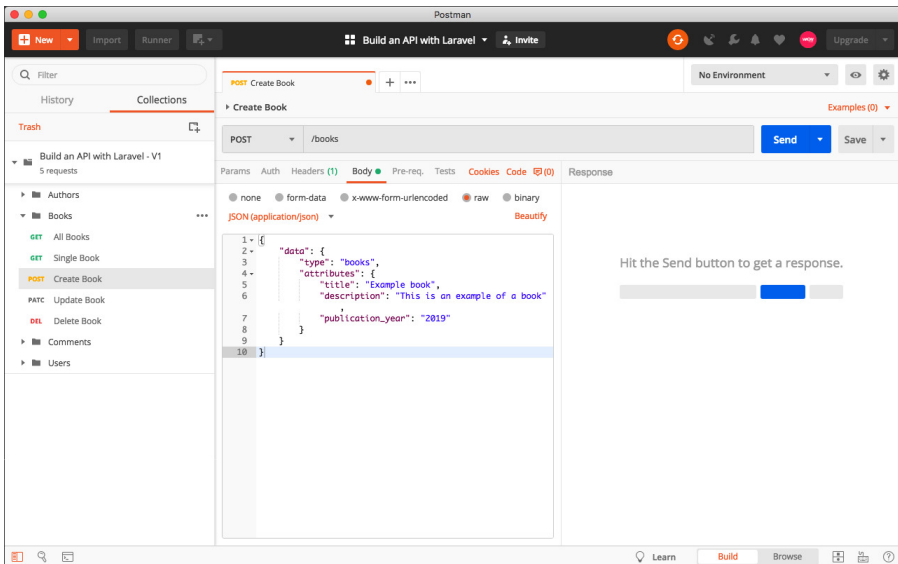
You should be at the same point as shown in the image above. Now it's time to define our request document.

First, we need to define the root of our request document and as the JSON:API specification states, we need to have a root object with a **data** member for our primary data of the document. Since we are documenting how to create a book, the primary data for our request document will be a resource object for a book. The important thing here is to define the **type** member as a string and then the **attributes** member as an object, which makes the entire request document look like this:

```
{
  "data": {
    "type": "books",
    "attributes": {
      "title": "Example book",
      "description": "This is an example of a book",
```

## BUILD AN API WITH LARAVEL

```
}
  "publication_year": "2019"
}
```



In Postman it should look like the image above.

We're almost done defining the **Create Book** request. We just have to go back to that **Headers** tab and take a look at the value of the **Content-Type** header defined by Postman.

The value here is **application/json** but since we want to adhere to the JSON:API specification, we need to use the right **Content-Type** defined in the specification. This needs to be changed to **application/vnd.api+json**.

Before we move on, we might as well set the **Accept** header as well. Like the **Content-Type**, this should be **application/vnd.api+json** to tell the server that



we expect to receive the data in the same format.

Click the “**Save**” button next to the “**Send**” button again to save your changes.

Just a little caveat. If you go back to the **Body** tab, the content under **Raw** has changed back to the **text**. This is because Postman does not recognize the JSON:API specifications **Content-Type** and does not know that it is indeed JSON. We hope they will fix this in a future update, but for now we will have to live with it. If you need to make alterations, you can change it from **text** to **application/json** again, but be aware that the **Content-Type** in the header tab will change back to **application/json** as well, which means that you need to change this to **application/vnd.api/json** when you’re done.

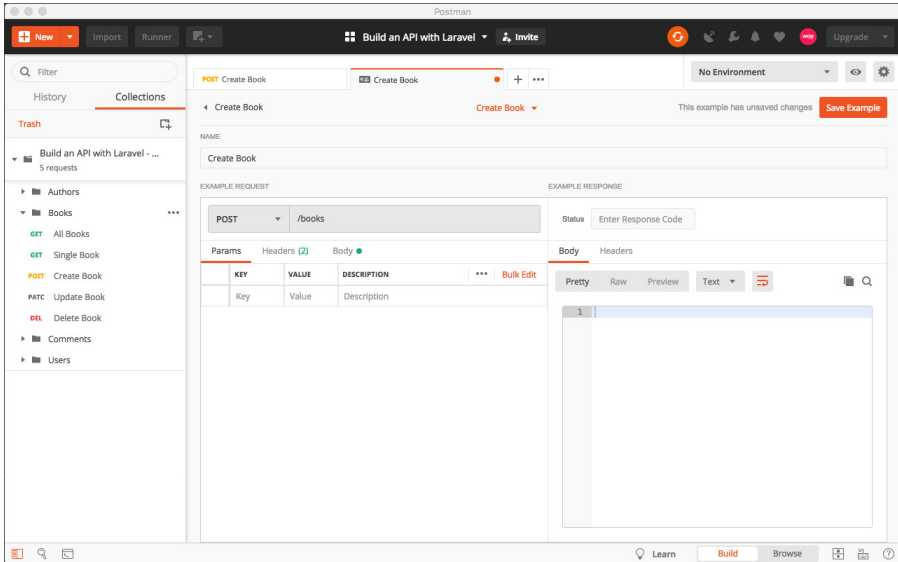
If we look at our documentation now by clicking **Browse** in the status bar at the bottom, and clicking **Build an API with Laravel** afterward, we can see that our documentation now contains the attributes for our resource together with the correct request document. It even gives an example of how to implement this request. But wouldn’t it be great if we could get an example of how the response document would look as well? Then everybody would know which data are being returned. Postman has us covered here with the ability to make request/response examples. Let’s take a look at that next.

### *Request/Response Examples*

As we mentioned, the way Postman handles documentation of responses is through examples. You find the ability to do this, right above the “**Save**” button in the top right corner.

Click on “**Examples**” and afterward click on the “**Add Example**” button.

## BUILD AN API WITH LARAVEL



The interface changes a bit as you can see in the image above.

The ability to make a request is now substituted with a name for the example, but we still have our request to the left and response to the right.

If you take a look at the request body, it is the same as we had before, but on the response side, we have some work to do.

First, we need to give a status code. Since it's a request that creates a book, we need the appropriate status code. If you remember from the chapter about the JSON:API specification, when a resource is created, we respond with a **201 Created** status code.

Next up is the body. As stated in the JSON:API specification, we need to return the newly created resource as the primary data of our response document.

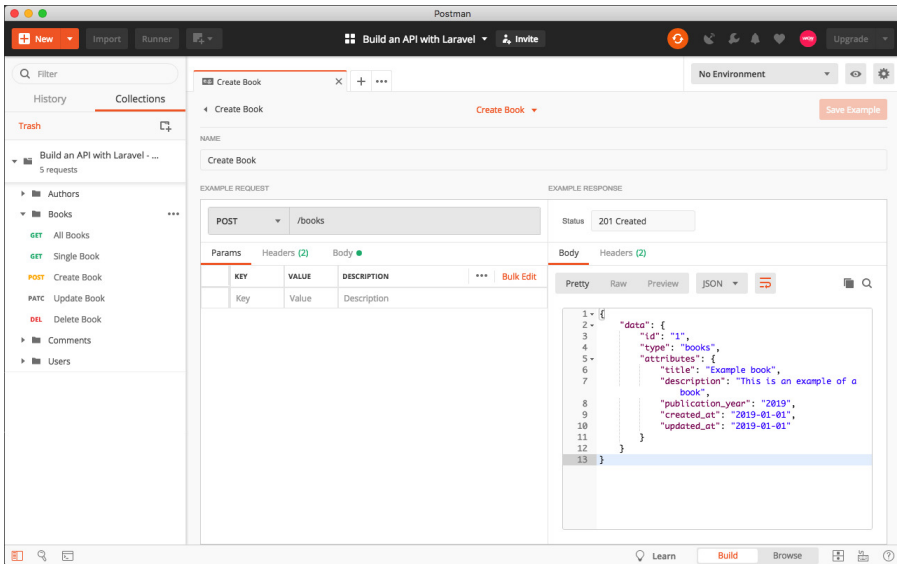
Easy enough, we already have that in the requested document, so let's copy it over. We need to do some editing though. Unless you expressly tell it not

to, Laravel includes a **created\_at** and an **updated\_at** attribute to its models. We love that Laravel does that: it has helped us countless times to have these timestamps, so let's include these as well. Since the resource is created now, we will have to give it an **ID** as well. This should result in a response document that looks like this:

```
{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Example book",
      "description": "This is an example of a book",
      "publication_year": "2019",
      "created_at": "2019-01-01",
      "updated_at": "2019-01-01"
    }
  }
}
```

In Postman it should look like the image below.

## BUILD AN API WITH LARAVEL



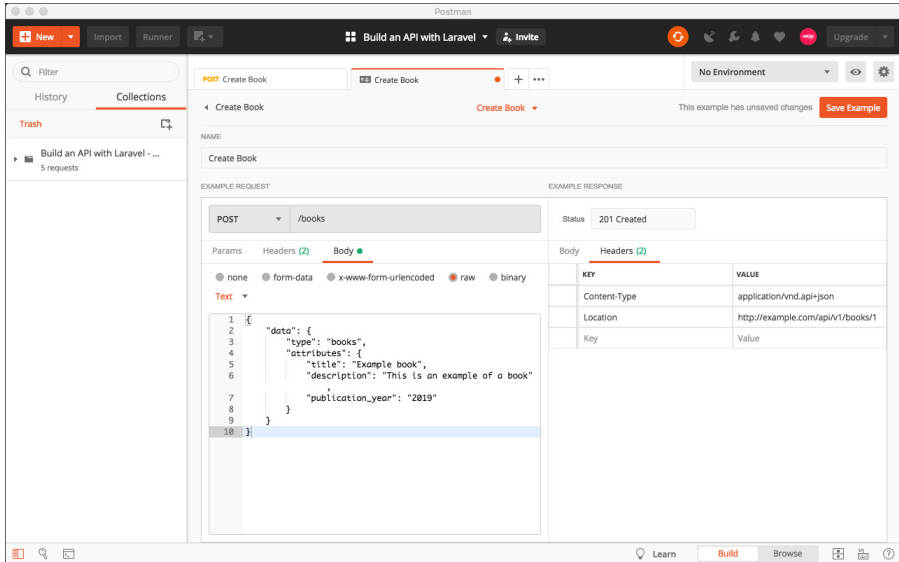
Now, let's take a look at the headers. If you click on the “**Header**”, you should see an empty list. Here, you should provide the **Content-Type** as well as the **Location** for the newly created resource.

By now you must know what the **Content-Type** should be, but the location is a bit more tricky. Since we don't have a domain yet, let's use the **example.com** domain to emphasize that this is just an example. The value of the **Location** header would then be:

```
GET: http://example.com/api/v1/books/1
```

We use the **/api** to signal that it's an API you're making requests to and the **/v1** is our way of telling the version of the API.

## PLANNING



If you have done it correctly, you should have the same as in the image above. Remember to hit the **“Save Example”** button.

If you go through **“Browse”** again to see the result in the documentation, you can see that the response has also been added.

Great. That was an example for the **Create Book** request, and now let’s make examples for the rest of them.

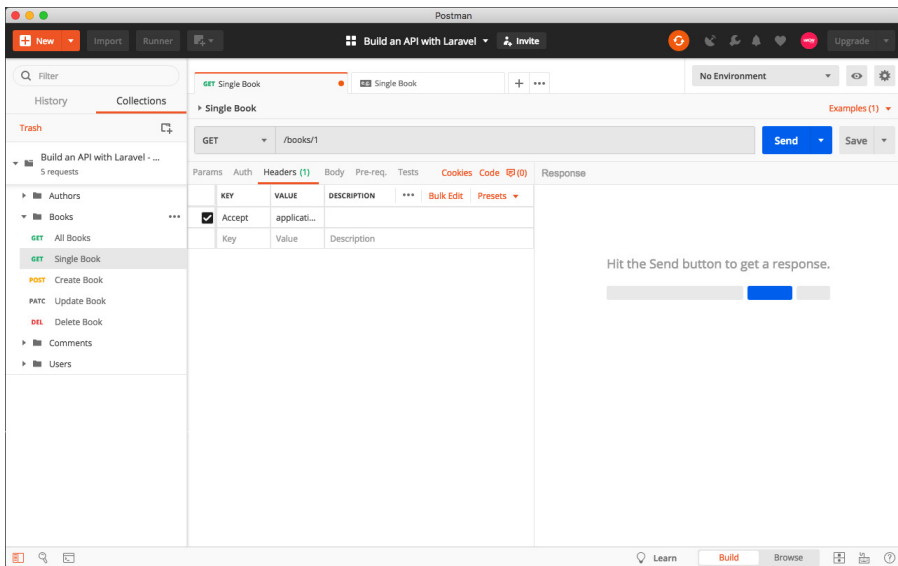
### *Examples for all endpoints*

Back in **Build** mode, let’s begin with the **Single Book** request since we actually have everything needed for this endpoint.

## Single Book

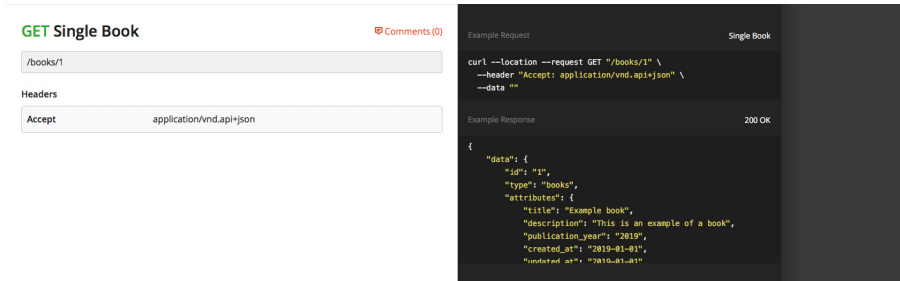
Before clicking away from the **Create Book** example, copy the contents of the body of the response and then go to the **Single Book** request.

Before making the example, there is one thing we are missing. Can you guess what that is? We need to define one header, namely the **Accept** header. Remember that the JSON:API specification states that we should include the **Accept** header with a value of **application/vnd.api+json** to tell the server that we expect the response to be in this format. Let's add that to the request.



If done right, you should have what is shown in the image above. Remember to save the request, and now let's make a new example for this request, using the dropdown above the **"Save"** button. We don't have to do anything about the request — we can leave that as it is. We actually only have to fill out the status code, body, and header of the response. Set the status to **200 OK**, since we are reading data this time, paste in the contents in the body, and set a header of

**Content-Type** with the value **application/vnd.api+json**.



If done correctly, when looking at the documentation, you should see that the endpoint is documented like in the image above.

Let's take a look at **All Books** next, since we almost have everything for this.

## *All Books*

Here, you should use the same procedure as with **Single Book**, adding the **Accept** header to the request and afterward making an example. In the example, the request can be left alone and in the response, the exact same status and header should be used.

Now, you are welcome to come up with the contents of the body yourself, as long as you remember that we are dealing with a collection of books as the primary data this time.

If you need some inspiration, here's what we have inserted into our example:

```
{
  "data": [
    {
      "id": "1",
```

## BUILD AN API WITH LARAVEL

```
        "type": "books",
        "attributes": {
            "title": "Example book",
            "description": "This is an example of a book",
            "publication_year": "2019",
            "created_at": "2019-01-01",
            "updated_at": "2019-01-01"
        }
    },
    {
        "id": "2",
        "type": "books",
        "attributes": {
            "title": "Example book two",
            "description": "This is yet another example of a
                book",
            "publication_year": "2019",
            "created_at": "2019-01-01",
            "updated_at": "2019-01-01"
        }
    }
]
}
```

### GET All Books

🔒 Comments (0)

/books

#### Headers

Accept application/vnd.api+json

Example Request

All Books

```
curl --location --request GET "/books" \
--header "Accept: application/vnd.api+json" \
--data ""
```

Example Response

200 OK

```
{
  "data": [
    {
      "id": "1",
      "type": "books",
      "attributes": {
        "title": "Example book",
        "description": "This is an example of a book"
      },
      "publication_year": "2019"
    }
  ]
}
```

Now, take a look at your documentation using “**Browse**”. It should look like the image above.



Ok, let's tackle the **Update Book** request next.

## *Update Book*

We will be taking the same approach as when creating a book. To begin, we will build up the body of the request, then set the correct headers, and save the request. Thereafter, we will be making the example and building up the response.

So let's start with the request body. Here, we only need to include the attributes we want to update, so for instance, if we only want to update the title, that's all we need. Let's do that, and the request document will look like this:

```
{
  "data": {
    "id": 1,
    "type": "books",
    "attributes": {
      "title": "Update book title"
    }
  }
}
```

Remember that we need to include the **id** and **type** in the update request and then the **attribute** object containing the attributes we want to update.

Next, let's tackle the header. You have most likely set the content to be **JSON (application/json)** so that it is easier to work with the content, but remember that Postman then sets the **Content-Type** header as well, so this will have to be corrected. We also need to add the **Accept** header as well. By now, you should be able to remember what the values must be, so let's move on to the example, but before you click on example, be sure to copy the contents of the request body.

Set the status to **200 OK** and paste the contents of the request into the body. We need to include the last attributes, which you can find in the **Single Book** example if you can't remember them. The reason why we are returning the resource object is that we are planning ahead. The JSON:API specification states that you need to return the resource object if it has an **updated\_at** attribute that will be updated as well. Laravel does that, so we need to return the resource object. To show that the update of the **updated\_at** attribute is happening as well, set the date to be the 2nd of January 2019.

Set the **Content-Type** header as well and we are done with this request.

#### PATCH Update Book

Comments (0)

/books/1

##### Headers

|              |                          |
|--------------|--------------------------|
| Content-Type | application/vnd.api+json |
| Accept       | application/vnd.api+json |

Body raw (application/vnd.api+json)

```
{
  "data": {
    "id": 1,
    "type": "books",
    "attributes": {
      "title": "Update book title"
    }
  }
}
```

##### Example Request

##### Update Book

```
curl --location --request PATCH "/books/1" \
--header "Content-Type: application/vnd.api+json" \
--header "Accept: application/vnd.api+json" \
--data '{
  "data": {
    "id": 1,
    "type": "books",
    "attributes": {
      "title": "Update book title"
    }
  }
}'
```

##### Example Response

```
{
  "data": {
    "id": 1,
    "type": "books",
    "attributes": {
      "title": "Update book title",
      "description": "This is an example of a book",
      "publication_year": "2019",
      "created_at": "2019-01-01",
    }
  }
}
```

If you have done it correctly, it will be like in the image above when looking at your documentation.

We are almost done. We only need the last **Delete Book** request, so let's go through that now.

## Delete Book

This one is probably the easiest one. We don't need a request document, and we don't need to do a detailed response example since you don't have to respond with any response documents unless you have some **meta-data** you want to

include. In our planning, we don't have that, so the only thing we need to do is to add the **Accept** header to the request and make an example where we include the status code **204 No Content**. To make it show up in the documentation, go into the response body and make a new line.

That is all the request and examples for the **Books** resource. Now we just have to repeat the same tasks for the rest of the resources.

### *The remaining resources*

Now it's time for you to be a little on your own in Postman. For the next resources, you have to create all the requests and define the attributes, examples, and so forth. We are not going to force attributes on you — we want you to think about these yourself. If you still feel a little unsure or just need inspiration, here is a list of attributes we have defined for each resource in our planning phase:

- **Authors**
  - - Name
  - - Updated At (Comes from Laravel)
  - - Created At (Comes from Laravel)
- **Comments**
  - - Message
  - - Updated At (Comes from Laravel)
  - - Created At (Comes from Laravel)
- **Users**
  - - Username
  - - Name
  - - Email
  - - Updated At (Comes from Laravel)
  - - Created At (Comes from Laravel)

We have kept it pretty simple here, and there are probably many more

attributes that could be a part of each resource. If you have many more than us, that's fine. Just don't include so many that you introduce unnecessary complexity into your application.

If you still find yourself a bit lost, we have an export of our finished collection/documentation in our Github repository — just ignore the relationships and query parameters for now, as we will get into these next.

## *Documenting Query Parameters*

In the part about sorting resources and pagination in the chapter about the JSON:API specification, we learned that this is done through query parameters. It gives us the ability to sort books by their publication year by writing:

```
GET: /books?sort=publication_year
```

We need to document this as well. Since we are in the vein of books why don't we, once again, take a look at our **Books** resource in Postman.

## *Sorting*

Here, the sorting only really makes sense in the **All Books** request, since we get a list of books that may need to be sorted.

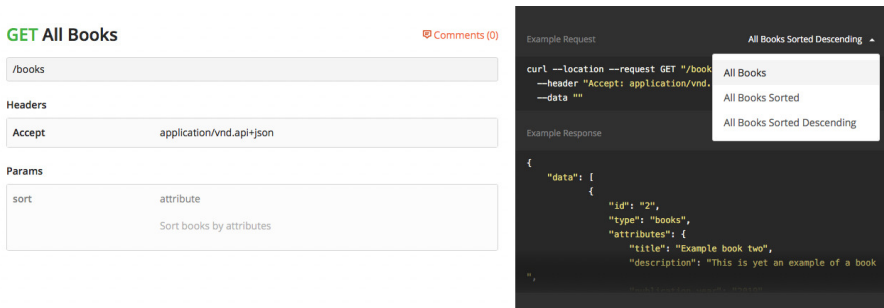
Select the **All Books** request in the side-panel. In the main area, in the request part to the left, we see that we are already at the “**Params**” tab. The way we define query parameters is the same as headers. We have a table where we define the key and give a value. If we want, we can give a description as well. Set the key to **sort**, the value to **attribute**, and in the description you write: “**Sort books by attributes**”. Right next to the key, there's a checkmark. Remove this, since we don't want the query parameter to be a part of the endpoint in

the documentation, but only inform that it is possible. Save the request and take a look at the documentation.

Now there's a **Params** section, informing about the **Sort** query parameter. An example is often the most efficient means of communicating information, so let's make a new example that shows how the sort query parameters work.

In the example dropdown, click "**Add Example**" and name it "**All Books Sorted**". In the **Params** tab, check the checkbox at the **sort** key and change the value to **publication\_year**. Copy the contents in body from the **All Books** example and change the **publication\_year** value to **2018** for the first book. Then add the correct status code and header and save.

Make another example, this time for sorting in descending order and name it "**All Books Sorted Descending**". Perform the exact same steps as before, but this time the value of the query parameter needs to be **-publication\_year**. After you have copied the contents of the response body and changed the **publication\_year**, the two books need to switch places, so the book with **id** of value **2** is first in the array.



The screenshot shows a REST client interface with the following sections:

- GET All Books** (with a red comment icon and 0 comments)
- URL:** /books
- Headers:**
  - Accept: application/vnd.api+json
- Params:**
  - sort: attribute (with a sub-label "Sort books by attributes")
- Example Request:**

```
curl --location --request GET "/books" \
  --header "Accept: application/vnd." \
  --data ""
```
- Example Response:**

```
{
  "data": [
    {
      "id": "2",
      "type": "books",
      "attributes": {
        "title": "Example book two",
        "description": "This is yet an example of a book"
      }
    },
    {
      "id": "1",
      "type": "books",
      "attributes": {
        "title": "Example book one",
        "description": "This is yet an example of a book"
      }
    }
  ]
}
```

If you take a look at the documentation now, we have some examples that show both how to use the sort query parameter but also how it works with descending order. There is one last example we need, namely an example showing a comma separated list for sorting by multiple attributes.

Add a new example and name it “**All Books Sorted by Multiple Attributes**” and repeat the steps from the previous example. This time for the query parameter write: **-publication\_year,title**. And that’s it. Look at the documentation to see if everything is how it should be.

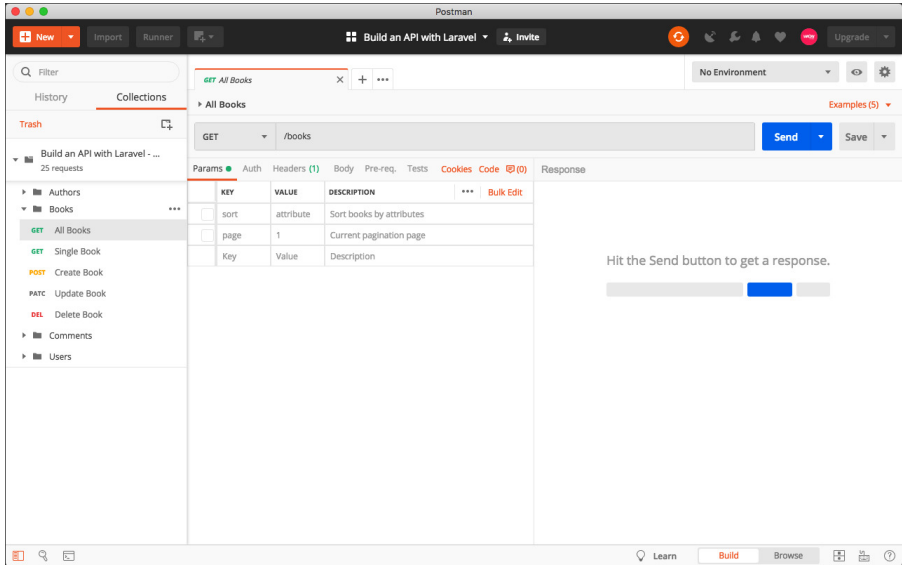
## *Pagination*

Documenting pagination for our books takes the same approach as the **sort** query parameter. It only really makes sense in the **All Books** request and luckily we only need one example.

Before we can make that example, however, we first need to add pagination query parameter to our **All Books** request part so that it will show up in our documentation.

Under the **sort** parameter, add a new parameter with the key **page**, value 1 and description: “**Current pagination page**”. Uncheck the checkbox next to **page** and save the request.

## PLANNING



You should see something like in the image above.

Let's make an example for it. Create a new example and name it **"All Books Paginated"**. In this new example, check the checkbox next to **page**, change the value from **1** to **2**, and add the following into the response body:

```
{
  "data": [
    {
      "id": "3",
      "type": "books",
      "attributes": {
        "title": "Example book 3",
        "description": "This is an example of a book",
        "publication_year": "2019"
      },
      "relationships": {}
    }
  ],
}
```

```

{
  "id": "4",
  "type": "books",
  "attributes": {
    "title": "Example book 4",
    "description": "This is an example of a book",
    "publication_year": "2019"
  },
  "relationships": {}
}
],
"links": {
  "first": "/books?page=1",
  "last": "/books?page=5",
  "prev": "/books?page=1",
  "next": "/books?page=3"
}
}

```

For brevity we have omitted the relationship, since they don't have anything to do with what we are trying to convey here, namely the new **links** member with pagination links inside the object. Don't forget the header and **Content-Type** and save the example.

## *Documenting Relationships*

By now, we have been working with our resources and which attributes these should have, as well as sorting and pagination by query parameters. We have even documented these findings. Now, it's time to do the same with relationships. Here, we will edit the existing resources and add the various resource identifier objects and also, once again, have a look at the **included** top-level member of our response documents.

We always use the **included** member by default - remember that this is optional - and have often not supported the feature of the **include** query parameter. But we want to teach you all we know, so of course we will support it in this book.



## *Adding Relationships*

When we identified our relationships, we made a list of these where we also identified the relationship links based on the conventions from the JSON:API specification.

These are:

- **Books and Authors** – A many-to-many relationship with endpoints:
  - – **Self**

```
GET: /books/1/relationships/authors
```

- – **Related**

```
GET: /books/1/authors
```

And the inverse relationship from authors

- – **Self**

```
GET: /authors/1/relationships/books
```

- – **Related**

```
GET: /authors/1/books
```

- **Books and Comments** – A one-to-many relationship with endpoints:
  - – **Self**

```
GET: /books/1/relationships/comments
```

- - **Related**

```
GET: /books/1/comments
```

And the inverse relationship from comments

- - **Self**

```
GET: /comments/1/relationships/books
```

- - **Related**

```
GET: /comments/1/books
```

- **Comments and Users** - A one-to-one relationship with endpoints:

- - **Self**

```
GET: /comments/1/relationships/books
```

- - **Related**

```
GET: /comments/1/books
```

And the inverse relationship from users

- - **Self**

```
GET: /users/1/relationships/comments
```

- - **Related**

```
GET: /users/1/comments
```

Let's use the **Books** resource again. Here, we need to update our requests in Postman, as well as add new examples, so our documentation reflects what you can include in the requests.

Why don't we start with the **Single Book** request first. Returning to Postman, open up the example which right now should have the name **Single Book** as well. We will change that in a bit, but for now you should add the **relationships** member to the response body like this:

```
{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Example book",
      "description": "This is an example of a book",
      "publication_year": "2019",
      "created_at": "2019-01-01",
      "updated_at": "2019-01-01"
    },
    "relationships": {
      "authors": {
        "links": {
          "self": "/books/1/relationships/authors",
```

```
    "related": "/books/1/authors"
  },
  "data": [
    {
      "id": "1",
      "type": "authors"
    },
    {
      "id": "2",
      "type": "authors"
    }
  ]
},
"comments": {
  "links": {
    "self": "/books/1/relationships/comments",
    "related": "/books/1/comments"
  },
  "data": [
    {
      "id": "1",
      "type": "comments"
    },
    {
      "id": "2",
      "type": "comments"
    }
  ]
}
}
```

Just to give an example for our documentation, we have included two related resources for both **Authors** and **Comments**. Save the example and let's take a look at the documentation for our **Books** resource and the **Single Book** request.

The example is now too long to be shown in the documentation, but if you click to expand it, it should reflect the response document given in the example

above.

Now that we have the **Single Book**, we also have the recipe for the changes to **All Books**. We'll let you do this on your own, but remember to change the relationships, especially for the comments, so two books don't share the same comments. If you get stuck, remember that we do have a finished version of the collection/documentation in our Github repository.

For the **Create Book** and **Update Book** requests, we need to make examples that show how to make requests that not only creates or updates a resource, but also creates a relationship to another resource. Let's start with **Create Book**.

We will only create a book with a relationship to authors. It doesn't make sense to create a book with a relationship to a comment, since it requires that the comment on the book already exists. Create a new example and name it Create Book with Authors.

Now, to create relationships along with a resource, we need to include the **relationship** member in the request document. Just like in the response document, this needs to be on level with the **attributes** member. We define the relationship by adding members for the various resources we want to create a relation to, in this case **authors**. We add a **data** member or resource linkage inside the **authors** object where we give resource identifier objects to the specific resources we want a relationship to.

It would look like this:

```
{
  "data": {
    "type": "books",
    "attributes": {
```

```

    "title": "Example book",
    "description": "This is an example of a book",
    "publication_year": "2019"
  },
  "relationship": {
    "authors": {
      "data": [
        {
          "id": "1",
          "type": "authors"
        },
        {
          "id": "2",
          "type": "authors"
        }
      ]
    }
  }
}

```

If you have the same as above, you can continue on to the response body. Take a copy of the request body and make the right adjustments, such as adding an **id** member for the book and adding the **links** member to each relationship, together with an object containing the **self** and **related** links.

That would look like this:

```

{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Example book",
      "description": "This is an example of a book",

```

```

        "publication_year": "2019",
        "created_at": "2019-01-01",
        "updated_at": "2019-01-01"
    },
    "relationships": {
        "authors": {
            "links": {
                "self": "/books/1/relationships/authors",
                "related": "/books/1/authors"
            },
            "data": [
                {
                    "id": "1",
                    "type": "authors"
                },
                {
                    "id": "2",
                    "type": "authors"
                }
            ]
        }
    }
}

```

Remember to add the correct status code and headers.

Next up, is the **Update Book** request. We can follow the same steps as above, but you don't need to include all of the attributes, only the ones that need to be updated.

Create a new example and name it **Update Book with Authors**. Copy the request body from the **Update Book** example and add the relationship to **authors** with an array of resource identifier objects like this:

```
{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Updated title",
    },
    "relationship": {
      "authors": {
        "data": [
          {
            "id": "4",
            "type": "authors"
          }
        ]
      }
    }
  }
}
```

Here, we will remove all existing authors from the book and add the author with the **id** of **4**. Just like the **Create Book with Authors** example, you can copy over the request body to the response body and fill out the relationship links.

It would look like this:

```
{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Updated title",
      "description": "This is an example of a book",
      "publication_year": "2019",
      "created_at": "2019-01-01",
      "updated_at": "2019-01-02"
    }
  }
}
```



```

    },
    "relationships": {
      "authors": {
        "links": {
          "self": "/books/1/relationships/authors",
          "related": "/books/1/authors"
        },
        "data": [
          {
            "id": "4",
            "type": "authors"
          }
        ]
      }
    }
  }
}

```

Again, remember the correct status code and header.

Now that we have updated all of our resources to reflect the relationships, it is time to look at the **included** member and how it works with the **include** query parameter.

### *The Include Query Parameter*

The **include** query parameter is actually an optional feature, according to the JSON:API specification. As we mentioned earlier, we usually don't support this feature, but make use of the **included** member of our response document where all the relationship are queried and included in the response especially since it's so easy with Laravel. But, of course, this puts a toll on performance and you could speed up requests by omitting this and letting the consumer decide for themselves. Also, we want to teach you everything we know — that's why we have chosen to go through this as well.

In terms of documentation, this feature is not that complex since it's a query parameter like the **sort** query parameter. But in contrast to the **sort** query parameter, the **include** parameter can be used when fetching all resources and a single resource. So let's document that in Postman now and again, let's use the **Books** resource and start in the **Single Book** request.

Just like when we created the **sort** query parameter, we need to make a parameter in the request, uncheck it, and afterwards copy it over to new examples where it needs to be checked. Add a new parameter with the key **include**, value **resource** and description: "**Related resources to be included in the request**" and save the request.

Create a new example and name it "**Single Book including Comments**". Check the checkbox at the parameter and change the value to **comments**.

Copy the response body from the first Single Book example and add the **included** member containing the resource objects of all the comments, which should look like this:

```
{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Example book",
      "description": "This is an example of a book",
      "publication_year": "2019",
      "created_at": "2019-01-01",
      "updated_at": "2019-01-01"
    },
    "relationships": {
      "authors": {
        "links": {
          "self": "/books/1/relationships/authors",
```

```

    "related": "/books/1/authors"
  },
  "data": [
    {
      "id": "1",
      "type": "authors"
    },
    {
      "id": "2",
      "type": "authors"
    }
  ]
},
"comments": {
  "links": {
    "self": "/books/1/relationships/comments",
    "related": "/books/1/comments"
  },
  "data": [
    {
      "id": "1",
      "type": "comments"
    },
    {
      "id": "2",
      "type": "comments"
    }
  ]
}
},
"included": [
  {
    "id": "1",
    "type": "comments",
    "attributes": {
      "message": "Hello world"
    },
    "relationships": {
      "users": {

```

```

        "data": {
            "id": "1",
            "type": "users"
        }
    },
    "links": {
        "self": "/comments/1"
    }
},
{
    "id": "2",
    "type": "comments",
    "attributes": {
        "message": "Foo bar"
    },
    "relationships": {
        "users": {
            "data": {
                "id": "2",
                "type": "users"
            }
        }
    },
    "links": {
        "self": "/comments/2"
    }
}
]
}

```

Remember to add the correct header and **Content-Type**.

Let's also make an example for related resources, which in this case is the user that creates each of the comments.

Create a new example and name it “**Single Book including Comments and commenting User**”. Check the parameter and change the value to **com-**

**ments.users.** Copy the contents of the response body from the previous example and add the two users' resource objects to the **included** array of resource objects, like this:

```
{
  "data": {
    "id": "1",
    "type": "books",
    "attributes": {
      "title": "Example book",
      "description": "This is an example of a book",
      "publication_year": "2019",
      "created_at": "2019-01-01",
      "updated_at": "2019-01-01"
    },
    "relationships": {
      "authors": {
        "links": {
          "self": "/books/1/relationships/authors",
          "related": "/books/1/authors"
        },
        "data": [
          {
            "id": "1",
            "type": "authors"
          },
          {
            "id": "2",
            "type": "authors"
          }
        ]
      },
      "comments": {
        "links": {
          "self": "/books/1/relationships/comments",
          "related": "/books/1/comments"
        },
        "data": [
```

```

        {
            "id": "1",
            "type": "comments"
        },
        {
            "id": "2",
            "type": "comments"
        }
    ]
}
},
"included": [
    {
        "id": "1",
        "type": "comments",
        "attributes": {
            "message": "Hello world"
        },
        "relationships": {
            "users": {
                "data": {
                    "id": "1",
                    "type": "users"
                }
            }
        },
        "links": {
            "self": "/comments/1"
        }
    },
    {
        "id": "2",
        "type": "comments",
        "attributes": {
            "message": "Foo bar"
        },
        "relationships": {
            "users": {
                "data": {

```

```

        "id": "2",
        "type": "users"
      }
    },
    "links": {
      "self": "/comments/2"
    }
  },
  {
    "id": "1",
    "type": "users",
    "attributes": {
      "username": "johndoe",
      "name": "John Doe",
      "email": "john@example.com",
      "created_at": "2019-01-01",
      "updated_at": "2019-01-01"
    },
    "relationships": {
      "comments": {
        "data": {
          "id": "1",
          "type": "comments"
        }
      }
    },
    "links": {
      "self": "/users/1"
    }
  },
  {
    "id": "2",
    "type": "users",
    "attributes": {
      "username": "janedoe",
      "name": "Jane Doe",
      "email": "jane@example.com",
      "created_at": "2019-01-01",
      "updated_at": "2019-01-01"
    }
  }
}

```

```

    },
    "relationships": {
      "comments": {
        "data": {
          "id": "2",
          "type": "comments"
        }
      }
    },
    "links": {
      "self": "/users/2"
    }
  }
]
}

```

Phew, that was quite an amount of data for a book like this! Don't forget the headers and **Content-Type**.

We need one more example though: one showing a comma separated list of related resources.

Make yet another example and name it “**Single Book including Authors and Comments**”. Check the checkmark at the parameter and change the value to **authors,comments**.

Copy the contents of the resource body from the last example and change out the users to authors. This time you should try to edit the data yourself — if you have forgotten what the author's resource object looks like, you can copy it from your **Author** resources **Single Author** example. Don't forget the header and **Content-Type**.



## Summary

In this chapter we have been through a lot. First, we started the planning of our project. When developing an API, we no longer have to think about UI/UX and can focus more on our data.

We began our planning phase by identifying our resources and the relationships between these resources. We saw how we can conveniently think of our resources like a mapping of our models and database table.

We learned about Postman and how we can use it to document our API and how it can help us think about our resource attributes and relationships. By documenting our API early, we forced ourselves to work with the JSON:API specification and our applications data.

Sometimes, it's tedious work, plotting all of these examples into Postman, but if you have been taking some notes, you already have a good idea of where the more complex parts of the implementation lie and where the easiest parts are.

We learned a bit more about the **included** top-level member of our response documents, and how we can leverage the **include** query parameter to tell our API which related resources we want included in the response.

We hope that you finished your documentation and explored Postman a little more. A big win, which we also touched upon in this chapter, is that when the documentation is done, a mock server can be set up for the frontend people or other consumers of our API, which can relieve the pressure on our shoulders as backend developers.

Next up, we will begin building our API. We won't be leaving Postman yet, but we will finally start to do some coding, and isn't that why we are all here anyway? Great! Let's get on with it!

## BUILD AN API WITH LARAVEL

\* \* \*

# 4

## Build your API

It's finally time to get into some coding and, after all that theory and planning, we bet you are ready for it too.

In this chapter, we will first take a look at the features of Laravel Passport, and how you can leverage these whether you are consuming your own API or making a public API.

We will install Laravel Passport and use this to secure our API and go through the different ways of authentication.

We will then take a look at where to start and how to get the ball rolling while we build the first resource of our API and implement the first features of the JSON:API specification.

Afterward, we will test the resource in Postman to see if it works as intended.

We will be using Laravel Valet for our local setup. If you are on a Mac, we highly recommend that you use Laravel Valet for your local configuration, since it is so easy to set up. It proxies all requests to the **\*.test** top-level domain to point to sites in your development folders, which makes it very convenient to

work with. It also features ngrok tunnels that makes it possible to share your site from your local machine to the internet, without you having to deploy to a server, so you can show off your work to clients or co-workers more easily. The best thing is that Valet automatically detects what type of PHP application you are working on, and finds a suitable driver that knows how to serve your request. It's not only for Laravel but PHP in general.

If you are on a different platform than Mac, we recommend Laravel Homestead since it includes everything you need to develop Laravel applications in a Vagrant box. Vagrant boxes are virtual machines that, in this case, run a Linux server — just like the servers you will deploy your Laravel applications on when releasing your applications. We've used this before Laravel Valet was released. It's a bit heavier and there's a bit more to set up than with Valet, but the nice thing is that it maps your folders on your local machine into your Vagrant box, so if something goes wrong in your Vagrant box, you can just remove it and spin up a new one. Here, it's also possible to map your projects to domains on your local machine.

There's also the Laradock project, which instead of spinning up an entire virtual machine, uses Docker to spin up containers for the various servers you need. It's a bit less cumbersome than Vagrant but has all the same great features. It's not a Laravel first-party solution, but we have heard a lot of good things about it and have been using Docker for Laravel projects as well, which worked out very well.

Before we move on, we will quickly mention the difference between the local setup solutions and domains. Since we use Laravel Valet, our domain will look like this:

```
GET: annas-bookstore.test/api/v1
```

If you happen to be on another setup than Laravel Valet, you might be working

on localhost, which will then make your domain look like this:

```
GET: localhost/annas-bookstore/api/v1
```

That is completely fine. We just wanted to point that out to avoid any confusion.

If you are on a Docker setup, you can use **dnsmasq** to map to a local top level domain like **.test** like Laravel Valet provides, in fact Laravel Valet uses **dnsmasq** to do just this and it's not very difficult to set up.

Now that we have the local setup out of the way, let's get into the authentication.

## Authentication with Laravel Passport

Laravel Passport is a first party package for the Laravel framework that specifically handles API authentication. Laravel Passport is built on top of OAuth2, which is an industry standard protocol for authorization. By leveraging the OAuth2 standard, using Laravel Passport makes it possible for you to provide a standardized way of authenticating your API built with Laravel.

If you have tried to authenticate services, like letting Laravel Forge access your Digital Ocean account for server provisioning, the authentication is done through OAuth. If you have let any application access your Dropbox, Spotify or Google account, this is also done through OAuth.

OAuth is everywhere, and it's a good reason for you to have it in your API as well — especially if you are developing an API open to the world.

If you want a private API that a single page application deployed together

with your Laravel application should consume, Laravel Passport has got you covered here as well. Let's take a more in-depth look at Laravel Passport. First, we need to install it.

## *Installation*

Before we install Laravel Passport, we need a fresh Laravel application. Go into your favorite terminal application and use the Laravel Installer to make a new application named **annas-bookstore** like this:

```
laravel new annas-bookstore
```

At the time of writing this book, Laravel 5.8 has just been released and this will be the version of Laravel we will be building the API up against.

After the installation, Laravel makes sure to create a database for the application. You can name it whatever you want, as long as you make sure to put the right pieces of information inside your **.env** file. We will be naming ours **annas\_bookstore**.

Now we are ready to install Laravel Passport through Composer like this:

```
composer require laravel/passport
```

Next, we have to install Laravel Passport into our Laravel application.

Laravel Passport needs some database tables to store clients and access tokens. Fortunately, these are provided in the migrations that are included in the composer package. So the only thing we need to do is to run our database

migrations like this:

```
php artisan migrate
```

You should be seeing about 5 tables being created and that's it.

Now we can run the installation of Laravel Passport itself like this:

```
php artisan passport:install
```

Laravel Passport has been installed. Next, we want to add a **Laravel\Passport\HasApiTokens** trait to our User model, so that Laravel Passport can work with our User models and we get the ability to inspect which token and scope belongs to which user:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use Notifiable, HasApiTokens;

    protected $fillable = [
        'name', 'email', 'password',
```

```
];

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

protected $casts = [
    'email_verified_at' => 'datetime',
];
}
```

To be able to issue or revoke access tokens, clients, and personal access tokens, we need to register the routes for this. We do this by calling the **Passport::routes** method in the **boot** method of our **AuthServiceProvider** like this:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as
    ServiceProvider;
use Laravel\Passport\Passport;

class AuthServiceProvider extends ServiceProvider
{
    protected $policies = [
        // 'App\Model' => 'App\Policies\ModelPolicy',
    ];

    public function boot()
    {
        $this->registerPolicies();
        Passport::routes();
    }
}
```



```
}
```

Then we need to tell Laravel that we want to use Laravel Passport as a driver for API authentication. We do this in the **config/auth.php** file like this:

```
<?php

return [

    'defaults' => [
        'guard' => 'web',
        'passwords' => 'users',
    ],

    'guards' => [
        'web' => [
            'driver' => 'session',
            'provider' => 'users',
        ],

        'api' => [
            'driver' => 'passport',
            'provider' => 'users',
            'hash' => false,
        ],
    ],
],
```

Everything is set up now, and we are ready to use Laravel Passport as our API authentication. But before we can dive further into the authentication, we need a user to authenticate.

In a fresh Laravel application, the easiest way of getting a user in our database is by using the artisan **tinker** command. So let's get into the terminal and start tinkering.

In the root of the project, go into artisan tinker like this:

```
php artisan tinker
```

Here, we will create a test user for now, using the infamous John Doe like this:

```
User::create(['name'=>'John Doe', 'email'=>'john@example.com', 'password'=>bcrypt('secret')])
```

When the user has been created you will see something like this:

```
=> App\User {#2968
    name: "John Doe",
    email: "john@example.com",
    updated_at: "2019-03-07 10:49:36",
    created_at: "2019-03-07 10:49:36",
    id: 1,
}
```

Let's then take a look at authentication and how this is done in Laravel Passport.

## *Authentication*

When you install Laravel, it comes with some routes already defined, so you can make sure that it works. If you hit the URL of the application, you will be greeted by a page like this:

# Laravel

[DOCS](#)[LARACASTS](#)[NEWS](#)[BLOG](#)[NOVA](#)[FORGE](#)[GITHUB](#)

To find the route of this page, go into the **routes/web.php** file. Here you will find a single route that returns the **welcome** view.

If you take a look at the **routes/api.php** file, you can see that there's a route defined as well. You can also see that the route is using the `auth:api` middleware. This tells us that only authenticated users can make requests to this route since it's protected by our authentication middleware. Let's test this in Postman. Here, you should close the request you are working on to get a new untitled request.

Enter the request URL:

```
GET: /api/user
```

Remember to add your domain to the URL. In the case of our setup, the domain is **annas-bookstore.test** which will result in a URL like this:

```
http://annas-bookstore.test/api/user
```

In the request section, click the **Headers** tab, and in the **Key** field enter **Accept**. Next, in the **Value** field enter **application/json** and hit the send button.

In the response, you will get a **401 Unauthorized** status with a body containing the following JSON:

```
{
  "message": "Unauthenticated."
}
```

Unsurprisingly, we get this message. We haven't done any form of authentication yet, so this is merely telling us that the authentication works as it should when it comes to unauthorized requests.

## *Access tokens*

Since we are using Laravel Passport, which adheres to the standards of OAuth2, the only way we can make an authenticated request is by having a valid bearer token or access token.

An access token is a unique token that represents a user's permission for a client to access their data. When a client possesses an access token, it can use it to make subsequent authenticated requests on behalf of that user. A client in this context is another application, either an application running on a server or an application running in the browser or mobile phone.

When a client acquires an access token, it is called a **grant**, since the user is granting the client access to its data. There are many different types of grants, so let's look at them for a moment.

In the list below, you can see the most common grant types and the ones supported by Laravel

Passport:

- Authorization Code
- Implicit
- Password Credentials
- Client Credentials
- Personal Access Token

### *Authorization Code*

The Authorization Code grant type is likely the grant type most developers are familiar with. This is the grant type used by Facebook or Google when you sign into an application using your credentials from one of these services. It is also the way Laravel Forge authenticates with your Github or Bitbucket accounts.

The actors that are part of this grant type are:

- A user who can grant access to his/her resources
- A server issuing access tokens and hosting the protected resources
- - Just a quick disclaimer - the server isn't always the one issuing tokens. This can just as well be done by a dedicated authorization server or service.
- A client that needs to request the protected resources

To show how this grant type works, we will give a little practical example. However, before we can do so, we have a little setup to do.

We have our server that can issue tokens since we have our **annas-bookstore** project and have Laravel Passport installed. We just need one thing for this to work, namely some kind of authentication we can visit with our browser. Easy enough, we can do this in Laravel with a simple artisan command like this:

```
php artisan make:auth
```

That's the server part, but we also need a client application and like any good TV cooking show, we have prepared this ahead of time.

If you haven't cloned our repository yet, this is an excellent time to do so, as we will be using an application called Passport OAuth Client from our GitHub repository. So start by cloning our repository like this:

```
git clone https://github.com/WackyStudio/build-an-api-with-laravel.git
```

This application is just a regular Laravel application, so let's run a **composer install** to install the PHP dependencies and **npm install** to install the JavaScript dependencies followed by a **npm run dev**, so our assets are compiled. If you are on Mac or Linux, you can do it all in one go like this:

```
composer install && npm install && npm run dev
```

Then we need to configure the application, so make a copy of the **.env.example**, rename it to **.env** and open the file in your editor. First, set the **APP\_URL** environment variable to the URL of the application on your configuration. On our Laravel Valet configuration, we will make a Valet link to the application and call it **passport-oauth-client**, which will make our **APP\_URL** look like this:

```
http://passport-oauth-client.test
```

The environment variable will be used in the application; this is why it's

essential.

Just as important is the variable in the bottom of the `.env` file. The **BOOKSTORE\_URL** environment variable should point to the URL of the annas-bookstore application, which in our case is:

```
http://annas-bookstore.test
```

Now, for the **PASSPORT\_CLIENT\_ID** and **PASSPORT\_CLIENT\_SECRET** we need to explain a little.

For a user to grant a client access to their data on the server, the server first needs to know about the client. For the server to do so, we need to create a client on the server. With Laravel Passport this can be done in two, or technically, three ways. The first way is to create the client using an artisan command. The second way is to use a dedicated JSON API, that comes with Laravel Passport. Don't worry, this API is not open to the public. In fact, it can only be accessed by an authenticated user, meaning that it's only accessible by the users in your application. The "third" way is to use a set of Vue components that ships with Laravel Passport. These can be included on any page behind authentication and will give you everything you need in terms of creating clients in your application. Behind the scenes, the components use the Laravel Passport API to issue tokens, which is the reason why, technically, they are the third option. However, by using these, you can skip a lot of work.

We will be using these later on when we start building our API, but more on that then. For now, we can just use the artisan commands to create our client. This can be done like so:

```
php artisan passport:client
```

You will be asked which user **ID** the client should be assigned to.

Since we already have John Doe created in our system, go ahead and type **1** as the **ID**.

You will then be asked to name the client. You can call it whatever you want – we will be calling it **passport-client-test**.

The last question you will get is where to redirect the request after the authorization. This depends on your local setup, but since our **passport-oauth-client** has the following URL:

```
http://passport-oauth-client.test
```

Our callback URL will then become:

```
http://passport-oauth-client.test/callback
```

You will then see an output containing the **Client ID** and **Client Secret** like this:

```
Client ID: 1  
Client secret: GfVhVAgFyRW1DaP9nDYwYMT9wr1SZz4zYMa5rvBg
```

Copy the **ID** into the **PASSPORT\_CLIENT\_ID** environment variable of the passport-oauth-client application and copy the Client secret into the **PASSPORT\_CLIENT\_SECRET** environment variable as well.

Before we move on, don't forget about the **APP\_KEY** environment variable, this can be generated by Laravel through this artisan command:



```
php artisan key:generate
```

We are now ready to go through the Authorization Code grant, so let's open up the **passport-oauth-client** in our browser.

Here, you should see something like this: a simple application to demonstrate the flow of the Authorization Code Grant:



Before you click the link, let's just go through the flow.

There are 2 parts to this authorization flow.

In the first part, the client application will redirect the user to the server application they should grant access to. The server application needs to know which client it's allowing access to and the client needs to send that with the redirection of the user. If you open the **passport-oauth-client** project in your code editor and go to **routes/web.php**, take a look at the following route:

```
<?php
```

```
Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => env('PASSPORT_CLIENT_ID'),
        'redirect_uri' => env('APP_URL').'/callback',
        'response_type' => 'code',
        'scope' => '',
    ]);

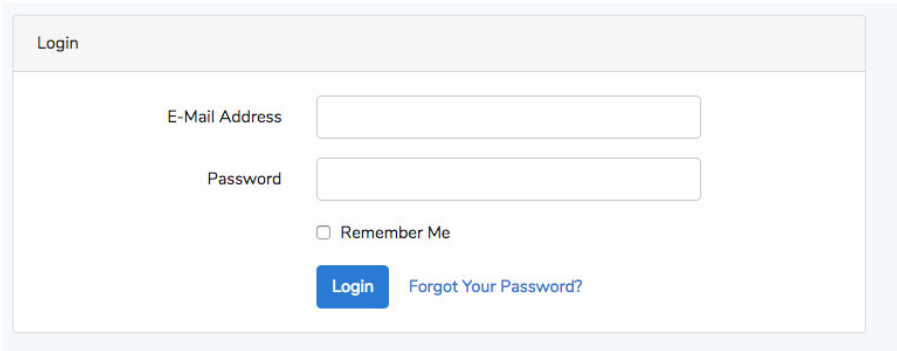
    return redirect(env('BOOKSTORE_URL').'/oauth/authorize?'.$query);
});
```

In this redirect, we are adding 4 query parameters to give the information to the server while redirecting the user which is:

- **Client ID** - The client id we have created before through an artisan command. This is used by the server to identify which client is being authorized.
- **Redirect URI** - The URL the user should be redirected to once the validation and authorization are done.
- **Response Type** - The type of grant we are using. Since this is the Authorization Code grant, we use the value code.
- **Scope** - A list of scopes the client is authorized to. As an example, Github uses scopes to tell if a client can access the user repositories and if the client has both read and write access to the repositories.

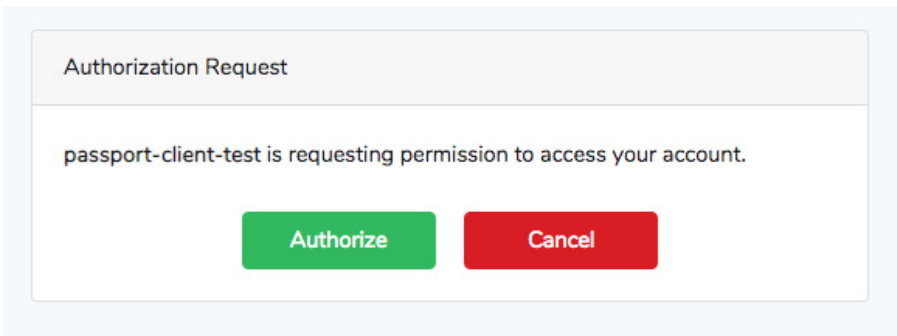
Let's click on the link and see what happens. If everything is set up correctly, you should end up on the login page like below:

## BUILD YOUR API



A screenshot of a web form titled "Login". The form has a light gray header with the title. Below the header, there are two input fields: "E-Mail Address" and "Password". Below the "Password" field, there is a checkbox labeled "Remember Me". At the bottom of the form, there is a blue button labeled "Login" and a link labeled "Forgot Your Password?" in blue text.

If you enter the credentials for John Doe we created earlier, you should end up at a page like the one below:



A screenshot of a web form titled "Authorization Request". The form has a light gray header with the title. Below the header, there is a message: "passport-client-test is requesting permission to access your account." At the bottom of the form, there are two buttons: a green button labeled "Authorize" and a red button labeled "Cancel".

When we registered our client earlier, we told the server application the name of the client. In the screenshot above, you can see that the server application has acknowledged the **Client ID** sent from our client application, and that the user can authorize the client now.

When you click on the authorize button, the server will redirect you to the URL given in the **redirect\_uri** route query parameter, which we defined in the **/redirect** route in our **passport-oauth-client** project earlier. The server will add a query parameter to this redirect named **code**, which will contain the

authorization code. This will also be the beginning of the second part of the flow.

Go back to the **passport-oauth-client** project in your editor, again in the **routes/web.php** file. Here, we will take a look at the **/callback** route since this is the route the user is being redirected back to.

```
<?php

Route::get('/callback', function (Request $request) {
    $http = new GuzzleHttp\Client;

    $response = $http->post(env('BOOKSTORE_URL').'/oauth/token', [
        'form_params' => [
            'grant_type' => 'authorization_code',
            'client_id' => env('PASSPORT_CLIENT_ID'),
            'client_secret' => env('PASSPORT_CLIENT_SECRET'),
            'redirect_uri' => env('APP_URL').'/callback',
            'code' => $request->code,
        ],
    ]);
    $tokens = json_decode((string) $response->getBody(), true);
    $user = fetchUser($tokens['access_token'], $http);

    return view('authenticated', array_merge($tokens, $user));
});

function fetchUser($accessToken, $http){
    $response = $http->get(env('BOOKSTORE_URL').'/api/user', [
        'headers' =>[
            'Accept' => 'application/json',
            'Authorization' => 'Bearer '.$accessToken,
        ]
    ]);

    return json_decode((string) $response->getBody(), true);
}
```

In this route, we make a POST request to the server application, where we send along the following parameters:

- **Grant type** - Which in this case is `authorization_code` since this is the grant type we are using.
- **Client ID** - Our client ID again
- **Client Secret** - The client secret code we received when we created the client in the server application.
- **Redirect URI** - Which is the redirect URL the user was redirected back to.
- **Code** - The authorization code we just received, when we were redirected back.

The server application will validate all this information to make sure that the client making the POST request is indeed the client that has just received the authorization code and has the proper secret.

The server will then send back a JSON object containing:

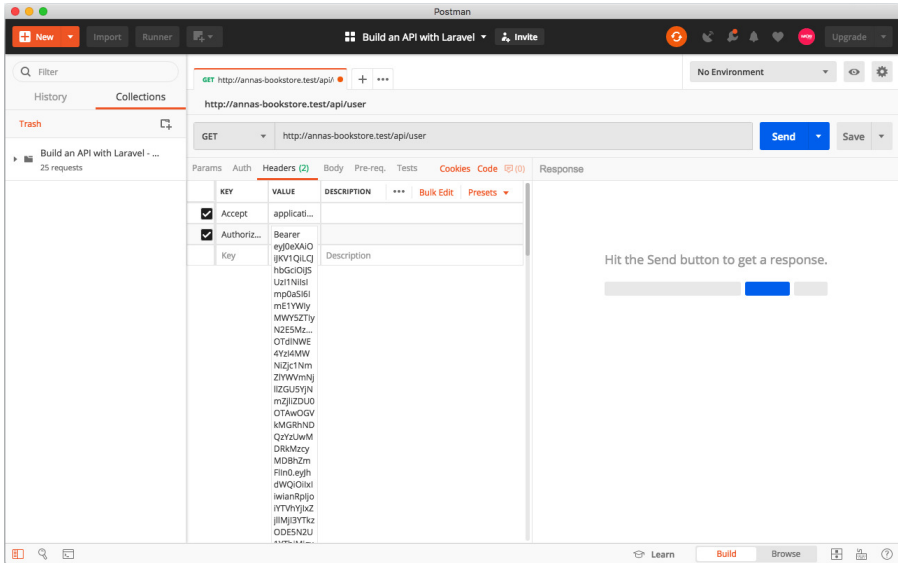
- **Access token** - Which is the token we need to make an authorized request to the server.
- **Refresh token** - Which is a token we can use to get a new access token when the current one is expired.
- **Expires in** - Which is an integer that conveys how long the current access token is valid.
- **Token type** - Which in this case contains the word "Bearer" to tell that the token is a bearer token. In the code above, you can see that we use this in our Authorization header to be able to make authenticated requests.

You can see that we receive this data in our **\$tokens** variable, which we then utilize to fetch the user from the server application and present these data along with the token data in a view.

Click the **Authorize** button if you haven't done so already and you should see



## BUILD YOUR API



Then try to make the request again, and you should see a status **200 OK** along with a JSON object of the user.

This shows how important the access token is and how much you can do with it once you get it. This is also why it's essential to protect this token, so it doesn't fall into the wrong hands. A token like this would be stored in an encrypted database of the client application — the same goes for the refresh token.

In many cases, the access token has a short life so that the refresh token is needed more often. The benefit of this is that the refresh token is not the only thing needed to get a new access token.

To get a new access token through a refresh token, you would need to do something like this:

```
<?php

$http = new GuzzleHttp\Client;

$response = $http->post('http://your-app.com/oauth/token', [
    'form_params' => [
        'grant_type' => 'refresh_token',
        'refresh_token' => 'the-refresh-token',
        'client_id' => 'client-id',
        'client_secret' => 'client-secret',
        'scope' => '',
    ],
]);

return json_decode((string) $response->getBody(), true);
```

Here, you still need the **Client ID** and **Client Secret** along with the refresh token. That's a lot of information a potential hacker would need to get.

Laravel Passport lets you decide how long your access tokens will live, but a good estimate is that they will last for about a year.

### *Implicit*

The Implicit grant is very similar to the Authorization Code grant, in that it makes use of the first flow of the Authorization code. The client will redirect the user to the server, where the user will authenticate itself and authorize the client, but then it differs from the Authorization Code grant by returning an access token, instead of the second part where a POST request should be done before the access token, refresh token, and so on are received.

The implicit grant is intended to be used for single page applications and the like. This is the reason why there is no refresh token, because the browser does not have any way of keeping refresh tokens private since all code and data are easily accessible.



Because it is very similar to the Authorization Code grant, we can actually test it out relatively easily. It only requires a few changes in our **annas-bookstore** and **passport-oauth-client** projects. Let's start by updating our **annas-bookstore** project to tell our server to enable implicit grant.

In the **app/Providers/AuthServiceProvider.php** add the **Passport::enableImplicitGrant()** method just under the **Passport::routes()** method like this:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as
    ServiceProvider;
use Laravel\Passport\Passport;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        // 'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();
    }
}
```

```

        Passport::routes();

        Passport::enableImplicitGrant();
    }
}

```

That is all we need to do in this project. Now go into the **passport-oauth-client** project and let's open the **routes/web.php** file and edit a bit in our routes.

In the **/redirect** route, change the **response\_type** from **code** to **token** like this:

```

<?php

Route::get('/redirect', function () {
    $query = http_build_query([
        'client_id' => env('PASSPORT_CLIENT_ID'),
        'redirect_uri' => env('APP_URL').'/callback',
        'response_type' => 'token',
        'scope' => '',
    ]);

    return redirect(env('BOOKSTORE_URL').'/oauth/authorize?'.$query)
        ;
});

```

Since we are reusing our Authorization Code redirect URL, we need to stop this from going through the regular process of posting and so forth. Go to the **/callback** route and add a call to **dd()** in the top of the route like this:

```

<?php

Route::get('/callback', function (Request $request) {
    dd();
});

```

```

$http = new GuzzleHttp\Client;

$response = $http->post(env('BOOKSTORE_URL').'/oauth/token', [
    'form_params' => [
        'grant_type' => 'authorization_code',
        'client_id' => env('PASSPORT_CLIENT_ID'),
        'client_secret' => env('PASSPORT_CLIENT_SECRET'),
        'redirect_uri' => env('APP_URL').'/callback',
        'code' => $request->code,
    ],
]);
$tokens = json_decode((string) $response->getBody(), true);
$user = fetchUser($tokens['access_token'], $http);

return view('authenticated', array_merge($tokens, $user));
});

```

In the browser, try going through the authorization process again. You should end up on a blank page, but take a look at the URL, which should look something like this:

```

http://passport-oauth-client.test/callback#access_token=
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImp0aSI6IjYjQ5Mzk1NGIxMWUxZjQ1ZTY0ZDZlLmEidViDIIZdoWZ6cNsuzP7T6botfxDhvJxAb7gRLKgs817AsN1fi0J-VVsQ9vKg9EB60Pa0rwMjAsmr5JKG0IXiqbeY4Hd-a7Sd2D7ZEshCUZCDFyGUY17FyMeB8Vel-A06q5F0z3VLTyXBsheLK5iEG1wov03l-mCo97w1sacpMgbP4Gf1-eexjP3zJtmji2Pi7MBCsiKIS1yD0T38ign9dk3mTlmiPYwp2Z9B9oayauCgAv3P3Lri87b8tWaxWFm8JS0t4sT3i5AXzdbnn9RZq4DEWBby07Ksx1y0p-CkXX68iC_G6KGprC7cLJQPrpyF7qFqL-Kq8L0GiQtj0jgqVks1x19EQ710LYCdEp1B0CyxsVBQDmAAPfZ5nET_qHibwA-E1mibehxxtA_tnUCcAZ9zr99NRv1RrHr-Sga10bBWg_lho_9tXGshgenD6zmFtqbEhT5gSC98boZjH8kHtd80Bwv2wuKVdDESrk9f7PHQyx

```

```
-oATfuw&token_type=Bearer&expires_in=31622400
```

Take a look at the sign after /callback, which is a # sign instead of a ? sign. This is done in such a way since a ? sign will make a server able to see the access token, but to avoid that, and ensure that a browser can read the value through JavaScript, a # is used instead.

### *Password Credentials*

Password Credentials is a grant for first-party clients on the web, desktop, and mobile devices. It's made to make the best user experience on these platforms, but it's worth noting that it's only intended for trusted first-party clients.

We can test this grant pretty easily through Postman, but before we do this, we should ensure that we have a client we can use. Since this grant is different to Authorization code and Implicit grant, we cannot reuse our client. We have to get into the terminal and rerun the artisan command, and at the same time tell artisan that we want a password grant client through a flag like this:

```
php artisan passport:client --password
```

You will be asked what to name your client — you can name it what you want, but we will name ours **Password Grant Client**.

You will then see an output containing the **Client ID** and **Client Secret** like this:

```
Client ID: 2  
Client secret: jRH1WyN9oPXkH96L3d1Qyap7yVBbUxYkuxV8qxAv
```

With that taken care of, we can continue in Postman. Here, you should create a new request. Enter the request URL:

```
POST: /oauth/token
```

Remember to add your domain to the URL. In the case of our setup, the domain is **annas-bookstore.test** which will result in a URL like this:

```
http://annas-bookstore.test/oauth/token
```

In the request section, click the **Body** tab and choose **form-data**. Since we have multiple parameters, it would be a little easier for us to use **Bulk Edit** mode, so click on that, and you will see an empty text area. For the Password Credentials grant, we need the following parameters:

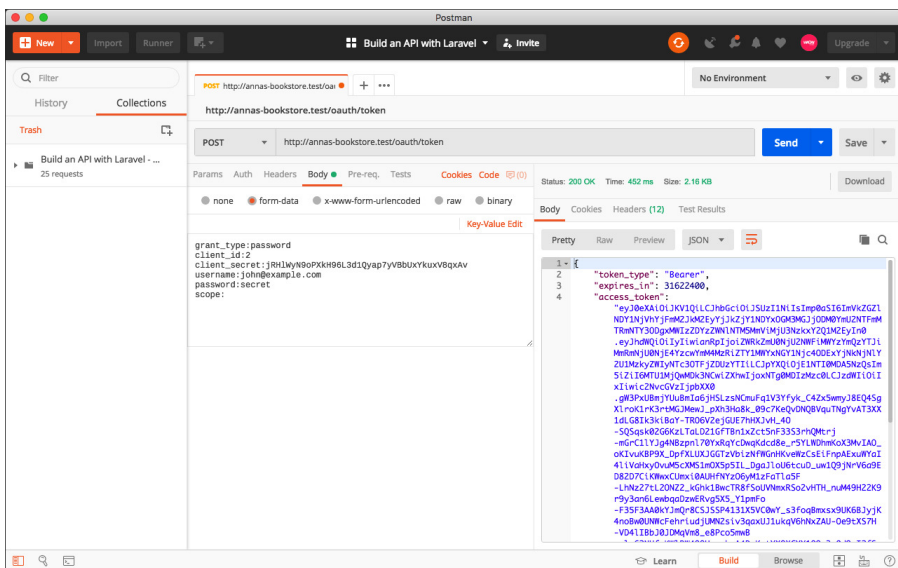
- **Grant type** – To tell which grant type we are using — in this case, it's password.
- **Client ID** – Which is the client ID we created above.
- **Client Secret** – Which is the client secret we created above.
- **Username** – Which is the email of the user.
- **Password** – Which is the password of the user
- **Scope** – Which, just like before, tells the scope of the client. We will just leave this empty.

In Postman, these parameters should be typed in like this, in **Bulk Edit** mode:

```
grant_type:password  
client_id:2
```

```
client_secret:jRHlWyN9oPXkH96L3d1Qyap7yVBbUxYkuxV8qxAv
username:john@example.com
password:secret
scope:
```

Remember to use your own **Client ID** and **Client Secret**. Click the **Send** button and you should see a response with a token like in the image below:



In this case, you will get both an access token and refresh token.

## Client Credentials

This grant type is the simplest of those we have listed, but it can also be the most dangerous if it's not used in the correct context. It's mostly suitable for **machine-to-machine** authentication where one machine is doing work on the other via an API.

In contrast to the other grant types, this one does not require a specific user's permission to work, but instead you use a middleware on the particular routes that this grant can access.

Let's test it out. Just like the Password Credentials grant, we need to create a new specific client for this grant type. We have to get into the terminal again and rerun the artisan command. This time, we will tell artisan that we want a password grant client through a flag like this:

```
php artisan passport:client --client
```

You will again be asked to name your client, and we will name it **Client Credentials Grant** Client. You will then see an output containing the **Client ID** and **Client Secret** like this:

```
Client ID: 3  
Client secret: C9wpwJizRQa0RPFH0Ka0Vg8Spsd8mIQWNtWPD4kT
```

Before we test this out, we have to make some changes to our **annas-bookstore** project again. First, go into **app/Providers/AuthServiceProvider.php** and comment out the call to the **Passport::enableImplicitGrant()** method.

Then go into **app/Http/Kernel.php** file and add the **CheckClientCredentials** middleware, with the key **client** to the **\$routeMiddleware** property like this:

```
<?php
```

```

use Laravel\Passport\Http\Middleware\CheckClientCredentials;

protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\
        AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings
        ::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::
        class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::
        class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::
        class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified
        ::class,
    'client' => CheckClientCredentials::class,
];

```

Then go into **routes/api.php** and add a test route we can use to check the authentication like this:

```

<?php
Route::get('/test', function(Request $request){
    return 'authenticated';
})->middleware('client');

```

Go into Postman again and make a new request. Enter the request URL:

```
GET: /api/test
```

Remember to add your domain to the URL. In the case of our setup, the domain



is **annas-bookstore.test** which will result in a URL like this:

```
http://annas-bookstore.test/api/test
```

In the request section, click the **Headers** tab, and in the **Key** field enter **Accept**. In the **Value** field enter **application/json** and hit the send button.

You should see a **401 Unauthorized** status code and a message saying that you are unauthenticated.

Create a new request. Enter the request URL:

```
POST: /oauth/token
```

Remember to add your domain to the URL. In the case of our setup, the domain is **annas-bookstore.test** which will result in a URL like this:

```
http://annas-bookstore.test/oauth/token
```

In the request, section click the **Body** tab and choose **form-data**. Since we have multiple parameters, it would be a little easier for us to use **Bulk Edit** mode, so click on that and you will see an empty text area. For the Password Credentials grant, we need the following parameters:

- **Grant type** - To tell which grant type we are using — in this case, it's **client\_credentials**
- **Client ID** - Which is the Client **ID** we created above
- **Client Secret** - Which is the client secret we created above
- **Scope** - Which, just like before, tells the scope of the client, we will just leave this empty

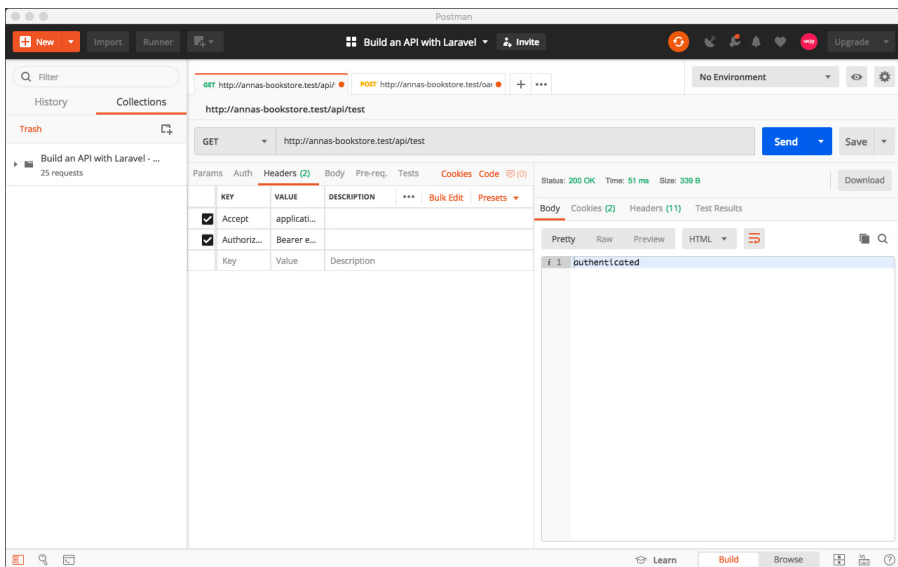
In Postman, these parameters should be typed in like this, in **Bulk Edit** mode:

## BUILD AN API WITH LARAVEL

```
grant_type:client_credentials
client_id:3
client_secret:C9wpwJizRQa0RPFH0Ka0Vg8Spsd8mIQWNtWPD4kT
scope:
```

Hit the send button and copy the contents of the **access\_token** member.

Go back to the other request from before, add a new header in the **Key** field type **Authorization**, and in the value type **Bearer** add a space and add the contents of the access token, before you as the last step hit the **Send** button. You should then see an **authenticated** message and a status code of **200 OK** like this:



## *Personal Access Tokens*

This isn't a grant type but a functionality provided by Laravel Passport. This makes it possible for a user to login to the server application and issue a personal access token, which can be used by a client to access the user's data on the server.

It removes the entire authentication process and puts the responsibility on the user. It's a great way to quickly test out your API or a more straightforward way to issue tokens.

We won't be covering this type right now, but we will make use of it when we start to develop our API. At this point, we will also take a look at the Vue components that ship with Laravel Passport.

## *Which to choose?*

We have now been through the most common grant types when it comes to OAuth2. Which one to use depends a lot on your API and how the consumers are going to work with it.

If you have an API that will be consumed by other web applications, with a server making the requests to the API, the Authorization Code Grant is the one to use.

If you have an API that will be consumed by a web application that mainly runs in the browser or a mobile app, it depends whether the consumer is first party or third party.

For third-party consumers, you should use the Implicit grant. For first-party consumers, you can use the Password Credentials Grant.

However, there is one more solution to authenticate first-party applications

that come right out of the box with Laravel Passport. It's especially convenient if you plan on hosting your frontend application together with your Laravel application. Also, in the case where you are using Vue, React or a similar javascript framework to create a single page application instead of using Laravel Blade and server-side rendered HTML pages.

For this case, Laravel ships with a middleware called **CreateFreshApiToken** that takes care of the entire token flow, so you don't have to spend time issuing anything. In fact, Laravel ships with a Vue boilerplate where everything is set up and ready for you to build something with it.

The way this is kept safe, is by using the middleware on the **web** group of middlewares, so that API tokens are only issued when a user is authenticated and signed in. Your logins would use the old conventional sign-in forms, like you would do with the artisan command **make:auth**. When your users are being redirected to the admin page, you would then serve your Vue or React application, and the API token flow would take place.

The advantage is that you can use all of the robust security features that ships with Laravel when signing users in. If you plan on hosting your frontend application together with Laravel, this is an excellent way of making a single page application, which is a way we would highly recommend over using the Password Credentials grant, since so much is taken care of right out of the box.

If you are using the Vue boilerplate that ships with Laravel, you only have to add the middleware to the **web** property of the **app/Http/Kernel.php** file like this:

```
<?php
```

```
protected $middleware = [
    \App\Http\Middleware\CheckForMaintenanceMode::class,
    \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
    \App\Http\Middleware\TrimStrings::class,
    \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull
        ::class,
    \App\Http\Middleware\TrustProxies::class,
    \Laravel\Passport\Http\Middleware\CreateFreshApiToken::class,
];
```

This middleware will attach a token cookie to your responses, which Vue will use to consume your API.

This is everything we need to know about Passport, and for now, since it is installed into our application, our API is secured. Let's take a look at how we can get the ball rolling and actually begin to build our API.

## Get the ball rolling

We now know how to secure our API, we have planned what to develop and how our API should look. It's time to start writing some code. But since there are multiple resources and an entire specification we should implement, where would be the best place to start?

In this case, it's easy to start thinking of our application from the outside-in, where you focus on authentication first, since this is also the first thing a user would be presented to. Then moving on to how users are going to work, how you handle books, and so on and so forth.

In many cases, authentication can be complicated and a steep climb initially, when you start on a project. Why choose the hardest part and end up demotivated because you never found the flow and joy in building your project?

Here, we recommend that you find an easier approach. Instead of thinking of

your application from the outside-in, instead asking yourself the question: “What is the easiest part I can build first?”. Not because you are trying to cheat, but because you’ll get the ball rolling, you’ll become motivated, you’ll get things done and feel the progression.

Take a look at the following resources we have identified in our planning phase:

- **Books**
- **Authors**
- **Comments**
- **Users**

Which of these is easiest to implement and which would also make it easy for us to implement the conventions from the JSON:API specification? Let’s go through each of them.

### *Users*

If we start by implementing our users, we end up with the outside-in approach, and since we are dealing with a multi-tenant application, we would be implementing roles for administrators as well. If you remember from our planning phase, these will share the same model and therefore also share a lot of the same code. Also, users can comment on books — we would not be able to begin this part until books were implemented. From this, we can conclude that beginning with users will give us a much harder start and that’s not what we want.

### *Books*

A Book is our central resource — it’s what the entire application will revolve around. It would make sense to start here since these cover so much ground. But a book has a relationship to one or more authors which makes this a bit

more complicated, since a book can't exist without an author and you would have to implement these as well.

Also, books can have comments, which again will require us to work through the entire user implementation. So maybe books aren't the best place to start either.

## *Authors*

If we take a look at an author, the model is actually straightforward. A single resource, that only provides information about a person, and that's it. It does have a relationship to one or more books, but it is not preventing us from building the author model and most of the API around it. If we take a look at the relationship to books, whether it's a **one-to-many** or **many-to-many** relationship, the direction of the relationship will tell that it's always either the book that would have a foreign key, pointing to the author, or a **many-to-many** pivot table that has a foreign key that points to the author. This tells us that authors can stand on their own, they do not need anything to exist in our application, and therefore makes a good candidate for the first thing to implement.

## *Comments*

The model for comments is, as authors, pretty simple. A comment has a message and a timestamp for when it's been created, and that's it. The thing that makes them a bit more complicated is the relationships, since it has a relation to both the user who has written the comment and the book the comment has been written for. Because of this, they can't stand on their own and wouldn't make a natural starting point.

*And the winner is...*

In this case, we will be starting with Authors. We will have a natural model to implement, and it would be possible to begin implementing the JSON:API specification into Laravel as well with this resource. The only hindrance we will face is when we need to implement a relationship, but then it would be possible to implement books since authors will exist in our application.

By looking at your application this way, you find an easy approach. We know where to start, and it would be much easier to decide what to implement next and just follow the path. Like we have already mentioned, it would make much sense to build the books resource next, as authors are related to books. Of course, there will be obstacles. After all, it is software development. But when we hit these, we are in a better position, where it's easier to make a decision because we are in the flow, we are progressing and heading toward the goal. At this point, we will have a better grasp on what we need to finish — we're not standing in the beginning in front of an overwhelming amount of possibilities.

Now that we know what to do, we will begin by building the Authors resource, and we finally get to do some actual coding.

## Building our first resource

We are ready to start writing some code and implementing our API. We have planned everything out, we have documented our API, so let's build it.

For the rest of this chapter, we will build the author resource. We will start by making sure we have an API key we can use with Postman. To do this, we will use the Vue components that ship with Laravel Passport, since these make it really easy to issue personal access tokens.

We will then commence by taking a look at the attributes we have identified



for our resource and make the model for it.

Then we will make sure that we have some test data to develop up against and show you how you can create artisan commands yourself for a smoother development cycle.

After this, we will create our first route for fetching a resource, setting up our controller and then we will have a look at Laravel's API Resources and use these to adhere to the JSON:API specification's conventions about resource objects.

Afterward, we will build our next route to fetch a collection of resources. Here, we will have a look at Laravel's API Resources collections to again adhere to the JSON:API specifications conventions about resource collections.

We will build a route for creating a resource by posting a resource object to our API, which should create a new model from this.

Then we will build a route to update a resource, using a PATCH request that only updates the attributes included.

We will then end this chapter by building a delete request for deleting resources through our API. Throughout building all the requests, we will test these using Postman.

You might wonder why we won't be building everything for the authors' resource to get this out of the way, but we want to begin by building a basis we can work from to ease you into tests later on and have the authors' resource backed by tests that give us much more confidence. This is merely a stepping stone to that point. But don't worry, there's plenty of interesting things to learn here as well, so don't be discouraged.

Let's get on with it and start by installing Laravel Passport's Vue components.

## Authentication and Laravel Passport Vue Components

As mentioned earlier, Laravel Passport ships with some Vue components. These can be included on any page behind authentication and will give you everything you need in terms of creating clients in your application. Behind the scenes, the components use the Laravel Passport API to issue tokens, so by using these, you can skip a lot of work.

We will, of course, leverage this since we need an access token to test our API from Postman. So go into your terminal and run the following artisan command:

```
php artisan vendor:publish --tag=passport-components
```

This command will copy the Vue components from the Laravel Passport folder inside the **vendor** directory to your **resources** directory, so you can actually compile the assets and use them in your application.

Next, we need to register the components in the **resources/js/app.js** file that is created out of the box by the Laravel installer. Here, you should register the components like this:

```
Vue.component(
  'passport-clients',
  require('./components/passport/Clients.vue').default
);

Vue.component(
  'passport-authorized-clients',
  require('./components/passport/AuthorizedClients.vue').default
);
```

## BUILD YOUR API

```
Vue.component(
  'passport-personal-access-tokens',
  require('./components/passport/PersonalAccessTokens.vue').
    default
);

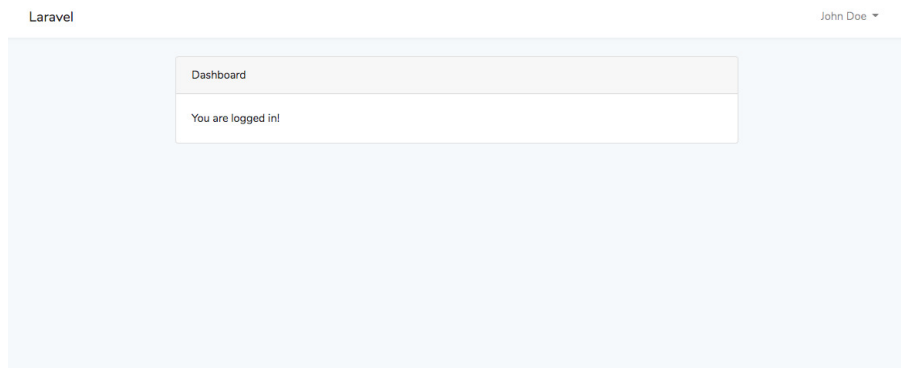
const app = new Vue({
  el: '#app'
});
```

Then you should go back to the terminal and compile your assets like this:

```
npm run dev
```

We are almost there, we just have to make sure to add the components to our markup, so that they will appear in the right place.

Earlier, we ran the **make:auth** artisan command to scaffold our entire authentication. If you sign in to our application, you should see something like this:



Since our application shouldn't do anything else — it will be an API only application — we can cheat a bit and insert the components here.

We find this view by going into **resources/views/home.blade.php**. Here, you should remove the “You are logged in!” text and insert the components like this:

```
<div class="card-body">
    @if (session('status'))
        <div class="alert alert-success" role="alert">
            {{ session('status') }}
        </div>
    @endif

    <passport-clients></passport-clients>
    <passport-authorized-clients></passport-authorized-clients>
    <passport-personal-access-tokens></passport-personal-access-
        tokens>
</div>
```

If you refresh the page you should see something like this:

The screenshot shows the Laravel dashboard interface. At the top left is the "Laravel" logo, and at the top right is the user name "John Doe" with a dropdown arrow. The main content area is titled "Dashboard" and contains two sections:

- OAuth Clients**: This section has a "Create New Client" link in the top right. It contains a table with the following data:
 

| Client ID | Name                 | Secret                                       |   |
|-----------|----------------------|--|---|
| 1         | passport-client-test | GfVhVAgFyRw1DaP9nDywYMT9wr1SZz4zYMa5r<br>vBg | <a href="#">Edit</a> <a href="#">Delete</a> |
- Personal Access Tokens**: This section has a "Create New Token" link in the top right. Below the link, it says "You have not created any personal access tokens."

If you made a lot of authorizations when we were testing the **passport-oauth-client** you might have a lot of registered clients. If so, you can just revoke all of these.

Just one more thing before we move on. If you take a look at the **resources/views/layouts/app.blade.php** file and look at the first **div** inside the **body**, you should see that the **div** has an **id="app"** attribute.

Remember this when you go back to the **resources/js/app.js** file and take a look at the Vue instance like below:

```
const app = new Vue({  
  el: '#app'  
});
```

Here, the **el** property tells Vue to mount the application on the element that has the **id** attribute with a value of **app**. Since the **resources/views/home.blade.php** extends the **resources/views/layouts/app.blade.php** file, everything inside **resources/views/home.blade.php** will be inside the mounted Vue application, and that is why we could just copy the components in here, and they worked right away.

Now we have a way to create personal access tokens we can use with Postman, so let's move on to making the model for our resource.

### *The model*

We will start with the model both to set up our database, and also because the artisan command for making models can do a lot of other work for us. Go into your terminal and run the command:

```
php artisan make:model -a -r Author
```

Because we gave it the **-a** flag, this command will create a model, a database migration, a factory, and a controller. The **-r** flag tells the generation of the controller to create a resource controller. What this means is that the controller will contain methods for each REST verb from the beginning. You can do this when creating controllers as well, and even give an **-api** flag that removes the methods for edit and create, since these are used to show the creation and edit views in a regular/non-API application.

We will have some renaming to do since we want to keep most of our naming in a plural form. It's better to do this now, so we keep consistency. Because it's required by Laravel that models keep a singular naming form, this will be the only class that has a name in singular, the rest of our classes that revolve around resources will be in plural naming form.

Let's rename the various classes then, remember to rename both the filename and the class name inside the class:

- **app/Http/Controller/AuthorControllers** to **app/Http/Controllers/AuthorsController**
- **database/factories/AuthorFactory** to **database/factories/AuthorsFactory**

Let's edit the generated migration so our database will be set up correctly. If we take a quick look at the attributes we identified for our authors resource:

- **Authors**
  - - Name
  - - Updated At (*Comes from Laravel*)
  - - Created At (*Comes from Laravel*)

As mentioned earlier, it's a quick, simple model, with only one attribute that holds the name of the author. For this, we just need a string which then will create a VARCHAR in our database. We will remove any restrictions on the length so the name can be at least 255 characters long to be sure that it can hold long names as well.

Go to **database/migrations/xxxx\_xx\_xx\_create\_authors\_table.php** and create a new string column with the column name of name like this:

```
<?php

public function up()
{
    Schema::create('authors', function (Blueprint $table) {
        $table->bigIncrements('id');
        $table->string('name');
        $table->timestamps();
    });
}
```

Go to **app/Author.php** and let's set the fillable attributes array and add the **name** attribute like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Author extends Model
{
    protected $fillable = ['name'];
}
```

```
}
```

This ensures that we can add the name attribute when using a method like **Author::create()** because it's now mass assignable.

We have now made sure that we can store our data about authors, but we need some data to both build and test up against. Let's take care of this next.

### *Test data*

The Laravel framework comes with a method to handle test data, called factories. Factories make it possible to generate models with test data and populate your database with test data very fast and easy.

When we generated the various classes for our authors' resource, we also got a factory, so let's take a look at our factory file at **database/factories/Authors-Factory.php**. You should be seeing a file that contains the following:

```
<?php

use Faker\Generator as Faker;

$factory->define(App\Author::class, function (Faker $faker) {
    return [

    ];
});
```

In the array, we can list the attributes we want to have populated by our factory. We only have one attribute, since Laravel takes care of the **created\_at** and **updated\_at** attributes, we just need to add our **name** attribute here like this:



```
<?php

use Faker\Generator as Faker;

$factory->define(App\Author::class, function (Faker $faker) {
    return [
        'name' => $faker->name(),
    ];
});
```

The great thing about factories is that they utilize the Faker package. The Faker package can create fake test data for a lot of different things like emails, phone numbers, addresses and in our case generating names. This is especially one of the reasons why factories make it so easy to create test data.

Now we need something that can call our factory and actually create the models with test data. For this we need to create a database seeder. Unfortunately, this wasn't created when we generated our model, but it can be done in a quick artisan command like this:

```
php artisan make:seeder AuthorsTableSeeder
```

Of course, you can name the seeder whatever you want — we have named ours **AuthorsTableSeeder** to follow our naming conventions.

To be able to call this database seeder, we need to add it to the **database/seeds/DatabaseSeeder.php** class like this:

```
<?php

use Illuminate\Database\Seeder;
```

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        $this->call(AuthorsTableSeeder::class);
    }
}
```

Now go into the **database/seeds/AuthorsTableSeeder.php** file and make a call to our factory like this:

```
<?php

use Illuminate\Database\Seeder;

class AuthorsTableSeeder extends Seeder
{
    public function run()
    {
        factory(\App\Author::class, 5)->create();
    }
}
```

In the call, we can declare which model we want the factory to work with and how many models to create. In this case we will be making 5 authors.

Now that it's done, we should migrate and seed our database. We can do both with a single artisan command like this:

```
php artisan migrate --seed
```

If you want to start fresh, you can use the following artisan command, but remember that you will have to create both user and clients again though:

```
php artisan migrate:fresh --seed
```

If you have already run the **migrate:fresh** command, don't worry. We'll just cover a quick tip that we use a lot when developing our applications that can help you.

### *Development Commands*

In many projects, we use small development artisan commands to be able to set up everything for development quickly. Some projects are more elaborate than others, especially when it comes to data. Sometimes, you need to be able to do a complete reset of the application, or be able to set up your project quickly on another machine, and it is in such cases that the commands come in handy.

We also use some of these commands to fill out an application with test data, so that when handing it over to customers, they know what kind of data that is expected and they can see the functionality of the application.

Let's quickly build a command that will migrate our database from fresh, run our seeders and also make sure that we have a client we can use to create a personal access token for the first user in our database. Even though we could use the super handy **routes/console.php** file, let's instead create a dedicated command, so everything is in its own class.

Go into your terminal and run the following artisan command:

```
php artisan make:command SetupDevEnvironment --command=dev:setup
```

In your editor, go into **app/Console/Kernel.php** and in the **\$commands** property add the **SetupDevEnvironment** class to the array like this:

```
<?php

protected $commands = [
    SetupDevEnvironment::class,
];
```

If you run artisan without any commands you should see our dev:setup command on the list now. Next, let's describe the command and code the things it should do like this:

```
<?php

namespace App\Console\Commands;

use App\User;
use Illuminate\Console\Command;
use Illuminate\Support\Facades\Artisan;
use Laravel\Passport\PersonalAccessTokenResult;

class SetupDevEnvironment extends Command
{
    protected $signature = 'dev:setup';

    protected $description = 'Sets up the development environment';

    public function __construct()
    {
```

```

    parent::__construct();
}

public function handle()
{
    $this->info('Setting up development environment');

    $this->MigrateAndSeedDatabase();
    $user = $this->CreateJohnDoeUser();
    $this->CreatePersonalAccessClient($user);
    $this->CreatePersonalAccessToken($user);

    $this->info('All done. Bye!');
}

public function MigrateAndSeedDatabase()
{
    $this->call('migrate:fresh');
    $this->call('db:seed');
}

public function CreateJohnDoeUser()
{
    $this->info('Creating John Doe user');

    $user = factory(User::class)->create([
        'name' => 'John Doe',
        'email' => 'john@example.com',
        'password' => bcrypt('secret'),
    ]);

    $this->info('John Doe created');
    $this->warn('Email: john@example.com');
    $this->warn('Password: secret');

    return $user;
}

public function CreatePersonalAccessClient($user)
{

```

```

        $this->call('passport:client', [
            '--personal' => true,
            '--name'      => 'Personal Access Client',
            '--user_id'   => $user->id
        ]);
    }

    public function CreatePersonalAccessToken($user)
    {
        $token = $user->createToken('Development Token');

        $this->info('Personal access token created successfully.');
```

*(Note: The original image contains a stray closing tag </p> here, which has been removed for accuracy.)*

```

        $this->warn("Personal access token:");
        $this->line($token->accessToken);
    }
}

```

If you take a look at the **handle()** method, we go through each step in our setup. We have a dedicated method for each step, so that we have a nice descriptive name that explains what is happening. If you go into your terminal and run the artisan command, you will be presented with the migrations, the seeder, the creation of John Doe, the client creation and in the end we output the access token so that it can be copied into Postman easily.

We now have an artisan command we can run any time we want to setup the project for development and we are now able to begin working on our first route.

### *The first route*

Before we begin developing our first route, we have some cleanup to do. If you go into the **routes/api.php** file, we have a method from when we were testing the Client Credentials Grant. Remove this route, since we are not going to use it anymore.

Next, we want to create a group for our routes. The advantage here is that through groups we can make sure that all routes in that group are going through the **auth:api** middleware and so that we can also prefix our future routes. As discussed in the chapter about the JSON:API specification, it's a good idea to version our API. This can also be done via groups through prefixing, which will give us URLs that look like this:

```
GET: /api/v1/authors
```

All routes in the **routes/api.php** file are always prefixed with **api** so we only need to make a prefix for v1, which we can easily do by changing the existing user route that looks like this:

```
<?php

Route::middleware('auth:api')->get('/user', function (Request
    $request) {
    return $request->user();
});
```

Into a group like this, with the existing user route transformed into a regular route:

```
<?php

Route::middleware('auth:api')->prefix('v1')->group(function(){
    Route::get('/user', function (Request $request) {
        return $request->user();
    });
});
```

Any route we defined inside the group closure will automatically be prefix with **/api/v1**.

Now we can make our first route. Again, it's about picking the easiest path. Here, we always start out with the route for a single resource. With the routes for creating or updating a resource, there are far more things you need to consider and with the route for fetching all resources, you pretty much have to have done the work for a single resource first.

Right under the **/user** route, create a new GET route with a URI with a route parameter like this: **/authors/{author}** so that we can capture the **ID** of the author.

It should look something like this:

```
Route::middleware('auth:api')->prefix('v1')->group(function(){
    Route::get('/user', function (Request $request) {
        return $request->user();
    });

    // Authors
    Route::get('/authors/{author}', 'AuthorsController@show');
});
```

We make sure to reference our authors' controller, more specifically the show method on the controller. Since we make a resource controller, Laravel has already set up the various methods for us to use, where the **show** method is typically used to show a resource and that fits this route perfectly.

Before we move on, let's try to hit this route through Postman, so in Postman you should create a new request with a GET method and hit the newly created route. Again, the domain depends on your local configuration but for us the URL is:



```
http://annas-bookstore.test/api/v1/authors/1
```

Make sure to set a header that tells the server that you will **Accept** a response in **application/json** and a **Authorization** header with **Bearer <your access token>** before testing the route, like this:

| Params                              | Auth         | Headers (2)   | Body        | Pre-req. | Tests     | Cookies | Code | 🔔 (0) |
|-------------------------------------|--------------|---------------|-------------|----------|-----------|---------|------|-------|
|                                     | KEY          | VALUE         | DESCRIPTION | ...      | Bulk Edit | Presets | ▼    |       |
| <input checked="" type="checkbox"/> | Accept       | applicatio... |             |          |           |         |      |       |
| <input checked="" type="checkbox"/> | Authoriza... | Bearer ey...  |             |          |           |         |      |       |
|                                     | Key          | Value         | Description |          |           |         |      |       |

If you've done it right, you'll receive a 200 OK response with no content in the body. We need to have some code in our controller that returns a proper response, so let's do that next.

## The Controller

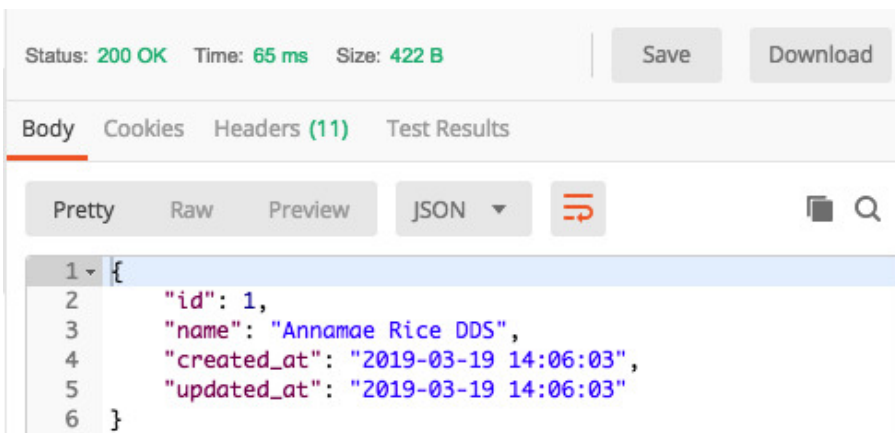
Go into **app/Http/Controllers/AuthorsController.php** and take a look at the methods Laravel has generated for us. There are some we don't need, more specifically the create and edit routes. The reason is that they are dedicated to showing a creation or edit form in a regular application, but since we are building an API, we don't need that. So let's start by removing these two methods.

Next, we need to work with the **show** method. If we look at the parameters, we see that in Laravel's generation of the controller, it has referenced our Author model for use with route model binding, so that behind the scene Laravel will find the model associated with the ID that comes in through the route.

To be able to get some results, we can be a bit lazy and just return our model as our response like this:

```
public function show(Author $author)
{
    return $author;
}
```

This is one of the areas where Laravel shines: it makes it easy for developers to get results. Go into Postman, hit the send button again, and you should see a result like this:



This result does not really adhere to the JSON:API specification so we have to do something about this.

## *Adhering to the JSON:API Specification*

If you take a look at the JSON returned by Laravel, we are not even close to adhering to the JSON:API specification. First, we need to adhere to the top level document structure having a data member that contains a resource object which will be the representation of our model.

So how do we do this? We could just go into the controller again and build the response ourselves like this:

```
public function show(Author $author)
{
    return response()->json([
        'data' => [
            'id' => $author->id,
            'type' => 'authors',
            'attributes' => [
                'name' => $author->name,
                'created_at' => $author->created_at,
                'updated_at' => $author->updated_at,
            ]
        ]
    ]);
}
```

It would give us the correct structure, but we would have to use this everywhere we want to transform our model into a response that adheres to the JSON:API specification. We could extract it and create a transformer class that could take care of it, and then reuse this. But like in so many other areas, Laravel actually has something like this already through Eloquent: API Resources.

An API Resource makes it possible for us to define how we want to transform our models into API resources. It gives us a convention we can leverage again and again. Better yet, it also handles collections, so it is not only a benefit for

single resources but for collections of resources as well.

To create a resource, we do it through an artisan command like this:

```
php artisan make:resource AuthorsResource
```

This will generate a new APIResource class, which can be found here: **app/Http/Resources/AuthorsResource.php**

If we take a look at the file, we can see one method named **toArray** that takes a request as a parameter and needs to return an array. We can use this array to define our structure for our resource. We know what the top most members we need to define should be, so let's define the **data** member in our array. Let's recreate our structure from before like this:

```
public function toArray($request)
{
    return [
        'data' => [
            'id' => (string)$this->id,
            'type' => 'authors',
            'attributes' => [
                'name' => $this->name,
                'created_at' => $this->created_at,
                'updated_at' => $this->updated_at,
            ]
        ]
    ];
}
```

You might wonder how we can reference our model attributes through **\$this** but as you will see in a moment, the model we are transforming is actually

passed upon instantiation of the resource. The **(string)** part before **\$this->id** is a casting, telling PHP that we want to cast the contents of the **id** property to a string. Remember that according to the JSON:API specification, the **IDs** must be strings.

Let's do that, go back into the controller at **app/Http/Controllers/AuthorsController.php** and replace the contents of the **show** method with the following:

```
public function show(Author $author)
{
    return new AuthorsResource($author);
}
```

This is a far cleaner approach. Everything about our transformation is kept within its own class and we can just instantiate this resource, pass it the model and we are done for now. Let's test it out in Postman, where you should see a response like this:

Status: 200 OK Time: 77 ms Size: 481 B Save Download

Body Cookies Headers (11) Test Results

Pretty Raw Preview JSON ↺

```

1 {
2   "data": {
3     "id": "1",
4     "type": "authors",
5     "attributes": {
6       "name": "Annamae Rice DDS",
7       "created_at": "2019-03-19T14:06:03.000000Z",
8       "updated_at": "2019-03-19T14:06:03.000000Z"
9     }
10  }
11 }

```

There you go. This was actually our first route implemented. Let's take a look at how we can return a collection of authors next.

### Resource collections

As we mentioned earlier, the API Resources functionality built into Laravel can help us transform both single resources and collection of resources as well. It's actually possible to get a collection from the single resource — let's try to see what kind of structure that will give us first.

Go into **app/Http/Controllers/AuthorsController.php** where we will work in the **index** method this time. Let's first make a query for all the Authors in the database and then try to make a collection of our **AuthorsResource**, like this:

```

public function index()
{

```

```
$authors = Author::all();  
return AuthorsResource::collection($authors);  
}
```



Before we go into Postman to test this, we need to define the route first. Jump into **routes/api.php** and just above the existing **/authors/{author}** route make a new GET route to **/authors** pointing to the **AuthorsController**'s **index** method like this:

```
// Authors  
Route::get('/authors', 'AuthorsController@index');  
Route::get('/authors/{author}', 'AuthorsController@show');
```

Now we can go into Postman and test the route. By now you should be familiar with what to do, so we will let you do it on your own. If you did everything correctly, you should see a result like the one below:

Status: 200 OK Time: 64 ms Size: 1.42 KB Save Download

Body Cookies Headers (11) Test Results

Pretty Raw Preview JSON  

```

1 {
2   "data": [
3     {
4       "data": {
5         "id": "1",
6         "type": "authors",
7         "attributes": {
8           "name": "Annamae Rice DDS",
9           "created_at": "2019-03-19T14:06:03
10            .000000Z",
11           "updated_at": "2019-03-19T14:06:03
12            .000000Z"
13         }
14       },
15       {
16         "data": {
17           "id": "2",
18           "type": "authors",
19           "attributes": {
20             "name": "Javier Emard",
21             "created_at": "2019-03-19T14:06:03
22              .000000Z",
23             "updated_at": "2019-03-19T14:06:03
24              .000000Z"
25           }
26         }
27       }
28     ]
29   }
30 }

```

The structure isn't right: the top level **data** member is being repeated both for the collection but again for each. How should we fix this.

Let's try removing the data member from our **AuthorResource's toArray** method, which will result in something like this:

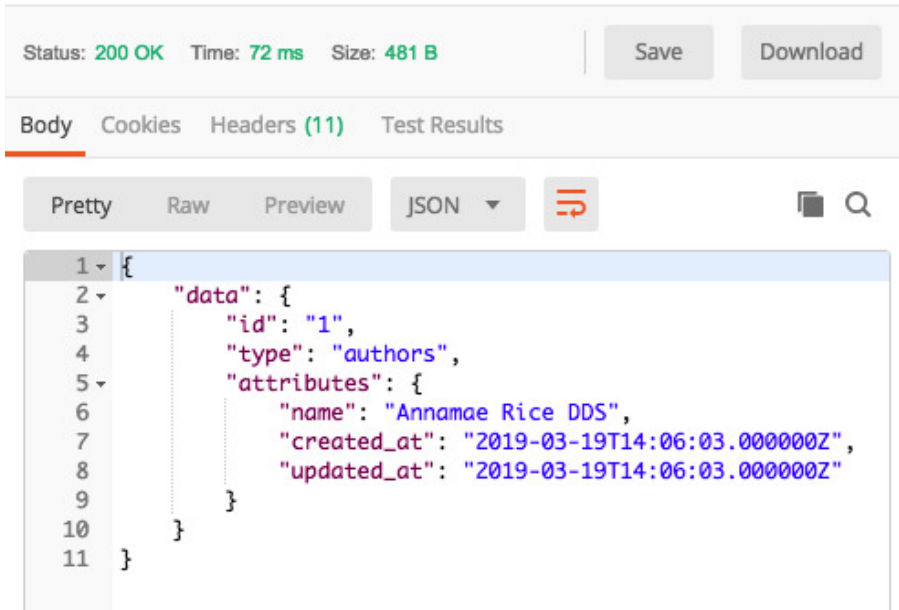


```
public function toArray($request)
{
    return [
        'id' => (string)$this->id,
        'type' => 'authors',
        'attributes' => [
            'name' => $this->name,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ]
    ];
}
```

Let's send the request again and you should see something like this:

```
{
  "data": [
    {
      "id": "1",
      "type": "authors",
      "attributes": {
        "name": "Annamae Rice DDS",
        "created_at": "2019-03-19T14:06:03.000000Z",
        "updated_at": "2019-03-19T14:06:03.000000Z"
      }
    },
    {
      "id": "2",
      "type": "authors",
      "attributes": {
        "name": "Javier Emard",
        "created_at": "2019-03-19T14:06:03.000000Z",
        "updated_at": "2019-03-19T14:06:03.000000Z"
      }
    }
  ]
}
```

The structure is correct now, but what about our single resource? If we try to make a request to it in Postman, you will get a result like this:



By default, API Resources in Laravel actually creates a top level document with a **data** member so we don't need to do anything for these members. We do, however, need to do some work for other members like **links** and **includes**, and right now that is not possible. For this, we will need a dedicated collections class and of course Laravel has us covered here as well.

This is done through the same artisan command we used to create our **AuthorResource**. You can either tell artisan that you want a collection through a **-collection** flag or by simply including the word **Collection** in your class name. We will do the latter like this:

```
php artisan make:resource AuthorsCollection
```

Like our **AuthorsResource** class, the collection class has a **toArray** method that takes a request as a parameter and returns an array. We need to edit the **toArray** method so that it returns our resource collection in the **data** member,

we do this like this:

```
public function toArray($request)
{
    return [
        'data' => $this->collection,
    ];
}
```

This does not make the biggest difference right now, but then it's a preparation for future work.

Now that we have implemented how to fetch both a single resource and a collection of resources, let's implement how to create a resource next.

### *Creating a resource*

It's time to look at resource creation. If you recall from the chapter about the JSON:API specification, in order to create a resource we need to send a valid resource object to the server. The resource object will have to contain the **type** member of the resource as well as the **attributes** member with the attributes for the resource to be a valid resource object for creating a resource. You could add an **ID** member for the server to use, but we feel that generating **IDs** is more a concern of the server or database, so we will let these take care of this.

To implement this, let's start out by defining a route. This time we need a POST route to **/authors**, pointing to **AuthorsController's store** method. Before you begin writing this route, hold on. It's actually possible to take a shortcut here — one that also makes thing a bit more descriptive. We can use a single route that creates all the routes we need for a resource, and coincidentally the route method is named **Routes::apiResource**. Let's change our existing routes into this single route, like this:

```
Route::apiResource('authors', 'AuthorsController');
```

You should test your existing routes in Postman, where they should still work just fine. If you want to see what this route does for you, go into your terminal and use the `route:list` command to get a list of the routes in the application. At the top, you will see all the routes for the authors resource, even the ones we haven't implemented yet, so let's implement these now. Go back to **app/Http/Controllers/AuthorsController.php** and let's focus on the **store** method first.

Here, it's our responsibility to take the data from a resource object and create a new author from these, so let's take the naive approach and just pretend that we will get a correct resource object and create the model like this:

```
public function store(Request $request)
{
    $author = Author::create([
        'name' => $request->input('data.attributes.name'),
    ]);
    return new AuthorsResource($author);
}
```

We create a new **Author** model and again we leverage one of the conveniences of Laravel through the **input** method on the **Request** object, which makes it possible to access data sent in the request through dot notation. We know through the conventions of the JSON:API specification that there will be a top member called **data**, we know that it will contain a resource object and we know that the **attributes** for this resource object will be placed in a **attributes** member. Last, but not least, we know that we only need the **name** attribute to create an author.

If you remember from the chapter on the JSON:API specification, we need

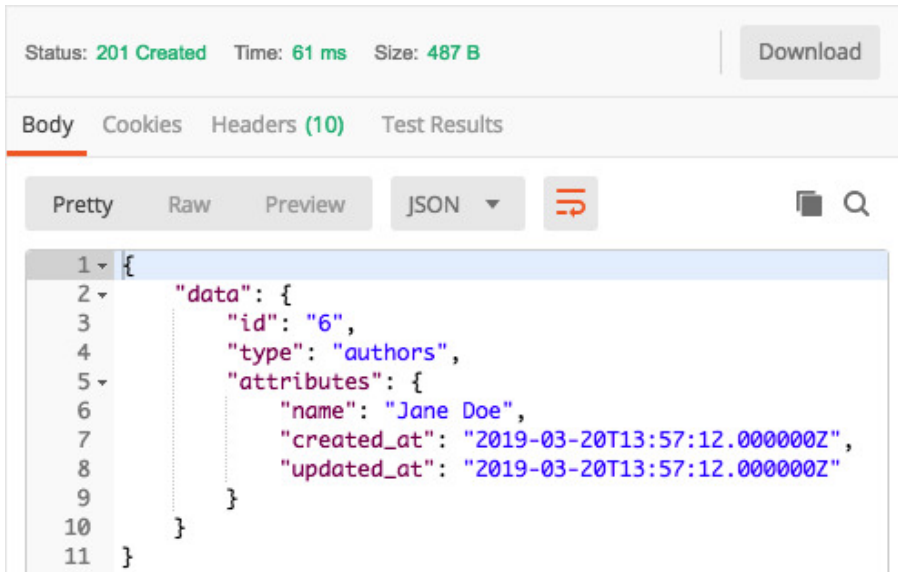
to return the created resource, we can do this easily through our existing **AuthorsResource**.

Let's test this by going into Postman, creating a new request, and making a POST request to the route **/api/v1/authors** the URL in our configuration will look like this:

```
http://annas-bookstore.dev/api/v1/authors
```

Remember the correct headers which is an **Accept** header with the value **application/json**, an **Authorization** header with the value **Bearer <your access token>** and also a **Content-Type** with the value **application/json**. We will continue using **application/json** as our content type — in the next chapter, we will take a look at how to adhere to the JSON:API specification here as well.

In the body of the request, you should create the request document with a resource object like this:



Send the request and in the response area you should see something like this:

You should also take a look at the headers sent from the server, since there is something missing here. We are not thinking about the **Content-Type: application/vnd.api+json** header just yet, but the **Location** header we should include when creating resources. Let's get back to our controller and implement this. Here, we will need to wrap the instantiation of the

**AuthorsResource** in brackets to be able to work with the instance right away. We will use this instance to access the response and add our header like this:

```
public function store(Request $request)
{
    $author = Author::create([
        'name' => $request->input('data.attributes.name'),
    ]);
    return (new AuthorsResource($author))
```

```
->response()  
->header('Location', route('authors.show', ['author' =>  
    $author]));  
}
```

We use the **route** helper function to make the URL for the Location header and since we are using the **Route::apiResource** route method, we can leverage the automatic naming of our routes together with our model to make the URL.

If you go back and test the route in Postman and take a look at the headers of the response, you should see a Location header like this:

**Location** → <http://annas-bookstore.test/api/v1/authors/7>

Great, the creation of resources is implemented. Of course, this is an extremely naive implementation, where we don't validate on anything. This is not good, you should always validate any user input, but for now we will continue on to updating our resources and then return to validation in a bit.

### *Updating a resource*

When updating a resource, we need to send a resource object again. This time, the resource object should contain both the **type** and **id** members. In the **attributes**, all or some can be included and it will be the server's responsibility to only update the attributes mentioned.

Let's head into the **app/Http/Controllers/AuthorsController.php** again and this time focus on the **update** method. Just like the **show** method for fetching a single author resource, we get a route model binding to the requested **Author** model. So our job here is to take this model and update it according to the



attributes in the resource object. This can be easily done like this:

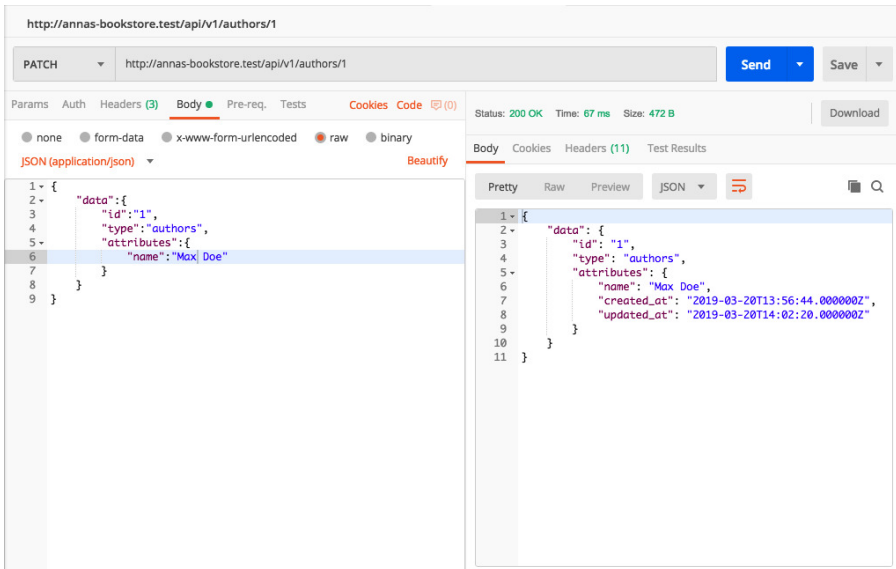
```
public function update(Request $request, Author $author)
{
    $author->update($request->input('data.attributes'));
    return new AuthorsResource($author);
}
```

Laravel makes this super easy for us — we don't even have to worry about only updating the attributes given. All of that is taken care of by simply giving the array of attributes to the **update** method on our **Author** model.

By default, Laravel will update our model with a new **updated\_at** timestamp and according to the JSON:API specification, whenever the server changes a resource other than what is being specified in the request, the server must respond with the updated resource, which we will do by leveraging the **AuthorsResource** yet again.

Let's test this out in Postman. Remember that you need to provide a URL with the ID of the resource, just like when we were fetching a single resource, and also provide that same ID in the resource object of the request document and don't forget the headers, these are the same as the create request. If you have done it correctly you should see something like this:

## BUILD AN API WITH LARAVEL



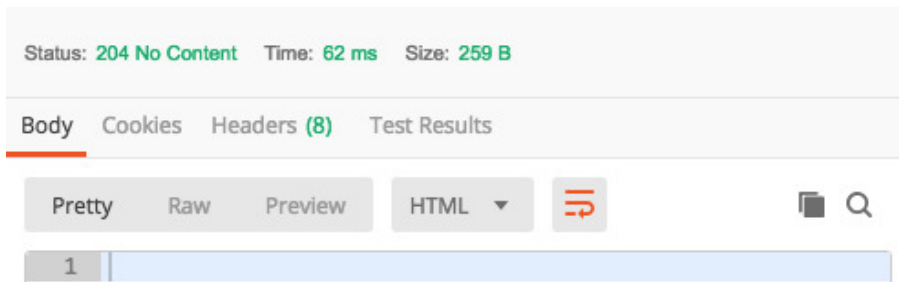
Notice how the `updated_at` member has a different timestamp than the `created_at` member now- This is because Laravel updates this behind the scenes, even though we have only updated the name.

### *Deleting a resource*

It's time to take a look at how to delete resources. This might be the simplest request we can make at all. Since it does not require any resource objects to be sent, it doesn't even need any data to be returned from the server. Let's jump into our **AuthorsController** and direct our focus to the **destroy** method. This time, we are once again being spoiled by Laravel and getting the model through route-model binding. This makes it very easy for us to delete the model since we only need to call the **delete** method on it to delete it from the database. Because we don't need to return any data, we can respond with a **204 No Content** status code and leave the body of the response empty, like this:

```
public function destroy(Author $author)
{
    $author->delete();
    return response(null, 204);
}
```

Let's test it out in Postman. Make a new delete request to **/api/v1/authors/1** and for this request we still need the **Accept** and the **Authorization** headers. If you have done everything right, you should see a response like this:



Wasn't that easy? It doesn't get simpler and it is all thanks to Laravel.

We have built the entire resource now, but we do have one thing we need to take care of, namely validation. We are being a bit too naive at the moment.

## Validation

It's time to protect ourselves a little more against user inputs. The way we do this is through validation, where we ensure that the user gives us the required data in the required data types we need. We ensure that the user delivers a request document that adheres to our conventions, so that we can get more predictable inputs we know will work.

In Laravel, validation can be done in many ways, which we will not be covering

in this book. The way we will be covering is validation through Form Requests. This makes it possible for us to have our validation logic in separate classes with a nice descriptive name. And the only thing we need to do to use them, is to change the reference from the **Request** object in our **store** and **update** methods to the new form request classes.

Let's start with a form request class for the **store** method. We can create this with an artisan command as well, so let's jump into the terminal and call the following command:

```
php artisan make:request CreateAuthorRequest
```

Next, jump into the newly created class in **app/Http/Requests/CreateAuthorRequest.php** and take a look at the **authorize** method first. This method is used to determine if the user is authorized to make the request. As of right now, we have no way of implementing this, so just set this to return **true** instead of **false**. In the **rules** method, we should return an array of validation rules. Here, we should provide the rules for both the structure of a request document, but also the data we expect. For this, we will need the following rules that requires:

- A top level **data** member
- That the **data** member contains a resource object
- That the resource object contains a **type** member
- That the **type** member has the value **authors**
- The the resource object contains a **attributes** member
- That the **attributes** member contains a **name** member
- That the **name** member is not empty

We can do it like this:

```

public function rules()
{
    return [
        'data' => 'required|array',
        'data.type' => 'required|in:authors',
        'data.attributes' => 'required|array',
        'data.attributes.name' => 'required|string',
    ];
}

```

We make sure the **data** member is **required** and make sure the data type is an **array**. This is not meant as an array in the JSON document, but a PHP array, because Laravel will decode JSON objects to associative arrays. We ensure that the **type** member is **required** and that it has the value of **authors**. We make sure that the **attributes** member is **required** and as with the **data** member we get an **array** meaning that it is an object on the JSON side. We ensure that **name** is required and that it is a **string**.

To actually use this form request and make the validation work, we need to change out the **Request** object in our **AuthorsController**'s **create** method, like this:

```

public function store(CreateAuthorRequest $request)
{
    //...
}

```

Let's test it out in Postman. We will make a request to create a new author but misspell the value of the **type** member like this:

```



1 {
2   "data": {
3     "type": "author",
4     "attributes": {
5       "name": "Max Doe"
6     }
7   }
8 }

```

When sending the request you should get a response like this:

Status: **422 Unprocessable Entity** Time: 42 ms Size: 392 B [Download](#)

Body Cookies Headers (9) Test Results

Pretty Raw Preview JSON  

```

1 {
2   "message": "The given data was invalid.",
3   "errors": {
4     "data.type": [
5       "The selected data.type is invalid."
6     ]
7   }
8 }

```

Our validation works now, but it's not quite the error message that we want. We will be tackling this in the next chapter though, so let's continue to the next form request, the one for validating our update request.

Go back into the terminal and call the following command:

```
php artisan make:request UpdateAuthorRequest
```

This will create yet another form request class, this time it's: **app/Http/Requests/UpdateAuthorRequest.php** so let's go into this.

Like before we make the **authorize** method return **true** and for the **rules** method, you can copy everything from the **CreateAuthorRequest's** rules method and paste it in here. Right above the rule for the **type** member, add a rule for the **id** member. This should be **required** and needs a data type of **string**. Add the **sometimes** rule in front of the **attributes** and **name** members, which should make the entire method look like this:

```
public function rules()
{
    return [
        'data' => 'required|array',
        'data.id' => 'required|string',
        'data.type' => 'required|in:authors',
        'data.attributes' => 'sometimes|required|array',
        'data.attributes.name' => 'sometimes|required|string',
    ];
}
```

Remember when updating a resource only the **id** and **type** is actually required—The **attributes** member and its contents are optional.

Let's change the **Request** object for the **update** method in the **AuthorsController** now, like this:

```
public function update(UpdateAuthorRequest $request, Author $author
)
{
    $author->update($request->input('data.attributes'));
```

```
return new AuthorsResource($author);
}
```

If we test it out in Postman and again make a typo for the **type** member and maybe forget the id member like this:

```
1 {
2   "data": {
3     "type": "author",
4     "attributes": {
5       "name": "Max Doe"
6     }
7   }
8 }
```

When you send this request, it should give you a response like this:

Status: 422 Unprocessable Entity Time: 52 ms Size: 437 B Download

Body Cookies Headers (9) Test Results

Pretty Raw Preview JSON

```
1 {
2   "message": "The given data was invalid.",
3   "errors": {
4     "data.id": [
5       "The data.id field is required."
6     ],
7     "data.type": [
8       "The selected data.type is invalid."
9     ]
10  }
11 }
```



Awesome. Our validation for the update request works as well.

## Summary

In this chapter, we went through authentication with Laravel Passport, where we showed how to install it into a fresh Laravel application.

We went through how OAuth 2 authentication works and went through each of the most common grant types and gave an example of how they worked — both through a client application and in Postman as well.

We talked about which grant type to use, especially when thinking of a consumer application that is first or third party.

Then we shifted gears and actually commenced the development of our API. We looked at how to get the ball rolling and which parts to implement first, from a model and outwards perspective.

We ended the chapter by implementing our authors resource and most of the conventions of the JSON:API specification.

In the next chapter, we will get much more deeper into developing our API, but before that we will take a look at Test Driven development and the tools around Laravel to help us with Test Driven development, which will give us an awesome workflow as well. Then we will set off into this new way of development by revisiting our author resource and later on continue to our books resource and implement this together with relationships between authors and books.

See you in the next chapter!

\* \* \*

## 5

# Test-driven Workflow

It's time to take a look at test-driven development — a development flow we have learned to love and we are sure you will too. Whether you are new to test-driven development or have done it for years, we're sure you will be able to learn something here.

Test-driven development is a big field in itself and of course we won't be able to cover everything about it in this book. We will instead cover how to use test-driven development to drive out the features of our API and ensure we implement the JSON:API specification correctly.

To some people, test-driven development can seem tedious, boring, and repetitive. These people have obviously not tried looking for bugs while the phone is ringing off the hook with calls from angry customers.

To us, the repetitive and boring work and the extra lines of code is worth it. If you think about it, the hours spent on writing tests are hours well spent, in contrast to when things go wrong and you spend even more hours trying to fix a bug. The extra work is also worth the confidence you get when you ship your product. You don't have to worry about anything breaking in one place if you change something in another place, and you have tests that prove your application works — or at least tell you where things break, before you

push this bug out into production. If a bug should show up — and they most certainly will — you write a test for fixing that bug and now you know it won't show up again.

If you go to Github and take a look at one of your favorite projects as well as the tests for that package, you can actually see that the tests can be used as a kind of documentation, since they show how to work with the project. They have to, to be able to prove that the project works and that is also a huge benefit.

It isn't always fun writing tests, but it's a necessity and when you get into the habit of writing tests, especially for API development, you get a workflow that is actually quite good, especially in contrast to testing everything manually in Postman. Let's just face it: this is what you will be doing if you are not writing tests, and you have to ensure somehow that things work correctly.

When you see how much ground you can cover by testing your APIs through PHPUnit instead of Postman, you won't be going back. This is one of the things we hope you will take away from this book, and we will do our best to show you how to get into a flow of writing tests and implementing the source code for those tests to pass until everything is implemented and you're done.

In this chapter, we will focus on getting started with test-driven development in Laravel and for API development in general

We will first take a look at Laravel's test tools and introduce you to the various tests types that Laravel supports out of the box.

Then, we will go through setting up your test environment, so you are ready to start your journey into test-driven development. Here, we will also introduce you to a Laravel package we have developed for a smoother testing workflow.

Then we will get right back to coding as well as implementing tests for our existing author resource, so you can see the difference from our manual

Postman tests to the test-driven approach.

We will conclude this chapter by implementing our book resource, which is the resource the entire application is centered around and the resource that touches upon many different concepts and relationships.

One thing we want you to remember before moving on, is that it does seem a bit more repetitive when you are reading about tests and are instructed to write your test in a certain way, which you will be in the rest of this book. When you have learned to write tests and know what you want to write in each test, it becomes much easier and faster. Just keep in mind that we want to teach you as much as possible, so we will try to go through as many tests as possible. Let's begin.

## Laravel Test tools

We'll start by taking a look at our **composer.json** file at the root of our **annas-bookstore** application.

Here, we want to specifically look at the **require-dev** member to see which dependencies are being installed, when you make a new Laravel installation.

At the time of writing this, Laravel 5.8 includes these development dependencies:

```
"require-dev": {  
    "beyondcode/laravel-dump-server": "^1.0",  
    "filp/whoops": "^2.0",  
    "fzaninotto/faker": "^1.4",  
    "mockery/mockery": "^1.0",  
    "nunomaduro/collision": "^3.0",  
    "phpunit/phpunit": "^7.5"
```

```
}
```

Development dependencies mean that these are only needed when you are developing your application and shouldn't be required when your application is actually going into production.

Here, you should especially note **phpunit** which is the package we will be using a lot in this chapter. PHPUnit is a testing framework for PHP applications and is not only used in Laravel, but in many PHP applications around the world. It's not the only testing framework for PHP out there, but it's certainly one of the most widely known.

### *Unit and integration tests*

Laravel uses PHPUnit at the core of all of its test tools, providing a special class that makes sure that the Laravel framework is bootstrapped and ready for each test you run.

This makes it possible to leverage PHPUnit for isolated unit tests, where you test that a single object works as it should. Tests like these are called **unit tests**. It also makes it possible to leverage the integration with Laravel to test that a set of objects work together as they should. These tests are called **integration tests**.

To present a clearer image of the two test types and how they differ, let's imagine that we have a bike and we want to test that the bike works as it should.

We use unit tests to check that each part works as it should in isolation. The bell can make a sound, the wheels are round and can turn, the pedals can rotate, and so on.

In our integration tests, we check that a set of parts work together as they should. For instance, we would test if the pedals and the chain make the wheel turn. Should this test fail, it is not the wheels that are failing — we have already tested them in isolation and proven that they work — but the parts put together are failing. It's the interaction between each part that needs to be done differently.

So a unit test is used to test a single unit and integration tests are used to test that multiple units work together as intended.

### *End-to-end or Feature tests*

Feature tests are, much like the name implies, made to ensure that a certain feature works as it should.

These tests will encompass the entire application from one end to the other, from a request to a response. To be able to do this, you approach the test from the user's perspective in a browser. This could be a test that makes certain that when a form is submitted, the data is saved to the database and thereby ensures that the application works from one end to the other.

If you want to do end-to-end browser testing, Laravel has a tool called Laravel Dusk that fits this job perfectly. Laravel Dusk makes you able to write tests through browser automation. This means that you write the tests as if you were a user visiting your application in the browser. You use language like: "Visit this url and click that button", which makes these tests very easy to understand. A downside though is that the tests are very slow, since you are testing in an actual browser and doing each task sequentially, waiting for page loads and so on.

We will be using feature tests to test our API endpoints, since we want to ensure that the application works as it should when an API endpoint is requested by the user, and that the correct data is returned in the response.

Luckily, when testing APIs, we don't need to use a browser or any other tools, as we can use what is already included by Laravel out of the box.

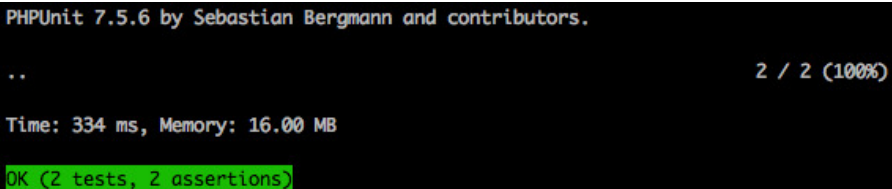
### *Getting up and running*

To set everything up and get started with test-driven development in Laravel, you don't have to do a lot. When installing the Laravel framework, the PHPUnit test framework is installed as well, besides that Laravel includes a PHPUnit configuration file and also a couple of example test files so you know that everything works and know what to do to write a test. If you want to see what the config file contains, you should take a look at the **phpunit.xml** file in the root of our **annas-bookstore** project. This config file ensures that PHPUnit will run with the desired settings and instruct it to look after tests in the tests directory, more specifically tests in **tests/Unit** and **tests/Feature**. It will also set some environmental variables, but we will look into this in a bit.

If you go into your terminal and into the root of our **annas-bookstore** project, try and run the following command:

```
./vendor/bin/phpunit
```

And you should see a result like this:



```
PHPUnit 7.5.6 by Sebastian Bergmann and contributors.  
  
..  
Time: 334 ms, Memory: 16.00 MB  
OK (2 tests, 2 assertions)
```

This tells us that there are two tests and in those two tests there are two



successful assertions, which means that PHPUnit is set up and is running the two example tests. Let's take a look at the **tests/Unit/ExampleTest.php** file:

```
<?php

namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testBasicTest()
    {
        $this->assertTrue(true);
    }
}
```

Besides being a PHP file in the two folders defined in the PHPUnit config file, there are actually a couple of things that define this as a test file and that the test method **testBasicTest** is being tested by PHPUnit.

The first thing to notice is that the class extend **TestCase**. To make PHPUnit able to test a class, the class needs to extend a **TestCase**. Whenever you are using PHPUnit outside of Laravel, PHPUnit provides the **TestCase** class to extend, but in this case, Laravel ships with its own **TestCase** class, which extends the original class from PHPUnit and ensures that the framework is bootstrapped when PHPUnit is testing each test method. So whenever you need a new test class, you must extend Laravel's **TestCase** class.

Another thing to notice is the name of the only **test** method in this class. Right now, the word **test** is used twice, but the first **test** word is actually a necessity since it tells PHPUnit that it should test this test method.

If you go ahead and remove the first **test** from **testBasicTest** so it becomes **BasicTest** and then go into the terminal and run PHPUnit again, you should see something like this:

```
PHPUnit 7.5.6 by Sebastian Bergmann and contributors.

W.                                                                2 / 2 (100%)

Time: 144 ms, Memory: 16.00 MB

There was 1 warning:

1) Warning
No tests found in class "Tests\Unit\ExampleTest".

WARNINGS!
Tests: 2, Assertions: 1, Warnings: 1.
```

Now, we're getting a warning telling us that there are no tests in our test class, which shows us why we need that **test** prefix in our test method names. This naming convention has always annoyed us a bit, but luckily there is another way to tell PHPUnit that it should test a method. You do this by using annotations, like you do when you write docblocks for your regular methods in your classes. So to tell that a method in a test class actually is a test, you do it with a **@test** annotation, like this:

```
<?php

namespace Tests\Unit;

use Tests\TestCase;
```

```

use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @test
     */
    public function BasicTest()
    {
        $this->assertTrue(true);
    }
}

```

Go into the terminal, run PHPUnit again, and you should see the green bar in the bottom again, telling you that there are two tests with two successful assertions.

This is all you have to remember to be able to write tests in PHPUnit and Laravel as well. But before you go ahead and start to write tests, there are some more things we would like to cover, especially environment variables.

## *Environment Variables*

When you install Laravel through the Laravel installer, an **.env** file will be created for you, in which you can define the environment variables to use in your application. Laravel provides some variables right out of the box that are being used to populate your config files with the correct information to connect to the various services your application needs to connect to.

The different environments that Laravel supports are the following:

- **Local** - the environment you use when developing on your local machine.
- **Production** - the environment on your server, when your application is

live and accessible to the public.

- **Testing** – the environment your application will run under, when running tests.

The cool thing about the **.env** file, is that you can swap this file for each environment you are in, and the settings can change without having to edit your config files.

This means that you can have a **.env** file for each of your local development machines, which tells Laravel how to connect to the database on that specific machine, which is almost always different.

This also means that you can have a **.env** file for each server your application is running on, telling Laravel how to connect to the various services from that server.

Last, but not least, it also means that you can have a **.env** file for testing, that can tell Laravel which services to connect to while running tests. One thing we especially want to change is the database Laravel is connecting to. We certainly don't want to fill our database with test data — we much rather want a dedicated or in-memory database for this part. The **.env** file for testing enables us to make a dedicated environment only for testing, where we do not always hit the actual services used by the application when in **local** or **production** state.

An **.env** file for testing doesn't have the same name though. Here, you should use a **.testing** suffix so that the **.env** file for the local environment doesn't get overridden. This will result in an **.env.testing** filename instead.

An important thing to note is that the **.env** file is not a file that you share with others. You keep it to yourself or the machine you are working on, or the server that the application is running on. You do this because the file might contain sensitive information, such as information to connect to your account on a

service.

The **.env.testing** file can be shared if you like, since this most often doesn't contain any sensitive information, but if it does, don't share it.

Let's start making our environment file for testing by copying the existing **.env** file and naming the new file **.env.testing** and then opening the file in your editor.

The first thing we should do is to edit the **APP\_ENV** variable and set this to **testing**.

Then, we should figure out what we want to do with our database. We would recommend creating a separate database dedicated for testing and give that database a name close to the existing database, something like **annas\_bookstore\_testing**. Go back into the **.env.testing** file and change the database settings to the testing database.

Then we always set our **MAIL\_DRIVER** variable to **log**, so that we can see the output of the mails if we need to without having to wait for a service like **Mailtrap** or an actual email to receive anything. Most often, we are using test helpers that catch our emails, but if we forget it, it's nice that nothing is sent to any actual people.

That's it for our environment variables. We now have a dedicated file for our testing environment so that we ensure that all changes happen to our testing database.

### *Running tests automatically*

When you go into the terminal and run PHPUnit, you can see that all of our tests in our testsuite are being tested. You have to run PHPUnit every time you want to run your tests and this can get annoying fast, especially if you

have many tests which will force you to wait until PHPUnit is done. Of course, it's nice to know that all your tests succeed, but when you are working on a specific feature, it's good to focus on the tests for this feature only and get a faster feedback.

Some editors and IDEs can run PHPUnit for you and also tell PHPUnit to run specific tests, but if you don't have any setup like this or are new to testing, we have developed a package for Laravel that can help run your tests automatically. In fact, we will be using this in the rest of this book, so that we are sure that everyone can follow along, no matter which editor or IDE you are using.

The package is called **Laravel Test Watcher** and it can be installed with composer. So let's install this package into our **annas-bookstore** project. Go into your terminal in the root of our project and type the following:

```
composer require wackystudio/laravel-test-watcher
```

If everything went well, you should be able to see that Laravel has auto discovered the package through this message:

```
Discovered Package: wackystudio/laravel-test-watcher
```

If you then call artisan without any commands, you should see that a new command has been added to the list under the **tests** category:

```

storage
storage:link      Create a symbolic link from "public/storage" to "storage/app/public"
tests
tests:watch       Watch tests and source code for changes and run tests automatically
vendor
vendor:publish    Publish any publishable assets from vendor packages
view
view:cache        Compile all of the application's Blade templates
view:clear        Clear all compiled view files

```

This means that the package is installed correctly and we can now begin to use it. Let's run the artisan command to start watching for changes in our application, and then we will take a look at how we can specify which tests to run whenever there's a change.

In the terminal in the root of our project, run the following artisan command:

```
php artisan tests:watch
```

```

Laravel Test Watcher
By Wacky Studio

```

```
No test cases to watch
```

You should then see the screen above, telling you there are no test cases to watch.

Let's jump into the **tests/Unit/ExampleTest.php** again. To make Laravel Test Watcher run this file whenever there's a change, we give it a **@watch** annotation just like we did with the **@test** annotation like this:

```

<?php

namespace Tests\Unit;

use Tests\TestCase;
use Illuminate\Foundation\Testing\RefreshDatabase;

class ExampleTest extends TestCase
{
    /**
     * @test
     * @watch
     */
    public function BasicTest()
    {
        $this->assertTrue(true);
    }
}

```

This should make the screen change in your terminal, where you should now see something like this:



```

Laravel Test Watcher
By Wacky Studio

-----

Tests\Unit\ExampleTest
BasicTest

```

Here, Laravel Test Watcher tells us that it's watching a test inside the ExampleTest file and when the filename is green, it means that the test is successful. This gives us a nice overview of our tests and which file these are in, so we can always see where something might fail.



Speaking of which, let's see what happens if our test fails. Go back to **tests/Unit/ExampleTest.php** and change the value inside the assertion from **true** to **false** like this:

```
/**
 *
 * @test
 * @watch
 */
public function BasicTest()
{
    $this->assertTrue(false);
}
```

As soon as you save this file, Laravel Test Watcher will see the change and test the file, giving you the following output on the screen:



```
Laravel Test Watcher
By Wacky Studio

-----

Tests\Unit\ExampleTest
BasicTest

1 test(s) are failing:

Tests\Unit\ExampleTest::BasicTest
Failed asserting that false is true.
/Users/tgn-wacky/Code/build-an-api-repository/annas-bookstore/tests/Unit/ExampleTest.php:18
```

Our test is now red and the error output is listed below, giving us a nice way to see what has happened and in which test it occurred. You see, by using Laravel

Test Watcher, we don't have to run PHPUnit manually again, and we can keep working in our editor.

On that note, go back to **tests/Unit/ExampleTest.php** and change the value from **false** back to **true** and save the file so that the test passes. Then try to remove the **@watch** annotation. Laravel Test Watch will automatically stop watching the file and remove it from the list and give you the “No test cases to watch” message again.

By using the **@watch** annotation, we are able to pick out the tests we want to test and we get the quick feedback we want, so that we can stay focused and build our applications, making sure all of our tests are successful.

We now know what to do to write a test class and test methods that can be tested. We also know how we can use Laravel Test Watcher to automatically run our tests, so let's get back to coding.

## Revisiting the Authors resource

It's time to get right back into it and continue working on our API. At the moment, we have something that works — we know that from our tests in Postman.

Even though Postman is a great program that makes testing your APIs very easy, it's still a manual process you have to do over and over again. PHPUnit and tests in general can help us automate this process, and the best thing is that whenever the tests are written, they can run over and over again without much effort on our part.

So let's get back to our authors resource and start writing some tests for it.

## *The first test*

For our first test, we will be testing how to fetch a single author. One of the first things you will learn about test-driven development, is that you should write the test first and then do the implementation until the test is passing. Those are rules that are often broken and here we can't really do that, since we already have everything implemented, so we will do it a bit backwards right now. That's not a bad thing though — it will make us able to teach you a bit better. But just to make it clear, you don't have to write tests first, but it is a good idea. We will show you this later in the book.

Let's take a look at our controller to see what it is actually doing when we request a single author. Go into the **app/Http/Controllers/AuthorsController.php** file and take a look at the **show** method:

```
public function show(Author $author)
{
    return new AuthorsResource($author);
}
```

This method takes an **Author** model as an argument, but we don't have to care about this. Laravel will inject this for us using route-model bindings. Through route-model bindings, Laravel will find the corresponding model to the **ID** we give, when making a request to the single author route. The method then returns an **AuthorsResource** which takes the **Author** model.

Just from this information, it seems like we can write a test where we make a request to the route, using an ID of an author and then assert that we get that author back as a correct JSON:API specification resource object.

As we mentioned earlier, we will be writing feature tests for our API, since we will be going through multiple layers in our application in order to work with

the received requests and send back a proper response. In that case, a feature test is appropriate for what we're doing.

Let's go into the **tests/Feature** folder and create a new class. Let's name it **AuthorsTest**, since it will become the test class for all requests involving our authors resource. The class should look like this:

```
<?php

namespace Tests\Feature;

class AuthorsTest
{

}
```

Here, we need to communicate that this is a test class and we do this by extending the TestCase class Laravel provides like this:

```
<?php

namespace Tests\Feature;

Use Tests\TestCase;

class AuthorsTest extends TestCase
{

}
```

Next, we need to define our first test. But before we do that, we have to talk a bit about naming of tests.

Test names are not named like a typical method. Instead, you want to describe the intention of what you are testing. The great thing about this is that you will know what the test is testing from the name only, and it provides a better context, especially when your tests are failing.

There are different conventions for writing the test name — some uses camel casing, others use snake casing. We have always used snake casing, since it makes tests names read more like sentences, which is a bit harder to grasp with camel casing. We will be using this convention in the rest of this book. Whether you follow this advice is completely up to you, but to make it a bit easier for yourself, we recommend that you follow along.

Make a new method and name it: **it\_returns\_an\_author\_as\_a\_resource\_object** like this:

```
<?php

namespace Tests\Feature;

use Tests\TestCase;

class AuthorsTest extends TestCase
{
    /**
     * @test
     * @watch
     */
    public function it_returns_an_author_as_a_resource_object()
    {
    }
}
```

Don't forget to annotate the test so PHPUnit can test it, and while we are at it,

let's watch this method and start Laravel Test Watcher from our terminal like this:

```
php artisan tests:watch
```

Right now, we don't have any assertions so our test will be marked green as passing.

Let's quickly change this by writing the contents of our test.

First, we want to make a request to the route for a single author, with an ID of an author. But how do we get the ID of an author? In the last chapter, we actually touched upon factories, when we created our model. Here, we defined which attributes that should be populated with fake data in our **database/factories/AuthorsFactory.php** file. Great, let's use that like this:

```
/**
 * @test
 * @watch
 */
public function it_returns_an_author_as_a_resource_object()
{
    $author = factory(Author::class)->create();
}
```

The first thing that will happen after you save the test is that it will fail and complain about a missing table or view. This is because we haven't migrated the database and it is completely empty of tables.

Don't worry, Laravel provides us with a **DatabaseMigration** trait we can use to migrate our database for each test in our test class and it's as easy as just adding the trait to our test class like this:

```

<?php

namespace Tests\Feature;

use App\Author;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Tests\TestCase;

class AuthorsTest extends TestCase
{
    use DatabaseMigrations;

    /**
     * @test
     * @watch
     */
    public function it_returns_an_author_as_a_resource_object()
    {
        $author = factory(Author::class)->create();
    }
}

```

You should now see a green test even though it doesn't really test anything. Let's change this by making a request to our show author route, giving it the **ID** of our newly created author.

We do this by leveraging the **TestCase** that Laravel provides and our test class extends. This comes with a lot of convenient methods for testing against our application and one of these methods is for doing requests to our application.

Let's make a request to our application like this:

```

/**
 * @test

```

```

* @watch
*/
public function it_returns_an_author_as_a_resource_object()
{
    $author = factory(Author::class)->create();

    $this->getJson('/api/v1/authors/1');
}

```

You see that on our class, we have access to a **getJson** method that can make requests for us. There are a lot of other convenient methods, but we won't touch on all of them. If you want to learn more about them, you can find the details in the documentation for Laravel.

Our test is still green and passing, but even though we have made a request to our application we are still not testing anything. We need to make assertions about some kind of state to be able to actually test if we get the desired results. But what do we want to make an assertion about here?

We want to know if the request was a success and we can do that by making an assertion about the status code, so let's try that out first.

In this case, Laravel makes it very easy for us: we can simply chain assertions onto our existing request, so let's make an assertion about the status of the request like this:

```

/**
 * @test
 * @watch
 */
public function it_returns_an_author_as_a_resource_object()
{
    $author = factory(Author::class)->create();
}

```



```
$this->getJson('/api/v1/authors/1')->assertStatus(200);
}
```

Now, we are making an assertion and we will also see that our test failed immediately in the terminal. Here, we are getting a **401 Unauthorized** status because we are not authenticated.

No worries, Laravel Passport also has a convenient method we can call to authenticate our requests from our tests, but it does require that we have a user we want to authenticate. We already know how to create an author through a model factory, and conveniently Laravel ships with a default model factory for users, which we can leverage to create a user and then use Laravel Passport to authenticate this user like this:

```
/**
 * @test
 * @watch
 */
public function it_returns_an_author_as_a_resource_object()
{
    $author = factory(Author::class)->create();
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/authors/1')->assertStatus(200);
}
```

And now our test is back to green again.

We need one more assertion, though. We want to test that the API is returning the correct resource object for our author.

To make an assertion about this, we can add another call to the request chain,

asserting that we get the right JSON back like this:

```
/**
 * @test
 * @watch
 */
public function it_returns_an_author_as_a_resource_object()
{
    $author = Author::create([
        'name' => 'John Doe',
    ]);
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/authors/1')
        ->assertStatus(200)
        ->assertJson([
            "data" => [
                "id" => '1',
                "type" => "authors",
                "attributes" => [
                    'name' => $author->name,
                    'created_at' => $author->created_at->toJSON(),
                    'updated_at' => $author->updated_at->toJSON(),
                ]
            ]
        ]);
}
```

To test that we get the right JSON back, we define a PHP array, nesting all the details about the resource object as an associative array. The root is our **data** member, then we have the **id**, **type** and **attributes** members as children which each contains the right data. Notice how we are making a call to **toJSON()** on the **created\_at** and **updated\_at** attributes. We do this to be able to compare the result from the API with a Carbon instance, which Laravel is casting our models **created\_at** and **updated\_at** attributes to.

Our test is green and passing, and everything is looking good. To be able to confirm that the test is testing the result correctly, try to change the **type** attributes value from **authors** to **author** and see if the test fails. If it does, we can move on and be confident about our test working. Don't forget to change the value back to **authors** again.

Great! We have written our first test and we can be a bit more confident about the API performing as it should.

Let's take a close look at what we just wrote, since there's a pattern here we would like for you to notice from the beginning.

The way we see it, a test is divided into 3 parts.

1. In the first part, we set up our world.
2. In the second part, we run the code to be tested.
3. In the last part, we make all of our assertions.

```
public function it_returns_an_author_as_a_resource_object()
{
    // 1. Setup of our world
    $author = Author::create([
        'name' => 'John Doe',
    ]);
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    // 2. Run the code
    $this->getJson('/api/v1/authors/1')

    // 3. Make assertions
    ->assertStatus(200)
    ->assertJson([
```

```

        "data" => [
            "id" => '1',
            "type" => "authors",
            "attributes" => [
                'name' => $author->name,
                'created_at' => $author->created_at->toJSON(),
                'updated_at' => $author->updated_at->toJSON(),
            ]
        ]
    });
}

```

We want you to remember this pattern, especially because we will use it when we start to move a bit faster with our tests. Don't worry, we won't be doing this just yet. Right now, we will continue with the basics.

Let's move on to some of the other requests we need to write tests for. We will stay in the same lane as now, and implement a test for the route for **all authors** next.

### *Testing the all authors route*

For this test, we don't have as much setup to do. We have our test class already and we have the model factories we need. All we need to do is to define another test method. But before we do so, let's take a look at our controller again to see what we're actually testing. Let's head into the **app/Http/Controllers/AuthorsController.php** file and focus on the **index** method:

```

public function index()
{
    $authors = Author::all();
    return new AuthorsCollection($authors);
}

```

```
}
```

Here, we don't need to provide any parameters. We just make a query for all authors and pass those into an **AuthorsCollection** instance.

Looking at this, the thing to test here is that the API endpoint does return a collection of authors and that the collection of authors is a collection of resource objects.

Going back to our **tests/Feature/AuthorsTest.php** file, let's define a new test and name it: **It\_returns\_all\_authors\_as\_a\_collection\_of\_resource\_objects** like this:

```
/**
 * @test
 * @watch
 */
public function
    it_returns_all_authors_as_a_collection_of_resource_objects()
{
}
}
```

From our last test, we know that we need to make an authenticated request in order to test our API, so here we can just copy the code for authentication from our last test like this:

```
/**
 * @test
 * @watch
 */
```

```
public function
    it_returns_all_authors_as_a_collection_of_resource_objects()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
}
```

Besides this, we also know that we need data to exist in our database in order for our API to return them. So we will also need to create some authors in our database. Since we need a collection of authors, it would be nice to have at least 3 authors we can make assertions against.

No problem, we can just leverage our model factory for this like so:

```
/**
 * @test
 * @watch
 */
public function
    it_returns_all_authors_as_a_collection_of_resource_objects()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $authors = factory(Author::class, 3)->create();
}
```

The global factory function can take two arguments: the first is which model we want the factory to create, which is a required argument, and the next is the amount of models we want the factory to create, which is optional. If no number is given, it will just create a single model.

One thing to note is also that the returned object changes, whether you are requesting one or more models to be created. If you request a single model to

be created, you get that model back. If you request multiple models, then you will get a collection of those models back.

Next, we need to make the actual request. We already know how to do that and while we are at it, let's also make an assertion that we get a status code of **200** back:

```
/**
 * @test
 * @watch
 */
public function
    it_returns_all_authors_as_a_collection_of_resource_objects()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $authors = factory(Author::class, 3)->create();

    $this->get('/api/v1/authors')->assertStatus(200);
}
```

You should see all green and passing tests in the terminal.

But this is not enough for us to be confident about the API endpoint being well tested, so let's make some assertions about what data we expect to get back.

We know what to expect from both our last test, but also from the chapter about the JSON:API specification, which is a collection of resource objects, instead of a single resource object. We can copy most of what we have already written in the previous test, and then make sure we are referencing both the right ID and the right object in our authors collection that we got from the factory:

```

/**
 * @test
 * @watch
 */
public function
    it_returns_all_authors_as_a_collection_of_resource_objects()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $authors = factory(Author::class, 3)->create();

    $this->get('/api/v1/authors')->assertStatus(200)->assertJson([
        "data" => [
            [
                "id" => '1',
                "type" => "authors",
                "attributes" => [
                    'name' => $authors[0]->name,
                    'created_at' => $authors[0]->created_at->toJSON
                        (),
                    'updated_at' => $authors[0]->updated_at->toJSON
                        (),
                ]
            ],
            [
                "id" => '2',
                "type" => "authors",
                "attributes" => [
                    'name' => $authors[1]->name,
                    'created_at' => $authors[1]->created_at->toJSON
                        (),
                    'updated_at' => $authors[1]->updated_at->toJSON
                        (),
                ]
            ],
            [
                "id" => '3',
                "type" => "authors",
                "attributes" => [
                    'name' => $authors[2]->name,

```



```

        'created_at' => $authors[2]->created_at->toJSON
        ( ),
        'updated_at' => $authors[2]->updated_at->toJSON
        ( ),
    ]
],
]
]);
}

```

We assert that the structure is correct and that we get the right data for each author. You should still see a green and passing test in the terminal. Try changing the **ID** on the third resource object to the value **4** and the test should fail, confirming that everything is being tested correctly.

Now that we know how to test when we want to fetch data, let's look at how we can store data and make a test for the **Create Author** route.

Before we move on, don't forget to remove the **@watch** annotation from your test, since we don't need to test this test case any longer.

### *Testing the create author route*

For this test, we have to do things a bit differently. We don't need that much setup, since we are providing the data to the API. Like we did in the other tests, let's first visit our **app/Http/Controllers/AuthorsController.php** file and focus on the **store** method:

```

public function store(CreateAuthorRequest $request)
{
    $author = Author::create([
        'name' => $request->input('data.attributes.name'),
    ]);
}

```

```

return (new AuthorsResource($author))
    ->response()
    ->header('Location', route('authors.show', [
        'author' => $author,
    ]));
}

```

This method takes an argument, which is our own **CreateAuthorRequest** that we use to validate the request. Then we create a new author, using the name attribute from the resource being sent to the server and afterward return a new instance of an **AuthorsResource** with the author model. We add the **location** header to adhere to the JSON:API specification, giving it a value of the new author's link.

To test this, we actually need two tests: one that tests that the validation works as intended, and one that tests that the actual create functionality works as intended. We will be looking at validation later, so let's just make sure that the create functionality works as intended.

In our test, we will make a request with a proper resource object and then assert that we get the correct status code and resource object back. We will also assert that the **location** header is a part of the response as well. The most important part and the thing that actually makes us able to verify if the create functionality works, is the assertion we can make against the database, to see if a new author has been created. Let's create a new test in our **tests/Feature/AuthorsTest.php** and name it: **it\_can\_create\_an\_author\_from\_a\_resource\_object** like this:

```

/**
 * @test
 * @watch
 */

```

```
public function it_can_create_an_author_from_a_resource_object()
{

}
```

This time, we don't have to setup like in the previous tests, but we do need to have a user to make authenticated requests. Let's copy that from one of our previous tests to this test like this:

```
/**
 * @test
 * @watch
 */
public function it_can_create_an_author_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
}
```

Next, we need to make a POST request to our API with the necessary data to create an author. From the chapter about the JSON:API specification, we know that we must send a resource object and we know exactly how that should look. We need a **data** member in the top level with the value of our resource object. We want the backend to create the **ID**, so we won't have to add that. We only need to provide the **type** and the **attributes**, which in this case is only the **name** of the author. Since we are making a request to create an author, we need to make sure our status code is a **201 Created**, so we might as well add this right away so that we can see if our test fails or not:

```
/**
 * @test
```

```

* @watch
*/
public function it_can_create_an_author_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/authors', [
        'data' => [
            'type' => 'authors',
            'attributes' => [
                'name' => 'John Doe',
            ]
        ]
    ]->assertStatus(201);
}

```

Right now, our test should pass and be green, but to be sure, let's change the status code from **201** to **200** and see if it fails. You should immediately see a failing test with the reason "Expected status code 200 but received 201" in the terminal. This means that it works, so let's change the status code back to **201** again and add the last assertions. We need to assert that we get the created resource back as a resource object. We have already written this assertion before when creating the tests for a single resource, so let's steal the assertion from there and copy it into this test. Change the value of the name to **John Doe** and for **created\_at** and **updated\_at** we can use the **now()** helper function to get a fresh Carbon instance. We can convert this to JSON using the **toJSON()** method, just like on the Carbon instances you get from the date attributes of a model. This will make the test fail, since our manually created Carbon instances will include milliseconds, which Carbon instances created by Laravel Eloquent will not. To fix this issue, we will need to set the milliseconds on the Carbon instances to **0** and the assertion will work:

```

/**
 * @test
 * @watch
 */
public function it_can_create_an_author_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/authors', [
        'data' => [
            'type' => 'authors',
            'attributes' => [
                'name' => 'John Doe',
            ]
        ]
    ])
    ->assertStatus(201)
    ->assertJson([
        "data" => [
            "id" => '1',
            "type" => "authors",
            "attributes" => [
                'name' => 'John Doe',
                'created_at' => now()->setMilliseconds(0)->
                    toJSON(),
                'updated_at' => now()->setMilliseconds(0)->
                    toJSON(),
            ]
        ]
    ]);
}

```

Our test should still be green, but we are missing one thing to be sure that we have tested everything, namely the **Location** header. To test this, we can use the **assertHeader()** method, which can also be chained onto our existing chain like this:

```

/**
 * @test
 * @watch
 */
public function it_can_create_an_author_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/authors', [
        'data' => [
            'type' => 'authors',
            'attributes' => [
                'name' => 'John Doe',
            ]
        ]
    ])
        ->assertStatus(201)
        ->assertJson([
            "data" => [
                "id" => '1',
                "type" => "authors",
                "attributes" => [
                    'name' => 'John Doe',
                    'created_at' => now()->setMilliseconds(0)->
                        toJSON(),
                    'updated_at' => now() ->setMilliseconds(0)->
                        toJSON(),
                ]
            ]
        ])
        ->assertHeader('Location', url('/api/v1/authors/1'));
}

```

Lastly, we need to ensure that the data has been saved to the database, so we know that the create functionality is actually working. For this, we will use the **assertDatabaseHas()** method. This, unfortunately, cannot be added to our chain, so we will have to end the chain here and make a new call to **\$this** to call the **assertDatabaseHas()** method. This method takes two arguments, the database and an array with the columns and values you want to assert a row

in the database has like this:

```
/**
 * @test
 * @watch
 */
public function it_can_create_an_author_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/authors', [
        'data' => [
            'type' => 'authors',
            'attributes' => [
                'name' => 'John Doe',
            ]
        ]
    ])
        ->assertStatus(201)
        ->assertJson([
            "data" => [
                "id" => '1',
                "type" => "authors",
                "attributes" => [
                    'name' => 'John Doe',
                    'created_at' => now()->setMilliseconds(0)->
                        toJSON(),
                    'updated_at' => now() ->setMilliseconds(0)->
                        toJSON(),
                ]
            ]
        ])
        ->assertHeader('Location', url('/api/v1/authors/1'));

    $this->assertDatabaseHas('authors', [
        'id' => 1,
        'name' => 'John Doe'
    ]);
}
```

Notice that we are referencing the **ID** as an integer and we no longer follow the conventions of the JSON:API specification, since we are referencing columns of our database.

We have now implemented the test for creating an author, and if you have done everything right, you should see a green and passing test in the terminal. Before we move on, don't forget to remove the **@watch** annotation from your test, since we don't need to test this test case any longer.

Let's move on to the test for updating an author.

### *Testing the update author route*

By now, you should be a bit more familiar with the drill. Let's go to the **app/Http/Controllers/AuthorsController.php** file and look at the **update** method:

```
public function update(UpdateAuthorRequest $request, Author $author
    )
{
    $author->update($request->input('data.attributes'));
    return new AuthorsResource($author);
}
```

This method takes an argument, which is our own **UpdateAuthorRequest** that we use to validate the request. Then, it takes an **Author** model as an argument. Again, we don't have to care about this. Laravel will inject this for us, using route-model bindings. The author model is then updated using the resource object sent in the request and then returns the updated model as a resource. Remember that we need to return a resource if anything has been changed by the server upon an update. Laravel will change the **updated\_at** attribute and therefore we have to return the resource.



Now, in this test we have to do a bit more. We first need to set up our world, where we, besides making sure we can make authenticated request, need to make sure an author already exists in the system. Before we can do that, we need to create our test, so make a new test method in the `tests/Feature/AuthorsTest.php` file, name it: **`it_can_update_an_author_from_a_resource_object`** and make sure we can make authenticated request and that an author exists like this:

```
/**
 * @test
 * @watch
 */
public function it_can_update_an_author_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();
}
```

Then we should make a request to the API with the data we want to update. We know from the JSON:API specification that we need to make a PATCH request, and that we need to send the data as a resource object like this:

```
/**
 * @test
 * @watch
 */
public function it_can_update_an_author_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();
}
```

```

$this->patchJson('/api/v1/authors/1', [
    'data' => [
        'id' => '1',
        'type' => 'authors',
        'attributes' => [
            'name' => 'Jane Doe',
        ]
    ]
])->assertStatus(200);
}

```

The test should be green and passing at the moment, but we are far from done. Just like the test for creating an author, we need to make an assertion about the resource sent back in the response, as well as making an assertion about the changed data in the database, so that we know that the update functionality works as it should:

```

/**
 * @test
 * @watch
 */
public function it_can_update_an_author_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();

    $this->patchJson('/api/v1/authors/1', [
        'data' => [
            'id' => '1',
            'type' => 'authors',
            'attributes' => [
                'name' => 'Jane Doe',
            ]
        ]
    ]->assertStatus(200)->assertJson([

```

```

        'data' => [
            'id' => '1',
            'type' => 'authors',
            'attributes' => [
                'name' => 'Jane Doe',
                'created_at' => now()->setMilliseconds(0)->toJSON(),
                'updated_at' => now() ->setMilliseconds(0)->toJSON()
            ],
        ],
    ];

    $this->assertDatabaseHas('authors', [
        'id' => 1,
        'name' => 'Jane Doe',
    ]);
}

```

The test should still be green and passing. There is something here that is a bit misleading, namely the assertions about the **created\_at** and **updated\_at** timestamps. We are creating a model and updating it afterward, so the timestamps should not be alike. In theory, the **updated\_at** timestamp should be completely different from the **created\_at** because we have modified the model since its creation. But since tests happen so fast and we don't measure in milliseconds, this scenario can happen. If you want to ensure that there is a difference, we can add a call to the global **sleep()** function right after we have created the author. To make sure that we have the timestamp from when the author was created, we should save a Carbon instance from that moment just before sleeping, and then use this variable when asserting against the **created\_at** timestamp like this:

```

/**
 * @test

```

```

* @watch
*/
public function it_can_update_an_author_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();

    $creationTimestamp = now();
    sleep(1);

    $this->patchJson('/api/v1/authors/1', [
        'data' => [
            'id' => '1',
            'type' => 'authors',
            'attributes' => [
                'name' => 'Jane Doe',
            ]
        ]
    ]->assertStatus(200)->assertJson([
        'data' => [
            'id' => '1',
            'type' => 'authors',
            'attributes' => [
                'name' => 'Jane Doe',
                'created_at' => $creationTimestamp->setMilliseconds(
                    0)->toJSON(),
                'updated_at' => now()->setMilliseconds(0)->toJSON(),
            ]
        ]
    ]);

    $this->assertDatabaseHas('authors', [
        'id' => 1,
        'name' => 'Jane Doe',
    ]);
}

```

This will achieve the difference between **created\_at** and **updated\_at** so you know that this part works. However, it is not really testing our code, it's

more a test of the intentions of Laravel's models and that it will update a models **updated\_at** timestamp whenever you save changes to the database. This shouldn't be something we should test, but something Laravel covers in the tests for the framework and be something that we can rely on if it's been documented. If you like the test to be explicit about the difference in timestamps, you can keep it in your code. We will revert back to using the same timestamp.

Before we move on, don't forget to remove the **@watch** annotation from your test, since we don't need to test this test case any longer.

### *Testing the delete author route*

We are almost there: we almost have a test for each of our routes. The next one is an easy one — we only need an author we can delete. Taking a look at the **app/Http/Controllers/AuthorsController.php** file and **destroy** method, we can see that it takes an Author model as the only argument. Once more, Laravel will help us here with route-model bindings. Next, we delete the model and return an empty response with the **204 No Content** status code.

For this test, we can copy the entire setup from the test for the single author route. Then, we need to make a **delete** request to the API and assert that we get the **204 No Content** status code back. We should also make an assertion to the database to make sure the deletion functionality works as it should. Create a new test method and name it: **it\_can\_delete\_an\_author\_through\_a\_delete\_request** and add this code:

```
/**
 * @test
 * @watch
 */
public function it_can_delete_an_author_through_a_delete_request()
```

```

{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();

    $this->delete('/api/v1/authors/1', [], [
        'Accept' => 'application/vnd.api+json',
        'Content-Type' => 'application/vnd.api+json',
    ])->assertStatus(204);

    $this->assertDatabaseMissing('authors', [
        'id' => 1,
        'name' => $author->name,
    ]);
}

```

The test should be green and passing, so to test that it works, try changing the status code from **204** to **200** and the test should fail. Change it back and remove the **@watch** annotation.

### *Testing validation*

So far, we have tested the individual routes in a very naive way, assuming that our consumers will get everything right each time. In the real world, this is far from the truth and we should be able to handle that. Again, we are in a scenario where it is actually implemented so we don't need to cover why we validate, but focus more on how to test that our validation actually works as it should. We don't need to test validation for each route since it is only when creating and updating resources we will receive input from our consumers and therefore need to ensure that we validate these data.

Let's take care of the create author route first. To be able to do that, let's first open the **app/Http/Requests/CreateAuthorRequest.php** file and look at the **rules** method:

```

public function rules()
{
    return [
        'data' => 'required|array',
        'data.type' => 'required|in:authors',
        'data.attributes' => 'required|array',
        'data.attributes.name' => 'required|string',
    ];
}

```

Here, we validate for multiple things, and first we validate that we receive an object in the **data** member.

We then validate the **type** of the resource object, both that it is actually a part of the resource object, but also that the value is **authors**. Let's test this first. Make a new test method directly beneath the **it\_can\_create\_an\_author\_from\_a\_resource\_object** test and name it: **it\_validates\_that\_the\_type\_member\_is\_given\_when\_creating\_an\_author** and copy the content from **it\_can\_create\_an\_author\_from\_a\_resource\_object** test into the newly created test. Remove the **authors** value from the **type** member, change the asserted status to be **422** instead of **201**, and remove both the **assertJson**, **assertHeader** assertions. Rename the **assertDatabaseHas** method to **assertDatabaseMissing** and the whole thing should result in something like this:

```

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_type_member_is_given_when_creating_an_author
    ()
{
    $user = factory(User::class)->create();
}

```

```

Passport::actingAs($user);

$this->postJson('/api/v1/authors', [
    'data' => [
        'type' => '',
        'attributes' => [
            'name' => 'John Doe',
        ]
    ]
])->assertStatus(422);

$this->assertDatabaseMissing('authors', [
    'id' => 1,
    'name' => 'John Doe'
]);
}

```

The test should be passing and green, but we are not done yet. We haven't done any assertion against what is being returned from our API. Here, we would like a correct error form that adheres to the conventions of the JSON:API specification, so let's tackle that next.

### *Handling validation errors*

Handling validation errors can be done in multiple ways. We can choose to return the JSON response from our **CreateAuthorRequest** class and have everything contained, so we know where to find both the validation rules and what is return when a validation is failing. The downside to this is that we then would need to copy the error handling for each new request class we will make. We could mitigate this by extracting the error handling to its own class and let that class extend **FormRequest** like our **CreateAuthorRequest** is doing now, and then let **CreateAuthorRequest** our new error handling class. Then it would be easy to reuse the error handling functionality again and again.

There's a third way that we think is a little better. Laravel comes with an



error and exception handler right out of the box. It makes sure that your exceptions are reported to the right sources, which could be your log file, an error notification service, and so on. It is also the handler that makes sure that your errors or exceptions are rendered in an HTTP response to give feedback to the user. This rendering is happening whether we are requesting JSON or HTML.

The handler has already a dedicated method for handling validation errors, so let's override this method so our new error response will fit right into the existing flow and work throughout the application. This will also make it possible for us to show the benefits of test driven development where no source code has been written yet, meaning that we will finish our tests assertions of what we will receive and then build the validation error response till the test pass.

Let's jump into **tests/Feature/AuthorsTest.php** and add a **assertJson** to the end of the existing **postJson** chain after our **assertStatus** assertion. Remember from the chapter about the JSON:API specification, that we need a **errors** top level member for a collection of error objects. We then recommended that your error objects would consist of a title member with a **title** of the error, which in this case would be **Validation Error**, a **details** member describing the error. Here, we will use the existing error messaging from Laravel, since it's quite good. Lastly, we need a **source** object with a JSON pointer, pointing to which attribute the error is about. Laravel's validator object is actually doing this work for us already, but it uses dot notation instead of slashes, so we will need to replace the dots with slashes. With all this knowledge, we can write our assertion. Laravel actually has some assertions in terms of JSON validation errors, but now that we have changed the response document from Laravel usually works with, we need to make regular JSON assertions like this:

```

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_type_member_is_given_when_creating_an_author
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/authors', [
        'data' => [
            'type' => '',
            'attributes' => [
                'name' => 'John Doe',
            ]
        ]
    ])
        ->assertStatus(422)
        ->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The data.type field is required',
                    '.',
                    'source' => [
                        'pointer' => '/data/type',
                    ]
                ]
            ]
        ]);

    $this->assertDatabaseMissing('authors', [
        'id' => 1,
        'name' => 'John Doe'
    ]);
}

```

For the first time, our test is failing telling us it is unable to find the JSON in

the response. Let's go to the source code and write the code that will make our test pass.

Go into the **app/Exceptions/Handler.php** file and let's override the **invalidJson** method, which is actually on the parent **ExceptionHandler** class like this:

```
protected function invalidJson($request, ValidationException
    $exception)
{
}
}
```

Now, the feedback from our failing test has changed from being unable to find the right JSON to receiving a 500 response. Let's make sure that we are actually returning a JSON response like this:

```
protected function invalidJson($request, ValidationException
    $exception)
{
    return response()->json([

    ], $exception->status);
}
```

And we are back to our test failing, since it can't find the right JSON. From the JSON:API specification, we know that we want a collection of errors and the **ValidationException** actually contains an error array we could use, but unfortunately this array only contains the message we need for our **detail** member. We need to loop through each validation error and get both the attribute that failed as well as the message. Luckily, the **ValidationException** also contains the validation object that made the validation and from this

we can get an array of errors with both the failing attributes as well as the messages. We then need to transform this array into an array of errors. There are many ways this can be done, but let's use one of the best features of Laravel, namely the Collections. Laravel's Collections is like an array on steroids and makes it possible for us to easily map the existing array into an array of error objects that adhere to the JSON:API specification like this:

```
protected function invalidJson($request, ValidationException
    $exception)
{
    $errors = ( new Collection($exception->validator->errors()) )
        ->map(function ($error, $key) {
            return [
                'title'    => 'Validation Error',
                'details' => $error[0],
                'source' => [
                    'pointer' => '/' . str_replace('.', '/', $key),
                ]
            ];
        })
        ->values();

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

Here, we instantiate a new Collection and give it the validator's array of errors. Through the **map** method, we go through each entry in the array and map it into the desired error object that adheres to the JSON:API specification.

Lastly, we call the **values** method on the collection, which will reset the keys to consecutive integers. The main reason for this is that the error array from the validator is an associative array and when mapping this, we will keep the associative keys, which will make PHP's JSON encoder encode this array as a

JSON object to keep the keys. Since we need to have an array, calling the **values** method will remove the associative keys and make the items in the array have keys of consecutive integers.

We then return this in our JSON response and our test will pass.

This means that we have an implementation of validation errors that adheres to the JSON:API specification. To be sure that it covers everything we need and to be sure that our validation works as intended. Let's continue with our testing of our validation.

### *Continuing validation testing*

Right now, we have tested that the validation rule, which requires a **type** member of our request object being present, works as intended. If we go back into the **app/Http/Requests/CreateAuthorRequest.php** file, we can see that there are more rules to test.

```
public function rules()
{
    return [
        'data' => 'required|array',
        'data.type' => 'required|in:authors',
        'data.attributes' => 'required|array',
        'data.attributes.name' => 'required|string',
    ];
}
```

Looking at the type member again, we can see that we also need to test that the validation should validate if the value authors is given for the type member. We can do this pretty easily — let's start by creating a new test right under the previous test and name it: **it\_validates\_that\_the\_type\_member\_has\_the\_value\_of\_authors\_when\_creating\_an\_author**. Be sure to

annotate test with both the **@test** and **@watch** annotations and be sure to remove the **@watch** annotation from the previous test. Copy the contents from the previous test, and change the test to the following:

```
/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_type_member_has_the_value_of_authors_when_creating_an_
    (
    {
        $user = factory(User::class)->create();
        Passport::actingAs($user);

        $this->postJson('/api/v1/authors', [
            'data' => [
                'type' => 'author',
                'attributes' => [
                    'name' => 'John Doe',
                ]
            ]
        ])
            ->assertStatus(422)
            ->assertJson([
                'errors' => [
                    [
                        'title' => 'Validation Error',
                        'details' => 'The selected data.type is invalid',
                        '.',
                        'source' => [
                            'pointer' => '/data/type',
                        ]
                    ]
                ]
            ]);

        $this->assertDatabaseMissing('authors', [
```

```

        'id' => 1,
        'name' => 'John Doe'
    });
}

```

We will send a request with the **type author** instead of **authors** which should make the validation fail, then assert we get the validation error in the response with the correct detail message. This should give you a green test, because of our previous implementation in the exception handler class.

Moving on to the attributes member, we can reuse a lot of what we have already. Copy the previous test and rename it to: **it\_validates\_that\_the\_attributes\_member\_has\_been\_given\_when\_creating\_an\_author** and change the following:

```

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_attributes_member_has_been_given_when_creating_an_author
    ()
    {
        $user = factory(User::class)->create();
        Passport::actingAs($user);

        $this->postJson('/api/v1/authors', [
            'data' => [
                'type' => 'authors',
            ]
        ]->assertStatus(422)
            ->assertJson([
                'errors' => [
                    [
                        'title' => 'Validation Error',

```

```

        'details' => 'The data.attributes field is
                        required.',
        'source' => [
            'pointer' => '/data/attributes',
        ]
    ]
}

$this->assertDatabaseMissing('authors', [
    'id' => 1,
    'name' => 'John Doe'
]);
}

```

In the request, we leave out the **attributes** member and afterward we assert that the validation is catching this, telling us that the **attributes** member is required. You should see a green and passing test.

Because the **attributes** member will contain the attributes of our resources, this must be an object, so let's test that the validation will catch this in a new test right beneath the previous one:

```

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_attributes_member_is_an_object_given_when_creating_an_
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/authors', [

```



```

        'data' => [
            'type' => 'authors',
            'attributes' => 'not an object',
        ]
    ])->assertStatus(422)
        ->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The data.attributes must be an
                        array.',
                    'source' => [
                        'pointer' => '/data/attributes',
                    ]
                ]
            ]
        ]);
    $this->assertDatabaseMissing('authors', [
        'id' => 1,
        'name' => 'John Doe'
    ]);
}

```

Before you get confused about the validation message, remember that JSON objects will be decoded as either objects or associative arrays in PHP. Laravel will decode these as associative arrays.

We are almost done. The last things we need to test is if the validation will catch that a name attribute is required, and that it must be a string. Again, we can reuse what we have in the previous test, so create a new test beneath the previous one for testing that a **name** attribute is given with this code:

```

/**
 * @test

```

```

* @watch
*/
public function
    it_validates_that_a_name_attribute_is_given_when_creating_an_author
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/authors', [
        'data' => [
            'type' => 'authors',
            'attributes' => [
                'name' => '',
            ],
        ],
    ])
        ->assertStatus(422)
        ->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The data.attributes.name field is
                        required.',
                    'source' => [
                        'pointer' => '/data/attributes/name',
                    ]
                ]
            ]
        ]);

    $this->assertDatabaseMissing('authors', [
        'id' => 1,
        'name' => 'John Doe'
    ]);
}

```

We leave the value of the **name** member in the **attributes** object empty and assert that the validation catches this and responds with a validation error.

You should be seeing a green test by now, telling you that it's implemented correctly.

Moving on to the last test, let's copy the previous test again and create a new one beneath for testing that a **name** attribute is a string like this:

```
/**
 * @test
 * @watch
 */
public function
    it_validates_that_a_name_attribute_is_a_string_when_creating_an_author
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/authors', [
        'data' => [
            'type' => 'authors',
            'attributes' => [
                'name' => 47,
            ],
        ],
    ])
    ->assertStatus(422)
    ->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The data.attributes.name must be
                    a string.',
                'source' => [
                    'pointer' => '/data/attributes/name',
                ]
            ]
        ]
    ])
}
```

```

    });

    $this->assertDatabaseMissing('authors', [
        'id' => 1,
        'name' => 'John Doe'
    ]);
}

```

In this test, we give an integer value in the **name** member instead of a string and assert that the validation catches this. If you have done it correctly, you should see a green test telling us that the validation rule has been implemented correctly.

This concludes testing all the rules on the **app/Http/Requests/CreateAuthorRequest.php** file, we then have to implement the test for the **app/Http/Requests/UpdateAuthorRequest.php**.

### *Testing UpdateAuthorRequest*

If we take a look at the **app/Http/Requests/UpdateAuthorRequest.php** files rule method, we see that it's not that different from our **app/Http/Requests/CreateAuthorRequest.php** file, which actually means we can reuse a lot of our test code again:

```

/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        'data' => 'required|array',
    ];
}

```

```

        'data.id' => 'required|string',
        'data.type' => 'required|in:authors',
        'data.attributes' => 'required|array',
    ];
}

```

Let's begin by testing the validation for the **ID** member and, just like before when we implemented the tests for **app/Http/Requests/CreateAuthorRequest.php**, let's start out by copying the **it\_can\_update\_an\_author\_from\_a\_resource\_object** test and paste it in right beneath. Next, rename the test to: **it\_validates\_that\_an\_id\_member\_is\_given\_when\_updating\_an\_author** and replace the code with this:

```

/**
 * @test
 * @watch
 */
public function
    it_validates_that_an_id_member_is_given_when_updating_an_author
    ()
    {
        $user = factory(User::class)->create();
        Passport::actingAs($user);
        $author = factory(Author::class)->create();

        $this->patchJson('/api/v1/authors/1', [
            'data' => [
                'type' => 'authors',
                'attributes' => [
                    'name' => 'Jane Doe',
                ]
            ]
        ])
        ->assertStatus(422)
        ->assertJson([

```

```

        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The data.id field is required.',
                'source' => [
                    'pointer' => '/data/id',
                ]
            ]
        ]
    });

    $this->assertDatabaseHas('authors', [
        'id' => 1,
        'name' => $author->name,
    ]);
}

```

We have removed the **ID** member in the request and then we assert that the validation will catch this and respond with a validation error, telling us that the **ID** member is required. There is nothing new here: the assertions follow the patterns from the previous tests. The difference is that we are doing PATCH request instead of a POST request, which means we will hit the **app/Http/Requests/UpdateAuthorRequest.php** rules instead, making it possible to test these rules. We will be using the same approach to testing these rules, which by now you should be familiar with it. So for the rest of the rules in **app/Http/Requests/UpdateAuthorRequest.php**, we will let you work on your own without explaining everything. We will give you the code for the rest of the tests, but we highly recommend that you start out by trying to work on your own. Take a look at the existing tests if you get stuck and only as a last resort look at the following code:

```

/**
 * @test

```

```

*/
public function
    it_validates_that_an_id_member_is_a_string_when Updating_an_author
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();

    $this->patchJson('/api/v1/authors/1', [
        'data' => [
            'id' => 1,
            'type' => 'authors',
            'attributes' => [
                'name' => 'Jane Doe',
            ]
        ]
    ])
        ->assertStatus(422)
        ->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The data.id must be a string.',
                    'source' => [
                        'pointer' => '/data/id',
                    ]
                ]
            ]
        ]
    );

    $this->assertDatabaseHas('authors', [
        'id' => 1,
        'name' => $author->name,
    ]);
}

/**
 * @test
 */

```

```

public function
    it_validates_that_the_type_member_is_given_when Updating_an_author
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();

    $this->patchJson('/api/v1/authors/1', [
        'data' => [
            'id' => '1',
            'type' => '',
            'attributes' => [
                'name' => 'Jane Doe',
            ]
        ]
    ])
        ->assertStatus(422)
        ->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The data.type field is required',
                    '.',
                    'source' => [
                        'pointer' => '/data/type',
                    ]
                ]
            ]
        ]);

    $this->assertDatabaseHas('authors', [
        'id' => 1,
        'name' => $author->name,
    ]);
}

/**
 * @test
 */

```



```

public function
    it_validates_that_the_type_member_has_the_value_of_authors_when_updating_a
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();

    $this->patchJson('/api/v1/authors/1', [
        'data' => [
            'id' => '1',
            'type' => 'author',
            'attributes' => [
                'name' => 'Jane Doe',
            ]
        ]
    ])
        ->assertStatus(422)
        ->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The selected data.type is invalid
                    .',
                    'source' => [
                        'pointer' => '/data/type',
                    ]
                ]
            ]
        ]);

    $this->assertDatabaseHas('authors', [
        'id' => 1,
        'name' => $author->name,
    ]);
}

/**
 * @test
 */

```

```

public function
    it_validates_that_the_attributes_member_has_been_given_when_updating_an_auth
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();

    $this->patchJson('/api/v1/authors/1', [
        'data' => [
            'id' => '1',
            'type' => 'authors',
        ]
    ])
        ->assertStatus(422)
        ->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The data.attributes field is
                        required.',
                    'source' => [
                        'pointer' => '/data/attributes',
                    ]
                ]
            ]
        ]);

    $this->assertDatabaseHas('authors', [
        'id' => 1,
        'name' => $author->name,
    ]);
}

/**
 * @test
 */
public function
    it_validates_that_the_attributes_member_is_an_object_given_when_updating_an_

```

```

    ()
  {
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();

    $this->patchJson('/api/v1/authors/1', [
      'data' => [
        'id' => '1',
        'type' => 'authors',
        'attributes' => 'not an object',
      ]
    ])
      ->assertStatus(422)
      ->assertJson([
        'errors' => [
          [
            'title' => 'Validation Error',
            'details' => 'The data.attributes must be an
              array.',
            'source' => [
              'pointer' => '/data/attributes',
            ]
          ]
        ]
      ]);

    $this->assertDatabaseHas('authors', [
      'id' => 1,
      'name' => $author->name,
    ]);
  }

  /**
   * @test
   * @watch
   */
  public function
    it_validates_that_a_name_attribute_is_a_string_when Updating_an_author
    ()

```

```

{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $author = factory(Author::class)->create();

    $this->patchJson('/api/v1/authors/1', [
        'data' => [
            'id' => '1',
            'type' => 'authors',
            'attributes' => [
                'name' => 47,
            ],
        ],
    ])
    ->assertStatus(422)
    ->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The data.attributes.name must be
                    a string.',
                'source' => [
                    'pointer' => '/data/attributes/name',
                ]
            ]
        ]
    ]);

    $this->assertDatabaseHas('authors', [
        'id' => 1,
        'name' => $author->name,
    ]);
}

```

Note that attributes are optional when updating a resource — that is the reason why we’re not testing that the name is given when updating an author.

By now, you should be more familiar with testing, test naming and the structure of tests.

Up until this point, our tests have followed the same patterns which we hope you are noticing. We have both written tests for source code that was already implemented and written tests for source code that had not been written yet. From here on out, we will write our tests first and then do the implementation until these tests pass.

## Adhering to the JSON:API specification

So far, we have implemented our resource and validation error in line with the conventions of the JSON:API specification, but there are some things we haven't implemented yet. Firstly, we haven't done anything to ensure the correct content negotiation. This ought to be the primary thing to do, since it's the first convention mentioned in the specification, but we have been holding off to get the ball rolling. Now that we have started implementing our API and we have even started on test-driven development, we can also tackle the implementation of the content negotiation fairly easily.

We haven't implemented the features for fetching our resource, and by features we mean the ability to sort and paginate when fetching our resources.

Last, but not least, we haven't implemented error handling for general exceptions thrown by the application, which we should also be tackling. Let's start out with content negotiation.

### *The correct content negotiation*

If you remember back to the chapter about the JSON:API specification, we talked about how the client must send requests with the following header:

```
Accept: application/vnd.api+json
```

The client **must** also use the following header when sending data to the server:

```
Content-Type: application/vnd.api+json
```

And how the server should deliver its responses with the following header to tell the client that the data send lives up to the protocols of the JSON:API specification:

```
Content-Type: application/vnd.api+json
```

We can't control what the client is sending us, but we can reject the request with a response **406 Not Acceptable** status code, if the **Accept** header is not correct, as stated in the JSON:API specification. We will also ensure that we respond with a **415 Unsupported Media Type** if we don't receive the correct **Content-Type header** and lastly ensure that we send all our responses from the server with the correct **Content-Type header**.

In this case, it's best to use a middleware, so we can intercept requests to our API and check if they live up to our conventions, but also so we can assure that any responses leaving our application live up to the server's responsibilities.

Let's start by creating a test for our middleware. This time we will create a unit test — we don't need to test this out on our API through a feature test yet, but we will get back to this.

Create a new **tests/Unit/Middleware/EnsureCorrectAPIHeadersTest.php** file and make sure to both put it in the right namespace and also extend the Laravel TestCase class like this:

```
<?php

namespace Tests\Unit\Middleware;

use Tests\TestCase;

class EnsureCorrectAPIHeadersTest extends TestCase
{

}
```

Let's begin by tackling the first convention and checking for the Accept header, so create a new test method and name it: **it\_aborts\_request\_if\_accept\_header\_does\_not\_adhere\_to\_json\_api\_spec** like this:

```
/**
 * @test
 * @watch
 */
public function
    it_aborts_request_if_accept_header_does_not_adhere_to_json_api_spec
    (){

}
```

To setup the world of our test, we need an instance of our middleware to be able to test against it. Let's create the middleware through artisan, so get into your terminal and run the following command:

```
php artisan make:middleware EnsureCorrectAPIHeaders
```

Let's open the newly created **app/Http/Middleware/EnsureCorrectAPIHead-**

**ers.php** middleware class to see what it contains:

```
<?php

namespace App\Http\Middleware;
use Closure;

class EnsureCorrectAPIHeaders
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
}
```

It's a class with a single method that takes two arguments: a Request and a closure that actually takes a request as an argument as well.

Calling the next closure will let Laravel continue to the next middleware with the request and so on. You can actually think of middlewares like layers of an onion. When a request to a Laravel application is made, the request travels through each layer into the core, where your code will be executed and you return a response, which will travel all the way out through each layer again. Calling the closure makes the request travel into the next layer beneath and returning from your middleware makes the response travel out to the layer above.

This also means that we can stop a request from travelling further into the



application and rejecting it, if it doesn't meet certain criteria, which in our case are the correct headers.

So to test our middleware, we will need a request we can send to it and then we can assert that we get a certain response from that middleware back. Let's write that test. Go back into **tests/Unit/Middleware/EnsureCorrectAPIHeadersTest.php** and add the following:

```
/**
 * @test
 * @watch
 */
public function
    it_aborts_request_if_accept_header_does_not_adhere_to_json_api_spec
    (){
    $request = Request::create('/test', 'GET');
    $middleware = new EnsureCorrectAPIHeaders;

    /** @var Response $response */
    $response = $middleware->handle($request, function($request){
        $this->fail('Did not abort request because of invalid Accept
            header');
    });

    $this->assertEquals(406, $response->status());
}
```

When setting up our world, we create a new request and create an instance of our middleware. For the first argument of our middleware, we give the request and for the **Closure** argument we give an anonymous function. The anonymous function is actually really great since we can use it to test that the middleware is not forwarding the request to the next middleware, and if it does, we will force the test to fail.

We then save the response to be able to make an assertion that we get a **406**

**Not Acceptable** status code back.

The test should be red and failing right now, so let's create the implementation and make the test pass by jumping right back into the **app/Http/Middleware/EnsureCorrectAPIHeaders.php** middleware.

Here, we will use a simple conditional to check if the Accept header is present and that it has the value:

```
Content-Type: application/vnd.api+json
```

If it doesn't, we will return a new response with a **406 Not Acceptable** status code like this:

```
public function handle($request, Closure $next)
{
    if($request->headers->get('accept') !== 'application/vnd.api+json'){
        return new Response('', 406);
    }

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

The test should pass now and we have a middleware that can reject requests that do not contain the header:

```
Accept: application/vnd.api+json
```

To make sure we are not rejecting everyone that makes requests to our API, let's also test that requests, which do contain the header above, get through

the middleware. To ensure this, we can copy the previous test and alter it slightly like this:

```
/**
 * @test
 * @watch
 */
public function
    it_accepts_request_if_accept_header_adheres_to_json_api_spec(){
    $request = Request::create('/test', 'GET');
    $request->headers->set('accept', 'application/vnd.api+json');

    $middleware = new EnsureCorrectAPIHeaders;

    /** @var Response $response */
    $response = $middleware->handle($request, function($request){
        return new Response();
    });

    $this->assertEquals(200, $response->status());
}
```

This time, we need the **Accept** header to exist and have the correct value, and to do this we set the header on the request.

Then we change the contents of the closure. This time, we want the middleware to forward the request to the closure and in order to simulate what would happen in a real Laravel application, we instantiate a new response and return it. We then make an assertion against the response, asserting that it gives us a **200 OK** status back.

The test should be green and passing now and this is actually it for the **Accept** header, so let's create a test for the **Content-Type** next.

The **Content-Type** is a little different since it's not required on every request,

but only the request where data is actually sent to the server — more specifically, the POST and PATCH requests.

In our `tests/Unit/Middleware/EnsureCorrectAPIHeadersTest.php` let's create a new test and name it: `it_aborts_post_request_if_content_type_header_does_not_adhere_to_json_api_spec`. Here, we can steal most of the contents from the first test in the file and then make sure to set the method to POST. We also need to add the correct **Accept** header, since we don't want to test that part.

Then we need to change the status code from **406** to **415** so that we adhere to the JSON:API specification like this:

```
/**
 * @test
 * @watch
 */
public function
    it_aborts_post_request_if_content_type_header_does_not_adhere_to_json_api_spec()
    {
        $request = Request::create('/test', 'POST');
        $request->headers->set('accept', 'application/vnd.api+json');
        $middleware = new EnsureCorrectAPIHeaders;

        /** @var Response $response */
        $response = $middleware->handle($request, function($request){
            $this->fail('Did not abort request because of invalid
                Content-Type header');
        });

        $this->assertEquals(415, $response->status());
    }
}
```

Before we begin implementing this part in our source code, you should make sure that you are watching all the test methods in this test class. We do this because we want to see how our implementation can affect the other

tests. Right now, you should see two passing tests and one failing for our **EnsureCorrectAPIHeadersTest** file. If you do so, let's move on to the implementation.

In the **app/Http/Middleware/EnsureCorrectAPIHeaders.php** file, let's copy the existing conditional and change the checks from **accept** to **content-type** like this:

```
public function handle($request, Closure $next)
{
    if($request->header('accept') !== 'application/vnd.api+json'){
        return new Response('', 406);
    }

    if($request->header('content-type') !== 'application/vnd.api+
        json'){
        return new Response('', 415);
    }

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

If you look at the terminal, our test for this implementation has passed but the test before it is now failing. We need to be a bit more specific when it comes to our **Content-Type** header. Right now, we are checking that the **Content-Type** header is included on every request. However, we only want to check if the **Content-Type** header is included when we are doing a POST or PATCH request. Let's wrap it into another conditional checking for exactly these conditions like this:

```
public function handle($request, Closure $next)
{
```

```

    if($request->header('accept') !== 'application/vnd.api+json'){
        return new Response('', 406);
    }

    if($request->isMethod('POST') || $request->isMethod('PATCH')){
        if($request->header('content-type') !== 'application/vnd.api+json'){
            return new Response('', 415);
        }
    }

    return $next($request);
}

```

Now, all our tests should be green and passing again. Let's do a test for a PATCH request like this:

```

/**
 * @test
 * @watch
 */
public function
    it_aborts_patch_request_if_content_type_header_does_not_adhere_to_json_api_specification()
    {
        $request = Request::create('/test', 'PATCH');
        $request->headers->set('accept', 'application/vnd.api+json');
        $middleware = new EnsureCorrectAPIHeaders;

        /** @var Response $response */
        $response = $middleware->handle($request, function($request){
            $this->fail('Did not abort request because of invalid
                Content-Type header');
        });

        $this->assertEquals(415, $response->status());
    }

```

This should also be green and passing — so far, so good. Let's then test that we can actually get through the middleware sending a correct request. From our first two tests, we know it's possible with a GET request, but we have to test it to see if it is the same for a POST or PATCH request. Again, we can reuse from what we have already, more specifically from the second test of the test class like this:

```
/**
 * @test
 * @watch
 */
public function
    it_accepts_post_request_if_content_type_header_adheres_to_json_api_spec
    (){
    $request = Request::create('/test', 'POST');
    $request->headers->set('accept', 'application/vnd.api+json');
    $request->headers->set('content-type', 'application/vnd.api+json
        ');

    $middleware = new EnsureCorrectAPIHeaders;

    /** @var Response $response */
    $response = $middleware->handle($request, function($request){
        return new Response();
    });

    $this->assertEquals(200, $response->status());
}
```

This test should be green a passing already. To make sure we have everything covered, let's also make a test for a PATCH request like this:

```
/**
 * @test
```

```

* @watch
*/
public function
    it_accepts_patch_request_if_content_type_header_adheres_to_json_api_spec
    (){
        $request = Request::create('/test', 'PATCH');
        $request->headers->set('accept', 'application/vnd.api+json');
        $request->headers->set('content-type', 'application/vnd.api+json
            ');

        $middleware = new EnsureCorrectAPIHeaders;

        /** @var Response $response */
        $response = $middleware->handle($request, function($request){
            return new Response();
        });

        $this->assertEquals(200, $response->status());
    }

```

This test should be green and passing, as well, and it's the last test we need for our unit test of our middleware right now. Before we move on, let's first remove all the **@watch** annotations from our tests in this test class.

Next, we should register our middleware as a route middleware and add it to our api middleware group. Go into **app/Http/Kernel.php** and add the middleware to the end of the **\$routeMiddleware** array like this:

```

protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\
        AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings
        ::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::

```



```

        class,
        'can' => \Illuminate\Auth\Middleware\Authorize::class,
        'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
        'signed' => \Illuminate\Routing\Middleware\ValidateSignature::
            class,
        'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::
            class,
        'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified
            ::class,

        'json.api.headers' => EnsureCorrectAPIHeaders::class
    ];

```

Afterward, add the middleware to the api middleware group like this:

```

protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::
            class,
        \Illuminate\Session\Middleware\StartSession::class,
        // \Illuminate\Session\Middleware\AuthenticateSession::class
        ,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],

    'api' => [
        'throttle:60,1',
        'bindings',
        'json.api.headers'
    ],
];

```

Go out into the terminal and stop Laravel Test Watcher. We now have to use

PHPUnit manually to see if all our tests are still passing. Let's go out into the terminal and type the following command:

```
./vendor/bin/phpunit
```

Here, you should see that all of our tests in the **tests/FeatureAuthorTest.php** file are failing. This is because these tests are not following the correct content negotiation any longer. Let's fix that.

Back in the **tests/FeatureAuthorTest.php**, let's start with the first test, where we can add the headers to the request like this:

```
/**
 * @test
 */
public function it_returns_an_author_as_a_resource_object()
{
    $author = factory(Author::class)->create();
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/authors/1', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200)
        ->assertJson([
            "data" => [
                "id" => '1',
                "type" => "authors",
                "attributes" => [
                    'name' => $author->name,
                    'created_at' => $author->created_at->toJSON(),
                ],
            ],
        ]);
}
```

```

        'updated_at' => $author->updated_at->toJSON(),
    ]
    ]
});
}

```

The second argument in the **getJson** method lets us supply the headers for the request in an array. Here, we add both the **Accept** and **Content-Type** header, because Laravel will add a

```
Content-Type: application/json
```

header by default when using the **getJson** method and we don't want that since it will abort the request. Run PHPUnit again and you should see one less test failing.

We can use the exact same array in the next test like this:

```

/**
 * @test
 */
public function
    it_returns_all_authors_as_a_collection_of_resource_objects()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $authors = factory(Author::class, 3)->create();

    $this->get('/api/v1/authors', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(200)->assertJson([
        "data" => [
            [

```

```

        "id" => '1',
        "type" => "authors",
        "attributes" => [
            'name' => $authors[0]->name,
            'created_at' => $authors[0]->created_at->toJSON
                (),
            'updated_at' => $authors[0]->updated_at->toJSON
                (),
        ]
    ],
    [
        "id" => '2',
        "type" => "authors",
        "attributes" => [
            'name' => $authors[1]->name,
            'created_at' => $authors[1]->created_at->toJSON
                (),
            'updated_at' => $authors[1]->updated_at->toJSON
                (),
        ]
    ],
    [
        "id" => '3',
        "type" => "authors",
        "attributes" => [
            'name' => $authors[2]->name,
            'created_at' => $authors[2]->created_at->toJSON
                (),
            'updated_at' => $authors[2]->updated_at->toJSON
                (),
        ]
    ],
    ],
    ];
}

```

It is as easy as that to adhere to our content negotiation through our tests. When doing **postJson** requests, you add the array of headers after the array of data like this:

```

/**
 * @test
 */
public function it_can_create_an_author_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/authors', [
        'data' => [
            'type' => 'authors',
            'attributes' => [
                'name' => 'John Doe',
            ]
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ])
        ->assertStatus(201)
        ->assertJson([
            "data" => [
                "id" => '1',
                "type" => "authors",
                "attributes" => [
                    'name' => 'John Doe',
                    'created_at' => now()->setMilliseconds(0)->
                        toJSON(),
                    'updated_at' => now() ->setMilliseconds(0)->
                        toJSON(),
                ]
            ]
        ])
        ->assertHeader('Location', url('/api/v1/authors/1'));

    $this->assertDatabaseHas('authors', [
        'id' => 1,
        'name' => 'John Doe'
    ]);
}

```

And the exact same goes for **patchJson** requests:

```
/**
 * @test
 */
public function
    it_validates_that_the_attributes_member_is_an_object_given_when_updating_an_
    (
    {
        $user = factory(User::class)->create();
        Passport::actingAs($user);
        $author = factory(Author::class)->create();

        $this->patchJson('/api/v1/authors/1', [
            'data' => [
                'id' => '1',
                'type' => 'authors',
                'attributes' => 'not an object',
            ]
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ])
        ->assertStatus(422)
        ->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The data.attributes must be an
                        array.',
                    'source' => [
                        'pointer' => '/data/attributes',
                    ]
                ]
            ]
        ]);

        $this->assertDatabaseHas('authors', [
```

```

        'id' => 1,
        'name' => $author->name,
    });
}

```

Go through all the rest of the tests in the **tests/Feature/AuthorsTest.php** file and add the correct headers, run PHPUnit again, and you should see that all tests are passing.

The last thing we need to ensure, so that our content negotiation is implemented correctly, are the correct headers on our responses from the server. Here, we need the

```
Content-Type: application/vnd.api+json
```

header on all of our API responses.

Of course, we are going to write a test for it, but this time we will let you work on your own. How would you write a test that asserts that the correct **Content-Type** header has been added to the response? Try on your own before looking at our test. As a hint, you can use the previous test in the file as a starting point, since it also asserts against the response and has the correct headers for the request already.

```

/**
 * @test
 */
public function
    it_ensures_that_a_content_type_header_adhering_to_json_api_spec_is_on_resp
    ()

```

```

{
    $request = Request::create('/test', 'GET');
    $request->headers->set('accept', 'application/vnd.api+json');
    $request->headers->set('content-type', 'application/vnd.api+json');

    $middleware = new EnsureCorrectAPIHeaders;

    /** @var Response $response */
    $response = $middleware->handle($request, function($request){
        return new Response();
    });

    $this->assertEquals(200, $response->status());
    $this->assertEquals('application/vnd.api+json', $response->
        headers->get('content-type'));
}

```

We have made very few changes from the previous test. We added the assertion for the right **Content-Type** header and changed the method to GET. Let's write the implementation, which is quite simple in that we just need to add the header to the response that comes into the middleware like this:

```

<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Response;
use Symfony\Component\HttpFoundation\Response as BaseResponse;

class EnsureCorrectAPIHeaders
{
    /**
     * Handle an incoming request.

```



```

*
* @param \Illuminate\Http\Request $request
* @param \Closure $next
* @return mixed
*/
public function handle($request, Closure $next)
{
    if($request->header('accept') !== 'application/vnd.api+json')
    {
        return new Response('', 406);
    }

    if($request->headers->has('content-type') || $request->isMethod('POST') || $request->isMethod('PATCH')){
        if($request->header('content-type') !== 'application/vnd.api+json'){
            return new Response('', 415);
        }
    }

    return $this->addCorrectContentType($next($request));
}

private function addCorrectContentType(BaseResponse $response)
{
    $response->headers->set('content-type', 'application/vnd.api+json');
    return $response;
}
}

```

We have added a `addCorrectContentType` method that takes one argument, which is a `BaseResponse`. The **BaseResponse** is worth noting here, since it is an alias of the **Symfony\Component\HttpFoundation\Response** class. We need this alias since we are already importing Laravel's own **Response**. The reason that we are referencing Symfony's **Response** class is that Laravel is based on components from the Symfony framework, especially the components that handle requests and responses, which means that the

**Symfony\Component\HttpFoundation\Response** is actually the parent of the Laravel Response class. By referencing the parent class in the argument, we are able to pass in any class that inherits from the **Symfony\Component\HttpFoundation\Response** class. This means that a **Illuminate\Foundation\Testing\TestResponse** we use when we are testing our API, as well as regular **Illuminate\Http\Response**, can be passed in since both of these are inheriting from **Symfony\Component\HttpFoundation\Response**.

If you run PHPUnit in the terminal again, you should see all green and passing tests.

We have done this in a separate method so that we can reuse it, since we also need to send the correct headers, even when we abort a request. So we will need to add this method around the responses sent in the conditionals testing for missing **Accept** and **Content-Type** headers.

But before we do that, we need to write some tests for it. We will let you work on these on your own, as well as the implementation and just give you our implementation below. Again, try to work on your own before you look at our implementations. If you want to go back to using Laravel Test Watcher, feel free to do so. It's a bit easier than having to run PHPUnit all of the time.

#### **tests/Unit/Middleware/EnsureCorrectAPIHeadersTest.php:**

```
/**
 * @test
 * @watch
 */
public function
    when_aborting_for_a_missing_accept_header_the_correct_content_type_header_is
    (
    {
        $request = Request::create('/test', 'GET');
```

```

$middleware = new EnsureCorrectAPIHeaders;

/** @var Response $response */
$response = $middleware->handle($request, function($request){
    return new Response();
});

$this->assertEquals($response->status(), 406);
$this->assertEquals('application/vnd.api+json', $response->
    headers->get('content-type'));
}

/**
 * @test
 * @watch
 */
public function
    when_aborting_for_a_missing_content_type_header_the_correct_content_type_h
    ()
{
    $request = Request::create('/test', 'POST');
    $request->headers->set('accept', 'application/vnd.api+json');

    $middleware = new EnsureCorrectAPIHeaders;

    /** @var Response $response */
    $response = $middleware->handle($request, function($request){
        return new Response();
    });

    $this->assertEquals(415, $response->status());
    $this->assertEquals('application/vnd.api+json', $response->
        headers->get('content-type'));
}

```

**app/Http/Middleware/EnsureCorrectAPIHeaders.php:**

```

public function handle($request, Closure $next)
{
    if($request->header('accept') !== 'application/vnd.api+json'){
        return $this->addCorrectContentType(new Response('', 406));
    }

    if($request->headers->has('content-type') || $request->isMethod(
        'POST') || $request->isMethod('PATCH')){
        if($request->header('content-type') !== 'application/vnd.api+json'){
            return $this->addCorrectContentType(new Response('', 415));
        }
    }

    return $this->addCorrectContentType($next($request));
}

```

We hope you got through that on your own without looking too much at our implementations. Right now, all our tests should pass, and we are ready to move on. As of right now, our API is almost adhering to the JSON:API specification — the only thing we are missing is actually exception handling. We will take a look at this next and build on top of what we have already done with error handling of our validation. After this, we will be looking at the more optional features, which the JSON:API specification doesn't force us to implement, namely sorting and pagination.

### *Exception handling*

It's time to look at exception handling and build onto what we have already started when building the validation error handling. While the validation error handling works as it should and does adhere to the conventions of the JSON:API specification, the rest of our exception responses do not. Here, we are mostly concerned with the exceptions that our consumers could meet when making requests to our API. This could be an **AuthenticationException**, which Laravel

will convert into a JSON response with a single **message** member telling that you are unauthenticated. We want to change this so it adheres to the JSON:API specifications error responses.

To do this, we will work in the **app/Exceptions/Handler.php** file again, so let's open that up. Right now, it contains the **report**, **render** and **invalidJson** methods. When an exception or error happens in our application, Laravel will call the **report** and **render** methods in this class.

The **report** method is for tracking and logging exceptions so you can get a better understanding of what went wrong.

The render method is for presenting the exception to the user. Here, it depends on the debug state of your application as well as the **Accept** header of the request. If your application has debugging enabled, you will get a nice error page where you can dig down into the stack trace of the exception. If your **Accept** header is set to **application/json** or alike, you will receive a detailed error response with the entire stack trace in JSON. If your application has debugging disabled, you will see an error page or a short JSON object with a short message.

In our case, we want to change the response for the JSON part only. So let's take a closer look at what actually happens in the **render** method to determine what to do. Open up the **vendor/laravel/framework/src/Illuminate/Foundation/Exceptions/Handler.php** file and look at the **render** method:

```
public function render($request, Exception $e)
{
    if (method_exists($e, 'render') && $response = $e->render(
        $request)) {
        return Router::toResponse($request, $response);
    } elseif ($e instanceof Responsable) {
```

```

        return $e->toResponse($request);
    }

    $e = $this->prepareException($e);

    if ($e instanceof HttpResponseException) {
        return $e->getResponse();
    } elseif ($e instanceof AuthenticationException) {
        return $this->unauthenticated($request, $e);
    } elseif ($e instanceof ValidationException) {
        return $this->convertValidationExceptionToResponse($e,
            $request);
    }

    return $request->expectsJson()
        ? $this->prepareJsonResponse($request, $e)
        : $this->prepareResponse($request, $e);
}

```

When the **render** method is called, Laravel will check to see what kind of exception it is. If the exception contains a **render** method, Laravel will delegate the rest of the process to this method.

If it is a **HttpException** it will just return this.

If it's a **ValidationException** it will delegate the exception the necessary methods for handling validation errors, which we have already covered.

It also checks if an **AuthenticationException** has been thrown and again delegates to the necessary methods for handling this kind of exception. This is something our consumers could get, so we will have to take care of this.

If the exception being thrown does not contain a **render** method on it's own class, Laravel will handle the rest of the rendering of the exception. Here, Laravel will call a **prepareJsonResponse** method for handling JSON responses. If the client expects JSON, we will need to override this method to be able to

build the response we want.

As with anything else from here on out, we will start by making a test and then doing the implementation. To be able to test the various types of exceptions and that we will be getting the right result, the easiest way to test this class is through a unit test. So let's create a new class at **tests/Unit/Exceptions/HandlerTest.php** like this:

```
<?php

namespace Tests\Unit\Exceptions;

class HandlerTest extends TestCase
{

}
```

For our first test, let's make sure that a general exception is being handled the way we want, which will be a response with a single error object. To be able to test this, we need to have an instance of our **Handler** class so that we can call the **render** method on this. Let's go back to the codeblock showing the **render** method. Here, we can see that it takes a request and an exception as arguments and will return a response. This is perfect, since we can then send in the various exceptions we want to test against, making sure we get the correct response back. Let's create the first test like this, and afterward, we will go through it and explain further:

```
/**
 * @test
 * @watch
 */
public function
    it_converts_an_exception_into_a_json_api_spec_error_response()
```

```

{
    $handler = app(Handler::class);

    $request = Request::create('/test', 'GET');
    $request->headers->set('accept', 'application/vnd.api+json');

    $exception = new \Exception('Test exception');

    $response = $handler->render($request, $exception);
    TestResponse::fromBaseResponse($response)->assertJson([
        'errors' => [
            [
                'title' => 'Exception',
                'details' => 'Test exception',
            ]
        ]
    ]);
}

```

We start out by instantiating our **Handler** class by using the built in **app** function. This is a helper function that makes it possible to tell Laravel to create a new instance of a class. If the class has any dependencies, Laravel will do its best to fetch these and inject them where they are needed. The **Handler** class' parent actually has a dependency to the Laravel Container, so instead of us having to do the hard work, we can just let Laravel do what it does best, injecting the container into the **Handler**.

Next up, we instantiate a **Request** like we have done before and set the correct **Accept** header so that the **Handler** knows that we expect JSON.

Then, we instantiate an **Exception** and here we are using the good old global PHP **Exception** class.

We call the render method, passing in both the request and exception.

We then capture the response, which we will use with the **TestResponse's**



**fromBaseResponse** static method. This method makes it possible for us to assert against the JSON structure, just like responses of our feature tests. We actually use this to assert that we get the right JSON back.

If you haven't started Laravel Test Watcher, let's do that now with the following command:

```
php artisan tests:watch
```

Our test is failing because we are not getting the right response back, so let's change that by going into **app/Exceptions/Handler.php** and override the **prepareJsonResponse** like this:

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;
use Illuminate\Support\Collection;
use Illuminate\Support\Str;
use Illuminate\Validation\ValidationException;

class Handler extends ExceptionHandler
{
    protected $dontReport = [
        //
    ];

    protected $dontFlash = [
        'password',
        'password_confirmation',
    ];
}
```

```

public function report(Exception $exception)
{
    parent::report($exception);
}

public function render($request, Exception $exception)
{
    return parent::render($request, $exception);
}

protected function prepareJsonResponse($request, Exception $e)
{
    return response()->json([
        'errors' => [
            [
                'title' => Str::title(Str::snake(class_basename(
                    $e), ' ')),
                'details' => $e->getMessage(),
            ]
        ], $this->isHttpException($e) ? $e->getStatusCode() : 500);
}

protected function invalidJson($request, ValidationException
    $exception)
{
    $errors = ( new Collection($exception->validator->errors())
        )
        ->map(function ($error, $key) {
            return [
                'title'    => 'Validation Error',
                'details' => $error[0],
                'source' => [
                    'pointer' => '/' . str_replace('.', '/',
                        $key),
                ]
            ];
        })
        ->values();
}

```

```

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

```

We place the method right between the **render** method and the **invalidJson** method.

Let's take a closer look at what happens inside the method:

```

protected function prepareJsonResponse($request, Exception $e)
{
    return response()->json([
        'errors' => [
            [
                'title' => Str::title(Str::snake(class_basename($e),
                    ' ')),
                'details' => $e->getMessage(),
            ]
        ]
    ], $this->isHttpException($e) ? $e->getStatusCode() : 500);
}

```

We return a JSON response just like before and in the response we have the **error** member as our top level member. The **error** member is a collection of errors where, in this case, we will only have one. In this error object, we cannot use the **source** member, since we don't have another JSON member to reference to, so we will omit this here, keeping only the **title** and **details** members.

For the **title** member, we use the class name of the exception to get the title. Unfortunately, class names are written using camel casing, but we want to

have a nicely formatted title with spaces in between words. To do this, we use two helper methods Laravel provides out of the box to manipulate strings. The first is **Str::snake**, which takes two arguments, a string to convert, and a delimiter. The great thing about this method is that it can take camel cased strings and convert these to snake casing as well. If you remember when we covered snake casing, the delimiter is the `_` character in between words. The **Str::snake** method actually lets us choose the delimiter ourselves, which we can use to swap the `_` with a space, which will make our camel case string in to a string with spaces instead. Then we use **Str::title** to upper case all the words in the string, which will give us the final title of the exception.

For the **detail** member, we take the message from the exception, which will tell the details of the exception.

After the data for our response, we give the status code. Since this could be an **HttpException**, depending on the exception type, a status code can be included and if it is, we will just return that or else we will just return **500** for a **500 Internal Server Error**.

Our test should now be green and passing. But let's also ensure that we test everything. Right now we don't assert against the returned status code, so let's do that:

```
/**
 * @test
 * @watch
 */
public function
    it_converts_an_exception_into_a_json_api_spec_error_response()
{
    /** @var Handler $handler */
    $handler = app(Handler::class);
```

```

$request = Request::create('/test', 'GET');
$request->headers->set('accept', 'application/vnd.api+json');

$exception = new \Exception('Test exception');

$response = $handler->render($request, $exception);
TestResponse::fromBaseResponse($response)->assertJson([
    'errors' => [
        [
            'title' => 'Exception',
            'details' => 'Test exception',
        ]
    ]
])->assertStatus(500);
}

```

Great, the test is still passing and it works as it should. Let's move on to an HTTP Exception then.

This test is not very different from the one we just wrote. The only thing we have to change is the exception and what we expect to get back in the error objects **title** and **details** members.

So let's swap out the exception and change the error object to make the following test:

```

/**
 * @test
 * @watch
 */
public function
    it_converts_an_http_exception_into_a_json_api_spec_error_response
    ()
{
    /** @var Handler $handler */

```

```

$handler = app(Handler::class);

$request = Request::create('/test', 'GET');
$request->headers->set('accept', 'application/vnd.api+json');

$exception = new HttpException(404, 'Not found');

$response = $handler->render($request, $exception);
TestResponse::fromBaseResponse($response)->assertJson([
    'errors' => [
        [
            'title'    => 'Http Exception',
            'details' => 'Not found',
        ]
    ]
])->assertStatus(404);
}

```

This test should also be passing already, since it relies on the implementation we just made.

But what about an **AuthenticationException** — will it hit the same method? The only way to find out is to create a test and see if it passes. Again, we just need to change the exception and the **title** and **details** values of the error object like this:

```

/**
 * @test
 * @watch
 */
public function
    it_converts_an_unauthenticated_exception_into_a_json_api_spec_error_response
    ()
    {
        /** @var Handler $handler */
    }

```

```

$handler = app(Handler::class);

$request = Request::create('/test', 'GET');
$request->headers->set('accept', 'application/vnd.api+json');

$exception = new AuthenticationException();

$response = $handler->render($request, $exception);
TestResponse::fromBaseResponse($response)->assertJson([
    'errors' => [
        [
            'title' => 'Unauthenticated',
            'details' => 'You are not authenticated',
        ]
    ]
]);
}

```

This test fails and complains about the response, which shows us that we are not using the same method to handle the response.

If we go back and take a look at the **vendor/laravel/framework/src/Illuminate/Foundation/Exceptions/Handler.php** file and the **render** method again, we can see that when a **AuthenticationException** is thrown, the **unauthenticated** method is taking care of the response:

```

public function render($request, Exception $e)
{
    if (method_exists($e, 'render') && $response = $e->render(
        $request)) {
        return Router::toResponse($request, $response);
    } elseif ($e instanceof Responsable) {
        return $e->toResponse($request);
    }
}

```

```

    $e = $this->prepareException($e);

    if ($e instanceof HttpResponseException) {
        return $e->getResponse();
    } elseif ($e instanceof AuthenticationException) {
        return $this->unauthenticated($request, $e);
    } elseif ($e instanceof ValidationException) {
        return $this->convertValidationExceptionToResponse($e,
            $request);
    }

    return $request->expectsJson()
        ? $this->prepareJsonResponse($request, $e)
        : $this->prepareResponse($request, $e);
}

```

So to take care of the response when a **AuthenticationException** is being thrown, we just have to override the **unauthenticated** method in our **app/Exceptions/Handler.php** class. Let's add this method right after the **invalidJson** method and add the following code:

```

protected function unauthenticated($request,
    AuthenticationException $exception)
{
    if($request->expectsJson()){
        return response()->json([
            'errors' => [
                [
                    'title' => 'Unauthenticated',
                    'details' => 'You are not authenticated',
                ]
            ]
        ], 403);
    }
    return redirect()->guest($exception->redirectTo() ?? route('login'));
}

```



```
}
```

Since this method is responsible for what happens to requests that expect JSON and requests that expect HTML, we need to handle this case. So if JSON is expected, we handle this with an error response that adheres to the JSON:API specification, just like the ones before. If we expect HTML, we will use the existing code from the parents implementation of the **unauthenticated** method for redirecting the user to the login page.

This is it for our exception handling — this part now also adheres to the JSON:API specification, so let's move on to the more optional things to implement.

## *Sorting*

Let's look at sorting as one of the first optional implementations of the JSON:API specification we can make. We have already described what sorting does, so we won't bother you with the details again. The thing to worry about here is how we are going to implement this functionality. It will require some kind of conversion of the **sort** query parameter to the "**ORDER BY**" part of our SQL sentence, when querying the database for authors. We could go and build this manually, but for this part we will use a third party package called **Laravel Query Builder** that will do the heavy lifting for us.

The Laravel Query Builder package is built by the Belgian company [Spatie](#) and it gives us the ability to sort on our Eloquent queries. The best thing is that it does this according to the JSON:API specification. So let's install this package and start implementing it into our author resource.

The installation is easily done through Composer like this:

```
composer require spatie/laravel-query-builder
```

And for now that is it: Laravel will auto discover the package so we are ready to implement its functionality into our authors resource. Before we do so, let's go back to the **tests/Feature/AuthorsTest.php** file and create a new test. For this test, we can start out by copying the contents of the **it\_returns\_all\_authors\_as\_a\_collection\_of\_resource\_objects** and then make the following changes:

```
/**
 * @test
 * @watch
 */
public function
    it_can_sort_authors_by_name_through_a_sort_query_parameter()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $authors = collect([
        'Bertram',
        'Claus',
        'Anna',
    ])->map(function($name){
        return factory(Author::class)->create([
            'name' => $name
        ]);
    });

    $this->get('/api/v1/authors?sort=name', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(200)->assertJson([
        "data" => [
            [
                "id" => '3',
            ]
        ]
    ]));
}
```

```

        "type" => "authors",
        "attributes" => [
          'name' => 'Anna',
          'created_at' => $authors[2]->created_at->toJSON
            (),
          'updated_at' => $authors[2]->updated_at->toJSON
            (),
        ]
      ],
      [
        "id" => '1',
        "type" => "authors",
        "attributes" => [
          'name' => 'Bertram',
          'created_at' => $authors[0]->created_at->toJSON
            (),
          'updated_at' => $authors[0]->updated_at->toJSON
            (),
        ]
      ],
      [
        "id" => '2',
        "type" => "authors",
        "attributes" => [
          'name' => 'Claus',
          'created_at' => $authors[1]->created_at->toJSON
            (),
          'updated_at' => $authors[1]->updated_at->toJSON
            (),
        ]
      ],
    ],
  ];
}

```

```

}

```

The thing to notice here is firstly the way we set up our world by creating the authors, which is a nice little trick to use. Here, we instantiate a new **Collection** with an array of names. We then use the **map** method to map each name into a new Author model, which both forms a new collection of Author models,

but also creates them in the database. We then have a collection of authors, just like if we have told our **factory** function to create three authors, but with specific names instead and we have created the authors in the database at the same time.

There is a purpose with the name order, since we want to know that they are sorted correctly. If we had entered them alphabetically, we would not be sure about the sort functionality working, but we can with a mixed order and we also know which **ID** will be attached to which name.

In the **getJson** method, we then add the **sort** query parameter to the URL, giving it the name attribute as the attribute we want to sort on. In the **assertJson** method, we order the authors in alphabetical order, since we expect the sort functionality to work this way.

Right now, the test is failing because we haven't implemented the Laravel Query Builder, but let's do that now. Open the **app/Http/Controllers/AuthorsController.php** file and let's focus on the **index** method:

```
public function index()
{
    $authors = QueryBuilder::for(Author::class)->allowedSorts([
        'name'
    ])->get();
    return new AuthorsCollection($authors);
}
```

Here, we exchange the existing Eloquent query with the QueryBuilder and tell it to work with our **Author** model. We tell it which attributes it is allowed to sort, which is **name**. The main reason for us to give a list of attributes to the **allowedSorts** method, is that we get a white list of attributes that can be sorted. This is not to limit our consumers, but to protect ourselves a bit more against SQL injections. Then, just like you would with an Eloquent query after

you have chained various methods onto the query, we call the **get** method.

Like before, we will get a collection of authors which we can pass to our **AuthorsCollection** class. Our test should now be passing and green.

Next up, let's create a test for sorting in descending order:

```
/**
 * @test
 * @watch
 */
public function
    it_can_sort_authors_by_name_in_descending_order_through_a_sort_query_param
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $authors = collect([
        'Bertram',
        'Claus',
        'Anna',
    ])->map(function($name){
        return factory(Author::class)->create([
            'name' => $name
        ]);
    });

    $this->get('/api/v1/authors?sort=-name', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(200)->assertJson([
        "data" => [
            [
                "id" => '2',
                "type" => "authors",
                "attributes" => [
```

```

        'name' => 'Claus',
        'created_at' => $authors[1]->created_at->toJSON
            (),
        'updated_at' => $authors[1]->updated_at->toJSON
            (),
    ]
],
[
    "id" => '1',
    "type" => "authors",
    "attributes" => [
        'name' => 'Bertram',
        'created_at' => $authors[0]->created_at->toJSON
            (),
        'updated_at' => $authors[0]->updated_at->toJSON
            (),
    ]
],
[
    "id" => '3',
    "type" => "authors",
    "attributes" => [
        'name' => 'Anna',
        'created_at' => $authors[2]->created_at->toJSON
            (),
        'updated_at' => $authors[2]->updated_at->toJSON
            (),
    ]
],
]
]);
}

```

Here, we add a minus in front of our **name** value in the query parameter to convey that we want to sort in descending order. Then we reorder the authors in the **assertJson** method so that they match the expected sort order.

This test should already be green and passing, since Laravel Query Builder is adhering to the convention of the JSON:API specification. To cement this

further, let's create tests for sorting on multiple attributes, but before we do so, we have to update our controller and set another attribute on which we can be sorting. In this case, we might as well add both our **created\_at** and **updated\_at** attributes like this:

```
public function index()
{
    $authors = QueryBuilder::for(Author::class)->allowedSorts([
        'name',
        'created_at',
        'updated_at',
    ])->get();
    return new AuthorsCollection($authors);
}
```

Then we can write our test to see if the sorting functionality works as it should like this:

```
/**
 * @test
 * @watch
 */
public function
    it_can_sort_authors_by_multiple_attributes_through_a_sort_query_parameter
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $authors = collect([
        'Bertram',
        'Claus',
        'Anna',
    ])->map(function($name){
```

```

        if($name === 'Bertram'){
            return factory(Author::class)->create([
                'name' => $name,
                'created_at' => now()->addSeconds(3),
            ]);
        }

        return factory(Author::class)->create([
            'name' => $name,
        ]);
    });

    $this->get('/api/v1/authors?sort=created_at,name', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(200)->assertJson([
        "data" => [
            [
                "id" => '3',
                "type" => "authors",
                "attributes" => [
                    'name' => 'Anna',
                    'created_at' => $authors[2]->created_at->toJSON
                        (),
                    'updated_at' => $authors[2]->updated_at->toJSON
                        (),
                ]
            ],
            [
                "id" => '2',
                "type" => "authors",
                "attributes" => [
                    'name' => 'Claus',
                    'created_at' => $authors[1]->created_at->toJSON
                        (),
                    'updated_at' => $authors[1]->updated_at->toJSON
                        (),
                ]
            ]
        ],
    ]),

```



```

    [
      "id" => '1',
      "type" => "authors",
      "attributes" => [
        'name' => 'Bertram',
        'created_at' => $authors[0]->created_at->toJSON
          (),
        'updated_at' => $authors[0]->updated_at->toJSON
          (),
      ]
    ],
  ];
}

```

In this test, we sort the **created\_at** attribute first and then the **name** attribute. The way the sorting works here is that it will sort the authors by the **created\_at** date first, and if any authors should be created on the same date, it will sort these by the **name** attribute. To be able to test this, we will need one of our authors to have a different **created\_at** date than the rest, so that we can test that the sorting functionality works. For this purpose, we have created a simple conditional when creating our authors, where Bertram will be created later than the others. We then change the order of the authors in the **assertJson** method so that they match our expected outcome.

This test should be green and passing, which means that the sorting works exactly as it should. But what about the descending order? Let's test that now:

```

/**
 * @test
 * @watch
 */
public function
    it_can_sort_authors_by_multiple_attributes_in_descending_order_through_a_s

```

```

    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $authors = collect([
        'Bertram',
        'Claus',
        'Anna',
    ])->map(function($name){
        if($name === 'Bertram'){
            return factory(Author::class)->create([
                'name' => $name,
                'created_at' => now()->addSeconds(3),
            ]);
        }
        return factory(Author::class)->create([
            'name' => $name,
        ]);
    });
    $this->get('/api/v1/authors?sort=-created_at,name', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(200)->assertJson([
        "data" => [
            [
                "id" => '1',
                "type" => "authors",
                "attributes" => [
                    'name' => 'Bertram',
                    'created_at' => $authors[0]->created_at->toJSON(),
                    'updated_at' => $authors[0]->updated_at->toJSON(),
                ]
            ],
            [
                "id" => '3',
                "type" => "authors",
                "attributes" => [
                    'name' => 'Anna',

```

```

        'created_at' => $authors[2]->created_at->toJSON
        (),
        'updated_at' => $authors[2]->updated_at->toJSON
        (),
    ]
],
[
    "id" => '2',
    "type" => "authors",
    "attributes" => [
        'name' => 'Claus',
        'created_at' => $authors[1]->created_at->toJSON
        (),
        'updated_at' => $authors[1]->updated_at->toJSON
        (),
    ]
],
]
]);
}

```

In this test, we add a minus in front of the **created\_at** value of our sort query parameter which indicated that we want to sort our **created\_at** attribute in descending order. We then reorder the authors in the **assertJson** method so that they match the expected order.

This is it for the sorting functionality. Everything is implemented as it should and we can move on to pagination.

## Pagination

It's time for the last part of adhering to the JSON:API specification, namely pagination.

You might already know this feature from Laravel itself, which already has a pagination feature with the ability to split a large set of rows from the database into chunks of 10 rows per page, so you can both save that database from going

through all rows but also save the user from scrolling on a long list. The pagination feature built into Laravel works very well, but unfortunately it doesn't adhere to the JSON:API specification. Just like with sorting, we can use a third party package and that is just what we will do. It is also a package from Belgian Spatie called Laravel JSON API Paginate.

Again, it's easily installed through composer like this:

```
composer require spatie/laravel-json-api-paginate
```

Just like with the sorting part, we need to first write a test in our **tests/Feature/AuthorsTest.php** file. Just like before, we will copy from the **it\_returns\_all\_authors\_as\_a\_collection\_of\_resource\_objects** test, since it gives us a nice starting point to work from. To set up our world, we will need more than three authors for this test though, so let's bump it up to ten and work from there. We will, of course, also need to be authenticated. Let's just present the test to you and go through it afterward:

```
/**
 * @test
 * @watch
 */
public function
    it_can_paginate_authors_through_a_page_query_parameter()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $authors = factory(Author::class, 10)->create();

    $this->get('/api/v1/authors?page[size]=5&page[number]=1', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
```

```

])->assertStatus(200)->assertJson([
  "data" => [
    [
      "id" => '1',
      "type" => "authors",
      "attributes" => [
        'name' => $authors[0]->name,
        'created_at' => $authors[0]->created_at->toJSON
          (),
        'updated_at' => $authors[0]->updated_at->toJSON
          (),
      ]
    ],
    [
      "id" => '2',
      "type" => "authors",
      "attributes" => [
        'name' => $authors[1]->name,
        'created_at' => $authors[1]->created_at->toJSON
          (),
        'updated_at' => $authors[1]->updated_at->toJSON
          (),
      ]
    ],
    [
      "id" => '3',
      "type" => "authors",
      "attributes" => [
        'name' => $authors[2]->name,
        'created_at' => $authors[2]->created_at->toJSON
          (),
        'updated_at' => $authors[2]->updated_at->toJSON
          (),
      ]
    ],
    [
      "id" => '4',
      "type" => "authors",
      "attributes" => [
        'name' => $authors[3]->name,

```

```

        'created_at' => $authors[3]->created_at->toJSON
            (),
        'updated_at' => $authors[3]->updated_at->toJSON
            (),
    ]
],
[
    "id" => '5',
    "type" => "authors",
    "attributes" => [
        'name' => $authors[4]->name,
        'created_at' => $authors[4]->created_at->toJSON
            (),
        'updated_at' => $authors[4]->updated_at->toJSON
            (),
    ]
],
],
'links' => [
    'first' => route('authors.index', ['page[size]' => 5, '
        page[number]' => 1]),
    'last' => route('authors.index', ['page[size]' => 5, '
        page[number]' => 2]),
    'prev' => null,
    'next' => route('authors.index', ['page[size]' => 5, '
        page[number]' => 2]),
]
]);
}

```

We set the **page[size]** and **page[number]** query parameters to our URL. We set the size to **5** so that we know that we will get two pages with five authors on each. We then set the expected amount of authors in the **assertJson** method and also set the **links** member, which is a required member for pagination according to the JSON:API specification.

When starting, this test is failing but it is easy to make this one pass. Once more, we need to go into **app/Http/Controllers/AuthorsController.php** and again

we are working in the **index** method. To use the Laravel JSON API paginate package, all we have to do is to call the **jsonPaginate** method instead of **get** when we want to make the query to the database, so let's replace the call like this:

```
public function index()
{
    $authors = QueryBuilder::for(Author::class)->allowedSorts([
        'name',
        'created_at',
        'updated_at',
    ])->jsonPaginate();
    return new AuthorsCollection($authors);
}
```

Now our test is green and passing, it is as easy as that. Behind the scenes the **jsonPaginate** method will look for the **page[size]** and **page[number]** query parameters and do the entire pagination for us, and also add the **links** member needed for pagination as well.

This is all we have to do to implement pagination in our API, and it also concludes the parts we need to do to adhere to the JSON:API specification.

## Summary

We hope you are still with us! We know this chapter was a long one, but it made it possible for us to build a lot of the fundamentals for both testing and implementing the basic features of the JSON:API specification.

And we did do a lot in this chapter: we started by looking at the test tools Laravel provides out of the box and how we can use these together with PHPUnit to test our application from small unit test all the way to end-to-end tests.

We went through our own package for test-driven development and used this throughout this chapter to automate our tests further by only adding a `@watch` annotation to our tests. We revisited the author resource and made sure that we had a test for each endpoint so that we can be sure that these work as they should.

We then tested validation and began implementing how to handle errors according to the JSON:API specification.

Lastly, we ended this chapter by implementing the last parts we needed to adhere to the JSON:API specification, which included the content negotiation we had left out until now, then exception handling and lastly the optional sorting and pagination features.

We have now implemented what we can of our authors resource. To get any further, we need a relationship to books, but we cannot implement this until books has been implemented, so let's do that in the next chapter.

\* \* \*



## 6

# Books

It's time for us to look at the main part of both our API and also this book. Looking at Anna's Bookstore, the books are the main thing. They are what everything revolves around and since we are reflecting what Anna's Bookstore is all about, naturally our application and API will also be revolving around books.

So in this chapter we will be building our book resource first, which will also be a good repetition of the concepts of the last chapter.

Then, we will be looking at relationships, especially how Laravel's relationships can be mapped to the conventions of the JSON:API specification and how to implement these.

We will conclude this chapter by looking at the **include** query parameter and the **included** top-level for including related resource objects of the relationships to a single or collection of resources.

Before we move into this chapter and start working, we just want to cover how we will be explaining tests for the rest of this book, because it's a little different than we have done so far.

Earlier in this book, we talked about structuring tests into three parts and we want to emphasize this further, because it can help you to more easily know what you should type when writing your own test.

The structure we are talking about is the following structure to a test:

1. In the first part, we set up our world
2. In the second part, we run the code that should be tested
3. In the last part, we make all of our assertions

We will be breaking tests into this structure, so you know what we are about to write. Then, we will be showing the code for our tests, which you can use as help when writing the test on your own. Lastly, we will, of course, be going through how to implement what we are trying to test. The reason for doing it this way, is that you should begin writing more tests on your own. From these breakdowns, you should write as many tests as you can and use our test code as help if you get stuck, but also compare what you have done in your own tests to ours. In this way, you are learning by doing, instead of us telling you what to do throughout the entire book, which will get boring and tedious fast. Next up, we will essentially be writing the same tests for books as we have done for authors, so it's a great way to revise what we have just been through, which will make it easier for you to know what to write in your tests.

We really want you to learn this form of testing — even though it gets repetitive from time to time — because it's a great workflow once you get into it. We have written entire APIs for applications without leaving our IDE, without having to go back and forth to the browser or Postman. We have even written most of an API on a train without having an internet connection. With Laravel's test tools and your IDE or editor, you can drive out the implementation of your APIs much easier and when you really get into the flow, you can also go faster.

Remember that it is all right to deviate from what we have written. You might even be familiar with assertions or tricks that we are not and that's fine — you

should use them in that case. But do yourself the favor and don't deviate in the way that you are removing assertions to move faster. You should make assertions until you are confident in your tests.

## The Books resource

First, we will be building the books resource. Here, we will do some revision from the last chapter again, to further cement the knowledge on both how to implement our API using the JSON:API specification and also how to do this in a test-driven way.

### *Preparing our tests*

Just like the last chapter, we need to write tests for our books resource and before we can do that, we need a test file. Again, we are doing our tests from an API point so we will need a feature test. Let's create our new test file as this: **tests/Feature/BooksTest.php**.

By now, you should be a bit more familiar with the concepts of extending the **TestCase** class and while we are at it, let's add the **DatabaseMigrations** trait as well like this:

```
<?php

namespace Tests\Feature;

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

class BooksTest extends TestCase
{
    use DatabaseMigrations;
}
```

We want to get the ball rolling a bit faster with this test, and we know that what we want to implement essentially are the same features as we have implemented for authors. After all, we are implementing a specification to give us more consistency. So to give ourselves a better start, why don't we copy all the test method names from the **tests/Feature/AuthorsTest.php** file into our new file. At the same time, we will also change all the test method names that have either **authors** or **author** in them to **books** and **book**. Remember that the attributes won't be the same for this resource, so the tests that test validation for the **name** member should be deleted, and we need to make new validation tests for the: **title**, **description** and **publication\_year** members.

All the test methods with the changes should then be like this:

```
<?php

namespace Tests\Feature;

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

class BooksTest extends TestCase
{
    use DatabaseMigrations;

    /**
     * @test
     */
    public function it_returns_an_book_as_a_resource_object()
    {

    }

    /**
     * @test
     */
```

```

public function
    it_returns_all_books_as_a_collection_of_resource_objects()
{

}

/**
 * @test
 */
public function it_can_create_an_book_from_a_resource_object()
{

}

/**
 * @test
 */
public function
    it_validates_that_the_type_member_is_given_when_creating_an_book
    ()
{

}

/**
 * @test
 */
public function
    it_validates_that_the_type_member_has_the_value_of_books_when_creating_
    ()
{

}

/**
 * @test
 */
public function
    it_validates_that_the_attributes_member_has_been_given_when_creating_an
    ()

```

```

{

}

/**
 * @test
 */
public function
    it_validates_that_the_attributes_member_is_an_object_given_when_creating_
    (
    {

    }

/**
 * @test
 */
public function
    it_validates_that_a_title_attribute_is_given_when_creating_an_book
    (
    {

    }

/**
 * @test
 */
public function
    it_validates_that_a_title_attribute_is_a_string_when_creating_an_book
    (
    {

    }

/**
 * @test
 */
public function
    it_validates_that_a_description_attribute_is_given_when_creating_an_book
    (

```

```

{

}

/**
 * @test
 */
public function
    it_validates_that_a_description_attribute_is_a_string_when_creating_an_
    ()
{

}

/**
 * @test
 */
public function
    it_validates_that_a_publication_year_attribute_is_given_when_creating_a_
    ()
{

}

/**
 * @test
 */
public function
    it_validates_that_a_publication_year_attribute_is_a_string_when_creatin
    ()
{

}

/**
 * @test
 */
public function it_can_update_an_book_from_a_resource_object()
{

```

```

}

/**
 * @test
 */
public function
    it_validates_that_an_id_member_is_given_when_updating_an_book
    ()
{

}

/**
 * @test
 */
public function
    it_validates_that_an_id_member_is_a_string_when_updating_an_book
    ()
{

}

/**
 * @test
 */
public function
    it_validates_that_the_type_member_is_given_when_updating_an_book
    ()
{

}

/**
 * @test
 */
public function
    it_validates_that_the_type_member_has_the_value_of_books_when_updating_an
    ()
{

```



```

}

/**
 * @test
 */
public function
    it_validates_that_the_attributes_member_has_been_given_when_updating_an
    ()
{

}

/**
 * @test
 */
public function
    it_validates_that_the_attributes_member_is_an_object_given_when_updatin
    ()
{

}

/**
 * @test
 */
public function
    it_validates_that_a_title_attribute_is_a_string_when_updating_an_book
    ()
{

}

/**
 * @test
 */
public function
    it_validates_that_a_description_attribute_is_a_string_when_updating_an
    ()
{

```

```

}

/**
 * @test
 */
public function
    it_validates_that_a_publication_year_attribute_is_a_string_when_updating_
    ()
{

}

/**
 * @test
 */
public function it_can_delete_an_book_through_a_delete_request()
{

}

/**
 * @test
 */
public function
    it_can_sort_books_by_title_through_a_sort_query_parameter()
{

}

/**
 * @test
 */
public function
    it_can_sort_books_by_title_in_descending_order_through_a_sort_query_param
    ()
{

}

/**

```

```

    * @test
    */
    public function
        it_can_sort_books_by_multiple_attributes_through_a_sort_query_parameter
        ()
    {

    }

    /**
     * @test
     */
    public function
        it_can_sort_books_by_multiple_attributes_in_descending_order_through_a_
        ()
    {

    }

    /**
     * @test
     */
    public function
        it_can_paginate_books_through_a_page_query_parameter()
    {

    }

    /**
     * @test
     */
    public function
        it_can_paginate_books_through_a_page_query_parameter_and_show_different
        ()
    {

    }
}

```

That's quite a number of tests, but don't get discouraged and remember that

they are here to help us. The time spent writing all the tests will, in many cases, be much less time spent than trying to find a bug in production. Also, we can steal much more from our **AuthorsTest** than we have done already. We just want to do it test by test, so it's easier to manage.

### *The first test and resource setup*

Let's write the first test then. Don't forget to add the `@watch` annotation so we can run our test automatically, and if you haven't already, also to start the test watcher. Here, the concept is exactly the same as the first test in our **AuthorsTest**: we are testing that we can fetch a single book and that it is returned as a resource object. As mentioned, we will be breaking down our tests now and we have broken this test down to the following structure:

- 1. We set up our world
  - a. We need a book to exist to be able to fetch it
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint
  - b. We add the right **Accept** and **Content-Type** headers
- 3. We assert against the result that
  - a. We get a status code **200 OK** back
  - b. We get the correct response object back

If you feel lost, remember that we will be showing you our implementation, but also remember that you can look at the tests for authors. We have written our test like this — remember that it's ok if you are deviating in assertions, just as long as you are confident in your test:

```
/**  
 * @test
```

```

* @watch
*/
public function it_returns_an_book_as_a_resource_object()
{
    $book = factory(Book::class)->create();
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/books/1', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(200)
    ->assertJson([
        "data" => [
            "id" => '1',
            "type" => "books",
            "attributes" => [
                'title' => $book->title,
                'description' => $book->description,
                'publication_year' => $book->publication_year,
                'created_at' => $book->created_at->toJSON(),
                'updated_at' => $book->updated_at->toJSON(),
            ]
        ]
    ]);
}

```

Right now, our test is failing since we do not have a **Book** model just yet, so let's make it. Here, we take the same approach as in chapter 4, where we used an artisan command to make the model and also the controller, factory and migrations through a simple **-a** flag, like this:

```
php artisan make:model Book -a
```

Just like we did in chapter 4, don't forget to rename all the files and classes

from **Book** to **Books** except for the model so that we will end up with files named like this:

- **app/Book.php**
- **app/Http/Controllers/BooksController.php**
- **database/factories/BooksFactory.php**
- **database/migrations/xxxx\_xx\_xx\_XXXXXX\_create\_books\_table.php**

Then we need to setup our migrations. We already know which columns we want in our database so this will be quite straightforward. Open up the **database/migrations/xxxx\_xx\_xx\_XXXXXX\_create\_books\_table.php** file and add the following:

```
<?php

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

class CreateBooksTable extends Migration
{
    public function up()
    {
        Schema::create('books', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('title');
            $table->text('description');
            $table->string('publication_year');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('books');
    }
}
```

## BOOKS

```
}  
}
```

Then we can set up our **database/factories/BooksFactory.php** like this:

```
$factory->define(App\Book::class, function (Faker $faker) {  
    return [  
        'title' => $faker->name,  
        'description' => $faker->sentence,  
        'publication_year' => (string)$faker->year  
    ];  
});
```

Next, we need to import the **Book** model in our test like this (**note** that we are only showing a portion of the class to save space):

```
<?php  
  
namespace Tests\Feature;  
  
use App\Book;  
use App\User;  
use Illuminate\Foundation\Testing\DatabaseMigrations;  
use Laravel\Passport\Passport;  
use Tests\TestCase;  
  
class BooksTest extends TestCase  
{  
  
    ...
```

And our failing test should change from the missing **Book** model being the issue to the wrong status code being the issue. Let's do the implementation so

the test will pass, and in our new **app/Http/Controllers/BooksController.php** let's focus on the **show** method, which is currently looking like this:

```
public function show(Book $book)
{
    //
}
```

If you remember from chapter 4, we created a **AuthorsResource** class for transforming our Eloquent model to a resource object adhering to the conventions of the JSON:API specification. We need to do this again for our books resource, so get back into the terminal and run the following artisan command to create the resource:

```
php artisan make:resource BooksResource
```

This should create a new **app/Http/Resources/BooksResource.php** file. Let's jump into it and map the correct structure to our response object based on our model like this:

```
<?php

namespace App\Http\Resources;
use Illuminate\Http\Resources\Json\JsonResource;

class BooksResource extends JsonResource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     */
}
```



```

    * @return array
    */
    public function toArray($request)
    {
        return [
            'id' => (string)$this->id,
            'type' => 'books',
            'attributes' => [
                'title' => $this->title,
                'description' => $this->description,
                'publication_year' => $this->publication_year,
                'created_at' => $this->created_at,
                'updated_at' => $this->updated_at,
            ]
        ];
    }
}

```

Then, we can go right back into our controller and return our new resource in the show method like this:

```

public function show(Book $book)
{
    return new BooksResource($book);
}

```

Our test is still not passing because we are missing the routes that points to our controller, so let's go into the **routes/api.php** file and use the **Route::apiResource** method once again like this:

```

Route::middleware('auth:api')->prefix('v1')->group(function(){
    Route::get('/user', function (Request $request) {

```

```

        return $request->user();
    });

    // Authors
    Route::apiResource('authors', 'AuthorsController');

    // Books
    Route::apiResource('books', 'BooksController');
});

```

And our test should be green and passing. We now have most of the classes for our resource, except for a **ResourceCollection**, but we will create and implement this through the next test.

### *Implementing the Resource Collection*

Our next test is currently looking like this:

```

/**
 * @test
 */
public function
    it_returns_all_books_as_a_collection_of_resource_objects()
{
}

```

If we break down the structure again:

- 1. We set up our world
  - a. We need multiple books to exist to be able to fetch them
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint

- b. We add the right **Accept** and **Content-Type** headers
- 3. We assert against the result that
  - a. We get a status code **200 OK** back
  - b. We get the correct response object back

Note that for this test you can reuse some of the code from the previous test, especially the resource object structure in the **assertJson** method. Don't forget that we are testing for a collection, which means that we will need multiple resource objects in an array instead.

We have written this test like so:

```
/**
 * @test
 * @watch
 */
public function
  it_returns_all_books_as_a_collection_of_resource_objects()
{
  $user = factory(User::class)->create();
  Passport::actingAs($user);
  $books = factory(Book::class, 3)->create();

  $this->get('/api/v1/books', [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
  ])->assertStatus(200)->assertJson([
    "data" => [
      [
        "id" => '1',
        "type" => "books",
        "attributes" => [
          'title' => $books[0]->title,
          'description' => $books[0]->description,
          'publication_year' => $books[0]->
            publication_year,
        ]
      ]
    ]
  ]);
}
```

```

        'created_at' => $books[0]->created_at->toJSON(),
        'updated_at' => $books[0]->updated_at->toJSON(),
    ]
],
[
    "id" => '2',
    "type" => "books",
    "attributes" => [
        'title' => $books[1]->title,
        'description' => $books[1]->description,
        'publication_year' => $books[1]->
            publication_year,
        'created_at' => $books[1]->created_at->toJSON(),
        'updated_at' => $books[1]->updated_at->toJSON(),
    ]
],
[
    "id" => '3',
    "type" => "books",
    "attributes" => [
        'title' => $books[2]->title,
        'description' => $books[2]->description,
        'publication_year' => $books[2]->
            publication_year,
        'created_at' => $books[2]->created_at->toJSON(),
        'updated_at' => $books[2]->updated_at->toJSON(),
    ]
],
    ]
    });
}

```

Our test is failing, telling us that invalid JSON was returned from the route. We should fix that, but before we do, and since we know we will need it, let's create a **ResourceCollection** for our books resource in our terminal, through the artisan command we used earlier to make a **BooksResource** like this:

```
php artisan make:resource BooksCollection -c
```

Notice that we are using the **-c** flag this time, instead of relying on the name, which is a bit more consistent.

Let's open up the newly created **app/Http/Resources/BooksCollection.php** file and add the following:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class BooksCollection extends ResourceCollection
{
    public $collects = BooksResource::class;

    public function toArray($request)
    {
        return [
            'data' => $this->collection,
        ];
    }
}
```

Like before, we add the **data** top-level member and we do this as a preparation for what needs to be implemented later on. With our **BooksCollection** **ResourceCollection** created, we can continue to the implementation in the **app/Http/Controllers/BooksController.php**'s **index** method.

From the implementation of our authors resource, we know that we will be working with the **index** method a few times, especially when implementing

the sort and pagination features. So to do this the easiest way, we will handle these next. Right now, to just get the test to pass we will do the implementation like this:

```
public function index()
{
    $books = Book::all();
    return new BooksCollection($books);
}
```

The test passes so for now the implementation works as it should. Let's move on to the test for sorting next.

### *Sorting and pagination*

At the moment, our first sort test looks like this:

```
/**
 * @test
 */
public function
    it_can_sort_books_by_title_through_a_sort_query_parameter()
{

}
```

If we break it down:

- 1. We set up our world
  - a. We need multiple books, with specific titles to exist to be able to sort by a title
  - b. We need to be authenticated

- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint with the **sort** query parameter
  - b. We add the right **Accept** and **Content-Type** headers
- 3. We assert against the result that
  - a. We get a status code **200 OK** back
  - b. We get the correct response object back

To get a bit ahead here, you can copy most of the previous test, since we need a collection for sorting.

We have written this test like so:

```
/**
 * @test
 * @watch
 */
public function
    it_can_sort_books_by_title_through_a_sort_query_parameter()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $books = collect([
        'Building an API with Laravel',
        'Classes are our blueprints',
        'Adhering to the JSON:API Specification',
    ]->map(function($title){
        return factory(Book::class)->create([
            'title' => $title
        ]);
    }));

    $this->get('/api/v1/books?sort=title', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
```

```

])->assertStatus(200)->assertJson([
    "data" => [
        [
            "id" => '3',
            "type" => "books",
            "attributes" => [
                'title' => 'Adhering to the JSON:API
                    Specification',
                'description' => $books[2]->description,
                'publication_year' => $books[2]->
                    publication_year,
                'created_at' => $books[2]->created_at->toJSON(),
                'updated_at' => $books[2]->updated_at->toJSON(),
            ]
        ],
        [
            "id" => '1',
            "type" => "books",
            "attributes" => [
                'title' => 'Building an API with Laravel',
                'description' => $books[0]->description,
                'publication_year' => $books[0]->
                    publication_year,
                'created_at' => $books[0]->created_at->toJSON(),
                'updated_at' => $books[0]->updated_at->toJSON(),
            ]
        ],
        [
            "id" => '2',
            "type" => "books",
            "attributes" => [
                'title' => 'Classes are our blueprints',
                'description' => $books[1]->description,
                'publication_year' => $books[1]->
                    publication_year,
                'created_at' => $books[1]->created_at->toJSON(),
                'updated_at' => $books[1]->updated_at->toJSON(),
            ]
        ],
    ],
])

```



```
    });  
  
}
```

Like we did with authors, we are using a collection to create the various book titles, so that we can sort these. Then, we are arranging the response in the **assertJson** method so that it reflects the expected outcome of the sorting.

The test is failing right now, which is expected. Before we go and write the implementation, why don't we write the test for our pagination, so that we can implement both simultaneously and save some time.

Right now, our first pagination test looks like this:

```
/**  
 * @test  
 */  
public function  
    it_can_paginate_books_through_a_page_query_parameter()  
{  
  
}
```

If we break it down:

- 1. We set up our world
  - a. We need multiple books to be able to paginate
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint
  - b. We add a page[size] query parameter to tell how many books per page
  - c. We add a page[number] query parameter to tell which page we are on
  - d. We add the right Accept and Content-Type headers

- 3. We assert against the result that
  - a. We get a status code 200 OK back
  - b. We get the correct response object back
  - c. We get the correct links top-level member back
  - d. We get the correct link structure back

Again, you can reuse a lot of the contents of the **it\_returns\_all\_books\_as\_a\_collection\_of\_resource\_objects** test to not having to write as much code. This is also one of the benefits of following a specification: the reuse of test code is actually quite a lot, which can also make you work faster once you get into the flow.

Our implementation of this test is the following:

```
/**
 * @test
 * @watch
 */
public function
    it_can_paginate_books_through_a_page_query_parameter()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $books = factory(Book::class, 10)->create();

    $this->get('/api/v1/books?page[size]=5&page[number]=1', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(200)->assertJson([
        "data" => [
            [
                "id" => '1',
                "type" => "books",
                "attributes" => [
                    'title' => $books[0]->title,
```

## BOOKS

```

        'description' => $books[0]->description,
        'publication_year' => $books[0]->
            publication_year,
        'created_at' => $books[0]->created_at->toJSON(),
        'updated_at' => $books[0]->updated_at->toJSON(),
    ]
],
[
    "id" => '2',
    "type" => "books",
    "attributes" => [
        'title' => $books[1]->title,
        'description' => $books[1]->description,
        'publication_year' => $books[1]->
            publication_year,
        'created_at' => $books[1]->created_at->toJSON(),
        'updated_at' => $books[1]->updated_at->toJSON(),
    ]
],
[
    "id" => '3',
    "type" => "books",
    "attributes" => [
        'title' => $books[2]->title,
        'description' => $books[2]->description,
        'publication_year' => $books[2]->
            publication_year,
        'created_at' => $books[2]->created_at->toJSON(),
        'updated_at' => $books[2]->updated_at->toJSON(),
    ]
],
[
    "id" => '4',
    "type" => "books",
    "attributes" => [
        'title' => $books[3]->title,
        'description' => $books[3]->description,
        'publication_year' => $books[3]->
            publication_year,
        'created_at' => $books[3]->created_at->toJSON(),
    ]
],

```

```

        'updated_at' => $books[3]->updated_at->toJSON(),
    ],
    [
        "id" => '5',
        "type" => "books",
        "attributes" => [
            'title' => $books[4]->title,
            'description' => $books[4]->description,
            'publication_year' => $books[4]->
                publication_year,
            'created_at' => $books[4]->created_at->toJSON(),
            'updated_at' => $books[4]->updated_at->toJSON(),
        ]
    ],
],
'links' => [
    'first' => route('books.index', ['page[size]' => 5, '
        page[number]' => 1]),
    'last' => route('books.index', ['page[size]' => 5, 'page
        [number]' => 2]),
    'prev' => null,
    'next' => route('books.index', ['page[size]' => 5, 'page
        [number]' => 2]),
]
]);
}

```

Both tests are failing now, so let's do the implementation of both sorting and pagination so they will pass. Go back to the controller and add this in the **index** method:

```

public function index()
{
    $books = QueryBuilder::for(Book::class)->allowedSorts([
        'title',
        'publication_year',

```

```

        'created_at',
        'updated_at',
    ])->jsonPaginate();
    return new BooksCollection($books);
}

```

Again, we leverage Spatie's **Laravel Query Builder** and **JSON API Paginate** to do the hard work for us.

Now, let's implement the rest of both the sorting and pagination tests. We won't be breaking these down, since they all use the same concepts.

The tests for sorting are the following:

```

/**
 * @test
 * @watch
 */
public function
    it_can_sort_books_by_title_in_descending_order_through_a_sort_query_parametrized()
    {
        $user = factory(User::class)->create();
        Passport::actingAs($user);

        $books = collect([
            'Building an API with Laravel',
            'Classes are our blueprints',
            'Adhering to the JSON:API Specification',
        ]->map(function($title){
            return factory(Book::class)->create([
                'title' => $title
            ]);
        }));
    }

```

```

$this->get('/api/v1/books?sort=-title', [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])->assertStatus(200)->assertJson([
    "data" => [
        [
            "id" => '2',
            "type" => "books",
            "attributes" => [
                'title' => 'Classes are our blueprints',
                'description' => $books[1]->description,
                'publication_year' => $books[1]->
                    publication_year,
                'created_at' => $books[1]->created_at->toJSON(),
                'updated_at' => $books[1]->updated_at->toJSON(),
            ]
        ],
        [
            "id" => '1',
            "type" => "books",
            "attributes" => [
                'title' => 'Building an API with Laravel',
                'description' => $books[0]->description,
                'publication_year' => $books[0]->
                    publication_year,
                'created_at' => $books[0]->created_at->toJSON(),
                'updated_at' => $books[0]->updated_at->toJSON(),
            ]
        ],
        [
            "id" => '3',
            "type" => "books",
            "attributes" => [
                'title' => 'Adhering to the JSON:API
                    Specification',
                'description' => $books[2]->description,
                'publication_year' => $books[2]->
                    publication_year,
                'created_at' => $books[2]->created_at->toJSON(),
                'updated_at' => $books[2]->updated_at->toJSON(),
            ]
        ]
    ]
]);

```

```

        ],
    ],
]);

}

/**
 * @test
 * @watch
 */
public function
    it_can_sort_books_by_multiple_attributes_through_a_sort_query_parameter
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $books = collect([
        'Building an API with Laravel',
        'Classes are our blueprints',
        'Adhering to the JSON:API Specification',
    ]->map(function($title){

        if($title === 'Building an API with Laravel'){
            return factory(Book::class)->create([
                'title' => $title,
                'publication_year' => '2019',
            ]);
        }

        return factory(Book::class)->create([
            'title' => $title,
            'publication_year' => '2018',
        ]);
    }));

    $this->get('/api/v1/books?sort=publication_year,title', [
        'accept' => 'application/vnd.api+json',
    ]

```

```

        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(200)->assertJson([
        "data" => [
            [
                "id" => '3',
                "type" => "books",
                "attributes" => [
                    'title' => 'Adhering to the JSON:API
                        Specification',
                    'description' => $books[2]->description,
                    'publication_year' => $books[2]->
                        publication_year,
                    'created_at' => $books[2]->created_at->toJSON(),
                    'updated_at' => $books[2]->updated_at->toJSON(),
                ]
            ],
            [
                "id" => '2',
                "type" => "books",
                "attributes" => [
                    'title' => 'Classes are our blueprints',
                    'description' => $books[1]->description,
                    'publication_year' => $books[1]->
                        publication_year,
                    'created_at' => $books[1]->created_at->toJSON(),
                    'updated_at' => $books[1]->updated_at->toJSON(),
                ]
            ],
            [
                "id" => '1',
                "type" => "books",
                "attributes" => [
                    'title' => 'Building an API with Laravel',
                    'description' => $books[0]->description,
                    'publication_year' => $books[0]->
                        publication_year,
                    'created_at' => $books[0]->created_at->toJSON(),
                    'updated_at' => $books[0]->updated_at->toJSON(),
                ]
            ],
        ],
    ]);

```



```

    ]
  });
}

/**
 * @test
 * @watch
 */
public function
  it_can_sort_books_by_multiple_attributes_in_descending_order_through_a_sorting_method
  ()
{
  $user = factory(User::class)->create();
  Passport::actingAs($user);

  $books = collect([
    'Building an API with Laravel',
    'Classes are our blueprints',
    'Adhering to the JSON:API Specification',
  ]->map(function($title){

    if($title === 'Building an API with Laravel'){
      return factory(Book::class)->create([
        'title' => $title,
        'publication_year' => '2019',
      ]);
    }

    return factory(Book::class)->create([
      'title' => $title,
      'publication_year' => '2018',
    ]);

  }));

  $this->get('/api/v1/books?sort=-publication_year,title', [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
  ]->assertStatus(200)->assertJson([
    "data" => [

```

```

[
    "id" => '1',
    "type" => "books",
    "attributes" => [
        'title' => 'Building an API with Laravel',
        'description' => $books[0]->description,
        'publication_year' => $books[0]->
            publication_year,
        'created_at' => $books[0]->created_at->toJSON(),
        'updated_at' => $books[0]->updated_at->toJSON(),
    ]
],
[
    "id" => '3',
    "type" => "books",
    "attributes" => [
        'title' => 'Adhering to the JSON:API
            Specification',
        'description' => $books[2]->description,
        'publication_year' => $books[2]->
            publication_year,
        'created_at' => $books[2]->created_at->toJSON(),
        'updated_at' => $books[2]->updated_at->toJSON(),
    ]
],
[
    "id" => '2',
    "type" => "books",
    "attributes" => [
        'title' => 'Classes are our blueprints',
        'description' => $books[1]->description,
        'publication_year' => $books[1]->
            publication_year,
        'created_at' => $books[1]->created_at->toJSON(),
        'updated_at' => $books[1]->updated_at->toJSON(),
    ]
],
]
]);
}

```

The tests for pagination are the following:

```
/**
 * @test
 * @watch
 */
public function
    it_can_paginate_books_through_a_page_query_parameter_and_show_different_pa
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $books = factory(Book::class, 10)->create();

    $this->get('/api/v1/books?page[size]=5&page[number]=2', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(200)->assertJson([
        "data" => [
            [
                "id" => '6',
                "type" => "books",
                "attributes" => [
                    'title' => $books[5]->title,
                    'description' => $books[5]->description,
                    'publication_year' => $books[5]->
                        publication_year,
                    'created_at' => $books[5]->created_at->toJSON(),
                    'updated_at' => $books[5]->updated_at->toJSON(),
                ]
            ],
            [
                "id" => '7',
                "type" => "books",
                "attributes" => [
                    'title' => $books[6]->title,
                    'description' => $books[6]->description,
                    'publication_year' => $books[6]->
```

```

        publication_year,
        'created_at' => $books[6]->created_at->toJSON(),
        'updated_at' => $books[6]->updated_at->toJSON(),
    ]
],
[
    "id" => '8',
    "type" => "books",
    "attributes" => [
        'title' => $books[7]->title,
        'description' => $books[7]->description,
        'publication_year' => $books[7]->
            publication_year,
        'created_at' => $books[7]->created_at->toJSON(),
        'updated_at' => $books[7]->updated_at->toJSON(),
    ]
],
[
    "id" => '9',
    "type" => "books",
    "attributes" => [
        'title' => $books[8]->title,
        'description' => $books[8]->description,
        'publication_year' => $books[8]->
            publication_year,
        'created_at' => $books[8]->created_at->toJSON(),
        'updated_at' => $books[8]->updated_at->toJSON(),
    ]
],
[
    "id" => '10',
    "type" => "books",
    "attributes" => [
        'title' => $books[9]->title,
        'description' => $books[9]->description,
        'publication_year' => $books[9]->
            publication_year,
        'created_at' => $books[9]->created_at->toJSON(),
        'updated_at' => $books[9]->updated_at->toJSON(),
    ]
]
```

```

        ],
    ],
    'links' => [
        'first' => route('books.index', ['page[size]' => 5, 'page[number]' => 1]),
        'last' => route('books.index', ['page[size]' => 5, 'page[number]' => 2]),
        'prev' => route('books.index', ['page[size]' => 5, 'page[number]' => 1]),
        'next' => null,
    ]
  });
}

```

This is it for both our **BooksCollection** ResourceCollection, the sorting functionality, and pagination functionality. Now, we can move on to the implementation of creating books.

### *Creating and validating books*

Taking a look at our test for creating books, it currently looks like this:

```

/**
 * @test
 */
public function it_can_create_an_book_from_a_resource_object()
{

}

```

To break this down:

- 1. We set up our world
- a. We need to be authenticated

- 2. We run the code we are testing here
  - a. We make a POST request to the right API endpoint
  - b. We add the right **Accept** and **Content-Type** headers
  - c. We add a valid request object for the creation of our book
- 3. We assert against the result that
  - a. We get a status code **201 Created** back
  - b. We get the correct response object back

For creation of resources, it is a good idea to use assertions against the database to see that the creation has taken effect and we can see the new resource reflected in a row in the database. From now on, we will add this as the fourth part of our structure like this:

- 4. We assert against the database that
  - a. The data has been saved

We have written this test like so:

```
/**
 * @test
 * @watch
 */
public function it_can_create_an_book_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/books', [
        'data' => [
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
            ]
        ]
    ]);
}
```

```

        'publication_year' => '2019',
    ]
]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])
->assertStatus(201)
->assertJson([
    "data" => [
        "id" => '1',
        "type" => "books",
        "attributes" => [
            'title' => 'Building an API with Laravel',
            'description' => 'A book about API development',
            'publication_year' => '2019',
            'created_at' => now()->setMilliseconds(0)->
                toJSON(),
            'updated_at' => now() ->setMilliseconds(0)->
                toJSON(),
        ]
    ]
])->assertHeader('Location', url('/api/v1/books/1'));

$this->assertDatabaseHas('books', [
    'id' => 1,
    'title' => 'Building an API with Laravel',
    'description' => 'A book about API development',
    'publication_year' => '2019',
]);
}

```

Right now, our test is failing so let's implement the creation of our book in the **app/Http/Controller/BooksController.php** store method:

```

public function store(Request $request)
{
    $book = Book::create([
        'title' => $request->input('data.attributes.title'),
        'description' => $request->input('data.attributes.
            description'),
        'publication_year' => $request->input('data.attributes.
            publication_year'),
    ]);
    return (new BooksResource($book))
        ->response()
        ->header('Location', route('books.show', [
            'book' => $book,
        ]));
}

```

The concept here should be familiar, since it's the exact same as in the **AuthorsController**. We get the data from the request using the **input** method, and we leverage the **create** static method on our model to do the entire creation of our book. We then use our **BooksResource** to return the book as a resource object adhering to the JSON:API specification.

Our test should be passing and green. Of course, this is again a very naive approach, so let's implement validation so we know that we have something that catches mistakes.

Right now, our first test for validating the creation of a book looks like this:

```

/**
 * @test
 */
public function
    it_validates_that_the_type_member_is_given_when_creating_an_book
    ()

```



```
{
}
```

Let's break down the structure:

- 1. We set up our world
  - a. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a POST request to the right API endpoint
  - b. We add the right **Accept** and **Content-Type** headers
  - c. We need an invalid request object for the creation of our book by either leaving out a member or setting it with the wrong datatype.
- 3. We assert against the result that
  - a. We get a status code **422** back
  - b. We get the correct error object back

We have written the test this way:

```
/**
 * @test
 * @watch
 */
public function
  it_validates_that_the_type_member_is_given_when_creating_an_book
  ()
{
  $user = factory(User::class)->create();
  Passport::actingAs($user);

  $this->postJson('/api/v1/books', [
    'data' => [
      'type' => '',
      'attributes' => [
```

```

        'title' => 'Building an API with Laravel',
        'description' => 'A book about API development',
        'publication_year' => '2019',
    ]
]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])
->assertStatus(422)
->assertJson([
    'errors' => [
        [
            'title' => 'Validation Error',
            'details' => 'The data.type field is required.',
            'source' => [
                'pointer' => '/data/type',
            ]
        ]
    ]
]);

$this->assertDatabaseMissing('books', [
    'id' => 1,
    'title' => 'Building an API with Laravel',
    'description' => 'A book about API development',
    'publication_year' => '2019',
]);
}

```

At the moment, our test is failing, which is because we don't have anything to validate against. Let's create a new **Request** class, which can be done easily through an artisan command like this:

```
php artisan make:request CreateBookRequest
```

Next, we need to define the rules of **Request** so let's open **app/Http/Requests/CreateBookRequest.php** and first return **true** in the **authorize** method and next focus on the rules of the **rules** method.

Here, we define the following rules for both validating our request document but also the attributes:

```
public function rules()
{
    return [
        'data' => 'required|array',
        'data.type' => 'required|in:books',
        'data.attributes' => 'required|array',
        'data.attributes.title' => 'required|string',
        'data.attributes.description' => 'required|string',
        'data.attributes.publication_year' => 'required|string',
    ];
}
```

The concepts here are very similar to what we have been through with authors, just with a few more attributes.

Next, we need to add the **CreateBookRequest** to our **store** method in our **app/Http/Controllers/BooksController.php** so that we are actually validating against the rules like this:

```
public function store(CreateBookRequest $request)
{
    $book = Book::create([
        'title' => $request->input('data.attributes.title'),
        'description' => $request->input('data.attributes.
            description'),
        'publication_year' => $request->input('data.attributes.
```

```

        'publication_year'),
    ]);
    return (new BooksResource($book))
        ->response()
        ->header('Location', route('books.show', [
            'book' => $book,
        ]));
}

```

This will make our test green and passing.

For the next validation tests, the structure is the same and you can reuse a lot of your test code:

- 1. We set up our world
  - a. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a POST request to the right API endpoint
  - b. We add the right **Accept** and **Content-Type** headers
  - c. We need an invalid request object for the creation of our book by either leaving out a member or setting it with the wrong datatype.
- 3. We assert against the result that
  - a. We get a status code **422** back
  - b. We get the correct error object back

We will let you work through these on your own, and the code will be below if you need guidance:

```

/**
 * @test
 * @watch
 */
public function

```

```

        it_validates_that_the_type_member_has_the_value_of_books_when_creating_an_
        ()
    {

        $user = factory(User::class)->create();
        Passport::actingAs($user);

        $this->postJson('/api/v1/books', [
            'data' => [
                'type' => 'book',
                'attributes' => [
                    'title' => 'Building an API with Laravel',
                    'description' => 'A book about API development',
                    'publication_year' => '2019',
                ]
            ]
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ])->assertStatus(422)
        ->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The selected data.type is invalid
                        .',
                    'source' => [
                        'pointer' => '/data/type',
                    ]
                ]
            ]
        ]);

        $this->assertDatabaseMissing('books', [
            'id' => 1,
            'title' => 'Building an API with Laravel',
            'description' => 'A book about API development',
            'publication_year' => '2019',
        ]);
    }
}

```

```

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_attributes_member_has_been_given_when_creating_an_book
    (
    {
        $user = factory(User::class)->create();
        Passport::actingAs($user);

        $this->postJson('/api/v1/books', [
            'data' => [
                'type' => 'books',
            ]
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ]->assertStatus(422)
            ->assertJson([
                'errors' => [
                    [
                        'title' => 'Validation Error',
                        'details' => 'The data.attributes field is
                            required.',
                        'source' => [
                            'pointer' => '/data/attributes',
                        ]
                    ]
                ]
            ]]);
    }

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_attributes_member_is_an_object_given_when_creating_an_

```

```

    ()
  {

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/books', [
      'data' => [
        'type' => 'books',
        'attributes' => 'this is not an object'
      ]
    ], [
      'accept' => 'application/vnd.api+json',
      'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(422)
    ->assertJson([
      'errors' => [
        [
          'title' => 'Validation Error',
          'details' => 'The data.attributes must be an
            array.',
          'source' => [
            'pointer' => '/data/attributes',
          ]
        ]
      ]
    ]);
  }

  /**
   * @test
   * @watch
   */
  public function
    it_validates_that_a_title_attribute_is_given_when_creating_an_book
    ()
  {

    $user = factory(User::class)->create();

```

```

Passport::actingAs($user);

$this->postJson('/api/v1/books', [
    'data' => [
        'type' => 'books',
        'attributes' => [
            'description' => 'A book about API development',
            'publication_year' => '2019',
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(422)
    ->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The data.attributes.title field is
                    required.',
                'source' => [
                    'pointer' => '/data/attributes/title',
                ]
            ]
        ]
    ]);
}

/**
 * @test
 * @watch
 */
public function
    it_validates_that_a_title_attribute_is_a_string_when_creating_an_book
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

```



```

$this->postJson('/api/v1/books', [
  'data' => [
    'type' => 'books',
    'attributes' => [
      'title' => 42,
      'description' => 'A book about API development',
      'publication_year' => '2019',
    ]
  ]
], [
  'accept' => 'application/vnd.api+json',
  'content-type' => 'application/vnd.api+json',
])
->assertStatus(422)
->assertJson([
  'errors' => [
    [
      'title' => 'Validation Error',
      'details' => 'The data.attributes.title must be
        a string.',
      'source' => [
        'pointer' => '/data/attributes/title',
      ]
    ]
  ]
]);
}

/**
 * @test
 * @watch
 */
public function
  it_validates_that_a_description_attribute_is_given_when_creating_an_book
  ()
{

  $user = factory(User::class)->create();
  Passport::actingAs($user);

```

```

$this->postJson('/api/v1/books', [
    'data' => [
        'type' => 'books',
        'attributes' => [
            'title' => 'Building an API with Laravel',
            'publication_year' => '2019',
        ]
    ]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])
->assertStatus(422)
->assertJson([
    'errors' => [
        [
            'title' => 'Validation Error',
            'details' => 'The data.attributes.description
                        field is required.',
            'source' => [
                'pointer' => '/data/attributes/description',
            ]
        ]
    ]
]);
}

/**
 * @test
 * @watch
 */
public function
    it_validates_that_a_description_attribute_is_a_string_when_creating_an_book
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/books', [
        'data' => [

```

```

        'type' => 'books',
        'attributes' => [
            'title' => 'Building an API with Laravel',
            'description' => 42,
            'publication_year' => '2019',
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
->assertStatus(422)
->assertJson([
    'errors' => [
        [
            'title' => 'Validation Error',
            'details' => 'The data.attributes.description
                must be a string.',
            'source' => [
                'pointer' => '/data/attributes/description',
            ]
        ]
    ]
]);
}

/**
 * @test
 * @watch
 */
public function
    it_validates_that_a_publication_year_attribute_is_given_when_creating_an_b
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/books', [
        'data' => [
            'type' => 'books',

```

```

        'attributes' => [
            'title' => 'Building an API with Laravel',
            'description' => 'A book about API development',
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
->assertStatus(422)
->assertJson([
    'errors' => [
        [
            'title' => 'Validation Error',
            'details' => 'The data.attributes.publication
                year field is required.',
            'source' => [
                'pointer' => '/data/attributes/
                    publication_year',
            ]
        ]
    ]
]);
}

/**
 * @test
 * @watch
 */
public function
    it_validates_that_a_publication_year_attribute_is_a_string_when_creating_an_
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/books', [
        'data' => [
            'type' => 'books',
            'attributes' => [

```

```

        'title' => 'Building an API with Laravel',
        'description' => 'A book about API development',
        'publication_year' => 2019,
    ]
]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])
->assertStatus(422)
->assertJson([
    'errors' => [
        [
            'title' => 'Validation Error',
            'details' => 'The data.attributes.publication
                year must be a string.',
            'source' => [
                'pointer' => '/data/attributes/
                    publication_year',
            ]
        ]
    ]
]);
}

```

All of our tests should be green and passing now, so let's move on to updating books.

### *Updating and validating books*

Take a look at our test for updating a book resource in our tests/Feature/Book-sTest.php which currently looks like this:

```

/**
 * @test

```

```
*/
public function it_can_update_an_book_from_a_resource_object()
{

}
```

This has a bit more to the setup of our world, so let's break it down:

- 1. We set up our world
  - a. We need our book to be created so we can update it with new data
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a PATCH request to the right API endpoint
  - b. We add the right **Accept** and **Content-Type** headers
  - c. We need a valid request object for the update of our book
- 3. We assert against the result that
  - a. We get a status code **200 OK** back
  - b. We get the correct resource object back
- 4. We assert against the database that
  - a. The data has been updated

The test we have written looks like this:

```
/**
 * @test
 * @watch
 */
public function it_can_update_an_book_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $book = factory(Book::class)->create();
```

```

$this->patchJson('/api/v1/books/1', [
    'data' => [
        'id' => '1',
        'type' => 'books',
        'attributes' => [
            'title' => 'Building an API with Laravel',
            'description' => 'A book about API development',
            'publication_year' => '2019',
        ]
    ]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])
->assertStatus(200)
->assertJson([
    "data" => [
        "id" => '1',
        "type" => "books",
        "attributes" => [
            'title' => 'Building an API with Laravel',
            'description' => 'A book about API development',
            'publication_year' => '2019',
            'created_at' => now()->setMilliseconds(0)->
                toJSON(),
            'updated_at' => now() ->setMilliseconds(0)->
                toJSON(),
        ]
    ]
]);

$this->assertDatabaseHas('books', [
    'id' => 1,
    'title' => 'Building an API with Laravel',
    'description' => 'A book about API development',
    'publication_year' => '2019',
]);
}

```

Our test is failing because we haven't implemented anything in the controller

yet, so jump over to the `app/Http/Requests/UpdateBookRequest.php` file and add the following:

```
public function update(Request $request, Book $book)
{
    $book->update($request->input('data.attributes'));
    return new BooksResource($book);
}
```

This will make our test green and passing.

If we take a look at our first test for validation update request, it looks like this:

```
/**
 * @test
 */
public function
    it_validates_that_an_id_member_is_given_when Updating_a_book()
{

}
```

When breaking this test down:

- 1. We set up our world
  - a. We need our book to be created so we can update it with new data
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a PATCH request to the right API endpoint
  - b. We add the right Accept and Content-Type headers
  - c. We need an invalid request object for the creation of our book by either



leaving out a member or setting it with the wrong datatype.

- 3. We assert against the result that
  - a. We get a status code 422 back
  - b. We get the correct error object back

This can be written like so:

```
/**
 * @test
 * @watch
 */
public function
    it_validates_that_an_id_member_is_given_when Updating_an_book()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(422)
    ->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
```

```

                'details' => 'The data.id field is required.',
                'source' => [
                    'pointer' => '/data/id',
                ]
            ]
        ]
    });

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

```

This test is failing at the moment, so we need to go through the same steps as with creation of books and make a Request we can use for validation. So get into our terminal and run the following artisan command:

```
php artisan make:request UpdateBookRequest
```

Like before, we will jump into the newly created **app/Http/Requests/Update-BookRequest.php** file and return **true** in the **authorize** method and return an array with the following **rules** in the rules method:

```

public function rules()
{
    return [
        'data' => 'required|array',
        'data.id' => 'required|string',
        'data.type' => 'required|in:books',
        'data.attributes' => 'required|array',
        'data.attributes.title' => 'sometimes|required|string',
        'data.attributes.description' => 'sometimes|required|string'
    ];
}

```

```

        'data.attributes.publication_year' => 'sometimes|required|
        string',
    ];
}

```

Like with authors, we add the **sometimes** rule in front of our attributes, since it is optional if these should be updated. The nice thing about the **sometimes** rule, is that it will validate the attribute if it is present in the request.

Next, we should add the **UpdateBookRequest** to our **update** method in our **app/Http/Controllers/BooksController.php** like this:

```

public function update(UpdateBookRequest $request, Book $book)
{
    $book->update($request->input('data.attributes'));
    return new BooksResource($book);
}

```

This will make our test green and passing.

Like with the creation of books, the concept of the validation tests are basically the same, so we will let you work on your own with the rest. We will post the code here, but try to work on your own first.

```

/**
 * @test
 * @watch
 */
public function
    it_validates_that_an_id_member_is_a_string_when Updating_an_book

```

```

    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'id' => 1,
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(422)
    ->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The data.id must be a string.',
                'source' => [
                    'pointer' => '/data/id',
                ]
            ]
        ]
    ]);

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

/**

```

```

* @test
* @watch
*/
public function
    it_validates_that_the_type_member_is_given_when Updating_an_book
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'id' => '1',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(422)
    ->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The data.type field is required.',
                'source' => [
                    'pointer' => '/data/type',
                ]
            ]
        ]
    ]));

    $this->assertDatabaseHas('books', [
        'id' => 1,
        'title' => $book->title,
    ]

```

```

    });
}

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_type_member_has_the_value_of_books_when_updating_an_book()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'id' => '1',
            'type' => 'booo',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(422)
    ->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The selected data.type is invalid',
                'source' => [
                    'pointer' => '/data/type',
                ]
            ]
        ]
    ])
}

```

```

        ]
    });

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

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_attributes_member_has_been_given_when_updating_an_book()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'id' => '1',
            'type' => 'books',
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(422)
    ->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The data.attributes field is
                    required.',
                'source' => [
                    'pointer' => '/data/attributes',
                ]
            ]
        ]
    ]);
}

```

```

        ]
    ]
});

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

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_attributes_member_is_an_object_given_when_updating_an_
        ()
    {
        $user = factory(User::class)->create();
        Passport::actingAs($user);
        $book = factory(Book::class)->create();

        $this->patchJson('/api/v1/books/1', [
            'data' => [
                'id' => '1',
                'type' => 'books',
                'attributes' => 'this is not an object'
            ]
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ])
        ->assertStatus(422)
        ->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The data.attributes must be an
                        array.',
                    'source' => [

```



```

        'pointer' => '/data/attributes',
    ]
  ]
  });

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

/**
 * @test
 * @watch
 */
public function
    it_validates_that_a_title_attribute_is_a_string_when Updating_an_book
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'id' => '1',
            'type' => 'books',
            'attributes' => [
                'title' => 42,
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(422)
    ->assertJson([
        'errors' => [
            [

```

```

        'title' => 'Validation Error',
        'details' => 'The data.attributes.title must be
            a string.',
        'source' => [
            'pointer' => '/data/attributes/title',
        ]
    ]
}

]);

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

/**
 * @test
 * @watch
 */
public function
    it_validates_that_a_description_attribute_is_a_string_when Updating_an_book
    ()
    {
        $user = factory(User::class)->create();
        Passport::actingAs($user);
        $book = factory(Book::class)->create();

        $this->patchJson('/api/v1/books/1', [
            'data' => [
                'id' => '1',
                'type' => 'books',
                'attributes' => [
                    'description' => 42,
                ]
            ]
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',

```

```

    })
    ->assertStatus(422)
    ->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The data.attributes.description
                    must be a string.',
                'source' => [
                    'pointer' => '/data/attributes/description',
                ]
            ]
        ]
    ]);

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

/**
 * @test
 * @watch
 */
public function
    it_validates_that_a_publication_year_attribute_is_a_string_when_updating_a
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'id' => '1',
            'type' => 'books',
            'attributes' => [
                'publication_year' => 2019,
            ]
        ]
    ]);
}

```

```

    ]
  ], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
  ])
  ->assertStatus(422)
  ->assertJson([
    'errors' => [
      [
        'title' => 'Validation Error',
        'details' => 'The data.attributes.publication
          year must be a string.',
        'source' => [
          'pointer' => '/data/attributes/
            publication_year',
        ]
      ]
    ]
  ]);

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

```

All the tests should be passing now. This is it for the update of our books resource, so let's move on to deletion, which fortunately is a very easy one.

### *Deleting books*

In our test class, the last test we need is our test for deleting books, which looks like this:

```
/**
 * @test
 */
public function it_can_delete_an_book_through_a_delete_request()
{

}
```

Breaking this down:

- 1. We set up our world
  - a. We need our book to be created so we can delete it
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a DELETE request to the right API endpoint
  - b. We add the right **Accept** and **Content-Type** headers
- 3. We assert against the result that
  - a. We get a status code **204 No Content** back

This can be written like this:

```
/**
 * @test
 * @watch
 */
public function it_can_delete_an_book_through_a_delete_request()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->delete('/api/v1/books/1', [], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]
    );
}
```

```

    ])->assertStatus(204);

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

```

And to make this test pass, we implement the following in our controllers **destroy** method:

```

public function destroy(Book $book)
{
    $book->delete();
    return response(null, 204);
}

```

All of our tests are now passing and this is actually it for the basic API implementation of the books resource. Now, we are at the same point as with our authors resource. We need to implement the relationship part and also make our relationship adhere to the JSON:API specification.

## Relationships

It is finally time to look at relationships — a very essential part of our journey into adopting the JSON:API specification. After this, we have enough knowledge to build the rest of our API adhering to the conventions of the specification and we can focus a little bit more on our application and code.

### *Setting up the first test*

Like anything else, we start with our test and in this case we want to create a new test file. We could have kept everything inside our **tests/Feature/BooksTest.php**, but let's create a new **tests/Feature/BooksRelationshipsTest.php** file so we have a nice separation of the concepts. This means that our **test/Feature/BooksTest.php** will contain tests for the basic API implementations and our new **test/FeatureBooksRelationshipsTest.php** will contain our API relationships implementations for our books resource. This will also keep our **test/Feature/BooksTest.php** file from getting any longer.

After you have created the file, don't forget to extend the **TestCase** class that ships with Laravel, as well as use the **DatabaseMigration** trait. Also remember to import all of the classes.

Then you should add the first test and name it: **it\_returns\_a\_relationship\_to\_authors\_adhering\_to\_json\_api\_spec** like this:

```
<?php

namespace Tests\Feature;

use App\Author;
use App\Book;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Tests\TestCase;

class BooksRelationshipsTest extends TestCase
{
    use DatabaseMigrations;

    /**
     * @test
     */
```

```

    public function
        it_returns_a_relationship_to_authors_adhering_to_json_api_spec
        ()
    {

    }
}

```

Before we move on, let's break down what we actually want to test:

- 1. We set up our world
  - a. We need a book to be able to get it through our API
  - b. We need a couple of authors to exist to be able to add them as author for our book
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint
  - b. We add the right Accept and Content-Type headers
- 3. We assert against the result that
  - a. We get a status code 200 OK back
  - b. We see the relationships member
  - c. We see the authors relationship
  - d. We see the links member inside the relationship
  - e. We see the resource linkage inside the relationship
  - f. We see the resource identifier objects for the authors

This will be a test that covers almost all the implementations we need to make, all the way from the general relationship methods in our models, to the migration of the pivot table, and to the changes in our resource objects we might have to make. If you feel comfortable enough to write the test on your own, we encourage you to do so. If you feel a little lost, remember that we are actually just building on what we have already, so this test will pretty much reflect the first test of our **tests/Feature/BooksTest.php**. Again, we



could have built the relationship part onto our tests in that test file, but later on there will be some specific implementations for relationships only, which is why we want to keep them separated in a file for each.

If it's too overwhelming, we totally understand. It is a lot of information at once, so let's just take a look at what we have written here:

```
/**
 * @test
 * @watch
 */
public function
    it_returns_a_relationship_to_authors_adhering_to_json_api_spec
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 3)->create();
    $book->authors()->sync($authors->only('id'));

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/books/1', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(200)
    ->assertJson([
        'data' => [
            'id' => '1',
            'type' => 'books',
            'relationships' => [
                'authors' => [
                    'links' => [
                        'self' => route(
                            'books.relationships.authors',
                            ['id' => $book->id]
                        )
                    ]
                ]
            ]
        ]
    ])
```

```

        ),
        'related' => route(
            'books.authors',
            ['id' => $book->id]
        ),
    ],
    'data' => [
        [
            'id' => $authors->get(0)->id,
            'type' => 'authors'
        ],
        [
            'id' => $authors->get(1)->id,
            'type' => 'authors'
        ]
    ]
    ]
    ]
    ];
}

```

There are quite a bit of assertions here, but we wanted to add the most important parts of a relationship to start with, so that we can let our test drive our implementation. You might be wondering why we aren't making any assertions for the **attributes** member, but remember that we have already done those assertions on the exact same route in the **tests/Feature/Book-sTest.php**, so there is no reason to do them again and vice versa, once we have implemented our relationships.

At the moment, our test is failing with: *LogicException: App\Book::authors must return a relationship instance.*

We can fix that very easily by implementing the **authors** method on our **Book** model. In fact, why don't we implement the relationship on both the **Author** and **Book** models right away? Since we have a **many-to-many** relationship,

meaning that an author can have written many books, but a book can also have been written by many authors, we need to define our relationships on our models with the following: **app/Book.php**:

```
public function authors()
{
    return $this->belongsToMany(Author::class);
}
```

**app/Author.php**:

```
public function books()
{
    return $this->belongsToMany(Book::class);
}
```

Now our test is failing with a new error: *Illuminate\Database\QueryException: SQLSTATE[42S02]: Base table or view not found: 1146 Table 'annas\_bookstore\_testing.author\_book' doesn't exist...*

The great thing about this error is that it's actually telling us what to do next, we have made the relationships on our models but we haven't made the migration for a pivot table, which is needed for a **many-to-many** relationship. What's even better is that the error tells us exactly what we need to call the table, so that we can easily use this to create our migration through an artisan command like this:

```
php artisan make:migration create_author_book_table --create=author_book
```

Let's open the newly created **database/migrations/xxxx\_xx\_xx\_xxxxxx\_create\_author\_book\_table.php** file and start filling in the needed columns:

```
<?php

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

class CreateAuthorBookTable extends Migration
{
    public function up()
    {
        Schema::create('author_book', function (Blueprint $table) {
            $table->unsignedBigInteger('author_id');
            $table->foreign('author_id')
                ->references('id')
                ->on('authors')
                ->onDelete('cascade');
            $table->unsignedBigInteger('book_id');
            $table->foreign('book_id')
                ->references('id')
                ->on('books')
                ->onDelete('cascade');
        });
    }

    public function down()
    {
        Schema::dropIfExists('author_book');
    }
}
```

We create the two needed **ID**'s and add a foreign constraint on them, making sure that when either a book or an author is deleted, if one of these have a relationship, the relationship will be deleted automatically. This is more of a MySQL feature than a Laravel feature, but Laravel makes it easy to set up through a couple of chained methods, rather than having to write it out in SQL.

Now, our failing test has changed once again into the following error: *InvalidArgumentException: Route [books.relationships.authors] not defined.*

This means that we are actually starting on the assertions, which is a great thing. Let's create some temporary routes like this in our **routes/api.php** file, right under our **Books::apiResource:**

```
Route::get('books/{book}/relationships/authors', function(){
    return true;
})->name('books.relationships.authors');

Route::get('books/{book}/authors', function(){
    return true;
})->name('books.authors');
```

Now, our test is failing with new output again — this time it is the JSON response, where we can see that it cannot find anything about the relationship in the returned JSON. The reason for this is that we haven't implemented it in our resource yet, so let's do that now — with a bit of a naive implementation just to get our test to pass. In our **app/Http/Resources/BooksResource.php**, we add the following to the **toArray** method:

```
public function toArray($request)
{
```

```

return [
    'id' => (string)$this->id,
    'type' => 'books',
    'attributes' => [
        'title' => $this->title,
        'description' => $this->description,
        'publication_year' => $this->publication_year,
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ],
    'relationships' => [
        'authors' => [
            'links' => [
                'self' => route(
                    'books.relationships.authors',
                    ['id' => $this->id]
                ),
                'related' => route(
                    'books.authors',
                    ['id' => $this->id]
                ),
            ],
            'data' => $this->authors->map(function($author){
                return [
                    'id' => $author->id,
                    'type' => 'authors',
                ];
            })
        ],
    ],
];
}

```

We add the **relationships** member and add the **authors** relationship to the object. Within this relationship, we add the linkage following the relationship linkage from the JSON:API specification. We add our resource linkage through the **data** member and create a quick implementation through a Collection map method, returning the **id** of each model and a hardcoded **type**. Our test is now green and passing.

## *Refactoring Resource Identifier Objects*

As of right now, the implementation of our resource identifier objects is pretty ugly. Even though we love Laravel's Collections, this code isn't actually telling us what happens. It would be much better with a dedicated object that could give a bit more explanation.

So let's make a new resource and call it **AuthorsIdentifierResource**, which will be the blueprint for our resource identifier object for authors:

```
php artisan make:resource AuthorsIdentifierResource
```

In our newly created **app/Http/Resources/AuthorsIdentifierResource.php** file, let's add the following to the **toArray** method:

```
public function toArray($request)
{
    return [
        'id' => (string)$this->id,
        'type' => 'authors',
    ];
}
```

By using this resource object, we will only return the attributes needed for a resource identifier object. Since Laravel can make collections of single resource files, we can make use of this to create a collection of **AuthorIdentifierResources**. Let's go back to our **app/Http/Resources/BooksResource.php** file and change the relationship linkage data member to this:

```

public function toArray($request)
{
    return [
        'id' => (string)$this->id,
        'type' => 'books',
        'attributes' => [
            'title' => $this->title,
            'description' => $this->description,
            'publication_year' => $this->publication_year,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ],
        'relationships' => [
            'authors' => [
                'links' => [
                    'self' => route(
                        'books.relationships.authors',
                        ['id' => $this->id]
                    ),
                    'related' => route(
                        'books.authors',
                        ['id' => $this->id]
                    ),
                ],
                'data' => AuthorIdentifierResource::collection(
                    $this->authors),
            ],
        ],
    ];
}

```

Now it's much more readable: we can clearly read that we get a collection of **AuthorIdentifierResources** back.

What we did here is called a **refactoring**: we rewrite existing code so it becomes simpler and easier to understand without changing the functionality. This is also one of the benefits of having tests to back us up: we can begin refactoring our code while ensuring that we aren't breaking anything while doing so.



Now that we have implemented a resource for resource identifier objects for authors, we should move on to our relationship link, since we will be able to implement this now.

### *Relationship links*

If you recall the chapter about the JSON:API specification, we talked about relationship links which we can use to modify the relationship itself. This means that we can add and remove authors to our books however we like, without deleting any books or authors. We already have a test class for this, our **tests/Feature/BooksRelationshipsTest.php** file, but before we go in and write the test, let's break down what we need to do:

- 1. We set up our world
  - a. We need a book to be able to fetch it through our API
  - b. We need a couple of authors to exist to be able to fetch them as author for our book
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint
  - b. We add the right Accept and Content-Type headers
- 3. We assert against the result that
  - a. We get a status code 200 OK back
  - b. We get to see the authors as proper resource identifier objects

A tip when writing this test is to use the **withoutExceptionHandling** helper method in the beginning, so that you see the exact exception being thrown. Later on, you will need to remove the call to this method, so that your test will pass. You use this method like this:

```
/**
 * @test
 * @watch
 */
public function test_case()
{
    $this->withoutExceptionHandling();
}
```

Based on the breakdown of our test, we have written the following:

```
/**
 * @test
 * @watch
 */
public function
    a_relationship_link_to_authors_returns_all_related_authors_as_resource_id_ob
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 3)->create();
    $book->authors()->sync($authors->pluck('id'));

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/books/1/relationships/authors', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200)
        ->assertJson([
            'data' => [
                [
                    'id' => '1',
                    'type' => 'authors',
                ],
            ],
        ])
```

## BOOKS

```
        [
            'id' => '2',
            'type' => 'authors',
        ],
        [
            'id' => '3',
            'type' => 'authors',
        ],
    ]
});
}
```

This test is failing at the moment because we are returning a boolean from our **routes/api.php** file for the route, so let's create a new controller first and then change the route. Let's create the controller through the artisan command like this:

```
php artisan make:controller BooksAuthorsRelationshipsController
```

Then back in the **routes/api.php** file we change the route to this:

```
Route::get('books/{book}/relationships/authors', '
    BooksAuthorsRelationshipsController@index')->name('books.
    relationships.authors');
```

Let's then jump into our newly created **app/Http/Controllers/BooksAuthorsRelationshipsController.php** file and create a new **index** method like this:

```
public function index(Book $book)
{
    return AuthorsIdentifierResource::collection($book->authors);
}
```

We add our **Book** model as an argument to the **index** method, since we need to know which book we are accessing the relationship to authors on. Then like before, we return a collection of `AuthorsIdentifierResource`. Our test should be green and passing now.

### *Modifying relationships*

Next, we need to be able to modify our relationships through our relationship link. If you look back at the chapter about the JSON:API specification, we made a comparison between updating a **one-to-many** or **many-to-many** relationship and the **sync** method on an Eloquent relationship, where everything is first removed and then the given IDs are being added as new relationships. Since we have a **many-to-many** relationship, we can actually leverage the **sync** method for the next part, we are about to implement.

Before we do that, we should break down what we want to happen:

- 1. We set up our world
  - a. We need a book to be able to fetch it through our API
  - b. We need a 10 authors to exist to be able to add some as author for our book
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a PATCH request to the right API endpoint
  - b. We give the resource identifier objects for the authors we want to add
  - c. We add the right Accept and Content-Type headers
- 3. We assert against the result that
  - a. We get a status code 204 No Content back

- 4. We assert against the database that
  - a. We see each author has a relationship to the book

Before we move on to writing the tests, we want to explain the **204 No Content** status code here. In this case, we are only making changes to our **author\_book** pivot table and since it does not have any timestamps, we are not modifying anything besides what is given in the PATCH request and therefore it is enough with an empty response containing a **204 No Content** status code.

Based on that, our test looks like this:

```
/**
 * @test
 * @watch
 */
public function
    it_can_modify_relationships_to_authors_and_add_new_relationships
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 10)->create();

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->patchJson('/api/v1/books/1/relationships/authors',[
        'data' => [
            [
                'id' => '5',
                'type' => 'authors',
            ],
            [
                'id' => '6',
                'type' => 'authors',
            ]
        ]
    ])
```

```

], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])->assertStatus(204);

$this->assertDatabaseHas('author_book', [
    'author_id' => 5,
    'book_id' => 1,
])->assertDatabaseHas('author_book', [
    'author_id' => 6,
    'book_id' => 1,
]);
}

```

Our test is failing at the moment. First, we need to add the **update** method to our **api/Http/Controllers/BooksAuthorsRelationshipsController.php** like this:

```

public function update(Request $request, Book $book)
{

}

```

Like before, we leverage route-model bindings through the **Book** model argument, and we use the **Request** argument to be able to access the data from the request.

Before we do the implementation, we need to add the route to our **routes/api.php** file like this:

```

Route::patch('books/{book}/relationships/authors', '
    BooksAuthorsRelationshipsController@update')

```

```
->name('books.relationships.authors');
```

Then we can begin to implement the **update** method in the controller like this:

```
public function update(Request $request, Book $book)
{
    $ids = $request->input('data.*.id');
    $book->authors()->sync($ids);
    return response(null, 204);
}
```

As we mentioned before, we want to use the **sync** method, which we can call on our relationship method on our model. The **sync** method takes an array of **IDs** of the authors we want a relation to, so we need to extract the **ID**. We do that by leveraging the **input** method on the request again. Here, we get an array of only the IDs by telling the path to the **IDs** using dot notation. The **\*** wildcard here makes us able to tell the input that we are looking in an array and therefore don't have a specific key we can reference, but that we want the key **id** on the children. It's a great feature that you can also leverage when writing validation rules, but more on that later.

After we have synced the relations, we can return our empty response with the **204 No Content** status code and our test should be green and passing.

While we are at it, let's also make a test for removing relationships, even though we know it should work by now.

Breaking this down:

- 1. We set up our world
- a. We need a book to be able to fetch it through our API

- b. We need a couple of authors to exist and they need to be with our book, for us to test that some has been removed
- c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a PATCH request to the right API endpoint
  - b. We give the resource identifier objects for the authors we want to add, which should remove the ones not given
  - c. We add the right Accept and Content-Type headers
- 3. We assert against the result that
  - a. We get a status code 204 No Content back
- 4. We assert against the database that
  - a. We see each author given has a relationship to the book
  - b. We don't see a relationship to the authors that was not given

We have written the following test for this:

```
/**
 * @test
 * @watch
 */
public function
    it_can_modify_relationships_to_authors_and_remove_relationships
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 5)->create();
    $book->authors()->sync($authors->pluck('id'));

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->patchJson('/api/v1/books/1/relationships/authors',[
        'data' => [
            [
                'id' => '1',
```



```

        'type' => 'authors',
    ],
    [
        'id' => '2',
        'type' => 'authors',
    ],
    [
        'id' => '5',
        'type' => 'authors',
    ],
  ],
], [
  'accept' => 'application/vnd.api+json',
  'content-type' => 'application/vnd.api+json',
])->assertStatus(204);

$this->assertDatabaseHas('author_book', [
  'author_id' => 1,
  'book_id' => 1,
])->assertDatabaseHas('author_book', [
  'author_id' => 2,
  'book_id' => 1,
])->assertDatabaseHas('author_book', [
  'author_id' => 5,
  'book_id' => 1,
])->assertDatabaseMissing('author_book', [
  'author_id' => 3,
  'book_id' => 1,
])->assertDatabaseMissing('author_book', [
  'author_id' => 4,
  'book_id' => 1,
]);
}

```

This test should pass already since we are leveraging the **sync** method on our relationship.

To make sure we follow the JSON:API specification on removing all relationships, let's write a test that shows that giving an empty collection of resource

identifier objects will remove all relationships. Let's break it down:

- 1. We set up our world
  - a. We need a book to be able to fetch it through our API
  - b. We need a couple of authors to exist and they need to be with our book, for us to test that they are removed
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a PATCH request to the right API endpoint
  - b. We give an empty collection as data
  - c. We add the right **Accept** and **Content-Type** headers
- 3. We assert against the result that
  - a. We get a status code **204 No Content** back
- 4. We assert against the database that
  - a. We see that no relationships exist

We have written this test like so:

```
/**
 * @test
 * @watch
 */
public function
    it_can_remove_all_relationships_to_authors_with_an_empty_collection
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 3)->create();
    $book->authors()->sync($authors->pluck('id'));

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->patchJson('/api/v1/books/1/relationships/authors', [
```

```

        'data' => []
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(204);

    $this->assertDatabaseMissing('author_book', [
        'author_id' => 1,
        'book_id' => 1,
    ]->assertDatabaseMissing('author_book', [
        'author_id' => 2,
        'book_id' => 1,
    ]->assertDatabaseMissing('author_book', [
        'author_id' => 3,
        'book_id' => 1,
    ]);
}

```

This test should also pass right away. Up until now, we have taken a pretty naive route again: we haven't implemented any validation and what happens if you give an **ID** of a resource that does not exist. In this case we would get a **QueryException** back, which tells a bit too much about our database, so maybe we need to do some exception handling ourselves. According to the JSON:API specification, we should return a **404 Not Found** if a resource referenced cannot be found, so let's implement that.

First, we write a test, so let's break it down:

- 1. We set up our world
  - a. We need a book to be able to fetch it through our API
  - b. We only need a few authors
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a PATCH request to the right API endpoint
  - b. We give a collection where at least one resource identifier object points

to and ID of a non-existing author.

- c. We add the right **Accept** and **Content-Type** headers
- 3. We assert against the result that
  - a. We get a status code **404 Not Found** back
  - b. We get the right error document back

Our test looks like this:

```
/**
 * @test
 * @watch
 */
public function
    it_returns_a_404_not_found_when_trying_to_add_relationship_to_a_non_existing
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 5)->create();

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->patchJson('/api/v1/books/1/relationships/authors', [
        'data' => [
            [
                'id' => '5',
                'type' => 'authors',
            ],
            [
                'id' => '6',
                'type' => 'authors',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(404)->assertJson([
```

```
'errors' => [  
  [  
    'title' => 'Not Found Http Exception',  
    'details' => 'Resource not found',  
  ]  
]  
]);  
}
```

Our test is failing right now, telling us that we get a **500** status code back. We haven't implemented anything to handle `QueryExceptions` yet, but before we do, maybe we should think about where to implement it. If we do it in the controller, we would need to repeat that implementation for all resources and we won't catch all `QueryExceptions`, which could leak a bit too much information about our database. Our **app/Exceptions/Handler.php** is a better place, since it works on a more global level, so let's implement it there.

Now, we can use the test we already have, but since we already have a unit test for this class, let's continue there so we have both a unit and a feature test backing us on this part.

To break this test down:

- 1. We set up our world
  - a. We need an instance of our Handler
  - b. We need a request
  - c. We need the exception we want to test against
- 2. We run the code we are testing here
  - a. We call the render on our handler to trigger the exception handling
- 3. We assert against the result that
  - a. We get the right error document back

The test looks like this:

```

/**
 * @test
 * @watch
 */
public function
    it_converts_a_query_exception_into_a_not_found_exception()
{
    /** @var Handler $handler */
    $handler = app(Handler::class);

    $request = Request::create('/test', 'GET');
    $request->headers->set('accept', 'application/vnd.api+json');

    $exception = new QueryException('select ? from ?', ['name', '
        nothing'], new \Exception(''));

    $response = $handler->render($request, $exception);
    TestResponse::fromBaseResponse($response)->assertJson([
        'errors' => [
            [
                'title' => 'Not Found Http Exception',
                'details' => 'Resource not found',
            ]
        ]
    ]);
}

```

This test will also fail since we have no implementation, so let's go into the **app/Exceptions/Handler.php** file and this time focus on the **render** method. We implement this conversion of our exception in the **render** method, since it's both recommended by Laravel's documentation, and they are actually also doing it in the parents **render** method.

Here, we should add this:

```

public function render($request, Exception $exception)
{
    if($exception instanceof QueryException){
        $exception = new NotFoundException('Resource not found')
        ;
    }

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

```

Now, both our unit tests and feature tests pass. Again, we are taking a bit of a naive approach — we don't know if our consumers will send the correct resource identifier objects, so let's do some validation to ensure that we only update relationships if the correct request document is sent to our server.

### *Validating update of relationship links*

We have been through validation a couple of times, so this time we will go a bit quicker. Essentially, we want to test if the correct resource identifier object has been given, so to break down the concept of the tests:

- 1. We set up our world
  - a. We need a book to do the request
  - b. We need some authors to do the request
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We send a PATCH request to the right API endpoint
  - b. We send an invalid request object to trigger the validation
- 3. We assert against the result that
  - a. We get the status code 422 back
  - b. We get the right error document back

This is the concept for all the tests. The only thing we are changing is the wrong data given for the **ID** attribute and **type** attribute. In the **tests/Feature/-**

**BooksRelationshipsTest.php**, we have written the following tests:

```
/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_id_member_is_given_when_updating_a_relationship
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 5)->create();

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->patchJson('/api/v1/books/1/relationships/authors',[
        'data' => [
            [
                'type' => 'authors',
            ],
        ],
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(422)->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The data.0.id field is required.',
                'source' => [
                    'pointer' => '/data/0/id',
                ],
            ],
        ],
    ]));
}
```



```

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_id_member_is_a_string_when Updating_a_relationship
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 5)->create();

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->patchJson('/api/v1/books/1/relationships/authors',[
        'data' => [
            [
                'id' => 5,
                'type' => 'authors',
            ],
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ]->assertStatus(422)->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The data.0.id must be a string.',
                    'source' => [
                        'pointer' => '/data/0/id',
                    ],
                ],
            ],
        ]));
}

/**
 * @test

```

```

* @watch
*/
public function
    it_validates_that_the_type_member_is_given_when Updating_a_relationship
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 5)->create();

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->patchJson('/api/v1/books/1/relationships/authors',[
        'data' => [
            [
                'id' => '5',
            ],
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ]->assertStatus(422)->assertJson([
            'errors' => [
                [
                    'title' => 'Validation Error',
                    'details' => 'The data.0.type field is required.',
                    'source' => [
                        'pointer' => '/data/0/type',
                    ],
                ],
            ],
        ], [
            ], [
        ]);
}

/**
* @test
* @watch
*/
public function

```

```

    it_validates_that_the_type_member_has_a_value_of_authors_when_updating_a_r
    ()
  {
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 5)->create();

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->patchJson('/api/v1/books/1/relationships/authors',[
      'data' => [
        [
          'id' => '5',
          'type' => 'books',
        ],
      ],
    ], [
      'accept' => 'application/vnd.api+json',
      'content-type' => 'application/vnd.api+json',
    ])->assertStatus(422)->assertJson([
      'errors' => [
        [
          'title' => 'Validation Error',
          'details' => 'The selected data.0.type is invalid.',
          'source' => [
            'pointer' => '/data/0/type',
          ],
        ],
      ],
    ]);
  }
}

```

All these tests will fail at the moment, since we haven't implemented a request and added it to our controller. Let's start by making the request through artisan, so go to your terminal and run the following command:

```
php artisan make:request BooksAuthorsRelationshipsRequest
```

Let's jump into the new file at **app/Http/Requests/BooksAuthorsRelationshipRequest.php** and add the following:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class UpdateBooksAuthorsRelationshipsRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'data' => 'present|array',
            'data.*.id' => 'required|string',
            'data.*.type' => 'required|in:authors',
        ];
    }
}
```

```
}
```

We are not quite there yet. We still have to add the request to our controller **app/Http/Controllers/BooksAuthorsRelationshipsController.php** like this:

```
public function update(BooksAuthorsRelationshipsRequest $request,
    Book $book)
{
    $ids = $request->input('data.*.id');
    $book->authors()->sync($ids);
    return response(null, 204);
}
```

Our tests should be green and passing now, meaning that our validation works as intended.

### *Related link*

The last thing we need to implement is the **related** link and, if you recall from our chapter about the JSON:API specification, we need to return a collection of the related authors as resource objects. Let's write a test for it first.

To break this test down:

- 1. We set up our world
  - a. We need a book to do the request
  - b. We need some authors to do the request
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We send a GET request to the right API endpoint
- 3. We assert against the result that
  - a. We get the status code **200 OK** back

- b. We get the right response document back

We have written our test like this:

```
/**
 * @test
 * @watch
 */
public function
    it_can_get_all_related_authors_as_resource_objects_from_related_link
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 3)->create();
    $book->authors()->sync($authors->pluck('id'));

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/books/1/authors')
        ->assertStatus(200)
        ->assertJson([
            'data' => [
                [
                    "id" => '1',
                    "type" => "authors",
                    "attributes" => [
                        'name' => $authors[0]->name,
                        'created_at' => $authors[0]->created_at->
                            toJSON(),
                        'updated_at' => $authors[0]->updated_at->
                            toJSON(),
                    ]
                ],
            ],
            [
                "id" => '2',
                "type" => "authors",
            ]
        ]
    );
}
```

## BOOKS

```
        "attributes" => [
            'name' => $authors[1]->name,
            'created_at' => $authors[1]->created_at->
                toJSON(),
            'updated_at' => $authors[1]->updated_at->
                toJSON(),
        ]
    ],
    [
        "id" => '3',
        "type" => "authors",
        "attributes" => [
            'name' => $authors[2]->name,
            'created_at' => $authors[2]->created_at->
                toJSON(),
            'updated_at' => $authors[2]->updated_at->
                toJSON(),
        ]
    ],
]
});
}
```

At the moment this test is failing, since we have no implementation, so let's make that now. For this, we need a new controller so jump out into the terminal again and run the following artisan command:

```
php artisan make:controller BooksAuthorsRelatedController
```

Next, jump into the **routes/api.php** file and let's update the route for the **related** link like this:

```
Route::get('books/{book}/authors', '
    BooksAuthorsRelatedController@index')->name('books.authors');
```

Now, we are ready to work on our implementation, so let's go into **app/Http/Controllers/BookAuthorsRelatedController.php**. Here, we only need one method: we could have just used the **\_\_invoke** magic method, which you can use for single method controllers, but we like to be a bit more explicit so let's just create the **index** method, which also corresponds with the method given in our route.

This implementation is an easy one. We already have a collection for our authors, so we can just reuse this and send in a collection of the authors that are related to our book and return this:

```
public function index(Book $book)
{
    return new AuthorsCollection($book->authors);
}
```

And our test turns green and passes now. This was an easy one and we are essentially finished implementing this relationship. There is one more concept we are missing, which is the **include** member, so let's tackle that next.

## Include and included

The **included** top-level member makes it possible for you to include the related resources defined in the relationship linkage **data** member in the **relationships** object. Instead of having to make a new request for the related resources, they can just be sent together with the current response, making it possible for your consumer to save the hassle.



You can choose to always use the **include** member whenever there's a relationship on a resource ,or you can choose to leverage the **include** query parameter as well.

The **include** query parameter makes it possible for your consumers to tell whether they want to include the related resource objects in the response or not, making it possible to save time on requests that could take a long time.

We will implement both the **include** top-level member and the **include** query parameter now and, luckily for us, we don't have to implement the **include** query parameter, since this feature is also included in Spatie's **Laravel Query Builder** package.

Up until now, we have leveraged Laravel or third party packages to help us implement the conventions of the JSON:API specification, but for the **included** top-level member, we have to implement a bit more for this to work.

Let's start out by figuring out how to add an **included** top-level member, when fetching a single resource. Like before, we want to start by writing a test so we know when our implementation works. Here, we can write a unit test that tests our resource in isolation or we can continue with our feature tests.

If we go with a unit test to test a resource in isolation, we would have to provide the model and a request in order to test the resource. This can be done using mocks, which is an object that simulates the behavior of a real object. We use mocks because the thing we are testing has dependencies to other objects. To mock our Laravel models is quite a task, and one could argue that it would be easier to just hit the database and avoid mocking this part. This leaves us with a unit test that is not really being tested completely in isolation. We are actually using parts of our application and it all starts to bleed out from the original intention of a unit test. In our opinion, there's nothing wrong with hitting the database in a unit test, every once in a while. Remember, you test to ensure everything works as it should, not to follow some completely strict

rule set that will force you to spend hours trying to mock everything out.

Because we are so close to a real request/response flow, it would be much easier to just continue using a feature test and test that we get the correct resource in the **included** top-level member.

Let's break down what we want to test:

- 1. We set up our world
  - a. We need a book to do the request
  - b. We need some authors that will be related to the book
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We send a GET request to the right API endpoint
  - b. We need to add an **include** query parameter with the value of **authors**
- 3. We assert against the result that
  - a. We get the status code **200 OK** back
  - b. We get the right response document back in which
    - i. We see the **relationship** member
    - ii. We see the **author** relationship
    - iii. We see the relationship links
    - iv. We see the resource identifier objects for the related authors
  - c. We get the **included** top-level member back
  - d. The **included** top-level member contains a collection resource objects for authors

It is quite a number of things we test for here, but remember that most of the assertions are done through the `assertJson` method, asserting that we get the expected JSON back.

We have written this test like so:

## BOOKS

```
/**
 * @test
 * @watch
 */
public function
    it_includes_related_resource_objects_when_an_include_query_param_is_given
    ()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 3)->create();
    $book->authors()->sync($authors->pluck('id'));

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/books/1?include=authors', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200)
        ->assertJson([
            'data' => [
                'id' => '1',
                'type' => 'books',
                'relationships' => [
                    'authors' => [
                        'links' => [
                            'self' => route(
                                'books.relationships.authors',
                                ['id' => $book->id]
                            ),
                            'related' => route(
                                'books.authors',
                                ['id' => $book->id]
                            ),
                        ],
                    ],
                ],
            ],
            'data' => [
                [
                    'id' => (string)$authors->get(0)->
                        id,
```

```

        'type' => 'authors'
    ],
    [
        'id' => (string)$authors->get(1)->
            id,
        'type' => 'authors'
    ]
    ]
    ]
    ],
    'included' => [
        [
            "id" => '1',
            "type" => "authors",
            "attributes" => [
                'name' => $authors[0]->name,
                'created_at' => $authors[0]->created_at->
                    toJSON(),
                'updated_at' => $authors[0]->updated_at->
                    toJSON(),
            ]
        ],
        [
            "id" => '2',
            "type" => "authors",
            "attributes" => [
                'name' => $authors[1]->name,
                'created_at' => $authors[1]->created_at->
                    toJSON(),
                'updated_at' => $authors[1]->updated_at->
                    toJSON(),
            ]
        ],
        [
            "id" => '3',
            "type" => "authors",
            "attributes" => [
                'name' => $authors[2]->name,
                'created_at' => $authors[2]->created_at->

```

## BOOKS

```
        toJSON(),
        'updated_at' => $authors[2]->updated_at->
        toJSON(),
    ],
],
]);
}
```

Our test is, of course, failing at the moment, but we are ready to start implementing the **included** top-level member into our resource.

Let's go into the **app/Http/Controllers/BooksController.php** and take a look at the **show** method, which is used to fetch a single book.

```
public function show(Book $book)
{
    return new BooksResource($book);
}
```

As we mentioned earlier, we want to use the QueryBuilder from Spatie to handle the work with the **include** query parameter, so we will need to make some changes here. At the moment, we are leveraging Laravel's route-model bindings, but unfortunately we have to give this up and let the QueryBuilder do the query from now on. We still need the **ID** of the book to make the query, so let's implement it like this:

```
public function show($book)
{
    $query = QueryBuilder::for(Book::where('id', $book))
        ->allowedIncludes('authors')
        ->firstOrFail();
}
```

```
return new BooksResource($query);
}
```

Notice that by removing the reference to the **Book** model, we just get the **ID** back instead. The QueryBuilder's role here is first and foremost to fetch our Book, but also to fetch the related models according to what is given in the **include** query parameter. Next, we continue like before by returning a BookResource. The QueryBuilder is only fetching our related models, and doesn't do anything else or return a response. We will have to make the response ourselves.

So how do we know when a consumer wants the related resource objects for a certain relationship included in the response?

The strategy recommended by Spatie is to use the **whenLoaded** method on our Laravel resources to check if a relation has been loaded and then add the relations to our response document.

Let's take a look at our **app/Http/Resources/BooksResource.php** file to see what we do now:

```
public function toArray($request)
{
    return [
        'id' => (string)$this->id,
        'type' => 'books',
        'attributes' => [
            'title' => $this->title,
            'description' => $this->description,
            'publication_year' => $this->publication_year,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
```

```

    ],
    'relationships' => [
        'authors' => [
            'links' => [
                'self' => route(
                    'books.relationships.authors',
                    ['id' => $this->id]
                ),
                'related' => route(
                    'books.authors',
                    ['id' => $this->id]
                ),
            ],
            'data' => AuthorsIdentifierResource::collection(
                $this->authors),
        ],
    ],
];
}

```

What we want to focus on here is the relationship linkage or **data** member of our **authors** relationship. Here, we leverage the `AuthorsIdentifierResource` to create a collection of resource identifier objects for the related authors of the book. What we want you to notice is that we always send all the related authors in the response, even though we also have the links member. According to the JSON:API specification, we only need one of them and the **data** member is only required if we **include** relations in the response document.

Now that we are using the `QueryBuilder` from Spatie and we are letting it decide which relationships that should be fetched according to the **include**, our authors will only be loaded whenever they are set to be included through the **include** query parameter. So to avoid any errors, let's change the implementation to this:

```

public function toArray($request)
{
    return [
        'id' => (string)$this->id,
        'type' => 'books',
        'attributes' => [
            'title' => $this->title,
            'description' => $this->description,
            'publication_year' => $this->publication_year,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ],
        'relationships' => [
            'authors' => [
                'links' => [
                    'self' => route(
                        'books.relationships.authors',
                        ['id' => $this->id]
                    ),
                    'related' => route(
                        'books.authors',
                        ['id' => $this->id]
                    ),
                ],
                'data' => AuthorsIdentifierResource::collection(
                    $this->whenLoaded('authors')),
            ],
        ],
    ];
}

```

Here, we leverage the **whenLoaded** method instead, which makes sure that we only add the **data** member and its contents if the relation to authors has been loaded. Methods like these are what makes it a joy to work with Laravel.

Before we continue our implementation, we just want to touch upon Laravel's Eloquent API resources once more, because it is important you grasp this part, in order to understand the implementation we are about to make.



A resource is basically a transformer that converts the data of your Eloquent model into the desired JSON object you want for your API. The transformation happens through the **toArray** method, where you define both the structure and values for your members according to the attributes on your Eloquent model.

When using this resource, Laravel takes the structure from your **toArray** method and creates our response document. It's actually Laravel that makes sure that our single resource gets a **data** top-level member.

It is possible to deactivate this behavior and control the top-level members for yourself, but you will face an issue when creating collections of this resource, since these top-level members will be included in the collection. If we let Laravel handle this, it will automatically remove the top-level members of each resource when making a collection of them.

We can add to the top-level members by using a **with** method in our resources. We will use this to add the **included** top-level member to our single resource, but before we do so, we also have to think about how to implement the way we want to add content to our **included** member.

The **included** member has to be a single flat array with resource objects for all related resources listed one after the other. To make it clearer, let's take this example from the JSON:API specification:

```
"included": [{  
  "type": "people",  
  "id": "9",  
  "attributes": {  
    "first-name": "Dan",  
    "last-name": "Gebhardt",  
    "twitter": "dgeb"  }  
}]
```

```

    },
    "links": {
        "self": "http://example.com/people/9"
    }
}, {
    "type": "comments",
    "id": "5",
    "attributes": {
        "body": "First!"
    },
    "relationships": {
        "author": {
            "data": { "type": "people", "id": "2" }
        }
    },
    "links": {
        "self": "http://example.com/comments/5"
    }
}, {
    "type": "comments",
    "id": "12",
    "attributes": {
        "body": "I like XML better"
    },
    "relationships": {
        "author": {
            "data": { "type": "people", "id": "9" }
        }
    },
    "links": {
        "self": "http://example.com/comments/12"
    }
}]

```

In this example, we can see that resource objects for both people and comments are in the same flat array — there is no nesting according to type or anything.

So we know we need a single array to contain all of our resource objects, and we also know that we only need to include the resource objects of related resource

identifier objects given in the **data** member of each relationship. So let's break the implementation down and write it out as we want it to be in a perfect world:

- We want to collect the related resources
- We only want to collect the related resources needed, which are the ones given in the **include** query parameter
- We want everything to end up in a single flat array

First, we want a way to collect the related resources, so let's create a method for this where we collect these using an array:

```
private function relations()
{
    return [

    ];
}
```

That looks good, but how do we only collect the related resources given in the **include** query parameter? We can reuse what we did in the **toArray** method when conveying that we only want to add the **data** member when a relation is loaded like this:

```
private function relations()
{
    return [
        AuthorsResource::collection($this->whenLoaded('authors')),
    ];
}
```

Through this, we either get a collection of resource objects for authors or an empty collection.

Let's see how far we are from having everything implemented, and add the **with** method and the **included** member through that:

```
public function with($request)
{
    return [
        'included' => $this->relations(),
    ];
}
```

The output for the failing tests changes and if we take a look at what we get back in the **included** member, we are not far from being done:

```
"included": [
  [
    {
      "id": "1",
      "type": "authors",
      "attributes": {
        "name": "Lizeth Renner",
        "created_at": "2019-05-02T11:59:42.000000Z",
        "updated_at": "2019-05-02T11:59:42.000000Z"
      }
    },
    {
      "id": "2",
      "type": "authors",
      "attributes": {
        "name": "Euna Bosco",
        "created_at": "2019-05-02T11:59:42.000000Z",
        "updated_at": "2019-05-02T11:59:42.000000Z"
      }
    }
  ],
  {

```

```

        "id": "3",
        "type": "authors",
        "attributes": {
            "name": "Dr. Fannie Orn MD",
            "created_at": "2019-05-02T11:59:42.000000Z",
            "updated_at": "2019-05-02T11:59:42.000000Z"
        }
    }
]

```

It's not a flat array yet, but we can do something about that. The first thing that might come to mind is to flatten the array, so let's try to do that. The easiest way to do this is to use a collection like this:

```

public function with($request)
{
    return [
        'included' => collect($this->relations())->flatten(),
    ];
}

```

Looking at our test, it's still failing and looking at the **included** member of the result, it's still the same problem. The reason is that we use the `AuthorsResource` class, which also has a **toArray** method being called but it happens after we have flattened the array or collection. So the best thing to do here might be to call the **toArray** method ourselves and then flatten the collection like this:

```

public function with($request)
{

```

```

return [
    'included' => collect($this->relations())
        ->flatMap(function ($resource) use($request) {
            return $resource->toArray($request);
        })
];
}

```

And our test is green and passing. But what about the scenario where we don't want to **include** anything — does the **included** top-level member then show up? Let's write a test for it.

Let's break down what we want to test:

- 1. We set up our world
  - a. We need a book to do the request
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We send a GET request to the right API endpoint
- 3. We assert against the result that
  - a. We get the status code **200 OK** back
  - b. We **don't** get the **included** top-level member back

We have written the test like this:

```

/**
 * @test
 * @watch
 */
public function
    it_does_not_include_related_resource_objects_when_an_include_query_param_is_

```

```

    ()
{
    $this->withoutExceptionHandling();
    $book = factory(Book::class)->create();

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/books/1', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200)
        ->assertJsonMissing([
            'included' => [],
        ]);
}

```

This test actually fails with a **500** status code immediately, so we chose to include the **withoutExceptionHandling** method in the test to give us a bit more information. Right now, we get an error telling us that we are trying to get property of 'map' of non-object. The reason for this is that we have no relations loaded, which results in our **relations** method and returns an array with empty **AuthorResource** collections. Whenever we call **toArray** on an empty **AuthorResource** collection, it returns an empty array and an empty cannot be mapped. It would be nice if we could filter away those empty **AuthorsResource** collections before we try to map them, so let's try to do that, leveraging our Laravel collection to filter these away like this:

```

public function with($request)
{
    return [
        'included' => collect($this->relations())
            ->filter(function($resource){

```

```

        return $resource->collection !== null;
    })
    ->flatMap(function ($resource) use($request) {
        return $resource->toArray($request);
    })
];
}

```

Our test now changes and tells us that it actually found the **included** top-level member.

A simple conditional can fix this for us, but before we add this, let's move the included implementation of the **with** method to its own **included** method to make it a bit more readable.

Then we can more easily add the conditional in our **with** method, making the code a bit more descriptive:

```

private function relations()
{
    return [
        AuthorsResource::collection($this->whenLoaded('authors')),
    ];
}

public function included($request)
{
    return collect($this->relations())
        ->filter(function ($resource) {
            return $resource->collection !== null;
        })
        ->flatMap(function ($resource) use ($request) {
            return $resource->toArray($request);
        });
}

```



```

}

public function with($request)
{
    $with = [];

    if ($this->included($request)->isNotEmpty()) {
        $with['included'] = $this->included($request);
    }

    return $with;
}

```

Our test is green and passing now and we have successfully implemented **included** top-level member for single resources. However, what about when we need to include related resource objects for many resources in a collection? Let's tackle that next.

## *Collections*

For collections, we use dedicated Resource collection classes and the reason we do this is actually because of the **included** top-level member we are about to implement now. The problem we are facing is that we need to collect all the related resources objects from all of the resources in our resource collection and put them into an **included** top-level member in a flat structure. As always, let's break down what we want to test and then write the test:

- 1. We set up our world
  - a. We need some books to do the request
  - b. We need some authors related to one book
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We send a GET request to the right API endpoint
  - b. We add the **included** query parameter

- 3. We assert against the result that
  - a. We get the status code **200 OK** back
  - b. We get the right relationships for each resource
  - c. We see the **included** top-level member
  - d. The **included** top-level member contains the correct resource objects

We have written this quite long test like this:

```
/**
 * @test
 * @watch
 */
public function
    it_includes_related_resource_objects_for_a_collection_when_an_include_query_
    ()
{
    $books = factory(Book::class, 3)->create();
    $authors = factory(Author::class, 3)->create();
    $books->each(function($book, $key) use($authors){
        if($key === 0){
            $book->authors()->sync($authors->pluck('id'));
        }
    });

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->get('/api/v1/books?include=authors', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(200)->assertJson([
        "data" => [
            [
                "id" => '1',
                "type" => "books",
                "attributes" => [
                    'title' => $books[0]->title,
```

## BOOKS

```

        'description' => $books[0]->description,
        'publication_year' => $books[0]->
            publication_year,
        'created_at' => $books[0]->created_at->toJSON(),
        'updated_at' => $books[0]->updated_at->toJSON(),
    ],
    'relationships' => [
        'authors' => [
            'links' => [
                'self' => route(
                    'books.relationships.authors',
                    ['id' => $books[0]->id]
                ),
                'related' => route(
                    'books.authors',
                    ['id' => $books[0]->id]
                ),
            ],
            'data' => [
                [
                    'id' => $authors->get(0)->id,
                    'type' => 'authors'
                ],
                [
                    'id' => $authors->get(1)->id,
                    'type' => 'authors'
                ],
                [
                    'id' => $authors->get(2)->id,
                    'type' => 'authors'
                ]
            ]
        ]
    ],
],
[
    "id" => '2',
    "type" => "books",
    "attributes" => [
        'title' => $books[1]->title,

```

```

        'description' => $books[1]->description,
        'publication_year' => $books[1]->
            publication_year,
        'created_at' => $books[1]->created_at->toJSON(),
        'updated_at' => $books[1]->updated_at->toJSON(),
    ],
    'relationships' => [
        'authors' => [
            'links' => [
                'self' => route(
                    'books.relationships.authors',
                    ['id' => $books[1]->id]
                ),
                'related' => route(
                    'books.authors',
                    ['id' => $books[1]->id]
                ),
            ],
        ],
    ]
],
[
    "id" => '3',
    "type" => "books",
    "attributes" => [
        'title' => $books[2]->title,
        'description' => $books[2]->description,
        'publication_year' => $books[2]->
            publication_year,
        'created_at' => $books[2]->created_at->toJSON(),
        'updated_at' => $books[2]->updated_at->toJSON(),
    ],
    'relationships' => [
        'authors' => [
            'links' => [
                'self' => route(
                    'books.relationships.authors',
                    ['id' => $books[2]->id]
                ),
                'related' => route(

```

## BOOKS

```

        'books.authors',
        ['id' => $books[2]->id]
    ),
],
]
],
],
'included' => [
    [
        "id" => '1',
        "type" => "authors",
        "attributes" => [
            'name' => $authors[0]->name,
            'created_at' => $authors[0]->created_at->toJSON
                (),
            'updated_at' => $authors[0]->updated_at->toJSON
                (),
        ]
    ],
    [
        "id" => '2',
        "type" => "authors",
        "attributes" => [
            'name' => $authors[1]->name,
            'created_at' => $authors[1]->created_at->toJSON
                (),
            'updated_at' => $authors[1]->updated_at->toJSON
                (),
        ]
    ],
    [
        "id" => '3',
        "type" => "authors",
        "attributes" => [
            'name' => $authors[2]->name,
            'created_at' => $authors[2]->created_at->toJSON
                (),
            'updated_at' => $authors[2]->updated_at->toJSON
                (),
        ]
    ]
]

```

```

        ],
    ],
]);
}

```

We'll start out with relations on one book at first. The test is, of course, failing at the moment but before we jump into our **app/Http/Resources/BooksCollection.php** file, let's think about what we want to implement:

1. We want an **included** top-level member next to the **data** top-level member
2. We want the **included** member to contain all the related resource objects for each resource in the collection
3. We want the all the resource objects to be in a flat structured array.

To add the **included** top-level member, we simply add another key to the array with that name, so that's easy enough.. We then need a method that can collect all the related resource objects from each resource in the collection, and then return it as a flattened array. As it turns out, we have already done all of this hard work through the **included** method in our **BooksResource** class, so we can just call this for each resource. This can be done like this:

```

public function toArray($request)
{
    return [
        'data' => $this->collection,
        'included' => $this->mergeIncludedRelations($request),
    ];
}

private function mergeIncludedRelations($request)
{

```

```

    $includes = $this->collection->flatMap(function ($resource) use(
        $request){
        return $resource->included($request);
    });

    return $includes;
}

```

With this method our test is already passing. This is great, but maybe we should do like before and ensure that the **included** top-level member is not shown when the **include** query parameter is not given. Again, we will write a test for it where:

- 1. We set up our world
  - a. We need some books to do the request
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We send a GET request to the right API endpoint
- 3. We assert against the result that
  - a. We get the status code **200 OK** back
  - b. We **don't** see the **included** top-level member

We have written the test like this:

```

/**
 * @test
 * @watch
 */
public function
    it_does_not_include_related_resource_objects_for_a_collection_when_an_incl
    ()
{
    $books = factory(Book::class, 3)->create();
}

```

```

$user = factory(User::class)->create();
Passport::actingAs($user);

$this->get('/api/v1/books', [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])->assertStatus(200)
    ->assertJsonMissing([
        'included' => [],
    ]);
}

```

This test fails right away, so let's implement a way to avoid adding the **included** top-level member to the response document when there are no related resource objects to be included. If you recall from our **BooksResource** class, we used the **whenLoaded** method like this:

```

'data' => AuthorsIdentifierResource::collection($this->whenLoaded(
    'authors')),

```

This method either gets the related resources if they are loaded or it returns a **MissingValue** object. Laravel then knows that it should not include a member in the JSON response, when its value is this object. You might be thinking that we should have used this in our **with** method, but this only works in the **toArray** method.

With this knowledge, we can easily implement the solution using a ternary operator like this:



```
private function mergeIncludedRelations($request)
{
    $includes = $this->collection->flatMap(function ($resource) use(
        $request){
        return $resource->included($request);
    });

    return $includes->isNotEmpty() ? $includes : new MissingValue();
}
```

This makes our test green and passing.

Now, imagine a scenario where we have a collection of books and the same authors have written those books. In that case, it wouldn't make a lot of sense to have all of these authors repeat for each book they have written. We would like to see each author once in our included member. But does our current implementation handle this? Let's break down how to test this:

- 1. We set up our world
  - a. We need some books to do the request
  - b. We need some authors related to all books
  - c. We need to be authenticated
- 2. We run the code we are testing, here
  - a. We send a GET request to the right API endpoint
  - b. We add the included query parameter
- 3. We assert against the result that
  - a. We get the status code 200 OK back
  - b. We get the right relationships for each resource
  - c. We see the included top-level member
  - d. The included top-level member contains the correct resource objects
  - e. We see that a resource object is only included once

We have written the test like this:

```

/**
 * @test
 * @watch
 */
public function
    it_only_includes_a_related_resource_object_once_for_a_collection
    ()
{
    $books = factory(Book::class, 3)->create();
    $authors = factory(Author::class, 3)->create();
    $books->each(function($book, $key) use($authors){
        $book->authors()->sync($authors->pluck('id'));
    });

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->get('/api/v1/books?include=authors', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(200)->assertJson([
        "data" => [
            [
                "id" => '1',
                "type" => "books",
                "attributes" => [
                    'title' => $books[0]->title,
                    'description' => $books[0]->description,
                    'publication_year' => $books[0]->
                        publication_year,
                    'created_at' => $books[0]->created_at->toJSON(),
                    'updated_at' => $books[0]->updated_at->toJSON(),
                ],
                'relationships' => [
                    'authors' => [
                        'links' => [
                            'self' => route(
                                'books.relationships.authors',
                                ['id' => $books[0]->id]
                            ),

```

## BOOKS

```

        'related' => route(
            'books.authors',
            ['id' => $books[0]->id]
        ),
    ],
    'data' => [
        [
            'id' => $authors->get(0)->id,
            'type' => 'authors'
        ],
        [
            'id' => $authors->get(1)->id,
            'type' => 'authors'
        ],
        [
            'id' => $authors->get(2)->id,
            'type' => 'authors'
        ]
    ]
]
],
[
    "id" => '2',
    "type" => "books",
    "attributes" => [
        'title' => $books[1]->title,
        'description' => $books[1]->description,
        'publication_year' => $books[1]->
            publication_year,
        'created_at' => $books[1]->created_at->toJSON(),
        'updated_at' => $books[1]->updated_at->toJSON(),
    ],
    'relationships' => [
        'authors' => [
            'links' => [
                'self' => route(
                    'books.relationships.authors',
                    ['id' => $books[1]->id]
                ),
            ]
        ]
    ]
]

```

```

        'related' => route(
            'books.authors',
            ['id' => $books[1]->id]
        ),
    ],
    'data' => [
        [
            'id' => $authors->get(0)->id,
            'type' => 'authors'
        ],
        [
            'id' => $authors->get(1)->id,
            'type' => 'authors'
        ],
        [
            'id' => $authors->get(2)->id,
            'type' => 'authors'
        ]
    ]
]
],
[
    "id" => '3',
    "type" => "books",
    "attributes" => [
        'title' => $books[2]->title,
        'description' => $books[2]->description,
        'publication_year' => $books[2]->
            publication_year,
        'created_at' => $books[2]->created_at->toJSON(),
        'updated_at' => $books[2]->updated_at->toJSON(),
    ],
    'relationships' => [
        'authors' => [
            'links' => [
                'self' => route(
                    'books.relationships.authors',
                    ['id' => $books[2]->id]
                ),
            ]
        ]
    ]
]

```

## BOOKS

```

        'related' => route(
            'books.authors',
            ['id' => $books[2]->id]
        ),
    ],
    'data' => [
        [
            'id' => $authors->get(0)->id,
            'type' => 'authors'
        ],
        [
            'id' => $authors->get(1)->id,
            'type' => 'authors'
        ],
        [
            'id' => $authors->get(2)->id,
            'type' => 'authors'
        ]
    ]
    ],
    ],
],
'included' => [
    [
        "id" => '1',
        "type" => "authors",
        "attributes" => [
            'name' => $authors[0]->name,
            'created_at' => $authors[0]->created_at->toJSON
                (),
            'updated_at' => $authors[0]->updated_at->toJSON
                (),
        ]
    ],
    [
        "id" => '2',
        "type" => "authors",
        "attributes" => [
            'name' => $authors[1]->name,

```

```

        'created_at' => $authors[1]->created_at->toJSON
            (),
        'updated_at' => $authors[1]->updated_at->toJSON
            (),
    ]
],
[
    "id" => '3',
    "type" => "authors",
    "attributes" => [
        'name' => $authors[2]->name,
        'created_at' => $authors[2]->created_at->toJSON
            (),
        'updated_at' => $authors[2]->updated_at->toJSON
            (),
    ]
],
]
])->assertJsonMissing([
    'included' => [
        [
            "id" => '1',
            "type" => "authors",
            "attributes" => [
                'name' => $authors[0]->name,
                'created_at' => $authors[0]->created_at->toJSON
                    (),
                'updated_at' => $authors[0]->updated_at->toJSON
                    (),
            ]
        ],
    ],
    [
        "id" => '2',
        "type" => "authors",
        "attributes" => [
            'name' => $authors[1]->name,
            'created_at' => $authors[1]->created_at->toJSON
                (),
            'updated_at' => $authors[1]->updated_at->toJSON
                (),
        ]
    ]
]

```

```

    ]
  ],
  [
    "id" => '3',
    "type" => "authors",
    "attributes" => [
      'name' => $authors[2]->name,
      'created_at' => $authors[2]->created_at->toJSON
        (),
      'updated_at' => $authors[2]->updated_at->toJSON
        (),
    ]
  ],
  [
    "id" => '1',
    "type" => "authors",
    "attributes" => [
      'name' => $authors[0]->name,
      'created_at' => $authors[0]->created_at->toJSON
        (),
      'updated_at' => $authors[0]->updated_at->toJSON
        (),
    ]
  ],
  [
    "id" => '2',
    "type" => "authors",
    "attributes" => [
      'name' => $authors[1]->name,
      'created_at' => $authors[1]->created_at->toJSON
        (),
      'updated_at' => $authors[1]->updated_at->toJSON
        (),
    ]
  ],
  [
    "id" => '3',
    "type" => "authors",
    "attributes" => [
      'name' => $authors[2]->name,

```

```

        'created_at' => $authors[2]->created_at->toJSON
            (),
        'updated_at' => $authors[2]->updated_at->toJSON
            (),
    ]
],
[
    "id" => '1',
    "type" => "authors",
    "attributes" => [
        'name' => $authors[0]->name,
        'created_at' => $authors[0]->created_at->toJSON
            (),
        'updated_at' => $authors[0]->updated_at->toJSON
            (),
    ]
],
[
    "id" => '2',
    "type" => "authors",
    "attributes" => [
        'name' => $authors[1]->name,
        'created_at' => $authors[1]->created_at->toJSON
            (),
        'updated_at' => $authors[1]->updated_at->toJSON
            (),
    ]
],
[
    "id" => '3',
    "type" => "authors",
    "attributes" => [
        'name' => $authors[2]->name,
        'created_at' => $authors[2]->created_at->toJSON
            (),
        'updated_at' => $authors[2]->updated_at->toJSON
            (),
    ]
],
]

```



```
    });
}
```

There's a lot of assertion in this one, but most of it is copied from our earlier test. The thing to notice is how every resource in the response document has a relationship, and that all of these related resources have been included. We also assert that we don't repeat these related resource objects, which does make everything a bit longer.

This test is failing because we, as anticipated, do not take care of repeating resource objects. Luckily, this is very easy to fix, leveraging the features of Laravel's collections like this:

```
private function mergeIncludedRelations($request)
{
    $includes = $this->collection->flatMap(function ($resource) use(
        $request){
        return $resource->included($request);
    }->unique()->values();

    return $includes->isEmpty() ? $includes : new MissingValue();
}
```

By using the **unique** method, we can remove duplicates from the collection. Now, these duplicates can be on various positions in the collection and to convert this collection to a JSON array later, we need the keys to be in chronological order, otherwise it will be converted as a JSON object. To solve this problem, we use the **values** method which will reset the key positions on the collection.

Our test is now passing and we are actually done implementing the **include** query parameter and the **included** top-level member for our books resource.

There are some things we can do to clean our code a bit more, so let's just go through this quickly. In both our **BooksCollection** and **BooksResource** classes, we use a **flatMap** method calling a method on our instances.

In **BooksCollection** we do it here:

```
private function mergeIncludedRelations($request)
{
    $includes = $this->collection->flatMap(function ($resource) use
        ($request){
            return $resource->included($request);
        })->unique()->values();

    return $includes->isNotEmpty() ? $includes : new MissingValue()
        ;
}
```

And in **BooksResource** we do it here:

```
public function included($request)
{
    return collect($this->relations())
        ->filter(function ($resource) {
            return $resource->collection !== null;
        })->flatMap(function ($resource) use ($request) {
            return $resource->toArray($request);
        });
}
```

In both cases, we are calling a method on each item in the collection and then we map the result of that method to our collection and finally flatten it. Calling methods on the items in collections is actually very common — so common that Laravel collections has a feature called “higher order messages”,

which is a short-cut to do these calls on items in our collections. Let's use the **BooksCollection** as example:

```
private function mergeIncludedRelations($request)
{
    $includes = $this->collection->flatMap->included($request)->
        unique()->values();
    return $includes->isEmpty() ? $includes : new MissingValue();
}
```

Instead of using the **flatMap** method, we can add a dynamic property which is also conveniently called **flatMap**, but the cool thing is that it allows us to skip the closure where we make the call on an item and chain that call into the existing chain of the collection. It makes our code much shorter and more readable.

Let's do the same for our **BooksResource**:

```
public function included($request)
{
    return collect($this->relations())
        ->filter(function ($resource) {
            return $resource->collection !== null;
        })->flatMap->toArray($request);
}
```

A little shorter and a little nicer. Before we move on, you should go into the terminal and run PHPUnit like this:

```
./vendor/bin/phpunit
```

Some of our earlier tests are failing because we have added the **whenLoaded** method in our **BooksResource** class. In these tests, you should add the **include** query parameter to the URL because of the new implementation and the tests will pass again.

## Summary

We've been through a lot in this chapter, so let's just summarize it a bit.

We started out by creating the books resource as a repetition of the last chapter and as a way to reinforce what you have learned about testing.

We then began our journey into relationships, starting out by implementing relationships into our **BooksResource** class.

We then implemented relationship links so that consumers can both access our relationship and modify these without deleting resources themselves. We created dedicated controllers for these parts and also implemented validation, so we secure ourselves from wrong requests from our users that could damage our application.

Lastly, we implemented the **include** query parameter using Spaties Query-Builder once again, and then we implemented the **included** top-level member into both our **BooksResource** and **BooksCollection** classes.

We have implemented almost all the features from the JSON:API specification at this point, and we know how to implement each part from now on, so we can focus a bit more on our application.

The only part we are missing is the ability to add or update relationships when creating or updating resources. We will implement these features when implementing our comments resources.

You might also think that we have forgotten about authors' relationship to books, but as you might also have noticed, we have written a lot of code twice. In fact, following conventions like the ones in the JSON:API specification requires you to write a lot of the same code. But why should we repeat ourselves all the time? Isn't there a better solution? We will take a look at this in the next chapter.

\* \* \*

## 7

# Don't repeat yourself

In this chapter, we will be looking at the **Don't repeat yourself** principle to see if we can clean up our code a bit, since the concept for resources are the same whether it's an author, a book, a user and so on. This is just what we deal with while implementing a specification with strict protocols, but remember that they are there for a reason. Let's see if we can avoid having to make the same implementation on and on, and make future implementations a bit faster.

The **Don't Repeat Yourself** principle is stated as such: *“Every piece of knowledge must have a single, unambiguous, authoritative representation within a system”* (Andy Hunt & Dave Thomas: *The Pragmatic Programmer*, 1999.), meaning that we should aim at reducing repetition in our software to avoid redundancy.

To put this more clearly, let's say that there's a change in how a resource object is structured. We would then have to change this in several places in order to meet this new requirement.

If we instead follow the Don't Repeat Yourself principle or **DRY**, which we will call it from now on, we could abstract our code a bit more and have a single resource object class we could reuse. Then we would only have one place to make a change if need be.

This might not seem as obvious right now, but don't worry, we will show you how, and remember that we are backed by our tests which will immediately tell us if anything is breaking.

We will do our refactoring in these steps

1. We will refactor our resources and collections into fewer classes
2. We will refactor our controllers to use a single service than can be used in future controllers
3. We will refactor our validation request classes

Some of these things will overlap a bit. For instance, we will have to make changes to our controllers almost all the way through this, since these contain most of the code for our API at the moment.

We do not refactor our tests, even though some of them could be reduced. We want these to be repetitive and not too abstracted so that they can function as documentation as well.

## Refactoring resources and collections

Let's start with our resources and collections by taking a look at the classes we already have in `app/Http/Resources`:

- `AuthorsCollection.php`
- `AuthorsIdentifierResource.php`
- `AuthorsResource.php`
- `BooksCollection.php`
- `BooksResource.php`

We have a bunch of classes that are actually doing the same stuff, namely transforming models into JSON responses. If we take a look at both **BooksResource.php** and **AuthorsResource.php** and for the moment ignore the

relationship part of books, which would be almost the same for authors as well, they are nearly the same. The thing that makes them different from each other is the **type** member and the **attributes**.

The resource objects actually work as an extension of our models, which makes it possible for us to use the **\$this** keyword to access the attributes and relationships on our model. On our models, we can get an array of all attributes for the model, so that part is easily solved. If we want a general resource we can reuse, we can no longer write the type on the resource, but we can move this to our model instead.

Let's implement this part and remember, we don't need any tests since we already have tests that cover how our resources should work. We can work with the implementations only now. To run our tests, however, we won't use the Laravel Test Watcher, but rather just run PHPUnit since it is a bit easier when refactoring.

The first thing we should do is to run PHPUnit to see if everything is passing:

```
./vendor/bin/phpunit
```

Great! Everything is passing and we now have a good starting point.

## Authors

Let's start by creating our resource like this:

```
php artisan make:resource JSONAPIResource
```



Again, we want to take the easiest route and since authors doesn't have any relationships yet, it is far easier to handle this resource first.

Jump into **app/Http/Controllers/AuthorsController.php** and let's begin by swapping out the **AuthorsResource** in the **show** method with our new **JSON-APIResource** like this:

```
public function show(Author $author)
{
    return new JSONAPIResource($author);
}
```

Then let's run PHPUnit right away to see what happens:

```
./vendor/bin/phpunit
```

We have one failing test, telling us that the response doesn't contain the correct response document. Go into **app/Http/Resources/JSONAPIResource.php** and let's begin by adding the correct structure, which we can just copy from the **AuthorsResource.php** file like this:

```
public function toArray($request)
{
    return [
        'id' => (string)$this->id,
        'type' => 'authors',
        'attributes' => [
            'name' => $this->name,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
```

```
    ]
  ];
}
```

This is far too specific for authors: we want to get away from this, and we want to get the **attributes** from our model and the **type** as well. We could just call the **getAttributes** method on the model, but this will return all the attributes — even those that are hidden. The most ideal thing would be to only get the attributes that are allowed to be shown to the public, so let's create a new method on our **app/Author.php** model that can return the allowed attributes like this:

```
public function allowedAttributes(){
    return collect($this->attributes)->filter(function($item, $key){
        return !collect($this->hidden)->contains($key) && $key !== '
            id';
    }->merge([
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ]);
}
```

We use Laravel's collection once again, because it makes it very easy to filter arrays. We filter away the attributes that are added to the array in the hidden property on the model, and then we merge in our **created\_at** and **updated\_at** attributes since we are also interested in these.

While we are in the model class, let's create another method for our type attribute like this:

```
public function type()
{
    return 'authors';
}
```

Now, we have the necessary methods we can call from our resource, so let's jump back to our **app/Http/Resources/JSONAPIResource.php** file and change the array in the **toArray** method to the following:

```
public function toArray($request)
{
    return [
        'id' => (string)$this->id,
        'type' => $this->type(),
        'attributes' => $this->allowedAttributes(),
    ];
}
```

Run PHPUnit again and let's see what happens. All the tests should pass again, which means our refactoring was successful. We are not done though: we need a way to do this for the rest of our resources as well, so how do we do that?

One thing we could do is to abstract our **allowedAttributes** method from our model into a trait, and add this to all the models that should be able to return allowed **attributes** for the attributes member. For the **type** member, we just need to remember to write the **type** method on our models and then everything would be fine.

In this case, a better solution might be to create an abstract class that extends Laravel's model class, which our models could extend. This class could then contain an abstract **type** method we need to implement before PHP would be happy with the implementation. In this class, we can then add the **allowedAttributes** method, which will let all the models that extend this

abstract class, inherit the method automatically. We think it's much better that PHP fails so that nothing works, which will guarantee that you notice a missing implementation, instead of relying on you running your tests to catch it.

Let's create an abstract class and call it **AbstractAPIModel.php** and place it in our **app** folder. Then, let's move the `allowedAttributes` to this class and add the abstract **type** method like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

abstract class AbstractAPIModel extends Model
{
    /**
     * @return string
     */
    abstract public function type();

    public function allowedAttributes(){
        return collect($this->attributes)->filter(function($item,
            $key){
                return !collect($this->hidden)->contains($key) && $key
                    !== 'id';
            }->merge([
                'created_at' => $this->created_at,
                'updated_at' => $this->updated_at,
            ]);
    }
}
```

In our **app/Author.php** model, let's remove the **allowedAttributes** method

and extend our **AbstractAPIModel** instead of **Model** like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Author extends AbstractAPIModel
{
    protected $fillable = ['name'];

    public function books()
    {
        return $this->belongsToMany(Book::class);
    }

    public function type()
    {
        return 'authors';
    }
}
```

Here, we implement the **type** method and return the **type** as a **string**. Let's run PHPUnit again where everything should still pass.

Great! We now have a convenient way of making our models prepared for our **JSONAPIResource**. They all simply need to extend our **AbstractAPIModel** and implement the **type** method and they will work. We won't jump over to books to implement this just yet. Instead, let's continue with authors.

Back in the **app/Http/Controllers/AuthorsController.php** let's replace the usage of **AuthorsResource** to our new **JSONAPIResource** in the rest of the methods like this:

```

<?php

namespace App\Http\Controllers;

use App\Author;
use App\Http\Requests\CreateAuthorRequest;
use App\Http\Requests\UpdateAuthorRequest;
use App\Http\Resources\AuthorsCollection;
use App\Http\Resources\AuthorsResource;
use App\Http\Resources\JSONAPIResource;
use Illuminate\Http\Request;
use Spatie\QueryBuilder\QueryBuilder;

class AuthorsController extends Controller
{
    public function index()
    {
        $authors = QueryBuilder::for(Author::class)->allowedSorts([
            'name',
            'created_at',
            'updated_at',
        ]->jsonPaginate());
        return new AuthorsCollection($authors);
    }

    public function store(CreateAuthorRequest $request)
    {
        $author = Author::create([
            'name' => $request->input('data.attributes.name'),
        ]);
        return (new JSONAPIResource($author))
            ->response()
            ->header('Location', route('authors.show', [
                'author' => $author,
            ]));
    }

    public function show(Author $author)
    {
        return new JSONAPIResource($author);
    }
}

```

```

    }

    public function update(UpdateAuthorRequest $request, Author
        $author)
    {
        $author->update($request->input('data.attributes'));
        return new JSONAPIResource($author);
    }

    public function destroy(Author $author)
    {
        $author->delete();
        return response(null, 204);
    }
}

```

Let's run PHPUnit again: Here, all the tests should still be passing.

Then it's time to look at collections for authors — let's see if we can make this a bit more general as well. Go out into the terminal and create a new resource with a collections flag, making sure that Laravel Artisan creates a resource collection for us:

```
php artisan make:resource JSONAPICollection --collection
```

Let's jump into the newly created **app/Http/Resources/JSONAPICollection.php** file and make sure that it lives up to our existing collection for authors. In contrast to our **AuthorsCollection**, we set the `collects` property to **JSONAPIResource** but we can keep the **toArray** as it is, since it's just adding the **data** top-level member, which we need:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class JSONAPICollection extends ResourceCollection
{
    public $collects = JSONAPIResource::class;

    public function toArray($request)
    {
        return [
            'data' => $this->collection,
        ];
    }
}
```

Let's go into our **app/Http/Controllers/AuthorsController.php** and replace the **AuthorsCollection** to **JSONAPICollection** in the **index** method like this:

```
public function index()
{
    $authors = QueryBuilder::for(Author::class)->allowedSorts([
        'name',
        'created_at',
        'updated_at',
    ])->jsonPaginate();
    return new JSONAPICollection($authors);
}
```

Let's run PHPUnit again and everything should still be passing, making our refactoring successful.

Right now, it doesn't seem like we have gained as much: we just took two



classes and made two new. Let's see if this changes when we begin to refactor books next.

## *Books*

Previously, we created a new convention, which is that all models that need to be used with our `JSONAPIResource` must extend `AbstractAPIModel`, so let's start by making our **app/Book.php** model extend this class and also implement the **type** method. Depending on your IDE or editor, this might show an error until you implement the method. This is also one of the conveniences of having an abstract method. The extension of the **AbstractAPIModel** class and implementation of the **type** method should look like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends AbstractAPIModel
{
    protected $fillable = [
        'title',
        'description',
        'publication_year',
    ];

    public function authors()
    {
        return $this->belongsToMany(Author::class);
    }

    /**
     * @return string
     */
}
```

```

public function type()
{
    return 'books';
}
}

```

Running PHPUnit, our tests should still be passing.

Go into **app/Http/Controllers/BooksController.php** and let's replace **BookResource** with **JSONAPIResource** in the **show** method:

```

public function show($book)
{
    $query = QueryBuilder::for(Book::where('id', $book))
        ->allowedIncludes('authors')
        ->firstOrFail();
    return new JSONAPIResource($query);
}

```

Running PHPUnit, a couple of tests will fail, complaining about the missing **relationship** member and the **included** member. Let's do some refactoring so that these tests will pass, but first let's just take a look at our **BooksResource.php**, especially the **relationship** member in the **toArray** method:

```

'relationships' => [
    'authors' => [
        'links' => [
            'self' => route(
                'books.relationships.authors',
                ['id' => $this->id]
            ),
            'related' => route(
                'books.authors',

```

```

        ['id' => $this->id]
    ),
],
'data' => AuthorsIdentifierResource::collection($this->
    whenLoaded('authors')),
],
]

```

This is very specific for our books resource, especially the **authors** relationship. If we are to make this a bit more general, we would still need a place to define that the relationship to authors exists and any other relationship for that matter, since these will increase. If we cannot do it in the resource class, then where could we do it?

To think a little ahead, let's take a look at our **app/Http/Controllers/-BooksController.php** controller again. In both the **index** and **show** methods, we are using **allowedSorts** and **allowedIncludes** to convey which attributes are sortable and which models can be included. This seems like something that belongs to a config file, so that it can easily be fetched when a request comes into the system. Placing this on the model would require us to have an instance of the model when trying to find out which sorts or includes are allowed, and in some cases that is not possible. So let's create a config file where we can give all the details per resource, such as relationships, allowed sorts and includes, and so on. We do this by creating a **config/jsonapi.php** file. However, this time we are not interested in a class, but rather a completely empty PHP file.

This file should return an array, and inside that array, we can define our configuration. It could be that we need to configure more things for our API, so let's make sure that we set a root key that conveys that we are working with **resources**, and then list out resources as children under this key:

```
<?php
return [
    'resources' => [
        'authors' => [],
        'books' => [],
    ]
];
```

Under each resource, we can then define the relationships and allowed sorts and so on, when we get there.

Let's go back to our **app/Http/Resources/BooksResource.php** file and take a look at relationships again. Looking at the **authors** relationship, we actually only need to know two things in order to create this from a config file. We need to know the type of the related resource, and then we need to know the name of the relationship method on the model. The rest of the information can be fetched from the resources' own type.

Go back into the **config/jsonapi.php** and add the following:

```
<?php
return [
    'resources' => [
        'authors' => [],
        'books' => [
            'relationships' => [
                [
                    'type' => 'authors',
                    'method' => 'authors',
                ]
            ]
        ]
    ]
];
```

Here, we have added a **relationships** array to contain all relationships for the books resource. We have added an array as child, which contains the **type** of the related resource and the **method** to be called on the **app/Book.php** model.

Back in our **app/Http/Resources/JSONAPIResource.php** file, we can then use this to map out the relationship details for each relationship listed in our config file like this:

```
private function prepareRelationships(){
    return collect(config("jsonapi.resources.{${this->type()}}.
        relationships"))
    ->flatMap(function($related){
        $relatedType = $related['type'];
        $relationship = $related['method'];
        return [
            $relatedType => [
                'links' => [
                    'self' => route(
                        "${this->type()}.relationships.${$relatedType}
                    )",
                    ['id' => $this->id]
                ),
                'related' => route(
                    "${this->type()}.${$relatedType}",
                    ['id' => $this->id]
                ),
            ],
            'data' => '',
        ],
    ];
};
```

We use a Laravel collection to collect the relationship array from our config file. Then we use the **flatMap** method to map the details from the config file into a relationship that adheres to the JSON:API specification, like we had before. We want each relationship to have a correctly named key e.g. **authors**.

Therefore, we return an array with the correct key, which will be flattened afterward so that the key and value will be merged correctly with any other relationship that might be:

```
return [
    $relatedType => [
        ...
    ],
];
```

We are missing the **data** member, so let's do something about this. Looking at our **app/Http/Resources/BooksResource.php**, we return a collection of **AuthorsIdentifierResource**. We must create a more general class of this as well. Go into your terminal and create the resource like this:

```
php artisan make:resource JSONAPIIdentifierResource
```

Let's open this resource and implement it by copying everything in the **toArray** method of **app/Http/Resources/AuthorsIdentifierResource.php** into the **toArray** of **app/Http/Resources/JSONAPIIdentifierResource.php** like this:

```
public function toArray($request)
{
    return [
        'id' => (string)$this->id,
        'type' => 'authors',
    ];
}
```

Then we just need to change the value of **type** to the call to the **type** method

on our model like this:

```
public function toArray($request)
{
    return [
        'id' => (string)$this->id,
        'type' => $this->type(),
    ];
}
```

And we are done with our general resource to return resource identifier objects. We can then jump back into **app/Http/Resources/JSONAPIResource.php** and implement the **data** member like this:

```
private function prepareRelationships(){
    return collect(config("jsonapi.resources.{ $this->type() }.relationships"))->flatMap(function($related){
        $relatedType = $related['type'];
        $relationship = $related['method'];
        return [
            $relatedType => [
                'links' => [
                    'self' => route(
                        "{$this->type()}.relationships.{ $relatedType }",
                        ['id' => $this->id]
                    ),
                    'related' => route(
                        "{$this->type()}.{ $relatedType }",
                        ['id' => $this->id]
                    ),
                ],
                'data' => !$this->whenLoaded($relationship)
                    instanceof MissingValue ?
                    JSONAPIIdentifierResource::collection($this->
```

```

        $relationship)) : new MissingValue(),
    ],
];
});
}

```

The implementation of the **data** member might seem a bit long here, but we are actually just using a ternary operator check if the relationship has been loaded. If not, we want to return a **MissingValue** object, since it makes Laravel remove the **data** member entirely if there aren't included any related resources or else we just return a collection of **JSONAPIIdentifierResource** we just created.

If we run PHPUnit, our tests are still failing, but this is because we haven't done anything about the **included** member, so let's do that now.

Most of the code for our included member, we can copy over from our **BooksResource** class, so let's copy over the **relations**, **included** and **with** methods from **BooksResource** to **JSONAPIResource**. We can reuse almost everything. We just need to adjust the **relations** so that these are being fetched from our collection instead and mapped to the correct collection class like this:

```

private function relations()
{
    return collect(config("jsonapi.resources.{${this->type()}}.
        relationships"))
    ->map(function($relation){
        return JSONAPIResource::collection($this->whenLoaded(
            $relation['method']));
    });
}

```

Here, we collect the relationships again and map these out into a collection of



**JSONAPIResource** to reuse this class for each resource object, which we need to return in our data member.

If we run PHPUnit now, our tests should pass again.

Let's go into our **app/Http/Controllers/BooksController.php** and change the returned resource from **BooksResource** to **JSONAPIResource** in our **store** and **update** methods like this:

```
<?php

namespace App\Http\Controllers;

use App\Book;
use App\Http\Requests\CreateBookRequest;
use App\Http\Requests\UpdateBookRequest;
use App\Http\Resources\BooksCollection;
use App\Http\Resources\BooksResource;
use App\Http\Resources\JSONAPIResource;
use Illuminate\Http\Request;
use Spatie\QueryBuilder\QueryBuilder;

class BooksController extends Controller
{
    public function index()
    {
        $books = QueryBuilder::for(Book::class)->allowedSorts([
            'title',
            'publication_year',
            'created_at',
            'updated_at',
        ])->allowedIncludes('authors')->jsonPaginate();
        return new BooksCollection($books);
    }

    public function store(CreateBookRequest $request)
    {
```

```

        $book = Book::create([
            'title' => $request->input('data.attributes.title'),
            'description' => $request->input('data.attributes.description'),
            'publication_year' => $request->input('data.attributes.publication_year'),
        ]);
        return (new JSONAPIResource($book))
            ->response()
            ->header('Location', route('books.show', [
                'book' => $book,
            ]));
    }

    public function show($book)
    {
        $query = QueryBuilder::for(Book::where('id', $book))
            ->allowedIncludes('authors')
            ->firstOrFail();
        return new JSONAPIResource($query);
    }

    public function update(UpdateBookRequest $request, Book $book)
    {
        $book->update($request->input('data.attributes'));
        return new JSONAPIResource($book);
    }

    public function destroy(Book $book)
    {
        $book->delete();
        return response(null, 204);
    }
}

```

If we run PHPUnit again, all of our tests should still be passing.

We only need to implement the collection part of our books resource, and we are done refactoring resources, so let's jump into our **app/Http/Resources/-**

**BooksCollection.php** file and take a look at what happens here:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;
use Illuminate\Http\Resources\MissingValue;

class BooksCollection extends ResourceCollection
{
    public $collects = BooksResource::class;

    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'included' => $this->mergeIncludedRelations($request),
        ];
    }

    private function mergeIncludedRelations($request)
    {
        $includes = $this->collection->flatMap->included($request)->
            unique()->values();
        return $includes->isEmpty() ? $includes : new
            MissingValue();
    }
}
```

The thing to note here is, of course, our own **mergeIncludedRelations** method, but since it relies on the **included** method we just ported over to our **JSON-APIResource** class, we can actually just copy our **mergeIncludedRelations** method to our **JSONAPICollection** class and add the key to the array as well:

```

<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;
use Illuminate\Http\Resources\MissingValue;

class JSONAPICollection extends ResourceCollection
{
    public $collects = JSONAPIResource::class;

    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'included' => $this->mergeIncludedRelations($request),
        ];
    }

    private function mergeIncludedRelations($request)
    {
        $includes = $this->collection->flatMap->included($request)->
            unique()->values();
        return $includes->isEmpty() ? $includes : new
            MissingValue();
    }
}

```

Let's go back into our **app/Http/Controllers/BooksController.php** file and change the returned collection in **index** to our **JSONAPICollection** like this:

```

public function index()
{
    $books = QueryBuilder::for(Book::class)->allowedSorts([
        'title',
        'publication_year',
        'created_at',
    ])
    ->paginate(10);

    return new JSONAPICollection($books);
}

```

```

        'updated_at',
    ]->allowedIncludes('authors')->jsonPaginate();
    return new JSONAPICollection($books);
}

```

When we then run PHPUnit, our tests should pass.

We now have a couple of controllers which is: **app/Http/Controllers/BooksAuthorsRelatedController.php** and **app/Http/Controllers/BooksAuthorsRelationshipsController.php**. In the **app/Http/Controllers/BooksAuthorsRelatedController.php** file, change the **index** method to this:

```

public function index(Book $book)
{
    return new JSONAPICollection($book->authors);
}

```

In the **app/Http/Controllers/BooksAuthorsRelationshipsController.php**, change the **index** method to this:

```

public function index(Book $book)
{
    return JSONAPIIdentifierResource::collection($book->authors);
}

```

Let's delete the following files:

- **app/Http/Resources/AuthorsCollection.php**
- **app/Http/Resources/AuthorsIdentifierResource.php**
- **app/Http/Resources/AuthorsResource.php**
- **app/Http/Resources/BooksCollection.php**

- `app/Http/Resources/BooksResource.php`

And run PHPUnit to see if anything breaks. All tests should be passing which means that we have successfully refactored the resources and collections above down into a **JSONAPIResource** for all resources, **JSONAPICollection** for all resource objects collection and **JSONAPIIdentifierResource** for all resource identifier objects.

Awesome! We can now move on to refactoring our controllers.

## Refactoring controllers

If we look at our controllers, we are repeating a lot of the same code in almost all of them. Repeating code in controllers cannot be avoided completely, but it would be nice to refactor the parts that involve the conventions of the JSON:API specification so that these could be in a service class that then could be called from our controllers. Then, we will only have the implementation in one place and when we need to implement new resources, the work is done for us, and we can just reuse the service class. Let's do this now, and just like with resources, we will begin with authors to get a nice and easy start.

Before we do that, let's create our new **app/Services/JSONAPIService.php** class, so that it's ready for us to create methods in it:

```
<?php

namespace App\Services;

class JSONAPIService
{

}
```

Let's then go into our **app/Http/Controllers/AuthorsController.php** and add a constructor to the top of the file, where we inject our services file and initialize it to a **service** property like this:

```
<?php

namespace App\Http\Controllers;

...

class AuthorsController extends Controller
{
    private $service;

    public function __construct(JSONAPIService $service)
    {
        $this->service = $service;
    }

    ...
}
```

The cool thing about this is that Laravel will take care of injecting the service into our controller whenever a request comes in, without us having to lift a finger. We can just begin to use the service class in our methods from now on.

### *The Show method*

Let's start by refactoring the **show** method first:

```
public function show(Author $author)
{
    return new JSONAPIResource($author);
}
```

You could argue that this is so simple that it does not need any refactoring, and you would be right, but it is only this simple because the relationship hasn't been implemented and we will do that soon, so let's just do the refactoring.

In our **app/Services/JSONAPIService.php**, let's create a new method and copy over the contents from our **show** method like this:

```
public function fetchResource($model)
{
    return new JSONAPIResource($model);
}
```

Back in our controller, let's call this method like this:

```
public function show(Author $author)
{
    return $this->service->fetchResource($author);
}
```

Run PHPUnit and all the tests are passing still.

### *The Index method*

Let's move on to the **index** method then:



```

public function index()
{
    $authors = QueryBuilder::for(Author::class)->allowedSorts([
        'name',
        'created_at',
        'updated_at',
    ])->jsonPaginate();
    return new JSONAPICollection($authors);
}

```

For now, let's just copy the contents and go over to our service class and create a method for it like this:

```

public function fetchResources(string $modelClass)
{
    $authors = QueryBuilder::for(Author::class)->allowedSorts([
        'name',
        'created_at',
        'updated_at',
    ])->jsonPaginate();
    return new JSONAPICollection($authors);
}

```

We need another way of getting the array for the **allowedSorts** method, since this is specific for the resource itself. Since we have already made a config file for this, why don't we add a key to our **authors** associated array for our allowed sorts like this:

```

<?php
return [
    'resources' => [
        'authors' => [
            'allowedSorts' => [

```

```

        'name',
        'created_at',
        'updated_at',
    ],
],
'books' => [
    'relationships' => [
        [
            'type' => 'authors',
            'method' => 'authors',
        ]
    ]
]
]
];

```

Then we need to reference this from our service class. This can be done through the **config** helper function, using a string like this:

```

public function fetchResources(string $modelClass, string $type)
{
    $models = QueryBuilder::for($modelClass)
        ->allowedSorts(config("jsonapi.resources.{$type}.
            allowedSorts"))
        ->jsonPaginate();
    return new JSONAPICollection($models);
}

```

We need to add another argument to the method, so we get the right type. Our controllers are specific for each resource, so we can just give the type through these.

Back in our controller, we will then call this method on our service class like this:

```
public function index()
{
    return $this->service->fetchResources(Author::class, 'authors');
}
```

Running PHPUnit our tests should be passing.

### *The Store method*

Moving on to the **store** method, let's first copy the contents to our service class in a new method like this:

```
public function createResource(string $modelClass, array
    $attributes)
{
    $author = Author::create([
        'name' => $request->input('data.attributes.name'),
    ]);
    return (new JSONAPIResource($author))
        ->response()
        ->header('Location', route('authors.show', [
            'author' => $author,
        ]));
}
```

Then, we need to change the parts that are too specific. Instead of **Author**, we have added a **modelClass** argument and an **attributes** argument as well. These can be used to dynamically create the new model and add the attributes of the new model. The newly created model can then be used for returning the correct response like this:

```

public function createResource(string $modelClass, array
    $attributes)
{
    $model = $modelClass::create($attributes);
    return (new JSONAPIResource($model))
        ->response()
        ->header('Location', route("{ $model->type() }.show", [
            Str::singular($model->type()) => $model,
        ]));
}

```

The use of the **Str::singular** method is a bit special — we use this to be able to add the right route parameter, which is singular.

Back in our controller, we can call the method of the service class like this:

```

public function store(CreateAuthorRequest $request)
{
    return $this->service->createResource(Author::class, $request->
        input('data.attributes'));
}

```

We give the class name and then fetch the attributes of the request.

Run PHPUnit and the tests should be passing.

### *The Update method*

The update method is also fairly simple: it can be moved to our service class like this:

```
public function updateResource($model, $attributes)
{
    $model->update($attributes);
    return new JSONAPIResource($model);
}
```

Again, we leverage our arguments to pass in the needed data. This time, we get a model because of route-model binding, but we reuse the concept of passing attributes.

In our controller, we can call this method on our service class like this:

```
public function update(UpdateAuthorRequest $request, Author $author
)
{
    return $this->service->updateResource($author, $request->input('
        data.attributes'));
}
```

If we run PHPUnit again our tests are still passing.

### *The Destroy method*

Our destroy method is also fairly simple. We don't have to change anything about it either: we can just copy it to our service class and use the model from route-model binding to delete the model like this:

```
public function deleteResource($model)
{
    $model->delete();
    return response(null, 204);
}
```

```
}
```

In our controller, we can call the method on our service like this:

```
public function destroy(Author $author)
{
    return $this->service->deleteResource($author);
}
```

Running PHPUnit will show us that our tests are still passing. We have then successfully refactored our **AuthorsController**, so let's move on to our **BooksController**, which will add a bit more to our service methods.

## Books Controller

Looking at the **index** method of our **app/Http/Controllers/BooksController**, we see some other allowed sorts. Let's copy these into our config file first like this:

```
<?php
return [
    'resources' => [
        'authors' => [
            'allowedSorts' => [
                'name',
                'created_at',
                'updated_at',
            ],
        ],
        'books' => [
            'allowedSorts'=> [
                'title',
```

```

        'publication_year',
        'created_at',
        'updated_at',
    ],
    'relationships' => [
        [
            'type' => 'authors',
            'method' => 'authors',
        ]
    ]
]
];

```

Here, we are also calling the **allowedIncludes** with relations to be included, and we should transfer this to our config file as well:

```

<?php
return [
    'resources' => [
        'authors' => [
            'allowedSorts' => [
                'name',
                'created_at',
                'updated_at',
            ],
        ],
    ],
    'books' => [
        'allowedSorts' => [
            'title',
            'publication_year',
            'created_at',
            'updated_at',
        ],
        'allowedIncludes' => [
            'authors'
        ],
    ],
];

```

```

        'relationships' => [
            [
                'type' => 'authors',
                'method' => 'authors',
            ]
        ]
    ]
];

```

Then, we should go to our **app/Services/JSONAPIService.php** file and add the **allowIncludes** method to our **QueryBuilder** chain like this:

```

public function fetchResources(string $modelClass, string $type)
{
    $models = QueryBuilder::for($modelClass)
        ->allowedSorts(config("jsonapi.resources.{$type}.
            allowedSorts"))
        ->allowedIncludes(config("jsonapi.resources.{$type}.
            allowedIncludes"))
        ->jsonPaginate();
    return new JSONAPICollection($models);
}

```

Back in the controller, we need to add the constructor to the top of the class like we did in the **AuthorsController** like this:

```

<?php

namespace App\Http\Controllers;

...

```



```

class BooksController extends Controller
{
    private $service;

    public function __construct(JSONAPIService $service)
    {
        $this->service = $service;
    }

    ...
}

```

Then, in the **index** method, we call the **fetchResources** method on our service like this:

```

public function index()
{
    return $this->service->fetchResources(Book::class, 'books');
}

```

For the **store** method, we don't need to do anything besides replacing the contents with the call to our service class, just like we do in our **AuthorsController** like this:

```

public function store(CreateBookRequest $request)
{
    return $this->service->createResource(Book::class, $request->
        input('data.attributes'));
}

```

Running PHPUnit, our tests should still pass.

For the **show** method, we have a bit more complex implementation, which we will have to move over to our service class. We also have to make some changes to our arguments in order to support both an instance of a model or a model class. We do this like this:

```
public function fetchResource($model, $id = 0, $type = '')
{
    if($model instanceof Model){
        return new JSONAPIResource($model);
    }

    $query = QueryBuilder::for($model::where('id', $id))
        ->allowedIncludes(config("jsonapi.resources.{$type}.
            allowedIncludes"))
        ->firstOrFail();
    return new JSONAPIResource($query);
}
```

Then, in our controller, we can call the method on our service class like this:

```
public function show($book)
{
    return $this->service->fetchResource(Book::class, $book, 'books
        ');
}
```

Let's run PHPUnit which should show us that everything passes.

For both the **update** and **destroy** methods, we can call our service class, so we don't need to make any changes here:

```

public function update(UpdateBookRequest $request, Book $book)
{
    return $this->service->updateResource($book, $request->input('
        data.attributes'));
}

public function destroy(Book $book)
{
    return $this->service->deleteResource($book);
}

```

Running PHPUnit should show that all the tests are passing.

## BooksAuthorsRelationshipsController

It's time to look at relationships, so let's open up our **app/Http/Controllers/BooksAuthorsRelationshipsController.php** file to see how we can refactor this:

```

<?php

namespace App\Http\Controllers;

use App\Book;
use App\Http\Requests\UpdateBooksAuthorsRelationshipsRequest;
use App\Http\Resources\AuthorsIdentifierResource;
use App\Http\Resources\JSONAPIIdentifierResource;
use Illuminate\Database\QueryException;
use Illuminate\Http\Request;

class BooksAuthorsRelationshipsController extends Controller
{

    public function index(Book $book)
    {

```

```

        return JSONAPIIdentifierResource::collection($book->authors)
            ;
    }

    public function update(UpdateBooksAuthorsRelationshipsRequest
        $request, Book $book)
    {
        $ids = $request->input('data.*.id');
        $book->authors()->sync($ids);
        return response(null, 204);
    }
}

```

If we take a look at the **index** method, it does not do a lot and it's actually pretty concise with what it is doing. We will still move it, because our service class already contains all the JSON:API specific code so this is where this belongs as well.

We will move this over to our service class like this:

```

public function fetchRelationship($model, string $relationship)
{
    return JSONAPIIdentifierResource::collection($model->
        $relationship);
}

```

We leverage PHP's dynamic abilities to take a string in an argument and then call the method on the model with the method name that matches that string.

Back in our controller, we need to inject our **JSONAPIService** class through the constructor like this:

```

<?php
namespace App\Http\Controllers;

...

class BooksAuthorsRelationshipsController extends Controller
{
    private $service;

    public function __construct(JSONAPIService $service)
    {
        $this->service = $service;
    }
    ...
}

```

Then in the **index** method, we call the **fetchRelationship** method on our service class like this:

```

public function index(Book $book)
{
    return $this->service->fetchRelationship($book, 'authors');
}

```

Let's then run PHPUnit again and see what happens. The tests are passing like before, so we can move on to the next method.

The update method is a bit special and the reason is that the code is very specific for the type of relationship, which in this case is a many to many. So maybe we should move this code to a **updateManyToManyRelationships** method like this:

```
public function updateManyToManyRelationships($model, $relationship
    , $ids)
{
    $model->$relationship()->sync($ids);
    return response(null, 204);
}
```

Here, we leverage the dynamic nature of PHP again with the **relationship** argument. This time, we are calling a method on the model with a method name corresponding to the incoming string and afterward we are calling sync on this with the given IDs.

In our controller, we can then call this method on our service class like this:

```
public function update(UpdateBooksAuthorsRelationshipsRequest
    $request, Book $book)
{
    return $this->service
        ->updateManyToManyRelationships($book, 'authors', $request->
            input('data.*.id'));
}
```

Our controller will take care of getting the data of the request, and then our service class will take care of the rest. If we run PHPUnit again, the tests will still pass.

## BooksAuthorsRelatedController

Let's take care of the last controller now. Fortunately, this is an easy one. Open up **app/Http/Controllers/BooksAuthorsRelatedController.php** and take a look at the **index** method:

```
public function index(Book $book)
{
    return new JSONAPICollection($book->authors);
}
```

Again, it's very concise but as we mentioned earlier, let's move it to our service class since it fits better into that class' concerns. So in our service class, we can create the method like this:

```
public function fetchRelated($model, $relationship)
{
    return new JSONAPICollection($model->$relationship);
}
```

Like before, we use a relationship string to call the right relationship on our model and return this in a collection. Back in our controller, we have to inject our **JSONAPIService** class into a constructor and afterward we can call the method on our service class like this:

```
public function index(Book $book)
{
    return $this->service->fetchRelated($book, 'authors');
}
```

If we run PHPUnit, our tests are still passing, nothing is breaking, and we have now successfully refactored our JSON:API specific implementations into a dedicated service class, which we can use from now on. You might be thinking why we still keep as many controllers and controllers specific to each resource. We still want these because they give us the ability to do application specific tasks. These can vary from resource to resource and request to request.

Our requests, that we use for validation on the other, do contain some repeating elements that could benefit from some refactoring, so let's take care of these next.

## Refactoring requests

If we take a look at our requests, we can see that we do a lot of repetitions — especially when it comes to the members that are required of the request document, such as the **data** top level member, the need for the **type** member, and sometimes also the **ID** member. It would be nice, if we didn't have to include these everytime we write our validation request. In the last sections, we have created and built onto a config file for all the specific things about our resources, such as relationships and which attributes can be sorted and so forth. The thing that is specific from resource to resource is the attributes, so maybe we could also place our specific validation rules in the config file and leverage this to create fewer request classes. Let's work on that now.

First, let's jump into the terminal and create a new request like this:

```
php artisan make:request JSONAPIRequest
```

Go into the newly created **app/Http/Requests/JSONAPIRequest.php** file and let's change the **authorize** method to return **true**. Then let's jump into **app/Http/Requests/UpdateBookRequest.php** and copy the first four **rules** of the **rules** method over to the **rules** method of our **JSONAPIRequest** class like this:

```
public function rules()  
{
```



```

return [
    'data' => 'required|array',
    'data.id' => 'required|string',
    'data.type' => 'required|in:books',
    'data.attributes' => 'required|array',
];
}

```

To be able to test whether this request works or not, we should add it to one of our controllers. Let's start by adding it to our store method in our **app/Http/Controllers/AuthorsController.php** file like this:

```

public function store(JSONAPIRequest $request)
{
    return $this->service
        ->createResource(Author::class, $request->input('data.attributes
            '));
}

```

Then, run PHPUnit and you will see that a bunch of tests are failing. This is because we are not validating the attributes yet. But before we do so, let's take a look at the rules we have just added first.

We have added the request to a **store** method, which receives a **POST** and where the **ID** is not needed, since we will let the server decide this. So we need a way to convey that the **ID** should not be validated for a **POST** request, but a patch request only.

This can be done through a simple ternary operator like this:

```

public function rules()
{
    return [
        'data' => 'required|array',
        'data.id' => ($this->method() === 'PATCH') ? 'required|
            string' : 'string',
        'data.type' => 'required|in:books',
        'data.attributes' => 'required|array',
    ];
}

```

In a request we have access to `method`, which we can use to determine which rules the ID has in the given request.

Next, let's look at the **type** member. Here, we are too specific and only validating for the **books** type. We want to validate that the right type comes in and that the type is an allowed one. In our config file, we have listed all of our resources, which coincidentally match our types as well.

We could use these keys as a lookup table and only allow the types that are given as resources like this:

```

public function rules()
{
    return [
        'data' => 'required|array',
        'data.id' => ($this->method() === 'PATCH') ? 'required|
            string' : 'string',
        'data.type' => ['required',Rule::in(array_keys(config('
            jsonapi.resources')))],
        'data.attributes' => 'required|array',
    ];
}

```

Here, it's necessary to give an array instead of strings, which we have used so far. The reason is that we now have an array of types and, to be able to use the in rule again, we need to use Laravel's **Rule** class to call it with an array. Here, we take the keys of our resources to validate against. This means that the consumer can only use the types we have defined in our config file.

We are not done yet — we still need a way to validate the attributes of each resource. Like we mentioned earlier, we can move our specific validation rules to our config file instead. But before we do this, we need to figure out what we do in the case of a POST request and a PATCH request, since the attributes are required when a POST request comes in, but they are optional when a PATCH request comes in. In this case, we could simply divide our rule array into two separate arrays: one for creating and one for updating.

Following this thought, our config file will become this:

```
<?php
return [
    'resources' => [
        'authors' => [
            'allowedSorts' => [
                'name',
                'created_at',
                'updated_at',
            ],
            'validationRules'=> [
                'create' => [
                    'data.attributes.name' => 'required|string',
                ],
                'update' => [
                    'data.attributes.name' => 'sometimes|required|
                        string',
                ]
            ],
        ],
    ],
],
```

```

'books' => [
    'allowedSorts'=> [
        'title',
        'publication_year',
        'created_at',
        'updated_at',
    ],
    'allowedIncludes' => [
        'authors'
    ],
    'validationRules'=> [
        'create' => [
            'data.attributes.title' => 'required|string',
            'data.attributes.description' => 'required|
                string',
            'data.attributes.publication_year' => 'required|
                string',
        ],
        'update' => [
            'data.attributes.title' => 'sometimes|required|
                string',
            'data.attributes.description' => 'sometimes|
                required|string',
            'data.attributes.publication_year' => 'sometimes
                |required|string',
        ]
    ],
    'relationships' => [
        [
            'type' => 'authors',
            'method' => 'authors',
        ]
    ]
]
];

```

We add the **validationRules** associative array to both resources, and on these arrays, we add the **create** and **update** associative arrays that will contain our

validation rules for each scenario.

Let's implement this into our **JSONAPIRequest** class. To do this, we would need to merge the correct array from our config file with the existing array with the existing rules. We also need to make sure that we select the correct array, according to which method is being used. We need to select the **create** array when there's a POST request and the **update** when there's a PATCH request.

We have written this implementation like this:

```
public function rules()
{
    $rules = [
        'data' => 'required|array',
        'data.id' => ($this->method() === 'PATCH') ? 'required|
            string' : 'string',
        'data.type' => ['required', Rule::in(array_keys(config('
            jsonapi.resources')))],
        'data.attributes' => 'required|array',
    ];

    return $this->mergeConfigRules($rules);
}

public function mergeConfigRules(array $rules): array
{
    $type = $this->input('data.type');

    if ($type && config("jsonapi.resources.{ $type }")) {

        switch ($this->method) {
            case 'PATCH':
                $rules = array_merge($rules, config("jsonapi.
                    resources.{ $type }.validationRules.update"));
            default:
                // Do nothing
        }
    }

    return $rules;
}
```

```

        break;

    case 'POST':
    default:
        $rules = array_merge($rules, config("jsonapi.
            resources.{ $type }.validationRules.create"));
        break;
    }

}

return $rules;
}

```

We save the existing rules in a variable and send that to our newly created **mergeConfigRules** method. Here, we get the type from the request and checks that it is given. Then we use a switch to merge the rules according to which method that has been chosen and lastly we always default to the POST rules, since these require every attribute to be given.

If you run PHPUnit, our tests should all pass again.

Now, we can replace all request classes in our controllers **store** and **update** methods with our new JSONAPIRequest, except for the controllers that handle relationships. We will return to those in a minute. Let's run PHPUnit again to see if anything has broken.

Great! Everything is passing, so let's look at relationships.

### *Relationship requests*

The reason we have not taken relationships into account just yet is that they require a different request document. Here, we don't send resource objects but resource identifier objects instead. We might be able to have it all in the **JSONAPIRequest** class, but it would not be pretty. Besides, we think it's ok to

have this request for relationships only to separate the concerns a bit.

Let's get out into the terminal once again and run the following artisan command:

```
php artisan make:request JSONAPIRelationshipRequest
```

While we are at it, why don't we open the newly created **app/Http/Requests/JSONAPIRelationshipRequest.php** file, make the **authorized** method return true, and then copy the contents of the **rules** method of the **app/Http/Requests/UpdateBooksAuthorsRelationshipsRequest.php** file into our **rules** method like this:

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class JSONAPIRelationshipRequest extends FormRequest
{
    public function authorize()
    {
        return true;
    }

    public function rules()
    {
        return [
            'data' => 'present|array',
            'data.*.id' => 'required|string',
            'data.*.type' => 'required|in:authors',
        ];
    }
}
```

```
    }
}
```

Just like our **JSONAPIRequest**, we need to change the **type** member from being too specific. Here, we can just copy the rule from our **JSONAPIRequest** class like this:

```
public function rules()
{
    return [
        'data' => 'present|array',
        'data.*.id' => 'required|string',
        'data.*.type' => ['required', Rule::in(array_keys(config('
            jsonapi.resources')))],
    ];
}
```

This is actually it. We don't have to do anything else besides go into our **app/Http/Controllers/BooksAuthorsRelationshipsController.php** and change the request from **UpdateBooksAuthorsRelationshipsRequest** to **JSONAPIRelationshipRequest** and then run PHPUnit.

We have one test that fails, namely the **it\_validates\_that\_the\_type\_member\_has\_a\_value\_of\_authors\_when\_updating\_a\_relationship** of our **tests/Features/BooksRelationshipTest.php** file. Let's just take a look at this for a second:

```
public function
    it_validates_that_the_type_member_has_a_value_of_authors_when_updating_a_rel
    ()
{
```



```

$book = factory(Book::class)->create();
$authors = factory(Author::class, 5)->create();

$user = factory(User::class)->create();
Passport::actingAs($user);

$this->patchJson('/api/v1/books/1/relationships/authors',[
    'data' => [
        [
            'id' => '5',
            'type' => 'books',
        ],
    ],
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])->assertStatus(422)->assertJson([
    'errors' => [
        [
            'title' => 'Validation Error',
            'details' => 'The selected data.0.type is invalid.',
            'source' => [
                'pointer' => '/data/0/type',
            ],
        ],
    ],
]);
}

```

Can you spot what is wrong?

We are testing that when updating an author's relationship to a book, we must give the **authors** value in the **type** member of our request document. In this case, we give **books** in order to make the validation fail which will make the test pass. Because our old request implementation had authors hardcoded in and therefore **books** wasn't an option, the validation would fail, but now that we have our config file as a lookup table and **books** actually exist, the validation will not fail anymore. So to fix this test, we just have to give another

value instead of books. In our case, we have just replaced it with the word **random** like this:

```
...
$this->patchJson('/api/v1/books/1/relationships/authors', [
    'data' => [
        [
            'id' => '5',
            'type' => 'random',
        ],
    ]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])->assertStatus(422)
...
```

Running PHPUnit again, all of our tests should be passing.

We have now refactored all of our requests into two request classes. Let's delete all request files except for our **JSONAPIRequest.php** and **JSONAPIRelationshipRequest.php** files and run PHPUnit again.

Awesome! Everything is still passing.

We have now successfully refactored our resources into a couple of classes, refactored all of the JSON:API specific implementations from our controllers into a services class, refactored our requests into a couple of request classes we can reuse, and created a config file we can use for the specifics about our resources. We did this without breaking anything, because of our tests that always made sure to tell us when we broke something.

## Summary

We went through a lot in this chapter and a lot of new concepts when learning about the **Don't repeat yourself** principle. We hope you see the benefits of doing this together with tests.

We started out by refactoring our resources and collections and going from a bunch of resources and collection to only three classes: a **JSONAPIResource** class, which became our main resource class, a **JSONAPICollection** class for collecting our **JSONAPIResources**, and lastly, a **JSONAPIIdentifierResource** class for returning resource identifier objects.

We refactored these classes in a way that can be used for any resource in an API adhering to the JSON:API specification.

From there, we moved on and refactored our code in our controllers and extracted this into a **JSONAPIService** class we could reuse in all of our controllers, so we didn't repeat the same code in our controllers for every resource we have.

We refactored our requests classes from two request classes for each resource down to two resource classes in total: one for requests to resources and one for requests to relationships.

Lastly, we moved on and implemented the relationship between books and authors, leveraging our newly created classes after our refactor, which made this implementation smooth as butter.

Next, let's finish up the last part of our API and application. Here, we will also build the inverse relationship between books and authors, so that relationship is completely done.

## BUILD AN API WITH LARAVEL

\* \* \*

## 8

# Finishing up

In this chapter, much as the title indicates, we will be finishing up the last missing parts of the implementation of our API and application.

Now that we have implemented the relationship between books and authors, it's time to implement the inverse relationship. Here, we will revise some of the concepts we have just been through in the last chapter and show you how fast we can do the inverse implementation by using what is already written.

We will then begin the implementation of users and first implement the basic parts, such as UUID and the JSON:API parts.

Then, we will look at comments and how to implement these, leveraging all the previous concepts in the book. Next, we will implement **one-to-many** relationships as well as finish up the last parts from the JSON:API specification, creating and updating relationships through a resource's endpoints instead of relationship endpoints.

We will finish the last relationship parts to books and users.

Finally, we will take a look at Cross-Origin Resource Sharing, making sure that we can access our API from other domains.

There's a lot to go through, so let's get started!

## Authors Books Relationship

It's time for us to finish the relationship between books and authors. At the moment, it's only possible to fetch the relationship from the books side of things, so let's implement the inverse to enable, for instance, fetching all the books for an author. To do this, we will go a bit faster than we have done before. This is because almost everything has been implemented, especially after we have refactored everything. Also in terms of the tests, we can take everything from the **tests/Feature/BooksRelationshipsTest.php** and just rewrite it so that we are fetching authors instead of books. Essentially, we can just flip everything we are doing. We will be going through a couple of tests just so that we know that everything is implemented correctly, and then you will be working on your own. All of the concepts should be familiar to you since it's what we have already been through with books. It's just from the authors' side of things now.

Let's start by creating a new **tests/Feature/AuthorsRelationshipTest.php** feature test. Make sure to extend the **TestCase** class and add the use the **DatabaseMigrations** trait in the top of the class. Then, we should copy the first test from our **tests/Feature/BooksRelationshipsTest.php** and flip the relationship between books and authors, so we now have one author that has written **three** books, and we make the get request to our author's endpoint, including books instead of authors this time. Then, we need to change the contents of the **assertJson** method so that we are returning the resource of an **author** with a relationship to **books** with the right **links** and resource identifier objects in the **data** member. We have written this test like so:

```
/**
 * @test
```

```

* @watch
*/
public function
    it_returns_a_relationship_to_books_adhering_to_json_api_spec()
{
    $author = factory(Author::class)->create();
    $books = factory(Book::class, 3)->create();
    $author->books()->sync($books->pluck('id'));

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/authors/1?include=books', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200)
        ->assertJson([
            'data' => [
                'id' => '1',
                'type' => 'authors',
                'relationships' => [
                    'books' => [
                        'links' => [
                            'self' => route(
                                'authors.relationships.books',
                                ['id' => $author->id]
                            ),
                            'related' => route(
                                'authors.books',
                                ['id' => $author->id]
                            ),
                        ],
                    ],
                ],
            ],
            'data' => [
                [
                    'id' => $books->get(0)->id,
                    'type' => 'books'
                ],
                [
                    'id' => $books->get(1)->id,

```

```

        'type' => 'books'
    ]
    ]
    ]
    ];
}

```

You can choose to use the Laravel Test Watcher here — it will certainly make things a bit easier, since we want to stay in the code. Immediately, this test will fail since we haven't created any routes for our authors books side of the relationships, but we are referencing these through our **links** member in our test, so let's add these first.

Go into the **routes/api.php** file and create the following routes right underneath our existing routes for authors like this:

```

Route::middleware('auth:api')->prefix('v1')->group(function(){
    Route::get('/user', function (Request $request) {
        return $request->user();
    });

    // Authors
    Route::apiResource('authors', 'AuthorsController');

    Route::get('authors/{author}/relationships/books', '
        AuthorsBooksRelationshipsController@index')
        ->name('authors.relationships.books');
    Route::patch('authors/{author}/relationships/books', '
        AuthorsBooksRelationshipsController@update')
        ->name('authors.relationships.books');

    Route::get('authors/{author}/books', '
        AuthorsBooksRelatedController@index')

```



```

->name('authors.books');

// Books
Route::apiResource('books', 'BooksController');

Route::get('books/{book}/relationships/authors', '
    BooksAuthorsRelationshipsController@index')->name('books.
    relationships.authors');
Route::patch('books/{book}/relationships/authors', '
    BooksAuthorsRelationshipsController@update')->name('books.
    relationships.authors');

Route::get('books/{book}/authors', '
    BooksAuthorsRelatedController@index')->name('books.authors')
    ;

});

```

Again, we can copy the most from our books routes, since the concept is exactly the same:

1. We want to be able to make a GET request to the relationship and get the resource identifier objects of books related to the author
2. We want to be able to make a PATCH request to the relationship to update it without adding or deleting resources themselves
3. We want to be able to get the related resource objects

By copying routes, we also get the names for our new controllers, so let's just create these right away. Go to your terminal and run the following artisan commands:

```
php artisan make:controller AuthorsBooksRelationshipsController
```

```
php artisan make:controller AuthorsBooksRelatedController
```

Our test still fails because of our implementation of the **show** method in our **app/Http/Controllers/AuthorsController.php** file. Here, we are using the old implementation before any relationship has been set up, where we just return the model given from the route-model binding.

If we go into our service class, the implementation is this:

```
public function fetchResource($model, $id = 0, $type = '')
{
    if($model instanceof Model){
        return new JSONAPIResource($model);
    }

    $query = QueryBuilder::for($model::where('id', $id))
        ->allowedIncludes(config("jsonapi.resources.{$type}.
            allowedIncludes"))
        ->firstOrFail();
    return new JSONAPIResource($query);
}
```

Our controller is hitting the first conditional, which does not take relationships into consideration. In order to skip this and use the QueryBuilder to fetch relationships, we need to give a **model** class and **id** and a **type** instead.

Let's edit our controller to do this:

```
public function show($author)
{
    return $this->service->fetchResource(Author::class, $author, '
        authors');
```

```
}
```

Now our failing test is changing, where we are now getting a **400** status code back. This is because we are using the include query parameter to **include** our **books**. At the moment, it is not allowed to be included because we haven't added it to our config file yet. Let's do this and let's also add the relationships.

Go to your **config/jsonapi.php** file and add the following to your **authors** array:

```
'authors' => [
  'allowedSorts' => [
    'name',
    'created_at',
    'updated_at',
  ],
  'allowedIncludes' => [
    'books'
  ],
  'validationRules' => [
    'create' => [
      'data.attributes.name' => 'required|string',
    ],
    'update' => [
      'data.attributes.name' => 'sometimes|required|string',
    ]
  ],
  'relationships' => [
    [
      'type' => 'books',
      'method' => 'books',
    ]
  ]
],
```

Our test is now green and passing and we didn't even have to do that much

coding — merely following our own conventions of information in the config file and then calling the right methods in our controller.

Let's move on to the next test. Again, we follow the same concept of flipping books and authors, and we have written this test like this:

```
/**
 * @test
 * @watch
 */
public function
    a_relationship_link_to_books_returns_all_related_books_as_resource_id_object
    ()
{
    $author = factory(Author::class)->create();
    $books = factory(Book::class, 3)->create();
    $author->books()->sync($books->pluck('id'));

    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson('/api/v1/authors/1/relationships/books', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200)
        ->assertJson([
            'data' => [
                [
                    'id' => '1',
                    'type' => 'books',
                ],
                [
                    'id' => '2',
                    'type' => 'books',
                ],
            ],
        ]
```

```

        'id' => '3',
        'type' => 'books',
    ],
    ]
    });
}

```

The test is failing, receiving a **500** status code instead of a **200** status code.

Our test is hitting the route for the first controller, but it has not been implemented yet so let's go into **app/Http/Controllers/AuthorsBooksRelationshipsController.php** and start by adding a constructor where we inject our **JSONAPIService** class like this:

```

<?php

namespace App\Http\Controllers;

...

class AuthorsBooksRelationshipsController extends Controller
{

    /**
     * @var JSONAPIService
     */
    private $service;

    public function __construct(JSONAPIService $service)
    {
        $this->service = $service;
    }

    ...
}

```

Next, we need to add the **index** method. Here, we can repeat the concepts from our **BooksAuthorsRelationshipsController** where we call the **fetchRelationship** method on our service class passing in the author model we get from route-model binding.

Then, we pass a string indicating which method to call on the model for the relationship, which is **books** in this case:

```
public function index(Author $author)
{
    return $this->service->fetchRelationship($author, 'books');
}
```

This will make our test green and passing.

Let's move on to the next test, which is the one for modifying relationships. Again, we use the same concepts, copying from **BooksRelationshipsTest** and rewriting the test for authors instead of books. We have written this test like this:

```
/**
 * @test
 * @watch
 */
public function
    it_can_modify_relationships_to_authors_and_add_new_relationships
    ()
{
    $authors = factory(Author::class)->create();
    $books = factory(Book::class, 10)->create();

    $user = factory(User::class)->create();
    Passport::actingAs($user);
```

```

$this->patchJson('/api/v1/authors/1/relationships/books',[
    'data' => [
        [
            'id' => '5',
            'type' => 'books',
        ],
        [
            'id' => '6',
            'type' => 'books',
        ]
    ]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])->assertStatus(204);

$this->assertDatabaseHas('author_book', [
    'author_id' => 1,
    'book_id' => 5,
])->assertDatabaseHas('author_book', [
    'author_id' => 1,
    'book_id' => 6,
]);
}

```

The test is failing at the moment, because we haven't implemented the method in our controller yet, so let's do that. Right underneath the **index** method, let's create the following method, calling the **updateManyToManyRelationships** method on our service class. Here, we will give our author a string for the relationship, which is **books** in this case and then the data received in the request:

```

public function update(JSONAPIRelationshipRequest $request, Author
    $author)

```

```

{
    return $this->service
        ->updateManyToManyRelationships($author, 'books', $request->
            input('data.*.id'));
}

```

Our test is now green and passing.

By now, you should understand the concept of changing our tests from books to the inverse relationship, so we won't go very much into the details of the rest of the tests here in the book, but rather let you work through these on your own. If you get lost, you can always find help in our Github repository.

To be able to implement the last route and controller, we will jump ahead to the test that tests the related link. We have written it like this:

```

/**
 * @test
 * @watch
 */
public function
    it_can_get_all_related_books_as_resource_objects_from_related_link
    ()
    {
        $author = factory(Author::class)->create();
        $books = factory(Book::class, 3)->create();
        $author->books()->sync($books->pluck('id'));

        $user = factory(User::class)->create();
        Passport::actingAs($user);

        $this->getJson('/api/v1/authors/1/books', [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ])
    }

```



```

->assertStatus(200)
->assertJson([
  'data' => [
    [
      "id" => '1',
      "type" => "books",
      "attributes" => [
        'title' => $books[0]->title,
        'description' => $books[0]->description,
        'publication_year' => $books[0]->
          publication_year,
        'created_at' => $books[0]->created_at->
          toJSON(),
        'updated_at' => $books[0]->updated_at->
          toJSON(),
      ]
    ],
    [
      "id" => '2',
      "type" => "books",
      "attributes" => [
        'title' => $books[1]->title,
        'description' => $books[1]->description,
        'publication_year' => $books[1]->
          publication_year,
        'created_at' => $books[1]->created_at->
          toJSON(),
        'updated_at' => $books[1]->updated_at->
          toJSON(),
      ]
    ],
    [
      "id" => '3',
      "type" => "books",
      "attributes" => [
        'title' => $books[2]->title,
        'description' => $books[2]->description,
        'publication_year' => $books[2]->
          publication_year,
        'created_at' => $books[2]->created_at->

```

```

        toJSON(),
        'updated_at' => $books[2]->updated_at->
        toJSON(),
    ],
],
]);
}

```

This test is failing because we did not implement the method in the controller, so let's do that now. Here we can reuse the concept from our **BooksAuthorsRelatedController** where we make a call to the **fetchRelated** method on our service class, giving the Author model we get through route-model binding and then a string conveying which method to call on the model to get the relationship, which is **books** in this case:

```

public function index(Author $author)
{
    return $this->service->fetchRelated($author, 'books');
}

```

And our test passes. Great! We have now implemented the inverse relationship of books and authors and we are done with this relationship and authors all together.

Now, we need to implement the final resources. Users is maybe the most complex resource, because of the multitenant part of this, but it is necessary for us to be able to implement comments that have a relationship to books, which in turn will conclude the entire implementation for both our API and application. So let's move on to the users resource now.

## *Users*

It's time to take a look at users. This resource is a bit special, not only because it's used to gain access to our application, but because we will make some changes to this resource that we didn't need for the other ones.

In this section, we will implement **UUIDs** so that users with malicious intents cannot guess other users' IDs easily. Then, we will add roles so that we can handle both general users and administrators later on.

Lastly, we will implement the CRUD functionality of users and make this adhere to the JSON:API specification, using the service class we have already implemented.

## *Uuids*

We have already described what **UUID** is, so you should be familiar with the concept. Let's just get right into implementing **UUID** for our User model, and the best way to do this is to create a test first. For this purpose, we will be using a unit test to test the model. This way, we can focus on the **UUID** alone and not have to implement anything else first. We will be hitting the database to make things a bit easier for ourselves, which goes against the testing code in isolation part of unit tests, but we're ok with this.

Let's create a new **tests/Unit/Models/UsersTest.php** file and open it up in our editor. Then, we need to make sure that the class extends Laravel's **TestCase** class and also make sure to use the **DatabaseMigrations** trait like this:

```
<?php

namespace Tests\Unit\Models;
```

```

use Tests\TestCase;

class UsersTest extends TestCase
{
    use DatabaseMigrations;
}

```

Let's break down what we want to test:

- 1. We set up our world
  - a. We need a user to be able to assert against the **id**
- 2. We run the code we are testing here
  - a. We don't have anything to run so we skip this step
- 3. We assert against the result that
  - a. The **ID** is not an integer
  - b. The **ID** is 36 characters long, since **UUIDs** have this length

We have written the test like this:

```

/**
 * @test
 * @watch
 */
public function a_users_ID_is_a_UUID_instead_of_an_integer()
{
    $user = factory(User::class)->create();
    $this->assertFalse(is_integer($user->id));
    $this->assertEquals(36, strlen($user->id));
}

```

This test fails at the moment, but let's change that. Let's start in the **database/migrations/xxxx\_xx\_xx\_XXXXXX\_create\_users\_table.php** and

change our primary key in the up method like this:

```
public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->uuid('id')->primary();
        $table->string('name');
        $table->string('email')->unique();
        $table->timestamp('email_verified_at')->nullable();
        $table->string('password');
        $table->rememberToken();
        $table->timestamps();
    });
}
```

We exchange the **bigIncrements** method to the **uuid** method and chain the **primary** method on to this, to tell that the **ID** is still the table's primary key.

This is not enough to get our test to pass, so let's go into the **app/User.php** file and make the necessary changes.

Here, we need to tell Laravel that we are no longer using auto incrementing keys in our database.

We do this through the **incrementing** property, which takes a boolean. We also need to tell Laravel which datatype our primary key is, and now that we have changed this to a **UUID**, it will be a string instead of an integer. We can tell Laravel this through a **keyType** property like this:

```
<?php

namespace App;
```

```

use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    protected $fillable = [
        'name', 'email', 'password',
    ];

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

    protected $casts = [
        'email_verified_at' => 'datetime',
    ];

    public $incrementing = false;

    protected $keyType = 'string';
}

```

Our test is still failing, but this is because we are not providing any keys for the **ID**. Now, that we have told Laravel that it should no longer make the database handle the generation or auto incrementation of keys, we need to provide these. Models in Laravel go through various stages based on what we are telling them to do. We have events for when a model is being created, has been created, is being updated, has been updated, and so forth. We can use these to hook into the creation process and set the **UUID** string to the **id** of the model.

We can hook into this using the **boot** method of the model like this:

```

<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Support\Str;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    protected $fillable = [
        'name', 'email', 'password',
    ];

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

    protected $casts = [
        'email_verified_at' => 'datetime',
    ];

    public $incrementing = false;

    protected $keyType = 'string';

    protected static function boot()
    {
        parent::boot();

        static::creating(function ($model) {
            $model->id = (string) Str::uuid();
        });
    }
}

```

Now, our test passes and we have successfully implemented **UUIDs** for our Users.

### *Fixing Laravel Passport*

Before we can use our **UUID** in our application, we also have to tell Laravel Passport about our changes to our primary key for our users. By default, Laravel Passport expects an integer for user **ids**, so we will have to tell Laravel Passport that this has been changed to a UUID.

Fortunately, this is pretty easy, since we only have to change the migrations for Laravel Passport. We can make Laravel publish these migrations for us to change by the following artisan command:

```
php artisan vendor:publish --tag=passport-migrations
```

This will publish the following files into our **database/migrations** folder:

- 2016\_06\_01\_000001\_create\_oauth\_auth\_codes\_table.php
- 2016\_06\_01\_000002\_create\_oauth\_access\_tokens\_table.php
- 2016\_06\_01\_000003\_create\_oauth\_refresh\_tokens\_table.php
- 2016\_06\_01\_000004\_create\_oauth\_clients\_table.php
- 2016\_06\_01\_000005\_create\_oauth\_personal\_access\_clients\_table.php

We need to change the datatype of the migrations that contain a reference to our **user\_id** which is the following files :

- 2016\_06\_01\_000001\_create\_oauth\_auth\_codes\_table.php
- 2016\_06\_01\_000002\_create\_oauth\_access\_tokens\_table.php
- 2016\_06\_01\_000004\_create\_oauth\_clients\_table.php



Here, we should change the datatype from:

```
$table->integer('user_id')->index()->nullable();
```

To:

```
$table->uuid('user_id')->index()->nullable();
```

For all the files. Please note that the **oauth\_auth\_codes** table does not need the call to the **index** and **nullable** methods.

This is it: Laravel Passport now knows that it should work with **UUIDs** instead of integers.

Now, that we are working with our migrations and model, let's implement the ability for different roles right away.

## *Roles*

There are many ways to implement user roles in Laravel, but most often it depends on your scenario. There's a lot to consider here, especially if you are building an application where each user can have multiple roles. In those complex situations, you might be better suited with a dedicated table for roles and then add roles as a relationship to your user. There's even third party packages that can handle roles for you. In our case, we want to keep things simple, also because this is a book about how to build an API — not a book dedicated to authorization and roles, so a column for roles in our users table will suffice. The column could then contain the string **user** when we are dealing with a general user and contain **admin** when we are dealing with an administrator.

Just to have a strategy for how our User models behave when they are created, we want a **User** model to default to the user role. Later we can create a dedicated method that creates a User model with the **admin** role.

Since we already have a tests for our User model, we can just build onto this, but let's break down what we want to test first. In this case, we want to test for the **user** role:

- 1. We set up our world
  - a. We need a user to be able to assert against the role
- 2. We run the code we are testing here
  - a. We don't have anything to run so we skip this step
- 3. We assert against the result that
  - a. The role attribute is on the model
  - b. The role attribute contains a **user** value.

Let's jump back into **tests/Unit/Models/UsersTest.php** file and write this test. We have written ours like this:

```
/**
 * @test
 * @watch
 */
public function it_has_a_role_of_user_by_default()
{
    $user = factory(User::class)->create();
    $this->assertEquals('user', $user->role);
}
```

Right now, this test is failing because we haven't implemented the role into neither our migration or User model so let's do that now. In our **database/migrations/xxxx\_xx\_xx\_XXXXXX\_create\_users\_table.php**, let's add the role column like this:

```

public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->uuid('id')->primary();
        $table->string('name');
        $table->string('email')->unique();
        $table->string('role');
        $table->timestamp('email_verified_at')->nullable();
        $table->string('password');
        $table->rememberToken();
        $table->timestamps();
    });
}

```

Now we know that it's possible to add a default value to columns in the migration, but this does not work for strings, at least not in the version of the framework and at the time of writing this book.

Fortunately, we can use the **attributes** property on our model to mitigate this issue, so let's go to **app/User.php** and add both the **attributes** property, but also make sure that our new attribute is part of the **fillable** array:

```

<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Support\Str;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

```

```

protected $fillable = [
    'name', 'email', 'password', 'role',
];

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

protected $casts = [
    'email_verified_at' => 'datetime',
];

protected $attributes = [
    'role' => 'user',
];

public $incrementing = false;

protected $keyType = 'string';

protected static function boot()
{
    parent::boot();

    static::creating(function ($model) {
        $model->id = (string) Str::uuid();
    });
}
}

```

Now, our test is passing and the **role** attribute a part of our model. Let's move on to implementing the parts specific to the JSON:API specification

## *Your assignment*

It's time to implement our User model as a resource adhering to the JSON:API specification, exactly like we have done with books and authors earlier. And just like we did with books and authors, we should take a test-driven approach.

We won't be going through each and every test like we have done for the other resources in this book. This will be our assignment for you. By now, you should be familiar with the tests needed and if you feel lost, you can look at the tests for the other resources or find the code in our Github repository.

Since there are parts that are a little different with users, we will be going through the tests that are enough for us to have everything configured so that our user resources take advantage of our **JSONAPIService**. The rest of the tests you get to write on your own.

## *Fetching a single user*

Let's get going! The first test we will write is for fetching a single user resource:

```
/**
 * @test
 * @watch
 */
public function it_returns_a_user_as_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $this->getJson("/api/v1/users/{ $user->id }", [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]) ->assertStatus(200)
        ->assertJson([
```

```

        "data" => [
            "id" => $user->id,
            "type" => "users",
            "attributes" => [
                'name' => $user->name,
                'email' => $user->email,
                'created_at' => $user->created_at->toJSON(),
                'updated_at' => $user->updated_at->toJSON(),
            ]
        ]
    });
}

```

Here, we can leverage that we are working with users and reuse the user we use for authentication to also be the user we want to fetch. We call the endpoint and assert against the result. The test is failing at the moment, we need to define the routes and make the controller first, so let's do that. In our **routes/api.php**, let's add the following right above the current user route that points to a closure and let's also edit this route so it corresponds with our conventions for resource naming:

```

// Users
Route::apiResource('users', 'UsersController');

Route::get('/users/current', function (Request $request) {
    return $request->user();
});

```

Then, we jump into the terminal to create the controller like this:

```
php artisan make:controller UsersController -r --api
```

In our newly created **app/Http/Controllers/UsersController.php** file, let's add a constructor that injects the **JSONAPIService** class into our controller before any of the other methods on the class:

```
private $service;

public function __construct(JSONAPIService $service)
{
    $this->service = $service;
}
```

Let's implement the **show** method then, since it is the route we are testing at the moment. Here, we can leverage our service class like this:

```
public function show($user)
{
    return $this->service->fetchResource(User::class, $user, 'users');
}
```

Before we continue, we should add our resource to our config file so that we don't get any errors because of missing configs. Here, we can just copy everything from our books resource and then edit it so it corresponds with our users resource:

```
'users' => [
    'allowedSorts'=> [],
    'allowedIncludes' => [],
    'validationRules'=> [
        'create' => [],
```

```

        'update' => []
    ],
    'relationships' => [
    ]
]

```

There's not a lot here, but you should add to the config as you move along with your tests. If you already know what you want in the config, you can just add it now.

We should also add the methods from our **AbstractAPIModel**, since our User model doesn't extend Laravel's Model class directly:

```

<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Support\Str;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, Notifiable;

    protected $fillable = [
        'name', 'email', 'password', 'role',
    ];

    protected $hidden = [
        'password', 'remember_token', 'email_verified_at'
    ];
}

```



```

protected $casts = [
    'email_verified_at' => 'datetime',
];

protected $attributes = [
    'role' => 'user',
];

public $incrementing = false;

protected $keyType = 'string';

protected static function boot()
{
    parent::boot();

    static::creating(function ($model) {
        $model->id = (string) Str::uuid();
    });
}

public function type()
{
    return 'users';
}

public function allowedAttributes(){
    return collect($this->attributes)->filter(function($item,
        $key){
            return !collect($this->hidden)->contains($key) && $key
                !== 'id';
        }
    )->merge([
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ]);
}
}

```

Our test is now passing and we can move on to the next test.

*Fetching a collection of users*

For this next test, we want to test that we can fetch a collection of users as a collection of resource objects. We have been through this a bunch of times, but the reason why we're including it is to show you that sometimes you will run into tests failing, not because the content itself is incorrect, but because of the order of the items. This often occurs when you are dealing with **UUIDs** since these will be sorted alphabetically when returned by Laravel, but when using factories they are just in the order they were created. Let's first show you the test and go through the code:

```
/**
 * @test
 * @watch
 */
public function
    it_returns_all_users_as_a_collection_of_resource_objects()
{
    $users = factory(User::class, 3)->create();
    $users = $users->sortBy(function ($item) {
        return $item->id;
    })->values();

    Passport::actingAs($users->first());

    $this->getJson("/api/v1/users", [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]) ->assertStatus(200)
        ->assertJson([
            "data" => [
                [
                    "id" => $users[0]->id,
                    "type" => "users",
                    "attributes" => [
                        'name' => $users[0]->name,
```

```

        'email' => $users[0]->email,
        'role' => 'user',
        'created_at' => $users[0]->created_at->
            toJSON(),
        'updated_at' => $users[0]->updated_at->
            toJSON(),
    ]
],
[
    "id" => $users[1]->id,
    "type" => "users",
    "attributes" => [
        'name' => $users[1]->name,
        'email' => $users[1]->email,
        'role' => 'user',
        'created_at' => $users[1]->created_at->
            toJSON(),
        'updated_at' => $users[1]->updated_at->
            toJSON(),
    ]
],
[
    "id" => $users[2]->id,
    "type" => "users",
    "attributes" => [
        'name' => $users[2]->name,
        'email' => $users[2]->email,
        'role' => 'user',
        'created_at' => $users[2]->created_at->
            toJSON(),
        'updated_at' => $users[2]->updated_at->
            toJSON(),
    ]
],
    ]
    );
}

```

In the beginning of the test, we are creating the users, then sorting them so they will match what we get back. The call to the **values** method is so that we

reset the keys or otherwise we won't gain anything from our sorting, since we are referencing the keys in our **assertJson** method.

This test is failing at the moment, but it is because we haven't implemented the **index** method in our controller so let's do that, leveraging our service class:

```
public function index()
{
    return $this->service->fetchResources(User::class, 'users');
}
```

Our test is still failing, but we can see that it is because we are getting the **relationship** member back with an empty array. We will, of course, implement relationships later — especially the relationship to comments. For now, however, we can use this to fix a bug in our **app/Http/Resources/JSONAPIResource.php** file, since we don't need to return the **relationship** member, if there are no relationships. Let's go into the file and make the following change to the **prepareRelationships** method:

```
private function prepareRelationships(){
    $collection = collect(config("jsonapi.resources.{${this->type()}}.relationships"))->flatMap(function($related){
        $relatedType = $related['type'];
        $relationship = $related['method'];
        return [
            $relatedType => [
                'links' => [
                    'self' => route(
                        "{${this->type()}.relationships.{${relatedType}}",
                        ['id' => $this->id]
                    )
                ]
            ]
        ];
    });
}
```

```

        ),
        'related' => route(
            "{$this->type()}.{$relatedType}",
            ['id' => $this->id]
        ),
    ],
    'data' => !$this->whenLoaded($relationship)
        instanceof MissingValue ?
        JSONAPIIdentifierResource::collection($this->{
            $relationship}) : new MissingValue(),
    ],
];
});

return $collection->count() > 0 ? $collection : new MissingValue
();
}

```

Instead of just returning the collection, we save it in a variable and then later down in the end of the method, we use a ternary operator to either return the collection of relationships or a **MissingValue** object, which Laravel can use to exclude the **relationships** member from the response document. Our test is passing now, so that's great.

Let's go back to the sorting part of the test, just to show you the issue so you are prepared. If you comment out the sorting part, which is this specific part of the test:

```

$users = $users->sortBy(function ($item) {
    return $item->id;
})->values();

```

If you are lucky the stars will align and the generated models will have UUIDs that are in correct alphabetical order. Chances are, sadly, that you won't and

this gives us a flaky test that will sometimes pass and sometimes don't. To mitigate this, we do the sorting of our array of generated models.

### *Creating users*

The last test we want to cover here is the test for creating users. This test is a bit more special because, as you might be aware of, when creating and updating users, passwords come into the mix and we should, of course, be hashing these. So in our controllers, we need to hash the password we get in the request and in our test, we also need to ensure that the password is being hashed. We have written the test like this:

```
/**
 * @test
 */
public function it_can_create_an_user_from_a_resource_object()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $response = $this->postJson('/api/v1/users', [
        'data' => [
            'type' => 'users',
            'attributes' => [
                'name' => 'John Doe',
                'email' => 'john@example.com',
                'password' => 'secret',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(201)
        ->assertJson([
            "data" => [
                "type" => "users",
```

```

        "attributes" => [
            'name' => 'John Doe',
            'email' => 'john@example.com',
            'role' => 'user',
            'created_at' => now()->setMilliseconds(0)->
                toJSON(),
            'updated_at' => now() ->setMilliseconds(0)->
                toJSON(),
        ]
    ];

    $this->assertDatabaseHas('users', [
        'name' => 'John Doe',
        'email' => 'john@example.com',
        'role' => 'user',
    ]);

    $this->assertTrue(Hash::check('secret', User::whereName('John
        Doe')->first()->password));
}

```

You should especially pay attention to the last line of this test, since we are testing the password hash here with the **Hash::check** method that Laravel provides.

In our controller, we need to hash the password before we pass it on to our service class like this:

```

public function store(JSONAPIRequest $request)
{
    return $this->service->createResource(User::class, [
        'name' => $request->input('data.attributes.name'),
        'email' => $request->input('data.attributes.email'),
        'password' => Hash::make(($request->input('data.attributes.

```

```

        password' ))),
    ]);
}
```

Unfortunately, we cannot use the input only anymore, but through an array like this, we can use the **Hash::make** method to hash the password correctly

You should use the same approach when updating users.

Before we let you work on your own, we also want to make you aware of a problem you will face when you start to work, especially with the requests that create a resource. Since we are using **UUIDs**, you won't be able to predict what the **IDs** will become. In these cases, you should remove the assertions for the **IDs** both in **assertJson**, **assertDatabaseHas** and **assertHeader**. We are using our service class and we have other tests that make sure these things work, so don't worry.

We will let you work on your own from here, and remember that you can just look in the test files for the other resources if you need to know which test to write next. If you feel lost, you can also always look in our Github repository where all the code is posted.

You don't have to implement relationships since we haven't created the comments resource yet.

## Comments

The next thing we will implement is comments. This resource is actually the one that will tie everything together, especially because it contains the relationship to our users, but also because it contains the last relationship to books. After this resource, we will be entirely done with relationships and can focus on the last bits like authorization.



Through most of this resource, we will let you work on your own again, especially when adhering to the JSON:API specification, like you have just done with the users resource.

We will implement a **one-to-many** relationship to both users and books together, and we will finish the last part of relationship, implementing the possibility to create and update relationships through a resource's endpoints, instead of the relationship endpoints for a resource.

### *Your assignment*

First, the basics of the resource should be implemented. We have done this a bunch of times now, so you should be familiar with this. Therefore, we will let you work through this on your own. Like we have mentioned before, we have multiple resources where you can look if you need a helping hand and, if you are entirely lost, you can look at our code in our Github repository.

Don't worry though, as you might remember from our planning phase, besides the relationships, the comments resource is a fairly easy one, containing only three attributes where the two of them are handled automatically by Laravel, leaving only one for us to worry about. Just to break down the assignment for you, you should now:

- 1. Create a feature test for our comments resource
- 2. Implement the first test, fetching a single comment
- 3. Use this test to drive your implementation
  - a. Making the model
  - b. Making the routes
  - c. Making the controller
  - d. Adding our service class to the controller
  - e. Calling the right method of our service controller
- 4. Implement the rest of the tests, driving out the rest of the implementations

You should implement comments all the way up to the state where our users resource are at now, so we can work together on the relationship between books, comments, and users. When you get to the sorting tests, you only have to sort by **created\_at** and will, of course, not be able to implement sorting by multiple attributes.

### *Users - Comments Relationships*

Now that we have both users and comments on the same level in terms of implementation, let's take a look at their relationships. We already have a relationship implemented in our **JSONAPIService** class, which covers many-to-many relationships. The relationship between users and comments, however, is a one-to-many relationship. This means that the relationship between users and comments and vice versa will actually get us around the full circle of implementing relationships, since we will implement the **to-many** and **to-one** part of relationships through this part. If this all seems daunting to you, don't worry. We will go through it all together.

### *First test*

Like before, we start with a test first, so let's create a **tests/Feature/UsersRelationshipsTest.php** file for us to write our relationships tests like this:

```
<?php

namespace Tests\Feature;

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

class UsersRelationshipsTest extends TestCase
{
    use DatabaseMigrations;
```

```
}
```

Let's write our first test. Here, we can take some inspiration from the tests we have already written, and which tests that the relationship is adhering to the JSON:API specification. If we break down this test, we

- 1. We set up our world
  - a. We need a user to be able to fetch the user's comments
  - b. We need some comments in order to fetch these
  - c. We need the comments to have a relationship to our user
- 2. We run the code we are testing here
  - a. We make a request to the right endpoint
  - b. We add the **include** query parameter with the value of **comments**
- 3. We assert against the result that
  - a. We get the correct user back
  - b. We get a relationship to **comments** back
  - c. We get the correct links
  - d. We get the correct resource identifier objects
  - e. We get the **included** top-level member back
  - f. The **included** array contains the related comments

We have written this test like so:

```
/**
 * @test
 * @watch
 */
public function
    it_returns_a_relationship_to_comments_adhering_to_json_api_spec
    ()
```

```

{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comments = factory(Comment::class, 3)->make();
    $user->comments()->saveMany($comments);

    $this->getJson("/api/v1/users/{$user->id}?include=comments", [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(200)
    ->assertJson([
        "data" => [
            "id" => $user->id,
            "type" => "users",
            "attributes" => [
                'name' => $user->name,
                'email' => $user->email,
                'created_at' => $user->created_at->toJSON(),
                'updated_at' => $user->updated_at->toJSON(),
            ],
            'relationships' => [
                'comments' => [
                    'links' => [
                        'self' => route(
                            'users.relationships.comments',
                            ['id' => $user->id]
                        ),
                        'related' => route(
                            'users.comments',
                            ['id' => $user->id]
                        ),
                    ],
                ],
                'data' => [
                    [
                        'id' => $comments->get(0)->id,
                        'type' => 'comments'
                    ]
                ]
            ]
        ]
    ]);
}

```

## FINISHING UP

```

        [
            'id' => $comments->get(1)->id,
            'type' => 'comments'
        ],
        [
            'id' => $comments->get(2)->id,
            'type' => 'comments'
        ]
    ]
]
],
'included' => [
    [
        'id' => '1',
        'type' => 'comments',
        'attributes' => [
            'message' => $comments->get(0)->message,
            'created_at' => $comments->get(0)->
                created_at->toJson(),
            'updated_at' => $comments->get(0)->
                updated_at->toJson(),
        ]
    ],
    [
        'id' => '2',
        'type' => 'comments',
        'attributes' => [
            'message' => $comments->get(1)->message,
            'created_at' => $comments->get(1)->
                created_at->toJson(),
            'updated_at' => $comments->get(1)->
                updated_at->toJson(),
        ]
    ],
    [
        'id' => '3',
        'type' => 'comments',
        'attributes' => [
            'message' => $comments->get(2)->message,

```

```

        'created_at' => $comments->get(2)->
            created_at->toJson(),
        'updated_at' => $comments->get(2)->
            updated_at->toJson(),
    ],
],
]);
}

```

It does seem like a daunting test at first, but in reality it's just because of the included member, which adds a bit more to the response document, because of all the resource objects needed.

At this moment, our test is failing which tells us that there's no **comments** method on our User model. This is the method that tells Eloquent about the relationship and we already know that it is a **one-to-many** relationship, which means that a comment belongs to one user, but a user can have many comments.

The relationship here would then be that the user has many comments, so let's add that:

```

<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Support\Str;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable

```

```

{
    ...

    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}

```

We add the method to the bottom of the class like this, and our test changes to an error telling us that we do not have the **user\_id** column in our comments table.

Let's fix that, so jump into **database/migrations/xxxx\_xx\_xx\_XXXXX\_create\_comments\_table.php** and add the following to the **up** method:

```

public function up()
{
    Schema::create('comments', function (Blueprint $table) {
        $table->bigIncrements('id');
        $table->text('message');
        $table->string('user_id')->nullable();
        $table->timestamps();
    });
}

```

Here, we add the **user\_id** as a string because the primary key of our users is a **UUID**, which is a string. We add **nullable** to avoid any errors about foreign key constraints, since it would require us to have all relationships set up everytime we want to do anything with a comment. We would have to go through all of our existing tests and add the relationship for the test to work. Also, if we should follow the JSON:API specifications rules for modifying relationships, it will be a constraint on that. If we want comments to be deleted whenever a user or book is deleted, we can add that functionality later on.

Our test is still failing, but giving us a different output: a **400** status code back. So let's call the **withoutExceptionHandling** method in the top of the test to get some more information about what is failing:

We are now told that the given include is not allowed, and that is because it's not set up in our config file, so let's do that:

```
'users' => [
  'allowedSorts'=> [
    'name',
    'email',
  ],
  'allowedIncludes' => [
    'comments',
  ],
  'validationRules'=> [
    'create' => [
      'data.attributes.name' => 'required|string',
      'data.attributes.email' => 'required|email',
      'data.attributes.password' => 'required|string',
    ],
    'update' => [
      'data.attributes.name' => 'sometimes|required|string',
      'data.attributes.email' => 'sometimes|required|email',
      'data.attributes.password' => 'sometimes|required|string',
    ],
  ],
  'relationships' => [
    [
      'type' => 'comments',
      'method' => 'comments',
    ]
  ]
],
```

Here, we add **comments** to the **allowedIncludes** array and, while we're at it,



we might as well add the relationships, so in the **relationships** array, we add the relationship for **comments**.

Now, it's complaining about our routes not being defined, so let's add these to our **routes/api.php** file. We don't have to worry about the controllers for now, we just have to add the routes like this:

```
// Users
Route::apiResource('users', 'UserController');
Route::get('users/{user}/relationships/comments', '
    UsersCommentsRelationshipsController@index')->name('users.
    relationships.comments');
Route::patch('users/{user}/relationships/comments', '
    UsersCommentsRelationshipsController@update')->name('users.
    relationships.comments');
Route::get('users/{user}/comments', '
    UsersCommentsRelatedController@index')->name('users.comments');

Route::get('/users/current', function (Request $request) {
    return $request->user();
});
```

We add the routes right under the apiResource routes, like we did with the other resources, and our test is passing.

### *Users - Comments Relationships*

Now that we have both users and comments on the same level in terms of implementation, let's take a look at their relationships. We already have a relationship implemented in our **JSONAPIService** class, which covers many-to-many relationships. The relationship between users and comments, however, is a one-to-many relationship. This means that the relationship between users and comments and vice versa will actually get us around the full circle of implementing relationships, since we will implement the **to-many**

and **to-one** part of relationships through this part. If this all seems daunting to you, don't worry. We will go through it all together.

### *First test*

Like before, we start with a test first, so let's create a **tests/Feature/UsersRelationshipsTest.php** file for us to write our relationships tests like this:

```
<?php

namespace Tests\Feature;

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

class UsersRelationshipsTest extends TestCase
{
    use DatabaseMigrations;
}
```

Let's write our first test. Here, we can take some inspiration from the tests we have already written, and which tests that the relationship is adhering to the JSON:API specification. If we break down this test, we

- 1. We set up our world
  - a. We need a user to be able to fetch the user's comments
  - b. We need some comments in order to fetch these
  - c. We need the comments to have a relationship to our user
- 2. We run the code we are testing here
  - a. We make a request to the right endpoint
  - b. We add the include query parameter with the value of **comments**
- 3. We assert against the result that
  - a. We get the correct user back

- b. We get a relationship to **comments** back
- c. We get the correct links
- d. We get the correct resource identifier objects
- e. We get the **included** top-level member back
- f. The **included** array contains the related comments

We have written this test like so:

```
/**
 * @test
 * @watch
 */
public function
    it_returns_a_relationship_to_comments_adhering_to_json_api_spec
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comments = factory(Comment::class, 3)->make();
    $user->comments()->saveMany($comments);

    $this->getJson("/api/v1/users/{ $user->id }?include=comments", [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(200)
    ->assertJson([
        "data" => [
            "id" => $user->id,
            "type" => "users",
            "attributes" => [
                'name' => $user->name,
                'email' => $user->email,
                'created_at' => $user->created_at->toJSON(),
                'updated_at' => $user->updated_at->toJSON(),
            ],
        ],
    ]),
```

```

'relationships' => [
  'comments' => [
    'links' => [
      'self' => route(
        'users.relationships.comments',
        ['id' => $user->id]
      ),
      'related' => route(
        'users.comments',
        ['id' => $user->id]
      ),
    ],
    'data' => [
      [
        'id' => $comments->get(0)->id,
        'type' => 'comments'
      ],
      [
        'id' => $comments->get(1)->id,
        'type' => 'comments'
      ],
      [
        'id' => $comments->get(2)->id,
        'type' => 'comments'
      ]
    ]
  ]
],
'included' => [
  [
    'id' => '1',
    'type' => 'comments',
    'attributes' => [
      'message' => $comments->get(0)->message,
      'created_at' => $comments->get(0)->
        created_at->toJson(),
      'updated_at' => $comments->get(0)->
        updated_at->toJson(),
    ]
  ]
]

```

```

    ],
    [
      'id' => '2',
      'type' => 'comments',
      'attributes' => [
        'message' => $comments->get(1)->message,
        'created_at' => $comments->get(1)->
          created_at->toJson(),
        'updated_at' => $comments->get(1)->
          updated_at->toJson(),
      ]
    ],
    [
      'id' => '3',
      'type' => 'comments',
      'attributes' => [
        'message' => $comments->get(2)->message,
        'created_at' => $comments->get(2)->
          created_at->toJson(),
        'updated_at' => $comments->get(2)->
          updated_at->toJson(),
      ]
    ],
  ];
}

```

It does seem like a daunting test at first, but in reality it's just because of the **included** member, which adds a bit more to the response document, because of all the resource objects needed.

At this moment, our test is failing which tells us that there's no **comments** method on our User model. This is the method that tells Eloquent about the relationship and we already know that it is a **one-to-many** relationship, which means that a comment belongs to one user, but a user can have many comments.

The relationship here would then be that the user has many comments, so let's add that:

```
<?php

namespace App;

use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Support\Str;
use Laravel\Passport\HasApiTokens;

class User extends Authenticatable
{
    ...

    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

We add the method to the bottom of the class like this, and our test changes to an error telling us that we do not have the **user\_id** column in our comments table.

Let's fix that, so jump into **database/migrations/xxxx\_xx\_xx\_XXXXXX\_create\_comments\_table.php** and add the following to the **up** method:

```
public function up()
{
    Schema::create('comments', function (Blueprint $table) {
        $table->bigIncrements('id');
```

```

        $table->text('message');
        $table->string('user_id')->nullable();
        $table->timestamps();
    });
}

```

Here, we add the **user\_id** as a string because the primary key of our users is a **UUID**, which is a string. We add **nullable** to avoid any errors about foreign key constraints, since it would require us to have all relationships set up everytime we want to do anything with a comment. We would have to go through all of our existing tests and add the relationship for the test to work. Also, if we should follow the JSON:API specifications rules for modifying relationships, it will be a constraint on that. If we want comments to be deleted whenever a user or book is deleted, we can add that functionality later on.

Our test is still failing, but giving us a different output: a **400** status code back. So let's call the **withoutExceptionHandling** method in the top of the test to get some more information about what is failing:

We are now told that the given include is not allowed, and that is because it's not set up in our config file, so let's do that:

```

'users' => [
    'allowedSorts'=> [
        'name',
        'email',
    ],
    'allowedIncludes' => [
        'comments',
    ],
    'validationRules'=> [
        'create' => [
            'data.attributes.name' => 'required|string',

```

```

        'data.attributes.email' => 'required|email',
        'data.attributes.password' => 'required|string',
    ],
    'update' => [
        'data.attributes.name' => 'sometimes|required|string',
        'data.attributes.email' => 'sometimes|required|email',
        'data.attributes.password' => 'sometimes|required|string',
    ],
],
'relationships' => [
    [
        'type' => 'comments',
        'method' => 'comments',
    ]
]
],

```

Here, we add **comments** to the **allowedIncludes** array and, while we're at it, we might as well add the **relationships**, so in the relationships array, we add the relationship for **comments**.

Now, it's complaining about our routes not being defined, so let's add these to our **routes/api.php** file. We don't have to worry about the controllers for now, we just have to add the routes like this:

```

// Users
Route::apiResource('users', 'UserController');
Route::get('users/{user}/relationships/comments', '
    UsersCommentsRelationshipsController@index')->name('users.
    relationships.comments');
Route::patch('users/{user}/relationships/comments', '
    UsersCommentsRelationshipsController@update')->name('users.
    relationships.comments');
Route::get('users/{user}/comments', '

```



```

    UsersControllerRelatedController@index')->name('users.comments');

Route::get('/users/current', function (Request $request) {
    return $request->user();
});

```

We add the routes right under the **apiResource** routes, like we did with the other resources, and our test is passing.

### *Relationship links - Fetching related comments*

Let's move on to the next test. Again, we follow the tests from our other resource relationship tests and create a test for relationship links, more specifically a test that shows that we can fetch resource identifier objects by sending a get request to our relationship link.

Now, let's just quickly break it down:

- 1. We set up our world
  - a. We need a user to be able to fetch it through our API
  - b. We need a couple of comments to exist to be able to fetch them as comment by our user
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint
- 3. We assert against the result that
  - a. We get to see the comments as proper resource identifier objects

We have written the test like this:

```

/**
 * @test
 * @watch
 */
public function
    a_relationship_link_to_comments_returns_all_related_comments_as_resource_id_
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comments = factory(Comment::class, 3)->make();
    $user->comments()->saveMany($comments);

    $this->getJson("/api/v1/users/{$user->id}/relationships/comments", [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200)
        ->assertJson([
            'data' => [
                [
                    'id' => '1',
                    'type' => 'comments',
                ],
                [
                    'id' => '2',
                    'type' => 'comments',
                ],
                [
                    'id' => '3',
                    'type' => 'comments',
                ],
            ],
        ]);
}

```

This test immediately fails with a **500** status code, and if we call the **withoutExceptionHandling** method at the top of the test, we can see that it is because

we are missing the **UsersCommentsRelationshipsController**, so let's jump into the terminal and make that:

```
php artisan make:controller UsersCommentsRelationshipsController
```

Now, our test is complaining about a missing **index** method, so let's just get in to **app/Http/Controllers/UsersCommentsRelationshipsController.php** right away and add a constructor to inject our **JSONAPIService** class into our controller and then add the **index** method:

```
private $service;

public function __construct(JSONAPIService $service)
{
    $this->service = $service;
}

public function index(User $user)
{
    return $this->service->fetchRelationship($user, 'comments');
}
```

Here, we will return a call to the **fetchRelationship** method on our service class and give it our user. We will get through route-model binding and the relationship we want to fetch which is **comments**.

Our test is now passing and everything is going great so far, because we are leveraging a lot of the code we have written already. In the next test, we will need to do a bit more work though, but let's just write the test first.

*Relationship links - Modifying relationships to comments*

In this test, we will test that we can use our relationships links to modify our relationships and through this add new relationships. In terms of comments, this might not make that much sense. When creating a comment, we want to make the relation between a user and a comment right away, and we won't be modifying which user a comment belongs to after this. Later on, we will implement the ability to add a relationship while creating a resource, but for now let's implement this part, also because we then get to implement another relationship type in our **JSONAPIService** class, but also to adhere to the JSON:API specification according to relationship links. Breaking this test down:

- 1. We set up our world
  - a. We need a user to be able to fetch it through our API
  - b. We need a couple of comments to exist to be able to add a relationship to them
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a PATCH request to the right API endpoint
- 3. We assert against the result that
  - a. We get a **204** status code back
- 4. We assert against the database that
  - a. The comments have a relation to our user

We have written the test like this:

```
/**
 * @test
 * @watch
 */
public function
```

```

        it_can_modify_relationships_to_comments_and_add_new_relationships
        ()
    {
        $this->withoutExceptionHandling();
        $user = factory(User::class)->create();
        Passport::actingAs($user);

        $comments = factory(Comment::class, 10)->make();
        $user->comments()->saveMany($comments);

        $this->patchJson("/api/v1/users/{$user->id}/relationships/
            comments",[
            'data' => [
                [
                    'id' => '5',
                    'type' => 'comments',
                ],
                [
                    'id' => '6',
                    'type' => 'comments',
                ]
            ]
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ]->assertStatus(204);

        $this->assertDatabaseHas('comments', [
            'id' => 5,
            'user_id' => $user->id,
        ]->assertDatabaseHas('comments', [
            'id' => 6,
            'user_id' => $user->id,
        ]));
    }

```

We have added a call to the **withoutExceptionHandling** to this test right away, because we want to know which exceptions are being thrown from the beginning. And, as you might have expected, the test is failing right away,

because we don't have an **update** method in our **app/Http/Controllers/User-  
sCommentsRelationshipsController.php** file. So let's add this now:

```
public function update(JSONAPIRelationshipRequest $request, User
    $user)
{
    return $this->service->updateToManyRelationships($user, '
        comments', $request->input('data.*.id'));
}
```

Here, we add our **JSONAPIRelationshipRequest** class as the request argument to this method and our User model as the second argument to leverage route-model binding. In the method, we make a call to the **updateToManyRelationships** method on our **JSONAPIService** class. This method does not exist yet, but we add it anyway, because it can then serve as a starting point for how we want the signature of the method to look like. Here, we have chosen to keep the arguments in line with the existing **updateManyToManyRelationships** method to be consistent.

Of course, our test is failing, because we are trying to call a method that does not exist, so let's go to our **app/Services/JSONAPIService.php** file now and add the method, and let's add it right above our **updateManyToManyRelationships** method like this:

```
<?php

namespace App\Services;

use App\Author;
use App\Book;
use App\Http\Resources\JSONAPICollection;
```

```

use App\Http\Resources\JSONAPIIdentifierResource;
use App\Http\Resources\JSONAPIResource;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Http\Response;
use Illuminate\Support\Str;
use Spatie\QueryBuilder\QueryBuilder;

class JSONAPIService
{
    ...

    public function fetchRelationship($model, string $relationship)
    {
        return JSONAPIIdentifierResource::collection($model->
            $relationship);
    }

    public function updateToManyRelationships($model, $relationship,
        $ids)
    {
    }

    public function updateManyToManyRelationships($model,
        $relationship, $ids)
    {
        $model->$relationship()->sync($ids);
        return response(null, 204);
    }

    ...
}

```

It was quite easy for us to implement the **updateManyToManyRelationships** method, because the **sync** method on **many-to-many** relationships naturally removes all relationships and builds new ones to those given. But for our **to-many** of our **one-to-many** relationship, it's a bit more complex.

Right out of the box, Laravel doesn't have a method for detaching children from a parent in a **one-to-many** relationship. You can only delete the children, but that is not what we want.

The only way for us to break the relationships from the parents' side of the relationship in Laravel is to find the related children, in our case comments, and set the value of their foreign key to null, hence the call to **nullable** in the migration earlier.

Then, when the relationships have been broken, we can create a relationship to the new children by finding these and setting their foreign key to our parents **ID**, which in this case is the **ID** of our user.

In this method, we get the model of the parent, which is a User model in our case. We get the relationship as a string, which in this case is **comments** and we get the **IDs** of the comments we want to add a relationship to. From this, we need to find all of the comments that have a foreign key that points to our user, but how can we do this when we don't have access to the comments model and the foreign key of each comment?

We could dismiss models and just make raw SQL queries, but where's the fun in that? In this case, you will need to do a little digging in Laravel's source code, because it is not documented. Fortunately for you, we have done the hard work for you, so don't worry about this. It turns out that Laravel actually can help us do this with models, but to find out about this, we have to do a bit of digging in the relationships classes, which our **HasMany** relationship inherits from.

The **HasMany** class inherits from **HasOneOrMany**, which inherits from the **Relation** class. On the **HasOneOrMany** class, we find the **getForeignKeyName** method, which we can call to get the foreign key for the relationship.

On the **Relation** class, we find the **getRelated** method, which can give us an



anonymous Comment model and from this we can create new queries on this model.

This might be a bit abstract, so let's just show you the code and explain what is going on:

```
public function updateToManyRelationships($model, $relationship,
    $ids)
{
    $foreignKey = $model->$relationship()->getForeignKeyName();
    $relatedModel = $model->$relationship()->getRelated();

    $relatedModel->newQuery()->where($foreignKey, $model->id)->
        update([
            $foreignKey => null,
        ]);

    $relatedModel->newQuery()->whereIn('id', $ids)->update([
        $foreignKey => $model->id,
    ]);

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

We use the aforementioned methods to get the foreign key name and the related model. The cool thing about these is that they are dynamic because of the nature of PHP, which we leverage through the **\$model->\$relationship()** call. Here, the PHP will interpret the string and call the right relationship on our model, so when our code is run, it would become this **\$model->comments()**.

Now that we have both of these, we can make a new query for comments and find the comments where the foreign key corresponds to our user id. Then, we can update all of these to have a foreign key with a value of null.

Then, using the same approach, we find the comments that have the given **ids** and update all of these to have a foreign key, pointing to our user's **id** and that's it.

Our test is now passing but before we move on, we just want to mention that sometimes it can pay off to dig a little into the classes of Laravel. There are a lot of useful methods that are not documented, but are perfect for edgecase situations like these.

### *Relationship links - Removing relationships*

For the next test, we want to test if it's possible to remove all relations to comments through the same endpoint. To break this down:

- 1. We set up our world
  - a. We need a user to be able to fetch it through our API
  - b. We need a couple of comments to be associated with the user in order to remove them
  - c. We need to be authenticated
- 2. We run the code that we are testing here
  - a. We make a PATCH request to the right API endpoint
- 3. We assert against the result that
  - a. We get a **204** status code back
- 4. We assert against the database that
  - a. The comments no longer have a relation to our user

We have written the test like this:

```
/**
 * @test
 * @watch
 */
```

```

public function
    it_can_modify_relationships_to_comments_and_remove_relationships
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comments = factory(Comment::class, 5)->make();
    $user->comments()->saveMany($comments);

    $this->patchJson("/api/v1/users/{$user->id}/relationships/
        comments",[
        'data' => [
            [
                'id' => '1',
                'type' => 'comments',
            ],
            [
                'id' => '2',
                'type' => 'comments',
            ],
            [
                'id' => '5',
                'type' => 'comments',
            ],
        ],
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ])->assertStatus(204);

    $this->assertDatabaseHas('comments', [
        'id' => 1,
        'user_id' => $user->id,
    ])->assertDatabaseHas('comments', [
        'id' => 2,
        'user_id' => $user->id,
    ])->assertDatabaseHas('comments', [
        'id' => 5,
        'user_id' => $user->id,
    ]);
}

```

```

    ])->assertDatabaseMissing('comments', [
        'id' => 3,
        'user_id' => $user->id,
    ])->assertDatabaseMissing('comments', [
        'id' => 4,
        'user_id' => $user->id,
    ]);
}

```

This test should pass right away because of our implementation in the **JSONAPIService** class.

Let's quickly follow this up with a test that sends a PATCH request with an empty collection removing all relations:

```

/**
 * @test
 * @watch
 */
public function
    it_can_remove_all_relationships_to_comments_with_an_empty_collection
    ()
    {
        $this->withoutExceptionHandling();
        $user = factory(User::class)->create();
        Passport::actingAs($user);

        $comments = factory(Comment::class, 3)->make();
        $user->comments()->saveMany($comments);

        $this->patchJson("/api/v1/users/{ $user->id }/relationships/
            comments",[
            'data' => []
        ], [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',

```

```

    ]->assertStatus(204);

    $this->assertDatabaseHas('comments', [
        'id' => 1,
        'user_id' => null,
    ]->assertDatabaseHas('comments', [
        'id' => 2,
        'user_id' => null,
    ]->assertDatabaseHas('comments', [
        'id' => 3,
        'user_id' => null,
    ]));
}

```

This test should pass as well.

### *Relationship links - Non existing comments*

Next, we need to secure ourselves a bit more when someone gives a comment that might not exist. Here, we would like the request to fail and send a **400** status code back. So let's write a test for this where:

- 1. We set up our world
  - a. We need a user to be able to fetch it through our API
  - b. We need a couple of comments to be associated with the user in order to remove them
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a PATCH request to the right API endpoint
  - b. We include a comment that does not exist in the request document
- 3. We assert against the result that
  - a. We get a **404** status code back

We have written the following test for this:

```

/**
 * @test
 * @watch
 */
public function
    it_returns_a_404_not_found_when_trying_to_add_relationship_to_a_non_existing
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comments = factory(Comment::class, 3)->make();
    $user->comments()->saveMany($comments);

    $this->patchJson("/api/v1/users/{$user->id}/relationships/
        comments",[
        'data' => [
            [
                'id' => '3',
                'type' => 'comments',
            ],
            [
                'id' => '4',
                'type' => 'comments',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(404)->assertJson([
        'errors' => [
            [
                'title' => 'Not Found Http Exception',
                'details' => 'Resource not found',
            ]
        ]
    ]);
}

```

This test is failing now, because we haven't implemented a check to see

if the given **IDs** exist in our **updateToManyRelationships** method in our **app/Http/Services/JSONAPIService** class, so let's do that now like this:

```
public function updateToManyRelationships($model, $relationship,
    $ids)
{
    $foreignKey = $model->$relationship()->getForeignKeyName();
    $relatedModel = $model->$relationship()->getRelated();

    $relatedModel->newQuery()->findOrFail($ids);

    $relatedModel->newQuery()->where($foreignKey, $model->id)->
        update([
            $foreignKey => null,
        ]);

    $relatedModel->newQuery()->whereIn('id', $ids)->update([
        $foreignKey => $model->id,
    ]);

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

Our test is still failing, but this is not because of our code, but rather that the message sent back isn't the one we were expecting. The one given actually reveals a bit too much about our backend, so let's change that. We do this in our **app/Exceptions/Handler.php** file. Here, we want to add another exception class to our check in the **render** method, so that it is not only on **QueryException** that we return a **NotFoundHttpException** with the "Resource not found" method, but also on **ModelNotFoundException**. We do this like so:

```

public function render($request, Exception $exception)
{
    if($exception instanceof QueryException || $exception instanceof
        ModelNotFoundException){
        $exception = new NotFoundHttpException('Resource not found')
        ;
    }

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

```

And our test passes.

### *Relationships links - Validation*

For the next tests, we will go a bit faster, since these cover validation and the same pattern of validating **id** and **type** that we have done many times now.

If you forgot the why and how, go back to **Relationship Links** in chapter 6.

The following tests are merely testing that our **app/Http/Requests/JSON-APIRelationshipRequest.php** validation rules work as they should. It's still important to test that they work with this relationship, so please add the following tests:

```

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_id_member_is_given_when Updating_a_relationship
    ()
{
    $user = factory(User::class)->create();
}

```



```

Passport::actingAs($user);

$comments = factory(Comment::class, 3)->make();
$user->comments()->saveMany($comments);

$this->patchJson("/api/v1/users/{$user->id}/relationships/
    comments",[
    'data' => [
        [
            'type' => 'comments',
        ],
    ],
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])->assertStatus(422)->assertJson([
    'errors' => [
        [
            'title' => 'Validation Error',
            'details' => 'The data.0.id field is required.',
            'source' => [
                'pointer' => '/data/0/id',
            ],
        ],
    ],
]);
}

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_id_member_is_a_string_when Updating_a_relationship
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comments = factory(Comment::class, 3)->make();

```

```

$user->comments()->saveMany($comments);

$this->patchJson("/api/v1/users/{$user->id}/relationships/
    comments",[
        'data' => [
            [
                'id' => 1,
                'type' => 'comments',
            ],
        ],
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(422)->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The data.0.id must be a string.',
                'source' => [
                    'pointer' => '/data/0/id',
                ],
            ],
        ],
    ],
]);
}

/**
 * @test
 * @watch
 */
public function
    it_validates_that_the_type_member_is_given_when Updating_a_relationship
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comments = factory(Comment::class, 3)->make();
    $user->comments()->saveMany($comments);

```

```

$this->patchJson("/api/v1/users/{$user->id}/relationships/
  comments",[
    'data' => [
      [
        'id' => '1',
      ],
    ]
  ], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
  ]->assertStatus(422)->assertJson([
    'errors' => [
      [
        'title' => 'Validation Error',
        'details' => 'The data.0.type field is required.',
        'source' => [
          'pointer' => '/data/0/type',
        ]
      ]
    ]
  ]
]);
}

/**
 * @test
 * @watch
 */
public function
  it_validates_that_the_type_member_has_a_value_of_authors_when_updating_a_r
  ()
{
  $user = factory(User::class)->create();
  Passport::actingAs($user);

  $comments = factory(Comment::class, 3)->make();
  $user->comments()->saveMany($comments);

  $this->patchJson("/api/v1/users/{$user->id}/relationships/
    comments",[
      'data' => [

```

```

        [
            'id' => '1',
            'type' => 'random',
        ],
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(422)->assertJson([
        'errors' => [
            [
                'title' => 'Validation Error',
                'details' => 'The selected data.0.type is invalid.',
                'source' => [
                    'pointer' => '/data/0/type',
                ]
            ]
        ]
    ]
    ]);
}

```

### *Relationship links - Related*

It's time to implement the last relationship link, which is the **related** link that will fetch all of the resource objects of a relationship. When writing the test for this:

- 1. We set up our world
  - a. We need a user to be able to fetch it through our API
  - b. We need a couple of comments to be associated with the user in order to fetch them
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint
- 3. We assert against the result that
  - a. We get a **200** status code back

- b. We get the correct resource objects for comments back

We have written the test like so:

```
/**
 * @test
 * @watch
 */
public function
    it_can_get_all_related_comments_as_resource_objects_from_related_link
    ()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comments = factory(Comment::class, 3)->make();
    $user->comments()->saveMany($comments);

    $this->getJson("/api/v1/users/{$user->id}/comments",[
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(200)
        ->assertJson([
            'data' => [
                [
                    "id" => '1',
                    "type" => "comments",
                    "attributes" => [
                        'message' => $comments[0]->message,
                        'created_at' => $comments[0]->created_at->
                            toJSON(),
                        'updated_at' => $comments[0]->updated_at->
                            toJSON(),
                    ]
                ],
                [
                    "id" => '2',
                    "type" => "comments",
```

```

        "attributes" => [
            'message' => $comments[1]->message,
            'created_at' => $comments[1]->created_at->
                toJSON(),
            'updated_at' => $comments[1]->updated_at->
                toJSON(),
        ]
    ],
    [
        "id" => '3',
        "type" => "comments",
        "attributes" => [
            'message' => $comments[2]->message,
            'created_at' => $comments[2]->created_at->
                toJSON(),
            'updated_at' => $comments[2]->updated_at->
                toJSON(),
        ]
    ],
]
]);
}

```

This test fails right away because we don't have the **UsersCommentsRelatedController** so let's get into the terminal and make this now:

```
php artisan make:controller UsersCommentsRelatedController
```

Let's jump into the **app/Http/Controllers/UsersCommentsRelatedController.php** file right away and add a constructor so that Laravel can inject our **JSONAPIService** into it. Then, let's add the **index** method like this:

```

<?php

namespace App\Http\Controllers;

use App\Services\JSONAPIService;
use App\User;
use Illuminate\Http\Request;

class UsersCommentsRelatedController extends Controller
{
    /**
     * @var JSONAPIService
     */
    private $service;

    public function __construct(JSONAPIService $service)
    {
        $this->service = $service;
    }

    public function index(User $user)
    {
        return $this->service->fetchRelated($user, 'comments');
    }
}

```

We call the **fetchRelated** method on our **JSONAPIService** class passing in our model and the relationship, and our test is passing.

### *Included*

For the next couple of tests, we want to test that giving and not giving the **include** query parameter in our request with a value of **comments**, will include the comments in our requests, both for a single user and a collection of users. Now, the first test we could write here, we actually wrote as our very first test, so let's start out by testing that we don't get the comments when we are not adding an **include** query parameter. For this test:

- 1. We set up our world
  - a. We need a user to be able to fetch it through our API
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint
  - b. We **don't** add an **include** query parameter
- 3. We assert against the result that
  - a. We get a **200** status code back
  - b. We **don't** get an **included** top-level member
  - c. We **don't** get any comments

We have written the test like this:

```
/**
 * @test
 * @watch
 */
public function
    it_does_not_include_related_resource_objects_when_an_include_query_param_is_
    () {
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comments = factory(Comment::class, 3)->make();
    $user->comments()->saveMany($comments);

    $this->getJson("/api/v1/users/{ $user->id }?include=comments", [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200)
        ->assertJsonMissing([
            'included' => [],
        ]);
}
```



This test should pass right away, since we are leveraging code we have already written.

For the next test, we want to test that we get the related resource objects in the **included** top-level member when making a request for a collection of users.

For this test:

- 1. We set up our world
  - a. We need a bunch of users to be able to fetch them through our API
  - b. We need a bunch of comments to be associated with the first user
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint
  - b. We add an **include** query parameter for **comments**
- 3. We assert against the result that
  - a. We get a **200** status code back
  - b. We get an **included** top-level member
  - c. We get the comments associated with the first user

We have written this fairly long test like so:

```
* @test
* @watch
*/
public function
  it_includes_related_resource_objects_for_a_collection_when_an_include_query
  ()
{
  $users = factory(User::class, 3)->create()->sortBy(function(
    $item){
      return $item->id;
    }->values();
```

```

$comments = factory(Comment::class, 3)->make();
$users->first()->comments()->saveMany($comments);

Passport::actingAs($users->first());

$this->getJson("/api/v1/users?include=comments", [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])
->assertStatus(200)
->assertJson([
    "data" => [
        [
            "id" => $users[0]->id,
            "type" => "users",
            "attributes" => [
                'name' => $users[0]->name,
                'email' => $users[0]->email,
                'role' => 'user',
                'created_at' => $users[0]->created_at->
                    toJSON(),
                'updated_at' => $users[0]->updated_at->
                    toJSON(),
            ],
            'relationships' => [
                'comments' => [
                    'links' => [
                        'self' => route(
                            'users.relationships.comments',
                            ['id' => $users->first()->id]
                        ),
                        'related' => route(
                            'users.comments',
                            ['id' => $users->first()->id]
                        ),
                    ],
                ],
                'data' => [
                    [
                        'id' => $comments->get(0)->id,

```

```

        'type' => 'comments'
    ],
    [
        'id' => $comments->get(1)->id,
        'type' => 'comments'
    ],
    [
        'id' => $comments->get(2)->id,
        'type' => 'comments'
    ]
    ]
    ]
    ],
    [
        "id" => $users[1]->id,
        "type" => "users",
        "attributes" => [
            'name' => $users[1]->name,
            'email' => $users[1]->email,
            'role' => 'user',
            'created_at' => $users[1]->created_at->
                toJSON(),
            'updated_at' => $users[1]->updated_at->
                toJSON(),
        ]
    ],
    [
        "id" => $users[2]->id,
        "type" => "users",
        "attributes" => [
            'name' => $users[2]->name,
            'email' => $users[2]->email,
            'role' => 'user',
            'created_at' => $users[2]->created_at->
                toJSON(),
            'updated_at' => $users[2]->updated_at->
                toJSON(),
        ]
    ],
    ],

```

```

    ],
    'included' => [
        [
            'id' => '1',
            'type' => 'comments',
            'attributes' => [
                'message' => $comments->get(0)->message,
                'created_at' => $comments->get(0)->
                    created_at->toJson(),
                'updated_at' => $comments->get(0)->
                    updated_at->toJson(),
            ]
        ],
        [
            'id' => '2',
            'type' => 'comments',
            'attributes' => [
                'message' => $comments->get(1)->message,
                'created_at' => $comments->get(1)->
                    created_at->toJson(),
                'updated_at' => $comments->get(1)->
                    updated_at->toJson(),
            ]
        ],
        [
            'id' => '3',
            'type' => 'comments',
            'attributes' => [
                'message' => $comments->get(2)->message,
                'created_at' => $comments->get(2)->
                    created_at->toJson(),
                'updated_at' => $comments->get(2)->
                    updated_at->toJson(),
            ]
        ],
    ]
    ]);
}

```

Like before, this test is passing because we are leveraging existing code.

For the next couple of tests, we will be using the same principles. In the next one, we are testing that we don't get an **included** top-level member when we don't add the **include** query parameter, while fetching a collection of users. We have written the test like this:

```
/**
 * @test
 * @watch
 */
public function
    it_does_not_include_related_resource_objects_for_a_collection_when_an_incl
    ()
{
    $users = factory(User::class, 3)->create()->sortBy(function(
        $item){
            return $item->id;
        }->values();

    Passport::actingAs($users->first());

    $this->getJson("/api/v1/users", [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200)
        ->assertJsonMissing([
            'included' => [],
        ]);
}
```

This test should, like the previous tests, pass already. This is it for the users' comments relationships. Now, we need to do the implementation for books and comments next.

## *Books - Comments Relationships*

The implementation of the books and comments relationship follows the exact same steps as the users comments part we have just implemented. In fact, now that we have also implemented the relationship in our **JSONAPIService** class, you just have to use the code we have already written, so you should be able to implement this on your own.

We recommend that you work in the **tests/Feature/BooksRelationships.php** file and actually follow the structure of the file as it is already. So here, you can just copy the existing test for books and authors and edit it to fit books and comments.

You can skip all the tests that validate the **id** and **type** members — no need to test the same class twice.

Another assignment for you is to write a test that includes both authors and comments in the same response when using the **include** query parameter.

If you get lost, you can always look at our code on Github

Good luck!

## *Comments - Users and Comments - Books Relationships*

Now that you have been through an implementation of relationship twice, we don't want to force you through yet another round of relationship tests, so for this section we will give you the tests and focus a bit more on the implementation of the **one-to** part of our **one-to-many** relationship. There will be slight changes in the tests and we will go through them shortly, so you know what is going on. However, for the large part you must be well aware about the why and the how of the tests.

## *Cloning from Github*

If you haven't cloned our Github repository yet, you should do it now by going into your terminal and typing the following command and make sure you are in the right directory before cloning it:

```
git clone git@github.com:WackyStudio/build-an-api-with-laravel.git
```

If you are not using SSH, you can clone through HTTPS like this:

```
git clone https://github.com/WackyStudio/build-an-api-with-laravel.git
```

When you have cloned the repository, go into the **build-an-api-with-laravel/steps/39\_implemented\_comments\_books\_and\_comments\_users\_relationships** folder. Here, you should copy the **tests/Feature/CommentsRelationshipsTest.php** file into the same position in your own project, open the file, and we'll start from there.

If you take a look at the first test: **it\_returns\_a\_relationship\_to\_user\_adhering\_to\_json\_api\_spec**, the thing we really want you to notice is the assertion for a single user resource, instead of a collection under the **users** relationship member. This will be the theme for what we need to implement now, since we don't have any implementations that handle single resources. So for this first test add a **@watch** annotation and let's see what the test tells us.

We are receiving a **400** status code back, which does not give us that much information so let's add a call to **withoutExceptionHandling** at the top of the method. Now, we get a bit more information and can see it's because of the

missing configuration, so let's jump to our **config/jsonapi.php** config file and add the **allowedIncludes** and **relationships** for our comments resource like this:

```
'comments' => [
    'allowedSorts'=> [
        'created_at'
    ],
    'allowedIncludes' => [
        'books',
        'users',
    ],
    'validationRules'=> [
        'create' => [
            'data.attributes.message' => 'required|string',
        ],
        'update' => [
            'data.attributes.message' => 'sometimes|required|string',
        ],
    ],
    'relationships' => [
        [
            'type' => 'books',
            'method' => 'books',
        ],
        [
            'type' => 'users',
            'method' => 'users',
        ],
    ],
]
```

Our test is now failing for a new reason, namely that we haven't defined our relationships on our Comment model, so let's fix that. We know that we are dealing with a **one-to-many** relationship, and we know that we are dealing with the last **one-to** part of it, so we also know what kind of relationship we



should define on our model for both users and books. We have done it like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends AbstractAPIModel
{
    protected $fillable = [
        'message',
    ];

    /**
     * @return string
     */
    public function type()
    {
        return 'comments';
    }

    public function user()
    {
        return $this->belongsTo(User::class);
    }

    public function users(){
        return $this->user();
    }

    public function book()
    {
        return $this->belongsTo(Book::class);
    }
}
```

```

    public function books()
    {
        return $this->book();
    }
}

```

We know that the plural methods pointing to the singular methods seem very strange, but it's actually because of our naming convention for our API that we run into this issue. We could have just had the plural name methods, adding the relationship calls into these methods, but in case we want to do more with our relationships on application level, we didn't want to change the method names and the conventions of Laravel. In this way, we satisfy both worlds, but you are free to do as you like.

Our output of the test has now changed and tells us that it's missing the routes for the relationships, so let's add these in to our **routes/api.php** file right under the **apiResource** method for **comments**:

```

// Comments
Route::apiResource('comments', 'CommentsController');
Route::get('comments/{comment}/relationships/users', '
    CommentsUsersRelationshipsController@index')->name('comments.
    relationships.users');
Route::patch('comments/{comment}/relationships/users', '
    CommentsUsersRelationshipsController@update')->name('comments.
    relationships.users');
Route::get('comments/{comment}/users', '
    CommentsUsersRelatedController@show')->name('comments.users');

Route::get('comments/{comment}/relationships/books', '
    CommentsBooksRelationshipsController@index')->name('comments.
    relationships.books');
Route::patch('comments/{comment}/relationships/books', '
    CommentsBooksRelationshipsController@update')->name('comments.

```

```
relationships.books');
Route::get('comments/{comment}/books', '
CommentsBooksRelatedController@show')->name('comments.books');
```

Now, our test output is telling us that it can't call an undefined method **mapInto** our User model. The reason for this is actually in our **JSONAPIResource** class and this is the first sign of us not having any code to handle cases of single resources instead of collections of resources, since **mapInto** is a method on a Laravel collection, but it's trying to call it on a model.

Let's jump into **app/Http/Resources/JSONAPIResource.php** and take a look at the **prepareRelationships** method. We look at this because, as we can see on the outputted stack of the exception, it is around **line 43** where it originates.

Of course, the line numbers vary in terms of your preferred spaces and so forth:

```
private function prepareRelationships(){
    $collection = collect(config("jsonapi.resources.{$this->type()}.
        relationships"))->flatMap(function($related){
        $relatedType = $related['type'];
        $relationship = $related['method'];
        return [
            $relatedType => [
                'links' => [
                    'self' => route(
                        "{$this->type()}.relationships.{$relatedType}",
                        ['id' => $this->id]
                    ),
                    'related' => route(
                        "{$this->type()}.{$relatedType}",
                        ['id' => $this->id]
                    ),
                ],
            ],
        ],
    );
}
```

```

        'data' => !$this->whenLoaded($relationship)
            instanceof MissingValue ?
            JSONAPIIdentifierResource::collection($this->{
                $relationship}) : new MissingValue(),
    ],
];
});

return $collection->count() > 0 ? $collection : new MissingValue
    ();
}

```

If you take a look at what happens at the **data** member, it's quite a long ternary operator. Up until now it has worked, but it is only handling collections or nothing, and in this case, where we get a single resource, it does not know what to do.

Let's refactor this into a method first. To make sure we are not breaking anything, we should stop our Laravel test watcher and call PHPUnit manually so we get to run through all the tests.

We'll refactor it into a method called **prepareRelationshipData** and place it right under our **prepareRelationships** method like this:

```

private function prepareRelationships(){
    $collection = collect(config("jsonapi.resources.{ $this->type() }.
        relationships"))->flatMap(function($related){
        $relatedType = $related['type'];
        $relationship = $related['method'];
        return [
            $relatedType => [
                'links' => [
                    'self' => route(
                        "{$this->type()}.relationships.{ $relatedType

```

```

        }",
        ['id' => $this->id]
    ),
    'related' => route(
        "{$this->type()}.{$relatedType}",
        ['id' => $this->id]
    ),
],
'data' => $this->prepareRelationshipData(
    $relatedType, $relationship),
],
];
});

return $collection->count() > 0 ? $collection : new MissingValue
();
}

private function prepareRelationshipData($relatedType,
    $relationship){
    if($this->whenLoaded($relationship) instanceof MissingValue){
        return new MissingValue();
    }

    if($this->$relationship() instanceof BelongsTo){
        return new JSONAPIIdentifierResource($this->$relationship);
    }

    return JSONAPIIdentifierResource::collection($this->
        $relationship);
}

```

Running PHPUnit now will give us a lot of failing tests, but if you look closely, you'll see that it's actually the tests we've been given in the **CommentsRelationshipsTest** and no other, so we're ok for now.

Let's then implement a way to return a single resource, whenever we are facing a relationship that will only return single model instances. We know

that Laravel does this for **BelongsTo** relationships, so if we do an introspection for this and then return a single resource in this case, let's see if that will make a difference:

```
private function prepareRelationshipData($relatedType,
    $relationship){
    if($this->whenLoaded($relationship) instanceof MissingValue){
        return new MissingValue();
    }

    if($this->$relationship() instanceof BelongsTo){
        return new JSONAPIIdentifierResource($this->$relationship);
    }

    return JSONAPIIdentifierResource::collection($this->
        $relationship);
}
```

If we run PHPUnit, we still get a lot of failing tests, but fewer failing tests than before: all of them still in the same test class. Let's run Laravel test watcher again to see what happened with our single test class.

It's still failing with that same message, so did we even do any changes? If you look at the stack again, the line where the exception origins have changed to a line further down in our **JSONAPIResource** class. It's actually originating from the **relations** method now. And it makes sense, since it's only handling collections at the moment, so we face the same problem here in this method that we did in **prepareRelationshipData**, where a model does not contain the **mapInto** method. Unfortunately, we cannot use the same implementation, since we do not have access to the relationship string in this method.

We can, however, do some introspection and see if we get a Model instance, since all of our models inherit from this and if we do, then return a single resource like this:

```

private function relations()
{
    return collect(config("jsonapi.resources.{$this->type()}.
        relationships"))
    ->map(function($relation){
        $modelOrCollection = $this->whenLoaded($relation['method']);

        if($modelOrCollection instanceof Model){
            $modelOrCollection = collect([new JSONAPIResource(
                $modelOrCollection)]);
        }

        return JSONAPIResource::collection($modelOrCollection);
    });
}

```

This makes our test green and passing. Before we move on, let's just rearrange the relations, **included** and **with methods** so that they are in this order:

1. **with**
2. **included**
3. **relations**

This follows the flow of how they are called and makes the class a bit more readable.

Let's move on to the next test: **it\_returns\_a\_relationship\_to\_book\_adhering\_to\_json\_api\_spec**, which actually tests the same parts as the previous one, just for books instead of users. If you add a `@watch` annotation to this test method, you should see that it passes.

Moving on to the: **it\_returns\_a\_relationship\_to\_both\_book\_and\_user\_adhering\_to\_json\_api** where we are testing that we can include both the book and the user for a comment. Watching this method, it should pass as well. The next test, which is the **a\_relationship\_link\_to\_user\_returns\_re-**

**lated\_user\_as\_resource\_id\_object**, you should add a **@watch** annotation to and you will see that it fails with a 500 status code. Let's add a call to **withoutExceptionHandler** to the top as well, so we can get some more information about the thrown exception. Here, we can see that it is because we haven't created the necessary **CommentsUsersRelationshipController** that we are referencing from our route. In fact, while we are at it, why don't we create all the needed controllers for the routes we just added, which are:

- CommentsUsersRelationshipsController
- CommentsUsersRelatedController
- CommentsBooksRelationshipsController
- CommentsBooksRelatedController

Go into your terminal and create all of these using the following artisan command:

```
php artisan make:controller CommentsUsersRelationshipsController
```

When you're done, let's jump into **app/Http/Controller/CommentsUsersRelationshipsController.php**. Here, we should add a constructor, injecting our **JSONAPIService** class and then adding the **index** method, like we have done so many times before:

```
<?php

namespace App\Http\Controllers;

use App\Comment;
use App\Services\JSONAPIService;

class CommentsUsersRelationshipsController extends Controller
{
```



```

/**
 * @var JSONAPIService
 */
private $service;

public function __construct(JSONAPIService $service)
{
    $this->service = $service;
}

public function index(Comment $comment)
{
    return $this->service->fetchRelationship($comment, 'users');
}
}

```

We will then receive the same output as earlier about the missing **mapInto** method. This time, it's not in our resource but in our **JSONAPIService** class, so let's go into the **app/Services/JSONAPIService.php** file and make the following change to the **fetchRelationship** method:

```

public function fetchRelationship($model, string $relationship)
{
    if($model->$relationship instanceof Model){
        return new JSONAPIIdentifierResource($model->$relationship);
    }

    return JSONAPIIdentifierResource::collection($model->
        $relationship);
}

```

Just like in our **JSONAPIResource** class, we will check for a single instance of a model and then return a single **JSONAPIIdentifierResource**. This will make our test pass.

Moving on to the next test, we will do the exact same thing in the **app/Http/Controllers/CommentsBooksRelationshipsController.php** file but reference books instead:

```
<?php

namespace App\Http\Controllers;

use App\Comment;
use App\Services\JSONAPIService;

class CommentsBooksRelationshipsController extends Controller
{
    /**
     * @var JSONAPIService
     */
    private $service;

    public function __construct(JSONAPIService $service)
    {
        $this->service = $service;
    }

    public function index(Comment $comment)
    {
        return $this->service->fetchRelationship($comment, 'books');
    }
}
```

Moving on to the **it\_can\_modify\_relationship\_to\_a\_user\_and\_change\_to\_another\_user** test, you should watch the test with Laravel test watcher, add a call to the **withoutExceptionHandler** to the top of the test, and see that it fails because of the missing **update** method in our **app/Http/Controller/CommentsUsersRelationshipsController.php**, so let's go in and add it. Again, we are facing a relationship we haven't yet implemented on our **JSONAPIService** class, so let's write the method call like we want it to be, continuing the

conventions from the other relationships:

```
public function update(JSONAPIRelationshipRequest $request, Comment
    $comment)
{
    return $this->service->updateToOneRelationship($comment, 'users
        ', $request->input('data.id'));
}
```

Now our test is changing the output, but it's not about the missing method like you might think, but rather the validation fails. The reason for this is that our **JSONAPIRelationshipRequest** is expecting a collection of resource identifiers, not just a single resource identifier.

Let's go into our **app/Http/Requests/JSONAPIRelationshipRequest.php** file and change the rules in the **rules** method to the following:

```
public function rules()
{
    return [
        'data' => 'present|array|nullable',

        'data.id' => [Rule::requiredIf($this->has('data.type')), '
            string'],
        'data.type' => [Rule::requiredIf($this->has('data.id')), Rule
            ::in(array_keys(config('jsonapi.resources')))],

        'data.*.id' => [Rule::requiredIf($this->has('data.0')), '
            string'],
        'data.*.type' => [Rule::requiredIf($this->has('data.0')),
            Rule::in(array_keys(config('jsonapi.resources')))],
    ];
}
```

Instead of just testing for a collection of resource identifiers, we test for either a collection or a single resource identifier. We do this by changing all of our rules string into an array, since we want to leverage the **Rule** class provided by Laravel. On this, we would like to call the **requiredIf** method, which we can use to tell if a member is required based on a condition. For the single resource identifier, the condition is that the `data.type` member must be present, before the **data.id** rules kick in and vice versa. For our previous collection rules, these will kick in as soon as a collection is given, as noted by the **data.o** which references the first key of an array.

After these corrections, our test is now failing because of the missing implementation in our **JSONAPIService** class, so let's fix that going into **app/Services/JSONAPIService.php** and add the **updateToOneRelationship** method right above the **updateToManyRelationships** method:

```
<?php

namespace App\Services;

use App\Author;
use App\Book;
use App\Http\Resources\JSONAPICollection;
use App\Http\Resources\JSONAPIIdentifierResource;
use App\Http\Resources\JSONAPIResource;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\QueryException;
use Illuminate\Http\Response;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Str;
use Spatie\QueryBuilder\QueryBuilder;

class JSONAPIService
{
    ...
    public function fetchRelationship($model, string $relationship)
```

```

{
  ...
}
public function updateToOneRelationship($model, $relationship,
    $id)
{

}
public function updateToManyRelationships($model, $relationship,
    $ids)
{
  ...
}

...
}

```

Unlike the **updateToManyRelationships** method, where we had to do most of the implementation ourselves, we can leverage some dedicated methods for associate and dissociate relationships and, in fact, they are named just that. Also, in order to find the new model that should be associated, we can reuse the **getRelated** method we found when implementing the **updateToManyRelationships** method.

We have implemented this method like so:

```

public function updateToOneRelationship($model, $relationship, $id)
{
    $relatedModel = $model->$relationship()->getRelated();

    $model->$relationship()->dissociate();

    if($id){
        $newModel = $relatedModel->newQuery()->findOrFail($id);
        $model->$relationship()->associate($newModel);
    }
}

```

```

    }

    $model->save();
    return response(null, 204);
}

```

We get the related model, and dissociate any existing relations. Then, to think a little ahead, we only want to associate a model if its **id** is given. Otherwise, we just remove the associated model and return. If an **id** is given, we try to find it and then we associate it.

And just like that, our test is passing.

For the next **it\_can\_modify\_relationship\_to\_a\_book\_and\_change\_to\_another\_book**, the concept is the same. Here, we add the following method to the **app/Http/Controllers/CommentsBooksRelationshipsController.php** file:

```

public function update(JSONAPIRelationshipRequest $request, Comment
    $comment)
{
    return $this->service->updateToOneRelationship($comment, 'books
        ', $request->input('data.id'));
}

```

We call the **updateToOneRelationship** method that we have just created and the test passes.

For the next couple of tests:

**It\_can\_modify\_relationship\_to\_a\_user\_and\_remove\_relationship** and **it\_can\_modify\_relationship\_to\_a\_book\_and\_remove\_relationship** we test that we can remove relations by giving a null in the PATCH

request to our relationship link. Remember that this is the equivalent of giving an empty array, which will remove relationships of a collection. Here, a null will remove a relationship to a single resource. If you watch both tests, they should pass with the implementation we have now.

Next, we have the following tests:

**`It_returns_a_404_not_found_when_trying_to_add_relationship_to_a_non-existing_user`** and **`It_returns_a_404_not_found_when_trying_to_add_relationship_to_a_non-existing_book`**. For these tests, we check that we get a **404** response back when trying to add a relationship to a non existing resource. In our implementation of **`updateToOneRelationship`**, we find our new model by using the **`findOrFail`** method, which will ensure that we give the **404** response in case we cannot find a model for the given **`id`**. If you watch these tests, both of them should be passing.

Since we have already made the implementation of a single resource identifier in our **`JSONAPIRelationshipRequest`** class, the following tests will all pass. You can watch them to see for yourself, or otherwise you can skip forward to the **`it_can_get_related_user_as_a_resource_object_from_related_link`** test. Watching this test and adding a call to the **`withoutExceptionHandler`** method, shows us that this test is failing because we are missing an **`index`** method, so let's add that:

```
<?php

namespace App\Http\Controllers;

use App\Comment;
use App\Services\JSONAPIService;

class CommentsUsersRelatedController extends Controller
{
```

```

/**
 * @var JSONAPIService
 */
private $service;

public function __construct(JSONAPIService $service)
{
    $this->service = $service;
}

public function index(Comment $comment)
{
    return $this->service->fetchRelated($comment, 'users');
}
}

```

Just like in any of the other related controllers, we call the **fetchRelated** method on our **JSONAPIService** class, which we are injecting through a constructor in the top of the class.

Our test is still failing though, but it's because of the same issue as before: we are only handling collections. So let's go into **app/Services/JSONAPIService.php** and take a look at the **fetchRelated** method to add the following:

```

public function fetchRelated($model, $relationship)
{
    if($model->$relationship instanceof Model){
        return new JSONAPIResource($model->$relationship);
    }

    return new JSONAPICollection($model->$relationship);
}

```

This will make the method return a single resource also and make our test



pass.

For the next test, the concept is the same and here we will add the following to our **app/Http/Controllers/CommentsBooksRelatedController.php**:

```
<?php

namespace App\Http\Controllers;

use App\Comment;
use App\Services\JSONAPIService;
use Illuminate\Http\Request;

class CommentsBooksRelatedController extends Controller
{
    /**
     * @var JSONAPIService
     */
    private $service;

    public function __construct(JSONAPIService $service)
    {
        $this->service = $service;
    }

    public function index(Comment $comment)
    {
        return $this->service->fetchRelated($comment, 'books');
    }
}
```

This is actually the last test where we have some implementation to do. For the rest of the tests in the test class, we are testing existing code. If you go through and watch all of them, they should all pass.

We have now implemented both our comments users and comments books

relationships. We only need one last thing before we are completely done with implementing relationship for this AP, and that's creating and updating relationships through resource endpoints. Let's implement these now.

### *Modify relationships while creating/updating a comment*

We are almost at the last stop when talking about relationships and the JSON:API specification. The last thing we need to cover is the ability to add or modify relationships when creating resources.

Up until now, the parts we have implemented have been the ability to get and manipulate relationships via the relationship links. And to create a comment and attach it to both a user and book will, as things are implemented now, require a request to create the comment and then a request for associating it with a book and a user.

If you recall the chapter about the JSON:API specification, there is a protocol for adding relationships when creating and updating resources as well. This would make things a bit easier when creating comments, since we can just add the user and book right away.

### *Creating resource*

Let's start by implementing the ability to add a relationship when creating a comment. The good thing is that we have the code for creating resources, but we also have the code for adding relationships. Now, we just have to combine these.

Let's create a test for this in our **tests/Feature/CommentsRelationshipsTest.php** file and place it in the bottom of the file. Before we write the test, let's just break it down:

- 1. We set up our world

- a. We need a user so the comment can be associated with this
- b. We need a book so the comment can be associated with this as well
- c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a POST request to the right API endpoint
  - b. We add the **relationships** member to our request document
  - c. We add the **users** member to our **relationships** object
  - d. We add the **books** member to our **relationships** object
  - e. We add the data member to both relationships
  - f. We give the resource identifiers for both the **user** and **book** we want associated with the comment
- 3. We assert against the result that
  - a. We get a **200** status code back
  - b. We get the correct response document for the comment
  - c. We get the relationships we have just added
- 4. We assert against the database that
  - a. We see that the user **ID** and book **ID** are added to the newly created comment

We have written the test like this:

```
/**
 * @test
 * @watch
 */
public function
  when_creating_a_comment_it_can_also_add_relationships_right_away
  ()
{
  $user = factory(User::class)->create();
  Passport::actingAs($user);

  $book = factory(Book::class)->create();
```

```

$this->postJson('/api/v1/comments', [
    'data' => [
        'type' => 'comments',
        'attributes' => [
            'message' => 'Hello world',
        ],
        'relationships' => [
            'users' => [
                'data' => [
                    'id' => $user->id,
                    'type' => 'users',
                ]
            ],
            'books' => [
                'data' => [
                    'id' => (string)$book->id,
                    'type' => 'books',
                ]
            ]
        ]
    ]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])

->assertStatus(201)
->assertJson([
    "data" => [
        "id" => '1',
        "type" => 'comments',
        "attributes" => [
            'message' => 'Hello world',
            'created_at' => now()->setMilliseconds(0)->
                toJSON(),
            'updated_at' => now() ->setMilliseconds(0)->
                toJSON(),
        ],
        'relationships' => [
            'books' => [
                'links' => [

```

## FINISHING UP

```

        'self' => route(
            'comments.relationships.books',
            ['id' => 1]
        ),
        'related' => route(
            'comments.books',
            ['id' => 1]
        ),
    ],
    'data' => [
        'id' => $book->id,
        'type' => 'books',
    ]
],
'users' => [
    'links' => [
        'self' => route(
            'comments.relationships.users',
            ['id' => 1]
        ),
        'related' => route(
            'comments.users',
            ['id' => 1]
        ),
    ],
    'data' => [
        'id' => $user->id,
        'type' => 'users',
    ]
]
]
]
->assertHeader('Location', url('/api/v1/comments/1'));

$this->assertDatabaseHas('comments', [
    'id' => 1,
    'message' => 'Hello world',
    'user_id' => $user->id,
    'book_id' => $book->id,
]);

```

```
}
```

Notice how easy it is for us to add the relationship by simply adding the resource identifier objects to the request document. Of course, this test is failing. It does create the resource, but no relationships are being added so let's implement this now. For this, we will need to go into **app/Http/Controllers/CommentsController.php** and take a look at the store method:

```
public function store(JSONAPIRequest $request)
{
    return $this->service->createResource(Comment::class, $request->
        input('data.attributes'));
}
```

Right now, we are calling the **createResource** method on our **JSONAPIService** class. If you ask us, it would be nice if we could just continue with this and then pass on the relationships from the controller and let the service class take care of the rest.

So let's write the code here, even though we know it will fail right away and the jump into the service class afterward:

```
public function store(JSONAPIRequest $request)
{
    return $this->service->createResource(Comment::class, $request->
        input('data.attributes'), $request->input('data.
        relationships'));
}
```

Again, we leverage the **input** method on our request to pick out the parts of the request document we want, which in this case are the relationships for users and books. In our **app/Services/JSONAPIService.php** file, let's take a look at the **createResource** method, which at the moment, looks like this:

```
public function createResource(string $modelClass, array
    $attributes)
{
    $model = $modelClass::create($attributes);
    return (new JSONAPIResource($model))
        ->response()
        ->header('Location', route("{ $model->type() }.show", [
            Str::singular($model->type()) => $model,
        ]));
}
```

It's pretty simple: we create the model and then return it through our **JSONAPIResource** class. What we want to do here is to add the relationship, if it is given, right before we return the model. To add the relationships, we would have to first add a new relationship argument, loop over the relationships given in this argument, and then call our existing **updateToOneRelationship** method for each relationship given like this:

```
public function createResource(string $modelClass, array
    $attributes, array $relationships = null)
{
    $model = $modelClass::create($attributes);

    if($relationships){
        foreach ($relationships as $relationshipName => $contents) {
            $this->updateToOneRelationship($model, $relationshipName
                , $contents['data']['id']);
        }
    }
}
```

```

    }

    $model->load(array_keys($relationships));

    return (new JSONAPIResource($model))
        ->response()
        ->header('Location', route("{${$model->type()}.show", [
            Str::singular($model->type()) => $model,
        ]));
    }

```

We add the new **relationships** argument and to now break anything else we have already implemented, we set it with a default value of null. Then, we loop over the relationships and call the **updateToOneRelationship** method.

Because we got the model after creating it earlier, the model is still in the same state, meaning that because we have added our relationships, the model we received does not know about these yet. It needs to load these first, or else we will not return the newly created relationships with the model. So to do this, we leverage eager loading through the **load** method on the model.

This makes our test green and passing.

Before we move on, maybe we should do some refactoring here. Since we know we are just about to make the same implementation for updating comments, we should try to reuse this code, so let's extract the implementation to another method and call this method **handleRelationship** like this:

```

protected function handleRelationship(array $relationships, $model)
    : void
{
    foreach ($relationships as $relationshipName => $contents) {
        $this->updateToOneRelationship($model, $relationshipName,

```



```

        $contents['data'][$id]);
    }

    $model->load(array_keys($relationships));
}

```

And our **createResource** method can now become this:

```

public function createResource(string $modelClass, array
    $attributes, array $relationships = null)
{
    $model = $modelClass::create($attributes);

    if($relationships){
        $this->handleRelationship($relationships, $model);
    }

    return (new JSONAPIResource($model))
        ->response()
        ->header('Location', route("{ $model->type() }.show", [
            Str::singular($model->type()) => $model,
        ]));
}

```

That's a bit better, since the concern of adding relationships are moved away from the **createResource** method, now it just delegates the job to the **handleRelationship** method and continues with its original task.

Our test is still passing, which means that it is a successful refactoring.

## Validating creation

Before we move on, we should do some validation. We don't have anything that catches if a consumer should give a request document that does not follow our conventions, so let's do this first. For our test:

- 1. We set up our world
  - a. We need a user for authentication
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a POST request to the right API endpoint
  - b. We add the **relationships** member to our request document
  - c. We add the **users** member to our **relationships** object without a **data** object
  - d. We add the **books** member to our **relationships** object with wrong data
    - i. We add an id that is not a string
    - ii. We add a type that does not exist
- 3. We assert against the result that
  - a. We get a **422** status code back
  - b. We get the correct error document

We have written this test like so:

```
/**
 * @test
 */
public function
    it_validates_relationships_given_when_creating_comment()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $book = factory(Book::class)->create();
```

```

$this->postJson('/api/v1/comments', [
    'data' => [
        'type' => 'comments',
        'attributes' => [
            'message' => 'Hello world',
        ],
        'relationships' => [
            'users' => [],
            'books' => [
                'data' => [
                    'id' => 1,
                    'type' => 'random',
                ]
            ]
        ]
    ]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])->assertStatus(422)->assertJson([
    'errors' => [
        [
            'title' => 'Validation Error',
            'details' => 'The data.relationships.users.data
                field is required.',
            'source' => [
                'pointer' => '/data/relationships/users/data',
            ]
        ],
        [
            'title' => 'Validation Error',
            'details' => 'The data.relationships.books.data.id
                must be a string.',
            'source' => [
                'pointer' => '/data/relationships/books/data/id',
            ]
        ]
    ],
    [

```

```

        'title' => 'Validation Error',
        'details' => 'The selected data.relationships.books.
            data.type is invalid.',
        'source' => [
            'pointer' => '/data/relationships/books/data/
                type',
        ]
    ],
]
});
}

```

Immediately this test fails, telling us it's getting a **500** status code back. So let's add a call to the **withoutExceptionsHandling** to the top of the test and see which exception is being thrown. Here, it's pretty clear that our validation is not catching the mistakes made. Fortunately for us, we already have the matching validation rules in our **app/Http/Requests/JSONAPIRelationshipsRequest.php** file's **rules** method:

```

public function rules()
{
    return [
        'data' => 'present|array|nullable',

        'data.id' => [Rule::requiredIf($this->has('data.type')), '
            string'],
        'data.type' => [Rule::requiredIf($this->has('data.id')),
            Rule::in(array_keys(config('jsonapi.resources')))],

        'data.*.id' => [Rule::requiredIf($this->has('data.0')), '
            string'],
        'data.*.type' => [Rule::requiredIf($this->has('data.0')),
            Rule::in(array_keys(config('jsonapi.resources')))],
    ];
}

```

```
}
```

We can just copy these to our **app/Http/Requests/JSONAPIRequest.php** and change the dot notation for the rules, since these will be a bit more nested now:

```
public function rules()
{
    $rules = [
        'data' => 'required|array',
        'data.id' => ($this->method() === 'PATCH') ? 'required|
            string' : 'string',
        'data.type' => ['required', Rule::in(array_keys(config('
            jsonapi.resources')))],
        'data.attributes' => 'required|array',

        'data.relationships' => 'array',
        'data.relationships.*.data' => 'required|array',

        'data.relationships.*.data.id' => [Rule::requiredIf($this->
            has('data.relationships.*.data.type')), 'string'],
        'data.relationships.*.data.type' => [Rule::requiredIf($this
            ->has('data.relationships.*.data.id')),
        Rule::in(array_keys(config('jsonapi.resources')))],

        'data.relationships.*.data.*.id' => [Rule::requiredIf($this
            ->has('data.relationships.*.data.0')), 'string'],
        'data.relationships.*.data.*.type' => [Rule::requiredIf(
            $this->has('data.relationships.*.data.0')), Rule::in(
            array_keys(config('jsonapi.resources')))],
    ];

    return $this->mergeConfigRules($rules);
}
```

First, we add the rules that will require the **relationships** member to contain

children if it's given. Then, we add a rule stating that each child must contain a **data** member and afterward, we do as we did in the **JSONAPIRelationshipsRequest** class where we test to see if a single resource is given. If so, the **data.relationships.users.data.id** is required and must be a string. The same goes for **data.relationships.users.data.type** as well.

If a collection has been given, then the same goes for each resource identifier in the collection. If we remove the call to the **withoutExceptionsHandling** method, our test is now passing.

### *Updating resource*

Let's look at how we can implement a way to update our relationships, while also updating our resource. For our test, the concepts are similar to the ones before, but let's just break the test down like we have done earlier:

- 1. We set up our world
  - a. We need a user the comment will be associated with
  - b. We need a book the comment will be associated with
  - c. We need another user the comment should be associated with instead
  - d. We need another book the comment should be associated with instead
  - e. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a PATCH request to the right API endpoint
  - b. We add the **relationships** member to our request document
  - c. We add the resource identifier object for the new **user** to the users relationship
  - d. We add the resource identifier object for the new book to the **books** relationship
- 3. We assert against the result that
  - a. We get a **200** status code back
  - b. We get the correct response document for the comment
  - c. We get the relationships we have just added

- 4. We assert against the database that
  - a. We see that the new user **ID** and new book **ID** are added to the newly created comment instead

We have written the test like this:

```
/**
 * @test
 * @watch
 */
public function
    when_updating_a_comment_it_can_also_update_relationships()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comment = factory(Comment::class)->make();
    $user->comments()->save($comment);

    $book = factory(Book::class)->create();
    $book->comments()->save($comment);

    $anotherUser = factory(User::class)->create();
    $anotherBook = factory(Book::class)->create();

    $this->patchJson('/api/v1/comments/1', [
        'data' => [
            'id' => (string)$comment->id,
            'type' => 'comments',
            'attributes' => [
                'message' => 'Hello world',
            ],
            'relationships' => [
                'users' => [
                    'data' => [
                        'id' => $anotherUser->id,
                        'type' => 'users',
```

```

        ]
    ], [
        'books' => [
            'data' => [
                'id' => (string)$anotherBook->id,
                'type' => 'books',
            ]
        ]
    ]
]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])
->assertStatus(200)
->assertJson([
    "data" => [
        "id" => '1',
        "type" => 'comments',
        "attributes" => [
            'message' => 'Hello world',
            'created_at' => now()->setMilliseconds(0)->
                toJSON(),
            'updated_at' => now() ->setMilliseconds(0)->
                toJSON(),
        ],
        'relationships' => [
            'books' => [
                'links' => [
                    'self' => route(
                        'comments.relationships.books',
                        ['id' => 1]
                    ),
                    'related' => route(
                        'comments.books',
                        ['id' => 1]
                    ),
                ],
            ],
            'data' => [
                'id' => $anotherBook->id,
            ]
        ]
    ]
])

```



```

        'type' => 'books',
    ],
    'users' => [
        'links' => [
            'self' => route(
                'comments.relationships.users',
                ['id' => 1]
            ),
            'related' => route(
                'comments.users',
                ['id' => 1]
            ),
        ],
        'data' => [
            'id' => $anotherUser->id,
            'type' => 'users',
        ]
    ]
]
]);

$this->assertDatabaseHas('comments', [
    'id' => 1,
    'message' => 'Hello world',
    'user_id' => $anotherUser->id,
    'book_id' => $anotherBook->id,
]);
}

```

Just like with our test for creating a resource with relationships added, we start out with a failing test, because it cannot see the updated relationships in the returned JSON. So let's just jump into our **app/Http/CommentsController.php** again and take a look at the **update** method:

```
public function update(JSONAPIRequest $request, Comment $comment)
{
    return $this->service->updateResource($comment, $request->input
        ('data.attributes'));
}
```

Here, we actually want to do the same with the **updateResource** method on our **JSONAPIService** class, just like we did with the **createResource** method, where we just pass in the relationships given in the request like this:

```
public function update(JSONAPIRequest $request, Comment $comment)
{
    return $this->service->updateResource($comment, $request->input
        ('data.attributes'), $request->input('data.relationships'));
}
```

In our **app/Services/JSONAPIService.php** file, we then take a look at the **updateResource** method. Here, we add the **relationships** argument to the method and just like before we set its default value to null, so we don't break anything. Then, we just copy the conditional from the **createResource** method, so that we also delegate the relationship part to our newly created **handleRelationship** method in our **updateResource** method like this:

```
public function updateResource($model, $attributes, $relationships
    = null)
{
    $model->update($attributes);

    if($relationships){
        $this->handleRelationship($relationships, $model);
    }
}
```

```
    return new JSONAPIResource($model);
}
```

Our test is now passing. We hope you noticed how we leveraged the refactoring we just did to reuse our code in the **updateResource** as well, making this implementation much easier.

### *Validating update*

Before we move on, we just want to add a test for testing the validation while updating as well. This test isn't much different than the last test for validating creation of a resource with relationships, but as we have mentioned many times, it's better to have the test than not. For this test:

- 1. We set up our world
  - a. We need a user the comment will be associated with
  - b. We need a book that the comment will be associated with
  - c. We need another user that the comment should be associated with instead
  - d. We need another book that the comment should be associated with instead
  - e. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a PATCH request to the right API endpoint
  - b. We add the **relationships** member to our request document
  - c. We add the **users** member to our **relationships** object without a **data** object
  - d. We add the **books** member to our **relationships** object with wrong data
    - i. We add an id that is not a string
    - ii. We add a type that does not exist
- 3. We assert against the result that
  - a. We get a **422** status code back

- b. We get the correct error document

We have written the test like this:

```
/**
 * @test
 * @watch
 */
public function
    it_validates_relationships_given_when_updating_comment()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $comment = factory(Comment::class)->make();
    $user->comments()->save($comment);

    $book = factory(Book::class)->create();
    $book->comments()->save($comment);

    $this->patchJson('/api/v1/comments/1', [
        'data' => [
            'id' => (string)$comment->id,
            'type' => 'comments',
            'attributes' => [
                'message' => 'Hello world',
            ],
            'relationships' => [
                'users' => [],
                'books' => [
                    'data' => [
                        'id' => 1,
                        'type' => 'random',
                    ]
                ]
            ]
        ], [
    ]
```

```

    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
  })
  ->assertStatus(422)->assertJson([
    'errors' => [
      [
        'title' => 'Validation Error',
        'details' => 'The data.relationships.users.data
          field is required.',
        'source' => [
          'pointer' => '/data/relationships/users/data
            ',
        ]
      ],
      [
        'title' => 'Validation Error',
        'details' => 'The data.relationships.books.data.
          id must be a string.',
        'source' => [
          'pointer' => '/data/relationships/books/data
            /id',
        ]
      ],
      [
        'title' => 'Validation Error',
        'details' => 'The selected data.relationships.
          books.data.type is invalid.',
        'source' => [
          'pointer' => '/data/relationships/books/data
            /type',
        ]
      ],
    ],
  ]
  ));
}

```

The difference here is the existing models that are associated with the existing comment and that we are making a PATCH request instead of a POST request.

This test should pass right away since we have implemented the correct validation.

We are now done implementing the ability to add relationships, while creating a comment resource, but let's do the same for books, making us able to add an author when creating a book.

### *Modify relationships while creating/updating a book*

As we just mentioned, we are going to implement the ability to add or modify an author while creating or updating a book. For this, we can leverage much of the code we have just written, but there will be some important implementations we have to do, since we are dealing with a different relationship between books and authors in contrast to users and comments.

### *Creating resource*

Let's just start from an end and create a test for the ability to add some authors while creating a book. To break the test down:

- 1. We set up our world
  - a. We need some authors the book can be related to
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a POST request to the right API endpoint
  - b. We add the **relationships** member to our request document
  - c. We add the **authors** member inside our **relationship** member
  - d. We add a collection of resource identifier objects for the authors we want to make a relationship to
- 3. We assert against the result that
  - a. We get a **200** status code back
  - b. We get the correct response document for the comment
  - c. We get the relationships we have just added

- 4. We assert against the database that
- a. We see that the book has a relationship to the authors we have given in the request

We have written the test like this:

```
/**
 * @test
 * @watch
 */
public function
    when_creating_a_book_it_can_also_add_relationships_right_away()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $authors = factory(Author::class, 2)->create();

    $this->postJson('/api/v1/books', [
        'data' => [
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ],
            'relationships' => [
                'authors' => [
                    'data' => [
                        [
                            'id' => (string)$authors[0]->id,
                            'type' => 'authors',
                        ],
                        [
                            'id' => (string)$authors[1]->id,
                            'type' => 'authors',
                        ],
                    ],
                ],
            ],
        ],
    ],
```

```

        ]
    ]
]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])
->assertStatus(201)
->assertJson([
    "data" => [
        "id" => '1',
        "type" => 'books',
        "attributes" => [
            'title' => 'Building an API with Laravel',
            'description' => 'A book about API development',
            'publication_year' => '2019',
            'created_at' => now()->setMilliseconds(0)->
                toJSON(),
            'updated_at' => now() ->setMilliseconds(0)->
                toJSON(),
        ],
        'relationships' => [
            'authors' => [
                'links' => [
                    'self' => route(
                        'books.relationships.authors',
                        ['id' => 1]
                    ),
                    'related' => route(
                        'books.authors',
                        ['id' => 1]
                    ),
                ],
            ],
            'data' => [
                [
                    'id' => $authors->get(0)->id,
                    'type' => 'authors'
                ],
                [

```



```

        'id' => $authors->get(1)->id,
        'type' => 'authors'
    ]
    ]
    ]
    ]
    ]->assertHeader('Location', url('/api/v1/books/1'));

$this->assertDatabaseHas('books', [
    'id' => 1,
    'title' => 'Building an API with Laravel',

]);

]);->assertDatabaseHas('author_book', [
    'book_id' => 1,
    'author_id' => $authors[0]->id,
]);

}

```

Like before, our test is failing because it does not get the relationships back in the returned JSON. We now know what this means, so let's just jump into our **app/Http/BooksController.php** and pass the given relationships from the requests into the **createResource** method on our **JSONAPIService** class, injected into our controller:

```

public function store(JSONAPIRequest $request)
{
    return $this->service->createResource(Book::class, $request->
        input('data.attributes'), $request->input('data.
        relationships'));
}

```

Our test is still failing, but this time it's getting a **500** status code back. Let's add a call to the **withoutExceptionHandling** method to the top of the test to get some more information about the exception being thrown.

We can now see that it fails because we are trying to access an **id** key that does not exist in the array. This is because we are getting a collection of resource identifier objects, instead of a single resource identifier object. Our implementation of the **handleRelationship** method in our **app/Services/JSON-APIService.php** only takes a **toOne** relationship into account, and thus it tries to access the **id** key for a single resource object. So let's go to the method and change the implementation so it takes our relationships into account and accesses the keys accordingly. We can do this with a conditional, using introspection on the models relationship, determining if it is a **BelongsTo** or a **BelongToMany**. If it is a **BelongsTo**, we would expect a single resource identifier object, and if it is a **BelongToMany** we would expect a collection. The implementation would then become this:

```
protected function handleRelationship(array $relationships, $model)
    : void
{
    foreach ($relationships as $relationshipName => $contents) {
        if ($model->$relationshipName() instanceof BelongsTo) {
            $this->updateToOneRelationship($model, $relationshipName
                , $contents['data']['id']);
        }
        if($model->$relationshipName() instanceof BelongToMany){
            $this->updateManyToManyRelationships($model,
                $relationshipName, collect($contents['data'])->pluck
                ('id'));
        }
    }

    $model->load(array_keys($relationships));
}
```

A thing to note here is that the first conditional contains the implementation as it was. In the next implementation, we call the **updateManyToMany** method with the model relationship name. To get an array of IDs, we use a Laravel collection and the **pluck** method on this, to pluck out the IDs from all the resource identifier objects, since we are only interested in the **IDs** in the **updateManyToManyRelationships** method.

This implementation will also make our test pass now, since we are calling the right method for handling our relationships.

### *Validating creation*

Like before, we would like to validate the request so we ensure that consumers cannot give wrong information and possibly do damage to our application. Again, we start with a test so let's break it down:

- 1. We set up our world
  - a. We need some authors the book can be related to
  - b. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a POST request to the right API endpoint
  - b. We add the **relationships** member to our request document
  - c. We add the **authors** member to our **relationships** object
  - d. We add the **data** member to our **authors** object with wrong data for each resource identifier object
    - i. We add an id that is not a string
    - ii. We add a type that does not exist
- 3. We assert against the result that
  - a. We get a **422** status code back
  - b. We get the correct error document

We have written the test like this:

```

/**
 * @test
 * @watch
 */
public function it_validates_relationships_given_when_creating_book
()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $authors = factory(Author::class, 2)->create();

    $this->postJson('/api/v1/books', [
        'data' => [
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ],
            'relationships' => [
                'authors' => [
                    'data' => [
                        [
                            'id' => $authors[1]->id,
                            'type' => 'authors',
                        ],
                        [
                            'id' => (string)$authors[1]->id,
                            'type' => 'random',
                        ],
                    ],
                ],
            ],
        ],
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(422)->assertJson([
        'errors' => [

```

```

        [
            'title' => 'Validation Error',
            'details' => 'The data.relationships.authors.data.0.
                id must be a string.',
            'source' => [
                'pointer' => '/data/relationships/authors/data
                    /0/id',
            ]
        ],
        [
            'title' => 'Validation Error',
            'details' => 'The selected data.relationships.
                authors.data.1.type is invalid.',
            'source' => [
                'pointer' => '/data/relationships/authors/data
                    /1/type',
            ]
        ],
    ]
    });
}

```

Because this test is testing the same validation class as we have just implemented while implementing the modification of relationships while creating comments, this test should pass right away.

### *Updating resource*

Let's move on and implement the ability to modify our relationship to authors while updating a book. Let's break down how we want to test this:

- 1. We set up our world
  - a. We need some authors the book can be related to and some we can exchange to
  - b. We need to be authenticated
- 2. We run the code we are testing here

- a. We make a PATCH request to the right API endpoint
- b. We add the **relationships** member to our request document
- c. We add the **authors** member inside our **relationship** member
- d. We add a collection of new resource identifier objects for the authors we want to make a relationship to instead
- 3. We assert against the result that
  - a. We get a **200** status code back
  - b. We get the correct response document for the comment
  - c. We get the relationships we have just added
- 4. We assert against the database that
  - a. We see that the book has a relationship to the new authors we have given in the request

We have written this test like so:

```
/**
 * @test
 * @watch
 */
public function
    when_updating_a_book_it_can_also_update_relationships()
{
    $this->withoutExceptionHandling();
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $book = factory(Book::class)->create();

    $authors = factory(Author::class, 3)->create();
    $book->authors()->sync($authors->pluck('id'));

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'id' => '1',
```

```

        'type' => 'books',
        'attributes' => [
            'title' => 'Building an API with Laravel',
            'description' => 'A book about API development',
            'publication_year' => '2019',
        ],
        'relationships' => [
            'authors' => [
                'data' => [
                    [
                        'id' => (string)$authors[2]->id,
                        'type' => 'authors',
                    ],
                ],
            ],
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
->assertStatus(200)
->assertJson([
    "data" => [
        "id" => '1',
        "type" => 'books',
        "attributes" => [
            'title' => 'Building an API with Laravel',
            'description' => 'A book about API development',
            'publication_year' => '2019',
            'created_at' => now()->setMilliseconds(0)->
                toJSON(),
            'updated_at' => now() ->setMilliseconds(0)->
                toJSON(),
        ],
        'relationships' => [
            'authors' => [
                'links' => [
                    'self' => route(
                        'books.relationships.authors',

```

```

        ['id' => 1]
    ),
    'related' => route(
        'books.authors',
        ['id' => 1]
    ),
],
'data' => [
    [
        'id' => $authors->get(2)->id,
        'type' => 'authors'
    ]
]
]
]
]);

$this->assertDatabaseHas('books', [
    'id' => 1,
    'title' => 'Building an API with Laravel',
])->assertDatabaseHas('author_book', [
    'book_id' => 1,
    'author_id' => $authors[2]->id,
]);
}

```

Right away, this test fails and like before it is because we are not passing our relationships to our **updateResource** method on the **JSONAPIService** in our **update** method in our controller, so let's go into **app/Http/BooksController.php** and fix this:

```

public function update(JSONAPIRequest $request, Book $book)
{

```



```

    return $this->service->updateResource($book, $request->input('
        data.attributes'), $request->input('data.relationships'));
}

```

This is enough — our test is now passing.

### *Validating update*

Once more, we want to test that the validation is working. We are reusing the same concepts here, so instead of going through the details yet again, we just present the test to you so you can implement it. It should be passing right away:

```

/**
 * @test
 */
public function
    it_validates_relationships_given_when Updating_a_book()
{
    $user = factory(User::class)->create();
    Passport::actingAs($user);

    $book = factory(Book::class)->create();

    $authors = factory(Author::class, 3)->create();
    $book->authors()->sync($authors->pluck('id'));

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'id' => '1',
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
            ],
        ],
    ]);
}

```

```

        'publication_year' => '2019',
    ],
    'relationships' => [
        'authors' => [
            'data' => [
                [
                    'id' => $authors[1]->id,
                    'type' => 'authors',
                ],
                [
                    'id' => (string)$authors[1]->id,
                    'type' => 'random',
                ],
            ],
        ]
    ]
], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])
->assertStatus(422)->assertJson([
    'errors' => [
        [
            'title' => 'Validation Error',
            'details' => 'The data.relationships.authors.
                data.0.id must be a string.',
            'source' => [
                'pointer' => '/data/relationships/authors/
                    data/0/id',
            ],
        ],
        [
            'title' => 'Validation Error',
            'details' => 'The selected data.relationships.
                authors.data.1.type is invalid.',
            'source' => [
                'pointer' => '/data/relationships/authors/
                    data/1/type',
            ],
        ],
    ],
])

```

```

        ],
    ],
    1);
}

```

This was the last part for us to implement of relationship for the JSON:API specification, and we now have everything we need. So let's move on to the last parts before ending this chapter and the main part of the book.

## Administrators and authorization

It's time to look at administrators and authorization. Right now, anybody who is authenticated to our API can do whatever they want. If a user wants to delete all books or edit other users' comments, they can do that.

The reason why we created the role attribute on our users model was for us to be able to restrict access to certain actions on our API, based on the role each user has. In other words, you are allowed certain actions, based on the type of tenant you are.

But before we start implementing our authorization, maybe we should make it easier to distinguish which users are regular users and which are administrators.

You might think it would have been easier for us to create a resource just for administrators, so that we have a clear separation for the consumers of our API. In theory, this would be easier, yes, but then you would have to build some logic into your authentication to return an administrator resource when an administrator is being authenticated for your API. You would have to reimplement a lot of relationships, since administrators should be able to comment on books as well.

By keeping everything in our users resource with a role attribute, our consumers can leverage this attribute to do the proper authorization on the frontend and we can reuse the implementation we have already done for users.

Through filters, we can distinguish the queries of our users by roles, and through authorization we can use the role attribute to authorize the right users to be able to call the endpoints they are allowed to.

Let's start by implementing the filtering first.

### *Filtering administrators*

Filters are mentioned in the JSON:API specification, but there are no rules or conventions about how it should work, neither is it a requirement that your API supports this feature. The reason why we implement this is to be able to filter users based on their role, which is a nice feature to have if you are making an administration dashboard and want to list the various users or admins.

For the implementation of our filters, we are actually going to leverage the QueryBuilder by Spatie again. Just like we have done with allowed sorts and allowed includes, the QueryBuilder supports allowed filters as well, which enables it to give an array, telling which attributes we want to filter our users by. Then, we can apply this **filter** through a filter query parameter, just like with **sort** and **include**.

We know we are repeating ourselves here, but let's start out by writing a test first. Breaking this test down:

- 1. We set up our world
  - a. We need some users to exist in order to fetch them
  - b. We need one of our users to be an administrator
  - c. We need to be authenticated
- 2. We run the code we are testing here

- a. We make a GET request to the right API endpoint
- b. We add the filter query parameter
- c. We make the filter query parameter point to our role attribute
- d. We make the value of the filter query parameter be admin
- 3. We assert against the result that
  - a. We get a 200 status code back
  - b. We count that we only get the administrator back
  - c. We get a correct resource object for the administrator back

We have written the test like this:

```
/**
 * @test
 * @watch
 */
public function it_can_filter_administrators_by_role()
{
    $users = factory(User::class, 3)->create();
    $users = $users->sortBy(function ($item) {
        return $item->id;
    })->values();
    $users->first()->role = 'admin';
    $users->first()->save();

    Passport::actingAs($users->first());

    $this->getJson("/api/v1/users?filter[role]=admin", [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]) ->assertStatus(200)
        ->assertJson([
            "data" => [
                [
                    "id" => $users[0]->id,
                    "type" => "users",
                    "attributes" => [
```

```

        'name' => $users[0]->name,
        'email' => $users[0]->email,
        'role' => 'admin',
        'created_at' => $users[0]->created_at->
            toJSON(),
        'updated_at' => $users[0]->updated_at->
            toJSON(),
    ]
},
]
])
->assertJsonMissing([
    "id" => $users[1]->id,
    "attributes" => [
        'name' => $users[1]->name,
        'email' => $users[1]->email,
        'role' => 'user',
        'created_at' => $users[1]->created_at->toJSON(),
        'updated_at' => $users[1]->updated_at->toJSON(),
    ]
])->assertJsonMissing([
    "id" => $users[2]->id,
    "attributes" => [
        'name' => $users[2]->name,
        'email' => $users[2]->email,
        'role' => 'user',
        'created_at' => $users[2]->created_at->toJSON(),
        'updated_at' => $users[2]->updated_at->toJSON(),
    ]
]);
}

```

We add the **filter** query parameter and the way we select which attribute we want to filter, we use square brackets, like this: **?filter[role]** then, like any other query parameter, we set its value, which, in this case, is **admin** so filter away regular users and only get the single admin back. A thing to note here is that we have removed the **type** member in the `assertJsonMissing` methods because this will make the test fail, since our administrator has a **type** member

as well.

At the moment, this test is failing because none of the users are being filtered. Go into **app/Service/JSONAPIService.php** and take a look at our **fetchResources** method:

```
public function fetchResources(string $modelClass, string $type)
{
    $models = QueryBuilder::for($modelClass)
        ->allowedSorts(config("jsonapi.resources.{$type}.
            allowedSorts"))
        ->allowedIncludes(config("jsonapi.resources.{$type}.
            allowedIncludes"))
        ->jsonPaginate();
    return new JSONAPICollection($models);
}
```

At the moment, we are using the **allowedSorts** method, pointing to an array in our **config/jsonapi.php** config file and the same goes for our **allowedIncludes** method. Since we already know that the QueryBuilder from Spatie supports filters through the **allowedFilters** method, which takes an array of the attributes allowed to filter resources on, we can leverage our config file once again. So let's add the following:

```
public function fetchResources(string $modelClass, string $type)
{
    $models = QueryBuilder::for($modelClass)
        ->allowedSorts(config("jsonapi.resources.{$type}.
            allowedSorts"))
        ->allowedIncludes(config("jsonapi.resources.{$type}.
            allowedIncludes"))
        ->allowedFilters(config("jsonapi.resources.{$type}.
            allowedFilters"))
```

```

->jsonPaginate();
return new JSONAPICollection($models);
}

```

Our test is failing because the array doesn't exist in our config file, so let's go in and add it. Now, this method is used in a lot of places in our application, which means we need to add the array to each resource we have in our config file. This doesn't mean that we have to give the attributes that each resource can be filtered on — we only have to give the **role** attribute for our users resource:

```

<?php
return [
    'resources' => [
        'authors' => [
            ...
            'allowedFilters' => [],
            ...
        ],
        'books' => [
            ...
            'allowedFilters' => [],
            ...
        ],
        'users' => [
            ...
            'allowedFilters' => [
                Spatie\QueryBuilder\FILTER::exact('role'),
            ],
            ...
        ],
        'comments' => [
            ...
            'allowedFilters' => [],
            ...
        ]
    ]
]

```



```
];
```

We want our consumers to be specific with the **role** attribute, which is why we are using the QueryBuilder package's **Filter::exact** method to define the filter.

After this, our test passes. Now, to be sure we didn't break anything, we should stop Laravel Test Watcher and run PHPUnit to see if all our tests are passing, which they should be.

Then, we can move on and do a test for the inverse, filtering our results for regular users instead of admin. This test has the same buildup except that the **filter** query parameter has a value of **user** this time, and the contents of the **assertJson** and **assertJsonMissing** has been swapped:

```
/**
 * @test
 * @watch
 */
public function it_can_filter_users_by_role()
{
    $users = factory(User::class, 3)->create();
    $users = $users->sortBy(function ($item) {
        return $item->id;
    })->values();
    $users->first()->role = 'admin';
    $users->first()->save();

    Passport::actingAs($users->first());

    $this->getJson("/api/v1/users?filter[role]=user", [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]) ->assertStatus(200)
        ->assertJson([
```

```

        "data" => [
            [
                "id" => $users[1]->id,
                "type" => "users",
                "attributes" => [
                    'name' => $users[1]->name,
                    'email' => $users[1]->email,
                    'role' => 'user',
                    'created_at' => $users[1]->created_at->
                        toJSON(),
                    'updated_at' => $users[1]->updated_at->
                        toJSON(),
                ]
            ],
            [
                "id" => $users[2]->id,
                "type" => "users",
                "attributes" => [
                    'name' => $users[2]->name,
                    'email' => $users[2]->email,
                    'role' => 'user',
                    'created_at' => $users[2]->created_at->
                        toJSON(),
                    'updated_at' => $users[2]->updated_at->
                        toJSON(),
                ]
            ]
        ]
    })
    ->assertJsonMissing([
        "id" => $users[0]->id,
        "attributes" => [
            'name' => $users[0]->name,
            'email' => $users[0]->email,
            'role' => 'admin',
            'created_at' => $users[0]->created_at->toJSON(),
            'updated_at' => $users[0]->updated_at->toJSON(),
        ]
    ]);
}

```

If you go ahead and start Laravel Test Watcher again, this test should be green and passing.

Now, we just have to do the last test for the scenario where a consumer is trying to filter users on an attribute that does not exist or is not allowed. For this test

- 1. We set up our world
  - a. We need some users to exist in order to fetch them
  - b. We need one of our users to be an administrator
  - c. We need to be authenticated
- 2. We run the code we are testing here
  - a. We make a GET request to the right API endpoint
  - b. We add the **filter** query parameter
  - c. We make the **filter** query parameter point to a **foo** attribute
  - d. We make the value of the **filter** query parameter be **bar**
- 3. We assert against the result that
  - a. We get a **400** status code back
  - b. We get the right error document back

Our test looks like this:

```
/**
 * @test
 * @watch
 */
public function
    it_cannot_fetch_a_resource_with_a_role_that_does_not_exist()
{
    $users = factory(User::class, 3)->create();
    $users = $users->sortBy(function ($item) {
        return $item->id;
    })->values();
```

```

$users->first()->role = 'admin';
$users->first()->save();

Passport::actingAs($users->first());

$this->getJson("/api/v1/users?filter[foo]=bar", [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
]) ->assertStatus(400)->assertJson([
    'errors' => [
        [
            'title' => 'Invalid Filter Query',
            'details' => 'Given filter(s) `foo` are not allowed.
                          Allowed filter(s) are `role`.'
        ]
    ]
]);
}

```

Since this part is already implemented in the `QueryBuilder` from Spatie, the test should pass. If they should change this in the future, we now have a test that can warn us if that happens, in which we would have to implement this exception ourselves. Of course, we hope that doesn't happen but like we've said before, it's better to have a test for it.

This is actually all there is for filtering. If you want, you can go ahead and implement filters for the other resources. We will be moving on to authorization now.

## Authorization

It's time to look at authorization and, like we have mentioned earlier, starting to specify which areas of our API and application can be accessed by which roles.

Before we do this, why don't we take a look at what each role should be able

to do. For this application, we have identified the following actions per role:

**Users** should have authorization for:

1. Fetching/reading books
2. Fetching/reading authors
3. Creating comments on books
4. Updating their own comment on a book
5. Deleting their own comment on a book
6. Updating their own user
7. Deleting their own user

**Administrators** should have authorization for:

1. Creating, updating, fetching/reading and deleting Users
2. Creating, updating, fetching/reading and deleting Books
3. Creating, updating, fetching/reading and deleting Authors
4. Creating comments on books
5. Updating their own comment on a book
6. Deleting their own comment on a book
7. Deleting regular users' comments on a book

Now, you might be wondering why a user does not have the ability to create a user. This is because we use user registration for this, which is not something we access in our API after we have been authenticated. The user registration will then happen on a set of routes that are outside of our authentication, so that anyone can access those and register for an account. This is also often called a guest user.

To implement authorization in an API with Laravel, we can actually leverage Laravel's Policies and Gates, like you would in any other Laravel application.

This is not specific to API development in Laravel, but the nice thing is that Laravel, through these features, makes it super easy to implement authorization into an API.

We won't be covering all resources and endpoints, but instead we'll give you the basic knowledge through the books resource. With this, you are then able to write tests and implement authorization for the rest of the resources.

### *Authorization for books*

For authorization of our books resource, we will be using Laravel's policies, a feature of Laravel that makes it possible to center the logic of the authorization for a specific model into a dedicated class. It is then possible for us to use a policy on a controller, for instance the **BookController** to have authorization on all the methods in this controller, without too much of a hassle. A thing to note here is that it is only possible to have one policy class per model, so we will not be able to use a policy in our relationship controllers without us having to do some extra work. In these classes, we will instead use Laravel's Gate feature, which makes it possible for us to do single authorization rules that we can apply inside each of our controller methods.

But before we get too ahead of ourselves here, let's learn about all of this through implementation. As always, let's start with a test first.

For testing authorization, we always create a dedicated test class, so that we separate the concerns of the tests a bit. For the tests we are about to write, let's create a **tests/Feature/BooksAuthorizationTest.php** file and make sure that we extend Laravel's **TestCase** class and use the **DatabaseMigrations** trait as well, like this:

```
<?php

namespace Tests\Feature;

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

class BooksAuthorizationTest extends TestCase
{
    use DatabaseMigrations;
}
```

Then, we are ready to write our first test. When dealing with authorization, we like to write our tests in the following form:

- A user **cannot**
- An admin **can**

Of course, there will be situations where both roles can access a resource or action and situations where it's reversed, but we generally write our authorization tests in this form. We also start from a point where we know this form can be used right away, like a test for creating a book. These test methods will then be named as such:

- A user **cannot** create a book
- An admin **can** create a book

The benefits of this is that we then protect the resource or action from the user, but also makes sure the admin can still access it.

While we are discussing the creation of a book, why don't we write this test first. Create a new test with the **a\_user\_cannot\_create\_a\_book** and go

into the **tests/Feature/BooksTest.php** and copy all of the contents from the **it\_can\_create\_an\_book\_from\_a\_resource\_object** and paste it into your newly created test method in the **tests/Feature/BooksAuthorizationTest.php** file.

Then, change the contents of the **assertJson** method to the following, and change the status code from **201** to **403**. We like to be explicit about the roles of our users, so for our users model we want to show that the role is user:

```
/**
 * @test
 * @watch
 */
public function a_user_cannot_create_a_book()
{
    $user = factory(User::class)->create([
        'role' => 'user',
    ]);
    Passport::actingAs($user);

    $this->postJson('/api/v1/books', [
        'data' => [
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(403)->assertJson([
        'errors' => [
            [
                'title' => 'Access Denied Http Exception',
                'details' => 'This action is unauthorized.',
            ]
        ]
    ])
```



```

        ]
    ]
    });
}

```

This test is failing at the moment, but before we set out on our course to implement this, let's create the test for the admin user as well.

Here, you can just copy the test method we have just made, change the name to **an\_admin\_can\_create\_a\_book**, remove the **assertJson**, and set the status code back to **201**. Remember to set the role in the factory to be admin instead of user:

```

/**
 * @test
 */
public function an_admin_can_create_a_book()
{
    $user = factory(User::class)->create([
        'role' => 'admin',
    ]);
    Passport::actingAs($user);

    $this->postJson('/api/v1/books', [
        'data' => [
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
}

```

```
->assertStatus(201);  
  
}
```

Great! We are now ready to start our implementation. As we mentioned earlier, in the cases where we work with the Book model itself, we will use a policy, so let's go out into the terminal and create a policy for our Book model like this:

```
php artisan make:policy BookPolicy --model=Book
```

The cool thing about policies is that if we follow the naming convention **ModelNamePolicy**, which, in our case, will be **BookPolicy**, and place our policies inside the **app/Policies** folder, Laravel can automatically discover these. Don't worry, when generating a policy through an artisan command, it is automatically placed inside the **app/Policies** folder. We just have to ensure that we follow the naming convention, which we have done.

Open up the newly created **app/Policies/BookPolicy.php** file. Here, you will see that Laravel has already done a lot of work for us. It has created a bunch of methods where most of them correspond to the naming in our controller. Now, go ahead and delete all methods except for:

- View
- Create
- Update
- Delete

We won't be needing the last two methods.

Then, let's go into our **app/Controller/BooksController.php** and inside the constructor, we add the following to use our newly created policy in this controller:

```
public function __construct(JSONAPIService $service)
{
    $this->service = $service;
    $this->authorizeResource(Book::class, 'book');
}
```

Since we are using a resource controller, Laravel has a helper method we can use in our constructor. This helper method is actually automatically adding a middleware in front of all of our methods in our controllers, making sure that the request goes through one of our methods in our policy class before hitting our method in our controller. This also means that the methods in our controller maps to the methods in our policy class.

The mapping is the following:

- The controller's **index** and **show** methods utilize the **view** method in the policy class
- The controller's **store** method utilizes the **create** method in the policy class
- The controller's **update** method utilizes the **update** method in the policy class
- The controller's **destroy** method utilizes the **delete** method in the policy class

To make sure that only a user with the role of admin can create a book, all we have to do is to add the following to the **create** method in our **app/Policies/-BookPolicy.php** file:

```
public function create(User $user)
{
    return $user->role === 'admin';
}
```

This change is enough for both of our tests to fail. The first test will get a **403** response back where as the admin will get a **201**, since it is allowed to create a book.

Let's move on to updating a book. The concept here is exactly the same, so we will just show you our test and explain:

```
/**
 * @test
 * @watch
 */
public function a_user_cannot_update_a_book()
{
    $user = factory(User::class)->create([
        'role' => 'user',
    ]);
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'id' => '1',
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(403)->assertJson([
        'errors' => [
            [
                'title' => 'Access Denied Http Exception',
                'details' => 'This action is unauthorized.',
            ]
        ]
    ])
```

```

        ]
    ]
});
}

/**
 * @test
 * @watch
 */
public function an_admin_can_update_a_book()
{
    $user = factory(User::class)->create([
        'role' => 'admin',
    ]);
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->patchJson('/api/v1/books/1', [
        'data' => [
            'id' => '1',
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(200);
}

```

Here, we have both the tests for the user and the admin users. We have just copied the contents from the **it\_can\_update\_an\_book\_from\_a\_resource\_object** test in the **test/Feature/BooksTest.php** file. Then, we want to be explicit about the role, so we add this to the factory. Then, we change the contents in the **assertJson** method with the exact same contents from the **a\_user\_cannot\_create\_a\_book** test, since it's the same response we

will get for all the authorization exceptions. We, of course, also changed the status code to **403**. In the **an\_admin\_can\_update\_a\_book**, we removed the **assertJson** method and kept only the status code. We are already testing what we get back — now, we are just interested in the admin being able to update a book.

To fix our failing test, let's go into the **app/Policies/BookPolicy.php** file and take a look at the **update** method. Here, we should essentially do the same as in the **create** method, since we only want the admin to access this:

```
public function update(User $user, Book $book)
{
    return $user->role === 'admin';
}
```

Our test is passing now, so let's move on to the next couple of tests.

Next, we will look at the deletion of books. Again, we follow the same concepts of copying from our other test, setting the role and changing the **assertJson** contents and status code:

```
/**
 * @test
 * @watch
 */
public function a_user_cannot_delete_a_book()
{
    $user = factory(User::class)->create([
        'role' => 'user',
    ]);
    Passport::actingAs($user);
    $book = factory(Book::class)->create();
```

```

$this->delete('/api/v1/books/1',[], [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])->assertStatus(403)->assertJson([
    'errors' => [
        [
            'title' => 'Access Denied Http Exception',
            'details' => 'This action is unauthorized.',
        ]
    ]
]);
}

/**
 * @test
 * @watch
 */
public function an_admin_can_delete_a_book()
{
    $user = factory(User::class)->create([
        'role' => 'admin',
    ]);
    Passport::actingAs($user);
    $book = factory(Book::class)->create();

    $this->delete('/api/v1/books/1',[], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(204);
}

```

In the **app/Policies/BookPolicy.php**, we do the same again, since we only want admins to be able to delete books:

```

public function delete(User $user, Book $book)
{

```

```

    return $user->role === 'admin';
}

```

And both our tests are passing.

The next couple of tests are a bit different. We do want both our regular users and administrators to be able to fetch a single book and a collection of books, so here we will change the test for the user a bit. We still copy from our other test, since all the code is already written there, but we remove the assertions, since they don't matter here:

```

/**
 * @test
 * @watch
 */
public function a_user_can_fetch_a_list_of_books()
{
    $user = factory(User::class)->create([
        'role' => 'user',
    ]);
    Passport::actingAs($user);
    $books = factory(Book::class, 3)->create();

    $this->get('/api/v1/books', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(200);
}

/**
 * @test
 * @watch
 */
public function an_admin_can_fetch_a_list_of_books()
{

```



```

$user = factory(User::class)->create([
    'role' => 'admin',
]);
Passport::actingAs($user);
$books = factory(Book::class, 3)->create();

$this->get('/api/v1/books', [
    'accept' => 'application/vnd.api+json',
    'content-type' => 'application/vnd.api+json',
])->assertStatus(200);
}

```

Both of our tests are passing already, so let's move to fetching a single book, since we will face some problems here. The tests for these are:

```

/**
 * @test
 * @watch
 */
public function a_user_can_fetch_a_single_book()
{
    $this->withoutExceptionHandling();
    $user = factory(User::class)->create([
        'role' => 'user'
    ]);
    Passport::actingAs($user);

    $book = factory(Book::class)->create();

    $this->getJson('/api/v1/books/1', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200);
}

```

```

/**
 * @test
 * @watch
 */
public function an_admin_can_fetch_a_single_book()
{
    $this->withoutExceptionHandling();
    $user = factory(User::class)->create([
        'role' => 'admin'
    ]);
    Passport::actingAs($user);

    $book = factory(Book::class)->create();

    $this->getJson('/api/v1/books/1', [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
        ->assertStatus(200);
}

```

Right now, these tests are failing and it is because of the **view** method in our policy class. At the moment, it utilizes route-model binding, but our controller does not. Here, we only get an **ID** for the book and then we let the QueryBuilder in the **JSONAPIService** class do the query for us. This mismatch will cause the policy to block this route, but fortunately we can do something about it.

To explain what we will do, let's take a look at the **create** method in our **app/Policies/BookPolicy.php** file again. This method does not get any models passed into it, since it's a route that doesn't leverage route-model binding. Laravel knows this and will therefore only pass in the authenticated user. We want the same to happen with our **view** method and it turns out it can be done pretty easily.

If we go into our **app/Http/Controllers/BooksController.php** and take a look at the constructor again, we can see that we are making a call to the **authoriz-**

**eResource** method on our controller class. This method actually stems from a trait that is being used on the parent class of our controller. Let's open this trait for a second, which is placed at the following path: **vendor/laravel/framework/src/illuminate/Foundation/Auth/Access/AuthorizesRequests.php**.

In the bottom of the trait, you should see the following property:

```
protected function resourceMethodsWithoutModels()
{
    return ['index', 'create', 'store'];
}
```

This property is actually responsible for telling which methods in our controller that utilizes route-model binding.

The thing with traits is that anything they provide can be overridden, so in this case we can just copy this property to our controller and change the array to contain our **show** method as well, like this:

```
...
class BooksController extends Controller
{
    protected function resourceMethodsWithoutModels()
    {
        return ['index', 'store', 'show'];
    }

    /**
     * @var JSONAPIService
     */
    private $service;

    public function __construct(JSONAPIService $service)
    {
```

```

        $this->service = $service;
        $this->authorizeResource(Book::class, 'book');
    }
    ...
}

```

We remove the **create** method since our controller doesn't contain it, and then add the **store** method instead. Let's go back to our **view** method in our **app/Policies/BookPolicy.php** and return true, since anyone can view our books. After this, both our tests will pass.

Again, this implementation is an example of the benefits of looking a bit deeper into Laravel's codebase.

This is actually it for the authorization rules for books. Next, we will look at the relationships between books and authors. Unfortunately, we cannot use policies here, but we can instead use Laravel's gate feature, which is just as convenient.

### *Authorization for books authors relationships*

For the authorization rules for books and authors, we cannot use policies as we just mentioned. Here, we will use Laravel's Gate features. This means that there's a bit more implementation for us to do, but on the positive side, we can skip some tests. Because policies work on a controller level, we have to test all scenarios, but gates will be used on a method level, which means that if a gate is not added to a method, all users can hit it. For our relationships, we want both users and admins to be able to fetch relationships and related resource, but we only want the admin to be able to modify the relationship. This means that we only need a test for this.

For this test, we can copy the contents of the **it\_can\_modify\_relationships\_to\_authors\_and\_add\_new\_relationships** test inside our

tests/Feature/BooksRelationshipsTest.php file. We will copy this into two new test methods on our **tests/Feature/BooksAuthorization.php** file named: **a\_user\_cannot\_modify\_relationship\_links\_for\_authors** and **an\_admin\_can\_modify\_relationship\_links\_for\_authors**:

```
/**
 * @test
 * @watch
 */
public function a_user_cannot_modify_relationship_links_for_authors
()
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 10)->create();

    $user = factory(User::class)->create([
        'role' => 'user'
    ]);
    Passport::actingAs($user);

    $this->patchJson('/api/v1/books/1/relationships/authors', [
        'data' => [
            [
                'id' => '5',
                'type' => 'authors',
            ],
            [
                'id' => '6',
                'type' => 'authors',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])->assertStatus(403)->assertJson([
        'errors' => [
```

```

        [
            'title' => 'Access Denied Http Exception',
            'details' => 'This action is unauthorized.',
        ]
    ]
});
}

/**
 * @test
 * @watch
 */
public function an_admin_can_modify_relationship_links_for_authors
(
)
{
    $book = factory(Book::class)->create();
    $authors = factory(Author::class, 10)->create();

    $user = factory(User::class)->create([
        'role' => 'admin'
    ]);
    Passport::actingAs($user);

    $this->patchJson('/api/v1/books/1/relationships/authors', [
        'data' => [
            [
                'id' => '5',
                'type' => 'authors',
            ],
            [
                'id' => '6',
                'type' => 'authors',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ]->assertStatus(204);
}

```

Like before, we are explicit with the role in the factory method, and we change the status code of the test that tests from the user's point of view. Here, we also add the **assertJson** contents for an authorization exception, just like in the other tests. In the test for the admin's point of view, we remove the **assertJson** and keep the rest.

For now, only one of the tests passes, so let's define our gate. We define gates in our **app/Providers/AuthServiceProvider.php** and in the **boot** method like this:

```
public function boot()
{
    $this->registerPolicies();

    Passport::routes();

    //Passport::enableImplicitGrant();

    Gate::define('admin-only', function($user){
        return $user->role === 'admin';
    });
}
```

We define a gate with the name **admin-only**, which we can use in all the controller methods where we only want the admin to have access. We give it a closure it will use for whenever the **admin-only** gate needs evaluation, and here we convey that the user should have a role of admin to pass this gate.

Let's then jump over to our **app/Http/Controllers/BooksAuthorsRelationshipsController.php** and take a look at the **update** method. Here, we want to add the gate and tell that if the gate denies access, then we will throw an **AuthorizationException** with the same message as we are getting from our policy classes to keep consistency. We do that like this:

```

public function update(JSONAPIRelationshipRequest $request, Book
    $book)
{
    if(Gate::denies('admin-only')){
        throw new AuthorizationException('This action is
            unauthorized.');
```

This is enough for both our tests to pass.

Laravel makes it really easy for us to add authorization to our applications and, as we mentioned earlier, this method is not only for APIs. You can use this in all Laravel applications.

If you want to know more about Laravel's policies and gates, we encourage you to look at the documentation. This is all we will cover in this book, but it should be enough for you to do authorization for the rest of the resources and relationships.

Before we move on though, we should run PHPUnit to check if anything has broken after our changes. Here, you will see that quite a lot of tests in both our **tests/Feature/BooksTest.php** and **tests/Feature/BooksRelationshipsTest.php** are failing. These are, of course, also affected by the authorization, but the fix is quite easy: you just have to add the **admin** role to all the users you authenticate for each of the tests.

The easiest way is to add a state to the **database/factories/UserFactory** like this:



```
$factory->state(App\User::class, 'admin', [
    'role' => 'admin',
]);
```

Then in your test, you can add the state to the factory like this:

```
/**
 * @test
 */
public function it_can_create_an_book_from_a_resource_object()
{
    $user = factory(User::class)->state('admin')->create();
    Passport::actingAs($user);

    $this->postJson('/api/v1/books', [
        'data' => [
            'type' => 'books',
            'attributes' => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
            ]
        ]
    ], [
        'accept' => 'application/vnd.api+json',
        'content-type' => 'application/vnd.api+json',
    ])
    ->assertStatus(201)
    ->assertJson([
        "data" => [
            "id" => '1',
            "type" => "books",
            "attributes" => [
                'title' => 'Building an API with Laravel',
                'description' => 'A book about API development',
                'publication_year' => '2019',
                'created_at' => now()->setMilliseconds(0)->
                    toJSON(),
                'updated_at' => now() ->setMilliseconds(0)->
                    toJSON(),
            ]
        ]
    ])
```

```

        ]
    ]
    ])->assertHeader('Location', url('/api/v1/books/1'));

    $this->assertDatabaseHas('books', [
        'id' => 1,
        'title' => 'Building an API with Laravel',
        'description' => 'A book about API development',
        'publication_year' => '2019',
    ]);
}

```

And the test passes again. You should do this for the rest of the failing tests.

The next thing we will be taking a look at is the remaining route for our user.

### *Current authenticated user*

If we take a look at our **routes/api.php** file, we see that we have a route for the current user, which is actually still using a closure:

```

Route::get('/users/current', function (Request $request) {
    return $request->user();
});

```

All this closure does is to return the current authenticated user, but it is not being done in the conventions of the JSON:API specification, so let's just create a controller for this route and add the correct response document.

First, we should create a test so let's create a new **tests/Feature/CurrentAuthenticatedUserTest.php** and add the following test to it:

```

<?php

namespace Tests\Feature;

use App\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Laravel\Passport\Passport;
use Tests\TestCase;

class CurrentAuthenticatedUserTest extends TestCase
{
    use DatabaseMigrations;

    /**
     * @test
     * @watch
     */
    public function
        it_returns_the_current_authenticated_user_as_a_resource_object
        ()
    {
        $this->withoutExceptionHandling();
        $user = factory(User::class)->create();
        Passport::actingAs($user);

        $this->getJson("/api/v1/users/current", [
            'accept' => 'application/vnd.api+json',
            'content-type' => 'application/vnd.api+json',
        ]) ->assertStatus(200)
            ->assertJson([
                "data" => [
                    "id" => $user->id,
                    "type" => "users",
                    "attributes" => [
                        'name' => $user->name,
                        'email' => $user->email,
                        'created_at' => $user->created_at->toJSON(),
                        'updated_at' => $user->updated_at->toJSON(),
                    ]
                ]
            ])
    }
}

```

```

        });

    }

}

```

This test is basically just a copy of our **it\_returns\_a\_user\_as\_a\_resource\_object** from our **tests/Feature/UsersTest.php**, but instead of fetching a user by an **ID**, we make a query for the specific route:

```
GET: /api/v1/users/current
```

At the moment this test fails — not because the route doesn't exist or the returned JSON is incorrect — it is because of the routes' position in our **routes/api.php** file. Right now, it is placed after the **Route::apiResource** method for our users, which will make Laravel think that we are querying for a user **ID** called **current**. To make our specific route callable, we need to place our route above the **Route::apiResource** like this:

```

// Users
Route::get('/users/current', '
    CurrentAuthenticatedUserController@show');
Route::apiResource('users', 'UsersController');
Route::get('users/{user}/relationships/comments', '
    UsersCommentsRelationshipsController@index')->name('users.
    relationships.comments');
Route::patch('users/{user}/relationships/comments', '
    UsersCommentsRelationshipsController@update')->name('users.
    relationships.comments');
Route::get('users/{user}/comments', '
    UsersCommentsRelatedController@index')->name('users.comments');

```

Also, while we are at it, we might as well give the reference to our controller in the route and then jump into our terminal and make the controller like this:

```
php artisan make:controller CurrentAuthenticatedUserController
```

If we go into our newly created **app/Http/Controllers/CurrentAuthenticatedUserController**, we can make a quick implementation of the **show** method that will make our test pass like this:

```
<?php

namespace App\Http\Controllers;

use App\Http\Resources\JSONAPIResource;
use Illuminate\Http\Request;

class CurrentAuthenticatedUserController extends Controller
{
    public function show(Request $request)
    {
        return new JSONAPIResource($request->user());
    }
}
```

This is it for our current authenticated user. It now also adheres to the JSON:API specification.

## Cross-Origin Resource Sharing

Cross-Origin Resource Sharing, also known as CORS, is a feature built into modern browsers to prevent a resource from one domain to make a request to another resource on another domain or sub-domain. In other words, if you are visiting

```
http://example.com
```

and a script on that page wants you to access the data through an API on

```
http://api.example.com
```

CORS will prevent this.

The benefit of this is that attackers can't plant scripts on websites that could then access your bank on your behalf, so it's simply a security measure.

But what do you do when you have a domain like example.com that needs the users to be able to make a request to the API domain?

This is done by the API server in the form of an **Access-Control-Allow-Origin** header that tells which domains can make requests to this domain. Your browser then reads this header and allows the communication between domains.

This is a very common scenario when working with APIs, since APIs are often placed on another domain or subdomain for better separations of concerns on a server.

In Laravel, this can be done in many ways. We can write a middleware ourselves that adds the needed **Access-Controller-Allow-\*** headers, or we can use a package that has already taken care of this for us.

We have talked about Spatie earlier, when we went through both sorting, pagination and filtering. That same company has another package that takes care of CORS for us, so let's install that into our application. We do this by requiring it through composer like this:

```
composer require spatie/laravel-cors
```

Then, when the package has been downloaded, we add the provided middleware inside our **app/Http/Kernel.php** in the array of the **middleware** property like this:

```
protected $middleware = [
    \App\Http\Middleware\CheckForMaintenanceMode::class,
    \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
    \App\Http\Middleware\TrimStrings::class,
    \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull
        ::class,
    \App\Http\Middleware\TrustProxies::class,
    \Spatie\Cors\Cors::class,
];
```

Next, we export the config file like this:

```
php artisan vendor:publish --provider="Spatie\Cors\
    CorsServiceProvider" --tag="config"
```

If you go into **config/cors.php**, you can see the default settings the package ships with. These are useful for now, but keep in mind for your own APIs that you give specific domain names instead of the **\*** given in the array of the **allow\_origins** key. This is especially needed if you are using cookies, which will only be allowed by specific domain names.

Awesome! Now we can make requests to our API from our browsers.

## Summary

Phew, this was a long chapter! We hope you got through it without too much difficulty. Let's just recap what we have covered, because it has certainly been a lot.

We went through the implementation of the relationship between authors and books, so that the entire relationship has been implemented.

We implemented our users resource and took UUIDs and roles into account.

We gave you an assignment to implement the comments resource on your own and if you have made it here, you have done a great job!

Then, we implemented the relationships between users and comments, where we also implemented the **toMany** relationship into our **JSONAPIService** class. We dug a little deeper into Laravel, and discovered that sometimes it can benefit to look under the surface, to find methods that can help us with our implementations.

We gave you yet another assignment to implement the relationship between books and comments, which you did really well.

After this, we went through the process of implementing the relationships between both comments and users, and comments and book. Here, we also implemented the **toOne** relationship, leveraging a lot of the knowledge we gained about models from implementing the **toMany** relationships.

From there, we implemented the last part of relationships, with the ability to both create and update relationships when creating and updating resources. We went through filtering of users, so we can distinguish users and administrators. We implemented authorization using Laravel's policy classes and Gate feature.



We implemented a route for the current authenticated user that adheres to the JSON:API specification

Lastly, we added CORS to our applications so we are able to make request to our API from our browser.

We are now done with our API, and we hope you liked the journey as much as we did. We also hope you have gained a lot of new skills that you want to try out immediately. In any case, we think that you should read on and learn where to go from here.

## Where to go from here

We have now implemented our entire API and we have implemented it according to the JSON:API specification. You have now gained an insight into how to build an API with Laravel leveraging the JSON:API specification. You have learned how to plan, build and test every part of the API and if you look at the application you have now, you actually have a bunch of classes you can port over to other applications as well. The code implemented in our requests, resources, and service classes does not contain anything specific to Anna's Bookstore and can be reused together with a config file that follows the same conventions.

From here on out, we first and foremost encourage you to work a bit more with this application. As of right now, visitors to the bookstore have to be registered to see the books they can buy at Anna's Bookstore. Maybe there are some routes that could be moved on the other side of authentication, so that the books can be shown.

We will, of course, also encourage you to take what you have learned and implement it into one of your own applications. You don't have to start from scratch — a small, existing application would also be a good starting point

to implement what you have learned. This will cement your knowledge even more and you will start to find yourself getting better at writing tests and actually seeing how much these benefit you more and more.

Another thing we think you should do now, is to try to test out one of the client libraries listed on the JSON:API specification website and see how much it benefits you to have an API with clear conventions.

## Chapter 9 - Bonus

In this chapter, we will give a bit of bonus information. It is completely optional for you to read this chapter, but we think you should read it anyway. We will be finishing our **SetupDevEnvironment** command, and then we will be looking at how to consume our API from the frontend. We will be looking at existing client libraries, which makes it super easy for us to implement our API on the frontend.

### Completing our SetupDevEnvironment command

First, let's complete our **SetupDevEnvironment** command so let's jump into our **app/Console/Commands/SetupDevEnvironment.php**. At the moment, it looks like this:

```
<?php

namespace App\Console\Commands;

use App\User;
use Illuminate\Console\Command;
use Illuminate\Support\Facades\Artisan;
```

```

use Laravel\Passport\PersonalAccessTokenResult;

class SetupDevEnvironment extends Command
{
    protected $signature = 'dev:setup';

    protected $description = 'Sets up the development environment';

    public function __construct()
    {
        parent::__construct();
    }

    public function handle()
    {
        $this->info('Setting up development environment');

        $this->migrateAndSeedDatabase();
        $user = $this->createJohnDoeUser();
        $this->createPersonalAccessClient($user);
        $this->createPersonalAccessToken($user);

        $this->info('All done. Bye!');
    }

    public function migrateAndSeedDatabase()
    {
        $this->call('migrate:fresh');
        $this->call('db:seed');
    }

    public function createJohnDoeUser()
    {
        $this->info('Creating John Doe user');
        $user = factory(User::class)->create([
            'name' => 'John Doe',
            'email' => 'john@example.com',
            'password' => bcrypt('secret'),
        ]);
    }
}

```

```

        $this->info('John Doe created');
        $this->warn('Email: john@example.com');
        $this->warn('Password: secret');

        return $user;
    }

    public function CreatePersonalAccessClient($user)
    {
        $this->call('passport:client', [
            '--personal' => true,
            '--name'      => 'Personal Access Client',
            '--user_id'   => $user->id
        ]);
    }

    public function CreatePersonalAccessToken($user)
    {
        $token = $user->createToken('Development Token');
        $this->info('Personal access token created successfully.');
```

*(Note: The original image contains a stray closing brace '}' at the end of the code block, which has been removed for accuracy.)*

If we take a look at the **handle** method, we see that we:

- Run a migrate fresh command to drop our database tables and migrate them from scratch
- Run a seed command, which will seed our database with authors
- We create a John Doe user
- We create an access client for John Doe
- We create an access token for John Doe

Now that we have implemented our entire API, we want some more models to be seeded when running this command. We still want a bunch of authors, but it would be nice with some books that are associated with these authors, so

that we have some data to query.

Now that we have multiple user roles, it would be nice if we had an admin user and a regular user as well, so we should add this too and also make sure that the new user gets an access token.

Last, but not least, we want some more room around the details being outputted — especially the details about our users, so we should do something about this too.

Let's start out by creating a seeder for books, so let's jump into our terminal and run the following command:

```
php artisan make:seeder BooksTableSeeder
```

Let's go into our new **database/seeds/BooksTableSeeder.php** and add the following code to the **run** method:

```
public function run()
{
    Author::all()->each(function(Author $author){
        $books = factory(Book::class, 2)->create();
        $author->books()->sync($books->pluck('id'));
    });
}
```

Here, we make a query for all the authors in our database, create two books for each author, and associate the books with the author, using the **pluck** method on the books collection to get the **ids** only.

To ensure that the seeder for authors is running, let's jump into the **database/seeds/DatabaseSeeder** and add our new seeder to the **run** method, and make sure to put it below the seeder for authors like this:

```
public function run()
{
    $this->call(AuthorsTableSeeder::class);
    $this->call(BooksTableSeeder::class);
}
```

If you run our artisan command now, you should see that the books are being seeded as well.

Back in the **app/Console/Commands/SetupDevEnvironment.php** file, we can continue to the next part, which is the creation of users. At the moment, we have a dedicated method for creating our John Doe user. Let's make this method a bit more general, so we can create the users we want and also make it possible to give a different role and password like this:

```
public function createUser($name, $email, $role = 'user', $password
    = 'secret')
{
    $this->info(PHP_EOL);
    $this->info("Creating {$name} $role");
    $user = factory(User::class)->create([
        'name' => $name,
        'email' => $email,
        'role' => $role,
        'password' => Hash::make($password),
    ]);

    $this->info("Done");
    return $user
}
```

With this method, we take in the **name**, **email**, **role** and **password** as arguments. Both **role** and **password** has a default value so that they can be skipped.

Inside the method, we add a call to the **info** method where we pass in a **PHP\_EOL** constant. We use this to make a line break in the terminal, so that we get some air around our user information. Then, we use a factory to create the user just like before, replacing the hard coded values with the arguments

values and return the user.

Our **handle** method can then be changed to this:

```
public function handle()
{
    $this->info('Setting up development environment');
    $this->MigrateAndSeedDatabase();

    $user = $this->createUser('John Doe', 'john@example.com', 'admin
    ');
    $this->CreatePersonalAccessClient($user);
    $this->CreatePersonalAccessToken($user);

    $this->info('All done. Bye!');
}
```

It doesn't make that much of a difference, but let's do something to those calls to **CreatePersonalAccessClient** and **CreatePersonalAccessToken**.

There's really no need for these to be called in the **handler**. We can just call these in our **createUser** method instead, or better yet, let's make a dedicated method for the call to these, so we can make sure to get some air around the information being outputted to our terminal like this:

```
public function createPersonalAccessClientAndTokenForUser(User
    $user): void
{
    $this->info(PHP_EOL);
    $this->info("Creating personal access client and token for {
        $user->name}");
    $this->CreatePersonalAccessClient($user);
    $this->CreatePersonalAccessToken($user);
    $this->info(PHP_EOL);
}
```

This method can then be called in our **createUser** method like this:

```

public function createUser($name, $email, $role = 'user', $password
    = 'secret')
{
    $this->info(PHP_EOL);
    $this->info("Creating {$name} $role");
    $user = factory(User::class)->create([
        'name' => $name,
        'email' => $email,
        'role' => $role,
        'password' => Hash::make($password),
    ]);

    $this->createPersonalAccessClientAndTokenForUser($user);
    $this->info("Done");
}

```

Then we do not need to pass the created user around in our handler anymore and we can just create users like this and be sure that the output in the terminal is nice and easy to read:

```

public function handle()
{
    $this->info('Setting up development environment');
    $this->migrateAndSeedDatabase();

    $this->createUser('John Doe', 'john@example.com', 'admin');
    $this->createUser('Jane Doe', 'jane@example.com');

    $this->info('All done. Bye!');
}

```

We also get the benefit of now easily being able to add more users, so here we have added Jane Doe as well, which is just a regular user. If you run this artisan command now, you will have everything except comments prepared for you, and multiple user roles to test through.



## JSON:API Client Implementations

One of the biggest benefits of using the JSON:API specification, and generally using a strict set of protocols, is the ability to create not only reusable rules, but also reusable implementations.

On the official JSON:API Site, there's a page dedicated to both client and server implementations and there is a lot to choose from ORM implementations that resemble Laravel Eloquent, to more simple implementations.

Unfortunately, at the time of writing this, the quality of the documentation from implementation to implementation varies a bit much, so we have chosen an implementation we thought was simple and easy to get working right away, which is named **devour-client**. This is also the client we will use later on to show you how you can easily consume your API with Vue, but for now, let's just see how this works and how easy it is to get working without API.

To be able to get something up and running quickly without having to do a lot of work with webpack first, we will be using Laravel Mix. Fortunately, it's possible to use it outside of a Laravel project and it's really easy to install thanks to a nice step by step guide by Jeffrey Way, the creator and maintainer of Laravel Mix.

For this part, we assume that you have **node** and **npm** installed on your machine.

So let's jump into our terminal and change into the directory on your machine, where you keep all of your code. Let's create a new folder here, we will be calling ours **jsonapi-client**.

Then we need to initialize our project, which we can do through **npm** like this:

```
npm init -y
```

This will create a new `package.json` file for our project, which can hold the dependencies for our project, just like `composer.json` files does for our PHP projects. We can then install Laravel Mix like this:

```
npm install laravel-mix --save-dev
```

When it is done installing, we need to copy the initial **webpack.mix.js** configuration file, which we can use to set up which files we need to compile, just like the **webpack.mix.js** configuration file you know from your Laravel projects. This is done through this command in the terminal:

```
cp node_modules/laravel-mix/setup/webpack.mix.js ./
```

We are almost done. We just need to add the following scripts into our `package.json` file:

```
"scripts": {
  "dev": "npm run development",
  "development": "cross-env NODE_ENV=development node_modules/
    webpack/bin/webpack.js --progress --hide-modules --config=
    node_modules/laravel-mix/setup/webpack.config.js",
  "watch": "npm run development -- --watch",
  "hot": "cross-env NODE_ENV=development node_modules/webpack-dev
    -server/bin/webpack-dev-server.js --inline --hot --config=
    node_modules/laravel-mix/setup/webpack.config.js",
  "prod": "npm run production",
  "production": "cross-env NODE_ENV=production node_modules/
    webpack/bin/webpack.js --no-progress --hide-modules --
    config=node_modules/laravel-mix/setup/webpack.config.js"
},
```

This will make our package.json file look like this:

```
{
  "name": "jsonapi-client",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "npm run development",
    "development": "cross-env NODE_ENV=development node_modules/
      webpack/bin/webpack.js --progress --hide-modules --config
      =node_modules/laravel-mix/setup/webpack.config.js",
    "watch": "npm run development -- --watch",
    "hot": "cross-env NODE_ENV=development node_modules/webpack-
      dev-server/bin/webpack-dev-server.js --inline --hot --
      config=node_modules/laravel-mix/setup/webpack.config.js",
    "prod": "npm run production",
    "production": "cross-env NODE_ENV=production node_modules/
      webpack/bin/webpack.js --no-progress --hide-modules --
      config=node_modules/laravel-mix/setup/webpack.config.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "laravel-mix": "^4.0.15"
  }
}
```

Then we should be all set. Laravel Mix just has to install the last dependencies, which it will do as soon as we run a compilation of our assets files, even though we don't have any yet:

```
npm run dev
```

After the additional dependencies have been installed, let's then create a

new folder for our assets, called **src**. Inside the **src** folder, let's create a new **src/app.js** javascript file, which will be our main javascript file. This is the file that will import the rest of the javascript files we need, which then makes this the file that needs to be compiled with Laravel Mix. We tell Laravel Mix which files it need to compile through the **webpack.mix.js** file, so let's open that up:

```
let mix = require('laravel-mix');

mix.js('src/app.js', 'dist/').sass('src/app.scss', 'dist/');
```

In this file, Laravel Mix is being required at the top and assigned to the **mix** variable, which we then use to call both the **js** and **sass** functions on. We won't be needing SASS in this project though, so let's delete the call to this function, leaving only the compilation of the **src/app.js** file:

```
let mix = require('laravel-mix');

mix.js('src/app.js', 'dist/');
```

When being compiled, the **src/app.js** file will be placed in a **dist** folder. We don't have that yet, so let's create this folder and let's also create a **dist/index.html** file we can use to run the **app.js** javascript file in our browser. We do this by adding the script to our **dist/index.html** file like this:

```
<!DOCTYPE HTML>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <meta name="viewport" content="width=device-width, initial-scale
```

## FINISHING UP

```
    =1.0, maximum-scale=1.0, user-scalable=0">
<meta name="apple-mobile-web-app-status-bar-style" content="black
">
<meta name="apple-mobile-web-app-capable" content="yes">

<title>JSON:API Client Implementation</title>
<script type="text/javascript" src="app.js" defer></script>
</head>

<body>
</body>

</html>
```

The last thing we need to do, is to be able to write some code and do some queries to our API the download of the actual client implementation. As we mentioned, we will be using `devour-client`, so let's install it like this:

```
npm install --save devour-client
```

While this is installing, let's take a look at the documentation for the client. You can find this at their Github repository:

```
https://github.com/twg/devour#readme
```

Looking at their quickstart, we can see that they import the client and afterward create an instance of this client and assign this to the **jsonApi** constant. They then use this instance to define a model, which in this case is a `Post`. Afterward, they show how to make calls to fetch posts and also how to create and update posts.

Let's try to recreate this with our own API. As we have mentioned earlier,

we are using Laravel Valet for our local setup, where the API is served on

```
http://annas-bookstore.test
```

you should take note on where yours is served, unless you are following the same setup as us.

To be able to test our javascript code, we should set up Laravel Valet for this as well. In our terminal, let's change directory so that we are in the **dist** directory. Here, we will leverage Laravel Valet's **link** feature to be able to serve the files inside the **dist** directory by a given domain name. Here, we want to serve these files under the **jsonapi-client.test**. By default, the top-level domain for Laravel Valet is **.test** so we will only need to give the name of the domain like this:

```
valet link jsonapi-client
```

Then, we can change back to the parent directory again so we can run Laravel Mix when needed.

To test that everything is set up, let's go into our **src/api.js** file in our editor and write the following line of code:

```
console.log('Hello world');
```

Save the file and go out into the terminal, and run Laravel Mix like this:

```
npm run dev
```

Go into your browser and, if you have the same setup as us, using Laravel Valet to do the linking we just showed, you should be able to see a blank page at

```
http://jsonapi-client.test
```

Here, you should open the console in your browser and be able to see **Hello World** being printed in the console.

Great! So far, so good. In the terminal, let's make Laravel Mix watch for changes and then compile our code, just like Laravel Test Watcher. You can do this by running the following:

```
npm run watch
```

Then, it's time to implement the `devour-client`, so we can make some calls to our API. We will be following the recipe given from the documentation, so let's first import the package and instantiate a new instance of `JsonApi` like this:

```
import JsonApi from 'devour-client';

const jsonApi = new JsonApi({
  apiUrl: 'http://annas-bookstore.test/api/v1',
});
```

Here, we are giving our **API URL**, so that our client can make all future calls to this. You should ensure that the right URL to the API on your setup is given.

Before we're able to make calls to our API, we should ensure that we are authenticated. We can do this by getting an access token from our API's Laravel Application through our **DevSetupEnvironment** command, so let's jump over to our API's Laravel Application and run the following artisan command:

```
php artisan dev:setup
```

You should get an output like this, except for the <placeholder> notes:

```
Creating personal access client and token for John Doe
Personal access client created successfully.
Client ID: 1
Client secret: <your client key>
Personal access token created successfully.
Personal access token: <your access token>
```

Take a copy of the access token and let's go back to our code in the **jsonapi-client** project and insert the token in the code like this:

```
const token = 'your access token';
jsonApi.headers['Authorization'] = `Bearer ${token}`;
```

Of course, this is bad practice. You should never hardcode access tokens in like this, but this is only for testing and exploring the client implementation.

Then, we should define a model the client can use to populate data from our



API into. This gives us a nice conversion from JSON into javascript objects we can use in our code, which is super convenient. Here, we will define a model for authors, since we will have test data for these by simply running our **DevSetupEnvironment** command on our Laravel applications:

```
jsonApi.define('author', {
  Name: '',
  Books: {
    jsonApi: 'hasMany',
    type: 'books',
  },
  created_at: '',
  updated_at: '',
});
```

We are now ready to begin making requests to our API. Let's follow the example from the documentation and fetch all authors, which is easily done like this:

```
jsonApi.findAll('author').then(author => {
  console.table(author.data);
});
```

We use the `console.table` to get a nice table layout in our browser's console and if you are using Google Chrome like us, you should see a result like this:

| (index) | id  | type      | name        | created_... | updated_... | books    |
|---------|-----|-----------|-------------|-------------|-------------|----------|
| 0       | "1" | "authors" | "Johan S... | "2019-05... | "2019-05... | Array(0) |
| 1       | "2" | "authors" | "Jarvis ... | "2019-05... | "2019-05... | Array(0) |
| 2       | "3" | "authors" | "Vivian ... | "2019-05... | "2019-05... | Array(0) |
| 3       | "4" | "authors" | "Twila C... | "2019-05... | "2019-05... | Array(0) |
| 4       | "5" | "authors" | "Emmanue... | "2019-05... | "2019-05... | Array(0) |

Let's try to make a request to a single author, which can be done like this:

```
jsonApi.find('author', 1).then(author => { console.table(author.data);});
```

This result is a bit different, since it's only outputting a single resource which will result in an object instead of an array in javascript. When using `console.table` or `console.log`, it will place the result in the top of the console which should look like this:

| (index)    | Value                         |
|------------|-------------------------------|
| id         | "1"                           |
| type       | "authors"                     |
| name       | "Johan Schmidt"               |
| created_at | "2019-05-21T12:12:50.000000Z" |
| updated_at | "2019-05-21T12:12:50.000000Z" |
| books      |                               |

As you hopefully recall from our API implementation, authors have a relationship to books, which is also being seeded when running our **DevSetupEnvironment** command, so let's try to include these.

Before we can do this though, we need to define the model for a book, so that the incoming JSON from our API can be transformed to a javascript object by **devour**. The model for a book is defined like this:

```
jsonApi.define('book', { title: '', description: '',
  publication_year: '', authors:{ jsonApi: 'hasMany', type: '
  authors', }, created_at: '', updated_at: '',});
```

We add the attributes we want to have transformed and add the inverse of the relationship, and that is actually it. We can then make a request, including the related books for our authors like this:

```
jsonApi.findAll('author', {include: 'books'}).then(author => {
  console.table(author.data);});
```

This will give you the following result in the console. Notice the books column in the table, which now states that there are two books for each author:

| (index) | id  | type      | name        | created_... | updated_... | books    |
|---------|-----|-----------|-------------|-------------|-------------|----------|
| 0       | "1" | "authors" | "Johan S... | "2019-05... | "2019-05... | Array(2) |
| 1       | "2" | "authors" | "Jarvis ... | "2019-05... | "2019-05... | Array(2) |
| 2       | "3" | "authors" | "Vivian ... | "2019-05... | "2019-05... | Array(2) |
| 3       | "4" | "authors" | "Twila C... | "2019-05... | "2019-05... | Array(2) |
| 4       | "5" | "authors" | "Emmanue... | "2019-05... | "2019-05... | Array(2) |

One of the cool things by using a client library like this, is that they do the heavy lifting for you in terms of filtering the included resources and attaching them to the author they actually belong to. If you recall, all of these are just looped out into the **included** top-level member in our response document. You can see this by unfolding the array right under the table in the console like this:

```

▼ Array(5) ⓘ
  ▼ 0:
    ▼ books: Array(2)
      ► 0: {id: "1", type: "books", title: "Mr. Cletus Armstrong V", descript...
      ► 1: {id: "2", type: "books", title: "Ali Gutmann", description: "Volup...
        length: 2
      ► __proto__: Array(0)
    created_at: "2019-05-21T12:12:50.000000Z"
    id: "1"
    name: "Johan Schmidt"
    type: "authors"
    updated_at: "2019-05-21T12:12:50.000000Z"
    ► __proto__: Object
  ► 1: {id: "2", type: "authors", name: "Jarvis Champlin II", created_at: "20...
  ► 2: {id: "3", type: "authors", name: "Vivian Ferry PhD", created_at: "2019...
  ► 3: {id: "4", type: "authors", name: "Twila Collins", created_at: "2019-05...
  ► 4: {id: "5", type: "authors", name: "Emmanuelle Schuster", created_at: "2...
    length: 5
  ► __proto__: Array(0)

```

Here, the first author in the array is unfolded and unfolding the books array. you can see the following books, which belong to this author.

We would go over create, update and delete as well, but we will be using these in the next section about Vue, so you will have to wait a bit with these.

For now, we hope you can see the big advantages of using a client implemen-  
tation and, if not now, you will quickly realize it when working with Vue.

## Consuming your API with Vue

Vue.js has been our go-to javascript framework for years now, ever since we were introduced to it by the Laravel community. Therefore, we wanted to include a section of this bonus chapter for you to learn how to integrate your API into a Vue application. This is not meant to teach you about Vue itself, but more to show you how to leverage an API with strict protocols that leads to client implementation that makes our Vue development easier as well.

For our Vue.js applications, we typically use other packages or plugins for Vue, like the Vue Router, the Vuex store, and many more. To make this as easy to grasp as possible, we won't be using Vuex but keep the calls to our API in our components, we will be using Vue router though, but we will keep the routes at the minimum and we will also demonstrate a drag and drop component we use very often.

Like any good cooking show on TV, we have prepared a little something from home, so you will have to clone our Github repository in order to get the code. We will have an **annas-bookstore-vue** application, where everything has been implemented, and we will go through this together.

Before we dive into the code, we think it is important to take a look at how we want to consume our API, how we are hosting the frontend and backend, and how we handle the authentication with Laravel Passport.

We will be hosting our Vue application together with our Laravel application for this scenario, which means that we can leverage the Laravel Passport middleware called **CreateFreshApiToken**, which we talked about in chapter 4 and the section about Laravel Passport. This middleware takes care of the entire token flow, so you don't have to spend time issuing anything. We especially do this since the Laravel framework ships with a Vue boilerplate, where everything is set up and ready for you to build something with it. This means that we can focus a little more on the communication with our API than the actual authentication parts.

For this example, we will once more be using Laravel Valet for our local setup. Remember, that if you are not using Laravel Valet on your setup, things might not be exactly the same.

## Setup

Let's jump into the **annas-bookstore-vue** directory. If you have been through the entire book, you should have our repository cloned down to your machine already, where you will find this directory. If not you should go through the Cloning from Github section in chapter 7.

First, we would like to be able to access this through our browser and the name of the directory is actually fine for the domain of this project, which means that calling the **link** command on Laravel Valet will suffice in order to set this up. So let's go into our terminal and, of course, make sure that we have changed directories, so that we are in the root of this **annas-bookstore-vue** project and then call the Laravel Valet command like this:

```
valet link
```

Then, we should install our dependencies, both for PHP but also Javascript. We'll start out with PHP using composer like this:

```
composer install
```

Then, we install the Javascript dependencies like this:

```
npm install
```

Last, but not least, to make sure that all dependencies has been downloaded, let's just run the asset compilation through Laravel mix like this:

```
npm run dev
```

To make sure that the Laravel application can connect to the database and so forth, you should copy over the `.env` file from your **annas-bookstore** project, since we want to connect to the same database and use the same data.

## Dependencies

Before we go into the code, we just wanted to list the dependencies that npm are installing here. Some of them you should be familiar with, since they are shipped with Laravel, but there are a few we have added that we want to point out, so let's open the **packages.json** file and have a look at the contents:

```
{
  "private": true,
  "scripts": {
    "dev": "npm run development",
    "development": "cross-env NODE_ENV=development node_modules/
      webpack/bin/webpack.js --progress --hide-modules --
      config=node_modules/laravel-mix/setup/webpack.config.js
    ",
    "watch": "npm run development -- --watch",
    "watch-poll": "npm run watch -- --watch-poll",
    "hot": "cross-env NODE_ENV=development node_modules/webpack-
      dev-server/bin/webpack-dev-server.js --inline --hot --
      config=node_modules/laravel-mix/setup/webpack.config.js
    ",
    "prod": "npm run production",
    "production": "cross-env NODE_ENV=production node_modules/
      webpack/bin/webpack.js --no-progress --hide-modules --
      config=node_modules/laravel-mix/setup/webpack.config.js"
  },
  "devDependencies": {
    "axios": "^0.18",
    "bootstrap": "^4.0.0",
```

```

    "cross-env": "^5.1",
    "jquery": "^3.2",
    "laravel-mix": "^4.0.7",
    "lodash": "^4.17.5",
    "popper.js": "^1.12",
    "resolve-url-loader": "^2.3.1",
    "sass": "^1.15.2",
    "sass-loader": "^7.1.0",
    "vue": "^2.5.17",
    "vue-template-compiler": "^2.6.7"
  },
  "dependencies": {
    "devour-client": "^2.0.17",
    "vuedraggable": "^2.21.0",
    "vue-router": "^3.0.6"
  }
}

```

We have arranged this file so that all of the dependencies defined from the Laravel framework installation are under the **devDependencies** member and the dependencies we have installed are under the **dependencies** member.

Here, we have installed the **devour-client** again, since we want to leverage this and show how easy it is to consume your API in Vue, using this client implementation.

Then, we have installed **vuedraggable**, which we will use to solve the attachment of authors to books.

Last, but not least, we have installed **vue-router** for navigation in our Vue application.

Now that we know about the dependencies installed, we can look at how our Vue app is being bootstrapped, since there are some things to note here.



## *Bootstrapping the Vue application*

Open up the **annas-bookstore-vue** directory in your favorite editor, so that you can access all the files in the project. The files we will focus on is mostly the files in **resources/js**.

The first file we should take a look at is the **resources/js/bootstrap.js** file, which is one of the files that Laravel ships with that actually takes care of the token flow for us:

```
/**
 * We'll load the axios HTTP library which allows us to easily issue
 * requests
 * to our Laravel back-end. This library automatically handles
 * sending the
 * CSRF token as a header based on the value of the "XSRF" token
 * cookie.
 */

window.axios = require('axios');

window.axios.defaults.headers.common['X-Requested-With'] = '
XMLHttpRequest';

/**
 * Next we will register the CSRF Token as a common header with
 * Axios so that
 * all outgoing HTTP requests automatically have it attached. This
 * is just
 * a simple convenience so we don't have to attach every token
 * manually.
 */

let token = document.head.querySelector('meta[name="csrf-token"]');

if (token) {
```

```

    window.axios.defaults.headers.common['X-CSRF-TOKEN'] = token.
    content;
  } else {
    console.error('CSRF token not found: https://laravel.com/docs/
      csrf#csrf-x-csrf-token');
  }

import Devour from 'devour-client';
window.jsonApi = new Devour({
  apiUrl: 'http://annas-bookstore-vue.test/api/v1',
});

jsonApi.axios = axios;

require('./models');

```

We have not included the top of the file, since it's just requiring **lodash** and **popper.js**.

The first part of this example shows how **Axios**, which is an HTTP client we use to make requests to our API, is set up. Here, the **CSRF** token is set up as a common header, so that we don't have to add this on every request. Laravel requires that we send along a **Cross-Site Request Forgery** token to ensure that we are, in fact, the correct user making the request and not somebody or something with a malicious intent, trying to make request on behalf of an authenticated user.

Next, we set up our **devour-client**, where we set the **apiUrl** to the new url of our system:

```
http://annas-bookstore-vue.test/api/v1
```

Remember that if your local setup is not like this, you will have to change this

URL so that it matches your local setup.

Since the **devour-client** is also using Axios for http requests, we can let it use the Axios client already set up by Laravel, which will also make us able to leverage the token flow with Laravel Passport without lifting a finger.

And lastly, we require the models, which we have been dedicated to their own files, so that it is easier to grasp. Opening up the **resources/js/models.js** file will show you the two models we have implemented for this example, which is actually the exact same implementation of models as we had in the last section about JSON:API Client implementations.

The next file we should take a look at, which we also view as a part of the bootstrapping of the application, is the **resources/js/app.js** file:

```
require('./bootstrap');
window.Vue = require('vue');
import VueRouter from 'vue-router';
Vue.use(VueRouter);

Vue.component('example-component', require('./components/
    ExampleComponent.vue').default);

Vue.component('passport-clients',
    require('./components/passport/Clients.vue').default
);
Vue.component('passport-authorized-clients',
    require('./components/passport/AuthorizedClients.vue').default
);
Vue.component('passport-personal-access-tokens',
    require('./components/passport/PersonalAccessTokens.vue').
        default
);

import routes from './routes';
```

```
const router = new VueRouter({
  routes
});

const app = new Vue({
  router,
  el: '#app'
});
```

Here, the file mostly follows what is shipped with Laravel out of the box. Of course, there are the passport components we added earlier, but the thing to note is the import of Vue router in the top of the file, right after the bootstrap and import of Vue. We are then telling Vue to use the Vue router as a plugin, and after the passport components, we import a file we have dedicated to our routes, much like we did with our models.

Then, we instantiate the router and add this to Vue. When we ran the **auth:make** artisan command during the chapter 4, in the section about Laravel Passport, we got a couple of Blade views that is actually serving as the basis for our entire authentication and Vue application. If we take a look at the **resources/views/layouts/app.blade.php** file, the app is being mounted at the top most **div** element using an id attribute with the value **app**. Then, we have all the implementation of the navigation, before we end up down at the **main** element, where we use a **yield** Blade directive. This directive makes it possible for us to tell which content that should be placed in between the **main** elements using Laravel Blade.

The content is determined by Laravel and the route you are accessing, if we added our Vue app here, it would be accessible by anyone. The place where we want our Vue app to kick in is when a user has been authenticated, which means that this has to happen in a view that inherits this one and give some content to the **yield** directive. Through the **auth:make** artisan command, we also get a **resources/views/home.blade.php** file, which is the view file for the

**home** route that we will land on when we are authenticated. Here, we have removed everything that Laravel provides and instead we add our router view, so that our Vue application won't start until we are authenticated and on this view. This also means that our Vue router won't take over until a user has logged in.

As mentioned, the navigation has been kept in the **resources/views/layout-app.blade.php** file. Here, we have also added the few links for the Vue router to keep everything in the same place. Not that these links won't be shown unless the user is authenticated, which is done by leveraging the Laravel Blade **@guest** directive.

Enough about the setup and bootstrapping. Let's look at the application and the actual API calls.

### *The application*

To understand this application, we think it's easiest to begin with the **resources/js/routes.js** file, since we can then see which routes are possible to take in our application, but also easily see which components that are associated with which route:

```
import Passport from './components/passport/passport.vue';

import AuthorsIndex from './components/authors/index.vue';
import AuthorsCreate from './components/authors/create.vue';
import AuthorsUpdate from './components/authors/update.vue';

import BooksIndex from './components/books/index.vue';
import BooksCreate from './components/books/create.vue';
import BooksUpdate from './components/books/update.vue';

export default [
```

```

{path: '/', component: Passport, name: 'passport'},

{path: '/authors', component: AuthorsIndex, name: 'authors.index'
},
{path: '/authors/create', component: AuthorsCreate, name: 'authors
.create'},
{path: '/authors/update/:id', component: AuthorsUpdate, name: '
authors.update'},

{path: '/books', component: BooksIndex, name: 'books.index'},
{path: '/books/create', component: BooksCreate, name: 'books.
create'},
{path: '/books/update/:id', component: BooksUpdate, name: 'books.
update'},
];

```

We have added seven routes to this application, where the first route is a route dedicated to the Laravel Passport components, so that you can still work with access tokens and so forth.

Then, we have created routes for the CRUD of authors and the CRUD of books. The delete part of the CRUD is actually done in the index view for both resources. This is why there are only three routes for each resource.

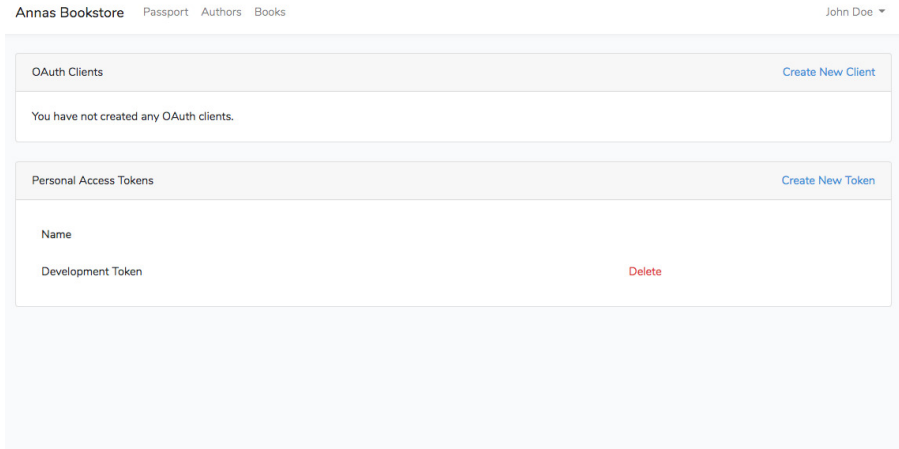
Before we look at the code, let's log in to the application and see how it works. If you are using the same setup as us, it's as easy as accessing:

```
http://annas-bookstore-vue.test/login
```

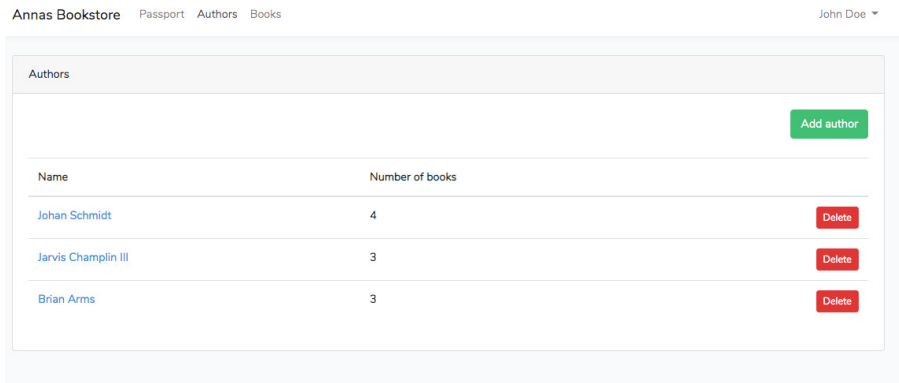
If you have run a **dev:setup** artisan command, there should already be a John Doe user ready for you to use, so let's log in as him with the email: john@example.com and password which is **secret**.

## FINISHING UP

The landing page here is still the **passport** components noted by the `/` path in the routes file, which is the place the router will first land:



Let's click on the **Authors** link in the menu, which should take us to the listing of authors as seen here:



If we look at the **resources/js/routes.js** file, we see that this route has the name of **authors.index** and that it points to the **resources/js/components/au-**

**thors/index.vue** file, so let's open this up:

```
export default {
  name: "index",
  data() {
    return {
      authors: [],
      authorToDelete: null,
    }
  },
  methods: {
    getAuthors(){
      jsonApi.findAll('author', {include: ['books']})
        .then(authors => {
          this.authors = authors.data;
        });
    },
    setToDelete(author){
      this.authorToDelete = author;
    },
    deleteAuthor(){
      jsonApi.destroy('author', this.authorToDelete.id)
        .then(response => {
          this.authorToDelete = null;
          this.getAuthors();
        });
    }
  },
  created() {
    this.getAuthors();
  }
}
```

Vue component files can get very long, especially if they contain both markup, javascript and styling, so we will only show the javascript part of this, since this is most important.



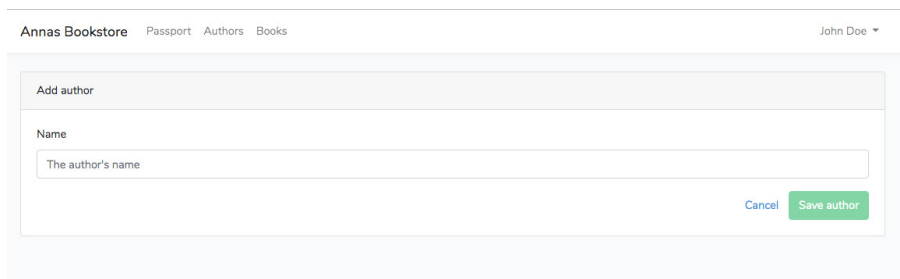
The list should be able to list authors. This is done through the **getAuthors** method, which is called as the component is being created.

Here, we are using **devour-client** to fetch all authors and it's actually done with just a few lines of code, saving the result in a property in our component.

This component also handles the deletion of an author, and again, you can see that only a few lines are needed. To be able to delete a resource, we use the **destroy** method on our **jsonApi** object that we created when bootstrapping our application. This method only takes the model name, and the **ID** of the author to be deleted. In our case, the request gets this **ID** from a property, since we present the user with a modal overlay that needs acceptance from the user, before we send the delete request.

Now, we could just remove the author from the array of authors and be done with it, but in order to sync up without database, we fetch all authors again.

If we click the **Add author** button, we will be redirected to the add author route, where we will be presented with a form like this:



The screenshot shows a web application interface for 'Annas Bookstore'. At the top, there is a navigation bar with links for 'Passport', 'Authors', and 'Books', and a user profile 'John Doe' with a dropdown arrow. Below the navigation bar is a modal form titled 'Add author'. The form has a section labeled 'Name' containing a text input field with the placeholder text 'The author's name'. At the bottom right of the form are two buttons: a 'Cancel' button and a green 'Save author' button.

Here, we should look in the **resources/js/components/authors/create.vue** file in order to see the code:

```

export default {
  name: "create",
  data() {
    return {
      name: '',
    }
  },
  computed: {
    validated() {
      return String(this.name).length > 0;
    }
  },
  methods: {
    save() {
      if(!this.validated){
        return;
      }

      jsonApi.create('author', {
        name: this.name
      }).then(response => {
        this.$router.replace({
          name: 'authors.index',
        });
      });
    }
  }
}

```

When the form is being submitted, we call the **save** method. Here, we check if the form is validated and if not, we call the **create** method on our **devour-client jsonApi** object that we created when bootstrapping the application. The cool thing about the **devour-client** is that it handles all of the serialization and deserialization, so we don't have to follow the structure of the JSON:API specification, when writing our applications. This means that we can just give the name attribute in an object and that's it. The **devour-client** will make sure that this gets the right structure and is being sent off to our API. Then, when a successful response is being sent back, we redirect the user to the index

page again. Speaking of which, let's go back to the index page and click on the name of an author, so that we end up at the Update author route, it should look something like this:

Annas Bookstore    Passport    Authors    Books    John Doe ▾

Add author

Name

Johan Schmidt

Cancel    Save author

To view the code, let's open **resources/js/components/authors/update.vue** file:

```
export default {
  name: "create",
  data() {
    return {
      author: null,
    }
  },
  computed: {
    validated() {
      return String(this.author.name).length > 0;
    }
  },
  methods: {
    getAuthor(){
      jsonApi.find('author', this.$route.params.id).then(
        author => {
          this.author = author.data;
        }
      );
    },
  },
}
```

```

    save() {
      if(!this.validated){
        return;
      }

      jsonApi.update('author', {
        id: this.author.id,
        name: this.author.name,
      }).then(response => {
        this.$router.replace({
          name: 'authors.index',
        });
      });
    },
    created() {
      this.getAuthor();
    }
  }
}

```

Here, we start out by fetching a single author through the **getAuthor** method.

To do this, we use the **find** method on our **devour-client** and use the id given in the Vue router parameter to fetch the right author from our API. We then assign the fetched author to an **author** property, which the user will edit in the form.

When the form is submitted, we call the **save** method where we test to see if everything is validated. Then, we call the **update** method on our **devour-client**, passing in the **ID** and **name**. Again, the serialization will take care of the correct structure: all we have to do is to wait for a response, and then redirect the user back to the index listing.

This is actually all there is for authors. You see that by leveraging the **devour-client**, we can do a lot with a few lines of code. Once our models are set up, we are free to do the calls we need.

## FINISHING UP

For books, things are mostly the same — especially on the index route. For the create and update routes, we also add a relationship to the create and update methods. So why don't we take a look at the create book route at first.

Up in the menu on in the browser, let's click on books and then click on the **Add book** button, so that you see the following interface:

The screenshot shows the 'Update book' form in the Annas Bookstore application. The form is titled 'Update book' and is located within a sidebar menu that includes 'Annas Bookstore', 'Passport', 'Authors', and 'Books'. The user 'John Doe' is logged in. The form contains the following fields:

- Title:** A text input field containing 'Mr. Cletus Armstrong V'.
- Description:** A text area containing 'Harum aperiam ipsam tempora nam ut aspernatur dolor.'
- Publication Year:** A text input field containing '1974'.
- Authors available:** A list of two authors: 'Jarvis Champlin III' and 'Brian Arms'.
- Authors selected:** A text input field containing 'Johan Schmidt'.

At the bottom right of the form, there are two buttons: 'Cancel' and 'Save book'.

Here, we have a couple more inputs than for authors, but the thing to note is the drag and drop functionality in the bottom, where you can assign the authors for the book.

The drag and drop functionality is done through the **vuedraggable** plugin that we mentioned earlier.

Let's look at the **resources/js/components/books/create.vue** file to see the code for this:

```

import draggable from 'vuedraggable'

export default {
  name: "create",
  components: {
    draggable,
  },
  data() {
    return {
      title: '',
      description: '',
      publication_year: '',
      authors: [],
      availableAuthors: [],
    }
  },
  computed: {
    validated() {
      return String(this.title).length > 0 &&
        String(this.description).length > 0 &&
        String(this.publication_year).length > 0 &&
        this.authors.length > 0;
    }
  },
  methods: {
    getAuthors() {
      jsonApi.findAll('author').then(authors => {
        this.availableAuthors = authors.data;
      });
    },
    save() {
      if (!this.validated) {
        return;
      }

      jsonApi.create('book', {
        title: this.title,
        description: this.description,
        publication_year: this.publication_year,
        authors: this.authors.map(item => {

```

```

        return {
          id: item.id,
        }
      )),
    }).then(response => {
      this.$router.replace({
        name: 'books.index',
      });
    });
  }
},
created() {
  this.getAuthors();
}
}

```

There's a bit more code here, but don't get confused by this. It's mostly because there are more fields in the form, which, of course, will result in a bit more properties needed.

When this component is being created, we call **getAuthors** which will fetch the authors from our API, which we will use to create relations from the book to be created and the selected authors. These authors will populate the list to the left in the user interface, since this list is getting its authors from the **availableAuthors** array. When we drag and drop an author to the selected authors list, this author is actually actually copied from the **availableAuthors** array over to the **authors** array.

When the form is submitted, we call the **save** method. Here, we check if validation is passing. All fields must be filled and at least one author be related before the validation is passing. Then, we call the **create** method on our **devour-clients jsonApi** object.

We do almost like we did with authors, but the thing to note here is how we handle the relationship to authors. Here, it's enough with an array of **IDs**. No

need to recreate the structure of the JSON:API specification — we can just work with things in a simple request object. To create the array of **IDs**, we take the **authors** array and run the **map** method on this. This does exactly like maps of Laravel Collections, javascript developers are just lucky enough to have these things included by standard in their language.

We map over the authors and take the **IDs**, and wait for a successful response, and finally redirect the user back to the books index list.

Let's go back to the books index list, and then select a book to edit. This should take us to the update route, which should look something like this:

The screenshot shows a web application interface for updating a book. At the top, there's a navigation bar with 'Laravel', 'Passport', 'Authors', and 'Books' links, and a user profile 'John Doe'. The main content area is titled 'Update book'. It contains several form fields: 'Title' with the value 'Emilio Bogisich DVM', 'Description' with the value 'Culpa ipsum quo dignissimos doloribus voluptas.', and 'Publication Year' with the value '1974'. Below these is an 'Authors available' section with a list of names: 'Eulalia Wolff', 'Miss Sarai Ruecker Jr.', 'Earline McKenzie I', and 'Prof. Demond Greenholt'. To the right of this list is an 'Authors selected' section with the name 'Janae Mosciski II'. At the bottom right of the form are two buttons: 'Cancel' and 'Save book'.

Let's look at the file for this route, so let's open **resources/js/components/-books/update.vue**:



```

import draggable from 'vuedraggable'

export default {
  name: "update",
  components: {
    draggable,
  },
  data() {
    return {
      book: null,
      availableAuthors: [],
    }
  },
  computed: {
    validated() {
      return String(this.book.title).length > 0 &&
        String(this.book.description).length > 0 &&
        String(this.book.publication_year).length > 0 &&
        this.book.authors.length > 0;
    }
  },
  methods: {
    getBook() {
      jsonApi.find('book', this.$route.params.id, {include: ['
        authors']}).then(book => {
        this.book = book.data;
        this.getAuthors();
      });
    },
    getAuthors() {
      jsonApi.findAll('author').then(authors => {
        this.availableAuthors = _.differenceWith(authors.
          data, this.book.authors, (author, otherAuthor)
            => {
              return author.id === otherAuthor.id
            });
      });
    },
    save() {
      if (!this.validated) {

```

```

        return;
    }

    jsonApi.update('book', {
      id: this.book.id,
      title: this.book.title,
      description: this.book.description,
      publication_year: this.book.publication_year,
      authors: this.book.authors.map(item => {
        return {
          id: item.id,
        }
      })
    }).then(response => {
      this.$router.replace({
        name: 'books.index',
      });
    });
  });
},
created() {
  this.getBook();
}
}

```

Here, we fetch the book when the component is being created. Just like with authors, we get the **ID** of the book through the route parameters of our Vue router and use this to fetch our book. When we get the response back for the book, we fetch all authors, but instead of saving these in the **availableAuthors** array right away, we filter them against the authors already related to the book, so that we can repeat the drag and drop functionality we have when creating books.

When the form is submitted, we hit the **save** method. Here, we use the same approach as when creating books, where we map over the selected authors and find the **IDs**, which are used for the association.

Again, **devour-client** makes it really easy for us, and it would have been almost the same for many of the other client implementations we could have selected.

The reason is, of course, the strict protocols and conventions of the JSON:API specification, which enables us to predict how the data structure of our API will be, whether it's complex relationships or just simple resource identifier objects.

We hope you learn something about how to leverage the JSON:API with Vue and hope you will try to build the comments section for yourself.

If you are really into the frontend part of this project, you could also use the role attribute on users to have different user interfaces for admins and users.

## Thank you

We would like to thank you for reading our book. We had a great time writing it and sharing it with you. If you find the book helpful, we hope you'll leave us a review on Gumroad or Goodreads and tell us your thoughts. This is our first publication, and we're eager to learn and improve with your help. Thank you again.

Writing APIs like the one in this book is only the tip of the iceberg, but with the knowledge you have now received, you'll be able to write a lot of great APIs that are easy to consume and easy to understand, with conventions that will never leave you in doubt of what to do next.

## About the Author

Christian and Thomas are the founders of Wacky Studio, which is a small design and development company, based in Denmark. Their field of expertise is mainly web-based, and their projects are developed in everything from Laravel, Vue, Go, HTML5, SASS to Unity3D.

Before they started Wacky Studio, they taught web design and development at a local technical college and now they've added this book to their extensive resumes.

Thomas, the coding wizard of the company, wrote most of the text. Christian, the design superhero, made it look like a topnotch book.

Sille, their intern, wrote this description of them knowing they'd never praise themselves (even though they should).

### **You can connect with me on:**

 <http://wackystudio.com>

 <https://twitter.com/wackystudiodk>

 <https://www.facebook.com/wackystudio>

### **Subscribe to my newsletter:**

 <https://wackystudio.com/en/contact>