

THE ULTIMATE GUIDE TO Building APIs & Single-Page Applications

WITH LARAVEL + VUE.JS + CAPACITOR



Build more with less code.
**Develop web and mobile apps
from the same codebase.**

ServerSide **UP**

Dan Pastori & Jay Rogers

This book is dedicated to all the self-learners out there.

“Formal education will make you a living;
self-education will make you a fortune.”

- Jim Rohn

Contents

Getting Started	1
Preface	2
Why build an API?	4
Prerequisites	7
What We Will Be Building	9
Why We Chose Laravel and NuxtJS	11
Configuring Laravel as an API	13
Configuring Your Development Environment	14
Installing Laravel 8.x	16
Setting Up Laravel to be an API	19
Preparing Your API for Automated Testing	25
Moving to the Frontend	27
Using NuxtJS for Web & Mobile	28
Installing and Configuring NuxtJS	29
Understanding NuxtJS' Structure	41
Abstracting Your API Calls Into Reusable Modules	46
Managing Events With the Event Bus	54
Saving Time With Mixins	57
Building Your First Layout	59
Adding Pages with NuxtJS	64
Using Laravel Sanctum for API Authentication	69
Building Secure Authentication API Endpoints	70
Implementing Cross-Origin Resource Sharing (CORS)	78
Laravel Sanctum vs. Laravel Passport	82
Installing and Configuring Laravel Sanctum	86

Implementing Authentication & Registration With Laravel Sanctum	93
Writing Authentication Tests	104
Set Up NuxtJS to Properly Authenticate With Your API	106
Configuring NuxtJS Auth Module + Laravel Sanctum	107
Building Authentication & Registration Components	115
Handling Guest Users With NuxtJS Middleware	137
Implementing an Email Validation System for New Registrations	140
On to "Fun"-ctionality	164
The Full-Stack Feature Approach	165
Using the Proper Methods for API Requests	166
Building a Feature From Start to Finish	169
Managing App Resources Through an API	173
Efficiently Building API Endpoints for Data Queries	174
Using Our API Endpoints With NuxtJS	214
Accessing Resources With Human-Readable API Endpoints	234
Uploading Files	243
Handling "Parent-Child" Relationships	267
Implementing Many-To-Many Relationships	286
Permissions, Validations, and Security	295
Permissions Overview	296
Preventing Unauthorized Access Using Middleware	306
Securing API Endpoints With Laravel Gates & Policies	312
Implementing Laravel's Custom Validation Rules	324
Securing Our Front-End With NuxtJS Middleware	329
Handling Unauthorized Actions on the Front-End	333
Displaying API Errors to Your Users	338
Build iOS & Android Apps with CapacitorJS	342
Installing and Configuring CapacitorJS	343
Server-Side Rendering vs. Single-Page Application	348

Speeding Up Builds With A Single Command	350
Token-Based Authentication with Laravel Sanctum	357
Using Native Phone Features In Your App	371
Deep-Linking: Open Your App From The Web	376
Implementing Social Logins	379
Installing and Configuring Laravel Socialite	380
Allowing logins from Facebook, Google, and Twitter	388
Securely Exchanging Tokens With an OAuth Provider	402
Opening Your API to Others	410
Configuring Laravel Sanctum for 3rd Party Access	411
Determining What API Endpoints Should Be Available to 3rd Parties	
414	
Create Routes to Manage Tokens	416
Rate Limiting Requests	424
NuxtJS Interface and Settings	431
Using A Personal Access Token	434
Scoping Access Tokens	435
Limiting Access	436
API Tips, Tricks, and Gotchas	438
RESTful Response Codes: How To Use Them	439
How Cross-Origin Resource Sharing (CORS) Works	445
Handling "Imperfect" RESTful Routes	451
Benefits of API Versioning	454
Tips & Tricks for Testing APIs	455
Configure Postman REST Client	473
Configure Insomnia REST Client	489
Securing Sensitive Data	501
Single-Page Application Tips, Tricks, and Gotchas	505
When to Use Vuex	506

The Truth About Spa Security.....	509
Continuing progress & Thank You.....	511
We couldn't have done this without you.....	512

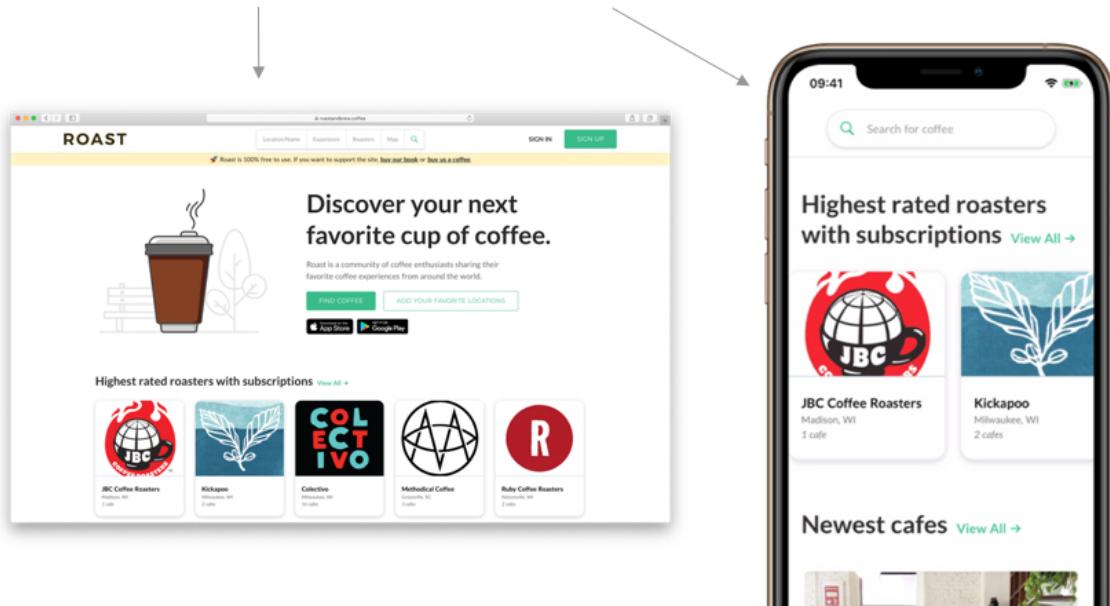
GETTING STARTED

Preface

After working with third-party APIs for over 10 years and developing multiple custom APIs, we decided to accumulate what we've learned into a helpful guide to show you how this process works. Our intent is to save you hours researching the little gotchas, error resolutions, and snags that come with developing your own API + SPA along the way. Seeing the troubles and hurdles that developers face when approaching developing an app this way inspired us to create this book.

We want to help make your development experience simple and cut through the crap so you can start writing beautiful, re-usable code as quickly as possible. So many books, university courses, tutorials show you how to do little pieces or snippets of code. This book, however, will take you through the entire process from start-to-finish.

Yup! This is the same code!



We aim to give you a holistic view of an entire, full-stack application. We'll show you how to design the architecture so it's re-usable, how to communicate between pieces of your application in a re-usable manner, but most of all... How to reuse the same code for whatever platform you wish to deploy to. We believe what's

missing from software-development education is the in depth approach to application structure. So we wrote a book, recorded some videos, and built an actual app to solve this need.

To start the book we will develop some foundations to constructing your API + SPA combination. Starting in two different repositories, they will feel like building two separate pillars. We will then solidify our foundation and connect everything together by implementing a secure authentication connection. Making our app functional and useful is right after. Opening up the data endpoints from the API and consuming these endpoints in our SPA seems tedious, but we will walk you through the entire process. Finally, we will take your beautifully engineered code, package it up, and prepare it to be deployed to iOS and Android. We will help you from installing Laravel and NuxtJS to having a fully functional application that can be packaged and distributed to any platform.

We hope you enjoy the adventure and our resources are valuable to help you share your idea with the rest of the world!

Why build an API?

This course will step you through the process of building an application centered around an API (Application Programming Interface).

First, let's answer "Why would you want to build an application around an API?". There's a ton of valid reasons, but let's talk about the three big ones.

Code Organization and Standardization

First, would be code organization and standardization. If you follow the practices we lay out in this course, you will be able to scale your application with ease, access it from a variety of devices such as mobile phones, desktop apps, web apps etc, and allow other developers to dive in and pick up where you left off. By following pre-defined standards of a RESTful application, other developers can pick up where you left off, 3rd party devs can integrate into your application, and most importantly you can fix bugs with ease.

Ease of Planning

Second, would be the ease of planning, testing, updating the application. Since all of your code will be going through proper RESTful ([Representational state transfer - Wikipedia](#)) API Endpoints, we know what HTTP method to call and test and what the proper response should be. We can not only document these end points easily, we can write testing code that helps ensure our code is stable and ready to be consumed.

Developing a RESTful API also takes some of the pain out of planning how communication between the multiple pieces of your application should operate. Whether you are creating data, retrieving data, updating or deleting data, the endpoint naming scheme and structure are already defined. You just have to create the piece of the puzzle which we will go through!

Allow Other Developers to Integrate

Third, is allowing other developers to integrate with your code. Most developers who have used APIs have used them in a way to enhance their own application. For example, if you wanted to grab the now playing song from Spotify, you could make an API call on behalf of the user to grab that information. The information is returned in JSON and you can use it in your application. Now we are building our application around our own API which is very similar on how you can consume other APIs. The coolest feature is when we are ready, we can allow OTHERS to consume the API without changing much code at all! Don't worry, it sounds crazy, but we will go through all of the processes to make this happen!

Why would you want other developers to integrate with your app? The goal is growth. Providing an API, users that use your app can use your data and processes and create their own unique tools, thus making your app more valuable and attracting more users. Even working in a large company and developing internal tools, providing an API is key to sharing and integrating data across departments, districts, etc.

Think About A Core

When designing your app, we are separating the front end user interface 100% from the backend code. In a standard call and response application, you could have a backend language, such as PHP, build and insert data into HTML before it's returned. Viewing application development from a Single Page Application point of view, you will be separating the user interface from the backend entirely.

Your interface will respond directly to the data loaded from the API. Don't worry if this sounds confusing now, we will be using [NuxtJS](#) to help with it and it's a breeze. For now, visualize the API as a gate keeper to your app where multiple sources can tie into, submit requests and access certain features. Fig. 1.1 helps with that visualization.

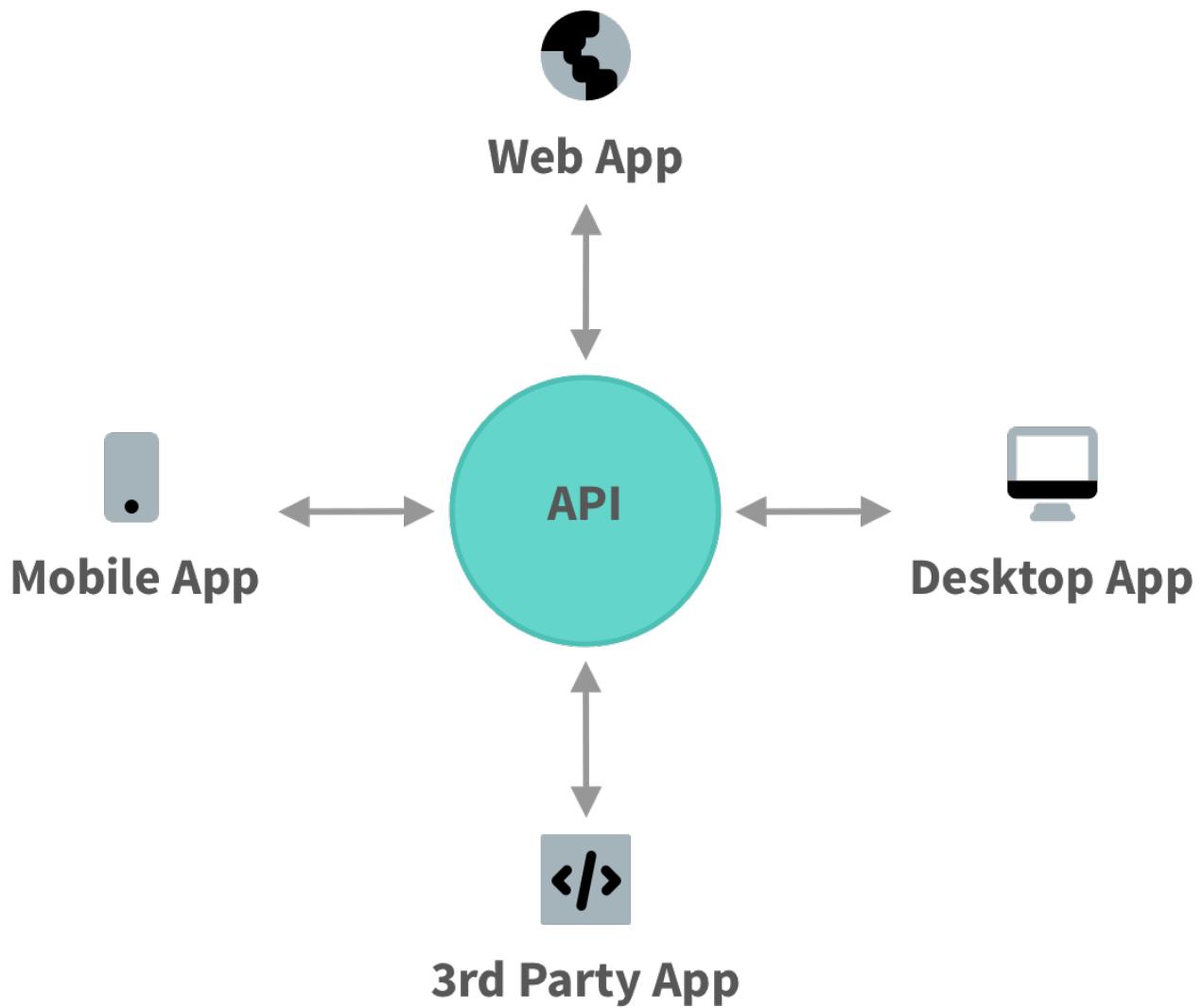


Fig 1.1

Prerequisites

Before jumping in, there are definitely a few things that we recommend to get the best learning experience from the book.

Knowledge of PHP and Javascript

This isn't a beginners level course on how to program, knowledge of PHP and Javascript is a must. If you are new to Laravel and VueJS that will be alright. We'll include links to documentation and other resources to help you along the way. If you have any questions, of course, feel free to reach out!

A Development Environment

Your development environment must be able to run [Laravel 8.x Requirements](#). You must also be able to run NPM to install NuxtJS and other tools. Other than that, you should be good to go!

Willingness to Learn

Cliché isn't it? It's true though. To learn you must be willing to learn. However, that goes both ways. We're open to comments and critiques as well. The beauty and art of software development is there is no 100% right way, there are just ways that are better than others. If you have any questions, comments, suggestions, just throw a message on [our community](#) and we'd be happy to have a discussion!

Begin With The End In Mind

Before we get too far deep into anything else, whether it's API related or development related, I wanted to introduce the concept of beginning a project with the end in mind.

If there is any concept I've taken to heart over the 15 years I've been developing software or learning new things throughout my life, it's been this. Whether you are a seasoned developer or someone new to software development, this is the

absolute most powerful starting point you can start from.

The concept is simple. Before learning something new, plan out what you want the end goal to be. Let's take that to software development. Before writing a single line of code, ask yourself what you want to build. How will it look? What value will it provide? What data will you need to have? What features do you want? This isn't only a great practice for project planning it's also motivating and fun!

Think about the boring software problems you may have solved in school, or if you haven't taken software development class, think of the 100 math problems you got assigned for homework. Yes, it helps expose you to concepts, but it really doesn't motivate you to continue in the subject. If you are trying to scratch your own itch, you are motivated to solve a problem that you have and you will learn the tools and processes to solve that issue efficiently and effectively and will feel PHENOMENAL when you figure it out. It's also motivating once you hit a road block. You will WANT to solve the problem because it will help you accomplish what you want.

In the next section, we will go through this process of laying out the app we will build. We will discuss what problems the app will solve, the goal behind it, why we want the application, etc. We will begin our development journey with the idea in mind of what we want at the end of the application. This doesn't mean we will have an exhaustive list of EVERYTHING the app will entail, but it will definitely give us a direction and help us stay focused on what we want to accomplished.

Oh yea, and the app we build will also be production ready! I'll save the rant on that for a later date, but you will be learning how to build an app that will actually be used, published on a production web server, pushed to the iOS and Android App stores, and able to be signed up for. I believe there is NO better way to learn then to experience an actual process and work through the mindset together. Any feature added to our application will be documented, explained and guided through. Let's begin!

What We Will Be Building

In this course we are building an actual, functioning, web and mobile application that you can use! This isn't a "Hello World" app or an application just to prove a point. It's an app that I want to use and hopefully others will too. The app is called [ROAST](#) and it helps coffee lovers find their next favorite cup of coffee. The users will be able to search coffee shops and roasters in their area, filter by brew methods, contribute their favorite companies and cafes, like coffee shops and a whole bunch more! And yes, the source code for web and mobile applications are the exact same!

We will be using modern platforms such as NuxtJS and Laravel 8.x. These platforms are well supported and make the development process a breeze. This book will dive in depth on authentication, development scoped from an API + SPA perspective, mobile compilation, and a bunch of tips, tricks & techniques that help you benefit from building an API Driven Application.

Can I Use the App?

Of course! If you really love artisan coffee, you will actually really love to use this app to find your next favorite cup of coffee. Even if you hate coffee, but just want to use the app, go for it! The app is open sourced and 100% functional. As we deploy new features we will expand the book to cover these features. Hopefully you will find your next favorite cup of coffee on ROAST. If you already have a favorite coffee shop, make sure it's added to the app.

Can I Contribute?

Contributions to the source code are not only accepted, but encouraged! The entire source code is available for the people who purchased our complete package. You can view the entire source code and submit pull requests through Gitlab.

The more we can learn together the better off we will be. Any questions and

issues can also be raised on [our community](#). There will be 2 separate repositories to show the separation of functionality. One for the Laravel API, and one for the NuxtJS front-end.

Feel free to make any PRs or submit ideas that you'd like to see in the application.

Why We Chose Laravel and NuxtJS

Choosing the right tool for the job is extremely important. Both Laravel and NuxtJS are most certainly the right choice. They are both beautifully documented, exceptionally maintained, and best of all have amazing communities.

Why Laravel?

So why did we choose Laravel for this book? Why not Ruby on Rails, Django, or any of the other PHP frameworks? Besides my familiarity with Laravel, the framework is absolutely amazing. It's secure, well documented, open sourced, beautifully functional and allows us to go from concept to functionality in minimal amount of time. Oh, and the ecosystem and community is amazing!

The documentation will help in a variety of scenarios, there's [Laracasts](#) which has so many useful videos (some free) and a wonderful community. There's a wide variety of official packages all with beautiful documentation and community support. There's also a Discord channel to ask help. Most importantly, it's widely supported and modern. Laravel brought PHP back from the dead and pushed the language's direction making it super fast, modern, and scalable.

The other reason we choose Laravel is it's easy for other developers to pick up on. With the ecosystem, documentation, and help available, if you want to contribute or want to hand off to another dev, it's a breeze. They can look at your code and pick up right where you left off. Gone are the days of trying to decipher a custom written code base!

Being open-sourced, Laravel is also always up-to-date with the latest security and bug fixes. In this book, we will be using the 8.x release. Even though this isn't LTS (Long-term support), it still comes with the most up-to-date security and bug fixes.

Why NuxtJS?

First of all, NuxtJS is our frontend framework used to run our Single Page Application (SPA). If you followed along with the ServerSideUp series, we used straight VueJS for our frontend. Personally, I love VueJS and think it's by far the best front end framework available.

So why are we using NuxtJS? Well, NuxtJS IS VueJS! Just built with a larger scale in mind and around the entire VueJS ecosystem (Vue Router, VueJS, Vuex). And it also has a beautiful structure that enables the organization of components very similar to how I'm used to.

We chose NuxtJS for our projects because it's actively maintained, clean, has beautiful documentation and written in VueJS. They have a wonderful ecosystem of first-class tools that take the pain out of a lot of difficult development, such as authentication modules, data stores, etc. The benefits of using a maintainable open-source framework, similar to Laravel, is that you can hand off your code to another dev or onboard another dev with ease! Everything is structured in a way that is friendly and easy to use!

In this book, we will cover all the fun details to get your NuxtJS app installed and running a Universal Application and a Single Page Application that will connect to our API. When you have this set up and configured, you can deploy to the hosting provider of your choice such as Netlify or your own custom rolled solution.

CONFIGURING LARAVEL AS AN API

Configuring Your Development Environment

Before I show you how I do it, I'd like to express, that this step is completely up to you! However, I find that this is the best way to organize your app so everything is completely separated and available to communicate. This is the benefit of doing an API Driven Application, the separation of concern between the frontend and backend are definitive.

Make API & Frontend Directories

Whenever I begin API Driven Development I like to have my folder structure set up and ready to rock and roll. Having it this way as a nice place to organize your code and I like to think of it like each directory is a bucket.

When I need to put code somewhere, I put it in that bucket. Personally, I think that it takes the extra thought out of where to put files when I'm in the developing zone. Both Laravel and NuxtJS have beautiful directories (buckets) set up on install and we will be doing more of the "bucket" approach as we get farther into each platform. However, this is a level above that, this is the separation of concerns between API and SPA

When doing API Driven Development, we need a place to put both the Laravel install and the NuxtJS install. For me, I usually do a simple folder structure like this:

```
/ {app-name}  
  /frontend  
  /api
```

In the `/frontend` directory I place my NuxtJS install and in the `/api` directory I place my Laravel install.

Since we are doing Roast and Brew, my directory structure looks like this:

```
/roastandbrew
  /frontend
  /api
```

We are also going to be building a mobile application using the same codebase (one of the main points of building an app this way). If you think about it though, the mobile app should be just our front end code so really we won't need another directory. The code for our mobile app will reside in `/frontend`. Don't worry, we will dive in to this when we install CapacitorJS and I'll show you how to integrate mobile specific features into your front end.

Installing Laravel 8.x

Let's start with the base layer. We will be using Laravel exclusively for our API and server side platform.

Laravel is beautiful PHP as a framework and provides all of the tools necessary to make a bullet proof API. I like to think of this layer to be the equivalent to the concrete foundation on which everything else is built upon. From this layer we will be have a way to transport data through our API to a variety of platforms such as web, mobile, and desktop (if needed).

After we get a Laravel install up and running on our development machine, we will make a few customizations to our install that prepare it for future packages (such as Passport, Socialite, and Sanctum) as well as remove some of the unnecessary scaffolding used for a monolithic approach to building an app (default views, resources, etc). Let's begin!

Step 1: Install Laravel

By far the best instructions for installing Laravel is to [check out their docs](#).

As you can see, there are multiple ways to install Laravel and the docs do an amazing job of covering everything in depth. However, my preferred method is to use the `composer create-project` command.

On your web server or development environment, open a terminal and run the following command. For us, this would be in the `/roastandbrew` directory and our `{directory_to_install_to}` would be `api` :

```
composer create-project --prefer-dist laravel/laravel  
{directory_to_install_to}
```

After the script has run, make sure composer finished without any errors. In their Laravel's documentation, there are a few alternatives to this method if they fit your need. You can even [grab it from their GitHub](#).

If you DO pull it from GitHub, run `composer install`, make sure you copy the `/.env.example` file to `/.env` and run:

```
php artisan key:generate
```

This command is usually run after Laravel is installed, but if you pull from the repo, it won't be and that will cause problems!

Step 2: Configure Basic .env Settings

Before we move to the next part of the setup, make sure you have your `.env` file set up correctly. Specifically your database connection parameters. Your `.env` file is located in the root level directory of your Laravel install.

```
APP_NAME=RoastAndBrew
APP_ENV=local
APP_KEY=YOUR_SECRET_KEY
APP_DEBUG=true
APP_URL=https://roastandbrew.521.test

LOG_CHANNEL=stack

DB_CONNECTION=mysql
DB_HOST=YOUR_DB_HOST
DB_PORT=3306
DB_DATABASE=YOUR_DB_NAME
DB_USERNAME=YOUR_DB_USERNAME
DB_PASSWORD=YOUR_DB_PASSWORD

BROADCAST_DRIVER=log
CACHE_DRIVER=file
QUEUE_CONNECTION=sync
SESSION_DRIVER=file
SESSION_LIFETIME=120
```

```
REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_MAILER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
MAIL_FROM_ADDRESS=null
MAIL_FROM_NAME="YOUR_APP_NAME"

AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
AWS_DEFAULT_REGION=us-east-1
AWS_BUCKET=

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
PUSHER_APP_CLUSTER=mt1

MIX_PUSHER_APP_KEY="PUSHER_APP_KEY"
MIX_PUSHER_APP_CLUSTER="PUSHER_APP_CLUSTER"
```

You might see a `.env.example` by default in your directory. Remove the `.example` and configure your new `.env` file.

In the next part, we will clean up the Laravel install and ensure that we don't have anything left over that we don't need and make room to build out our API.

Setting Up Laravel to be an API

Now it's time to configure our fresh new Laravel install! When doing API Driven Development, we can remove a few of the scaffolding directories that Laravel provides by default and we need to set some other pieces of the install such as linking the public storage directory.

Step 1: Remove Frontend Files

If you fired up the new Laravel install, you might have seen a beautiful and clean welcome screen that Laravel displays by default. We are going to get rid of it. This is strictly our API, so we don't need any blade or SASS files. All of this front end stuff will be in our NuxtJS front end.

So navigate to the `/resources` directory in the root of your Laravel install and delete the following folders:

```
/sass  
/views
```

We did NOT remove the `/js` or the `/lang/en` directories. If we need to implement Laravel Echo, we will need some JS support and for languages, we will need to return language information on API requests.

Step 2: Link Public Storage Directory

Next is a simple task. Linking the public storage directory. This allows us to have a directory to store information which can either be on S3, your local file system, or wherever you choose to add your files. To do this, run the following command on your webserver:

```
php artisan storage:link
```

Now your public storage directory is linked correctly. You are now housing your

storage OUTSIDE of the `/public` directory so you can choose who has access to what.

Step 3: Remove Default Web Routes

This one is pretty simple, I just remove the default Web route found in `/routes/web.php`. We won't need it, so let's not make it publicly available!

Your `/routes/web.php` file should look like:

```
<?php  
    //No routes in here!
```

Nice and clean!

Step 4: Add Directories for Controllers

There are a few directories that we can add by default. This is nice to set up right off the bat so we have a "bucket" to place files in when we are ready. The directory structure will also separate functionality via concern.

We will need to create the `app\Http\Controllers\API` directory. This directory will house all of the API controllers that respond to API routes. It will allow for a nice clean namespace for all of our API components.

Step 5: Let Laravel Know Where Your User Model Is

Since we updated the `User.php` model to be in the `app\Models` directory, we have to let Laravel know where it's located for authentication purposes.

To do that, open up the `config/auth.php` file and scroll down to the `providers` array. What we will do here is update the `users => model` key to be `App\Models\User::class`:

```
'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\Models\User::class,
    ],
]
```

We also need to adjust our database factory to account for the change. This will come in handy when testing our application so we can create testing users. To do this, open up the `database\factories\UserFactory.php` and adjust the namespace for the user to be:

```
use App\Models\User;
```

That should be all of the places we need to update for the user model.

Step 6: Run Initial Migrations

Laravel ships with initial migrations for managing users and password resets. We will want our database set up for this information. To do that, simply open the terminal on your webserver and type the following:

```
php artisan migrate
```

If all goes well, you will get a `Migration Successful` message. If not, ensure your database connection parameters are configured correctly.

Step 7: Optional - Create Services Directory

I personally love dividing complex logic into services. As lovely as it would be to have an app that just used eloquent models and everything mapped seamlessly, I doubt it exists. Now there are many ways to decouple complex code from controllers, but services are my favorite. Essentially, if you are creating a resource or processing data, you'd abstract the long process into a class dedicated to

handling the process. This leaves you with smaller controllers and re-usable code.

Why are small controllers important? They should simply route functionality from an endpoint to an object. From there you can process the information through a series of smaller methods and abstract functionality into multiple re-usable pieces of code. Having private methods within your controllers class makes it hard to re-use elsewhere. For example, the method we use to upload files is re-used in multiple places. In our service, we include this object and it helps us maintain a single responsibility for each piece of our app.

To set up our "bucket" for this, I recommend creating an `app/Services` directory to get started. Within that directory, we will create directories to mirror our models and handle complex business logic before persisting to the database.

Step 8: Clean up API Routes

Laravel 8.x ships with a specified routes file `routes/api.php` that handles all API requests. This makes for really clean development since you can separate web routes from api routes from console commands when developing your Laravel application.

By default, there is an API route in the `/routes/api.php` file that returns the existing user. Let's remove that route and make the `/routes/api.php` file should look like:

```
<?php
```

While we are in that file, let's create a wrapper for all of our initial API routes that prefixes them with `/v1`. To do that, add the following to your file:

```
<?php
```

```
Route::group(['prefix' => 'v1'], function(){
});
```

The `prefix => v1` means that every API request will append a `v1` to the route name. So let's say we want all users in the application and we have a `GET` request to a `users` endpoint. The route would be `GET: /api/v1/users`. Laravel automatically adds the `api` to any route in the `routes/api.php`.

The reason we prefix ALL of the routes with `v1` is there will undoubtedly be changes to our API in the future. As we deploy our API it's always good to have versioning in mind. This way, we can open our API to third party developers, they can begin using our endpoints, and when we are ready to roll out a new version of our API, we can give time for the third party applications to transition off of the first version. Don't worry, we will expand upon this later.

We will also be adding some middleware to these routes for security. We will be doing that on a per-route basis and we will be applying that in the route controllers so we don't clutter up our `/routes/api.php` file. Now we have both of the web and api routes files cleaned up and ready to go!

Step 9: Optional - Initialize Your GIT Repo

I hope you are using a code repository and honestly, I hope it's GIT. Laravel comes configured for a GIT repo right away with proper `.gitignore` files set up so all you have to do, is initialize it. If you haven't messed around and configured GIT, I'd highly recommend setting it up. For more information on GIT and all of its features from start to finish, I'd recommend reading [GIT SCM](#).

To initialize GIT, open up a terminal and navigate to the root of your API directory and type the following command:

```
git init
```

That's it! GIT is now initialized. If you are using VSCode, you should see a bunch of green files in the folder navigator and about 80 untracked files (a few more or less

depending on your environment). If you have like 5000 files, make sure your `.gitignore` is present and set up correctly. You don't want to submit unnecessary information to your repo, like vendor files.

After initializing GIT, I like to make an initial commit, just so I have a clean slate to go back to. Run the following commands in your terminal:

```
git add .
git commit -m "Initial Commit"
```

The first command adds all of the files to your staged files and the second command commits them all to your GIT repo with a message of "Initial Commit" which will be easy to find in your repo if you ever need to go back to it.

There we go! We have a basic Laravel install set up and ready to go! Next up, we will configure our testing environment. This is so we can perform beautiful automated tests on our code before deployment. This is extremely important for deploying quality code!

Preparing Your API for Automated Testing

A huge part of ensuring the app and API are bulletproof is to write tests for all of your functions. When we begin to write features for our app, we will be writing our feature and providing tests. For each feature, the tests will be explained at the end if there is anything unique we should discuss. To ensure you can follow along, let's configure our testing environment.

Laravel has some beautiful integration with PHP Unit that will make our lives a ton easier.

Set Up A Testing Database

On your database server, set up an empty database. This will be used to create the tables when you run tests and check for proper persistence. I always name it with the following naming conventions of `{app_name}_test`. Once that's created, we are on to making our environment file.

Make a Testing .env File

This file should be exactly the same as your current `.env` file with one exception. The `DB_DATABASE` field should be the name you made in the first step `{app_name}_test`. Once you make this file, save it in the root directory of the app under the name `.env.testing` next to your current `.env` file.

If you have configured GIT, you will want to add this `.env.testing` file to the `.gitignore` in the root of your `/api` directory. This file contains sensitive information that should not be stored in a GIT repo!

Create API Tests Directory and Add Suite

I like creating a directory to store all my API tests at `/tests/API`. This allows me to dump all of my API tests into an organized directory. This is huge because we

are building an API Driven Application and makes sense to have a specific directory for all of our API tests.

Once that directory has been added, open up your `/phpunit.xml` file and add the following test suite inside the `<testsuites>` tag:

```
<testsuite name="API">
    <directory suffix="Test.php">./tests/API</directory>
</testsuite>
```

We are now good to go! From here on out, we will be doing API Integration tests for our features. Each feature that we walk through in this course will be "app agnostic" meaning the tutorial will be more focused on the idea behind the implementation rather than the feature itself so it's easier to replicate in your own app.

Remove Example Test from Feature Folder

Since we removed the web facing routes, let's remove the `/tests/Feature/ExampleTest.php` file since that will definitely be a failure considering it checks to see if the web route returns a `200` response.

If you want a perfect primer for TDD, first run:

```
./vendor/bin/phpunit
```

Then delete the `ExampleTest.php` file and run the command again. The first time it will fail and then you delete the file and it will be green, meaning the tests pass.

As we write tests, we will focus mainly on how to do this in an API Driven fashion. That means some of the smaller unit tests we won't go into detail on and we won't discuss all of the million different things to test. However, we will go through the ways to test API Endpoints. If you purchased the complete package and have access to the source code, you can always browse the tests directory for a variety of different examples of tests.

Moving to the Frontend

Congratulations! At this point, you have Laravel set up and ready to build your API. Now we are ready to start building your SPA! We've just gotten far enough to begin setting up the frontend and getting every piece communicating and ready to rock!

The next section, we will install NuxtJS and configure the authentication system so we can log in to our API. Fair warning, authentication can be one of the most complex features when designing an app this way. However, it is also the most important. After authentication, we will work on adding a few features that may be difficult from the perspective of an API. Then it comes time to compile our app for mobile!

Let's install NuxtJS!

USING NUXTJS FOR WEB & MOBILE

Installing and Configuring NuxtJS

Like most of our beginning tutorials, the [official documentation does the best job walking you through the process](#). However, like we did with Laravel, I'll cut through the other options and explain exactly what we need for our NuxtJS powered front end!

This is one of the main changes between the Server Side Up tutorial and this book. I believe using NuxtJS is the best choice if you plan on making a VueJS Single Page Application. It's community supported, well tested and is under active development. It also allows you to easily do SSR and SPA out of the box which is super important as we deploy to other platforms. For more information on why we switched to NuxtJS, read the section Why NuxtJS?

Step 1: Open a terminal

Like Laravel, NuxtJS is installed from the terminal. They have a beautiful installer that runs through `npx` and guides you through the process. This saves an INSANE amount of time compared to setting all of this up your self.

In the terminal, navigate to your `/{app-name}` (for our app, this would be `/roastandbrew`) directory. You should see the `/api` directory as a sub directory. In the root of your `/{app-name}` run the following command:

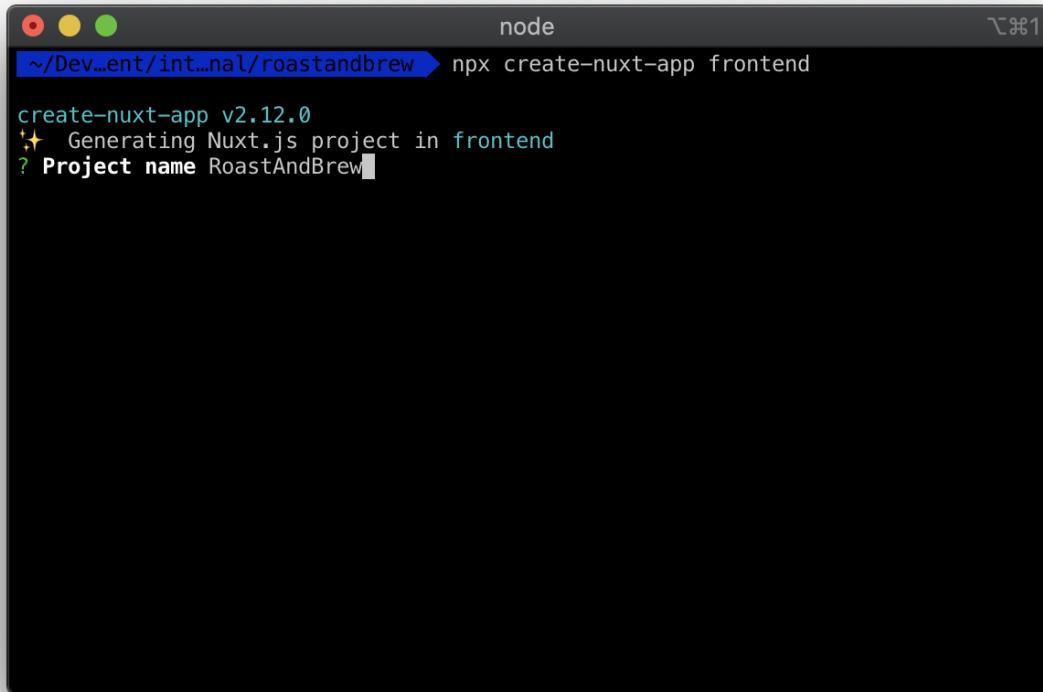
```
npx create-nuxt-app frontend
```

We create the NuxtJS app in the `frontend` directory since it's scoped within the `/{app-name}` directory. This gives a nice separation from our API and allows us to have an entire directory for our SPA.

After you run the command, NPM will install a bunch of packages and you will be prompted to set up your NuxtJS App.

Step 2: Set Name

The first prompt is to set up the name of the project. I call it `RoastAndBrew` with no spaces.

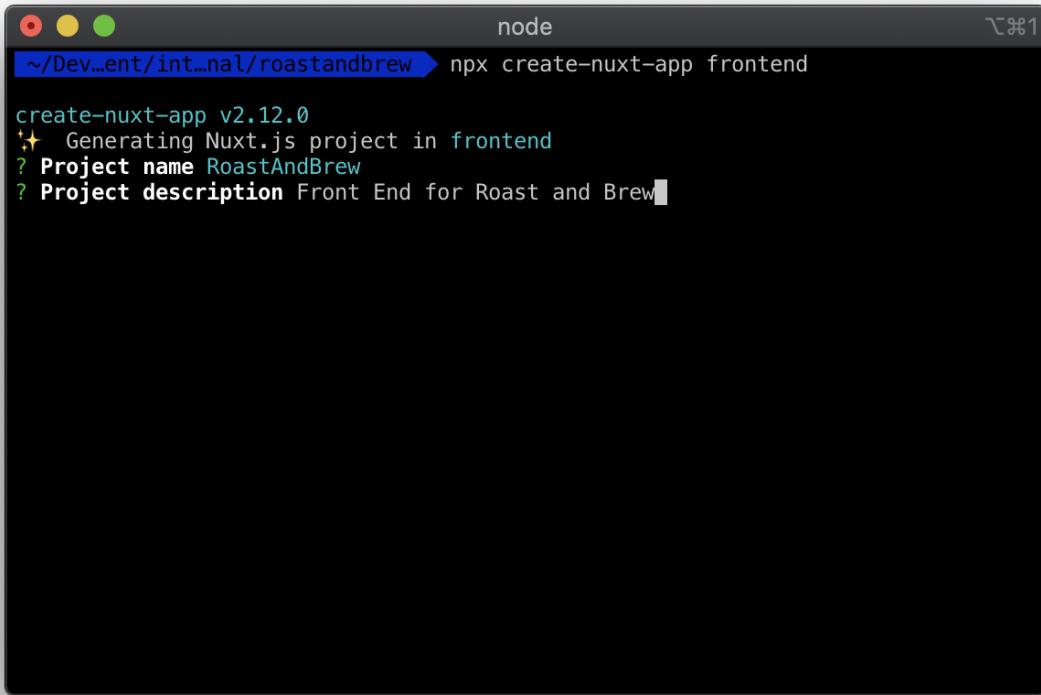


```
node
~/Dev/int...nal/roastandbrew ➔ npx create-nuxt-app frontend
create-nuxt-app v2.12.0
Generating Nuxt.js project in frontend
? Project name RoastAndBrew
```

A screenshot of a macOS terminal window titled "node". The window shows the command "npx create-nuxt-app frontend" being run. The output indicates that "create-nuxt-app v2.12.0" is being used to generate a Nuxt.js project named "frontend". A question mark prompt follows, asking for the "Project name", which is currently typed as "RoastAndBrew". The terminal has a dark background with light-colored text and icons.

Step 3: Set Project Description

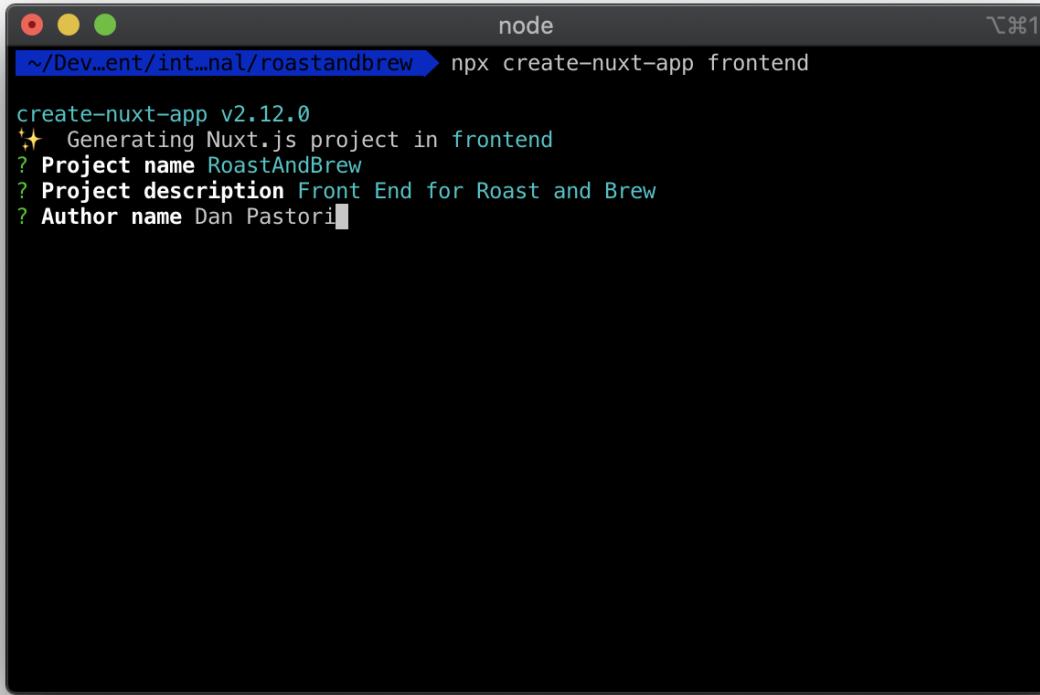
This can be a simple description of what your app is. For our sake, I just typed in "Front end for Roast and Brew"



```
node
~/Dev...ent/int...nal/roastandbrew▶ npx create-nuxt-app frontend
create-nuxt-app v2.12.0
✨ Generating Nuxt.js project in frontend
? Project name RoastAndBrew
? Project description Front End for Roast and Brew
```

Step 4: Set Author

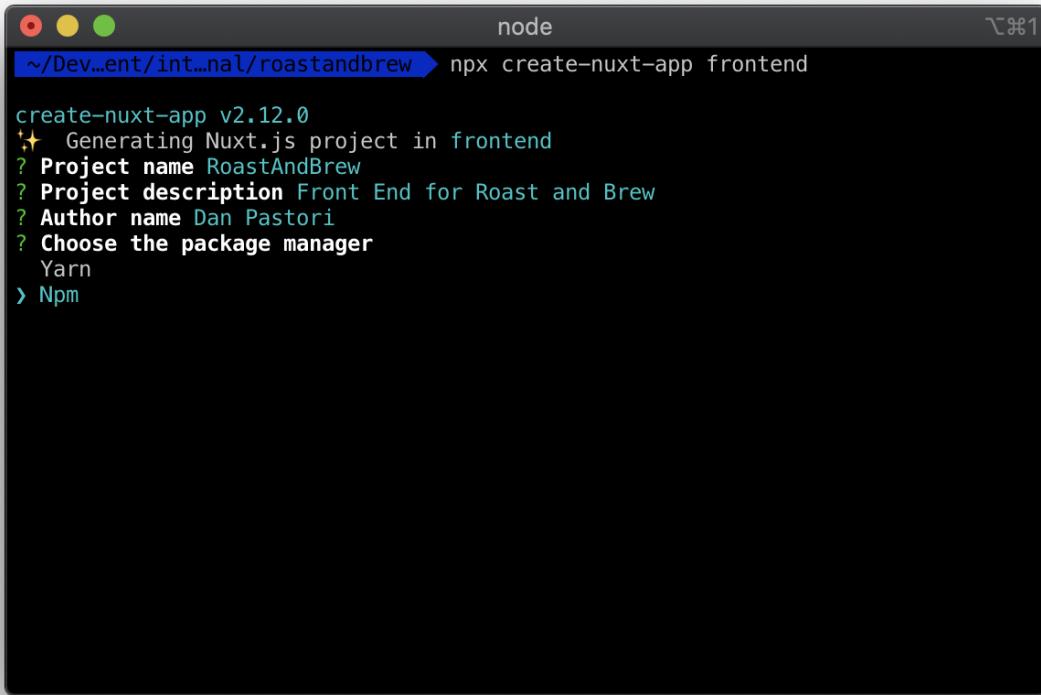
This one is straight forward, it should be your name or the name of your company. I typed in my name, "Dan Pastori".



```
node
~/Dev...ent/int...nal/roastandbrew ➔ npx create-nuxt-app frontend
create-nuxt-app v2.12.0
✨ Generating Nuxt.js project in frontend
? Project name RoastAndBrew
? Project description Front End for Roast and Brew
? Author name Dan Pastori
```

Step 5: Choose Your Package Manager

You have a choice between Yarn and NPM. For this app, we are using NPM but if you want to use Yarn, it won't make any difference. We have a deployment system with NPM so we wanted to make this app easily deployable, that's why we use NPM.

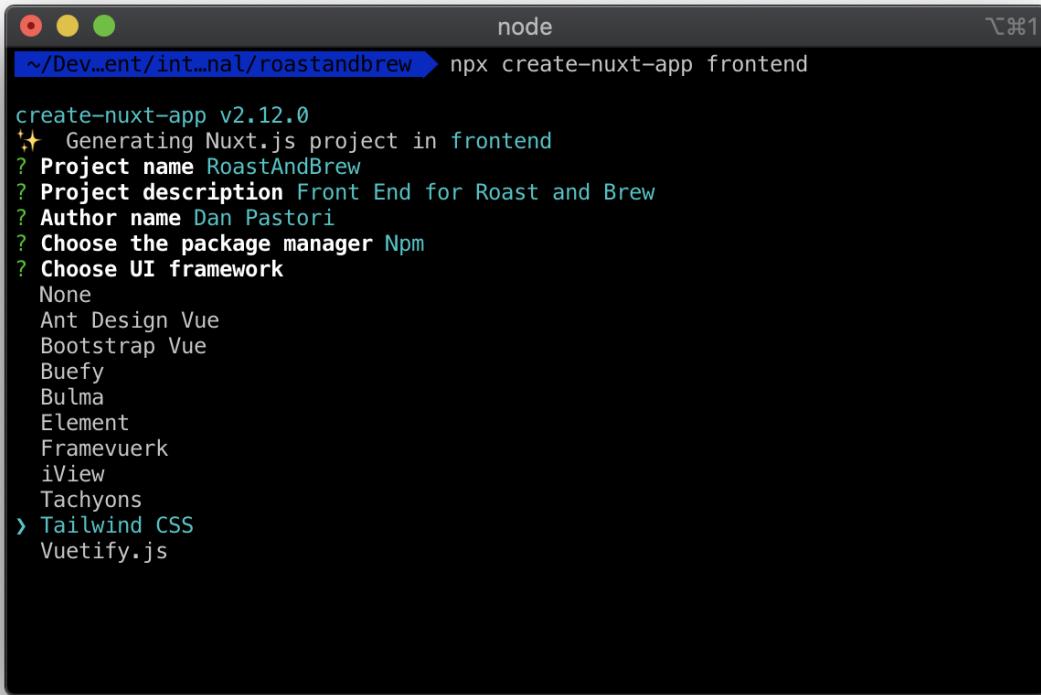


```
node
~/Dev...ent/int...nal/roastandbrew ➤ npx create-nuxt-app frontend

create-nuxt-app v2.12.0
✨ Generating Nuxt.js project in frontend
? Project name RoastAndBrew
? Project description Front End for Roast and Brew
? Author name Dan Pastori
? Choose the package manager
  Yarn
> Npm
```

Step 6: Choose your UI Framework

This is DEFINITELY up to you and what you want to design your app with. It will make really no functional difference through this tutorial. I personally LOVE [Tailwind CSS](#) and if you haven't tried it, I highly recommend giving it a try. It makes customizing the look and feel of your application a breeze and having it auto installed with NuxtJS is wonderful! That's what I selected for this project.

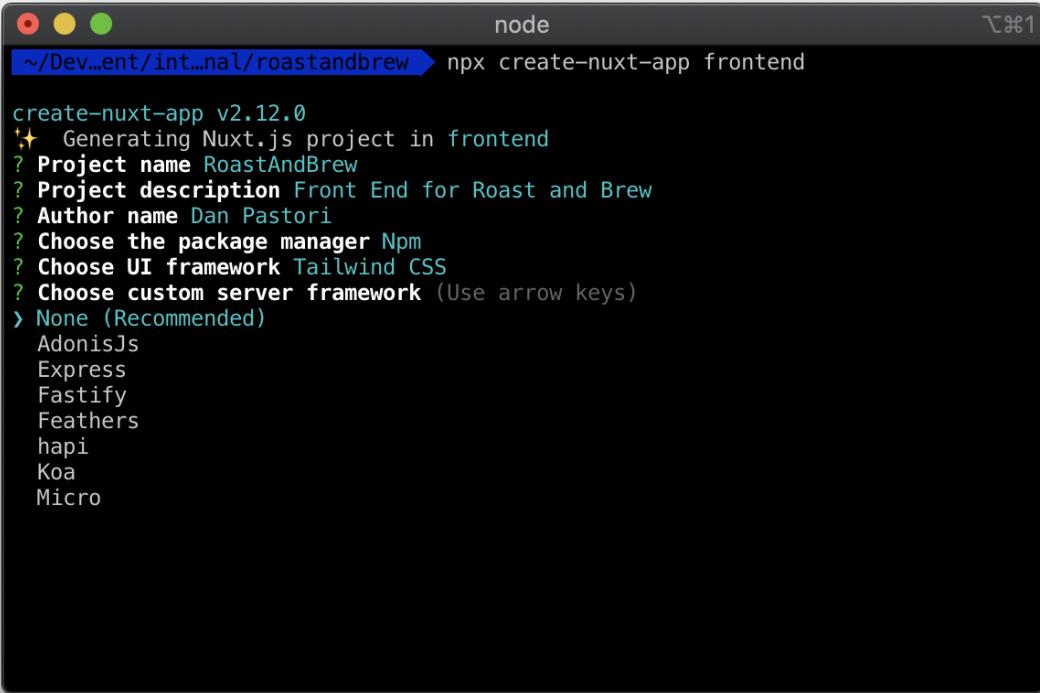


```
node
~/Dev...ent/int...nal/roastandbrew▶ npx create-nuxt-app frontend

create-nuxt-app v2.12.0
✨ Generating Nuxt.js project in frontend
? Project name RoastAndBrew
? Project description Front End for Roast and Brew
? Author name Dan Pastori
? Choose the package manager Npm
? Choose UI framework
  None
  Ant Design Vue
  Bootstrap Vue
  Buefy
  Bulma
  Element
  Framevuerk
  iView
  Tachyons
> Tailwind CSS
  Vuetify.js
```

Step 7: Choose your Server Framework

We've already got this set up! We are using Laravel for our API, so definitely select **None** for this step.



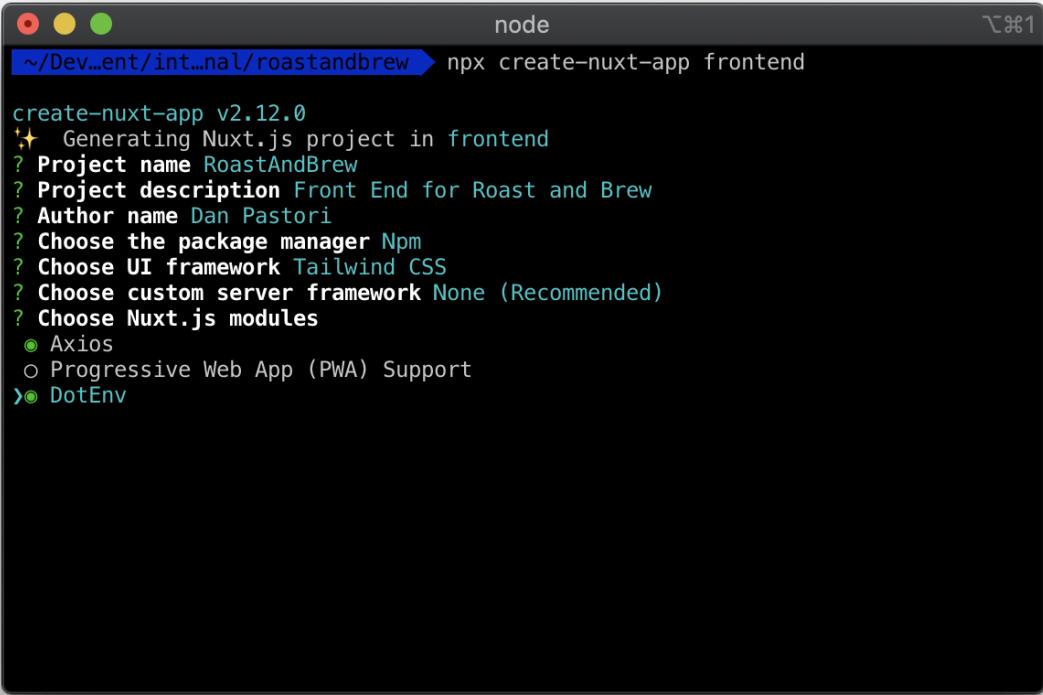
```
node
~/Dev...ent/int...nal/roastandbrew ➔ npx create-nuxt-app frontend

create-nuxt-app v2.12.0
✨ Generating Nuxt.js project in frontend
? Project name RoastAndBrew
? Project description Front End for Roast and Brew
? Author name Dan Pastori
? Choose the package manager Npm
? Choose UI framework Tailwind CSS
? Choose custom server framework (Use arrow keys)
❯ None (Recommended)
  AdonisJs
  Express
  Fastify
  Feathers
  hapi
  Koa
  Micro
```

Step 8: Choose Nuxt Modules to Install

Well, right off the bat we will want `Axios` and `DotEnv`. Axios will make calls to our API a breeze and we can configure our default settings for all headers, interceptors, everything so building our API Driven Application is a much more seamless process.

Next, select `DotEnv`. `DotEnv` is a way to configure environment variables for NuxtJS using a `.env` file. This will definitely come in handy when setting up domains for API calls from test to production environments and storing a variety of environment specific information.



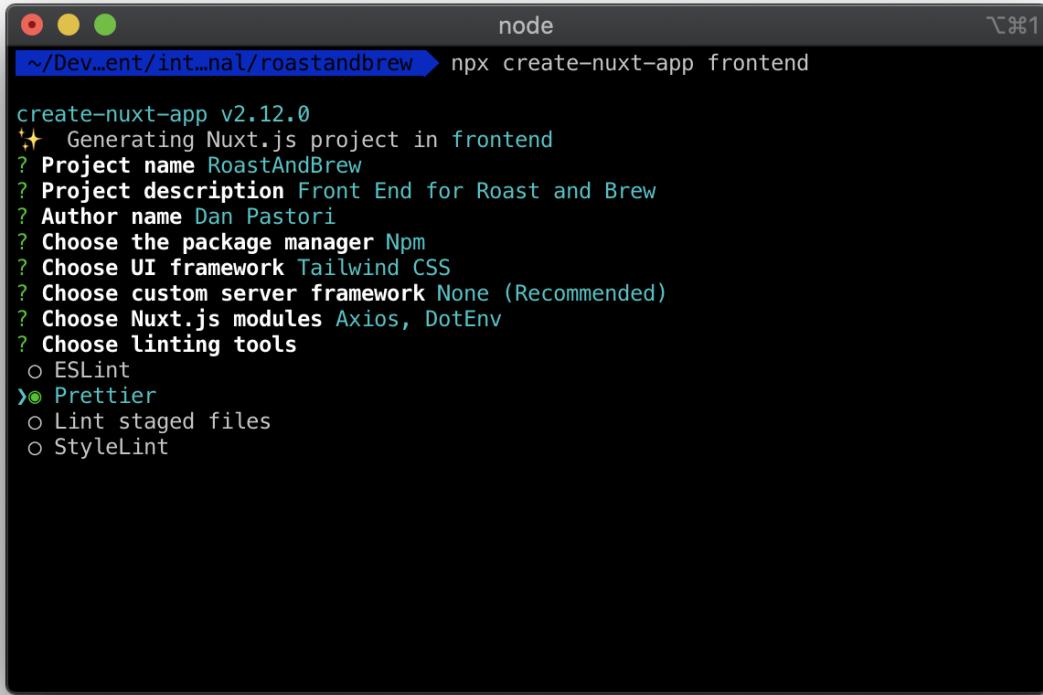
```
node
~/Dev...ent/int...nal/roastandbrew ➤ npx create-nuxt-app frontend

create-nuxt-app v2.12.0
✨ Generating Nuxt.js project in frontend
? Project name RoastAndBrew
? Project description Front End for Roast and Brew
? Author name Dan Pastori
? Choose the package manager Npm
? Choose UI framework Tailwind CSS
? Choose custom server framework None (Recommended)
? Choose Nuxt.js modules
  ● Axios
  ○ Progressive Web App (PWA) Support
  >● DotEnv
```

Coming up in the next tutorial, we will install another NuxtJS module, the Nuxt Auth module. Don't worry, we will go through it all. It's nice to have the installer auto set up a few of them for us though!

Step 9: Choose Linting Tools

If you've ever used a linter before, you will know the value. If you haven't think of it as a way to keep your code clean and sharp. Essentially, with a linting tool you can set up the default parameters for how you want your code visibly structured (tabs vs spaces). I personally love [Prettier](#). It's simple and easy to use and also very convenient to configure. Once again, this won't really affect any functionality of the application or our tutorials, so choose whatever you want or have familiarity with.

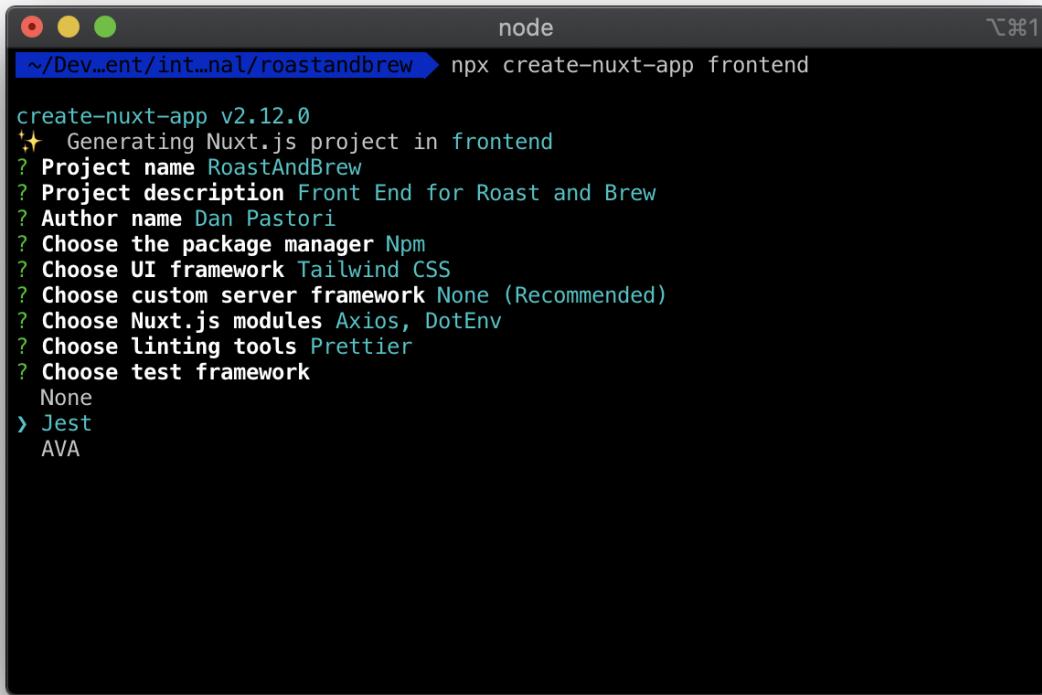


```
node
~/Dev...ent/int...nal/roastandbrew ➔ npx create-nuxt-app frontend

create-nuxt-app v2.12.0
✨ Generating Nuxt.js project in frontend
? Project name RoastAndBrew
? Project description Front End for Roast and Brew
? Author name Dan Pastori
? Choose the package manager Npm
? Choose UI framework Tailwind CSS
? Choose custom server framework None (Recommended)
? Choose Nuxt.js modules Axios, DotEnv
? Choose linting tools
  ○ ESLint
  >○ Prettier
  ○ Lint staged files
  ○ StyleLint
```

Step 10: Choose Testing Framework

So this option does make a difference on what you select. If you want to use some of the testing code that I'll be writing in ROAST, I'd recommend selecting Jest. I've used Jest before and it's an amazing javascript testing environment. Similar to PHP Unit, it's easy to organize and run tests for the front end of your application.



```
node
~/Dev...ent/int...nal/roastandbrew ➤ npx create-nuxt-app frontend

create-nuxt-app v2.12.0
✨ Generating Nuxt.js project in frontend
? Project name RoastAndBrew
? Project description Front End for Roast and Brew
? Author name Dan Pastori
? Choose the package manager Npm
? Choose UI framework Tailwind CSS
? Choose custom server framework None (Recommended)
? Choose Nuxt.js modules Axios, DotEnv
? Choose linting tools Prettier
? Choose test framework
  None
  > Jest
    AVA
```

Step 11: Chose Rendering Mode

There are two rendering modes you can choose from:

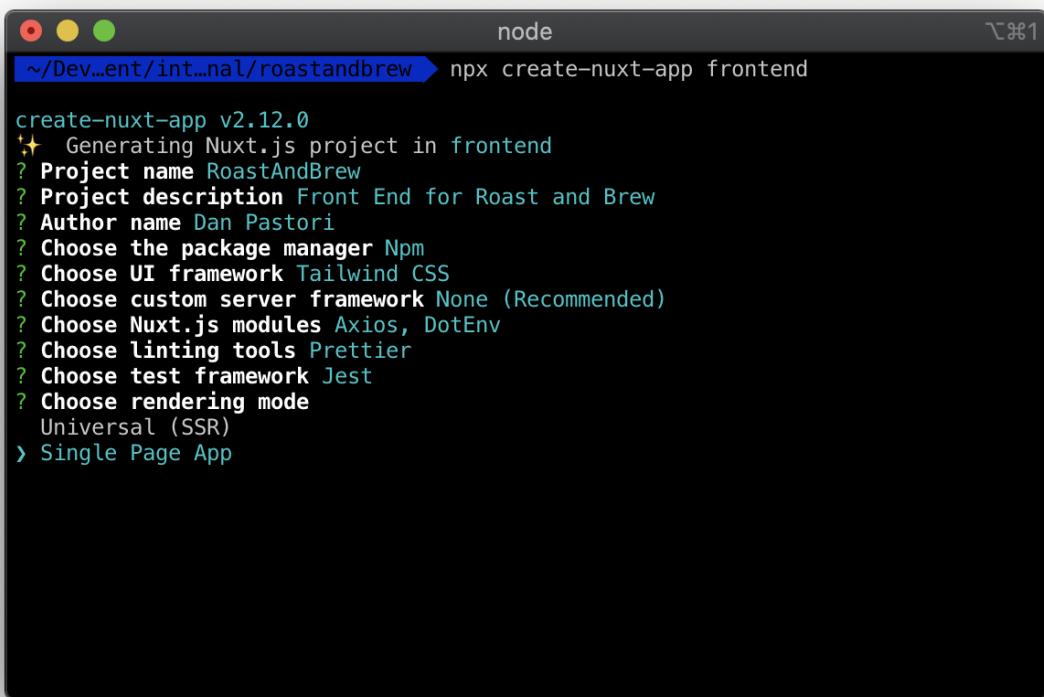
1. Universal (SSR)
2. Single Page App

Universal (SSR) is for Server Side Rendering. Using server side rendering, it actually renders the page on the server where the NuxtJS app lives and returns that data as a pre-rendered HTML page before initializing the single page application functionality. This is EXTREMELY powerful if your app needs SEO because you can actually return the proper metadata on response for scraping. We will select this option!

When we compile to mobile, we will use [Single Page App](#). With NuxtJS you can

switch this later or through the command line (which is AMAZING and we will be doing when working with CapacitorJS) and it installs all of the tricky config you'd usually have to do when setting up Server Side Rendering with VueJS.

The reason we will be doing it differently for the mobile app is we don't need to have everything pre-rendered and packaged for the deployed mobile app. There's no need for the extra packages and It's actually detrimental to the app since it increases file size. So we choose to go with [Universal \(SSR\)](#) for web and [Single Page App](#) for mobile. For now though, like mentioned, select [Universal \(SSR\)](#).

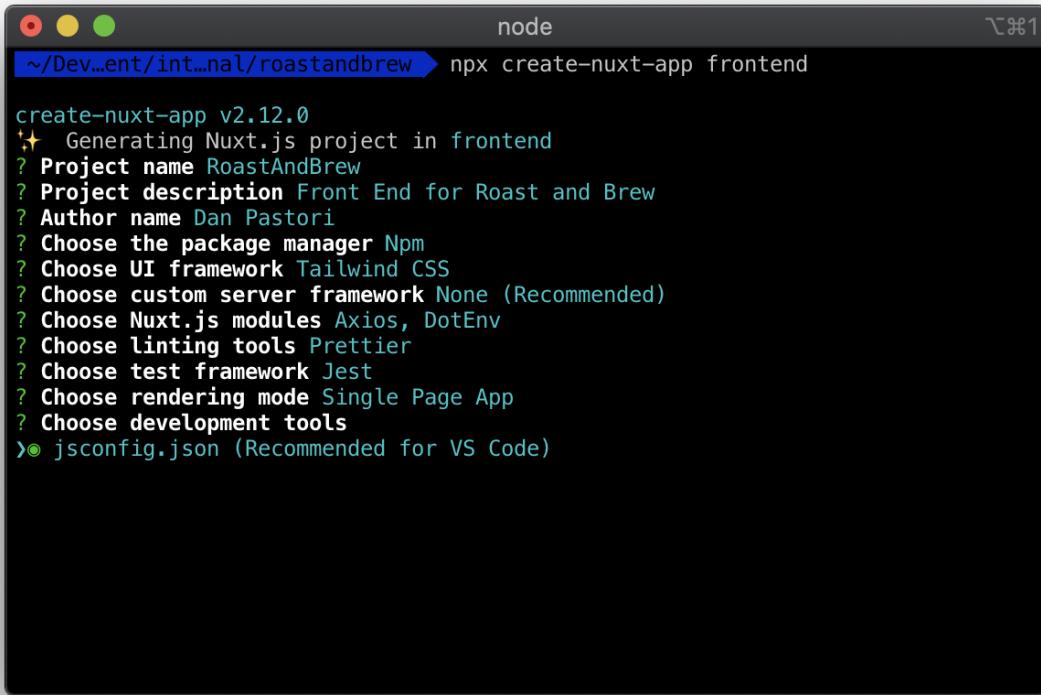


```
node
~/Dev...ent/int...nal/roastandbrew ➔ npx create-nuxt-app frontend

create-nuxt-app v2.12.0
? Generating Nuxt.js project in frontend
? Project name RoastAndBrew
? Project description Front End for Roast and Brew
? Author name Dan Pastori
? Choose the package manager Npm
? Choose UI framework Tailwind CSS
? Choose custom server framework None (Recommended)
? Choose Nuxt.js modules Axios, DotEnv
? Choose linting tools Prettier
? Choose test framework Jest
? Choose rendering mode
  Universal (SSR)
> Single Page App
```

Step 12: Choose Development Tools

If you are using VS Code (which I highly recommend), select the [jsconfig.json](#) file. This makes sure your environment for building your NuxtJS App is replicable across the team. If you are not using VS Code, skip this step.



```
node
~/Dev...ent/int...nal/roastandbrew ➤ npx create-nuxt-app frontend

create-nuxt-app v2.12.0
✨ Generating Nuxt.js project in frontend
? Project name RoastAndBrew
? Project description Front End for Roast and Brew
? Author name Dan Pastori
? Choose the package manager Npm
? Choose UI framework Tailwind CSS
? Choose custom server framework None (Recommended)
? Choose Nuxt.js modules Axios, DotEnv
? Choose linting tools Prettier
? Choose test framework Jest
? Choose rendering mode Single Page App
? Choose development tools
❯ jsconfig.json (Recommended for VS Code)
```

This should be the last step that you have to do! Now, you can sit back and watch NuxtJS use it's amazing installer and get your app ready to rock and roll!

Understanding NuxtJS' Structure

For those of you have followed along with the Server Side Up series, you will definitely see some similarities between how NuxtJS is set up and how we structured our environment. This is one of the main reasons that made me comfortable with switching to NuxtJS. For more reasons on why we chose NuxtJS instead of our previous set up, read, Why Nuxt?

For an in-depth overview, NuxtJS has a [beautiful guide](#) that dives into their directory, API, and all of the features of NuxtJS. After we do a quick run through of where we are spending our time with NuxtJS, we will begin implementing authentication functionality with our API and adding value to our Laravel and NuxtJS installs!

For now, let's get acquainted with NuxtJS.

A Beautiful Install of the VueJS EcoSystem

Just to begin, NuxtJS is essentially a beautiful install of the VueJS Ecosystem (VueJS + Vue Router + Vuex). Instead of installing all of libraries individually, the install process we went through has everything installed and configured right out of the box. We can use these libraries without much configuration. For now, know that the VueJS ecosystem is what we are using. Familiarity with VueJS was a pre-req, but Vuex and Vue Router might be a little bit foreign. For more information on Vue Router and Vuex you can browse their docs here:

<https://router.vuejs.org/>

<https://vuex.vuejs.org/>

We will be going through each of these libraries and covering a ton of gotchas, implementations, tricks, etc. Right now, just know they are installed and configured.

Configuration

The most important file with NuxtJS is the config file. This is found in the root directory of your theme and is named `nuxt.config.js`. You will become very familiar with this file. Every piece of configuration for our SPA will be in this file. All plugins, settings, features, etc. It is extremely nice to have this set up in one place and easily accessible. Feel free to open it up and explore some of the settings. We will be diving into this file regularly and explaining every step of the way.

Components vs Pages vs Layouts

In VueJS, everything is a [Vue Component](#). Within NuxtJS, everything is configured to be single file component as well which is wonderful! That means all CSS, JS, and HTML are in one convenient package.

Within NuxtJS there are 3 flavors of these components:

1. Components
2. Pages
3. Layouts

Let's start with components. These are the basics of re-usable code. In NuxtJS you will be using the components in your `/components` directory to be buttons, forms, headers, tables, navigation, etc. Anything re-usable or places you want to scope data and functionality.

The next step up from these components are pages. Pages are components as well, just viewed differently from within NuxtJS. When using Vue Router, you set up each page of your single page app as a component. Within that page component are smaller components that add to the layout, re-usability, features, etc.

If you were using Vue Router outside of NuxtJS, you'd make a route within your app that would render a page component. Within Nuxt, you will be creating these

components within the `/pages` directory and in sub-directories. Now one thing to be aware of, is there are special naming conventions. This is because NuxtJS will dynamically build routes OUT OF the page file names within your `/pages` directory. We will touch on this a lot more and you will see how this works as we build features into our application.

Quick example. We will be showing lots of coffee companies. We want a url that is `/companies` to show these companies. Inside of our `/pages` directory we'd make a `/companies` directory and an `index.vue` page component. This would resolve to the URL `/companies` within our single page application.

Say we want to show a specific company and provide a page to edit that company. We'd make a directory within the `/pages/companies` directory named `/_id` or `/_slug` (whatever unique identifier you'd like) and then inside that directory add an `index.vue` page or an `edit.vue` page. This would resolve to `/companies/{id|slug}` and `/companies/{id|slug}/edit` respectively.

For all the different naming schemes and scenarios, the official documentation on [NuxtJS routing](#) is really well done and will cover anything I missed.

Finally, and at the highest level, we have page layouts. This is essentially a shell that you can use on a variety of pages. I used to create a similar structure with my VueJS apps, but what is nice is NuxtJS has this baked in by default. Once again these are components that render a view and live in the `/layouts` directory. You'd use these top level components to show headers and navigations that are consistent along pages of your application. In our examples, we will be creating an `app` layout that all of our pages will inherit from. This layout will contain all of our global level app components.

Store Directory

We will be working HEAVILY in this directory since it will store all of our Vuex modules. Vuex is essentially a local way to store global data within our SPA. The rule of thumb I use for deciding when to use Vuex, is "if you need the data on more than one page, store it in Vuex". We will be using Vuex to store the

authenticated users, settings, etc. As we create functionality, we will show how to use Vuex in a lot more detail to come regarding Vuex and a ton of examples.

One important difference between the Server Side Up series and how I determine what goes into the Vuex store, using the rule of thumb I mentioned. Before, I'd hold mass quantities of data in Vuex and rely on it within my page component. For example, loading all cafes. Now, I prefer to load all cafes into the cafes page and through a method encapsulated within that page. This saves heavily on implementation time since it's something we only need, scoped to that page.

Static vs Assets Directory

These directories are different than what we used in the Server Side Up series. Both of these directories store images, css, js, and other files you may need within your single page application. To sum it up, the `/assets` directory contains assets that Webpack has access too, such as background images for CSS. The `/static` directory is used to store assets that don't need any sort of compilation such as a logo.

Tests Directory

This is another directory where lots of code will be placed since it will house all of our Jest tests. As we develop the app we will put all of our front end tests in this directory.

Middleware Directory

One of the most important and awesome features of NuxtJS, the SPA middleware. This allows us to block off certain actions and pages within our app if the user is not authenticated or does not have permission. We have to be careful with SPAs since they are all Javascript and are accessible for adjustments on the client side by nosy programmers. However, to block access and have these capabilities for UX, it's EXTREMELY helpful. Oh, and it pairs nicely with the Laravel middleware we use on the API side.

We will definitely be creating some middleware and going through proper use cases on how to use it within an SPA. For more information on security within an SPA, you can jump to:

The Truth about SPA Security

Additional Resources

Here are some additional resources on using NuxtJS. Even though, their documentation is beautiful and easily consumable, these resources just help explain concepts in a variety of different ways. After this, we will start to focus explicitly on our API driven functionality and into places less documented and more unique to an API Driven Application.

- <https://nuxtjs.org/guide>
- <https://vueschool.io/courses/nuxtjs-fundamentals>
- <https://jestjs.io/>

That's a quick top level overview of NuxtJS! There are a ton of cool features baked into NuxtJS, ready for use and we will be touching on more of them as we begin to implement our applications. Now it's time to start making these installs function more from a perspective of an API Driven Application and gear the tutorials to a more customized approach.

Abstracting Your API Calls Into Reusable Modules

So this is a concept that I found extremely useful while developing API Driven Applications, abstracting all of your API routes into nice little JS modules. By abstracting all of the used API route calls into nice little modules that you can integrate into separate parts of your application, it saves you time and allows you to easily re-use code. This allows you to make changes that, when needed, will affect the API call to the API endpoint from every place in the application leading to more consistent, re-usable code. You can even use the calls within `asyncData` which I will discuss shortly.

Since we haven't added any code to our application yet, I'll go through how to set this up, add your first module, and how we will use it in a component or Vuex store. It might not make a lot of sense right away, but when we get to adding our first resource, we will see how it ties together. We are adding more "buckets" and stubbing out some implementation.

What Makes Our Way Different

Alexander Licher (one of the core maintainer sof NuxtJS) provides a great example of this in his blog post here: <https://blog.lichter.io/posts/nuxt-api-call-organization-and-decoupling/>. He abstracts and decouples the API calls for each resource into a separate repository. The way I approach is inspired by his, but implemented differently and structured differently.

The main approach I took, was not making a re-usable set of endpoints for every resource, but explicitly making individual modules for each API resource. This is because some HTTP Verbs (GET, POST, DELETE, etc.) will not be used on every resource and more importantly, no matter HOW HARD you try to wrap everything in perfect RESTful endpoints, you will always have an endpoint that you want special or not as complex. Maybe you have an endpoint that does computational data on a resource that doesn't match a standard show or index endpoint? With

this approach, and my love for explicit maintainability in code, I believe that having explicit modules for each API resources is super clean and very helpful.

I will also allow parameters to be explicitly defined and defaults initialized as well. This allows the developer to pass URL parameters to GET requests and will help when uploading files through POST.

To make this work, I create a plugin for NuxtJS and make the methods available through a global variable. This plugin will be set up through `this.$api` which is accessible in our components and in our pages, layouts and vuex stores. Let's get started!

Step 1: Make `/api` directory within your NuxtJS Install

The first thing we need to do is make an `/api` directory at the root of your NuxtJS install. In this directory we will be storing our modules that access our Laravel API. In the future, feel free to add separate folders in here as well if you develop a massive API. We've had to do this for other projects and it's helped a ton!

Step 2: Add NuxtJS Plugin

To make this accessible we are creating a [NuxtJS Plugin](#). According to the documentation:

Nuxt.js allows you to define JavaScript plugins to be run before instantiating the root Vue.js Application. This is especially helpful when using your own libraries or external modules.

Our plugin will be our API adapter and we will be using the already installed `@nuxtjs/axios` module to make our required API calls. The reason we have to do this as a plugin is we need to have the global axios module accessible so we can have access to all of our configuration that we defined for axios. We also don't want to include axios from the package or we will have two of the same packages, different versions, multiple configuration files, and over all just messy.

The `@nuxtjs/axios` module has a little [documentation](#) on how to use it within your custom plugins as well.

Since we are building an app called ROAST, let's add a file in the `/plugins` directory called `roastAPI.js`. This will be where we store our API call plugin.

Step 3: Add Some Code to Plugin

Let's start by adding this code to our `/plugins/roastAPI.js` file:

```
export default function( { $axios }, inject ){
  const api = {

  }

  inject('api', api);
}
```

Okay, time to break this down. First, we are exporting a function that accepts an object and the `inject` method. We then de-structure `$axios` in the parameter since that's all we will be using. By gaining access to the global `$axios` plugin, we can use it to pass to the API modules to call our API with all of the configuration necessary.

The `inject()` function will then, after loaded, add the `api` plugin to our global Nuxt instance making it accessible throughout our application. The first parameter in the `inject()` method is the name of our plugin, the second parameter is the variable which provides the functionality. In this case, our plugin will now be accessible within pages and components by `this.$api`.

We are creating a constant `api` variable which is right now just an empty JS object. This is what will get mapped to global app and to the `api` plugin through the `inject` method. Every API module that we import will be a key in this object. This will provide nice scoping for each module and will make it available throughout the whole application.

Step 4: Register Plugin in `nuxt.config.js`

This is the last part of ensuring that our plugin is recognized by NuxtJS. We just need to let our NuxtJS install know our module exists. To do this open up the `nuxt.config.js` file and find the `plugins` key. In this key, map to our plugin like this:

```
plugins: [  
  '~/plugins/roastAPI.js'  
,
```

Now NuxtJS is aware of our module and we are ready to continue our set up of NuxtJS. We can go back to our plugin and get a module added.

Step 5: Set up Axios baseURL

This is extremely important and super helpful when calling the same API for the entire application. Especially routes behind authentication blocking.

When setting up the `baseURL`, you can configure Axios to automatically add the `withCredentials` header to the request. This will send the `X-XSRF-TOKEN` cookies if you are using Laravel Sanctum (which we will be setting up soon) with every request, which, as you can imagine, is super important for determining authentication of the user.

For every other request, you can just use the relative URL and the `baseURL` will be prepended to the request URL making everything super clean and also very configurable as you push to production.

To set up the `baseURL` open up the `nuxt.config.js` and find the `axios` key. In there, add the following property:

```
axios: {  
  baseURL: '{BASE_URL_HERE}'
```

```
},
```

It's that simple! And so extremely helpful. For our testing environment, our base URL is at <https://roastandbrew-api.521.test/>. When we use our Laravel API, we house our routes behind the `/api/v1` prefix, so it's nice that all we have to do is appended that to every request.

This will be extremely easy since we have extracted our resources into their own modules.

In the axios configuration, add the following right below your `baseUrl` configuration:

```
axios: {  
  baseURL: '{BASE_URL_HERE}',  
  credentials: true  
,
```

Now whenever we make a request to our base URL, we will add the `withCredentials` header so we can authenticate appropriately if the request needs authentication.

For now we have everything set up for when we start adding modules and making heavy API requests. We will run through an example of how to get a module added. When we get to our first feature, we will revisit this in a different context and it will tie everything together.

Step 6: Adding Your First Module

Now that we have our plugin set up, let's add our first module! If you feel comfortable to following along right now and see how this works, keep reading! Otherwise, skip to the section Managing App Resources Through an API. We will go through a full-stack example there.

Since our app is focused around cafes and coffee companies, one of the first

modules we will be adding will be a company module. This module will contain all of the methods necessary to manage a Company resource. The first step is to add the file `/api/companies.js` and add the following code:

```
export default $axios => ({
  async index( params ){
    return await $axios.$get('/api/v1/companies', {
      params: params
    });
  },
});
```

Let's step through this. First we export a default module and pass `$axios` as the first parameter. This is how we use the NuxtJS axios module in our plugin. We send the reference via a parameter.

Next up, we have our method signature which is `async index(params)` which accepts a JSON object of parameters to help and query the companies endpoint. If we want to add query variables to each of the endpoints, this is how we would properly add them. Axios takes care of appending them to the request. We also define this as an `async` method so we can use it with an `async/await` syntax AND with `asyncData` which is super convenient.

Now we get to our actual request. We return the promise from the axios module and submit a `GET` request to the `/api/v1/companies` endpoint. We haven't built this yet, but this will be the endpoint of your API resource you are mapping to. We will be building this shortly. The second parameter is a config parameter that we pass the `params` to. Axios will then append them to our request so we can use them in our API.

This is our first API request! As we build out our API and continue to add functionality, these modules will contain all of our request code. For now, we are doing a simple abstraction to show how it's done. If you want to skip right to the core of where we do this, check out the [Gitlab repo](#) available with The Complete

Package and you can see all of the requests within ROAST. We will also cover this in much more details in an upcoming chapter.

Step 7: Register with NuxtJS API Plugin

In Step 3, we added the `plugins/roastAPI.js` plugin which was pretty much an empty shell of a plugin.

```
export default function( { $axios }, inject ){
  const api = {

  }

  inject('api', api);
}
```

Let's register our companies module so we can use it! The first step is to include the companies api module on top of the plugin like this:

```
import CompaniesAPI from '@/api/companies.js';
```

Now we just need to add the module to our `const api` object like this:

```
import CompaniesAPI from '@/api/companies.js';

export default function( { $axios }, inject ){
  const api = {
    companies: CompaniesAPI( $axios )
  }

  inject('api', api);
}
```

So what we are doing is adding the newly created `CompaniesAPI` module to our `roastAPI.js` wrapper. We are giving it a key of companies and passing the `$axios` variable to our module to use. Now how will this work? In any component where you need to make a companies request, you will just run:

```
this.$api.companies.index()
```

The `.companies` key makes it easy to read showing that you are calling the companies API. We will be adding a `.create()` method to this module and a `.show()` method where we can pass the variables to add a new company and display an individual company. But if we need to update any of these requests, we can do it in one place which is ultra convenient! Let's continue our configuration of NuxtJS. We will do more with modules later.

Managing Events With the Event Bus

This is another trick that will pay dividends that I ALWAYS SET UP. It's an Event Bus. In VueJS 1.x there was a different way of passing and propagating events between components using the `$broadcast` and `$dispatch`. To be honest, I kind of liked it, even though I would consistently get confused of the DOM direction that each of the methods provided. However, there's a much better way to do that and that is in the form of a simple Event Bus.

In the [discussion of upgrading from Vue 1.x to 2.x](#), the concept of an Event Bus is discussed. However, Alligator.io provides a [WONDERFUL simple implementation](#). I mean it's 2 lines of code, but makes a WORLD of a difference. Even when using Vuex!

We will use the Alligator.io implementation. Let's start with some basics though.

Why an Event Bus?

Especially when using Vuex you might ask. Personally, there are some scenarios that I believe are best resolved through some form of component to component event propagation compared to sending data to Vuex. These examples include small data transmissions such as a success message after a response, or a flag that needs to be updated.

The argument can be made that the Vuex data store should be used indefinitely, no matter how small the state, but I tend to find some of that too cumbersome. Showing a 2 second notification when a request has been completed seems overkill for Vuex. I will dive into Vuex much more later on in this book, but I tend to think of using Vuex more for storing much more important state such as authenticated users, lists of data needed between components, or where to navigate after an action has taken place.

The other benefit of the Event Bus is you can register to listen to certain events and any component or page can work with it. You can then emit the event and the

component can react accordingly. There will be many examples of how we use the Event Bus coming up, so let's get one added and set up!

Step 1: Add your Event Bus File

It's really easy to get an Event Bus implemented. First, let's add a file at the root of your NuxtJS install called `/event-bus.js`.

Step 2: Add your Event Bus Code

Open up the file and add the following code:

```
import Vue from 'vue';
export const EventBus = new Vue();
```

That's it! What this allows us to do is have a light weight VueJS instance that we can listen to and react to when events are propagated. There are 3 methods that we will use:

1. `$on` This will subscribe a component to an event
2. `$off` This will unsubscribe a component to an event.
3. `$emit` This will emit an event that components can react to.

To include your event bus on a component, you will simply import the event bus like a standard javascript module with `import { EventBus } from '@/event-bus.js'`.

One thing to note, and the [VueJS docs do a good job highlighting it](#), is that you will want to unsubscribe from all events by using the `$off` method in the [beforeDestroy lifecycle hook](#). This will ensure that if your component is not being used, it will not react to events being propagated even if it's one it would tend to listen to. In a single page application this is VERY important to follow because there is no hard refresh between navigation to clear some components,

especially parent routes.

I also tend to bind to events in the `created()` life cycle hook so they are ready to operate right when the data is rendered.

If this is a little confusing now, don't worry! Like I said, there will be plenty of examples to try out and see how it works!

Saving Time With Mixins

Mixins are a simple, reusable way to add functions to your VueJS/NuxtJS project. Essentially a mixin is a global function or group of functions that can be added to any VueJS component.

This all sounds very similar to what we discussed with plugins, right? Before we get too deep into setting up our mixins, I want to make one important distinction between plugins and mixins. In NuxtJS, there are plugins which provide extra functionality to your app (similar to a mixin).

Honestly, you could use these interchangeably, but here's how I determine whether to use a plugin or a mixin. Does your extra functionality require global level components like `$axios`? If so, create a plugin. Will your functionality require a complex system of functions? If yes, create a plugin. NuxtJS has a beautiful plugin infrastructure. Are you creating a single function that you want in a few places but not global? Create a mixin.

On a side note, if you are choosing NOT to use NuxtJS you can easily make some complicated mixins that act like plugins.

So when should we use a mixin?

I like to use mixins when the functionality is not complex and doesn't require access to global app components. I also like to use a mixin when the functionality is limited to one re-usable function. Kind of similar to the Event Bus vs Vuex approach. Use Vuex to store complicated pieces of data and Event Bus for short, one time flags to communicate between components. Use a mixin if you have a single function you'd like to re-use and a plugin if you have a process or repository of functions to use.

Let's get started to prepare our app for mixins!

Step 1: Add a `/mixins` directory at the root of your NuxtJS install

Yup, you might have guessed, another “code bucket” to add! This directory will store all of mixing functions we will be creating. If you end up having a lot of these functions, considering making some scoped sub-directories. For now, just adding the `/mixins` directory will be perfect!

For now, that's honestly all we need to do! We are now prepared to use mixins with our application as well. The first mixin we will add will be to transform a form into a `FormData()` object so we can upload files. The reason we chose a mixin is because it shouldn't be globally present on all pages and is a single function that is re-usable in multiple places.

I'll touch on how to add a mixin when we get to that point! These little code snippets will save SO much time when you implement them.

Building Your First Layout

In my opinion, layouts are one of the most important built in features of NuxtJS. They allow you to design re-usable page layouts that can be easily implemented into your application. Let's walk through a basic example of a layout just to get a feel for how they should be used. We talked a little bit about page layouts here: Understanding NuxtJS' Structure

Let's dive in a little further and actually implement our first layout. This will be the baseline for most of our app's pages. If you wanted to make a different UI for different perspectives (app vs admin) you could make a separate layout for each. For now, let's just make a basic app layout.

What goes into a layout?

When designing a layout, it's usually pretty simple. Ask yourself the question, "Do you want every component present on every page that uses the layout?" If the answer is "yes" put it into the layout. The three things I can think consistently fit into an app layout are:

- Global Notifications
- A Header
- Sidebar/Navigation
- A Footer
- Authentication Modals

Of course, you are not limited to these features but this is a good place to start.

Our app layout

For now, we will start with our [App](#) layout. Right away, I want the design of ROAST to have a header that has a search and an avatar if the user is authenticated or a

link to login if not.

I also want to have a floating footer menu to contain all of our navigation within the app when in mobile responsive mode. For now, we will have two links:

- `/` which will be our default page
- `/search` which we will implement in one of our first tutorials. It's here for now as a placeholder.

Let's start by creating our `layouts/App.vue` file and adding the following code:

```
<template>
  <div id="app-layout">

  </div>
</template>

<script>
  export default {

  }
</script>
```

This is just a basic Vue component and that's exactly what we want right now. We need a blank shell that we can add components to.

Adding Global Components

The way I like to structure components used by multiple pages is to make a `/components/global` directory and add the components in there. In this directory, I added a `/Header` directory and the `AppHeader.vue` component. I also added a `/Footer` directory and added an `AppFooter.vue` component. I won't be going through all of the details for these components since they are very app specific, but let's go through a few notes.

First, I named the header component `AppHeader.vue`. The reason for this is the `<header>` tag already exists within HTML5. We will actually be using that tag WITHIN our component. Be aware that some of the global components might clash names with existing HTML5 elements. This is one of those examples.

Next, I created a directory for these global components. The reason being is that any component that needs sub components I like grouping in a directory so it's easy to focus. Essentially, for large components it's nice to keep their sub components that are only used within a parent component grouped by directory. However, once again, this is very app specific so I won't dive into many more details, but feel free to explore on GitLab and reach out with questions!

Within these base components we have a `<nuxt-link>` component. This is the NuxtJS wrapping for `<router-link>` within Vue Router. It accepts a `:to` parameter that we can use to link within pages. There are a lot of ways to use `<nuxt-link>` and the [NuxtJS documentation](#) goes through them extremely well.

Now that we have these components, let's go back to our `/layouts/App.vue` layout and add the following code:

```
<template>
  <div id="app-layout"
    class="flex flex-col min-h-screen w-screen select-
    none">
    <div>
      <app-header v-show="showHeader"/>
    </div>

    <main>
      <nuxt />
    </main>

    <app-footer v-show="showFooter"/>
  </div>
</template>
```

```
<script>
  import AppHeader from '@/components/global/Header/
  AppHeader.vue';
  import AppFooter from '@/components/global/Footer/
  AppFooter.vue';

  export default {
    data(){
      return {
        }
    },
    components: {
      AppHeader,
      AppFooter
    }
  }
</script>
```

What this does is loads our two global components, and creates an app layout. Within this layout you will see the `<nuxt/>` component. Like with the `<nuxt-link>` component, this is NuxtJS' wrapper for the `<router-view>` component within Vue Router. Within this component is where our application will render the pages that use the layout. We will now have a header and footer on each page. If you [explore the repo](#), you will see this layout has quite a few other components added. There are a handful of global components you want available that we will touch on as we build. This is a good starting point and basic overview.

I did also add a few computed values to show the header and the footer. There are a handful of pages that I don't want either to appear. Since they are on the majority of pages, I added them globally. With the few pages I don't want them to appear, I added a computed value that returns if we are on a specific page or not.

If we are, we just hide the specific component.

```
computed: {
  showFooter(){
    let pages = [
      ];

    return pages.indexOf( this.$route.name ) == -1;
  },
  showHeader(){
    let pages = [
      ];

    return pages.indexOf( this.$route.name ) == -1;
  }
}
```

That's all it takes to create a layout within NuxtJS! When we get into adding our first resource we will implement this layout and add some more functionality. Next up, we will discuss a little bit about how to add pages, then on to authentication. Authentication is definitely the most difficult part of building an API + SPA combo.

Adding Pages with NuxtJS

Right off the bat, this was one of the hardest things for me to grasp coming from using Vue Router directly. NuxtJS automatically generates pages based off of the name of the file and the directory structure. Before I explain some examples and some gotchas, [check out the docs](#). They do a wonderful job going through a lot of scenarios. However, even after reading the documentation, I had a few questions and interpreted things a little differently.

I'll explain the basics, how to structure your pages directory, and some gotchas.

Adding A Page

As we briefly touched on a page in NuxtJS is just a component. Exactly the same way it is if you are using Vue Router by itself. The difference between using Vue Router by itself vs in NuxtJS is you define your routes in a specific file. In the example we did on Server Side Up, we had a `routes.js` file that we defined our routes in.

In NuxtJS, it's backwards. You define your routes based on the directory structure of your `/pages` directory. Let's say I was going to add a page that listed all cafes in our application and were using Vue Router **outside** of NuxtJS. You'd probably add a file in your Vue Router config like this:

```
export default new VueRouter({
  routes: [
    {
      path: '/cafes',
      name: 'cafes',
      component: Vue.component( 'Cafes', require( './
pages/cafes.vue' ) )
    }
  ]
});
```

Then you'd go and add the `/pages/cafes.vue` file where you'd implement your `cafes` page.

In NuxtJS, you'd do it in reverse order. You'd navigate to the `/pages` directory and add a directory called `/cafes` then add a file `index.vue`. Similar to how you'd name routes through the API with resource controllers, you'd design your page structure within NuxtJS.

Adding a Dynamic Page

Using our above example, let's say we want to add an individual cafe page. This page should respond dynamically to what's in the URL. If we were to do this in plain Vue Router, you'd end up with something like this:

```
export default new VueRouter({
  routes: [
    {
      path: '/cafes',
      name: 'cafes',
      component: Vue.component( 'Cafes', require( './
pages/Cafes.vue' ) ),
    },
    {
      path: '/cafes/:id',
      name: 'cafe',
      component: Vue.component( 'Cafe', require( './
pages/Cafe.vue' ) )
    }
  ]
});
```

Then you'd navigate to your `/pages` directory and the `Cafe.vue` page.

In NuxtJS, you'd add a file in your `/pages/cafes` directory named `_id.vue` and

the route structure would be generated for you.

Once you get the hang of this, it is actually EXTREMELY powerful and way easier than using Vue Router. While you are running your app, these routes dynamically generate and build themselves. It just takes a little getting used to.

Adding a Layout

We did this on the Server Side Up series by creating a global parent page, calling it a layout, and having every other page be a child of that page. It worked pretty nice, but no where NEAR what Nuxt provides. A layout from our perspective is simply a format of the page that allowed us to use global components such as a header and sidebar.

Let's look at how we did this with Vue Router. We created a layout and set each child as a root of that layout like this:

```
export default new VueRouter({
  routes: [
    {
      path: '/',
      name: 'layout',
      component: Vue.component( 'Layout' ),
      require( './pages/Layout.vue' ) ),
      children: [
        {
          path: 'cafes',
          name: 'cafes',
          component: Vue.component( 'Cafes' ),
          require( './pages/Cafes.vue' ) ),
        },
        {
          path: 'cafes/:id',
          name: 'cafe',
          component: Vue.component( 'Cafe' ),
        }
      ]
    }
  ]
})
```

```
        require( './pages/Cafe.vue' ) )
    }
]
}
])
});
```

In NuxtJS, you'd create a layout in the form of a component with the name that you want in the `/layouts` directory (like in the last section). For example, add `App.vue` with the following code:

```
<template>
  <div id="app-layout" class="flex h-screen w-screen">
    <div>
      <navigation/>
    </div>
    <div class="flex flex-col flex-1 h-screen">
      <app-header/>

      <nuxt />
    </div>
  </div>
</template>
```

Say we have a navigation component and a header component and we want all the differences (what makes the page unique) rendered in the middle of the screen. That's where the `<nuxt/>` component comes in. This is the Nuxt wrapper for `<router-view>` component.

Now when we create our pages, all we have to do is add the `layout` property to our page component like:

```
export default {
  layout: 'App',
```

```
}
```

This will set the layout of the page to be the `App.vue` layout we defined. It's really that easy!

Adding Pages That Aren't RESTful

These are pages that aren't considered RESTful and are outside of the RESTful resource naming convention like a contact page or privacy policy page. Don't worry Nuxt makes it extremely easy. You literally just name the file what you want the route to be.

Say you want a `/contact` route. You'd add a page to the `/pages` directory named `contact.vue`. Want a `/register` route? Add `register.vue` to `/pages`. Nuxt makes this a breeze!

There are other validations you can do on route params and you can nest your pages as deep as your app pleases. The documentation explains all of this and we will cover it as we encounter it. This should give you a basic overview of how to add some pages to your Nuxt app.

USING LARAVEL SANCTUM FOR API AUTHENTICATION

Building Secure Authentication API Endpoints

For our application we will be using [Laravel Sanctum](#) for authentication. Laravel Sanctum was built for first party Single Page Application authentication with a Laravel API. Pretty much exactly what we are doing! While similar results can be achieved with Laravel Passport, we will be using that explicitly for granting 3rd party access to our API. This would be the case if a user wants to load coffee company and coffee shop data into their own application.

For now, we will be setting up authentication using Laravel Sanctum to get our app in motion!

Before We Get Started

Before we get started, I want to point out that we are setting up the routes necessary to work with Laravel Sanctum and for the [NuxtJS Auth](#) module.

As of this writing, that module is on the dev branch, but the Laravel Sanctum functionality is ready to use. We want to take advantage of this specialized

functionality provided by NuxtJS to work explicitly with Laravel Sanctum. They also allow this to work with sharing the proper authentication cookies through SSR which is extremely important. Otherwise, your initial requests may not be authorized correctly.

When we get to the mobile section of this book, we will be adding another authentication strategy that we will be covering in detail that allows for our mobile app to authenticate with our API. For now, let's just focus on Laravel Sanctum for web.

The routes that we will need to implement are the following:

1. `/login`
2. `/logout`
3. `/api/v1/user`
4. `/register`

We will go through all of them. One thing to point out, is 3 out of the 4 of them don't have the `/api/v1` prefix. In order to follow the RESTful guidelines and keep this application as "stateless" as possible, we will be creating these routes in the `/routes/web.php` file so we have complete separation of concerns with authentication and api requests. When we work with Sanctum, you will see that we will be applying a "stateful" middleware to the API routes. We will explain that more when we get there because it makes more sense. For now, consider the `/routes/web.php` file to manage our authentication and the `/routes/api.php` file to be stateless requests to our API.

When we get to mobile, we will have to put these methods behind the `/api/v1` prefix so they don't get blocked by CSRF protection, but we will ensure other protections are in place.

Also, if you looked at the NuxtJS Auth module's requirements, you will see the first 3 routes, but not the `/register` route. We will handle this one in a special

manner and eventually use the authentication module along with the functionality of this route.

To give credit where credit is due, I used fritsvt's [Laravel Nuxt Authentication boilerplate](#) for a lot of inspiration with this. He used the `tymon/jwt-auth` package and I followed similar functionality, just with the mindset of using Laravel Sanctum.

Step 1: Install Laravel UI

Right off the bat, you might be wondering if we are going to use the Laravel UI Auth package. The answer is, yes, sort of. Since we don't actually have any UI being rendered from our API, you might wonder why we even need the package. Well, it contains a few traits that we will be implementing as our application grows, such as the must verify email trait and functionality to reset passwords. All of this will be overwritten from an API perspective, but we still need the package.

To install the package, simply run:

```
composer require laravel/ui
```

We now have all the special functionality we need to handle the authentication procedures. You won't need to run the set up commands since we don't want to actually create any views. All of our views will be through our SPA. We will work more with this package in the future, but it's good to have it now.

Step 2: Create the AuthController.php

The first thing we need to do is create the `AuthController.php` file in the `app/Http/Controllers/Auth` directory and add the proper namespace:

```
<?php  
  
namespace App\Http\Controllers\Auth;  
  
use App\Http\Controllers\Controller;
```

```
class AuthController extends Controller
{
}
```

Now we are ready to begin our implementations. This will make more sense when we start doing frontend calls to these routes that the controller handles, but for now think of these routes as just placeholders that will be endpoints to receive authentication data and return tokens or authentication cookies.

Step 3: Register the `/register` route

For now, we are just going to register this route. There will be some specific implementations we will have to work with for Laravel Sanctum, but in the spirit of making "code buckets" let's just have this stubbed out. This is a good place to begin with an empty application. I mean, we have to have users before we can log them in and return their data.

First, we need to add the `/register` route to our `/routes/web.php`. Let's open up the `/routes/web.php` and add the `register()` route:

```
/**
 * Authentication Routes
 */
Route::post('/register', 'Auth\AuthController@register');
```

Next, we need to make a method in the `\Auth\AuthController` that handles the `register()` request. So open up the `app/Http/Controllers/Auth/AuthController` and add the following method:

```
public function register( Request $request )
{
}
```

and add the `Request` object to the `use` section of your controller:

```
use Illuminate\Http\Request;
```

The other aspect I'll point out is we will be sending a `POST` request. This is because our intent is to create a resource (User) and will have to transmit sensitive date (such as a password). So make sure you are using `https://` as well!

Step 4: Register the `/login` route

Like the `/register` route, we are just going to stub out this implementation and register the route to implement later on.

We will need to go back to our `/routes/web.php` file and add the `/login` route right after the `/register` route:

```
...
/**
 * Authentication Routes
 */
Route::post('/register', 'Auth\AuthController@register');
Route::post('/login', 'Auth\AuthController@login');
...
```

Next, we will open up the `Auth\AuthController` and stub out the `/login` method like this:

```
public function login( Request $request )
{
}
```

We are set up to implement this route and will do that in a few sections.

Step 5: Register the /logout route

Like the other routes before, we are simply going to stub out a `/logout` route that the user can use to terminate their session.

First, let's stay in the `/routes/web.php` file and add the `/logout` route:

```
...
/** 
 * Authentication Routes
 */
Route::post('/register', 'Auth\AuthController@register');
Route::post('/login', 'Auth\AuthController@login');
Route::post('/logout', 'Auth\AuthController@logout');

...
```

Like before, we are just going to stub this route out making a "bucket" to implement later on. This time, we should open the `Auth\AuthController` and add the `/logout` method like this:

```
public function logout( Request $request )
{
}
```

Step 6: Register the /api/v1/user route

Since the majority of our front-end will be NuxtJS, the NuxtJS auth module requires a route to load the authenticated user into the Vuex Store so the user is available through the entire SPA. This route will be `/api/v1/user`.

Now, since this has nothing to do with authentication, and we WILL have more routes that deal with users and user management, let's create `app\Http\Controllers\API\UsersController.php`:

```
<?php
namespace App\Http\Controllers\API;
```

```
use App\Http\Controllers\Controller;
use App\Models\User;
use Auth;

class UsersController extends Controller
{

}
```

Now, that we have the controller created, let's create the route and we will come back and implement the method to return the user. Open up the `app/routes/api.php` and add the following route within our `/v1` prefix:

```
/**
 * User Routes
 */
Route::get('/user', 'API\UsersController@show');
```

Before we jump over and stub out our method, notice I named it `show`. This is because of the standards used by Laravel with resource controllers. The `User` we are returning is a resource. Granted we don't have a specific ID we are returning, we are returning the logged in user so we are mapping that to the `show` method. We will be using the [Laravel Resource Controller](#) naming scheme heavily throughout this book. When we get into handling our first resources, this standard will be heavily implemented. For now, we are getting our first exposure to an application resource and that is our User.

Now, let's stub out our method and reopen the `app\Http\Controllers\API\UsersController.php` and add the following method:

```
public function show()
{
}
```

For now, we will leave it as is, but like the other routes, we will come back and implement it when we have a few more settings in place.

Implementing Cross-Origin Resource Sharing (CORS)

CORS is hard. There's no other way to put it. And it can be extremely confusing, frustrating, and difficult due to all of the complexities. In this section, I will go through what you need to get up and running so your NuxtJS install can access your Laravel API. However, if CORS is new to you, then I'd highly recommend reading the How Cross-Origin Resource Sharing (CORS) Works section of this book where we dive in more depth and discuss how CORS works.

For both Laravel Sanctum and Laravel Passport, having CORS configured correctly is a MUST. Otherwise you will be getting errors and notifications everywhere and nothing will work.

What is CORS?

CORS stands for "Cross-Origin Resource Sharing" and according to [MDN CORS](#) is a:

"mechanism that uses additional HTTP headers to tell browsers to give a web application running at one origin, access to selected resources from a different origin. A web application executes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, or port) from its own."

Essentially what that means is that if you are on domain "mycoolapp.com" and are trying to pull resources from "yourcoolapp.com", "yourcoolapp.com" has to be configured to allow access through headers to "mycoolapp.com". A more practical example even stems if you have a subdomain where your API lives and you need to access that API.

We will be working with the app Roast which will have a front end that will live at <https://roastandbrew.coffee> and the API we are building will live at <https://api.roastandbrew.coffee>. If you don't have CORS configured correctly, you will

have nice little messages and un-fulfilled responses until your hair comes out.

Luckily, Laravel comes with some packages to get you through the struggle and get CORS set up efficiently. You are able to customize these packages to add more security (what CORS is actually made for, not frustration) to your install.

For this application, we will be using the [fruitcake/laravel-cors package](#). As of Laravel 8.x, this package is [baked into Laravel](#). Their documentation does an amazing job of explaining CORS as well, so looking at that might help

Let's get started!

Step 1: Install the CORS Package

To install the CORS package, open up the terminal for your API install and run the following:

```
composer require fruitcake/laravel-cors
```

This will install the package in Laravel. All we have to do now is publish the config and set our configuration variables. If you are on 8.x, you don't have to do this step.

Step 2: Publish Config and Add Middleware

This is all clearly documented in the package's [beautiful documentation](#). I'll just add supplementary information.

First, we need to add the middleware to the **end** of the middleware array in our [app/Http/Kernel.php](#) file:

```
protected $middleware = [
```

```
//... OTHER Middleware
\Fruitcake\Cors\HandleCors::class,
]
```

What this does is allow every request to the API to be validated, apply all of the security, then the CORS validations.

Now, we need to publish the config to our application. The package comes with a nice config file that can be used to manage how CORS operates on the API. In your terminal, type the following command:

```
php artisan vendor:publish --tag="cors"
```

This will copy over the CORS config for us to edit. The main thing we need to focus on is enabling CORS on our API. Let's open up the `config/cors.php` file.

In this file, there will be an array key called `paths` with an array to be accepted. We need to add the following into the `paths` array:

```
/*
 * You can enable CORS for 1 or multiple paths.
 * Example: ['api/*']
 */
'paths' => [
    'login',
    'register',
    'api/v1/*'
],
```

What this does is ensure that any route that is prefixed with `/api/v1/` will have CORS enabled, this is our entire API. We will now be able to access the API routes from our NuxtJS application with proper CORS headers configured and hopefully no errors!

The `/login` and `/register` routes are the authentication routes that we added in Building Secure Authentication API Endpoints. These will need proper CORS headers as well, so we make sure they are added in this file! We will add a few more to this array when we work on resetting passwords, but for now, we should be good to go!

In the next steps, as we configure Laravel Sanctum and we will open up another path in this file. For now though, CORS is set up for what we need it for.

Laravel Sanctum vs. Laravel Passport

There are two different approaches to authentication with your API for Laravel. One is [Laravel Passport](#) which has been around since 5.3. The other is [Laravel Sanctum](#) which just recently came out.

For our application, we are using both, but in very different ways. First I'll explain a high level overview of what each of the intended use cases are and then I'll explain why we are using both.

Laravel Passport

Laravel Passport is the full fledged OAuth server that integrates with Laravel. It provides functionality for personal access tokens, password grant tokens, refresh tokens, etc. Don't worry if you don't understand these now, we will go through these when we begin to open up our API to 3rd party developers. The main use case for Laravel Passport is to allow Laravel to be a full featured OAuth server for 1st and 3rd party integrations through your API.

Some say this is overkill since maybe you never need the power of the OAuth server with your application. For our needs, we will. Our goal is to allow some of the data from our application to be shared easily to 3rd party developers and follow API standards when sharing and securing this data. Eventually, we want 3rd party developers to be able to securely access data, depending on what their use case is. Any time you want 3rd party integration is a flag that you should look at using Laravel Passport.

The downfall with using Laravel Passport with your SPA is the authentication methods require you to store tokens somewhere. This could be an access token, `client_id` or `client_secret`, or any other secure token that could be cross site scripted. We don't want to expose these to potential hackers and according to the preferred practice from [auth0](#):

Browser local storage (or session storage) is not a secure place to store sensitive information. Any data stored there:

- Can be accessed through JavaScript.
- May be vulnerable to cross-site scripting.

If an attacker steals a token, they can gain access to and make requests to your API. Treat tokens like credit card numbers or passwords: don't store them in local storage.

So what do we do? Well, on mobile, we can store these tokens in secure storage which is a good option. We can also severely limit and protect data from 3rd party developers who try to access our data with their own Personal Access tokens. Granted, relying on all developers to implement security practices is NEVER a good idea, so make sure to take pre-cautions in what you return and implement token scopes to 3rd parties. That's where Laravel Sanctum comes in.

Laravel Sanctum

While you can do everything with Laravel Passport, Laravel Sanctum has a lot of advantages, especially within your SPA being on the same top level domain as your API. For example, in our application, we have <https://roastandbrew.coffee> and we have <https://api.roastandbrew.coffee>. They are on the same top level domain, so guess what we can use for auth? Cookies! This is what [auth0 recommends](#) if you have the luxury of working with the set up we are going through.

Laravel Sanctum comes from the need to securely authenticate your Single Page Application (SPA) with an API. This is honestly the trickiest part of this type of development. Since we are NOT loading the SPA from inside the Laravel install, it's even trickier. However, since we have the same top level domain for both of our applications, Laravel Sanctum makes this process much easier.

Laravel Sanctum allows you to generate authentication tokens or store a secure token easily within your SPA so you can access resources on the server. While you CAN do this with Laravel Passport, it's much trickier and it's really easy to mess up. For example, even the official NuxtJS documentation recommends sending the `client_id` and `client_secret` with your authentication request. This is a bad idea since NuxtJS is Javascript, any malicious activity can scan and grab these secrets.

There are two approaches you can use with Laravel Sanctum. The first approach is to generate a secure HTTP only cookie that gets set when the user successfully authenticates. This gets set in the web browser and allows access to secured routes when passed along correctly. The second approach, is you can actually generate tokens with Laravel Sanctum for use within mobile applications. This is very similar to a Password grant with Laravel Passport.

In our application, we are actually going to use both approaches. We will use the cookie based approach for web, the token based approach for mobile, but we will also be working with Passport for 3rd party applications. This brings us to our next section...

How we will integrate both

The most amazing feature of both of these packages? You can integrate both of them on a single API! This is amazing! We will be using Laravel Sanctum for our 1st party NuxtJS SPA and mobile application. However, as we allow 3rd party access to some of the data, we will have the larger engine of Laravel Passport provide proper security for 3rd party integrations. This amount of flexibility and security is amazing! If you looked at the Laravel Sanctum documentation, this process will be a piece of cake.

If this was dense and a lot to think about, don't worry, you can always come back and reference how we will be using these packages after we go through the configuration and further detail later on. Just know, each has their place and we will use both! We will first start with Laravel Sanctum so we can begin the

communication between our SPA and API. Later on, after we have some data and functionality, we will come back and install Laravel Passport to open up 3rd party platform access. Let's get started!

Installing and Configuring Laravel Sanctum

Let's take a look at Laravel Sanctum. I will be explaining any perspective shifts when looking at the implementation from an API Driven perspective. Along the way I'll include "optional" information if you want to differ from the authentication structure we are using. Remember, we will be using Laravel Sanctum for SPA authentication and the mobile application, along with Laravel Passport for 3rd party auth.

Step 1: Composer Require Laravel Sanctum

The first thing we need to do is install the composer package into our API so let's open up our terminal and run the following command:

```
composer require laravel/sanctum
```

Now we have the Laravel Sanctum package installed! We are ready to configure our customizations for this package and our API.

Step 2: Publish and Migrate Sanctum Database Tables and Config

Just like the documentation states, we will be publishing the Laravel Sanctum config, then migrating the table that stores the personal access tokens.

The first thing we need to do is publish the Laravel Sanctum config file and migrations by running this command:

```
php artisan vendor:publish --  
provider="Laravel\Sanctum\SanctumServiceProvider"
```

Upon completion, you will now have new migrations to store the `personal_access_tokens` that Laravel Sanctum will create for your users upon authentication. You will also have a config file that will allow for configuration of Laravel Sanctum.

Once both are published, run the command to migrate the databases:

```
php artisan migrate
```

You will now see your `personal_access_tokens` table created. This is where all of the personal access tokens will be stored.

The screenshot shows a database management interface for a MariaDB 10.3.23 database named 'roastandbrew_api'. The left sidebar lists various tables, and the 'personal_access_tokens' table is highlighted with a blue selection bar at the bottom of the list. The main pane displays the schema of the 'personal_access_tokens' table, which includes columns for id, tokenable_type, tokenable_id, name, token, abilities, last_used_at, created_at, and updated_at. The table currently contains 0 rows. At the bottom of the interface, there are buttons for Data, Structure, Row, Columns, and Filters.

id	tokenable_type	tokenable_id	name	token	abilities	last_used_at	created_at	updated_at

If you see this table in your database management application, you are ready to rock and roll!

Step 3: Configure domains

Since we are creating an SPA, we have to configure the proper domains from which Sanctum will receive requests from. These will be stateful requests.

Whoa! Aren't we building a RESTful API which should be stateless? Yes, but why not have both? Let's leverage the stateful security provided by Laravel Sanctum and the stateless security with 3rd party applications provided by Laravel Passport.

Let's open up the `/app/config/sanctum.php` file. This is where all of the Laravel Sanctum security configuration is stored. In here, we will need to add all domains (development, staging, production) that will need to have stateful requests that are authenticated by cookie with Laravel Sanctum. The newest version of Laravel Sanctum has you configure these domains as comma separated values in your `.env` file and then run the `explode()` method to split them into an array in your `sanctum.php` file:

```
explode(',', env('SANCTUM_STATEFUL_DOMAINS',  
'localhost,127.0.0.1')).
```

The `.env` key is `SANCTUM_STATEFUL_DOMAINS`.

So the next step depends a lot on your environment for development. For us, we have a top level testing environment locally that's `521.test`. While building this application, our development will be `roastandbrew-api.521.test` & `roastandbrew-frontend.521.test` with respect to how we have our application structured. When we deploy to production, we will have the structure be `roastandbrew.coffee` and `api.roastandbrew.coffee`. In our case, our `stateful` array should look like this:

```
'stateful' => [  
    'roastandbrew-frontend.521.test',  
    'roastandbrew.coffee'  
,
```

Make sure you have this configured correctly depending on your development and production environments or you will have a bunch of frustrating errors!

Step 4: Add Sanctum Middleware

Next, we will need to add the Sanctum middleware to our API group. To do this, open up the `app/Http/Kernel.php` file and add the following code to the `use` declarations at the top:

```
use
Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreSt
ateful;
```

Once you have that added, find the `api` middleware group and add the middleware we just included to the top of the array like this:

```
/**
 * The application's route middleware groups.
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::c
lass,
        \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' => [  
    EnsureFrontendRequestsAreStateful::class,  
    'throttle:60,1',  
    'bindings',  
,  
];
```

Perfect! We now have Laravel Sanctum installed and ready to rock and roll! Now we need to make it work with our authentication routes that we stubbed out and open up the specific request URLs with the Laravel CORS package we installed.

Step 5: Configure CORS to work with Sanctum

So when we set up our CORS package in the Implementing Cross-Origin Resource Sharing (CORS) section, we only added the `/api/v1/*` and `authentication/registration routes` to our CORS enabled routes. This was what we needed at the point, but now with Laravel Sanctum, we will have to open up another route.

The first route we will allow through CORS, is the route we need to set our `csrf` cookie so we can properly authenticate. By default, this route is `/sanctum/csrf-cookie`. It is extremely important to open up this route or we won't get a valid cookie that we can use for authentication.

If you look at the documentation for [Laravel Sanctum](#), this route must be called before you authenticate your user. The flow, as you will implement later on with the NuxtJS Auth module, is to first call this route to get a secure cookie. We then authenticate the user which sets the cookie as an authenticated user is present. Without calling this route first, we won't have the ability to store a secure cookie on our machine making authentication fail.

You can see the `/login`, `/logout`, and `/register` routes in here as well. These are authentication routes and definitely need to be accessible from our SPA.

Let's add that to the `paths` array within `app/config/cors.php` like so:

```
/*
 * You can enable CORS for 1 or multiple paths.
 * Example: ['api/*']
 */
'paths' => [
    'sanctum/csrf-cookie',
    'login',
    'logout',
    'register',
    'api/v1/*'
],
```

Now that we have that route added, we have one more thing to configure in our `app/config/cors.php` file and that's the `supports_credentials` flag. At the bottom of the file, you will see a flag that is `supports_credentials` and it's defaulted to `false`. We need to turn this on:

```
/*
 * Sets the Access-Control-Allow-Credentials header.
 */
'supports_credentials' => true,
```

This is the final setting we need in this file and it's crucial. It allows authentication requests to CORS enabled routes. Axios will require this header to be sent in order to send the `X-XSRF-TOKEN` cookie along with each request. We now have CORS configured, Laravel Sanctum installed, but have one last step to iron out. That's to make the session cookie scoped to our domain and use the cookie as the session driver.

Step 6: Configure Sessions

This is the final step in setting up Laravel Sanctum on the API side. When it comes to authentication and securing our API we will have a few more things to touch on.

For now, open up your `.env` file and adjust the setting for `SESSION_DRIVER` to:

```
SESSION_DRIVER=cookie
```

This ensures Laravel is using our cookie to check for authentication. Next, we need to configure our session domain to work with the SPA. To do this, add a value to the `.env` file and set up your SPA/API domain like so:

```
SESSION_DOMAIN=.roastandbrew.coffee
```

Of course, use the proper domain for where you will be hosting your application. Notice the `.` before `.roastandbrew.coffee`? This opens up the session domain to sub domains. When you push to production, ensure you set that appropriately so correct authentication can be made. Like I mentioned before our API will sit at <https://api.roastandbrew.coffee> and the SPA will be at <https://roastandbrew.coffee>.

Now that we have Laravel Sanctum installed, it's time to set up our user registration and authentication routes!

Implementing Authentication & Registration With Laravel Sanctum

So, Where Are We At?

Well, we've stubbed out some authentication routes, essentially creating buckets we need to implement:

Building Secure Authentication API Endpoints

Next, we implemented and set up CORS so we can access those routes from our API:

Implementing Cross-Origin Resource Sharing (CORS)

Finally, we installed Laravel Sanctum and got it all ready to accept requests for authentication:

Installing and Configuring Laravel Sanctum

Now what we are going to do is implement the functionality to handle the registration of a user, the authentication of a user, and the logging out of the user. In this section, we will go through all of the implementation from the API side so when we get to the NuxtJS side, we can focus on the permissions and set up there.

Quick side note that you will see when we begin working with our API, this is similar to the flow that we use for shelling out smaller features. We first begin with the database, then do the modeling, add the routes, and implement the logic on the API side. Then we dive into the frontend side, load the data, display the data, and perform actions on the data. There will be a ton more on this as we develop, but get used to the flow! It makes the process really easy!

Step 0: Plan out what data is needed for a user

Before we get into our registration and authentication, we need to plan the data to capture to successfully register and log in a user. This is for a standard `email + password` authentication system, NOT a social auth system. We will work with that in a whole different section.

For our app, we will keep it extremely basic and require these 3 fields to register:

- Name
- Email
- Password

That's it! We will end up storing more information about our users as the app grows, but to create a user, that will be the data we need to collect.

Step 1: Implement the /register route

This will be our first step since we have to have users to login, before we can log them in! To do this, let's re-open the `app\Http\Controllers\Auth\AuthController.php`. Remember, this is where the buckets for our methods live.

Let's find the `register()` method and implement it with the following code:

```
public function register( Request $request )
{
    $validator = Validator::make( $request->all(), [
        'name' => 'required|string|max:255',
        'email' => 'required|string|email|max:255|
unique:users',
        'password' => 'required|string|min:6',
        'confirm_password' => 'required|same:password'
    ]);

    if( !$validator->fails() ){
        $user = User::create([
            'name' => $request->get('name'),
            'email' => $request->get('email'),
            'password' => bcrypt( $request-
>get('password') ),
            'avatar' => $gravatar->get()
        ]);
    }

    return response()->json( '', 204 );
}

return response()->json([
    'success' => false,
    'errors' => $validator->errors()
], 422);
```

```
}
```

And while we are at it, add the following classes and facades to your `use` section of the controller:

```
use App\Models\User;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Validator;
use Auth;
```

Well, that's a lot of code! Let's take a step through it and make sure it all makes sense.

Quick tip: If you were to look at the stubbed out registration controller that comes with Laravel UI, it looks practically the same. That's because we want it that way. With an API we don't submit our forms like we would through a standard "call and response" type application or "monolith". It isn't wrong, it's just different. We want to keep our application's functionality similar, we just have to implement it in a different way which we are walking through right now.

The Validator

The first thing we need to do create some validations on the request. If you haven't used Laravel's validation system before, I highly recommend checking it out and some of the [cool rules you can implement extremely easily](#).

What the validator does is allows us to pass in our request as the first parameter and a list of rules that the variables in the request array follow. You can chain together validation rules for each variable in the array. For example, if you look at the `name` validation you see `required|string|max:255`. That's a list of validations that the `name` field must resolve to for the validation to return true.

You will also see formats like `max:255` or `min:6`. Those are additional parameters for the individual validation that must pass in order for the validation to be correct. In these two examples, `max:255` means that the parameter must be less than `255`

characters in length. In `min:6` means the variable must be greater than **6** characters in length.

Let's break down our validator.

```
$validator = Validator::make( $request->all(), [
    'name' => 'required|string|max:255',
    'email' => 'required|string|email|max:255|
unique:users',
    'password' => 'required|string|min:6',
    'confirm_password' => 'required|same:password'
]);
```

In order for our validations to pass and a user to be created, all of the parameters and validations associated with them must be true.

- name → Must be present (required), be a string, and be less than 255 characters
- email → Must be present (required), be a string, match the regex for an email, be less than 255 characters and be unique on the `users` database table.
- password → Must be present (required), be a string, and be greater than 6 characters.
- confirm_password → Must be present (required) and match password.

As a quick side note, there will be multiple ways to do security (middleware, gates/policies, and validators). We will walk through how to use all of them and when to use all of them here:

Permissions, Validations, and Security

So now, we have a validator object and we can check to see if our form request's array passes that validation with the following code:

```
if( !$validator->fails() ){  
}  
}
```

If we have a valid input array, we go to the next step and create a user in the database with the information sent by the user wishing to register. If the validator fails, we return a `json` response with a `422` response code, a `boolean` flag of `false` and an array of errors.

```
return response()->json( [  
    'success' => false,  
    'errors' => $validator->errors()  
, 422);
```

One of the many awesome features of the validator? Custom responses telling exactly what failed. Our `errors` key will contain an array of issues the validator had in a beautiful sentence format. We can even override the default messages to provide a better experience for the user. Right now, we will continue to use what's provided but touch on the message overrides later. We also return a `422` response which is an `Unprocessable Entity` response. This simply means a validation failed along the way and we can handle this within our SPA, prompting the user to retry the registration.

Using proper response codes is extremely important when developing your API. It allows all developers to have better insight into why their request failed or how it succeeded.

The Creation of the User

Let's say the validator passed and we continue registering the user. The first thing we need to do is create our user:

```
$user = User::create([  
    'name' => $request->get('name'),  
    'email' => $request->get('email'),
```

```
'password' => bcrypt( $request->get('password') )  
]);
```

What this will do is use our `User` model and create a new user with a secure `Hash` of the password using the `bcrypt` method. Now that we have a user created, that user can authenticate. That will be handled through the `/login` route. All we have to do at this point is return an empty JSON response with a `204` HTTP code (which means empty response):

```
return response()->json('', 204);
```

When we receive this successful response on the front end, we can prompt the user to authenticate or even do it automatically. Depends on how you want to handle this within your application.

Now we have a user we can authenticate! Next up is the login method!

Step 2: Implement the `/login` route

Now that we have the ability to create users in our application, we can authenticate them. This route will be MUCH easier to implement and only a few lines of code. Since we already have the `Auth\AuthController.php` open, let's find the `login()` method and add the following code:

```
public function login( Request $request ){  
    if (Auth::attempt([  
        'email' => $request->get('email'),  
        'password' => $request->get('password')  
    ])) {  
        return response()->json('', 204);  
    }else{  
        return response()->json([  
            'error' => 'invalid_credentials'  
        ], 403);  
    }  
}
```

```
}
```

This is a much smaller method which is nice, but let's dive in and break it down a bit. The first thing we do, is simply attempt to log the user in with:

```
if (Auth::attempt([
    'email' => $request->get('email'),
    'password' => $request->get('password')
]))
```

We are sending the `email` and `password` with the request and running it through the `Auth::attempt` method provided by Laravel. If that passes, we will have an authenticated session and return an empty response. From here on out, we will rely on the cookie for authentication.

If the authentication fails, we will return a JSON response with an error and an HTTP code of `403` which is meant for `Forbidden` since the auth was rejected.

```
return response()->json([
    'error' => 'invalid_credentials'
], 403);
```

The JSON response with `204` code and error response with `403` code are two of the main reasons why we are not using the Laravel default authentication scaffolding. Even though most of it's the same code, by default the scaffolding tries to redirect. We aren't re-directing since we are coming from an SPA.

That's all we need to do with Sanctum authentication working! We will tie this all together when we implement the frontend side. For now, we have implemented the necessary routes to provide authentication for our users.

Step 3: Implement the /logout route

This is the last route we will be implement while in the `Auth\AuthController.php` and this will un-authenticate the user. It's really

simple and only 2 lines of code. First, find the `logout()` method we set up and add the following code:

```
public function logout( Request $request ){
    Auth::logout();

    return response()->json([], 204);
}
```

All we need to do call the `Auth::logout()` method provided by Laravel and then return an empty response with a `204` HTTP code. This will cancel the session and log out the user! For now, these are all of the authentication routes necessary for your application! Next time we touch on these routes, we will be creating a mobile app!

Step 4: Implement the /api/v1/user route

So now that we have a way to register a user, authenticate a user, and log out the user, let's implement one more route that will return the authenticated user. This is extremely important when working with NuxtJS.

We will be providing this route to NuxtJS to grab the authenticated user and store it locally in a Vuex store. Nuxt will then keep track of this user in the local state and allow the javascript application to access this user. Nuxt will also allow us to implement client side middleware to make UX decisions and allow access within our SPA.

Before we start this implementation, I want to point out a key difference when doing API Driven Development vs standard call and response development. Say we needed to inject user data into a PHP rendered page that's returned to the browser. We can simply load a user from the database and inject its parameters into the page (using a templating system like blade or straight PHP), return the page, and as long as everything is set up correctly, nothing extremely valuable should be leaked.

This is NOT the case with API Driven Development. When you send back a `User` object, you pass the object to the frontend as JSON through a web request. This is easily accessed by the client side and can be snooped for sensitive information. We really want to make sure that whatever we return from the server to the client is data that can be visible from the user viewing the page.

For more information on hiding sensitive data, check out this page:

Securing Sensitive Data

For now, let's open up the `app\Http\Controllers\API\UsersController.php` and add the following code to the `show()` method:

```
public function show( Request $request ){
    return response()->json( $request->user() );
}
```

What this does is simply return the authenticated user from the attached to the request. If there is no user that's authenticated, we should get a `401 Unauthenticated` response. This allows the Nuxt Auth module to determine if we can allow the user to log in or what permissions the user has.

The only other line we have to add is:

```
use Illuminate\Http\Request;
```

to the top of our controller. This allows us to use the `Request` object that's injected into our method.

When they do end up authenticating, this will return the `User` object as JSON.

To ensure we get the proper `401 Unauthenticated` response if the user is not authenticated, we should block off this route with some middleware to block access if there isn't an authenticated user. This is the first time we will be applying any sort of middleware to a route. I'll simply show it here, but we will be providing much more for examples later on.

What we need to do is in the `app\Http\Controllers\API\UsersController.php` file, we need to look at the `_construct()` method. In the method, we need to apply the following code:

```
public function __construct(){
    $this->middleware('auth:sanctum')->only('show');
}
```

What this does is apply the `auth:sanctum` middleware to the `show` method (the method we are using to return the user). If the user requesting the route is not authenticated, it will return a 401 response code. This will be similar functionality for all routes that require authentication. I like providing the middleware in the controller, through the `_construct()` method. Reason being, it makes our routes files (both `api` and `web`) clean and easy to read. It also groups functionality since a lot of middleware is dependent upon a resource controller.

As of right now, this is what we need to implement! We now have a fully functioning authentication system with Laravel Sanctum. Once we get Nuxt set up correctly, we will call these routes and get ready to add functionality!

Writing Authentication Tests

Testing your API can be tricky by default. Writing automated tests for your API with Laravel Sanctum adds a little bit of complexity. Here are a few tips to make sure the process goes smooth.

Create a Directory for Your Authentication Tests

For all of my tests, I try to group them as easily as possible and make them as verbose as possible. When grouping, we already set up the /tests/API directory. In that directory, I'd recommend creating an Authentication directory (/tests/API/Authentication) and placing all of your authentication and registration tests within it. This way you can scope the running of your tests using the following command:

```
php phpunit tests/API/Authentication
```

Now you can save time if you need to just ensure your authentication functionality is still passing! You won't have to run the entire test suite!

Use WithoutMiddleware Trait

To be honest, this tripped me up right away when writing API authentication tests. Especially when using Laravel Sanctum. Since you need to get a secure cookie to make requests with Laravel Sanctum you are essentially making a call to get a CSRF Token. That requires a request which returns a cookie back to you upon completion.

When we are testing, this is a little harder to accomplish especially without the use of a browser based test. You will get a load of [419](#) responses which are invalid CSRF tokens. To get around this, ensure your testing suite does not check the middleware.

Let's say you have a RegisterUserTest.php file and in that file. Ensure that you add the following to your use declarations:

```
use Illuminate\Foundation\Testing\WithoutMiddleware;
```

Then make sure you are using the trait on the class:

```
use Illuminate\Foundation\Testing\WithoutMiddleware;

class RegisterUserTest extends TestCase
{
    use RefreshDatabase;
    use WithoutMiddleware;
}
```

Now you can test any of your functionality where you are not focused on the middleware being a part of the test.

On some API routes you don't want to disable this middleware because the test should validate if the middleware is working correctly. Just keep that in mind when writing your tests!

These are a few of the specific automated test cases to authentication. In the appendix, I'll have more testing gotchas when dealing with API testing.

SET UP NUXTJS TO PROPERLY AUTHENTICATE WITH YOUR API

Configuring NuxtJS Auth Module + Laravel Sanctum

So this will be one of the bigger pieces of setting up your SPA with Nuxt. Making sure your application is secure and can access/submit to locked down routes on your API can be extremely difficult but is the single most important piece of your application. Without proper authentication, you either have an insecure app, a bunch of frustration with [401](#) responses, or your app probably doesn't do much. All 3 of these scenarios are bad.

Luckily NuxtJS has a first class [Auth Module](#) that takes the pain out of a lot of these scenarios. It can store tokens, cookies, keep track of the authenticated user, and set up the routes necessary to submit data to the API for authentication. We will be using that exclusively with both of our authentication structures (Laravel Sanctum cookie on web and Laravel Sanctum token on mobile).

One thing to note is our SPA does not require authentication to function. Users will be able to view our app and access some of the features without logging in. Being authenticated will allow the user to submit data to our API, access restricted data, and save features to their profile. Once we get basic authentication configured and working with our API, then we will explore this more as we add features to our application.

Step 1: Install the NuxtJS Auth Module

The first thing we need to do is install the NuxtJS auth module. To do that, you first need to open a terminal in your [/{app}/frontend](#) directory and run the following command:

```
npm install @nuxtjs/auth-next
```

If you are following the NuxtJS auth module documentation, they mention installing Axios here. We've already installed axios when we created our install, so

we just need the auth module. Finally, we will need to open up the `nuxt.config.js` file and add it right below our `@nuxtjs/dotenv` package in the `modules` array:

```
/*
 ** Nuxt.js modules
 */
modules: [
    // Doc: https://axios.nuxtjs.org/usage
    '@nuxtjs/axios',
    // Doc: https://github.com/nuxt-community/dotenv-module
    '@nuxtjs/dotenv',
    '@nuxtjs/auth-next'
],
```

Now we have the authentication module installed and ready to rock and roll! Next, we will configure the module with the routes that we have for authentication.

Step 2: Configure the NuxtJS Auth Module

The first thing we need to do is open up `nuxt.config.js` and make an `auth` key initialized to an empty object.

```
...
auth: {
},
...
```

We will also shell out some of the implementation for the routes that are supported by the module so when it comes time to implement with Laravel, we just have to configure the changes.

The way the Nuxt auth module works is it allows you to configure different authentication strategies within your application. These can be social, cookie,

token, etc.

Step 3: Configure the Laravel Sanctum strategy

With the Auth Next module, there is Laravel Sanctum support right out of the box which makes our lives insanely easy. All we have to do is configure the endpoints used by the module for certain methods and we are good to go!

As we get to mobile, we will swap some of this around since we use tokens, but for now, don't worry about it. We will get to it when we get to mobile compilation.

For now, let's stay in our `nuxt.config.js` and add the following parameters:

```
...
auth: {
  redirect: {
    login: '/',
    home: false
  },
  strategies: {
    'laravelSanctum': {
      provider: 'laravel/sanctum',
      url: process.env.API_BASE_URL,
      endpoints: {
        login: {
          url: '/login'
        },
        user: {
          url: '/api/v1/user'
        },
        logout: {
          url: '/logout'
        }
      }
    }
  }
},  
...
```

So there's a lot to break down in this JSON and we will be adding to it as we go. Let's just start with this and we will expand further. Essentially what we are looking at is 3 URLs that comprise the authentication functionality. They should look familiar in the fact that we implemented them in Using Laravel Sanctum for API Authentication.

Now we are at the point to implement them within Nuxt. Each strategy allows you to define these endpoints so Nuxt knows what to call when the user wants to perform a specific action.

Redirect

Right away in our auth module we have a redirect key. This determines how we want our redirection to happen when a user logs in and logs out. Within this redirect object, we've defined two keys, login and home. There are a few more you can define that are listed in the [documentation](#).

The login key handles the redirect for unauthenticated users. Essentially if a user is not authenticated and they try to access a path that is blocked by authentication, they will be redirected to this route. For now we have this set to `/` which is home. We will try to catch when the user wants to perform an action and save that as a pre-auth action so we can redirect correctly when they log in.

For example, if the user tries to edit a company but is not logged in, we will handle that by saving it to Vuex state, redirect to the home page with a setting to display the login modal, have the user submit their authentication, then redirect back to the page they were editing.

With that being said, the second key of `home` goes hand in hand. We set `home` to `false` because that key determines if we should redirect the user back to the home page after authentication or not. Since we want to handle this ourselves, we need to flag this as `false`.

When we dive more into permissions and and middleware, we will be setting up our redirects and handling them gracefully. When we get there, I just wanted to

point our our current set up.

URL

This is url that is the base for all API interactions. Configured in the env file we set up, this URL allows us to use relative URLs for each of the methods we define. In our `.env` file we have an `API_BASE_URL` set to <https://roastandbrew-api.521.test>. When we are on production, this will change, but having this configured through an `.env` file is extremely efficient. Setting it in the authentication module configures it for the entire strategy.

Login Endpoint

Let's start with the `login` endpoint. If you look under the `LaravelSanctum` key, there is an `endpoints` key. Here is where you can overwrite the endpoints you need.

This endpoint defines what we should send to the server and how it should hit the server. In this case, we want to hit the `/login` URL with our credentials through the POST HTTP method. Right now, this is all we need to do.

Just note, that the Nuxt auth module makes use of some of the defaults provided in the Nuxt Axios module. This is why we can do a relative URL for `/login` since it wil take in place the `baseURL` we defined in Abstracting Your API Calls Into Reusable Modules

When we implement our authentication component, we will be using a method provided by the NuxtJS auth module to authenticate with this URL and through this method.

Logout Endpoint

Similar to the `login` endpoint, the `logout` endpoint allows us to define a URL to hit to un-authenticate the user and what HTTP method we should use. In our case, we want to send a POST request to the `/logout` endpoint that we

implemented in Building Secure Authentication API Endpoints.

The authentication module will know to send a request via POST to this URL when we call the method to logout a user.

User Endpoint

This is our FIRST API Endpoint we will be hitting with our SPA! This is a special endpoint used by the Nuxt auth module to grab the authenticated user. Nuxt uses this route to determine if a user is authenticated and save response from the route as the user. This will allow us to use middleware that references an authenticated user and access the user through `this.$auth.user` when authenticated.

The Laravel Sanctum authentication module with NuxtJS knows to send the proper headers to this route (which we set up when we added the `credentials: true` to our Axios config). Unlike other authentication modules, this module knows that the user route returns a straight user. What that means is we aren't returning an object with a `user` key and having to grab the authenticated user from there.

With the first class support for Laravel Sanctum baked into the authentication module, that's all we need to do for now! When we implement mobile, we will be using the local strategy which is slightly different, but we will go through that when we get there.

Step 4: Initialize Vuex

This is unique with NuxtJS and, as you will see, a lot of how NuxtJS works is by monitoring folders and generating what needs to be generated. When installing NuxtJS, Vuex is installed by default which is awesome!

Real quick, Vuex is a local data store used for storing data that can be accessed between components. We will want our authentication module to store our authenticated user in this local store so we can keep track of what user is authenticated and use the user object in pages and components.

For now, we are just need to initialize Vuex so NuxtJS knows we want to use it and we want to store our authenticated user in the state provided. All we need to do is open the `/store` directory and add a blank `index.js` file. We will do a lot more with Vuex but for now, just add this blank file so our Nuxt Auth module can make use of the Vuex store.

Step 5: Using Auth Middleware

Like Laravel on the server side API, NuxtJS ships with its own capability to block routes through middleware. And you know what's awesome? It's super easy to create and apply. We will definitely be making use of both server and client side middleware.

By default, the NuxtJS module ships with an `auth` middleware. If the user is not authenticated, we will redirect them to the `/Login` page. For more information on this middleware check out [NuxtJS middleware documentation](#).

For example, if you wanted to block a page by if a user is authenticated or not, I'd add the following key to your page component:

```
export default {
  middleware: 'auth'
}
```

That's it! We will dive into middleware on both server and client side much later as we develop more advanced permission structures. We will be creating our own custom middleware as well and adding multiple pieces of middleware to certain pages.

For now, we are done with configuration! Let's move on to the next step and add some components to our application that allow the user to log in!

Building Authentication & Registration Components

Before we dive into actually registering and authenticating users, I'd like to explain how we will structure our front end authentication system. This will be the first time we touch on user interfaces so far. As I mentioned before, I will explain all of our functionality as being app agnostic. With the case of authentication, it's pretty much going to be the same. The styles will change, but that's about it. I won't be covering designs in this book at all. You will see screenshots of the app and some of its functionality, but color, layout, flow, animations, none of that will be covered in this book unless it specifically relates to functionality.

When building the front end authentication system, there are 2 components we have to build. They are the `Login` and `Register` components which will live at the front end of our app at `/components/auth/Login.vue` and `/components/auth/Register.vue`.

These components will have to be implemented on the frontend and work with our API through remote calls with the Auth Module.. We will go through the process of adding these components within Nuxt then sending that information to be processed by our API. Let's get started!

Creating a Registration Component and Registering a User

Well, it's time to start adding and authenticating our users! Luckily, we have more than half of the process ready to go since we already implemented our Laravel API and configured our authentication module.

The way that we will go through this is I'll design a simple component with the functionality necessary to perform a registration. If you are interested in the actual UI/UX, we are implementing this as a modal. You can check out the [source code](#) to see the modal styles, layout, and visual side if you want!

Let's dive in!

Step 1: Add Your Registration Component

All we have to do is navigate to the `/components/auth` directory and add a Vue component titled `register.vue`. You can put these components wherever you like, for me this makes sense as the best structure. Right now, I just have this as an empty component:

```
<template>
  <div id="register">

  </div>
</template>

<script>
export default {

}
</script>
```

Step 2: Plan the data we need

It's always good to start with a plan! Even with something as small as user registration. First, let's think about the data that we need. When we implemented Laravel Sanctum (Implementing Authentication and Registration With Laravel Sanctum), we discussed that in order register a user, we would simply need the following fields

- Name
- Email
- Password

However, in the spirit of good UX and security, we should validate the password field, so we will need a "Confirm Password" field as well.

Let's get this added to our local `data()` within our component:

```
data(){
    return {
        form: {
            name: '',
            email: '',
            password: '',
            confirm_password: ''
        }
    },
},
```

The one thing I'd like to point out with this, is we have all of our pieces of data nested within a `form` object. This is preferential to how I like to manage data that's grouped. It makes it easier to read and easier to submit to the server. We just have to pass `$this.form` as you will see when we submit our registration form to the server.

Having the form submitted is one thing, validating it is another. I always like to add a validation object to the local `data()` as well. This will allow us to show data based off of what fields are valid and which ones are not. To do this, I add a `validation` key that looks like this to the data:

```
data(){
    return {
        form: {
            name: '',
            email: '',
            password: '',
            confirm_password: ''
        },
        validations: {
            name: {
                valid: true,
                message: ''
            },
            email: {
                valid: true,
                message: ''
            },
            password: {
                valid: true,
                message: ''
            },
            confirm_password: {
                valid: true,
                message: ''
            }
        }
    }
},
```

Let's break it down. We have a `validations` object. For each piece of data in our

`form`, we have a validation assigned to it. This allows us to house certain pieces of information regarding that piece of data. Those pieces of information are `valid` and `message`.

The `valid` key on each piece of data is a boolean. It keeps state on whether the piece of data is valid or not. The `message` corresponds to a `false` validation. We can specifically tell the user what we need to do to make the data valid.

For example, the `email` field has two different validations:

1. Has it been entered.
2. Is it a valid email address

When we get to this validation method, we will populate this message field with the data that should be displayed. This is just a good practice to get into when working with forms. It helps really work with the UI and even offers a place to dump server side validations when they come into play.

Next, we will work with the methods that are necessary to make this registration system work.

Step 3: Add the methods we need

So with registration, we have a few methods that we have to custom implement. Yes, we have the Nuxt Auth module installed, but that only handles the actual authentication. We need to register some users before we authenticate them.

The actual registration takes place in only 1 method, but I'd recommend that we validate the form client side before submitting it to the server. The reason I like client side validation with all of our functionality before server side is usually you can catch an error and alert the user before they make a request. This is beneficial for the user as they can change the incorrect data faster and beneficial to the hosting of the app since it doesn't have to process requests it doesn't have to.

To do this, we will need only two methods in our Vue (page) component:

- `register()`
- `validateRegistration()`

Step 4: Validate the Registration

Let's add our `validateRegistration()` method first. Considering this is how I'd recommend the actual flow of the application registration process to go, I'm going to show exactly the code I have. As we dive into more features that are "app specific", the code I show will be more conceptual. Our `validateRegistration()` method should look like:

```
validateRegistration(){
    if( this.form.name == '' ){
        this.validations.name.valid = false;
        this.validations.name.message = 'A name is required
on this field'
    }else{
        this.validations.name.valid = true;
        this.validations.name.message = '';
    }

    if( this.form.email == ''
        || !this.form.email.match(/^(([^<>()[]\\.,;:\\s@\\"]+
(\. [^<>()[]\\.,;:\\s@\\"]+)*|(\".+\"))@((\\[[0-9]{1,3}\].
[0-9]{1,3}\. [0-9]{1,3}\. [0-9]{1,3}\])|(([a-zA-Z\-\_0-9]+\.
[a-zA-Z]{2,}))$/ ) {
        this.validations.email.valid = false;
        this.validations.email.message = 'A valid email
address has to be entered to register'
    }else{
        this.validations.email.valid = true;
        this.validations.email.message = '';
    }

    if( this.form.password == ''
```

```
    || this.passwordStrength.score < 4 ){
      this.validations.password.valid = false;
      this.validations.password.message = 'A secure
password must be entered.'
    }else{
      this.validations.password.valid = true;
      this.validations.password.message = '';
    }

    if( this.form.confirm_password == ''
    || this.form.confirm_password != 
this.form.password ){
      this.validations.confirm_password.valid =
false;
      this.validations.confirm_password.message =
'Your passwords must match to register';
    }else{
      this.validations.confirm_password.valid = true;
      this.validations.confirm_password.message = '';
    }

    return (
      this.validations.name.valid &&
      this.validations.email.valid &&
      this.validations.password.valid &&
      this.validations.confirm_password.valid
    ) ? true : false;
}
```

The method looks dense, but really it just checks to make sure the entered parameters match the requirements and returns `true` if they do and `false` if they don't. These are all validations that can be done on the client side. Validations like "email address has to be unique" can only be done server side, but the way we structured our validation system, we can populate the messages with the server side responses and use the same structure. Combine that with VueJS' reactive

properties and you have a pretty nice validation structure!

The validations we are checking for are:

1. Name

- Must be entered

2. Email

- Must be entered
- Must be a valid email

3. Password

- Must be entered
- Must have a score equal to 4 which is considered a highly secure password by zxcvbn.

4. Confirm Password

- Must be entered
- Must be the same as the password field

If all of these fields pass, we return `true` which we will pass to the `register()` function in the next part. If the method returns `false`, we will stop the registration process and allow the user to re-enter their form field.

One feature I'd like to point out is we use [zxcvbn](#) in to validate our password. This is a library built by Dropbox that gauges the security of a password entered. If you are interested in the full set up, check out our [Server Side Up](#) article on how to implement it. Essentially what it does is validates how random and hard to guess your password is based on a series of parameters. It then returns a score between

1 and 4 with 1 being the lowest and 4 being the highest (best) score.

Another benefit to this validation object is, since we are using Vue, we can also optionally bind classes to the HTML elements. This allows us to highlight the invalid input in `red` or do whatever we want. We can also optionally show and hide validations based on whether the field is valid or not.

Now that we have our validation method taken care of, let's implement the actual `register()` method.

Step 5: Submitting the Registration Form

So we now have the method set up to validate the user's registration. Now it's time to submit the actual form to our server. Before we implement our method, let's talk through what is going to happen.

First, we are going to see if our form is valid. If our form is valid, we will continue with the registration process. If the form is not valid, we will display any of the validations and allow the user to attempt to make changes.

Once the form is valid, we will grab a CSRF token from Laravel Sanctum. Since we are using Laravel Sanctum for authentication, our end game is to authenticate the user. EVEN though we are registering them, we don't want to register and then prompt them for log in if everything validates, we want them to be authenticated and ready to use the app right after validation.

After we receive our CSRF token, we will submit our form to the `/register` route where everything will validate. Upon a successful validation, we will submit the `email` and `password` through the `loginWith` method and our `laravelSanctum` strategy to log in the user. Barring any completely unforeseen circumstances, this will be a successful login. I mean, we just created a successful user and used the exact same email and password to authenticate them.

This is where the NuxtJS Auth module takes over. On a successful login, it automatically calls our `/api/v1/user` route to load the authenticated user into

local storage.

There's a lot happening in this method that's for sure, so let's take a look at it:

```
register() {
    if( this.validateRegistration() ){
        this.$axios.get('/sanctum/csrf-cookie')
            .then( function(){
                this.$axios.post('/register', this.form )
                    .then(function( response ){
                        this.$auth.loginWith( 'laravelSanctum', { data: this.form } )
                            .then( function(){
                                // ... Handle success
                            }.bind(this));
                        }.bind(this))
                    .catch( function( error ){
                        // ... Handle failure and show
                        validation messages
                    }.bind(this));
                }.bind(this));
    }
},
```

The first thing we do is validate the registration and wrap the rest of the functionality with:

```
if( this.validateRegistration() ){

}
```

Upon validation, we then get our CSRF Cookie from sanctum and wait for a successful completion in the `.then()` promise:

```
this.$axios.get('/sanctum/csrf-cookie')
    .then( function(){

}
```

```
}.bind(this));
```

When we have our cookie, we submit the form to the `/register` endpoint and listen for a successful completion with that request's `.then()` promise:

```
this.$axios.post('/register', this.form)
  .then(function( response ){
    //... Handle successful registration
  }).bind(this)
  .catch( function( error ){
    //... Display registration errors.
  }).bind(this));
```

Two things to notice about this method. The first, is we are not calling this through the API abstraction that we created.

This is correct, since technically the `/register` route is not part of our API. In a typical RESTful structure, this would be considered our "authentication" server, granted with Laravel Sanctum setting cookies, the requests to our API are "stateful".

Second, we submit the entire form using `this.form`. This sends all of the data in our local form object to the server to create a user if everything is correct. I love setting up the form submissions this way since we can just pass a single variable that's already created. It really cleans up the code.

Now that we have this set up, we can handle both an unsuccessful registration or a successful registration using the `.then()` and `.catch()` on the returned promise.

In the next step, we handle a successful registration and submit the entire form for authentication. In turn, this submits all of the data we housed in the local `form` data field. This is super convenient and really cleans up the code to have the local data monitored within the `form` object.

Step 6: Handling a Successful Registration

Upon a successful response, we then use the NuxtJS auth module to `loginWith` the `laravelSanctum` strategy:

```
this.$auth.loginWith( 'laravelSanctum', { data:  
  this.form } )  
  .then( function(){  
    // ... Handle successful authentication  
  }.bind(this));
```

This is what triggers the rest of the authentication. The NuxtJS auth module will authenticate with the /login route we defined, passing the data from the form which will be the same email and password for the user we just registered.

There are two things to note. First, we pass the form by the data key to the `loginWith()` method. The form will contain the email and password that the NuxtJS auth module will send to the server.

Second, this request is also returns a javascript promise! That means we can handle the request using `.then()` callback.

Upon successful login, the `/api/v1/user` route will be called, loading the authenticated user, setting the user in Vuex and redirecting back to the / page unless the user was trying to perform an action. Then the user will be redirected back to the page that they were initially on. We will touch on how to do that later!

Step 7: Handling an Un-Successful Registration

An un-successful registration can happen for a variety of reasons such as an email address is already taken or there was invalid data submitted. For whatever reason, we want to handle this response gracefully and display to the user the message in the form of a validation.

To do this, we will listen to the `.catch()` on the promise when we submit the data to the `/register` route. This will allow us to get any of the errors from the API and display them to the user.

Our `.catch()` function should look like this:

```
.catch( function( error ){
    //... Display errors from the API.
    // They will be in the error.response.data.errors
array!
}.bind(this));
```

Any error returned from the API will be in the `error.response.data.errors` array. We can use this array to set the `message` key on whatever field failed to validate. Pretty sweet!

There was a ton going on in that section, but really, this is it for registration! Next we will build the login component, which will be much easier to implement, and with a few less requests and we will be ready to start adding features!

Creating a Login Component and Authenticating a User

We now can register our users through our SPA to our API, now it's time to just authenticate existing users. I don't want to spoil the fun, but we've pretty much already have done all of this in our registration route so implementing this will be easy!

Step 1: Create the Login Page

The first thing we need to do is create a login component. To do that, add a component to the `/components/auth` directory named `Login.vue`. The file should contain the basic Vue component template:

```
<template>
  <div>

    </div>
</template>

<script>
export default {

}
</script>
```

You will now be able to include the component wherever you like! For us, we consider this a global component since we should be able to activate it at any time. Because of this, we include this component in our `/layouts/App.vue`.

Step 2: Plan the data needed

Similar to the registration component, we will need to plan the data we need. To

authenticate the user, we simply need the `email` and `password` fields, so add them to your page's `data()` variables:

```
export default {
  data(){
    return {
      form: {
        email: '',
        password: ''
      }
    },
  }
}
```

Once again, these are accessible through the `this.form` locally which makes cleaner calls to the `/login` route.

We will also have to add our validations so we can ensure the user has entered something to authenticate. For our purposes, we are just adding the following validations:

```
export default {
  data(){
    return {
      form: {
        email: '',
        password: ''
      },
      validations: {
        email: {
          valid: true,
          message: ''
        },
        password: {

```

```
        valid: true,
        message: ''
    },
    invalidLogin: {
        valid: true,
        message: ''
    }
}
},
}
```

This will ensure that the user at least enters an email and password before submitting. These will be validated the same way as the registration form.

If you look at the `validations` key, you will see an `invalidLogin` key. Since the server doesn't return which individual field was incorrect (for security reasons) this will just show the message that was returned from the server.

Next up, we will implement the `login()` and `validateLogin()` methods to finish the authentication with our API.

Step 3: Add the authentication methods

To authenticate the user, we will need to add two methods. The first one we will implement will be the `validateLogin()` method. This will not be as complex as the registration validation since we only need to ensure the user has entered an email and a password.

The second method we need to implement will be the actual `login()` method. This will have a few steps, but less than the `register()` method.

Step 4: Implement the Validate Login Method

Let's start here. The first thing we need to do is add the `validateLogin()` method to the component like this:

```
validateLogin(){
    if( this.form.email == ''
        || !this.form.email.match(/^(([^<>()[]\\.,;:\\s@\\"]+
        +(\.[^<>()[]\\.,;:\\s@\\"]+)*|(\\".+\\")@((\\[[0-9]{1,3}\].
        [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-\0-9]+\.
        [a-zA-Z]{2,}))$)/ ){
        this.validations.email.valid = false;
        this.validations.email.message = 'A valid email
address has to be entered to login'
    }else{
        this.validations.email.valid = true;
        this.validations.email.message = '';
    }

    if( this.form.password == ''){
        this.validations.password.valid = false;
        this.validations.password.message = 'A password
must be entered'
    }else{
        this.validations.password.valid = true;
        this.validations.password.message = '';
    }

    return (
        this.validations.email.valid &&
        this.validations.password.valid
    ) ? true : false;
}
```

The implementation of the validations is pretty simple. The first validation that we check to see if an email address is entered and it matches a valid email address. The second validation simply checks if something is entered for the password. Any mis-match combination we will handle in the response from our server request.

Step 5: Implement the Login Method

Here's the meat and potatoes of this page, the actual log in method. Similar to the registration method, let's discuss how this should work, then take a look at the implementation.

The first thing we need to do is ensure our log in is valid through the `validateLogin()` method. Upon a valid form, we move to the next step.

Once we ensure a valid log in form and since we are using Laravel Sanctum, the next request for any authentication is to get a CSRF cookie. To do this, we make a request to the `/sanctum/csrf-cookie` route.

After we get our cookie, we make a request to authenticate the user with the native `loginWith` method on the `$auth` object. Then Nuxt will take it from there, listen for a successful response, then load the user through the `/api/v1/user` route. Our method should look like:

```
login(){
    if( this.validateLogin() ){
        let email = document.getElementById('login-
email').value;
        let password = document.getElementById('login-
password').value;

        this.$auth.loginWith( 'laravelSanctum', { data: {
            email: email,
            password: password
        } } )
        .then( function(){
            // ... Handle successful login
        }.bind(this))
        .catch( function( error ){
            // ... Display an unsuccessful login
        })
    }
}
```

```
    }.bind(this));
}
},
```

Since we are using the laravelSanctum method with the login module, the call to get the CSRF cookie to `/sanctum/csrf-cookie` is handled by the module itself! Once the authentication module completes that request, it submits the data to the /login route via post to attempt to authenticate a user.

The loginWith method returns a promise so we can handle success or failure using the `.then()` and `.catch()` chained methods. Let's go through each of these scenarios.

Handle a Successful Authentication

If the log in is successful, we handle the success through the `.then()` method. This callback is super helpful because we can gracefully handle the promise.

Assuming the login is successful, the NuxtJS Auth module will automatically call our user route to load the authenticated user upon success. The user will then be stored in our Vuex store and accessible within our application through `$auth.user` or in a component with `this.$auth.user`.

We now have everything we need set up to securely authenticate with Laravel Sanctum and store a user within our application! When we compile to mobile there will a few slight adjustments to the authentication, but after that the entire app will operate the same!

Now what happens if the login attempt fails?

Handle an Un-Successful Authentication

We went through a request where everything works as planned, a user enters

valid credentials and gets redirected to the main page and all is well. Now we all know, that's not always the case, so how do we handle an un-successful login attempt? Luckily NuxtJS' auth module makes it extremely easy.

Since the actual request to log in returns a promise, we can apply a simple catch for if the promise is not fulfilled, which will be the case with invalid credentials. For more information on how to `catch` promises that don't fulfill, check out Mozilla's documentation on how to [catch promises that fail](#). Just a heads up, this will be a little different as we get into API requests since we will be using Async/Await. With the auth module, we can use a standard chained promise since that's what it returns.

If we look at our `login.vue` component and specifically at the `login()` method we just wrote, check out the line for when we actually send the request to authenticate the user:

```
this.$auth.loginWith( 'localSanctum', { data: this.form } )
```

What we can do here, is append a `.catch()` method to the end of the request, to handle any errors returned. To do this, adjust the code block to look like:

```
this.$auth.loginWith( 'local', { data: this.form } )
  .catch( function( error ){
    });
  
```

Within the `catch()` method, we can handle any errors that get returned from the authentication server. The way that I handled invalid credentials is as so. First, I added an `invalidLogin` key to the `validations` array and set up the following pieces:

```
...
invalidLogin: {
  valid: true,
  message: ''
```

```
 }  
 ...
```

Next, I adjusted our `catch()` method to show the invalid login message along with flag the email and password as invalid. This way, in our template these fields will be highlighted and the user will see that their credentials are invalid:

```
this.$auth.loginWith( 'laravelSanctum', { data:  
  this.form } )  
  .catch( function( error ){  
    this.validations.invalidLogin.valid = false;  
    this.validations.invalidLogin.message = 'Invalid  
credentials, please try again!';  
    this.validations.email.valid = false;  
    this.validations.password.valid = false;  
  }.bind(this));
```

Finally, I added the following validation below our form, so it appears if the user enters invalid credentials:

```
<span class="text-xs text-red-500" v-show="!  
validations.invalidLogin.valid">{{ validations.invalidLogin  
.message }}</span>
```

Now when the user enters invalid data, we can alert them and allow them to retry!

Conclusion

At this point, we have successfully authenticated a user and are ready to add some functionality to our application! The hardest part of setting up an API + SPA is over and we can begin to add some functionality to our app. We will be re-visiting authentication slightly when we compile to mobile, but for now, we can move to other parts of the app!

Logging Out A User

Now that we have our authentication structure in place, how do we log out a user? Well, we have pretty much everything done already! We have our API route built to handle a log out, our `/logout` route registered with Laravel Sanctum, and we have the NuxtJS auth module installed which has a method to perform a logout. So how do we implement it?

Step 1: Implement `logout()` method

This is really the only step and you can do it in any component that you want. In our example app, we implemented the method in the `/components/global/Header/AppHeader.vue` component since it's present on every page within our app. This gives the user the opportunity to logout at any time. We just added a link to the top for the user to click and call the following method:

```
methods: {  
    async logout(){  
        await this.$auth.logout();  
    }  
}
```

That's it! NuxtJS auth module will then call our `/logout` route, which clears the authentication with Laravel Sanctum and NuxtJS will redirect us to the home route.

Just a personal recommendation, I'd implement this method in a place that is accessible by the user at any point while using the app, just adds for better UX.

Handling Guest Users With NuxtJS Middleware

The hardest part of setting up our API Driven Application is complete! We now have an authentication system in place that's ready to register and authenticate users.

However, now that we have login and registration pages, we should add our first piece of middleware, a guest middleware.

Before we get too far along, let's discuss the current state of users that we have right now on the system. We have users that are just visiting our app and are not authenticated. The other type of user is authenticated. Pretty simple right now, we don't have any tiered permissions at this point, but we will.

For the authenticated user, the NuxtJS auth module ships with a middleware that blocks pages if the user is authenticated. You can apply it on a page level or component level. This will block that content based on whether or not the user is authenticated or not. More information about that middleware check out the [Next Auth middleware documentation](#).

What we should do, for some good UI/UX is add a **Guest** middleware. This is exactly the opposite of the auth middleware as it only allows access if the user is ****NOT**** authenticated. We will apply this middleware to any routes that should not be available to users who ARE authenticated. A few routes I can think of are marketing pages or if you wanted to have actual pages to login and register users. If the user is already logged in, they will never need to access these routes. The logic being, if you are already registered or authenticated, you don't have to do it again nor should you be shown marketing material if you already are in the app.

In this section we will go through and implement a new **Guest** middleware and show an example of how to use the **Auth** middleware to block pages that should not be accessible by a guest user.

Step 1: Add Guest Middleware

The first step to creating any custom middleware is to create a module that represents your middleware. The way I approach it is I add a file with the name of the middleware as the filename. So in our case, our first middleware will be in the `/middleware/guest.js` file. The first step is to create that file and add the following code:

```
export default function( context ) {
  if ( context.$auth.loggedIn ) {
    return context.redirect('/');
  }
}
```

What this does is create a middleware module that we can implement on certain pages and routes. NuxtJS will automatically register this middleware when the file has been created.

The function is extremely straight forward. First, it accepts the app context. This is essentially the global [NuxtJS object that contains all of our plugins, vue properties, etc.](#)

The `context` also has our `$auth` module as one of its plugins. The auth module has a property that determines if the user is authenticated or not called `loggedIn`. We check to see if that returns `true`, meaning that there is an authenticated user. If the user is authenticated, we use the global `redirect()` method which accepts a route as its parameter. In this case we pass the url of `/` which is just our home page.

That's it! We have now added our first custom middleware! We can now apply it to any page that we don't want authenticated users to visit. In the next step we will

go through implementing both this middle and the auth middleware.

Step 2: Implement Middleware

Now that we have our middleware, it's time to implement. Our middleware is accessible by what we named our file, so in this case, `guest`. Let's say that we implemented a page for authentication that's called `/login`. We don't want authenticated users to view this page. To prevent authenticated users from viewing this page, we'd need to add the following to the page component:

```
export default {
  middleware: 'guest',
  ...
}
```

Guess what? That's it! Now once you are authenticated, you will be redirected to `/` when trying to access this page.

On the flip side, say we have a profile page that can NOT be accessed by a guest. You'd open the profile page component and add the following code:

```
export default {
  middleware: 'auth',
  ...
}
```

What this does is prevents any guest users from accessing this page! We didn't make the auth middleware, that shipped with the NuxtJS Auth module. You can also use an array for the middleware key. This allows you to apply multiple middlewares to a route. Pretty awesome and very similar to Laravel!

Implementing an Email Validation System for New Registrations

Why is this important? Well for a lot of reasons. With the intelligence of modern bots you want to eliminate fake users right off the bat. Ensuring the newly registered user has validated their email address before allowing them to submit data is extremely important.

Laravel has a lot of this set up for us which is nice, but it's not designed to work in an SPA + API scenario. If you think about it, the SPA really doesn't know what the API does and the API doesn't know much about the SPA or who is using it. The email validations built into Laravel are beautiful but are designed to grab request variables in the sense of a monolithic app. Well when we re-direct a user with one of these request variables, it goes to our SPA front end and never touches our API. Now we have some work to do to get this set up.

Step 0: Overview of Verification Flow

The first problem we need to solve is how the flow is going to work. Normally, when a user registers, they are created with a null value set in the `email_verified_at` column in the database. This allows us to ensure the user has validated their email when accessing certain endpoints of the application.

After the user has been created, an email has been sent to the user's email address with a signed URL that they can click to validate the email address they entered upon registration. This URL contains a signature hash that gets grabbed by Laravel and updates the `email_verified_at` field for the corresponding user. The key piece of the flow is the signature gets grabbed by Laravel. In our SPA + API we don't have this luxury. The URL could also be deep-linked into our mobile application which we need to account for.

The new flow that we will use will be the following. First, a user will sign up

through our application. This will create a new user through the `/register` API endpoint and send out an email with a verification URL. This URL will be pointed to the front end URL of our application, the SPA. On mobile, these URLs will be deep linked to the mobile app (we will run through that later).

When generating the URL for the verification email, we will be appending the API endpoint with signed hash to the verification URL. Sounds like URL inception right? It kind of is. Our format will look like:

```
https://spa-url.com/verification?verify_url=https://api-
url.com/{verification-hash}
```

Now when the user clicks the link to verify the URL to verify their email, we check to see if the `?verify_url` query parameter is set. We then have the endpoint that we call to verify the user's email. We will call this in our SPA through Axios and it will flag when the user's email address was verified at.

The final piece of will be implementing some NuxtJS middleware that prevents certain actions based on whether the authenticated user has verified their email address or not. We will also be implementing the corresponding verified middleware on the API side as well so there's not loose ends.

While creating this piece of the application, I got some inspiration from the following repo and blog post:

<https://github.com/tolgayildizz/laravel-5.7-email-verification-and-auth-via-api>

[https://medium.com/@pran.81/how-to-implement-laravels-must-verify-email-
feature-in-the-api-registration-b531608ecb99](https://medium.com/@pran.81/how-to-implement-laravels-must-verify-email-feature-in-the-api-registration-b531608ecb99)

I then used this inspiration to construct an updated email verification system for an API with Laravel 8.x and NuxtJS.

Step 1: Create Necessary Verification Endpoints

There are a lot of moving parts in this feature, but I believe the best place to start is to begin by creating the necessary API endpoints that make this feature work.

For email verification, we will need two routes. The first is the route we will send the verification request to when the user clicks the link and the SPA sends the request to verify to the API. The second is a route that the user can use to re-send the verification email if they didn't receive it the first time.

Let's start with the first route. When the user clicks the link sent in the email, they will be brought to a page within the SPA that looks for the `?verify_url` query parameter. That page will grab the parameter from the request and submit it to the API endpoint to complete the verification. For now, we just need to register this route in our `/routes/api.php` file:

```
Route::group(['prefix' => 'v1'], function(){
    // ... Other routes
    Route::post('email/verify/{id}',
        'API\EmailVerificationController@verify')-
        >name('verificationapi.verify');
});
```

The first thing to notice is we gave this route a name of `verificationapi.verify`. This is extremely important, especially since I don't name routes often. This is because we will be using this when we generate our signed URL. We need to reference this route by name. We will get to this when we create our verification notification.

The second route we need to create is a route the user can hit to re-send the verification email. For now, let's add this route within `routes/api.php` file:

```
Route::post('email/resend',
'API\\EmailVerificationController@resend');
```

We don't have the EmailVerificationController built yet and that's fine, we will do that and implement these routes in the next step. For now, we just needed to stub them out as a place to start.

Step 2: Create and Implement Email Verification Controller

Since we have our routes created, we need to create a controller to handle these requests. This controller will take care of the actual functionality of the routes themselves.

First, let's add the following file `Http\Controllers\API\EmailVerificationController.php` and shell out the functionality:

```
<?php

namespace App\Http\Controllers\API;

use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\VerifiesEmails;
use Illuminate\Http\Request;
use Illuminate\Auth\Events\Verified;

class EmailVerificationController extends Controller
{
    use VerifiesEmails;

    public function __construct(){
        $this->middleware('auth:sanctum');
    }

    public function show()
    {
```

```
}

public function verify( Request $request )
{

}

public function resend( Request $request )
{
}

}
```

There's definitely a lot going on with this controller so let's break it down. First, it's a pretty "function-less" shell. We essentially just have the proper methods created but not yet implemented. However, there are some special features about this controller.

First, we added use `Illuminate\Foundation\Auth\VerifiesEmails`; to our use declarations. This is the trait that the controller class implements which gives us the methods we need to override if we are verifying an email. Those methods are `verify()` and `resend()`. We also, put the use `VerifiesEmails`; as the first line in the controller class.

Second, we added use `Illuminate\Auth\Events\Verified`; to our use declarations. This will dispatch an event that we can listen to when a user has been registered. We are essentially re-building what Laravel already provides, except hooking it up to work through an API. This way everything works as expected when you further develop your API.

Third, we added a `show()` method. This one is actually complete! We don't actually show anything from the API in the form of a page to verify an email since that's all handled on the SPA side. We just needed to implement this method to

satisfy the `VerifiesEmail` trait.

The final thing to notice for now, is we added the following to the `__construct()` method:

```
$this->middleware('auth:sanctum');
```

We haven't talked about middleware much yet, but what this does is makes sure that all of the routes that are handled by the controller are behind authentication. A guest user can not verify an email or resend an email verification. These endpoints will return a 401 Unauthenticated error if accessed by user who is not authenticated.

Now that we have our methods stubbed out, let's implement the first `verify()` method:

```
public function verify( Request $request )
{
    if ( $request->route('id') == $request->user()->getKey() &&
        $request->user()->markEmailAsVerified() ) {
        event(new Verified($request->user()));

        return response()->json('Email Verified');
    }else{
        return response()->json('Email failed to verify!', 400);
    }
}
```

For as small of a method as this is, it's pretty dense. So remember, we are calling this endpoint with an id that is the signature representing the user. We need to ensure that the ID passed in through the route parameter matches the ID of the user. We also need to ensure the user's email gets properly marked as verified. If those both pass, then the email has successfully verified and we return a successful response. If it fails to verify, then we return a 400 notifying the user that the email failed to verify.

Now, let's implement the `resend()` functionality:

```
public function resend( Request $request )
{
    if( $request->user()->hasVerifiedEmail() ){
        return response()->json( 'User already has a
verified email!', 422 );
    }

    $request->user()->sendApiEmailVerificationNotification();
    return response()->json( 'Please check your email to
verify' );
}
```

What this does is first check to see if the user has already verified their email. If they have, then we return a 422 response with the message “User already has a verified email”. If they haven’t verified their email, we resend the email verification through the method on the user model we will be implementing in Step 6: Add Notification Method to User Model. This will resend the email and return with the response to “Please check your email to verify”.

That’s all we need for the controller, however, we have quite a few more steps to implement the full functionality.

Step 3: Create Email Verification Mailable

The next step in the process will be to create our `EmailVerification.php` mailable. This is what will be sent out to every user after they register to confirm their email address. In the console, run the following artisan command:

```
php artisan make:mail EmailVerification
```

This will send out the email for the user to validate after they register. You should get an empty mailable that looks like:

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class EmailVerification extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('view.name');
    }
}
```

```
}
```

Let's update the `__construct()` method first. We will need to add two public properties on the Mailable model:

```
public $verifyURL;  
public $name;
```

These will be set with the data the necessary to build the email to send out. Let's set that data through our `__construct()` method:

```
/**  
 * Create a new message instance.  
 *  
 * @return void  
 */  
public function __construct( $url, $name )  
{  
    $this->verifyURL = $url;  
    $this->name = $name;  
}
```

When we create our mailable, we will pass in the information we need to create the mailable. The most important variable is the `$verifyURL`. This will be the URL the user clicks to verify their email address. The `$name` is the name of the user to send with the email. Nothing too special yet, we will tie the loop together after a few more steps and it will all make more sense soon.

The next method we need to implement is the `build()` method. This method is on the Mailable trait that gets called when the email is sent. You can customize this method to whatever suits the needs of your application (what view should send the email, the subject, who it's from, etc.). Our method looks like this:

```
/**
```

```
* Build the message.  
*  
* @return $this  
*/  
public function build()  
{  
    return $this->view('emails.verify')  
        ->text('emails.verify_plain')  
        ->subject('Account Activation')  
        ->from('noreply@roastandbrew.coffee',  
'ROAST');  
}
```

We are sending a verify blade file (which we will create next) in the emails directory. One for each HTML and plain text which is really important so the email doesn't go to SPAM. You have to account for email applications that don't support HTML. We also will be sending the email from noreply@roastandbrew.coffee with the name ROAST.

Now we need to create some blade views. Yup, you read that right, our API will need a few blade views. These are just email templates to send along with the verification email.

Step 4: Create Email Verification Views

The only time we will be using blade templates on our API is for when we create emails. This is one of those cases. To make this more explicit in our code base, we will create a `/resources/views/emails` directory. The `/emails` directory should be the only directory within the `/views` directory since we won't have any more blade templates.

Within the `/resources/views/emails` directory, add the following 2 files, `verify_plain.blade.php` and `verify.blade.php`. Both of these files will be the same email, just a plain text version and an HTML version. With the HTML version, the design is up to you and your branding. If you are curious on what

ROAST's looks like, check the repo. We will go through the plain text version since it's much easier to look at and gets the point across on what we need to do. Our plain text email looks like this:

Hey {{ \$name }},

We're excited to have you on board! Click the link below to activate your account.

[{{ \\$verifyURL }}]({{ $verifyURL }})

Need Help?

Contact Support: support@roastandbrew.coffee

We use the `$name` variable from our email and provide the `$verifyURL` to the user to click and verify their email address. These variables will be present in the HTML version as well, just wrapped in document structure elements and buttons. The next step is making the email verification notification.

Step 5: Create Email Verification Notification

So why do we need to create a notification when we have an email already created? Can't we just send the email? Well you could implement the functionality that way but we are following Laravel's implementation. It's also nice to use the notify method on the user and have a standard flow for notifications in place. We can implement multiple notifications and determine how we want to send them out following a standardized work flow. Let's go through the process and you will see how this works.

The first step is to create our notification by running the following command:

```
php artisan make:notification VerifyApiEmail
```

Running this command gives you a fairly complex stubbed out notification that looks like this:

```
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;
use Illuminate\Notifications\Notification;

class VerifyApiEmail extends Notification
{
    use Queueable;

    /**
     * Create a new notification instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Get the notification's delivery channels.
     *
     * @param mixed $notifiable
     * @return array
     */
    public function via($notifiable)
```

```
{  
    return ['mail'];  
}  
  
/**  
 * Get the mail representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return  
 \Illuminate\Notifications\Messages\MailMessage  
 */  
public function toMail($notifiable)  
{  
    return (new MailMessage)  
        ->line('The introduction to the  
notification.')  
        ->action('Notification Action',  
url('/'))  
        ->line('Thank you for using our  
application!');  
}  
  
/**  
 * Get the array representation of the notification.  
 *  
 * @param mixed $notifiable  
 * @return array  
 */  
public function toArray($notifiable)  
{  
    return [  
        //  
    ];  
}
```

Instead of running through all of these methods, we are actually going to delete most of them to get started. Reason being, we will be using the `Illuminate\Auth\Notifications\VerifyEmail` notification as the base which already implements most of what we need. We will just be overriding a few of the methods.

The first thing we need to do is get rid of the `toArray()`, `via()`, and `__construct()` methods. Once those are deleted, we should remove the functionality that comes in the `toMail()` method. We will be completely overriding this.

The final set up step is we need to need to add `use Illuminate\Auth\Notifications\VerifyEmail as VerifyEmailBase;` to our use declarations and make sure that our notification extends the existing notification. Our base notification should look like this:

```
<?php

namespace App\Notifications;

use Illuminate\Auth\Notifications\VerifyEmail as
VerifyEmailBase;

class VerifyApiEmail extends VerifyEmailBase
{

    public function toMail( $notifiable )
    {

    }
}
```

Now that we have our base notification set up, let's first override the `verificationUrl()` method that we use to create the verification url. To do this, add the method to our notification like this:

```
protected function verificationUrl($notifiable)
{
    $prefix = env('FRONTEND_URL').'/account/verify/';

    $temporarySignedUrl = URL::temporarySignedRoute(
        'verificationapi.verify', Carbon::now()-
>addMinutes(60), ['id' => $notifiable->getKey()]
    );

    return $prefix . '?verify_url=' .
urlencode( $temporarySignedUrl );
}
```

Before we break down this dense method, let's add the following use declarations so everything is set up and ready to work:

```
use Illuminate\Support\Carbon;
use Illuminate\Support\Facades\URL;
```

Okay, let's break this down! The first line of the method is we create the base of the URL we wish to direct the user to when they click the notification link. In our `.env` file, we have a variable set to be named `FRONTEND_URL` and that will be the URL of our SPA. With this we append the `/account/verify` path which will be implemented on the frontend. This is the base of our URL.

The next step is we create another URL string that is a temporarily signed route. This means that we will create a route that only exists for the amount of time we specify (why we are using Carbon) and we will be signing it to be for the id of the `$notifiable` entity (our user). This URL will be the API endpoint we call to verify

our user.

The following code creates this URL:

```
$temporarySignedUrl = URL::temporarySignedRoute(
    'verificationapi.verify', Carbon::now()-
>addMinutes(60), ['id' => $notifiable->getKey()])
);
```

Remember from Step 1 where we named the route we use to verify on the API? Here is where we use that name! We are signing that URL endpoint for 60 minutes (the second parameter). The third parameter is the URL variable we are using to add the signature to. In this case it's the id parameter that we are setting to be the key of the `$notifiable` entity. This case, the user who needs to verify their email.

Now that we have that route created, we need to append it to the front end route as a query parameter under the key `?verify_url`. We also need to `urlencode()` it since we will be passing it via URL to make the request.

Since we've gotten this set up, we need to update the `toMail()` method to look like this:

```
public function toMail( $notifiable )
{
    $url = self::verificationUrl( $notifiable );
    $name = $notifiable->name;

    return (new EmailVerification( $url, $name ) )-
>to( $notifiable->email );
}
```

Also, make sure we add our mailable from Step 3 to the use declarations at the top of this file:

```
use App\Mail\EmailVerification;
```

Now, when our notification is sent out and we run the `toMail()` method, we will be sending out the email that we created. The `$notifiable` parameter will be the `$user` instance that's injected into this notification.

From there, we first call the local method to create the verification URL. This is the method we just created. The second line, we extract the name parameter from the `$notifiable`. These two parameters look similar? The `$url` and the `$name`? That's because they are the two parameters we need for our email verification URL.

The final line of code in the method is to create a new instance of the `EmailVerification` mailable that we created, pass in the parameters and send it to the email of the user.

We aren't done yet, we still have a little more wiring up to take care of! The next step will be creating a method on the user model to send this notification when needed.

Step 6: Configure User Model to Verify Email

There are a few steps we need to take to configure our User model to require them to verify their email. The first step is we to do is make sure that the user implements the `MustVerifyEmail` contract. This allows us to override what we need and check to see if the User has verified their email. To do that, make sure you include the contract in the use declarations at the top:

```
use Illuminate\Contracts\Auth\MustVerifyEmail;
```

Once that's included, we need to make sure our model implements the contract like so:

```
class User extends Authenticatable implements  
MustVerifyEmail{  
}
```

We also need to make sure the User Model is notifiable. To do this, we have to add the following to our use declarations:

```
use Illuminate\Notifications\Notifiable;
```

Then ensure the model uses the Notifiable trait:

```
class User extends Authenticatable implements  
MustVerifyEmail  
{  
    use Notifiable;  
}
```

At this point, our user model is set up to implement the `MustVerifyEmail` functionality and is notifiable meaning we can send notifications to the model through Laravel's notification system.

Now that that's configured, we have to create a method to send the notification when needed for the user. The first step is to include the notification we created in Step 5:

```
use App\Notifications\VerifyApiEmail;
```

Now, let's add the following method to the model:

```
public function sendApiEmailVerificationNotification()  
{  
    $this->notify( new VerifyApiEmail );  
}
```

When the user registers, we call this method. What this will do is run the `notify()` method inherited from the `Notifiable` trait. We then pass our notification we created to this method to send to the user.

We are getting closer to tying this all up! Only a few more API steps and we are good to go.

Step 7: Alter Registration Method to Send Out Verification Email

At this point we have our notification created, email created, routes created to handle the callbacks and a method on our user model to send out a notification. Let's kick off the whole process by actually calling the method on the user model.

This happens when the user completes a registration. Remember, we want the user to sign up, but before they can perform any major tasks or functions, we want their email to be verified. The best place to send out this email is right after they register.

Let's open up the `AuthController.php` and find the `register()` command. Right after we ensure the validator passes, we create a new user. Let's add a call to the method we added in Step 6 to send out the email verification notification:

```
// If everything is valid, create a new user, send an email
// verification email
// and return a successful 204 response meaning an entity
// has been created.
if( !$validator->fails() ){
    $gravatar = new Gravatar( env('FRONTEND_URL').'/img/
user/user.png', $request->get('email') );

$user = User::create([
    'name' => $request->get('name'),
    'email' => $request->get('email'),
    'password' => Hash::make($request->get('password')),
    'verified' => false,
    'token' => Str::random(60),
]);
```

```
'name' => $request->get('name'),  
'email' => $request->get('email'),  
'password' => bcrypt( $request->get('password') ),  
'avatar' => $gravatar->get()  
]);  
  
$user->sendApiEmailVerificationNotification();  
  
return response()->json('', 204);  
}
```

That's the piece that ties it all together! From this point forward, we just need to implement some middleware on the API side and create the SPA route to handle the request when the user clicks the link in the email.

Step 8: Implement Verified Middleware on the API

We haven't discussed middleware too much so I just want to make you aware that Laravel ships with a verified middleware. What that means is you can block off access to certain API endpoints depending on whether the email address of the user is verified or not.

To apply this middleware, you need to open up any API endpoint controller and add the following to the `__construct()` method:

```
$this->middleware('verified');
```

This will block off all of the endpoints on that resource controller if the user does not have a verified email address. You can also block off certain methods and endpoints which we will discuss later when we dive in depth into middleware. That's all we need to do on the API side! Now let's move to the NuxtJS SPA side and implement the page that handles the verification for the user!

Step 9: Build NuxtJS Verification Page

At this point, our API is set up to notify the user to verify their email address and handle a response when they click the verify link. We just need to implement a front end route to handle this request.

Like we mentioned earlier, instead of redirecting back to a page that's controlled by Laravel, we are going to redirect to our frontend SPA which could be web or mobile. We then have to forward the call to our API so we can verify an email address.

This page will serve two functions. It will be the landing point when a user clicks "verify" from their email and it will be where we re-direct users if they try to access a page they can't access without a verified account.

Let's start with just building this page. It's up to you how you want to design the page, but there are some basic functions we will need to implement. We are also going to place this page in the `/pages/` account directory and name it `verify.vue`. This is so the URL that gets generated by NuxtJS matches what we are sending on the API side.

In the `mounted()` hook on the page, let's add the following code:

```
mounted(){
  if( this.$route.query.verify_url ){
    this.verifyEmail();
  }
},
```

What this does is when a user accesses this page, we check to see if the route has a query parameter named `verify_url`. If they have just clicked the link from the email, they will have this query parameter. It will be the signed URL of the API

endpoint we wish to hit. We then call the method `verifyEmail()` which is implemented next.

In the methods object, add the following two methods:

```
methods: {
    async verifyEmail(){
        this.verifyingEmail = true;

        this.message = await this.$axios.post(this.
$route.query.verify_url);
        this.reloadUser();
    },
    async reloadUser(){
        await this.$auth.fetchUser();
        this.verifyingEmail = false;
    }
}
```

Both of these methods are `async` methods. The first method actually sends the request to the API to validate the user's email. The API endpoint is the value of the `verify_url` parameter which is signed for the authenticated user. When the endpoint is hit, the user will have the date updated on their `email_verified_at` column.

Upon registration, this column is set to `null`. However, when we register the user, the user is also authenticated and returned through the NuxtJS auth module. We need to update the store of the user object so we can determine if the user has a valid email address on the SPA side. That's where the `reloadUser()` method comes in. The `reloadUser()` method calls the authentication module's `fetchUser()` method which in turn calls the `/api/v1/user` endpoint we defined to return the user. This will re-load and save the user object in our Vuex store. The user will then be accessible through the NuxtJS auth module and we can check if

their email address is verified. The next step is the middleware that handles this.

The second piece of functionality is this page should allow the user to re-send the email verification link. This could be due to a variety of reasons, most likely an expired URL. To do this, I just added the following method (we already had the endpoint created in Step 1):

```
async resendVerificationEmail(){
    await this.$axios.post('/api/v1/email/resend');
    this.emailResent = true;
},
```

We have this method wired up to be called from a button, but you can have it set up anyway you wish for your design. Now the user can receive an email if they didn't the first time.

This completes the entire loop of the process! A user now has to verify their email address upon registration! The next step is supplementary but is extremely important to making sure the functionality works from SPA to API and that's creating a middleware to block off parts of the front end for users who do not have validated email addresses.

Step 10: Block off Functionality for Non-Verified Users On the SPA

For this step, we need to create a simple NuxtJS middleware that prevents unauthorized access to certain parts of the application for users that do not have a verified email address. This will work hand in hand with the auth middleware most of the time, which means that most pages will have two middleware applied to them. Luckily, NuxtJS makes that extremely easy!

First, let's create our NuxtJS middleware. Start by adding the `/middleware/verified.js` file and add the following code:

```
export default function( { store, redirect } ) {
```

```
if ( store.state.auth.loggedIn &&
store.state.auth.user.email_verified_at == null ) {
    return redirect('/account/verify');
}
}
```

What this does is create a simple middleware that checks first if the user is logged in and second if the authenticated user's email address has been verified. It will be set to null if it hasn't. If there is a logged in user without a verified email, we redirect them to the page we just created `/account/verify` where they can resend the email if they haven't received it yet.

So where would we apply this middleware? Say the user is authenticated and wishes to add a new company. We encourage authenticated users to add new companies for us to review. However, if they don't have a verified email address, we don't want them performing these actions. It will prevent us from having to deal with tons of spam. Now we haven't implemented a resource yet, but the way you'd add the middleware to a page would be like this:

```
export default {
  middleware: ['auth', 'verified'],
}
```

This is an example of applying two middlewares to the same page. We are checking to see if the user is authenticated and if they are verified before allowing access. Now, yes, this is Javascript and a user could tinker with the variables through the console, but remember, we block off our endpoints on the API side as well so it would just reject any request that was un-authorized.

That's how we go through ensuring an SPA + API user has validated their email before using the app! There's a lot of moving parts, but it will save you headaches in the long run and get more users who are not bots on your platform!

On to "Fun"-ctionality

That was a LOT of set up. But you know what? It will start to pay dividends. Especially when you cash in on mobile app deployment. Here's what we have now:

- A bunch of code buckets to add secure data routes to
- Authentication structure in place on both API and SPA
- User registration
- Middleware on both API and SPA
- Email validation to make sure we don't have bots

We are now ready to start having some fun with our structure and make it our own. As I'll state again, the main difference in how we will approach this compared to the Server Side Up series I wrote will be from an "app agnostic" perspective. What that means, is I'll explain concepts, go through how they work and then show an example with ROAST.

This approach will give you the tools and knowledge necessary to implement these features into your own application much more seamlessly than trying to understand a concept, decouple it from what we were building, and then implement it.

Here we go, Let's rock and roll!

THE FULL-STACK FEATURE APPROACH

Using the Proper Methods for API Requests

So it's time to start working within our API and SPA to build some features and create value with our application! There are 4 main request types, which are formally known as **HTTP Verbs**, that make up all of our communication with our API: GET, POST, PATCH/PUT, DELETE. Each of them serves a different purpose to tell the API Endpoint what to do.

GET

GET is probably the most common of ALL requests and the one that you use without even noticing. A **GET** request is sent to a URL in order to retrieve data. When you navigate to a page in your web browser, you are submitting a **GET** request that returns page data. In the sense of an API, we will be using **GET** requests for a variety of purposes but with the same mindset in place, making a request to a URL and retrieving data. In our cases, this will be JSON and data.

We've actually already implemented an endpoint for a **GET** request and that's our `/api/v1/user` endpoint. When we submit a **GET** request to that endpoint, we get the authenticated user in JSON format. You can also use **GET** to pass parameters to the endpoint which we will be implementing heavily as well, making these requests a lot more complicated than you may think.

An example of these parameters would be like `start_date=XXXX-XX-XX` or `location=XYZ`. We can access these parameters on the API and make more efficient queries. Don't worry, we will be implementing a few examples of these. Just make sure you don't pass secure data through these parameters, that's for our next request.

POST

POST is the second most common of all of the requests. **POST** allows you to send a

lot of data to the API with the expectation that a resource gets created. When we submit forms, authenticate, etc. we send this data via **POST**. The reason we send the data via **POST** is because the data is sent in what's referred to as the body of the request. Over HTTPS, this would be securely encrypted. That's why we use **POST** to pass sensitive information such as log in credentials. As a matter of fact, this is the verb we used in our own login method.

PATCH/PUT

These two HTTP verbs are definitely less common and have only started to become more popular with more development of RESTful API Endpoints. They both operate very similar to **POST** in the fact they send the data in the body of the request, which is hopefully secured through HTTPS, and can contain sensitive data if needed.

What makes these two HTTP verbs different is their semantic use case. When sending a request via **PATCH** or **PUT** we are expecting to update a resource. Say that we send a **PATCH** request to `/api/v1/user/4` with a body of `name: 'Mike'`. We would update that user to have a name of **Mike**.

Now there is a difference in use case from **PUT** to **PATCH**. Wikipedia has a great definition here:

The main difference between the **PUT** and **PATCH** method is that the **PUT** method uses the request URI to supply a modified version of the requested resource which replaces the original version of the resource whereas the **PATCH** method supplies a set of instructions to modify the resource. If the **PATCH** document is larger than the size of the new version of the resource sent by the **PUT** method then the **PUT** method is preferred.

Think of it this way, if I were to send ONLY an updated field for a user, I'd send it through **PATCH**. If I sent the entire user JSON object back and updated ANYTHING to be swapped out, I'd send it through **PUT**. Definitely important to notice the difference for proper API structure. We will be making use of both of these HTTP

Verbs and ensuring we do it to design.

DELETE

This HTTP Verb is pretty self explanatory, but also not as common as **GET** or **POST**. The **DELETE** method, well, gets sent when you want to delete a resource. For example, say you have a user that needs to be removed from the system. You'd then send a **DELETE** request to `/api/v1/user/332` and the user with id `332` would be removed from the system (granted you have the right permissions).

So that's a quick overview of HTTP verbs we will be using throughout this book. There will be a lot of examples to go through with these verbs and we will do that in the next section with the basics of managing a resource. Now to make matters more fun, the state of processing by the API endpoint should return a response code according to the verb + action status. Don't worry, I have a guide here that explains the proper response codes and we will be showing examples of these as well:

[RESTful Response Codes: How To Use Them](#)

Building a Feature From Start to Finish

My thought process when implementing features is always start at the foundation, the API. I then work myself up the stack until I get to the UI. Within both the server and client side of the application, I approach each section in a certain order as well. Let's go through a basic approach.

Feature Planning

Before any development gets done, it's always good to plan the feature and gather requirements. This allows you to know what you are building and how you want it built. Whether you are building an app for yourself or working on a team, feature planning is the most important part of the development lifecycle. It sets expectations, what should be done, when it should be done by, and what you need to accomplish the task.

API

I always begin with the API side of the app first. In my mind, this is the base, the concrete foundation where everything links together. I like to ensure my business processes work efficiently and are tested to the max before we develop a UI.

This also includes all computed data that is necessary to return. To make efficient APIs, allowing for data to be computed on the server side should be attempted as much as possible. Granted, it's not always possible, but when it comes down to a choice, do it server side. For example, ordering a list of companies. You wouldn't want to load all of the companies in the database before ordering them with Vue, you'd want to order them through a SQL call before returning them. You also wouldn't want to return every user who liked a cafe to calculate with Vue. Not only would that leak data, but would also take forever on popular cafes. You'd want to return that data computed from the server side API.

I also have a flow in which I follow when developing the API side after everything is planned out. This flow is meant to cover the basic feature addition structure.

There will be more steps involved if we add features that require different middleware or user permissions, but this should encompass the basic flow.

1. Database

I like to develop an efficient data structure for the feature that we are adding first. Once we have everything planned out this should be pretty straight forward on what you need to store, it's relationship to other entities, etc. Within Laravel, this involves simply making a database migration.

2. Model

Within Laravel, we have access to Eloquent which is a super easy way to model and apply relationships to database entities that we can easily chain together through code. After I get the database set up and configured, I set up the models and their relationships to other models. This gives me a way to apply CRUD (Create, Read, Update, Delete) methods to my data through code efficiently and effectively.

3. Route(s) & Controller(s)

After I have my models created, I set up the route(s) we will be using to access our data and perform operations on it. This is where we build our "buckets" for implementation. We will have a place to store new data with our database, ways to access and perform operations with the model and now ways to expose those functions from our API. As you can see, we are building from the inside out.

4. Services

If the feature is large enough and we need to process data before adding it, or there are multiple routes that perform the same action, I like to break that functionality into services. Services provide an easy way to re-use code and allow processing before inserting into the database or retrieving code. This should be the heavy lifting part of the feature providing a lot of the logic and functionality of what's going on.

5. Validations (Middleware, Requests Validations, Policies)

This appears on a case-by-case basis. If you are expanding on additional data, you might not need to add any of this, but if you are adding a new feature with new data you will have to add one or all of these to your feature. Middleware to block certain requests, Validation objects to ensure that the data sent is correct before even processing it, and/or policies to block users from accessing data they don't have permission to access. There are times when I spritz the addition of this into the development lifecycle early on if I'm working with a specific route and the feature needs like 3 or 4 routes.

6. Tests

One of the most important and time consuming parts of the process is writing tests. Tests ensure that everything you have implemented works. For every feature we have, we are going to write tests for every small piece along the way. This ensures everything is functioning as expected before we start accessing the data from the UI.

SPA

As smooth as the API development side goes, the UI/UX side tends to be a little less logical. I find this to be the nature of app development. The goal is to display data in an easily consumable format for the user so this might mean little snippets here and there. I find my process much more loosely tied than the API side but here's the flow I try to follow.

1. Request

The first thing I need to do is create our `/api/{resource}.js` file or append the new routes to it. This allows the data to be easily accessible within our SPA and we can figure out how to get to it from there.

2. Store

Determine where we need to store the data. More or less, do we need to use Vuex

to store the data. My approach to when to use Vuex can be found:

When to Use Vuex

If we need to use Vuex, I will set up all the actions and storage structure before I begin any other implementation since I will know where my data is being housed.

3. Display Feature

Here's where things tend to get off the tracks and not be as straight forward. A resource/route may need a few components, tons of components, require a page, or even in the sake of a new permission level, a layout. This tends to be harder to develop a specific process for and depends a lot on the feature itself.

4. Test

Now it comes time to run our SPA tests to ensure the feature works as it's intended. With NuxtJS this will be with Jest.

This is a quick overview of how I approach the adding of features into an API driven application. With that being said, let's add our first feature and tie in some of what we learned. We will start with managing a resource.

MANAGING APP RESOURCES THROUGH AN API

Efficiently Building API Endpoints for Data Queries

The first API Driven Development feature we are going to walk through is how to manage resources. When managing resources, we will be able to do walk through the lifecycle with ease since it's pretty straight forward.

What is a Resource?

A resource is any piece of data that is managed by your API. These could be orders, invoices, images, users, anything that makes your app tick. All resources have some sort of endpoints to create, access, modify, and delete. In this tutorial, we are going to pick a resource central to the example application we are building and go through the process of implementing it.

Before we get started, I'll preface this section with not all resource

implementations will be this smooth. Some will require extra routes, extra config, or nothing special at all. We will take our first step in an example that's easy to follow and get our feet wet.

Planning

The first step in implementing any feature is planning what the purpose of the feature is and what the feature will achieve. For this example, we are going to allow users to manage a Company resource. In our application, we will be displaying coffee companies to users and they will be able to discover new coffee companies and find companies with cafes to visit. This will be one of the central resources of our application and will be expanding upon it as we develop. For now though, we just want users to be able to create companies, load companies, update companies and delete companies through the API.

Now we don't want really ANY person on the site to create companies, update companies, or delete companies, so we will need a middleware and some validation. Right now, we only have the two types of users: guest and authenticated so let's not think to deeply into it. We will block any modification on whether the user is authenticated or not and allow guests to load companies. We will also add some request validations to ensure our requests contain valid data before persisting. We will run through validations as well.

Cool, so that leads us to the routes we will need.

Loading All Of A Resource

To load all of a resource (in our example, companies), this will be done through a **GET** request. Usually when structuring a RESTful API, you'd have an endpoint that's the plural name of your resource. In our case, it'd be `/api/v1/companies`. For now, we are just going to return all of the companies in the system. Obviously, this should change as we end up getting a lot of companies and need to filter it and provide more quality results, but we are just starting small.

Loading an Individual Resource

Like the loading of all of a resource (companies), this will be done through a **GET** request, except the endpoint of the request will have a unique identifier at the end that we can grab in our API. In our example, the endpoint will be `/api/v1/`

`companies/{id}`. The `{id}` variable at the end can be grabbed by our Laravel API and we can load the specified resource.

Now what happens if the company doesn't exist? We get to return our favorite `404` error! If it does exist, we will return the JSON representation of the company.

Creating A Resource

To create a resource, we will be submitting a `POST` request to the plural endpoint of our resource (`/api/v1/companies`). This URL should be the same URL as the `GET` request to load ALL of a resource, except submitted with `POST`.

There will be two orders of validation. The first will be a middleware to confirm that a user is authenticated or not. If the user is NOT authenticated, we will return a `401` response which is **Unauthorized** and can not add a company.

Once we validate that a user is authenticated, we validate the body of the request to ensure we have the proper data we need. If the data is valid, we persist the company to the database, if it's invalid, we will return a `422` response with a bunch of errors to alert the developer that they need to make changes. `422` means "Unprocessable Entity" or in a verbose way, the data is invalid and needs to be changed before persisting. Upon success, we will return the new company along with a `201` response code meaning a resource has been created.

Updating a Resource

For this we will be using a `PUT` request. The reason we are choosing `PUT` instead of `PATCH` is we will be submitting the entire resource back with the changes that were made. Essentially the route will handle an entire company object with the new changes.

For now, this route will be a `PUT` request to `/api/v1/companies/{id}` (the same as the `GET` request to load an individual resource) with the updated data in the body of the request. The request will have the same two layers of validation of the add a company route; a middleware ensuring the user is authenticated and a route

validation that ensures the updated data is valid.

However, upon a successful response, we will not be returning the updated resource, and return a `204` response code which means the process was successful, but the response is empty.

Deleting a Resource

To accomplish the deletion of a resource, we will use the `DELETE` HTTP Verb. The request will be to the endpoint `/api/v1/companies/{id}` and will delete the company from the system. Only users who are authenticated will have permissions to delete the company, otherwise a `401` will be returned.

Because we are adding routes, it's assumed we will be handling the functions with a controller. Since are walking through the steps of managing a basic resource, we will only need one controller. This controller will handle all of the routes and direct any complex business logic off to a specialized service dedicated handle the request. Don't worry, we will step through all of this very soon.

Middleware

We've talked a lot about having two types of users, guests and authenticated. For the routes that require authentication, we will have to apply some middleware to ensure that the route is not accessed by a guest user.

For this example, we won't have to build any custom middleware which is nice! Laravel Sanctum already provides a middleware to ensure that a user is authenticated before accessing the route. This is what we will be using right now. We will also be applying the middleware to the routes within the controller file itself. I'll touch base more on that when we get there, but I prefer the method of applying the middleware in the controller, since it really cleans up the `/routes/api.php` file and consolidates functionality.

Requests Validations

For the creating and updating of a company, we will have to make request validations. These are a little different than the middleware as they validate the actual data sent over via the request.

We will only need to validate the creating and updating right now. In the future, it's possible to validate search and ordering of the `GET` request if those parameters are present. For more information on the differences between requests and middleware, see [Permissions, Validations, and Security](#).

Tests

Finally, after all of our work, we get to see if we can break the code. We will be writing specific tests to ensure our functionality is working and we have permissions to perform the action. I won't be providing all of the code for these directly in the book itself since they are incredibly verbose (wanted to keep the book less than 8000 pages), but they will be found in the [/tests/API/Companies/ directory in the repo](#).

Looks like we got our plan in order, let's get to some development!

Step 1: Create Database Migrations

For your resource you will need to create a database migration first. This will ensure that you have a place to store all of your data and everything is configured for the next layer. Since our resource is a company, let's run the following command:

```
php artisan make:migration added_companies_table --  
create=companies
```

A few things to note about this that will help as you develop your app.

First, I like to name all tables in the plural form. I don't know if there is a standard to this, but to me I think it makes the most sense since most of the time, you will

have multiple records in the table. In this case, our table will be `companies`.

Second, we defined our table name using the `--create=companies` flag after our migration. What this will do is provide a beautifully stubbed out migration with some boilerplate code in place with the proper name:

```
<?php

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

class AddedCompaniesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('companies', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
```

```
    Schema::dropIfExists('companies');
}
}
```

Even though the flag is not required, it helps to boilerplate out the migration even before it's written. If you are interested, our initial company migration looks like this:

```
/**
 * Run the migrations.
 *
 * @return void
 */
public function up()
{
    Schema::create('companies', function (Blueprint
$table) {
        $table->bigIncrements('id');
        $table->string('name');
        $table->integer('roaster');
        $table->integer('subscription');
        $table->text('description');
        $table->string('website');
        $table->string('address');
        $table->string('city');
        $table->string('state');
        $table->string('zip');
        $table->string('facebook_url');
        $table->string('twitter_url');
        $table->string('instagram_url');
        $table->bigInteger('added_by')->unsigned();
        $table->foreign('added_by')->references('id')-
>on('users');
        $table->softDeletes();
        $table->timestamps();
    });
}
```

```
});  
}
```

The only unique thing we have right now is the `added_by` field. This references the `id` of the user that added the company. We are also doing soft deletes and flagging the companies once they are deleted so we don't end up with data loss. Right now, we have a nice table ready to store our data!

A quick note on the deletion we are going to use. We will be using soft deletes which actually doesn't delete the resource, but essentially flag it as deleted so we maintain the persistence of the data. For more information on [Soft Deletions check out the Laravel documentation](#). This is extremely convenient since we will have all sorts of data relationships. We don't want to break our database and lose history, but we also don't want to display the data if we don't need it. Laravel Soft Deletes make this super easy!

Step 2: Build Required Models

Now that we have our database built, the next step will be to create our models. In Laravel, you can use [Eloquent](#) to map the fields in the database through code and make it easy to manage. To do this, we need to create a model for our new resource.

What we need to add a file to the `app\Models` directory named `Company.php` and filled with the following code:

```
<?php  
namespace App\Models;  
  
use Illuminate\Database\Eloquent\Model;  
use Illuminate\Database\Eloquent\SoftDeletes;  
  
class Company extends Model  
{
```

```
use SoftDeletes;

protected $table = 'companies';

public function addedBy(){
    return $this->hasOne('App\Models\User', 'id',
'added_by');
}

}
```

When adding your own models, you need to do the following.

1. Remember when we moved the `User.php` file to the `app\Models` directory? Make sure you use that as the namespace.
2. Make sure you have your model extend the `Illuminate\Database\Eloquent\Model` or you won't get access to any of Eloquent's model features.
3. As mentioned before, we are also using `SoftDeletes` which allow us to maintain the data, but not use it in our queries. To enable `SoftDeletes` on a model, add the following `use Illuminate\Database\Eloquent\SoftDeletes;` and make sure that the `use SoftDeletes` trait is set.
4. Finally, and this is my own personal recommendation, set the `$table` variable. This is extremely important for situations where plurals have different suffixes. For example, it's `companies` not `companys`. No matter what, even if it's easy to decipher for Eloquent, I explicitly set this so there's no unnecessary confusions.

I also added our `addedBy()` relationship. This is a single, one directional relationship to our `User` model. Like I mentioned when we added the `added_by` field, this will reference our `users` table. Each company is added by `one` user. By configuring this through Eloquent, we can easily make relational calls if needed.

Step 3: Define Routes and Create Controller

So we have our data store and have our resource modeled for easy access. Now we need to define the ways to access our resource through the API. These will just be stubbed out routes, but we just keep building on top of what we already have!

The first thing we need to do is open up our `routes/api.php` file and add the routes that we discussed for Viewing, Creating, Updating, and Deleting Companies.

In our app, we will add the following code within our `v1` prefix:

```
<?php
Route::group(['prefix' => 'v1'], function(){
    ...

    /**
     * Company Routes
     */
    Route::get('/companies',
    'API\CompanyController@index');
    Route::get('/companies/{company}',
    'API\CompanyController@show');
    Route::post('/companies',
    'API\CompanyController@store');
    Route::put('/companies/{company}',
    'API\CompanyController@update');
    Route::delete('/companies/{company}',
    'API\CompanyController@destroy');
});
```

These five routes provide access to our company resource. If you were to look ahead, you could guess that the next thing we will be doing is adding a `CompanyController.php` in our `API` namespace. You'd be correct. Before we get there though, a couple features to point out about these.

First, just to re-iterate, since the routes are in the `routes/api.php` file, the resolvable route will be prefixed with `/api`. We also added the routes within a route group that adds an additional prefix of `/v1` to each route. That's how we get final, resolvable routes like `/api/v1/companies`.

Next, anytime you see `{company}` or any term wrapped in `{}` that's going to be the variable being passed to the controller. When we implement our `CompanyController.php` the Company will be type hinted based off of this variable. This will allow for some cleaner code and not so many queries we have to write.

Finally, all of these routes resolve to methods names specifically for what type of action is taking place. These kind of routes are called "Resource Routes", imagine that? For more information on "Resource Routes" within Laravel, [they have amazing documentation](#). Since this is a pretty straight forward resource, you can generate it by a command and modify as needed.

To generate our resource controller, run:

```
php artisan make:controller API/CompanyController --resource  
--model=Models/Company
```

It's kind of a lengthy command and honestly, I don't run it often. There are few times throughout the course of API development that you are adding an entirely new resource or one that's fully fledged out. Smaller resources that don't have all of their functionality publicly exposed, can just be created as new files.

Anyways, let's break down that command. First, it's a straight forward `php artisan` command that makes a controller. That part is pretty explanatory if you've worked with Laravel before. If you haven't I'd suggest reading more about [artisan commands](#). There are a ton of useful commands you can run through the console.

Where the command starts to get more customized for the resource and more

specific to API dev is the first option of `API/CompanyController`. This is telling the command that we want to build a controller in the `App\Http\Controllers\API` namespace and call it `CompanyController.php`. It's super useful, otherwise Laravel defaults to placing it in the `App\Http\Controllers\` namespace.

The next flag of `--resource` is helpful too. All those routes and methods we implemented? Automagically get stubbed out when we add this flag. Our controller will have every method that we implemented named correctly with the `resource controller` naming scheme. Super helpful when developing new resources! However there will be a few routes to remove and we will get to that after we talk about the `--model` flag.

Finally, we have the `--model=Models/Company` flag. What this does is type hint all of our routes with the `App/Models/Company` object so they are all resolved correctly. Isn't that awesome? And it's all documented to boot! By default, this would use any models in the `/app` directory, but since we moved all models to the `/app/Models` directory, we have to explicitly state which namespace we are using.

If we open up our newly created `App\Http\Controllers\API\CompanyController.php` file you will see a whole bunch of stubbed out methods. Feel free to remove the following:

- `create()`
- `edit()`

These are to display pages in a standard “monolith” app and we will not be implementing them. Our controller should look like:

```
<?php  
  
namespace App\Http\Controllers\API;
```

```
use App\Http\Controllers\Controller;
use App\Models\Company;
use Illuminate\Http\Request;

class CompanyController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param \Illuminate\Http\Request $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        //
    }

    /**
     * Display the specified resource.
     *
     * @param \App\Models\Company $company
     * @return \Illuminate\Http\Response
     */
    public function show(Company $company)
    {
        //
    }
}
```

```
}

/**
 * Update the specified resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @param \App\Models\Company $company
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, Company
$company)
{
    //
}

/**
 * Remove the specified resource from storage.
 *
 * @param \App\Models\Company $company
 * @return \Illuminate\Http\Response
 */
public function destroy(Company $company)
{
    //
}
```

Everything is nicely laid out and formatted! That's one of the most beautiful parts of Laravel. Helper functions and commands to make your life a breeze! That's pretty much all we need to do to get our routes and controllers set up. Just a heads up, we will be updating some of the request validations as we write our own validations for these routes, but for now this is perfect. Time to implement our functionality!

Step 4: Build Our Services

Now that we have our routes created and our controller to handle the requests, it's time to remove the functionality from our controllers and into specified services. This makes code re-use much easier and can help organize the structure of the app. Let's say we have a form that can submit a new company if needed (like when we add a cafe ;)), it's nice to have the code to add the company be re-usable.

I like to design services as an object that encapsulates as much as possible about performing a certain function. These will break down the process into small, consumable, and re-usable methods.

The first service that we will add will be to create a company, so let's add the file `app/Services/Company/CreateCompany.php`. This file will contain all of the methods that we need to create a company. Any sub queries or small methods that help this process along will all be in this file.

The shell of our file will be as follows:

```
namespace App\Services\Company;

use App\Models\Company;

class CreateCompany {
    private $data;

    public function __construct( $newCompanyData )
    {
        $this->data = $newCompanyData;
    }

    public function save()
    {
```

```
    }  
}
```

Instead of going through every single one of our services, I'll just summarize. Reason being is these are "App Specific" logic that has nothing to do with the theme of the book. I will show you how we implement the services and by all means, check it out on the [Gitlab repo](#) if you are interested. We will have method on this service class with a `__construct()` method which accepts `$newCompanyData` as it's parameter. We also have a `save()` method that will take all of the information in the `$newCompanyData` variable, process it and save it as a company record.

When setting up the service, we just need to provide a namespace and declare that we use the `App\Models\Company` model within the service. Our service will essentially be preparing our request for persistent storage within the database so we need access to that model.

We will need to create a service for each request. I namespace this all within the `\Company` namespace in the `App\Services` directory so they are all grouped nicely. Now with `GET` requests and `DELETE` requests, we don't have too much action going on right now to think that we'd need a service. However, that will change as we will be adding parameters to build queries, actions to be emitted, etc. Say if we want to provide in-depth queries, the `GET` service will be massive. Same with the `DELETE` service if we want to insure all child components are delete upon deletion of a parent component. Trust me, they can get very large very fast. This is all specific logic to the application and any of that can be browsed in Gitlab.

Using a service is simple. We first need to include the services within our controller that handles the company requests. Let's open up `App\Controllers\API\CompanyController.php` and add the following line to include our `CreateCompany` service to the top:

```
use App\Services\Company\CreateCompany;
```

Perfect! Now, within our `store()` method, we just add the following code:

```
/*
 * Store a newly created resource in storage.
 *
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    $createCompany = new CreateCompany( $request->all() );
    $newCompany = $createCompany->save();

    return response()->json( $newCompany, 201 );
}
```

What we do is we pass all of the data from the `$request` to the `CreateCompany` service. The next line, we call the `save()` method that, when fully implemented, returns the new company object. When all said and done, we return the `$newCompany` and a `201` response code that signals that a new resource has been added.

When returning RESTful responses, it's important to add the proper response code and to return as JSON (or XML if that's your thing). To do this, you can use the beautiful methods already provided by Laravel. If you see with the new company request, we run the following return statement:

```
return response()->json( $newCompany, 201 );
```

This returns a response which is formatted as `json()`. The `json()` method takes two parameters that we use heavily (4 if you want options and other headers). Those two parameters are `data` and `status`. The `data` can be any resource or piece of data that you want to return and the `status` can be any HTTP status code. For returning a new company, this is `201` response code which signals that a new resource has been created. For deleting or updating a company and returning

an empty response, we pass `null` for the `data` and `204` for the status:

```
return response()->json( null, 204 );
```

These helper functions by Laravel are amazing and really help contribute to the ease of API Driven Development.

The other thing I'd like to point out, see how clean the controller looks? That's because we took all of the heavy business logic out of there and focused on letting the controller do what it does best, control. It handed off that functionality to our service which in turn did what it needed to do and returns the proper request! All of our other services will follow a similar approach. If there's anything specific to the API, I'll make sure to cover it!

Now, let's rock and roll to our security!

Step 5: Security Through Middleware and Request Validations

So there's two layers of security we are going to apply to our resource at this point. The first layer of security will be middleware. As I mentioned earlier we will want to ensure that there is an authenticated user accessing the route to create, update, and delete companies. This is an easy check that can happen through middleware.

Second, we will want to apply a request validation to each route where we manage data. Those are the create and update company routes. What this does is validate the data being submitted to ensure it is valid before persisting it. These request validations are extremely helpful because we can configure them to be validate the data before it hits any of our business logic. This is efficient not only for security purposes and for submitting a bad request, but for providing quality feedback to the user so they can make changes and re-submit the form.

Let's start with the outer layer of the shell, the middleware. The way to approach the validations will be kind of like an onion method. Middleware being the outermost layer, gates and policies (which we will talk about later in the book) the second layer, and request validations the third layer before running the request. The reason being is it's kind of like a tiered process, each layer blocks a more granular detail of the request. You will see this more as we dive in and apply the security a few times.

Middleware

To apply middleware to routes, you can do it a variety of ways. First, you can chain it to the end of the route definition. Second, you can apply it through a route group within the route file. Third, you can actually apply it in the controller handling the request through the `__construct()` method.

The third approach is the way we will be assigning middleware. There is nothing wrong with any of the ways and you can choose to do this as all ways support

multiple middlewares and custom middleware. The reason I choose the third approach, is it really keeps the `/routes/api.php` file clean. You don't have to have extra code hanging off of your route definitions, you don't have to group routes together based off of middleware, you are forced to use controllers (which you should be doing anyway) and you can have fine grained control over everything.

There is specific [documentation for middleware](#), but the documentation for [controller based middleware](#) is what we will be focusing on. In this first resource, we won't actually be creating a middleware since all we need to do is check to see if a user is authenticated or not. However, just a heads up, if I use any custom middleware I ALWAYS create a file. You can do a call back function, but I find that to be messy. I will go through custom middleware creation later on. For now, just know that's the way we are going to approach this.

Since we are using Laravel Sanctum, we have a middleware to use to ensure the user is authenticated. That middleware is `auth:sanctum` and it ships with the package. To apply this middleware, open up your `app\Http\Controllers\API\CompanyController.php` and add the following method on top of the class:

```
public function __construct()
{
    $this->middleware('auth:sanctum');
}
```

What this does is apply the middleware to the routes that the controller handles. Sounds perfect, right? Well, we only want to apply the middleware to routes that manage the data! These are the ones that require authentication.

No worries! Laravel provides a simple solution to this. You can apply middleware to certain routes by chaining the routes you want to include or exclude using `only()` or `except()` to the end of the middleware definition. In our case we want the middleware applied to only the `store()`, `update()`, and `destroy()` methods.

To do that, append the following to the middleware call:

```
$this->middleware('auth:sanctum')->only(['store', 'update',  
'destroy']);
```

Now the middleware is applied to only those methods which handle the routes we needed to apply the middleware to. The other thing to note about the controller middleware is that it's called in order. If you have any other middleware AFTER the authentication middleware in the code, it will wait until the authentication middleware runs before the next middleware. That's really the only thing we need to do to apply middleware to our resource!

Request Validations

Now we just need to create a few route validations. Luckily, Laravel provides a ton of cool validations ready for us out of the box.

A few notes about how we will be validating our routes. First, we need to figure out which routes handle form requests. Right now, that's only `store()` for creation, and `update()` for updating a company. Next, we will be using the [Form Request Validation](#). What this does is makes an individual file that handles complex validation for a request. Once again, I like the individual file approach since it really cleans up the code and encapsulates all of the features you need in one simple file. You can extend this form request to not only validate the data, but also if the user is authorized (more when we get into permissions) and what the error messages are.

Let's create our first form request for the `store()` method. The first thing we need to do is run the following command in our terminal:

```
php artisan make:request API\\Company\\CreateCompanyRequest
```

What this does is creates a `CreateCompanyRequest` in the `app/Http/Requests/API/Company` namespace. I like keeping code as organized as possible so instead of storing this code in the `app/Http/Requests` directory, I

added the `API\Company` namespace to ensure we have proper structure. This will help us a ton down the line when we end up with hundreds of requests to validate.

Upon initialization, our request should look like:

```
<?php

namespace App\Http\Requests\API\Company;

use Illuminate\Foundation\Http\FormRequest;

class CreateCompanyRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this
     * request.
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
        ];
    }
}
```

Currently, it initializes with two methods: `authorize()` and `rules()`. The `authorize()` method returns a boolean if the user is authorized to submit the request. When we get into permissions later on, we will be working with this method. For now, change it to return `true`:

```
/**  
 * Determine if the user is authorized to make this request.  
 *  
 * @return bool  
 */  
public function authorize()  
{  
    return true;  
}
```

The request is already behind middleware for if the user is authenticated. Next, we need to apply our rules. This is where your app will be completely customized and a lot of your logic will need to be tailored to your needs. To help better understand how this works, I'll go through our rules. The rules we go through throughout the entire book will hardly scratch the surface of what's [available with Laravel](#). If that's not enough and you need something really intense, you can create your own rules too. This gives you fine grained full control over your application.

I'll show you a few of these rules with the creation of a company. To create a company, we will have the following fields with these validations for now:

Name

- required
- string

Roaster

- required
- integer

Subscription

- required
- integer

Description

- string

Website

- url

Address

- required
- string

City

- required
- string

State

- required
- string
- size:2

Zip

- required
- numeric

Facebook URL

- nullable
- url

Twitter URL

- nullable
- url

Instagram URL

- nullable
- url

That looks super overwhelming, but with Laravel, it's straight forward. The definition of validations is as follows:

- required → The field must be present on the request
- numeric → The field must be numeric
- string → The field must be a string value
- url → The field must be a URL
- size:2 → The field must have a size/length of 2
- integer → The field must be an integer
- nullable → Only validate the field if it's not null

Combined, we can create very efficient validations and make sure we are getting quality data on every request. Let's see how our rules array looks when we handle these validations:

```
/**
```

```
* Get the validation rules that apply to the request.  
*  
* @return array  
*/  
public function rules()  
{  
    return [  
        'name' => 'required|string',  
        'roaster' => 'required|integer',  
        'subscription' => 'required|integer',  
        'description' => 'string',  
        'website' => 'url',  
        'address' => 'required|string',  
        'city' => 'required|string',  
        'state' => 'required|string|size:2',  
        'zip' => 'required|numeric',  
        'facebook_url' => 'url',  
        'twitter_url' => 'url',  
        'instagram_url' => 'url'  
    ];  
}
```

You can chain together as many validations as you want! When you see a `:` like in `size:2` it's passing a custom parameter to the method stating the size must be 2 characters. Now we just need to apply this validation to our route. To do that, open up [app\Http\Controllers\API\CompanyController.php](#).

On top of the file in your `use` declaration section, add the following line:

```
use App\Http\Requests\API\Company\CreateCompanyRequest;
```

This will allow us to use this request in our controller. Next, find the `store()` method. Upon the creation of our controller, the `store()` method's signature looks like:

```
/**  
 * Store a newly created resource in storage.  
 *  
 * @param \Illuminate\Http\Request $request
```

```
* @return \Illuminate\Http\Response
*/
public function store(Request $request)
{
    ...
}
```

The first parameter is `Request $request`. We need to change this to the `CreateCompanyRequest`. Let's change the signature to:

```
/**
 * Store a newly created resource in storage.
 *
 * @param App\Http\Requests\API\Company\CreateCompanyRequest $request
 * @return \Illuminate\Http\Response
 */
public function store(CreateCompanyRequest $request)
{
    ...
}
```

Now the method will operate the exact same and we will still have access to our request data. However, it will be validated through our request validator before it hits the business logic!

Finally, let's customize the error messages that get returned to make the whole validation process a lot more user friendly.

We first need to override the `messages()` method in the `CreateCompanyRequest.php` file. So add the following:

```
/**
 * Get the error messages for the defined validation rules.
 *
 * @return array
 */
public function messages()
```

```
{  
    return [  
        ];  
}
```

Now, we can override the messages with some custom information for each validation. To do this, you simply set the key of the return array to be field you are testing with a `.` and the validation rule. For example to customize the error message for the `name` field being `required` it'd be `name.required` as the key, and the value would be “The name of the company is required”:

```
/**  
 * Get the error messages for the defined validation rules.  
 *  
 * @return array  
 */  
public function messages()  
{  
    return [  
        'name.required' => 'The name of the company is  
required',  
        'name.string'  => 'The name must be a string',  
        'roaster.required' => 'A flag declaring the company  
as a roaster is required',  
        ...  
    ];  
}
```

You can go down the line with all of your fields + validations and return proper text for each one. As we develop more complexity to our application, these route validations will be extremely helpful and we will be working with much more complex validations such as `unique` and `required_if`. When we use complex validations, I'll explain their use. Otherwise, I won't waste your time going through the basics. The `update` route's validations are pretty much the same. If you want

to see all of the validations, definitely check out Gitlab. All of the code will be in there and if you need a hand, reach out!

From here, we have our resource successfully managed! We can now create, get, update, and delete resources, we've validated the routes, modeled the resource. Finally, we have to test these routes and begin our foray into API testing!

Step 6: Testing

Now that we have everything set up, we need to test our resource. Laravel provides all of the necessary tools to properly test our API. However, there are a few tricks that we need to incorporate in to our testing that I'll go through to ensure we can properly test API routes.

The majority of our tests will be [HTTP Tests](#). More specifically, [testing JSON APIs](#). This means we will make a request to a URL, possibly send data, and ensure that data is valid and authentication is either required or not.

That's a top level overview, but the tests we will be making will be much more sophisticated. I won't be running through each individual test in detail or we'd have hundreds of pages of testing code. I'll be running through the gotchas that come with API testing, providing examples, and explaining how to approach testing. Like everything else, all code will be in Gitlab for you to look at and review.

Like everything, I like to make buckets for code so we have a place to dump everything. With the verbosity of testing, we will want to make a ton of files and write tons of tests. Having a bucket system in place really helps keep this mess of code organized. Let's first create the following directory [/tests/API/Company](#).

The first test we will write will be to test the loading of all of our companies. This is a great place to begin learning how to test your API. Inside our new directory, add a [LoadCompaniesTest.php](#) file:

```
<?php

namespace Tests\Feature\API\Company;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class LoadCompaniesTest extends TestCase
```

```
{  
    use RefreshDatabase;  
}
```

This is just the shell of our test case. We will have ton of tests in here, all stored in individual methods. Before we add our first test case, I'll point out that all of the methods that perform tests begin with the `test` prefix. This ensures that PHPUnit will know to run that method as a test. We have this configured in our `phpunit.xml` file. All of our files that contain tests end in `Test.php` as well. Keep that in mind as we add more files.

The one thing to notice is we ensure to use the trait `RefreshDatabase`. What this does is run a migration and refresh our database for each test. This may take time, but we are heavily relying on proper data structure, it's a necessity to ensure we have our new migrations run on our testing database and that the requests work correctly.

Let's add a simple test that hits our API:

```
public function testNoCompaniesLoad()  
{  
    $response = $this->getJson('/api/v1/companies');  
  
    $response  
        ->assertStatus(200)  
        ->assertJsonCount(0);  
}
```

What this test does is sends a `GET` request that will return JSON to the url in the first parameter. We then receive the response and run our assertions. For those who are beginning with testing, assertions are our actual tests. Think of it this way, "when I get a response, I want to assert that these things are valid about my response".

In this case, we are asserting that the status was `200` (a success response

message) and that there are 0 entities in the array. Remember, our test database refreshes and we are using an actual testing database defined when we set up Laravel. This database does not have any of the companies that may have been added through out the development. So in this case, 0 is a successful response.

There are so many available assertions that you can try, and Laravel provides a [wonderful list of what's available](#).

Right away, I will warn you, your tests will get infinitely complex. We really need to structure the code in a way that is easy to come back to and modify if needed. Here are a few tricks.

Setup

Let's say you need to configure certain properties and set up your test EACH time you need to run a test in your test suite. You can add a `setUp()` method to your testing class. In that method, make sure you call `parent::setUp()` so all of your parent level tests are configured as well. This will override the `setUp()` method provided by the `TestCase` and allow you to run local setup:

```
protected function setUp(): void
{
    parent::setUp();

    // Your setup here
}
```

When setting up test cases, I like to initialize any users we'd need to `act as` during the test (guest, authenticated user, admin user, etc), any resources we need to add to the database so we can ensure data validity, or any local resources that we will use throughout our testing suite. This will save tons of repeated code and make our tests way more concise. Initializing resources however requires a combination of `factories` and `faker`.

Factories and Faker

The first question is, what are factories? [Factories](#), are objects that allow you to insert records into your database easily while testing. You can generate as many records in the database using your Eloquent models as you want. All of your factories are stored in the `database/factories` directory and can be used in a variety of scenarios to easily create testing data.

Factories work best with a PHP library called [Faker](#) that ships with Laravel by default. This library literally generates tons of fake data that you can use with your factory. This way you don't have to think about coming up with a load of unique information to simulate, you just run methods through [Faker](#). We will be using factories and faker a ton as we test our API. After I run through a quick overview, we will combine all of this into another test.

Acting As

This is one of the most important features of API tests (or really any test) and even more important when permissions are involved. You can create fake users through your factories and test features as the user. This will allow you to get access to restricted routes as both an admin or a general user and ensure your API reacts accordingly. As we create permissions it will ensure that certain routes block user permissions and certain routes accept user permissions. I'll run through an example of this when we create a company through a test as well.

Taking testing to the next level

Alright, let's expand on our original test suite. This time however we want to incorporate the factories and ensure that a company is in the database and returned. The approach I would take would be first, add `database/factories/CompanyFactory.php` and add the following code:

```
<?php
```

```
use Faker\Generator as Faker;

$factory->define(App\Models\Company::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
        'roaster' => 1,
        'subscription' => 0,
        'description' => $faker->text( 300 ),
        'website' => $faker->url,
        'address' => $faker->streetAddress,
        'city' => $faker->city,
        'state' => $faker->stateAbbr,
        'zip' => $faker->postcode,
        'facebook_url' => $faker->url,
        'twitter_url' => $faker->url,
        'instagram_url' => $faker->url
    ];
});
```

All of these faker properties are listed in the [faker documentation](#).

You can literally fake pretty much any piece of data which is so valuable for testing. All of the faked variables that we've defined, those will be used as defaults when we start to use the factories in our testing.

Now that we have our factory in place, let's go back to our `LoadCompaniesTest.php` and add the following test:

```
public function testFiveCompaniesLoaded()
{
    $user = factory(\App\Models\User::class)->create();
    factory(\App\Models\Company::class, 5)->create([
        'added_by' => $user->id
    ]);
}
```

```
$response = $this->getJson('/api/v1/companies');

$response
    ->assertStatus(200)
    ->assertJsonCount(5);
}
```

This is definitely a little more complex than our last test, but let's walk through it.

The first line `$user = factory(\App\Models\User::class)->create();` uses the `UserFactory` that ships with Laravel to create a user. Remember, we edited this factory when we Configured Laravel. In our app, we keep track of who added a company so we needed at least one user in our system.

The next line:

```
factory(\App\Models\Company::class, 5)->create([
    'added_by' => $user->id
]);
```

uses the factory that we just created and passes a second parameter of `5` to the `factory()` method. This second parameter tells our factory to save `5` of the specific model into the database. We also pass an array to the `create()` method. In our factory, we've provided a whole bunch of defaults through faker. When we pass an array of values through the `create()` method, we can override those defaults. In this example, we pass the `added_by` method to be the `id` of the user we created with our factory. This is important otherwise we will end up with errors upon insert.

Now the final pieces of the test first assert that the status is `200` which is successful, but unlike our first test, we assert that `5` companies are returned since we created `5` of them. If we run our tests, we should see the following results:

```
...
3 / 3 (100%)
Time: 987 ms, Memory: 24.00 MB
OK (3 tests, 5 assertions)
```

Once it's green, we are ready to go!

Before we move on to displaying our new resource in our SPA, let's do one quick run through with testing our API by adding a resource. This is a little different since 1, we have to have an authenticated user and 2 we have to confirm that what we add is persisted in the database correctly.

I'll not only go through as many gotchas as possible while we develop the app, but I also have an appendix that organizes them when you try to implement them in your application. You can check that out and you can also check out the Gitlab repo to see how we've implemented everything.

First, let's create this file in the `/tests/API/Company` directory with the name `CreateCompanyTest.php`:

```
<?php

namespace Tests\Feature\API\Company;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class CreateCompanyTest extends TestCase
{
    use RefreshDatabase;
```

```
protected function setUp(): void
{
    parent::setUp();
}

public function testCreateCompany()
{
}

}
```

We now have a shelled out test that will get picked up when we run our testing command. Now within our `testCreateCompany()` method add the following code:

```
public function testCreateCompany()
{
    $user = factory(\App\Models\User::class)->create();

    $this->actingAs( $user )
        ->json( 'POST', '/api/v1/companies', [
            'name' => 'Ruby Coffee',
            'roaster' => 1,
            'subscription' => 1,
            'description' => 'Colorful Coffees',
            'website' => 'https://rubycoffeeroasters.com/',
            'address' => '9515 Water St',
            'city' => 'Nelsonville',
            'state' => 'WI',
            'zip' => '54407',
            'facebook_url' => 'https://www.facebook.com/
rubyroasters',
            'twitter_url' => 'https://twitter.com/
rubyroasters',
        ]);
}
```

```
    'instagram_url' => 'http://instagram.com/
rubyroasters'
    ])>assertStatus( 201 );
/*
Ensures the database has the proper information.
*/
$this->assertDatabaseHas('companies', [
    'name' => 'Ruby Coffee',
    'added_by' => $user->id
]);
}
```

The first thing to notice we are not using the factory. This is because we want to test if our endpoint actually adds the company and not the factory. So let's step through this.

The first thing we need to do is create a user. Since our permissions are set up as you only need to be authenticated, that's all we need. The next line, we set up our tests to `actingAs($user)`. What this does, is takes the user that we created, and makes the request as the user. This gives us a user to run the request as.

We also chain the `→json()` method which sends a `POST` request to the url we have in our API to create a company: `/api/v1/companies`. Then it passes all of the data needed to create a company.

Within that chain, we end with the assertion that the status is a `201` which means a new resource has been successfully created. That's not the end though, we want to confirm the resource is actually in the database. If you look at the last section of the method, you will see:

```
/*
Ensures the database has the proper information.
*/
$this->assertDatabaseHas('companies', [
    'name' => 'Ruby Coffee',
```

```
'added_by' => $user->id  
]);
```

This assertion then checks our database and finds the `companies` table. Then confirms that in that table there is a company with the `name` that matches the name of the company we just added and the `added_by` which is the field that houses the ID of the user we were `actingAs`. You can pass as many different keys to your array to confirm it matches what was added.

Those are some basic testing scenarios! If you check out the ROAST Gitlab there will be a variety more tests we created for this resource and can browse them all there. You can also check out the appendix for more gotchas with API Driven Development testing.

Conclusion

Our first API resource is 100% complete. We can now access this resource from the frontend. As we move forward, we will be following this lifecycle over and over. Some resources/enhancements will be less complex, some way more complex. Either way, getting used to how to develop for an API is essential to API Driven Development. Now, let's see the fruits of our labor and begin adding a frontend to our application with NuxtJS!

Using Our API Endpoints With NuxtJS

We've got our first API resource added! Now, let's make it visible! With the separation of concerns between our API and our SPA, this entire section will take place on the NuxtJS SPA side so make sure you open up the `/spa` directory in your editor before you begin!

Step 1: Create our Company Resource

Now it's time to implement our reusable API modules (Abstracting Your API Calls Into Reusable Modules).

We are now going to add our first API resource for the Company resource that we just completed! This allows us to have a centralized location to call all of our resource's routes from, so we can re-use the code wherever we need!

Adding a Company Resource

The first thing we need to do is add a file to the `/api` directory within NuxtJS named `companies.js`. The file should have the following code:

```
export default $axios => ({  
});
```

What we did is extend the NuxtJS axios module and prepare it to house our API routes. Within this method, we will be working with `async/await` methods that allow us to seamlessly load data from the API asynchronously.

If you are unfamiliar with `async/await`, don't worry. I, too, emerged from the land of promises and callbacks. All `async/await` is a beautiful syntax that wraps the promise so it flows much easier. [Javascript.info](#) is an incredible resource for more information that thoroughly explains how [async/await works](#). Let's add our first

endpoint.

Add GET resource endpoint

The first endpoint we should add should be our `/api/v1/companies` endpoint since our example is a Company resource. To do this, add the following method to your `/api/companies.js` resource:

```
async index(){
  try{
    return await $axios.$get('/api/v1/companies');
  } catch ( err ){
    }
},
```

Let's break this down a little bit. First, notice the `async` keyword in front of `index()`? That signals that we are going to be running the method asynchronously. By adding this keyword we have to have an `await` inside of the method that is our promise, which we do. But let's break down the `try/catch` block before that.

We added the `try/catch` block around the request to catch any errors that may arise. This is extremely important for error handling and will touch on this a little later in a specific section regarding handling request errors within NuxtJS. For now, know that we will be returning the result of the promise as long as everything is successful.

Within the `try` block, we return the `await` of the axios request to our `companies` endpoint. What this does is allow us to return the successful result of the promise when it has been completed. If the request fails, then we can catch the error and deal with it accordingly.

That's our first endpoint within NuxtJS! Now, in order to run this method, we just have to run `this.$api.companies.index()` within a component and it will

make a request to our API. We will have to wrap this call in `async/await` as well since we are running it asynchronously, but we will get to that in Step 2.

Another quick thing to note, is we named this method `index()`. The reason we are doing this is to keep consistent with our API's resource routes. This will help us down the line as we continue our development. Let's add a few more endpoints to match our API.

Add POST companies endpoint

This is our request to create a company. The difference between this request and the `GET` request to the `/api/v1/companies` endpoint is that we are going to be passing data, so this is structured a little bit differently. We need to add this method to our `companies.js` resource:

```
async store( company ){
  try{
    return await $axios.$post('/api/v1/companies',
  company );
  } catch ( err ){
    }
}
```

The parameter we pass is the `company` object. This object will represent a new company when submitted to the API. It will contain all of the fields and information necessary to add this resource.

You might wonder why we didn't pass every parameter that we need to this method? Good question. You can, but it can get pretty hairy when you have to update every method call and really clutters up the code. With this approach, we are leaving the validations to whatever page or component that handles the creation of a new resource. This way we let that piece of code manage and validate the data before we send it off to the API endpoint. You will see how this works when we build out our calls.

Add **GET** company endpoint, **PUT** update company endpoint, and **DELETE** company endpoint

We've already covered the ins and outs of how to add the endpoints for the basic management of resources. To finish up, we will add the next 3 at the same time since they follow a very similar approach:

```
async delete( id ){
    try{
        return await $axios.$delete('/api/v1/
companies/'+id);
    } catch ( err ){
        }
    },
}

async show( id ){
    try {
        return await $axios.$get('/api/v1/companies/'+id);
    } catch ( err ){
        }
    },
}

async update( id, company ){
    try{
        return await $axios.$put('/api/v1/companies/'+id,
company );
    } catch ( err ){
        }
    },
}
```

The only thing to note about these methods is they accept an **id** parameter which helps you access a specific resource. This is appended to the end of the endpoint

so we know which resource we are working with.

PUT is also the same methodology as the **POST** with passing the entire company to update. This helps keep our final resource nice and clean so it's easily maintainable. Now, let's figure out how to use this resource and implement a couple pages.

Step 2: Where should we store our data?

Since the first implementation we are working with will be loading all of our resource, we need to figure out where to store this data. This is always the next question to ask, because with Vue and NuxtJS, there are a couple options and if we choose the Vuex option, we should shell out our store with all of the actions, getters, mutations, etc.

The first option is in the local data parameter which can be either in a component or on a page. The second option is in the Vuex data store that's global and accessible from everywhere. While there's no clear cut way which is right, I like to approach it with these questions:

1. Will the data be used in another page?
2. Will we have to pass the data meticulously down through props to another component?

If these questions are both no, then I say store the data in the individual page or local component. If either of these answers are yes, then store it in the Vuex data store.

I like to think of the Vuex data store is strictly the state of our application. For example, if we are setting some query parameters for companies and those should persist from page to page, then I'd store it in the Vuex data store.

However, with this specific example, we are going to store it in the local page, and that's for all routes. We won't need a list of our resource (company) or save any edits that are necessary outside of the page we are on. We will use Vuex later on

with pre-auth actions and notifications.

Step 3: Plan your front end URLs

It's finally time to visualize our new features! To do this, we will add a few pages that we can access our resource at!

This is extremely important. These URLs will be what the user will see when accessing your app. Remember, as we talked about when we discussed adding pages, these routes will be generated for you depending on the filename and directory structure that your page resides in. To brush up, take a look back at Understanding NuxtJS' Structure.

Now we have 4 endpoints that should have front end pages associated with them. We have a page that should display all companies, a page to create a company, a page to update a company, and a page to view an individual company. We should also realize that we don't want unauthenticated users accessing the create company and update company routes. With that in mind, let's get started.

If I were looking at the URL of the app, I'd expect these routes to be within the `/companies` frontend endpoint. I'd map the front end endpoints to their functions like this:

- All companies → `/companies`
- Individual company → `/companies/{id}`
- Create company → `/companies/new`
- Update company → `/companies/{id}/edit`

To get the `/companies` prefix to these routes, add the `/pages/companies` directory. NuxtJS will then use that directory as a base for the rest of our endpoints.

We will have to create one more directory while we are at it and that is the `/`

`pages/companies/_id` directory. This is a unique directory as it will accept the `id` of the company we are viewing and display the content in relation to that. Now there are other ways like adding an `_id.vue` file to the `/pages/companies` directory to display a unique company, but in the future we will have nested routes underneath this individual company route that will be nice to have in the `_id` directory.

Our pages directory should now look like this:

```
/pages/companies  
/pages/companies/_id
```

These will be the buckets that we use for the next section to add pages.

Step 4: Add Pages

So to match the routes that we defined above, we need to add four pages to our directories that we created. The first will be `index.vue` to the `/pages/companies` directory. Next, we will add `new.vue` to the `/pages/companies` directory. These two pages serve two different purposes. For now, I'd have both of them be just a templated page with the following content:

```
<template>  
  <div>  
  
    </div>  
  </template>  
  <script>  
    export default {  
  
    }  
  </script>
```

The `/pages/companies/index.vue` will resolve to `/companies` and display all of

the companies in the application as of right now. The `/pages/companies/new.vue` will resolve to `/companies/new` and will display a form to add a company. The `new.vue` page will eventually be blocked off by authentication middleware, but for now, let's just leave it.

After we add those pages, we will have to add two more. Within the `/pages/companies/_id` directory, create the file `index.vue`. After that's created, create the `/pages/companies/_id/edit.vue` file. Both files should have the same starting page template as above.

Within our `/pages/companies/_id` directory, we have another `index.vue` file. Since this is nested within the `_id` directory this just resolves to `/companies/{id}` when accessed from the URL. The `edit.vue` file will resolve to `/companies/{id}/edit` and will display the page to edit the company.

We now have all of our front end routes shelled out and ready for implementation. Let's do a little restriction on some of these routes since we don't want unauthenticated users accessing the create company page and the edit company page.

Remember, we are using companies as our example, but make sure to name these accordingly for whatever resource you are adding!

Using our App Layout

Remember when we made our app layout (Building Your First Layout)? This is where we first implement it! In each of the pages you created, let's add the following code in the `<script>` tag:

```
export default {  
  ...  
  layout: 'App',  
  ...  
}
```

That's it! Our pages will now render within our router view and use our layout! The key `layout` accepts the name of a file within the `/layouts` directory. In this case the name of the file is `App.vue` and is linked via the name `App`.

Step 5: Restricting Access to Authenticated Users

For now, we will work with the native NuxtJS Auth module and use the provided middleware. The auth module by default provides an [auth middleware](#).

What we will need to do is open up the following pages:

- `/pages/companies/new.vue`
- `/pages/companies/_id/edit.vue`

Then in the module part of the components, we will need to add the following key:

```
export default {
  middleware: 'auth'
}
```

That's all we need to do! Now when an un-authenticated user accesses that route it will automatically re-direct them back to the `/` page. When we bridge into permissions, we will apply some newly created middleware to these routes based on whether the user is an admin or owner, but for now this works fine.

Step 6: Implement Companies Page

So let's start with the most simple of pages, the `/companies` page. This page will literally load up all of the companies resource in the application and display them on the page. What you can take away from this is an example of the first time we completed a full loop from our SPA to API and back to displaying data.

One thing I'd like to point out is if you look at the source code on Gitlab, this page doesn't exist. Why is that? The structure of our application doesn't have a list of just companies. However, it's a great starting example and you can take a look at

the Cafes resource if you want to see how this works. Once again, trying to be app agnostic, this should just guide through the process of how to add a resource from API to SPA.

Before we get started, let's answer our first question.

Should we use `asyncData`?

Yes. Since our API calls are asynchronous, making the endpoints accessible through `asyncData()` really cleans up our code. Also, when using SSR, this actually loads the data right away on initial page load so it can be used for meta data and SEO.

What is `asyncData()`? If you are familiar with VueJS, each component has a `data()` function that returns a JSON object of data we can use within our component. `asyncData()` is very similar except instead of being static, we return an object containing the result of the async promises.

There's two things to note before we get started. First, the first parameter is the `ctx` or context of the app. This is all of our modules, plugins, etc available for us to use. The second is, as the NuxtJS docs point out right away, that we won't have access to `$this` from within our component since all of it is rendered and called before initiating the component.

Upon resolving these pieces of data, NuxtJS will automatically merge it into the `data()` parameter so we can use it all within our component. Using `asyncData()` we just have to add the following to access our API:

```
async asyncData(ctx) {
  return {
    companies: await ctx.app.$api.companies.index()
  }
},
```

The above example is how we access our companies API and store it in a way we

can access it within the page.

The first parameter we pass in is the `ctx` or `context`. This allows us to access our `app` plugins and modules which in turn contains the decoupled `$api` plugin.

So now, we are initializing a piece of data named `companies` that will get merged within our `data()` and accessible in our component when the data resolves.

Remember, our entire API abstraction is `async/await`. Now any data we need from our API we can run through `asyncData()` and use it within our component when it's resolved. It's also available for server side rendering so we can add the proper JSON-LD and metadata for SEO purposes. Such a cool feature!

Another awesome feature of `asyncData`? We can handle failures. Say we are performing an initial request to a resource that doesn't exist. If the promise fails, we can return an error or 404 page. This will greatly enhance the UX of your SPA!

One thing to note about `asyncData` is it's only available on your pages. You won't be able to use `asyncData` within your components. If you need a piece of data, make sure you use it on your page component and pass it down to the child components accordingly!

Back to Implementing our /companies page

So back to our implementation, we just need to add the above code to our `/companies/index.vue` file like this:

```
export default {
  async asyncData(ctx) {
    return {
      companies: await ctx.app.$api.companies.index()
    }
  },
}
```

Now, within our `/companies` page, we have access to `$this.comapnies` within

our local methods and `companies` within our template.

Say we wanted to display a card for each of our companies. We can loop over our `companies` in the template like:

```
<card v-for="company in companies"
      v-bind:key="'company-' + company.id"
      :company="company"/>
```

Exactly the same as if you were using the local `data()` within the component! For now, that's all we will be adding to our `/company` page.

If you want to link to the individual company page that we will implement next. The link looks like:

```
<nuxt-link :to="/companies/' + company.id">
  ...
</nuxt-link>
```

We just compute the URL based off of the company provided. When the user clicks on the link, they will be brought to the individual company page.

Step 7: Implement the `/companies/{id}` page

On this page, we will be displaying an individual company, or, more generically, an individual resource. Now we won't get into much of the specifics on how this page looks since that's very app specific, but I wanted to touch on how we call our API route.

We will be using the `asyncData()` to call our `/api/v1/companies/{id}` route defined in the `api/companies.js` file. Now remember, we will be calling the `show()` method which accepts an `id` parameter. This is the `id` of the company we will be loading. Now the trick is accessing this id and accessing it BEFORE the page is initialized. That's because we are using `asyncData()`. To do this, add the

following code to the `/pages/companies/_id/index.vue` file:

```
async asyncData(ctx) {
  return {
    company: await ctx.app.$api.companies.show( ctx.params.id )
  }
},
```

The trick is to use `ctx.params.id`. These are the way to access the route parameters BEFORE the page is fully loaded. After the page is loaded, you can access this through Vue Router using `this.$route.params.id`. From here, we have access to a local `company` variable that we can use to display the page. Name this whatever the name of your resource is.

One thing to note about these individual pages. Sometimes if you load a whole list of resources like we did at `/companies` you'd think you could just grab the resource out of the array and display it on the page without making another request to the API.

This is rarely the case. Usually resource pages that list a whole bunch of resources don't contain all of the information we need for the individual page. For our example as we build the app, loading an individual company will load images, cafes, events, etc. which we wouldn't want returned when loading every company if we are just listing them out.

Step 8: Implement the `/companies/new` page

So this is the page that we will use to fill in data to create a company and send it to our API. Remember, we blocked this off with authentication so the user has to be authenticated before they can access this page.

The first thing we have to do is create a link to this page. I created my link on the `/`

pages/companies/index.vue page with the following link:

```
<nuxt-link :to="/companies/new"
v-if="$auth.loggedIn">
  &plus; Add Company
</nuxt-link>
```

What I want to point out is not only is the page blocked by the authentication middleware, I don't display the link unless the user is logged in. This link is only displayed if the user is authenticated and we can check by using the `$auth.loggedIn` and coupling it with: `v-if`. Now the link won't appear if the user is not authenticated. If they are, they can get to the new company page.

So the way I structure form submissions is similar to how we structured the login and register forms. We encapsulate all of our data into a `form` object. This way we can send the entire form to our API for processing. We also add all of our validations under the `validations` object. This way everything is clean and easy to reference.

For this, we are not using `asyncData()` we are using plain old `data()`. As an example, the data for our new company looks like this:

```
data(){
  return {
    form: {
      name: '',
      roaster: 0,
      subscription: 0,
      website: '',
      address: '',
      city: '',
      state: '',
      zip: '',
      description: ''
    }
  }
}
```

```
        facebook_url: '',
        twitter_url: '',
        instagram_url: ''
    },
    validations: {
        name: true,
        city: true,
        state: true
    }
}
```

We have all of our data to model located under `form` and all of our validations under `validations`. Here is where we work with another async method. This is the method to submit the company to the API. Since our `/api/companies.js` encapsulates our API requests and returns an async interface, any method that calls our resource should be asynchronous as well. As an example, let's add the following to the `/pages/companies/new.vue` file:

```
methods: {
    async addNewCompany(){
        if( this.validateNewCompany() ){
            const newCompany = await this.
$api.companies.store( this.form );

            this.$router.push({
                path: '/companies/' + newCompany.id
            });
        }
    },
    ...
}
```

Let's start at the top. First, we wrap our `addNewCompany()` method with `async` since it's an asynchronous method. Within the `addNewCompany()` method we call our `validateNewCompany()` method. This method, though not shown, simply returns a boolean if the new company is valid or not. If the form data is valid, then we continue, otherwise we break.

Next we have the meat and potatoes of the method, the call to the API to store our new company. Since our API abstraction is asynchronous we can apply `await` and handle the return data from `this.$api.companies.$store()`. This method accepts a JSON object of our form which it will send to our API. This is where the benefit of adding `form` to our local data comes in to play. Instead of passing X amount of parameters, we can just pass our `form` variable. We just have to make sure our form pieces of data match what is required by the API to add a new company and named accordingly.

Once the promise is resolved through `await` we save the response of the request to a variable named `newCompany`. Since we return the new resource on a create request, we have access to what we need in the next step, the new company id.

Once we receive the response from the request, we can redirect to the individual company page. That page will load all of the data we need to display the company, in this case the new company. To do that, we have the following code that accesses the Vue Router directly:

```
this.$router.push({
  path: '/companies/' + newCompany.id
});
```

That's all we need to do regarding adding a company! Next up we have editing, which will be very similar!

Step 9: Implementing /companies/{id}/edit

There are two distinct differences between adding and editing a company. The first is the method we use to send data to the API. This method is PUT instead of POST. The second is that we aren't really creating the data but editing it. So the first thing we need to do is "pre-populate" the form with the data we are editing. We call this "hydrating the form".

Hydrating the Form

So what hydrating the form means is that you are pre-filling the form data. Since we are editing a resource, we have to pre-fill the data on our form so we can edit it and send it back to the server. Hydrating a form can be infinitely complex depending on the data you are loading, editing, any relationships between the data, etc. For this example, we will just run through the process I do to hydrate the form to keep this functionality maintainable.

The first step I take is loading the individual resource from the API. In the sense of companies, our API call would look like:

```
async loadCompany() {
    this.company = await this.$api.companies.show( this.
$route.params.company );
    this.hydrateForm();
},
```

You can also use `asyncData()` to have the company pre-loaded. However, where I'd like to focus is after the successful loading, we call a method `this.hydrateForm()`.

This method simply copies over what our resource is to the local data within the edit page/component. By abstracting this to its own separate method we can set defaults, copy data, load related entities, etc. It's also a nicely scoped clean method where we can focus once if we add more fields to our form.

A piece of the `hydrateForm()` method that we've implemented for our company resource looks like this:

```
hydrateForm(){
    this.form.name = this.company.name;
    this.form.roaster = this.company.roaster;
    this.form.website = this.company.website;
    //... Other form fields
}
```

Now, in our edit form, we have all of the data that the resource (company) has set in the database. When we are done editing the fields, we can submit the form.

Submitting the Form

Similar to creating a resource, we need to send all of the data over to our API through the body of the request. However, this time we are using the PUT HTTP Verb. Why PUT and not PATCH? Well in our case, we will be providing the entire resource when we send it over for updating. Remember, on our company resource we created an `update()` method? That's what we will be using.

In the page or component, add the following code to the `methods` object:

```
async updateCompany(){
    if( this.validateCompanyEdits() ){
        this.processing = true;

        await this.$api.companies.update( this.company.id,
this.form )
            .then( function( response ){
                //... Handle successful response.
}.bind(this));

    }
},
```

Essentially what this does is submit a `PUT` request to the `/api/v1/companies/{id}` endpoint along with all of the data as the (`this.form`). Our API will then accept the incoming response and grab the variables from the request to update the resource.

When we touch on uploading files there are a couple things to note. First, we will have to send the data as a `FormData()` object. This is for the proper encoding to handle a file upload. Second, we will actually be sending this as a `POST` request with a `_method` of `PUT` so Laravel can react accordingly. I'll cover this more when we talk about Uploading Files, but just giving you a heads up that if you looked at the source code, it might look a little bit different.

Step 10: Implementing DELETE

This is a unique command in that it usually doesn't require a page. It's also very powerful as it deletes a resource on your API. When we set up this on the API, we blocked off this endpoint with middleware so only authenticated users can access this endpoint. Your API should definitely be the gate keeper for users who can or can not access endpoints where your SPA should be a visual representation of how to use the app.

With that being said, implementing this method is as follows:

```
async deleteCompany(){
    await this.$api.companies.delete( this.company.id )
        .then( function( response ){
            //... Handle successful response.
        }.bind(this));
},
```

One thing to notice about this request, is that there is no body. The URL contains the ID of the company we are deleting. Make sure not to include any sensitive data in the URL or it could risk being exposed!

I'll say it again, but what I like about HTTP verbs is the fact they are explicit and

when used correctly help the developer understand what is going on in the application. The **DELETE** verb is as explicit as it can get.

Conclusion

So now that we've set up a very basic API and a single resource, it's time to get into a few fancier features. One thing you might have noticed is how everything is based off of an ID variable. While this is explicit, it's not very readable, this also translates directly to our front end URLs.

In the next section, we are going to create human readable URLs called slugs. If you have come from a Wordpress background or have worked with Wordpress in the past, you are probably familiar with this term. Essentially slugs are string unique identifiers that help to identify a resource. They also work great for SEO if your app requires it and are human readable!

Accessing Resources With Human-Readable API Endpoints

What are Slugs?

Before we get started, let's go through what a slug actually is. Yes, it's a kind of slimy garden pest, but that's not what we are talking about. A slug is a human readable representation of a resource. Essentially a string ID. When working with coffee companies, you might have Ruby Coffee Roasters (delicious coffee by the way!) with an ID of 1. As a human, I see 1 and I don't think much of it. The URL would resolve to `/api/v1/companies/1`.

However, when it comes to SEO and making beautiful URLs, we might need something a little more "human readable". What if we could have, `/api/v1/companies/ruby-coffee-roasters`? The trailing `/ruby-coffee-roasters` is the slug. That'd be really nice, then our front end we could also use `/companies/ruby-coffee-roasters` which would be super easy to pass to our API and get the company.

Well, that's what we will be implementing! There is a beautiful Laravel package that helps with this and it really begins to make our API much more professional. I like to implement this early on before we get too much data so we don't have many migrations and we can implement it on other resources. Let's rock and roll!

Creating Slugs on Your API

If you've noticed from the last section where we've managed resources, we have all of our public routes with the ID of the company in the URL. This is not necessarily ideal for a few reasons. First, it's not user friendly. What does ID 132 mean to anyone?

Second, before we implement server side rendering, we should have a way to structure clean URLs. This is extremely important considering we want search engines to pick up on this and index correctly.

Finally, exposing the incrementing ID of your resource can be used by malicious users to test and see what's available. If you have your app locked down, this shouldn't be a problem, but it's good to keep in mind if the URL is relatively guessable by an incrementing ID.

So how do we get around this? We can use what is called URL slugs. These are unique string identifiers that serve as a URL friendly key to your resource. If you are familiar with Wordpress, this is what Wordpress uses to generate Post and Page URLs.

Luckily, there is a beautiful package we can include into our Laravel API that handles all of the functionality that we need. The package binds to an eloquent model (in this case our company) and uses the fields we want to generate a unique URL slug. This package is by Colin Viebrock and is: <https://github.com/cviebrock/eloquent-sluggable>.

What we will do is implement this package and update our API resource to generate slugs on creation. These will be 100% unique identifiers of our resource that is not the resource ID. We will then jump to the SPA side and make sure our routes are bound off of the slug instead of the id.

A quick note about SEO. If you already have a production SPA in place, you might want to set up your **301** redirects correctly. This will ensure that any keywords you

rank for will remain in-tact switching from an ID based system to a slug based system.

Step 1: Install and Configure Eloquent Sluggable Package

Luckily, the package is well documented and easy to install. The first step we need to take is to require the package through Composer like this:

```
composer require cviebrock/eloquent-sluggable
```

The next step is optional as it creates a config file for sluggable. I like to have it available just in case we want to make changes in the future. For now we won't do much with it since our config is available on our models, but it's nice to have for future instances. To publish the config run:

```
php artisan vendor:publish --  
provider="Cviebrock\EloquentSluggableServiceProvider"
```

The default is perfectly fine for what we want. Next, we have to configure our resources to use the package. In our case, this will be the company resource.

Set Up Resources

To do this, open the model of the resource you want to make sluggable. In this case, we will open up the `/app/Models/Company.php` file. On the model, we will add the following code to our `use` declarations:

```
use Cviebrock\EloquentSluggable\Sluggable;
```

Next, we will add some configuration for what fields should be used to generate our slug. This is done through an overridden public function on the model class. Each resource you need sluggable implemented on will require different fields. For our example, we want the name to be sluggable. So if we add Ruby Coffee Roasters, we want the slug to be `ruby-coffee-roasters`. A little foreshadowing for future features, when we add cafes, we will want the individual cafe name to be sluggable or the address to be depending on what's provided.

For your own resources, think about what you need to use to generate the slug. You can pass an array to the `source` configuration for multiple fields. Anyway, add the following public method to your resource, for companies it's like this:

```
/**  
 * Return the sluggable configuration array for this model.  
 *  
 * @return array  
 */  
public function sluggable()  
{  
    return [  
        'slug' => [  
            'source' => 'name'  
        ]  
    ];  
}
```

When the model is saved and a new model is persisted to the database, the `sluggable` method is called. What we return for configuration is the key `slug`. This is the name of the database field we will store the created slug (which is added next).

The `source` key is set to `name`. This is the database field that we will convert to the slug. There are a lot of other configurations you can do and you can even pass an array to the source that would use multiple database columns if needed to create a complex slug.

For now, we are set up! However, we will need to create a quick migration to add the `slug` field so the library knows where to store the computed slug. This was the first key in the return array that we mentioned.

Step 2: Add slug field to resource

So whatever resource you are adding your slug to, you have to extend the database table to account for it. In this example, we are using the `Company` resource which has a corresponding `companies` database table.

To do this, we need to create a migration that adds the following column:

```
Schema::table('companies', function (Blueprint $table) {
    $table->string('slug')->after('name');
});
```

In our example we added this field after the name field but you will just need a string field. All this needs to be is a string with the name `slug` or whatever you named your column. After migration, whenever we add a new company, this field will get auto populated with the computed slug.

OPTIONAL: Migrate Existing Resources

If you are integrating into an existing application, you might have to run a migration on existing resources. When migrating data like this, I like to include it in a database migration like you would to add a column or table.

When I first began working with Laravel and migrating data, I did it either through manual scripts or temporary routes. I figured that since we have database migrations with Laravel already, I'd run my scripts in here and it seems to work out great!

So if you have a whole bunch of resources that need slugs generated, I'd make a

simple migration that looks something like this:

```
/**  
 * Run the migrations.  
 *  
 * @return void  
 */  
public function up()  
{  
    $companies = Company::all();  
  
    foreach( $companies as $company ){  
        $company->save();  
    }  
}
```

and make sure you add the model to your `use` statements on top:

```
use App\Models\Company;
```

When this migration runs, it will trigger the save command which will use the already entered name to generate the slug. Just like that, all of your resources have slugs!

Step 3: Adjust Implicit Model Binding On Routes

So the last thing we will need to adjust is the implicit model binding on routes. Now remember, this is how we can automatically pass the object type hinted to a resource controller. Since we will be passing the slug to the API to load the resource and not the ID, we have to make sure this is what gets type hinted correctly. Luckily, Laravel makes this a breeze!

To ensure your resource is set up correctly, open up the model for that resource. In our case, it's `App\Models\Company.php` and add the following method override:

```
/**  
 * Get the route key for the model.  
 *  
 * @return string  
 */  
public function getRouteKeyName()  
{  
    return 'slug';  
}
```

So instead of using the `id` we now use the `slug` field of the resource to load from the route binding. And believe it or not, from the API side, we've now implemented sluggable resources! We now can pass make requests to the API like `GET /api/v1/companies/ruby-coffee-roasters` and it will return the company! This will be so much easier to view through the URL and make the UX of our app feel more polished.

Now we just need to apply it to the URLs we have in the SPA and we are have it fully implemented!

Implementing Slugs in Your SPA

So now that we have slugs being generated for our resource from the API, we need to use these slugs in the front end URLs. This will be a pretty quick update, especially if you are just beginning your application. We pretty much need to change all references from `id` to `slug`.

Step 1: Rename any `_id` directory or `_id.vue` for your resource

Since NuxtJS auto generates pages based on the contents of directories and naming schemes for pages, we need to make sure these are configured correctly. First, we need to see change any reference from `_id` to `_slug` for directories relating to the resource we just updated. Next, if we have an individual page for the resource, make sure it's updated from `_id.vue` to `_slug.vue`.

If you are following along with our app, you can see we changed the `/pages/companies/_id` directory to be `/pages/companies/_slug` directory. Since we are using an `index.vue` file in that directory, we don't have any `_id.vue` to update.

Step 2: Update references from `company.id` to `company.slug`

Since using companies is our resource example, we have a couple places where we needed to update from calling the API with `company.id` to `company.slug`. The first place we updated is in the `<nuxt-link>` of any link to an individual company.

We need to change the `:to` to be `/companies/' + company.slug`. This way, when we load all of the companies and have a link to an individual company page, we use the slug instead of the ID.

We also need to update any call to the API to send the `slug` instead of the `id`. Since we have to remember that since we changed the directory to be `_slug`, the route parameter is now named `slug` as well. So when we are in the `/companies/_slug/index.vue` page which will load an individual company, the request we make to the API through `asyncData` needs to be:

```
async asyncData(ctx) {
  return {
    company: await ctx.app.$api.companies.show( ctx.params.slug )
  }
},
```

Notice the `ctx.params.slug`? The name of the route parameter is now `slug`. So pretty much anywhere there was a reference to `id` we now reference the `slug` which brings us to our final update, the abstracted API calls.

Step 3: Update API calls and replace `id`

So 3 out of the 5 resource API calls require an ID. These are the `show()`, `update()`, and `delete()` requests since they operate on a specific resource.

Now technically you don't HAVE to update this since the value passed to the `id` parameter will be a slug, it's just good coding practice to have your variable names actually match what is being passed.

Now that we have the `id` refactored to be a slug, we can visit a URL and pass the `slug` of the resource. So if we have Burlington Coffee Roasters with a slug of `burlington-coffee-roasters` we can visit `/companies/burlington-coffee-roasters` and it will load the appropriate company! This will be much better for SEO when we get to that point and much cleaner and readable URLs!

Uploading Files

Working with file uploads is one of those features that can definitely get complicated fast. Especially when working with an API and SPA. However, once we run through the tips and tricks, uploading files from an SPA to an API can be as easy as submitting a form.

Handling File Uploads on the API

To be honest, this is one of the trickier aspects of API driven application development, but also one of the most important. File uploading and management is always tricky, but doing it from an API perspective adds a little extra to the mix. This is because we have to submit the file through javascript to the API instead of submitting a form in a standard call and response application.

For our example, we are starting with some basic image file uploads. We are going to add a logo and a header image to our Company resource. Uploading these images will provide a good example of how to upload files from an SPA to an API. Like usual, we will start with the API side first.

Step 1: Plan storage directory

If you remember back to when we configured our Laravel install (Configuring Laravel as an API) we linked the public facing directory to our storage directory. This `/storage` directory is where we will store all of our files that we upload. However, it's good to have a plan in place for what this directory should look like and code the file uploads according to plan.

First, for now at least, we will only be uploading files that are going to be accessible to the public. Since I like clean code and buckets to place files in, let's start by making a `/storage/app/public/companies` directory. This directory will be the root of where we will store our files for our company resource. I'd name this for whatever resource you have so you have a starting point. All directories from here on out will be made through code.

Let's think about how this should look. We will need to have unique names for the folders in this directory and they will have to be scoped via company. This leaves us with either naming them after the `slug` or an `id`. I'd say we use the slug. If we have to do any server maintenance or managing through the command line, it's going to be much easier to find what we are looking for. It will also make our URLs for the images more accurate. So let's plan on having our companies URLs look

like:

```
/storage/app/public/companies/{slug}
```

Also, since the slug is unique and URL compatible, we can ensure they will make good directory names.

Perfect, so we have our storage directory planned out and our bucket created. Now, let's think about where we will store the URLs.

Step 2: Plan database storage

For this basic tutorial, we are going to have 2 fields that the user can upload images for, a header and a logo for our company resource. I prefer to store these URLs for each of the images in the database along with the resource. This will allow for us to easily display them in the SPA when we load our resource.

Let's add a migration for these two images in our database:

```
/**  
 * Run the migrations.  
 *  
 * @return void  
 */  
public function up()  
{  
    Schema::table('companies', function (Blueprint $table)  
{  
        $table->text('header_image_url')->after('name')  
        >nullable();  
        $table->text('logo_url')  
        >after('header_image_url')->nullable();  
    });  
}  
};
```

```
}
```

So the only thing to note about this is the field is a `text` column. This allows for long URLs if we so need it. Other than that, we now have an empty field that we can store our URLs for each of these images in the database. We will be able to generate these URLs when we upload the images and bind them to an `` tag within our SPA dynamically.

Step 3: Implement File Uploads

So we have our storage directory planned out and our database fields added to store the uploaded image URLs. Now let's implement these on the API routes needed.

There will be two routes that need this functionality, the POST `store()` route which creates a new company and the PUT `update()` route that updates a company. Both of these handle a form request that contains updated company data which will also include these image files.

Remember how we created services to handle the requests for all of our routes? We will be creating another service right now to handle image uploads and extending our other services to use it correctly. Since we need to manage handle uploads on two routes, we will be re-using this code.

Let's first create the following service file at `/app/Services/Company/UploadCompanyImages.php` with the following code:

```
namespace App\Services\Company;

use Storage;

class UploadCompanyImages{
    private $company;

    public function __construct( $company )
```

```
{  
    $this->company = $company;  
}  
}
```

Right now this is just the shell of a service, but we will be adding more functionality soon. Our service class is named `UploadCompanyImages` but you should name it according to whatever resource you have.

Next, we will store a private variable which will be the resource that we are uploading images for. This way we can handle all of the functionality within this service class and save the uploaded URLs. We will set accept a resource modal (in this case a company) through our constructor and set it locally.

The other thing to note, is we are using the `Storage` facade that comes with Laravel. This facade provides all of the necessary features to upload the files passed in through the request.

Before we implement the actual core of this service, let's think about how we can re-use some code. Both the header and the logo will be uploaded to the same directory, so we can re-use the upload code. The differences in each file is where the URL will be saved to.

Now that we've thought about that, Let's add the 3 methods to our service:

```
public function saveLogoImage( $logo )  
{  
    $url = $this->saveFile( $logo );  
  
    $this->company->logo_url = $logo;  
    $this->company->save();  
}  
  
public function saveHeaderImage( $header )  
{  
    $url = $this->saveFile( $header );
```

```
$this->company->header_image_url = $url;
$this->company->save();
}

private function saveFile( $file )
{
    $path = Storage::put( '/public/companies/'.$this-
>company->slug, $file );
    $fileURL = env('APP_URL').'/
storage/'.$str_replace( 'public/', '', $path );

    return $fileURL;
}
```

This is the guts of the whole process right here.

Let's take a quick little tangent. Laravel makes handling form requests and file uploads a lot easier no matter if you are doing an API Driven Application or call and response application. No matter if you send your form through JSON from Javascript or submit it as `multipart/url-encoded` you can access the fields the same way.

If you are coming from a straight PHP background, you will be familiar with the global `$_FILES` array. Laravel allows you to access these files from your `Illuminate\Http\Request` and even extracts helpful data for you and adds a variety of methods used to interact with the file. The file information is abstracted to the `Illuminate\Http\UploadedFile` class which can be passed directly to `Storage` functions to make file [uploading a breeze](#).

Let's break down the `saveFile()` method first since that's the core. First, this method accepts a file, doesn't matter which one since it's going to the same spot.

The first line of that method, we call the `put` method on the `Storage` facade. The

first parameter is the server URL where we want to store our file. Know what's cool about the `put` method? If the directory doesn't exist, it will automatically create it! So no more checking hierarchies throughout your app. It also allows us to concat our strings so we can pass in the `slug` for the company and build that directory if needed.

The second parameter is the file that we are uploading. This is the `Illuminate\Http\UploadedFile` object. We can pass this directly to the `put` method and it will upload that file and return the `path` it was uploaded to.

Now the next line is just as dense. Once we have the uploaded file path, we need to convert it to a URL so we can set it as an image source. The first step that we need to do is grab the `APP_URL` from the environment variable. This should be the root URL of your application.

We then append the `/storage/` directory where everything is stored. The final part is since we mapped this directory to the `/public` directory, we don't actually need the `/public` in the URL. So what we need to do is replace `public/` with an empty string on the path variable which holds the storage path. Now we have our file URL and we return this. This will be the same for both images we are trying to upload.

So the other two methods, `saveLogoImage()` and `saveHeaderImage()` are very similar. Each method accepts an instance of `Illuminate\Http\UploadedFile` from our request, then runs it through our `saveFile()` method. The only difference is where it saves the URL upon completion. The `saveLogoImage()` saves the URL to the `logo_url` field in the database for our resource and the `saveHeaderImage()` saves the URL to the `header_image_url` field in the database for our resource. Then we call `save()` to persist the changes to the database.

This is all the code we need to upload files for our resource through our API! Laravel really comes in clutch with the help on this. Now we just have to link this together in our services and ensure the requests are valid.

Step 4: Utilize our Upload File Service

So now that we have our service created to upload files, we need to implement that on each route that we will be uploading the files for. This will be the `POST /api/v1/companies` route and the `PUT /api/v1/companies/{slug}` route. Both of these routes have an associated service and a route validator.

Let's set up our route services first. I'll go through how we hook this together for the `POST` route service, but the `PUT` route service will be exactly the same. Since we've abstracted everything we need to add a company resource into a convenient service, we need to extend these services to handle file uploads. This is precisely why we abstracted the process in the first place. If all of this was in our controllers, we'd have MASSIVE un-maintainable controllers.

Let's look at our `app/Services/Company/CreateCompany.php` service. Let's first add our `UploadCompanyImages.php` service to the top with our `use` declarations:

```
use App\Services\Company\UploadCompanyImages;
```

Next, we need to add the following method:

```
private function saveImages( $company, $data )
{
    $uploadCompanyImages = new
    UploadCompanyImages( $company );

    if( isset( $data['logo'] ) ){
        $uploadCompanyImages-
>saveLogoImage( $data['logo'] );
    }

    if( isset( $data['header'] ) ){

```

```
$uploadCompanyImages-
>saveHeaderImage( $data['header'] );
}
}
```

What this does is simply encapsulate all of the code for checking if the file is present on the request and processing the upload into one method. In this method, we create a new instance of our `UploadCompanyImages` class. We then check to see if the logo is present in the request, if it is, we save the logo image. Next we check to see if the header is present in the request. If that's present, we save the header image.

The method accepts our resource and the request data as the two parameters. Since we have access to the newly created resource (which we will run through next), we pass that to our `UploadCompanyImages` service and any file uploads will be saved accordingly.

Finally, one important thing to note, I added a call to this method AFTER our resource was created by adding the following line:

```
$this->saveImages( $company, $this->data );
```

This ensures that we have a created resource before we try to attach images to it. We now have everything tied together! If you want to see how we did the `PUT` resource, check out Gitlab. The last step of the API side is to validate that the user added the files upon request and that they are valid.

Step 5: Validate File Uploads

So we will want to ensure that the files uploaded are valid before we do any processing. Now there will be a difference on logic between the create and update routes for our resource. The difference is we want to require the images be present when we create a company resource, but not when we update one. If the user doesn't need to update the logo or header image, we don't want to force them to.

The similarities are that we want to ensure the file was updated without errors and for each key, the `logo` and `header`, we want to make sure that the files uploaded are images.

So the validations for our `create` route for BOTH `logo` and `header` fields will be:

- required
- file
- image

The first validation ensures that the field is present, the second ensures that the field is a successfully uploaded file and the third validation is that the file is an image.

The validations for our `update` route for BOTH `logo` and `header` fields will be the same except instead of the `required` validation it will be the `sometimes` validation. This just means that it should validate the fields with the validations if the field is present.

Let's take a look at our

`App\Http\Requests\API\Company\CreateCompanyRequest.php` file and jump to the `public function rules()`. In the returned array, add the following keys:

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        ...
        'logo' => 'required|file|image',
        'header' => 'required|file|image'
```

```
];  
}
```

Now whenever we submit a request to create a resource, we ensure that both fields are present, the fields are files and the files are images. Pretty sweet that Laravel provides this functionality right out of the box for you!

You can also extend your error messages with these validations as well to alert the user if they send bad data. For the update request, simply change the validation rules to be `sometimes|file|image` and it will only validate if the field is present.

Now we can accept a header and logo image for our resource! Feel free to check out the Gitlab to see the architecture behind this implementation. This is one thing I hope helps with the book is seeing a larger app architecture while focusing on fine grained details.

Implementing this on the SPA side is where things get slightly different. Let's hop over there and see how it's done!

Uploading files Through an SPA

So we have our storage system in place for the uploaded files & endpoints to handle these uploads. Now comes the process of sending them to our API. This is where it differs slightly from a standard API request and we have to do some modification of the request before we can send along files. The two forms we will need to adjust are the create and update form for our resources. Let's get started.

Step 1: Add File Inputs to Create Form

For our example, we will be walking through the form we use to create a company resource. This can be replicated for any resource you have and will be the same for our update form as well. The first thing we need to do is open the form and add the following inputs to the template. This is our `/pages/companies/new.vue` page:

```
...
<label for="logo">
  Logo
</label>
<input ref="logoFile" v-on:change="handleLogoUpload()" accept="image/*" id="logo" type="file">

<label for="header">
  Header
</label>
<input ref="headerFile" v-on:change="handleHeaderUpload()" accept="image/*" id="header" type="file">
...
```

These are the two fields we are adding to our template to allow the user to select and upload files. Your HTML and CSS may look a little different but let's inspect one of the file inputs.

```
<input ref="logoFile" v-on:change="handleLogoUpload()"  
id="logo" type="file">
```

Both file inputs are very similar so the same description and functionality are going to be the same. The first attribute we need to inspect is the `ref` attribute. This is a unique identifier that we can access the element on through VueJS. This is considered an edge case in [Vue](#), but we will have to use it.

We can't apply the `v-model` to a file element since files are handled securely through the browser. We need to access the [FileList](#) and we can gain access to this through the `ref` attribute. When we work with this element within our methods, we will access the [FileList](#) by calling `this.$refs.logoFile.files`.

The next attribute we have is the `v-on:change`. This is important. What this will do is when the user selects a file from the system, the value of the input will change and we will trigger the appropriate method. For the logo file upload this is `handleLogoUpload()` for the header file upload this is the `handleHeaderUpload()` method. In the next step we will be implementing these methods.

The final attribute we have is just to limit the available file types to upload. We can limit these to just images by adding `accept="image/*"` to our file input. We should never take client side validation as truth, but to limit the upload types is good UX. All other attributes are just normal file attributes and styling. Let's implement our handler methods.

Step 2: Implement On Change Handlers

This part is broken into a couple sub steps. We first need to add data to handle the file we are uploading similar to what we have for a normal data attribute. In this case add the following to your returned data:

```
data(){
  return {
    form: {
      ...
      logo: '',
      header: '',
      ...
    }
  }
}
```

Remember, these are the names of the files that are being sent to the API so we have to make sure they match what will expected in the request. Right now, we just need them present in our form object that we will be sending to our API. We can't use `v-model`, but we will set these through our change handlers.

Implement change handlers methods

Now that we have a place to store our files that we wish to send to our API, we have to implement the methods that listen to a change on the file and set the data locally. Since the elements are `input[type="file"]` elements, when a file gets set it will be a type of `File`. We can grab this from the file list associated with the element. Let's implement our first "on change" handler for a header image upload:

```
handleHeaderUpload(){
  this.form.header = this.$refs.headerFile.files;
}
```

The method is actually pretty simple. When the file input changes, we grab the `FileList` from the input using the local `$refs` and then save that `FileList` to the local `header` variable to send to the API. For the logo, it's going to be exactly the same functionality.

Step 3: Preview Uploaded Image

This is technically optional, but I like to throw this in there since it adds good UX, and that's a preview of the uploaded image. The reason I say we look at this now is that if we are implementing our file uploads, this will be in the same code base no matter what resource you are implementing.

To preview an image upload, you have to do 3 steps. The first step is to add two pieces of data to the local data object:

```
data(){
    return {
        headerPreview: '',
        showHeaderPreview: false
        // ... Other data
    }
}
```

The `headerPreview` will be a base64 encoded preview of the image file that we can set to an image `src` attribute. The `showHeaderPreview` determines if we should show the preview image or not.

The next step is implementing these placeholders in your UI. To do that you can add the following anywhere that you want to show a specific preview. Right now, we are doing this for the header image for our company. If you want to do this for a logo or a profile image, I'd recommend updating the variable names. The HTML should look like:

```

```

All this is is an image tag that shows only when we choose to show the header preview and the source is set to the result of the next step, reading in the uploaded file.

The final step is to update our handleCompanyHeaderUpload() method to look like:

```
handleCompanyHeaderUpload(){
    this.form.header = this.$refs.headerFile.files;

    let reader = new FileReader();

    reader.addEventListener("load", function(){
        this.showHeaderPreview = true;
        this.headerPreview = reader.result;
    }.bind(this), false);

    if( this.form.header ){
        if ( /\.(jpe?g|png|gif)$/
i.test( this.form.header[0].name ) ) {
            reader.readAsDataURL( this.form.header[0] );
        }
    }
},
```

There's a little more to this method now, but let's break it down. We went through the setting of the local variable in the last step. Now, what we do is initialize a `FileReader()` object. This object allows us to read an uploaded file. We then bind to the “`Load`” event. This will trigger when the upload file has been loaded and read. We need to wait for this event to fire otherwise we won't have all of the data that we need.

However, this is the interesting part, is we need to trigger one more event before the load event will fire. The load event is just a call back. What we need to do is ensure we have an uploaded file and that the file is an image file type that we can preview:

```
if( this.form.header ){
    if ( /\.(jpe?g|png|gif)$/
```

```
i.test( this.form.header[0].name ) ) {  
    reader.readAsDataURL( this.form.header[0] );  
}  
}
```

If all of that is true, then we run the `readAsDataURL()` method and pass the file that was uploaded. This will read in the file that was uploaded and translate it to a base64 encoded data url that we can use to display in our image source. This method is what triggers the load event. When the uploaded file has been loaded and read, then we jump back to our callback and set the flag that we want to show the header preview (`showHeaderPreview`) to `true` and set the `headerPreview` variable to the result of the reader (`reader.result`). This will display a preview of our uploaded image!

Step 4: Modifying Requests to Handle Files

So unlike what we've worked with for most of our API requests, where we send straight JSON and call it good, we are sending a file to our API. This requires us to switch up how we send the requests by making two changes:

1. Ensuring we send the data as a `FormData` object
2. Applying the `Content-Type: multipart/form-data` header to the request.

Remember, we will be sending files with the `create` and `update` requests for our resource, so we will have to modify both of those requests. Let's first talk about transforming our data to be `FormData` instead of JSON.

Modifying to `FormData`

First of all, what is `FormData`? According to MDN:

The FormData interface provides a way to easily construct a set of key / value pairs representing form fields and their values, which can then be easily sent using the XMLHttpRequest.send() method. It uses the same format a form would use if the encoding type were set to "multipart / form-data"

<https://developer.mozilla.org/en-US/docs/Web/API/FormData>

This is super cool. What this means is we can use the Javascript representation of a form to create a request that feels like it would if we created a call and response application. This interface is also extremely easy to use which will make it nice to send through our API. Let's take a look at the method that handles the submission of a new resource to the API. In our case, this is the `async addNewCompany()` method in `/pages/companies/new.vue`.

Right now, this method sends `this.form` which is the entire form object to our API request which sends it to our backend API. However, we just added some files in there. In order for our server to handle file uploads we need to make sure that they are sent in a `FormData` object with the proper headers. Let's modify our function to look like this:

```
async addNewCompany(){
  if( this.validateNewCompany() ){
    let formData = this.transformToFormData();

    const newCompany = await this.
$api.companies.store( formData );

    this.$router.push({
      path: '/companies/' + newCompany.slug
    });
  },
}
```

So the main modification we changed was creating a `formData` variable and

assigning it the result of the local method `transformToFormData()`. Don't worry, we haven't showed it yet, but that's next! Then we pass the `formData` variable to our API request instead of the `this.form` object and we will have our files sent along as well.

Now we need to implement our `transformToFormData()` method. The reason we made this new method is to encapsulate the code and allow for `x` amount of form fields in the future. We also need to account for images (this would also be a good candidate for a mixin since we will use it in a few places). Let's add this method to our page component:

```
transformToFormData(){
    let fileKeys = ['logo', 'header'];

    let formData = new FormData();

    for (let [key, value] of Object.entries(this.form)) {
        if( fileKeys.indexOf( key ) > -1 ){
            formData.append( key, value[0] );
        }else{
            formData.append( key, value );
        }
    }

    return formData;
},
```

To break it down, let's look at the `fileKeys` first. This variable is a simple array that keeps track of all of the keys in our `$this.form` data that will be files. We need to know this later on because we will be accessing the values differently.

Next, we create an instance of the `FormData` interface. The `FormData` interface allows us to append values to it using a `key/value` type structure.

Now, we iterate over the entire `this.form` object, assigning each key to a

temporary `[key, value]` array. Inside of the loop is where we check to see if the key is a file key. If it is a file key, we grab the value at index `0`.

The reason we check for this is remember, the `input[type="file"]` we grab the `FileList` which is an array. We are only uploading one file per input and we need to grab the individual `File`. When we have that data, we append it to the `FormData` variable with the `key` being the `key` of the form element and the value of the `File`. With all of the other data we simply append it to the `FormData` variable with the `key` being the JSON key and the value being the value of the input. This way we maintain the same naming scheme expected by our API.

For example, `this.form.name` will still append to be `name`.

Remember, Laravel doesn't matter the encoding of the request when it's abstracted into a request object upon submission. However, it DOES matter that when sending files they are `multipart/form-data` encoded. We will get to that soon. Transforming the request into `FormData` is the first step.

Finally, we return the filled `FormData` that we send to the API.

Adding Appropriate Headers

At this point we have converted our form object into `FormData` and we have submitted it to our API request. If we try to upload the file now, it won't work quite right yet. This is because we need to add the appropriate headers to our request. Since we've abstracted our resources's API requests into it's own module, we need to open that module and add one small change.

Let's look at our `/api/companies.js` and find the `store` method (the update method will follow suite exactly so make sure to update that as well. Currently, our `store` method looks like:

```
async store( company ){
  try{
    return await $axios.$post('/api/v1/companies',
```

```
    company );
} catch ( err ){
}
},
```

What we need to do is add a header that informs the server that the incoming content is `multipart/form-encoded`. Luckily axios allows you to add some extra headers as part of the 3rd options parameter to your request. Let's add the header like this:

```
async store( company ){
try{
    return await $axios.$post('/api/v1/companies',
company, {
    headers: {
        'Content-Type': 'multipart/form-data'
    },
} );
} catch ( err ){
}
},
},
```

Now when we send our request, the `company` parameter will be our `FormData` instance and when we pass the data to the server we will have the proper headers to let the server know it's not JSON and it can extract files from the request.

Now to tie it all together, submit a new instance of one of your resources. For our case, let's submit a logo and header file for "Succulent Coffee Roasters" when adding the company. Upon completion, you should see not only the new resource in your database, but also two files under the `/storage/app/public/companies/succulent-coffee-roasters` directory. These will have unique file names and this is for security purposes so malicious users can't try to scrape URLs from our storage directory. But that's it! We now have completed a full loop

of file uploads through an API!

Sending Files Through PUT

However, there is one additional gotcha that we have to account for when submitting the `update()` method. This method is submitted through PUT right now. However, when doing a `multipart/form-data` encoded submission with a `FormData()` object, you have to actually submit the form through POST with a `_method` field that's set to PUT. This is because we are submitting a form similar to an HTML form and HTML can not submit as [PUT directly](#).

To make these changes, check out the `update()` in `/api/companies.js` and update the method to be `$post` for your request:

```
async update( slug, company ){  
    try{  
        return await $axios.$post('/api/v1/  
companies/'+slug, company, {  
            headers: {  
                'Content-Type': 'multipart/form-data'  
            },  
        } );  
    } catch ( err ){  
  
    }  
},
```

The final step is to open up your update form and when you transform your form to a `FormData()` object, you need to add:

```
formData.append('_method', 'PUT');
```

Even though our API accepts a PUT request and we are sending the request through POST the `_method` set to PUT will allow this request to be handled correctly via our API

Now that we have these images, I'll run through how to use them within your SPA!

Step 5: Using Uploaded Files

This is pretty straight forward to do. Since our resource, when loaded, will have database fields that contain the URLs to our header and logo file, we can use these to display the image within our SPA.

For our use case, we will want to display the logo of the company on the company card which shows up on the [/companies](#) page and we want to display both the header image and logo on the individual company page.

Let's run through displaying the logo on the company card found on the company page. When displaying a logo or image, we need to bind the `src` attribute of the `` tag since the image will be dynamic (dependent upon loading) and the URL of the logo:

```

```

The reason we do this is if we are dealing a delayed load of a resource or need to add a default image of the url field is empty. Let's say we just added the file uploads to our application. We don't want a whole bunch of blank logo spaces in our UI. It'd be a good idea to provide a default image. To do that, dynamically bind your `` src like this:

```

```

To see an example of our implementation, check out the individual company page in the Gitlab repo at: [/pages/companies/_id/index.vue](#).

Conclusion

There we go! We've successfully uploaded and displayed images! Our app is

starting to come along, but we've got a lot of ground to cover and explain a lot more API specifics! Having file uploads is fun now and can really breathe life into your app by providing beautiful UIs or managing quality data.

The next two chapters will touch on data relationships. These relationships are fairly common in Laravel applications, but need to be approached slightly different from an API perspective.

Handling "Parent-Child" Relationships

So what are “parent-child” relationships? They are usually relationships between resources defined by a `hasMany()` and `belongsTo()` relationship between two Eloquent models. In ROAST this would be the relationship between a Company and a Cafe. A Cafe belongs to a company. A Company has many cafes as well. These relationships are fairly common, extremely important and have a few gotchas when it comes to implementing them into an API. Let’s explore this a little further.

Sub Resources Using Has-Many Relationship

Having an entire application with a single resource is definitely not likely. Most of the time you will have an application with ten if not hundreds or thousands of different resources and lots of sub-resources. In this chapter, we are going to focus on "sub-resources" and some of the gotchas along the way.

Step 0: What is a Sub-Resource?

Sub-resources are also referred to as "children" in a "parent-child" relationship. They cannot exist without the presence of primary resource or they are better suited to have one. Either way, I like to refer to them as sub-resources because I think of drilling down into an app like a funnel. First you have primary resources on the top, then you have lots of sub-resources dependent upon that primary resource.

For our example application, we are going to have "Cafes" be a sub-resource of "Companies". Another way of thinking (and a foreshadow of our Laravel API model relationship) a "company can have many cafes". A cafe will also "belong to a company". For these relationships, we will be using the `hasMany()` and the `belongsTo()` relationships for our models. This will allow our parent resource "Company" to have many "Cafes" and our sub-resource "Cafe" to belong to a "Company".

Now most of the time, I mean like 95% of the time, these resources are managed the exact same way as a primary resource (so don't worry, this chapter won't be as intense as Efficiently Building API Endpoints for Data Queries), but there are a few things to cover so we know how to work with sub-resources.

I'll walk through how to structure your sub-resource and a few of the gotchas along the way. If you are interested in viewing all of the code, check out the source code on [Gitlab](#) and let me know what you need some clarity on!

Step 1: Create Sub-Resource and Set Up Relationships

The first step we need to complete is to build our database migrations and add our models. For our example of a sub resource, we are using `Cafes` so I added a database table named `cafes`. In the database migration, I made sure to add the following field:

```
...
bigInteger('company_id')->unsigned();
foreign('company_id')->references('id')-
>on('companies');
...
```

What this field allows is an internal database relationship to a company. This ensures our data structure matches the way we plan on setting up our relationships and requires us to add a cafe to a company.

Now that we have our database set up, we can add the `Cafe.php` model that is used to interface with the database. Start with an empty model:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Cafe extends Model
{
    use SoftDeletes;

    protected $table = 'cafes';
}
```

Right now this is just an empty model that can be used to create cafes. However, we want to make sure the relationship between cafes and companies is set up correctly. From the "sub-resource" point of view, the perspective is the resource

"belongs to" a parent resource, in our case a "Cafe" belongs to a "Company".

To set this up, add the following relationship to your "sub-resource" model, in our case, the `Cafe`:

```
/**  
 * A cafe belongs to one company  
 */  
public function company()  
{  
    return $this->belongsTo('App\Models\Company',  
        'company_id', 'id');  
}
```

The way this is structured is it adds the relationship through `company()`. When we call `$cafe→company` we will receive the parent company model on the cafe. The first parameter of the `belongsTo` method is the model we want to relate it to.

If you follow Laravel naming standards and have your ids and database tables set up correctly, this would be all you need. However, I don't like to guess and rely on "magic", so I fill in the next 2 parameters. The second parameter is the database column name of the child resource that keeps track of the parent resource it belongs to. In this case it's `company_id` which is the ID of the parent resource that the child resource belongs to. The final parameter in the `belongsTo` method is the database column name of the parent resource. In this case, it's `id`.

Our sub resource is set up correctly now, but we still need to ensure that the relationship is set up from the parent resource's perspective. In this case the parent resource, company, has many cafes. To set this up, we need to open up our parent resource `App\Models\Company.php` and add the following relationship:

```
/**  
 * A company has many cafes  
 */  
public function cafes()
```

```
{  
    return $this->hasMany('App\Models\Cafe', 'company_id',  
    'id');  
}
```

So this is essentially the inverse of the previous relationship except a parent level resource can have many child resources. Similar to the `belongsTo()` relationship, the first parameter is what is required. This is the child resource. If everything follows the naming standard, this would be all we need. The next parameters I always add just to ensure validity and not guess. The second parameter is `company_id`. This is the sub-resource's field that relates to the parent model. The third parameter is the column name of the parent resource that the sub resource is referencing.

Essentially the column on the parent resource that uniquely identifies it, the primary key. Now that we have this relationship set up, we can get all of the cafes on the company by calling the `→cafes()` relationship. We now have our sub-resource added and have it set up with our parent resource! It's now time to access these resources.

* For more information regarding `hasMany()` relationships check out the [Laravel Docs](#).

Step 2: Adding Sub-Resource Routes

This can be a little tricky because there a ton of ways to do this and a lot of variables being passed around. Especially with our implicit route bindings and form requests.

With a sub-resource we are going to end up with sub-routes. This means, for now, we are going to be accessing our sub resources through our parent resource. Let's add the following routes to our API:

```
/**  
 * Cafe Routes
```

```
/*
Route::get('/companies/{company}/cafes',
'API\CafesController@index');
Route::get('/companies/{company}/cafes/{cafe}',
'API\CafesController@show');
Route::post('/companies/{company}/cafes',
'API\CafesController@store');
Route::put('/companies/{company}/cafes/{cafe}',
'API\CafesController@update');
Route::delete('/companies/{company}/cafes/{cafe}',
'API\CafesController@destroy');
```

To go along with these routes, I added the `App\Http\Controllers\API\CafesController.php` with the following stubs:

```
namespace App\Http\Controllers\API;

use App\Http\Controllers\Controller;
use App\Models\Company;
use App\Models\Cafe;

use App\Http\Requests\API\Cafe\CreateCafeRequest;
use App\Http\Requests\API\Cafe\UpdateCafeRequest;

class CafesController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth:sanctum')->only(['store',
        'update', 'destroy']);
    }

    /**
     * Display the cafes for the company
     *
     * @param \App\Models\Company $company
     */
```

```
* @return \Illuminate\Http\Response
*/
public function index(Company $company)
{

}

/**
 * Saves a cafe
 *
 * @param \App\Models\Company $company
 * @param \App\Http\Requests\API\Cafe>CreateCafeRequest
$request
*
* @return \Illuminate\Http\Response
*/
public function store( CreateCafeRequest $request,
Company $company )
{

}

/**
 * Display the specified cafe
 *
 * @param \App\Models\Company $company
 * @param \App\Models\Cafe $cafe
*
* @return \Illuminate\Http\Response
*/
public function show( Company $company, Cafe $cafe )
{

}

/**
```

```
* Updates the specified cafe
*
* @param \App\Http\Requests\API\Cafe\UpdateCafeRequest
$request
    * @param \App\Models\Company $company
    * @param \App\Models\Cafe $cafe
    *
    * @return \Illuminate\Http\Response
*/
public function update( UpdateCafeRequest $request,
Company $company, Cafe $cafe )
{
}

/** 
 * Remove the specified cafe from storage.
*
* @param \App\Models\Cafe $cafe
* @return \Illuminate\Http\Response
*/
public function destroy( Company $company, Cafe $cafe )
{
}

}
```

A lot of it looks similar to our first resource we discussed, but let's touch on a few of the unique situations.

Multiple Implicit Bindings

So when we managed our first resource, we only had one resource to implicitly bind to a route. When we look at the `update()` method, we see 3 parameters:

1. The update request where we will store our validations

2. The company that's implicitly bound
3. The cafe that's implicitly bound

Similar to our parent resource, we need to tell Laravel what key we are using to look up the model through implicit route binding. To make this work, add the following to the `Cafe.php` model:

```
/**  
 * Get the route key for the model.  
 *  
 * @return string  
 */  
public function getRouteKeyName()  
{  
    return 'slug';  
}
```

FYI, we implemented sluggable on our sub-resource model as well (see below for a little bit of how this differentiates with different keys).

So now that we have this set up and we look at the `PUT /api/v1/companies/{company}/cafes/{cafe}` we have two implicitly bound resources that get injected into the controller. Both of them are off of our sluggable slug. However, when adding these nested resources and implicitly binding them to the routes there are a few things to note:

1. Order is of upmost importance. You won't be able to find a company based off of a cafe slug or vice-versa. If your company slug comes first, make sure your company resource is the first parameter and your cafe is second.
2. The request validator is ALWAYS THE FIRST parameter. If you look at any of the methods that require validation, the request validator is first before the bound models.

Placing The Controller

I've been all about buckets since the start. However if you notice with the `CafesController.php` to handle our sub resource you see I've placed it in the `app\Http\Controllers\API` namespace. Reason being is our app won't have many nested resources and we don't want to over-complicate the controller structure. If your app expands, I'd highly recommend making controller namespaces. Something like `app\Http\Controllers\API\primary-resource\`.

That's pretty much the big difference with adding sub-resources from the database up to the routes. Managing these resources is very similar. One of the last gotchas is with testing.

Step 3: Testing a Sub Resource

So the big gotcha with testing is the requirement to have a parent resource. In order to make this happen, we will be making heavy use of our factories. To test our sub resource I created the following file: `/tests/API/Cafe/CreateCafeTest.php`. I then added the following code:

```
namespace Tests\Feature\API\Cafe;

use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

class CreateCafeTest extends TestCase
{
    use RefreshDatabase;

    private $company;
    private $user;
```

```
protected function setUp(): void
{
    parent::setUp();

    $this->user = factory(\App\Models\User::class)-
>create();
    $this->company =
factory(\App\Models\Company::class)->create([
        'added_by' => $this->user->id
    ]);
}

public function testCreateCafeForCompany()
{
    $this->actingAs( $this->user )
        ->json( 'POST', '/api/v1/companies/'.$this-
>company->slug.'/cafes', [
            'location_name' => 'Stevens Point',
            'address' => '1410 3rd St.',
            'city' => 'Stevens Point',
            'state' => 'WI',
            'zip' => '54481',
            'tea' => 1,
            'matcha' => 0
        ] )->assertStatus( 201 );

    $this->assertDatabaseHas('cafes', [
        'company_id' => $this->company->id,
        'location_name' => 'Stevens Point'
    ]);
}
```

So the main thing to note is in the `setUp()` method. In that method, I create a

user but I also create a local company. This way we can test adding a cafe to the company by using its created slug to call the proper route. When creating the company through the factory, I overwrote the `added_by` field with the id of the `user` we are using for testing.

Since our route to add a cafe is locked behind middleware, we have to have the user as well. So we can use the `actingAs` method to use the user we created and create our cafe on the company. We store these variables as private on the testing class. This way we can easily write a lot of tests with less repeating code. As we continue to add features our testing will become more and more important and more and more complex. Maintaining these tests can be a challenge so to make it as modular as possible is very important.

That's pretty much the differences when dealing with a sub resource from an API perspective. Next, we will go through some of the gotchas on the NuxtJS SPA side.

Further Detail on "show" services

One thing I wanted to touch on is why do we have "show" services? When we added our `company` resource and when we implemented the `cafe` resource as a sub resource, we added the following services:

- `App\Services\Company>ShowCompany.php`
- `App\Services\Cafes>ShowCafe.php`

Looking at the code for the `ShowCompany.php` resource, it seems pretty redundant. We literally return the variable that was passed in. However, we have this in place for when we expand. These simple show methods can get out of hand fairly quickly. Especially as we start adding sub-resources and relationships, search algorithms, etc.

We've already started that with the `ShowCafe.php` service. If you take a look at that you will see:

```
public function show()
{
    $cafe = Cafe::where('id', '=', $this->cafe->id)
        ->with('company')
        ->first();

    return $cafe;
}
```

We return the company along with the cafe so we can reference that information as well. To foreshadow our "many-to-many" relationships, we will be pulling in "brew methods", "tags", and "alternative milk offerings". To keep our controllers clean, we will be abstracting those queries into our service. If we need to add filters we can add private methods in here as well. It's good to have this infrastructure in place now to lower the refactoring cost in the future.

Optional: Multiple Fields For Slug

One other thing we did with the cafe sub-resource is make a slug out of multiple fields. Reason being is you take a large city like Milwaukee, WI with a lot of cafes, we can't rely on the `location_name` being unique. For example, Collectivo coffee roasters and Valentine coffee roasters all have a Milwaukee cafe and they all have cafes in the same neighborhoods. It's very likely that there would be non-unique names. Also, since we are accessing them through a company, we can't combine the company name and location name or the URL would be funky.

To accomplish this, we made a slug from the `location_name` combined with the `address`, `city`, and `state` to make a completely unique URL. To do this was very straight-forward with Laravel Sluggable. What we did was add the following method to our `App\Models\Cafe.php`:

```
/**
 * Return the sluggable configuration array for this model.
 */
```

```
* @return array
*/
public function sluggable()
{
    return [
        'slug' => [
            'source' => ['location_name', 'address',
'city', 'state']
        ]
    ];
}
```

This allows us to define how are slug is created. When we applied this to our `Company.php` model, we only did a single field name and set the `source` to be that string. Since we are combining multiple field names, we can pass an array of field names to concatenate and make our slug. The order is VERY important as that's the order the library will concatenate the fields to generate our slug.

Implementing Sub Pages in Our SPA

Now that we have our sub-resource added in our API we have to find a way to display it. Considering we nested this resource within our primary resource, we will follow suite with the way we structure our front end.

The URL structure will follow `/{{primary-resource}}/{{primary-slug}}/{{sub-resource}}/{{sub-slug}}`. We will also display our sub resource on the `/{{primary-resource}}/{{primary-slug}}` page. So what this means, is we will have a listing of our cafes on the company page. From there, we will be able to navigate to an individual cafe page and view more information regarding the cafe's offerings. This will be very similar for any sub-resources you have in your application. Once again, the majority of managing and displaying these resources is very similar to what we went through, but we will go through some of the gotchas.

Step 1: Structure your sub resource directory

Considering NuxtJS structures your URLs based on the structure of your directory, this is extremely important. Our end game is to have urls structured similar to this: `/{{primary-resource}}/{{primary-slug}}/{{sub-resource}}/{{sub-slug}}` which will translate to `/companies/{{company-slug}}/cafes/{{cafe-slug}}` where applicable.

To do this we need our directory structure to look like this:

```
/companies  
/companies/_slug  
/companies/_slug/cafes  
/companies/_slug/cafes/_slug
```

Within those directories we will have the appropriate views to display an individual cafe and company. However, this presents an issue. Since we are setting a route variable of `_slug` which we had set up when we had a single resource, this will override the first `_slug` instance with the second `_slug` instance so we won't have access to the company slug when accessing an individual cafe due to the parameters being the same name. Definitely an issue!

To resolve this, we had to change the directory name from `/companies/_slug` to `companies/_company` and `/companies/_slug/cafes/_slug` to `/companies/_company/cafes/_cafe`. But this doesn't come without a slight refactor. Now in our `/companies/_company/index.vue` file, we load the company through `asyncData()` which is called through `ctx.params.slug`. Since we changed the directory structure we have to update any reference to the parameter to be `ctx.params.company`. The route parameter of `slug` has now become `company`.

Remember, with NuxtJS, we name our available routes and the name of the parameter is based off of the directory structure. So when we have `_company` the name of the associated parameter is `ctx.params.company`.

Now we have the following front-end routes available in our application:

```
/companies  
/companies/{company}  
/companies/{company}/cafes  
/companies/{company}/cafes/{cafe}
```

Step 2: Accessing and Loading Sub Resources

Accessing a sub resource requires us to build an API interface similar to what we did for our primary resource. For this, we added the `/api/cafes.js` file with the following code:

```
export default $axios => ({
```

```
async index( company ){
    try {
        return await $axios.$get('/api/v1/
companies/'+company+'/cafes');
    } catch ( err ){
        }
    },
}

async show( company, cafe ){
    try {
        return await $axios.$get('/api/v1/
companies/'+company+'/cafes/'+cafe);
    } catch ( err ){
        }
    },
}

async store( company, cafe ){
    try {
        return await $axios.$post('/api/v1/
companies/'+company+'/cafes', cafe, {
            headers: {
                'Content-Type': 'multipart/form-data'
            }
        });
    } catch ( err ){
        }
    },
}

async update( company, cafe, updates ){
    try{
        return await $axios.$post( '/api/v1/
companies/'+company+'/cafes/'+cafe, updates, {
            headers: {

```

```
        'Content-Type': 'multipart/form-data'
    }
}
} );
} catch ( err ){
}

},
}

async delete( company, cafe ){
try{
    return await $axios.$delete( '/api/v1/
companies/'+company+'/'+cafes+'/'+cafe, {
headers: {
    'Content-Type': 'multipart/form-data'
}
} );
} catch ( err ){
}

}
}
});
```

The only major differences are the amount of parameters that some of the methods accept. Any request that submits a file has to still be done through a `formData` object and the proper header of `'Content-Type': 'multipart/form-data'`.

When it comes time to load a sub resource, it is done the same way as a primary resource, through `asyncData()`. So in Roast, we will be loading the `cafes` for a `company` on the individual company page. For now, we are making a separate request to load the cafes (our sub-resource).

This is meant to show how to initially do the loading. As we grow our app and optimize, we can come up with a way to load the sub-resource with the primary resource while making our endpoints make sense and maintaining a clean API.

If we look at our `/pages/companies/_company/index.vue` file, let's make the `asyncData()` look like:

```
async asyncData(ctx) {
  return {
    company: await ctx.app.$api.companies.show( ctx.params.company ),
    cafes: await ctx.app.$api.cafes.index( ctx.params.company )
  }
},
```

Now we have access to two local variables within the individual company page. The first variable is `company`, which we already had, but now we also have `cafes`. This will contain all of the cafes for the company returned from the API. We can then use that data to display in the page.

Conclusion

Those are really the only gotchas on the frontend when dealing with a nested resource, making sure the URL params operate correctly, and loading multiple values through `asyncData()`. Our app is starting to come to life very fast which is awesome!

Next up, we have another relationship you might be familiar with. It's a many to many relationship. These are extremely powerful, but when implemented incorrectly, they can load a ton of unnecessary data when working with an API.

Implementing Many-To-Many Relationships

Many to many relationships are extremely common in most applications. Like the relationship name, they link together many of one resource to many of another. The most common implementation is a tagging system. Say you want to add certain tags to a resource. That resource has many tags and those tags can belong to many of the same resource.

An example of how we implement this in ROAST is with Brew Methods on a Cafe. An individual cafe can have multiple brew methods. An individual brew method can have multiple cafes.

There are actually a lot of scenarios like this within ROAST, however, the focus is not ROAST, but how you handle these scenarios in an API Driven Application. These relationships really only matter on the API side so we won't talk about them much from the perspective of an SPA.

Step 1: Set Up Your Many-To-Many Relationship

For this example, we will be setting up a many-to-many relationship between our cafes and brew methods. A cafe can brew coffee through drip, aeropress, French press, etc.

To initialize this relationship, we first need to create a Brew Methods resource. I won't walk through all of it, but it follows the same steps outlined in Efficiently Building API Endpoints for Data Queries. Essentially we need a database table (brew_methods) and a model Brew Method.

Once we have these set up, we need to create a pivot table that joins the two resources. The migration for our pivot table looks like this:

```
Schema::create('cafes_brew_methods', function( Blueprint  
$table ){  
    $table->bigInteger('cafe_id')->unsigned();  
    $table->foreign('cafe_id')->references('id')-  
>on('cafes');  
    $table->integer('brew_method_id')->unsigned();  
    $table->foreign('brew_method_id')->references('id')-  
>on('brew_methods');  
    $table->timestamps();  
});
```

Pivot tables usually have very few columns and exist solely for the purpose of joining two resources. However, you can add extra columns and store more information with the relationship if needed. I also like to use two plurals, cafes and brew_methods, in the naming of our table. This matches the relationship with many-to-many.

Our pivot table simply has a foreign relationship to the ID of the Cafe and the ID of the Brew Method. Next up, we have to model this relationship through Eloquent. To do that, open up both of the resources you are using (in our case Cafe and

Brew Method) and add the appropriate relationships. On our Cafe Model, we'd have a relationship to Brew Methods that looks like:

```
/**  
 * A cafe has many brew methods  
 */  
public function brewMethods()  
{  
    return $this->belongsToMany('App\Models\BrewMethod',  
        'cafes_brew_methods', 'cafe_id', 'brew_method_id')-  
        >withTimestamps();  
}
```

Like usual, I like to be as explicit as possible when setting up a relationship. In this case, we have use a `belongsToMany()` relationship and set up the parameters as follows. The first parameter is the model we are going to build the relationship to. In this case, it's `App\Models\BrewMethod`.

The second parameter, `cafes_brew_methods`, is the name of the table that we are using as a pivot table.

The third and fourth parameters are columns on the pivot table. The third parameter is the foreign key of the resource we are joining and the fourth parameter is the foreign key to the related resource.

Now how do we bind the two resources together?

Differences Between Attach, Sync, Sync Without Detaching, and Detach

So there are multiple ways to bind (and un-bind) two resources together in a many-to-many relationship and there are differences between how they act. It doesn't matter which method you choose, as long as you get the outcome you are looking for.

All of these methods are called on the relationship method on the model. Let's go through them.

Attach

This method is used if you want to join one resource to another. In our example, we want to add a brew method to a cafe:

```
$cafe->brewMethods()->attach( explode( ',', $brewMethods ) );
```

Two things to note about this method. First, it's called on the relationship method we defined on the Cafe model. Second, it takes an array of IDs of the resource we are joining. In this case, we have an array of Brew Method ids that we are joining to the cafe model.

However, with attach, it doesn't check if the relationship already exists. So you can end up with multiple of the same relationship (unless you have a joint primary key). That's where sync comes in.

Sync

The sync method operates very similarly to the attach method. However, it won't bind a resource if it's already bound. That means if we pass in a brew method that already has a relationship to the cafe, it won't get added again. I find that I use sync a lot more than attach. I almost explicitly use it when doing updates.

The other difference is it will remove any relationship that's not present in the synced relationships. So if you don't pass a brew method to this method that already exists on a cafe, it will be removed. Once again, this is great for updating.

To sync a relationship, it should look like:

```
$cafe->brewMethods()->sync( explode( ',', $brewMethods ) );
```

Once again, we have an array of brew method ids that will be joined to an individual cafe. Any existing relationships between the cafe and certain brew

methods that are not present in the sync are removed. There are times where you don't want to remove these relationships when you sync, but you want to add any relationships that don't exist AND not add any that do (no duplicates).

Luckily Eloquent provides a method that does that as well!

Sync Without Detaching

When you want to add a relationship but not twice and don't remove an existing relationship to a model, this is your method! Similar to the two methods to join two models, it's called on the relationship itself.

For our example, say we found out that a cafe has some new brew methods. However, maybe this is a user submission or from a 3rd party where they don't know if we already have a relationship between a specific brew method and the cafe. So they send along the array of brew methods.

To ensure that any new brew methods are added only once and none are removed, we call the following method:

```
$cafe->brewMethods()->syncWithoutDetaching( explode( ',',  
$brewMethods) );
```

Once again, this takes an array of brew method IDs as the parameter and joins the ones that don't exist to the `brewMethods()` relationship.

Finally, what happens if you want to remove a relationship?

Detach

To remove a relationship you just have to call the `detach()` method. Say a cafe no longer provides a specific Brew Method, you'd detach the brew method by passing an array of ids that should be removed:

```
$cafe->brewMethods()->detach( explode( ',', $brewMethods) );
```

Before we get to working with these relationships with respect to an API, I want to touch on one more subject and that's adding other data to your pivot table.

Timestamps and Other Pivot Data

I briefly touched on this, but you can actually add other pivot data to your pivot table. While we don't do this much in ROAST, I've done it in other applications and it can be extremely useful.

First up, let's say you want to add timestamps for when the two resources were related. You may have seen this when we built our model relationship, but you have to add the `→withTimestamps()` to the end:

```
/**  
 * A cafe has many brew methods  
 */  
public function brewMethods()  
{  
    return $this->belongsToMany('App\Models\BrewMethod',  
        'cafes_brew_methods', 'cafe_id', 'brew_method_id')-  
    >withTimestamps();  
}
```

Now as long as we added timestamps to our table, they will be updated when we join the relationship!

Other Pivot Data

Sometimes you want to store other data in your pivot table. Let's say that you have a special name for a brew method that a cafe wants to use (we don't actually do this in ROAST, but it's a good example!). Instead of Pour Over, the cafe explicitly wants to state they use Hario V60 pour over. To do this, we'd have another field named "`brew_method_name`" on our pivot table.

To add this data, we'd need to pass an associative array with any of the sync or attach methods discussed previously. For example, it'd look like:

```
$cafe->brewMethods()->attach( [  
    1 => [  
        'name' => 'Hario V60'  
    ]  
] );
```

In this example, 1 would be the ID of the brew method and name would be the additional pivot field we are updating with the relationship. With this in place, let's touch on what we need to do from an API perspective.

Step 2: Working with Many-to-Many and an API

What's so different about using these relationships in an API? Well first and foremost, you most likely don't want to return every relationship every time. While it's not efficient to do this even in a monolithic application, it's REALLY not efficient to do this in an API. What I mean by this is say we are just listing out cafes with basic data. We don't want to return every many to many relationship when we are just listing out data. However, what happens if we want to use that data in the listing?

Computed Properties

Another many-to-many relationship we have set up in ROAST is likes. A user can like many cafes and a cafe can be liked by many users. Well, this is a neat piece of UI to show in an SPA. However, if you have a really popular cafe, you could end up with thousands of likes. You don't want to return every like to show that number. Especially for every cafe if you are listing them out.

To get around this, you can do computed properties. Say you only want the number of likes a an individual cafe has:

```
Cafe::where('id', '=', $cafe->id)
    ->withCount('likes');
```

Or you want a flag if the authenticated user likes a specific cafe:

```
Cafe::where('id', '=', $cafe->id)
    ->withCount(['likes as liked' => function( $query ){
        $query->where('user_id', '=', 
Auth::guard('sanctum')->user()->id);
    }]);
});
```

By using computed properties, you don't have to return all of the relationships to get the data you need. I recommend that you do as much computation on the API side as possible to save on response size and make it easier to display in your

SPA.

Conclusion

Many-to-Many relationships are not specific to APIs by any means. However, they need to be approached with an API Driven mentality. Using computed properties saves on response size and allows for much cleaner SPA code.

Before we jump into adding any more specialty features and fun add ons, we will have to implement a little more security because there is a gaping security hole that could lead to corrupt data. We will touch on that next!

PERMISSIONS, VALIDATIONS, AND SECURITY

Permissions Overview

So that security hole. It's not necessarily a leak of data, but we do have an issue where we aren't protecting our data as well as we should. The issue is we allow anyone to add whatever they want and then modify the existing data by just by making a free account. This is definitely something we should account for. A malicious user could log in and corrupt everything which is NOT good.

So how do we resolve that? We come up with a permission system that allows certain users to access certain resources and make changes. At first thought a permission system probably brings up bad feelings of complication and heavy implementation, and you should feel those feelings. Permission systems and security can be an extreme pain. With Laravel though? It's nothing to be intimidated by.

Laravel provides a beautiful interface for this called [Gates and Polices](#). In this section we will be going through the following:

- Explaining the difference between Middleware, Request Validations, and Gates/Polices.
- Which security measure should you go through first
- Setting up a basic permission system
- How to handle data submission for non-authorized users.

This is going to be fairly intense, but before we think about heavy deployment and adding even more features, we are going to save some technical debt by diving into this security system right now. It's a lot harder to add permissions to existing code so I like to do it as early on as possible.

One quick note about the terminology used in this chapter. Authorization and Authentication are two different beings entirely. Authentication means a user is logged into the system while authorization means they have the permission to perform an action. A user can be authenticated by having an account, but may not

be authorized to perform an action.

Differences Between Middleware, Requests, Gates/Policies

There are 3 permission systems present within Laravel. We've already touched on 2 of these already, middleware and request validations, but we need to take a more holistic view and find out their place within our entire application.

Middleware

When I try to choose how I'm going to secure my application, I like to think about how fast I can prevent users from access, modifying, or creating data. From my perspective, I choose middleware as my first option.

However, I like to use middleware to do high level blocking. For example, if I can group multiple routes under a check if the user is authenticated or not, I prefer to do it this way. I like to think about the security of our application like the security of a building. Middleware is the security guard on the outside perimeter before the user even gets to the building. If the user is a guest, they can see the building, maybe park a car, but can not access the building or get inside the gates. If the user is authenticated, they get to go up to the building. From there, we can determine where to send them.

I rarely use middleware for permission system based on user structure. Now, the exception to this rule is if you have a group of administration routes that ONLY an administrator can access. This would be the backstage entrance to your building. We can have a whole separate area for those users to enter and block that with a specialized middleware. Otherwise I leave that up to the next layer, gates/policies.

Gates and Policies

This is the badge that gets you into certain rooms within your building/application. And this is where we are at within our application. We are blocking users from entering, but it's really easy for them to create an account and do whatever they

want within our application. We don't prevent them from exploring parts of the application they shouldn't or editing data that shouldn't be edited.

What Gates and Policies do is allow you to have fine grained control who can access what. Just because the user is authenticated, doesn't mean they can do everything. For example, an admin user can create and edit resources, but a general user should not have this power.

With Laravel, a gate and a policy are very similar, but two pieces of the same system. If you have a small permission system, you can use gates. According to the documentation:

Think of gates and policies like routes and controllers. Gates provide a simple, Closure based approach to authorization while policies, like controllers, group their logic around a particular model or resource. We'll explore gates first and then examine policies.

You do not need to choose between exclusively using gates or exclusively using policies when building an application. Most applications will most likely contain a mixture of gates and policies, and that is perfectly fine! Gates are most applicable to actions which are not related to any model or resource, such as viewing an administrator dashboard. In contrast, policies should be used when you wish to authorize an action for a particular model or resource.

Essentially a gate is a simple closure function that belongs a resource. A policy is a group of these permission systems related to a resource.

Want to know the monkey wrench? You can even apply these permissions to routes *through* middleware. And yes, in some cases we will be doing that, like with our admin only routes. We will be using a the policy as a middleware. The reason being is the functionality is clean, but more importantly, we can group all of our permissions into a single policy. We can block accordingly and there are some instances where we need like only 1 route to have middleware that's more secure than others. Once again, more of a "fine grain control" than just a huge outside perimeter. I like to think of the gates/polices in that more "fine grain" control.

Now what about Request Validations?

Request Validations

We've already used a few of these for adding and updating our resource and sub-resource. The way we used this is to validate our data. And that's the view we will continue through out our application. Requests should be used for validating data sent to a route.

You can apply authorizations to the request **WITHIN** the request, but I don't like to do that. Permissions should be handled through Gates and Policies, large blocks should be handled through middleware, let's keep requests to validating data once we get to the point.

There will be intertwined features where we validate the data and then see if the user has permissions to submit the data but those are few and far between. I will go through a few examples of these, but we will **STILL** keep the requests as only data validation and the policy as authorization. All behind a big perimeter of middleware.

Let's discuss our permissions

The best way to get started is to think about the permissions necessary to use your app. Even if these aren't final, which I doubt they will be, doing a little up-front planning will help for sure. In this section this might be a little more app specific than I'd want. But when we implement this, I will be using app agnostic principals in our example.

So right now, what we are going to do is take account what features we have and how we should add permissions to these features. Currently, we have the ability to perform CRUD (Create, Read, Update, Delete) features on a company and then add a cafe to the individual company and perform CRUD on that cafe as well. Any authenticated user can perform these functions which we want to change.

What we want is to limit WHO can perform these functions **AFTER** they are

authenticated. To do this, we will define 4 different user permissions:

1. Admin
2. Owner
3. Authenticated User
4. Guest

Let's break down how we will apply these permissions and what we need to do to make them work with our current system. This might not be as complex as your app, I mean you could have like 30 different permission levels and really fine grained detail. That's perfectly fine! These principals will apply to any application.

Admin Permission

This will be our top level permission. A user with an admin permission will have permission to perform any functionality within the application. The admin user will be able to perform all CRUD on all resources without any restriction. The admin user will also be able to manage the permissions of other users of the platform, including promoting users from authenticated, to an owner, or even up to admin.

Finally, an admin will be able to approve submissions by users for changes and additions to the data within the platform. This is a security measure I like to add to any application that I want users to feel free to add data to, but not go live right away. Let's say an authenticated user who is NOT an admin submits a new company. This would go into a "temporary action table" that an admin will need to approve before adding to the live data.

Now if the user was an admin, the company submission would go right into the production data with no approval. If you are interested in this approach, you can check out the Gitlab repository to see how we are able to store the submitted data for approval at a later date.

We will also have a few "non-publicly manageable" resources that only admins can

edit. These are app specific resources. For example, we will be allowing cafes to add brew methods so users can find cafes that match what they are looking for. The brew methods available to add will be managed through the admin section. Adding the brew methods to a cafe will be done through admin and owner of the company permissions.

Next, up we have a step down the chain to the Owner Permission. This will be the most unique permission level and make heavy use of the Gates and Policies set up.

Owner Permission

This is by far the most unique permission on our system and a great example of implementing a more complex permission structure. It's a stepping stone between a regular authenticated user and an admin. This permission level gives a user the permission to manage a company or other entity that they own, but not companies or entities that they don't. Essentially making them administrators **WITHIN** scope.

Let's say we have 2 different companies. One owned by user A and another owned by user B. We don't want user A to make adjustments to user B's company without being approved. We do, however, want user A to perform any company updates for their own company without being a part of the approval process.

We don't want a company owner to add a new company without an approval process, but they should be able to add a cafe resource to their own company. They can also promote general users to company owners for their own company, but only admins can promote to any company.

The owner permission will also apply to other scenarios such as future features for events and coffee schools. The owner permission is essentially a flag saying that the user owns an entity and it's up to the permission system to validate if they own the proper entity.

As you can see this "middle" level group can get extremely interesting, especially

as we expand our app. When we dive in more, you will see how this comes into play and how we can expand it more with more features attached.

Authenticated User Permission

An authenticated user has the least amount of permissions for registered user on our application. Any action submitted by the user will have to be approved by the owner or the admin. The only content the user will be able to manage is their own profile (and "like" cafes).

Guest

A guest is the most basic user of the application. They can view the companies, cafes, and other future resources. To even submit an action, they will have to create a user account.

Now that we have those permissions defined, let's stop and see if we can see what we will need for middleware, gates and policies. Right off the bat, I see any admin functionality relating to admin managed resources can be blocked through middleware. We won't even let other users access this data in any form. In any other route accessible by company owner or admin would be handled by gates and policies.

Now that we've got our system planned out, let's begin our implementation!

Setting up Permission System on Users

First, we will have to find a way to flag a user on a permission system. The implementation I find to be pretty straight forward, we will just have to add a flag for permission to the database record. To do that, I made a simple migration:

```
/**  
 * Run the migrations.  
 *  
 * @return void
```

```
 */
public function up()
{
    Schema::table('users', function( Blueprint $table ){
        $table->string('permission')->after('password')-
>default('user');
    });
}
```

What this allows is for us to assign a `permission` to the user. By default, the permission level is a `user`. I like doing this as a string so we don't pigeon hole ourselves with numbers. Before, I used to do a number and have it match to a permission level. This got difficult if we did something like 1-4 and needed to add another level between user and admin. Now our number system doesn't make sense.

With this system, we can add a name and not worry about the level. It's also good to incorporate readable code into our application. It will be easy to look at a user and our permission system to find out what we should be checking for.

We aren't done yet though. In order to have fine grained control over access permissions, we will have to build a secondary table to house which companies the user has permission to manage. Right now `owner` doesn't narrow down the specifics of what the user can manage. To do this, let's add a pivot table for our many to many relationship (a user can own many companies):

```
/**
 * Run the migrations.
 *
 * @return void
 */
public function up()
{
    Schema::create('companies_owners', function( Blueprint
$table ){
```

```
        $table->bigInteger('company_id')->unsigned();
        $table->foreign('company_id')->references('id')-
>on('companies');
        $table->bigInteger('user_id')->unsigned();
        $table->foreign('user_id')->references('id')-
>on('users');
        $table->timestamps();
    });
}
```

The `companies_owners` table will store a paired key of users who own a company. This will allow for a company to have many owners. It will also allow for users to be owners of many companies. Think of this as a user who "owns the right" to manage a company. With this set up, let's add the following to our `App\Models\User.php` model:

```
/**
 * A user owns many companies
 */
public function companies()
{
    return $this->belongsToMany('App\Models\Company',
'companies_owners', 'user_id', 'company_id');
}
```

and this relationship to our `App\Models\Company.php` model:

```
/**
 * A company has many owners
 */
public function owners()
{
    return $this->belongsToMany('App\Models\User',
'companies_owners', 'company_id', 'user_id');
}
```

So now that we have our structure set up, let's begin the implementation through middleware and policies.

Preventing Unauthorized Access Using Middleware

Our first layer of security is using middleware. As mentioned before, this is our outer most security. I tend to apply middleware to route groups to block off certain actions.

One thing I'd like to point out again is that you can apply your policies as middleware as well. However, I like to think of these as two separate pieces and use them in different scenarios. We will touch on using policies in the next chapter.

We've already applied a single piece of middleware to some of our endpoints through the controller. If we look at the `CompanyController.php`, and in the `__construct()` method, you will see the following code:

```
$this->middleware('auth:sanctum')->only(['store', 'update',  
'like', 'destroy']);
```

This blocks any of the requests to the methods defined in the `→only()` method for users who are not authenticated. Now that we have our permissions system in place, let's think about this through a different scenario. By blocking off access to certain endpoints based on the user's permission.

Step 0: A Side Note

When I determine if I need to create a middleware or a gates/policies, I like to think about how the resource is being accessed and what we are attempting to do. Honestly, besides basic authentication, I tend to use middleware for more system level blocking (CSRF protection, rate limiting, etc) than permission handling. I tend to apply gates/policies through middleware than use a middleware to handle permissions.

With that being said, there's nothing wrong or stopping you from using middleware to block off routes based on permissions. In super large APIs I will create an admin middleware and block off entire resource endpoints for admin users only. I'll show you how I'd apply that in ROAST, but to be honest, the guts of our permission system comes from gates and policies in the next section.

Step 1: Create Admin Middleware

If we were to block off parts of the application using admin middleware, the first step we would take would be to create our own custom middleware. Laravel makes the creation of middleware a breeze.

In your terminal, we'd first run:

```
php artisan make:middleware IsAdmin
```

This will create our new middleware file in the `app/Http/Middleware` directory. We will dive into this file soon, but for now let's register our middleware so we can use it.

Step 2: Register Middleware

Now that we have our middleware created, we have to register it so we can apply it to certain endpoints. To do that open up the `app/Http/Kernel.php` file and find the following array:

```
/**  
 * The application's route middleware.  
 *  
 * These middleware may be assigned to groups or used  
 * individually.  
 *  
 * @var array  
 */  
protected $routeMiddleware = [
```

```
'auth' => \App\Http\Middleware\Authenticate::class,
'auth.basic' =>
\Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
's',
'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,
'password.confirm' =>
\Illuminate\Auth\Middleware\RequirePassword::class,
'signed' =>
\Illuminate\Routing\Middleware\ValidateSignature::class,
'throttle' =>
\Illuminate\Routing\Middleware\ThrottleRequests::class,
'verified' =>
\Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
];
```

It's important to note that we are not using the api middleware group. This is because we want to apply the middleware to the individual endpoints we choose, not to the entire API.

Within that array, add the following in alphabetical order:

```
...
'guest' =>
\App\Http\Middleware\RedirectIfAuthenticated::class,
'isAdmin' => \App\Http\Middleware\IsAdmin::class,
'password.confirm' =>
\Illuminate\Auth\Middleware\RequirePassword::class,
...
...
```

Now we have our middleware registered and we can apply it to whatever routes

we need! However, we have to make the middleware functional first.

Step 3: Add Admin Check to Middleware

I guess it'd make sense to implement some functionality into our middleware now! Since we are doing a simple check and blocking off endpoints if the user is an admin or not, our middleware will be pretty straight forward.

Let's open up the `app\Http\Middleware\IsAdmin.php` file and add the following code to the handle method:

```
/*
 * Handle an incoming request.
 *
 * @param \Illuminate\Http\Request $request
 * @param \Closure $next
 * @return mixed
 */
public function handle($request, Closure $next)
{
    /*
     Any user with a permission that is not an admin will
     receive a 403 un authorized action response.
    */
    if (Auth::user()->permission != 'admin') {
        abort(403, 'Unauthorized action.');
    }

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

What this does is check to ensure that the user accessing the endpoint is an admin. If they are not an admin, then we abort the request and return a 403 which means Unauthorized Action so the developer can handle it accordingly.

Make sure you add the Auth facade to your use declarations on top of your middleware class:

```
use Illuminate\Support\Facades\Auth;
```

Some middleware can get pretty complex, but for this example it's a simple check. The next step will be applying the middleware to an endpoint.

Step 4: Implementing the Is Admin Middleware

Now that we have our middleware registered and set up, we now just need to implement it on an endpoint. If you remember in Step 2, we registered the middleware with a key of "isAdmin".

On any endpoint that you want to implement the middleware you just have to add the following to the constructor of the controller used to handle that endpoint:

```
public function __construct()
{
    $this->middleware('isAdmin');
}
```

Or the following if you want only some routes to block off any user that is not an admin:

```
public function __construct()
{
    $this->middleware('isAdmin')->only([]);
}
```

You can be extremely flexible with your middleware and how you choose to implement it. If you have a really large API, you can apply it as a group. I only do this if there is a resource that is ONLY accessible to admins and no one else. In the sense of Amenities in ROAST or Brew Methods in Roast, regular users can not submit changes or additions, only admins can. However, any user can load the

Amenities or Brew Methods. That means that the entire resource is not completely admin only which is why we didn't use a route group.

In a few chapters, we will implement middleware in NuxtJS to mimic our backend middleware. Let's first touch on how to use Laravel's Gates and Policies to provide authorization access on a more fine grained detail and even implement them as middleware. This is my preferred approach and hopefully you will see why!

Securing API Endpoints With Laravel Gates & Policies

In the last section, we went through adding an admin middleware to block off certain API endpoints. While this works, I prefer to use gates and policies wherever we can. It allows us fine grained control in complex scenarios. If you view the ROAST API source code, you will see that we have a variety of policies implemented and applied as middleware. Let's go through this process.

Step 0: Why Policies instead of Gates?

So why policies instead of gates? For me, I like to keep all permissions in a single file. It allows for scalability in the future, clean code encapsulation, and I know where to look if I need to make changes. Kind of like a bucket once again. We will have a bucket to place our permissions in and as we expand functionality. Policies are excellent for features that are accessed from multiple permission levels since you can account for whether a user in a variety of different scenarios can access a resource.

Let's add our first policy!

Step 1: Create A Company Policy

We can do this with a simple artisan command. Our first policy will be for our [Company](#) resource so let's run:

```
php artisan make:policy CompanyPolicy
```

Since we haven't created any policies yet, this will create an [app\Policies](#) directory and add a [CompanyPolicy.php](#) file within the directory. We will come back to this as soon as we register our policy.

Registering our policy will let Laravel know it exists and can be used on requests. There are ways to autoload these policies or "discover" them. I prefer to be

verbose and register them. To do that, open up the `app/Providers/AuthServiceProvider.php` file and add the following to your `use` declarations:

```
use App\Models\Company;
use App\Policies\CompanyPolicy;
```

Now that we have the classes being used in the auth service provider, we can map the class to the policy like this:

```
/**
 * The policy mappings for the application.
 *
 * @var array
 */
protected $policies = [
    Company::class => CompanyPolicy::class
];
```

What this will do is automatically bind the model we are validating into the policy so we can access it as needed. This will be very important for our owner permission level since we will need to see if the user is a registered owner on the company or other entity. With all of our policies, we will be registering them here. For example, when we add our `CafePolicy.php` we will be adding the following lines of code:

```
use App\Models\Cafe;
use App\Policies\CafePolicy;
```

and

```
/**
 * The policy mappings for the application.
 *
 * @var array
 */
```

```
protected $policies = [
    Company::class => CompanyPolicy::class,
    Cafe::class => CafePolicy::class
];
```

When looking at the company policy, the routes that should be protected by this policy are the routes used to create a company, edit a company, or delete a company. All three of these methods have the same logic. They can be processed if the user is an admin or an owner of the company.

However, if the user is not valid, we want to allow the user to submit a temporary action. Because of this, we won't be applying this policy as a middleware, but a little deeper and in the services themselves. The whole "temporary action" feature is kind of app specific, but also an example I've used in a lot of different apps. Check out the next chapter for more on the implementation.

Step 2: Add Policy Logic

When looking at a basic resource, such as a company, within our application we need to provide logic in our policy to handle the basic modifications for an entity. This will give permission on whether the user can modify resources or create resources based off of their user permission.

Let's go back into our `app\Policies\CompanyPolicy.php` and stub out the following methods:

```
public function store( User $user )
{
}

public function update( User $user, Company $company )
{
}
```

```
public function delete( User $user, Company $company )  
{  
  
}
```

also make sure we add the proper `use` declarations on top of the file:

```
use App\Models\User;  
use App\Models\Company;
```

In this instance, we only want the modification of the resource to have certain authorization parameters. We don't need any authorization on viewing any piece of this resource. Later on, with other resources such as job applications or event sign ups, we definitely will need an authorization based protocol set up to handle this since we don't want to leak public data.

Let's take a look at the `store()` and `update()` methods, the `delete()` method will follow the same logic as the `update()` method.

The `store()` method signature accepts an injected instance of the user which will be the currently authenticated user. To store a new company directly in the database, we will need to be an admin. Since it's a new company, there is no direct owner yet. That would be assigned after the creation. The final implementation of this method will look like:

```
public function store( User $user )  
{  
    if( $user->permission == 'admin' ) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

This method simply accepts a user, which will be the authenticated user. If the

user has a permission level of `admin` we return `true` meaning they have permission to perform the proper storage. If they are not an `admin`, we return `false` blocking them from performing the action. We will handle both of these scenarios in the next step.

With an API Driven Application, we want the route to be accessible to users but we need to apply this permission on the API side since any SPA "security" should be deemed unworthy. Remember the entire SPA is loaded client side. If someone wanted to really modify the code, they could and that could wreak havoc on our database.

Next, let's check out the `update()` method. The `update()` method has an additional parameter passed which is an instance of a company model. Since we are updating an existing company we can apply some different logic and a little more fine grain control. Remember our `owner` permission? A user who is not an `admin` can own a company and therefore manage that specific resource. Let's take a look at what that looks like:

```
public function update( User $user, Company $company )  
{  
    if( $user->permission == 'admin' ){  
        return true;  
    }else if( $user->companies->contains( $company->id ) ){  
        return true;  
    }else{  
        return false;  
    }  
}
```

When we call this method in the next steps, we will be passing the company along with the user. The first layer of security is if the user is an admin. If they are, then return `true` they can definitely perform the action. Next, we check if the user's company relationship contains the id of the company.

Remember, this is the many-to-many relationship that we set up. We can call the

`contains()` method on the relationship and pass in the id of the entity we are checking to see if there is a relationship set up. If there is, that means the user is an owner on the company and we return `true`. Finally, if neither of those scenarios are true, we return `false` and the user doesn't have any of the necessary permissions. The `delete()` method is implemented the exact same way with only allowing admins and owners to perform the method.

Now that we have our policy logic laid out, we can begin to implement the logic within our application. As our app grows and we add more policies, this flow for resources will generally remain same. Even our cafe policy follows a similar structure since it is a nested resource. If the user has permission to manage the company, they have permission to manage the cafe as well. When we add more features, if we ever come across a more unique scenario, I'll for sure document it and explain it in full. It's time to begin the implementation.

Step 3: Implementing Policies within Code Blocks

One of the ways we can implement a policy into our application is through blocks of code. Policies have a specialized method named `can()` that literally checks if the authenticated user can perform the action. You will see a mixture of code block implementation and middleware implementation in ROAST. We are doing this as a code block now because we want un-authorized users to submit data, we will just handle it differently.

Let's start with the most basic example of creating a resource, in our case a company. To do this, look at the `app\Services\Company\CreateCompany.php` service and find the `save()` method. In there, let's do a little refactoring to implement our policy.

First, let's extract the code from the existing `save()` method and make a new private method named `persistCompany()`:

```
private function persistCompany()  
{
```

```
$company = new Company();

$company->name = $this->data['name'];
$company->roaster = $this->data['roaster'];
$company->subscription = $this->data['subscription'];
$company->description = $this->data['description'];
$company->website = $this->data['website'];
$company->address = $this->data['address'];
$company->city = $this->data['city'];
$company->state = $this->data['state'];
$company->zip = $this->data['zip'];
$company->facebook_url = $this->data['facebook_url'];
$company->twitter_url = $this->data['twitter_url'];
$company->instagram_url = $this->data['instagram_url'];
$company->added_by = Auth::user()->id;

$company->save();

$this->saveImages( $company, $this->data );

return $company;
}
```

Now that we have that method in place, let's go back to the `save()` method and implement the policy logic:

```
public function save()
{
    if( Auth::user()->can('store', [ Company::class ] ) ){
        return $this->persistCompany();
    }
}
```

This is all we need to do to ensure we have a proper user who can perform a persistence to the database! Like I've mentioned before, to perform an action on an authenticated user who doesn't have permission, check out the source code

for how we set up our temporary data structure.

Let's digest this a little bit. We got to this point through a couple different forms of security. When a user sends a `POST` request to `/api/v1/companies` we first check if they are authenticated through the middleware. If they are authenticated, we validate that the data they are sending is what we need through a `request`. Once that is validated, then we hit this method to determine if they can actually perform this function.

That may sound a little backwards for some scenarios and it is. If we were entirely blocking off the method before even checking the data, we would be applying the policy as middleware instead of after the validation request. The reason we do this after the request, is because later down the logic stream, we determine how we want to perform the action. Do we want to persist the data right away if they are an admin? Or, do we want to create a temporary action and handle the request manually at a later point?

Now with this method, let's take a look at the first line:

```
if( Auth::user()->can('store', [ Company::class ] ) )
```

What this does is check to see if the currently authenticated user can run the `store` method. The second parameter is where it gets a little bit different. Since we aren't checking permission on a specific model, we just pass the class for the policy we want to use. Remember when we registered the policies in the `AuthServiceProvider.php`? This is why we did that. We could have multiple `store` permissions (and we do when storing a cafe), so we need to direct Laravel to check the `store` permission on the `Company` policy. That's why we pass the `Company::class`.

If that passes, meaning the user is an admin, we run the `persistCompany()` method that saves the company right away!

When I first implemented these authorization methods, I was like why don't we just put it all in the `if` statement in the code? I mean, that's what we are doing

anyway? Well, when I had to add permissions and check for different parameters later on, we had one single place we could update to do that, and that was in the policy. So when I used the permissions in like 10 different areas, I only had to update it once which was awesome! Policies completely organize code and have a single point of entry.

To take a look at another scenario, open up the `UpdateCompany.php` and look at the `update()` method. We once again, refactored the code out of their into a `persistUpdates()` method and applied the appropriate policy:

```
if( Auth::user()->can('update', $this->company) ){
    return $this->persistent();
}
```

So the difference between the `create` and `update` policy method is that in the `update` method accepts an actual instance of a resource that we are checking the policy against. When updating a company, we have the company we are updating, injected into our route so we know which specific company it is. The route is blocked by middleware and the data is already validated, we just determine what we want to use to handle the request, should we persist it, or handle the action later?

So that's one way of applying policies, the next is to apply it through middleware.

Step 4: Implementing a Policy as Middleware

In order to give an example on how to implement a policy through middleware, we are going to create a basic resource that only admins can manage. This way, it's a cut and dried way of blocking off a large chunk of functionality that no other user "might" be able to edit, like an owner of a resource.

What we will be doing is management of Brew Methods. Brew methods in our application are ways that a cafe can brew coffee such as espresso, pour over, siphon, etc. We want these Brew Methods to be managed by the admin if a new method becomes available or an industry name changes, etc. We will be able to

filter cafes by what brew methods they offer. Those who enjoy their coffee a certain way will definitely find this valuable!

The concept of an admin only resource can be used in a variety of scenarios such as managing teams, updating permissions, or really anything else that relates to the business logic of an app.

For this example, I went ahead and actually built the entire functionality for our brew method management. I'll share what is necessary to make sense for creating a policy.

So the Brew Methods resource has the standard routes for any resource, an `index`, `show`, `store`, `update`, `destroy`. Unlike other resources, we have some unique permissions. We want the `index` route to be publicly available, but every other route to be only accessible by an admin. No other user should access these routes, not even owners. Because of this black and white difference, we can apply a middleware to these routes.

Let's open up the `app/Http/Controllers/API/BrewMethodsController.php` and look at the `__construct()` method:

```
public function __construct()
{
    $this->middleware('auth:sanctum')->only('show',
'store', 'update', 'destroy');
    $this->middleware('can:show,method')->only('show');
    $this->middleware('can:store,App\Models\BrewMethod')-
>only('store');
    $this->middleware('can:update,method')->only('update');
    $this->middleware('can:delete,method')->only('delete');
}
```

This is where we apply all of the middleware on a controller. Like I mentioned before, I like doing this because it keeps our routes file clean and focuses the permission and auth logic in a convenient location.

Let's take a look at the middleware: `$this`

```
→middleware('can:show,method')→only('show');
```

What this is doing, is blocking off the `show()` method by utilizing our policy. The middleware checks if the authenticated user can perform the `show` action. The syntax for this is a little weird because after the `,`, we have `method`. Now remember when we set up our dependency injections? Well on our Brew Method routes, it's binding to the `method` route parameter and injects the brew method into that route.

With the `show` command, we are ensuring that the authenticated user is authorized to `show` a specific brew method. This is the same for the `update` and `delete` methods as well. The only difference is the `store` method which accepts the `App\Models\BrewMethod` as the class itself. This is similar to when we applied the policy to our other routes that didn't accept a specific entity.

We've also applied the polices as middleware to their corresponding routes by using the `only()` method. Now there's another way you can apply policies and that's through controller helpers. This would involve adding a line similar to: `$this→authorize('update', $brewMethod)` in the actual controller function itself.

Why didn't I do it that way? I like keeping those controller methods as clean as possible. The less business logic in the method itself, the better. I also feel that since we can apply this on the middleware level, we can stop even reaching that method if we don't have to and we can keep all security of the controller in one convenient location. This brings us to the next subject.

The order of middleware counts! If you look back at the `__construct()` method, we still have our authentication middleware as the first one: `$this`

```
→middleware('auth:sanctum')→only('show', 'store', 'update', 'destroy');
```

. Why do we have that first? Think of it as the biggest gate. Is the user authenticated or not? If they aren't authenticated, then we know for sure they are not an admin so we block the access right away. We also return a `401` error

which means **Unauthenticated** compared to a **403 Unauthorized** error if the middleware fails.

There will be a lot more examples of using policies throughout our application. Anytime anything unique comes up, I'll document it and explain it. If you want to see the full source code, check out [Gitlab](#) and see it live in action.

Now that we have our policies implemented, let's take a look at request validators, then move to the NuxtJS side of the application and create the corresponding middleware.

Implementing Laravel's Custom Validation Rules

Another place to implement security in your application is through request validations. These request validations ensure that the data being submitted to an endpoint is valid. You can also authorize users through these request validations. I don't recommend it since we have policies that are dedicated to do this.

I do recommend, however, using request validators on any request that has rules on what data can be sent to it.

We've already went through a few examples of implementing request validations. The first one was when we set up registration in Implementing Authentication and Registration With Laravel Sanctum. We used a validator in a method to ensure the data submitted to create a user was valid. The second request validator was in Efficiently Building API Endpoints for Data Queries where we added a request validator file that handled the error messages and data validation before continuing the request and processing the data.

What we are going to do now is expand upon the the request validation file by implementing a custom validation rule. Before you choose to add a custom request validation rule, I'd recommend taking a look at what [Laravel already offers](#). They have a lot of the bases covered, but they allow you to make app specific validations if needed. That's what we will do in this chapter!

Step 1: Create Custom Rule File

For this example, we will be creating a rule that will apply to when a user updates their password. To validate the request, we want the user to verify their existing password before they update their password. This may be a little app specific, but the principals will be the same for any custom validation you want to create.

The first step we need to do is create our custom rule with the following command:

```
php artisan make:rule ValidExistingPassword
```

This command will make a new directory called [App/Rules](#) and a file named [ValidExistingPassword.php](#).

Step 2: Adding Functionality to the Validation

Let's take a look inside the file that was just created. The newly created file should look like this:

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class ValidExistingPassword implements Rule
{
    /**
     * Create a new rule instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Determine if the validation rule passes.
     *
     * @param  string  $attribute
     * @param  mixed   $value
     * @return bool
     */
}
```

```
 */
public function passes($attribute, $value)
{
    return true;
}

/**
 * Get the validation error message.
 *
 * @return string
 */
public function message()
{
    return '';
}
```

The method we are going to focus on is the `passes()` method which accepts an attribute and a value. The attribute being the name of the value being sent. The value is the value of the attribute. This is what we want to check. The `message()` method allows you to return a meaningful message back to the user.

Let's take a closer look at the `passes()` method and add the following code:

```
if (Hash::check( $value, Auth::guard('sanctum')->user()-
>password ) ) {
    return true;
} else{
    return false;
}
```

This method simply returns true or false depending on if the value is valid or not.

What we are checking is if the hash of the value sent in is equal to the password of the user. If it is, then the user has submitted a valid password and we can allow them to update their password.

If you are implementing this specific validation, ensure you add the following to the use declarations on top of your file:

```
use Auth;  
use Hash;
```

The next part we want to update is the message() method. This method should simply return a clean message stating “Your existing password is invalid. Please enter your existing password”:

```
/**  
 * Get the validation error message.  
 *  
 * @return string  
 */  
public function message()  
{  
    return 'Your existing password is invalid. Please enter  
your existing password';  
}
```

The messages are extremely important when designing an API. We want to give the ability to the front end dev to display a meaningful message without writing a bunch of extra code.

That’s all it takes to implement a custom rule! Now, let’s added it to a request validator.

Step 3: Implementing the Custom Validation

So now that we have our custom validation created, we need to implement it. The first thing we need to do is find the request validator we want to use the custom validation with. In our example, we have an `UpdateUserRequest.php` request validator in the `Requests/API/User` directory.

The first step we have to do is include request validator in our use declarations:

```
use App\Rules\ValidExistingPassword;
```

Now that we have it available we have to apply it to the field that we wish to validate. In this case, it's the `existing_password` field. This field is required with the password field since we only check the password if it's being updated. Our `rules` method, we have the following:

```
public function rules()
{
    return [
        ...
        'existing_password' =>
        ['required_with:new_password', new
        ValidExistingPassword()],
        ...
    ]
}
```

As you can see in the rules array for the `existing_password` field, we have the new `ValidExistingPassword()` as a rule. This creates our new custom rule object and automatically passes the attribute and value parameters to the `passes()` method when called. We then take the value, validate it and return true or false.

That's all we need to do to add custom validation rules to our requests! These can get as complicated as needed to fit your app, but the ending should be a boolean of if it's valid or not.

Next up, we are going to switch to the SPA and implement some middleware to help the UX match the functionality we are looking for!

Securing Our Front-End With NuxtJS Middleware

So now that we have our user permissions set up and configured, we have to mimic their functionality on the front end in NuxtJS. What's a benefit for us is Nuxt has convenient middleware and we can access the user object on the front end pretty simply.

The negative side is since it's javascript (client based), the user if they are malicious can kind of mess around with the javascript and see what they can uncover. With that mind, we should NEVER assume that javascript permissions are 100% perfect. We should always account for tinkering and limit the damage control. Most of that can be mediated by good API security meaning worst case scenario, they submit bad data and get a 403. With that being said, let's implement our middleware on NuxtJS and get the party started!

Now with front end security, we don't have policies available like we do on Laravel. Because of this, we have to use middleware to block routes and structure it similar to our policies.

We will start with the owner middleware.

Step 1: Build owner middleware

Before we get started, let's review where we stand. We have 4 possible security levels, guest, user, owner, admin. An owner is elevated to manage their own resources and an admin can do everything.

So thinking about it, we want an owner to be able to see certain management sections of the app, but only handle data on what they own. The data that populates these pages or what the pages allow for processing will be decided by the API. On the front end, we just need to display these features if the user is an owner. To add this middleware, simply add the following file `/middleware/owner.js` and add the following code:

```
export default function( context ) {
  if ( !context.$auth.loggedIn || context.
$auth.user.permission == 'user' ) {
    return context.redirect('/');
  }
}
```

To break this middleware down, we first check if the user is not logged in or if they are logged in and their permission level is `user`. If either of these are true, we redirect back to the home page.

What this does is allow any user that's an `owner` or an `admin` access to the page. What ever data is loaded onto the page will be determined through our source of truth in the API.

That's all we have to do for the owner middleware! The hard part with authorization and permission is handled on the API side. To implement this middleware, you'd simply have to apply it to the `middleware` attribute on your component. This middleware will come in handy as we get into more complicated features, but for now, we just needed to create it. We will touch on a few more implementations with our `admin` middleware next!

Step 2: Build admin middleware

The initial construction of this middleware is pretty much the exact same as the `owner` middleware. We first have to add the following file `/middleware/admin.js` and add the following code:

```
export default function( context ) {
  if ( !context.$auth.loggedIn || context.
$auth.state.user.permission != 'admin' ) {
    return context.redirect('/');
  }
}
```

So what this middleware does is check to see if a user is not logged in, or the user does not have an admin permission. If either of those are true, we redirect them home, they are not authorized.

Now, we get to implement this middleware. All middleware within NuxtJS is implemented the same way, on the `middleware` property on a page component. Since the admin middleware is more powerful and we want to add a lot of functionality for administrators, we are going to create an entirely new section that will be blocked off by this middleware.

In your application you can follow a similar flow if you want. For our example, we are using our brew methods resource. This resource should only be manageable by admins, so it will be the first section we add. In the future we will be handling user promotions from here, other app settings, etc.

The first thing I did to implement an administrative section into our front end SPA was to create a `/pages/admin` directory. The first page I added was `/pages/admin/index.vue` which is just a simple landing spot to help coordinate what the administrator wants to manage. In page, I added the following in the script tag:

```
export default {
  layout: 'App',
  middleware: 'admin'
}
```

What this does is first apply the `App` layout to the page, but then it also applies our new `admin` middleware. Remember NuxtJS generates URLs based off of folder directory? In this case, if a non-authenticated user or user with a permission level of `user` or `owner` visits `/admin` on the front end, they will be re-directed home. Only users with proper permissions can access this page.

In reality, that's going to be repeated for any page within our `/pages/admin` directory.

While we are in this directory, I added another directory called `/admin/brew-method`. That's where we will layout out the management of our brew methods

resource similar to any other resource we are adding in the application. All pages within this directory will have the `middleware: 'admin'` property set so no other users can access. Since we are making requests to our API from the authenticated user, these requests will be blocked if the user is not an admin as well. This way we have security in both the SPA and the API.

Step 3: Add Link To Admin

So now that we have our user being loaded into our application, we have one more scenario we need to account for and that's blocking off parts of components based on user permission level.

For an example, we are going to block off the link to the `/admin` section if the user is not an admin. We can't use a middleware since it's within a component, we need fine grained control.

Say we add a link in a component. We'd have to check if the user is logged in and has a permission of admin. Luckily NuxtJS allows us to do that with ease:

```
<nuxt-link :to="/admin" v-if="$auth.loggedIn &&
$auth.user.permission == 'admin'">
    Admin
</nuxt-link>
```

So what we are doing is showing this link in our navigation if the user is authenticated and they are an admin. This will allow all admins to get to the admin dashboard extremely easily and hide the link if the user is not an admin.

Next up, we are going to handle un-authorized actions on the front end gracefully so we can encourage and inform the user to stay.

Handling Unauthorized Actions on the Front-End

So we have some front end middleware set up to prevent certain actions and restrict access, but what about more inline functionality? Say you want to display to your user's a piece of functionality when they are not logged in that when they interact they get a prompt to log in. What happens after they do log in? These are the un-authorized actions we will be going through. The way I like to approach them is to save them in a piece of state that allows you to re-direct after authentication so the flow is consistent.

For this example, we will be using the “like” company functionality. A user, if authenticated can like a company. If they are not authenticated, they get prompted to log in, then when they authenticate their submission goes through.

Step 1: Plan Your Pre-Auth Actions

We will be referring to these actions as pre-auth actions. They are actions that show the intent of the user before they are authorized to perform the action. They only exist on the front end side of the application and are handled through NuxtJS. We want to make a simple way to re-use this code for multiple actions in the future. To do this, I implement a basic `pendingActions.js` file in the `/store` directory. It's a basic Vuex store that has a piece of state and a mutation. My initialization looks like this:

```
export const state = () => ({
  preAuthAction: {}
});

export const mutations = {
  setPreAuthAction( state, action ){
    state.preAuthAction = action;
  }
}
```

```
}
```

The `preAuthAction` piece of state will hold the object that contains what the user was intending to do. This will be completely re-usable and able to be implemented in a variety of scenarios. The `setPreAuthAction()` mutation will simply set this piece of state in our Vuex store.

If you've come from a VueJS background you had the option of namespacing your Vuex modules. In NuxtJS this is done by default. The access to this module's mutations and state is `pendingActions/setPreAuthAction` and `pendingActions/preAuthAction`.

Step 2: Design Your Pre-Auth Action Object

Now what will this `preAuthAction` object look like? I tried to make it as flexible as possible to house multiple different actions that the user can perform. The only requirement for this object is that it has an action key. This will determine what functionality we run and if we should re-direct the user after they have authenticated/authorized.

However, this object can contain as many keys necessary to complete the task. When we implement this on the `likeCompany()` method, we will include the company in the object so we can handle the action as needed.

The flow of how we handle these actions is as follows:

1. User tries to perform an action they don't have access to.
2. A `setPreAuthAction` mutation is committed with information regarding what the user is trying to do.
3. The user is prompted to log in.
4. After a successful log in attempt, we check to see if there is a

`preAuthAction` set in the state.

5. If there is a `preAuthAction` set, we run the action and clear the state allowing the user to continue using the application.

Let's take a look at how this is implemented for liking a company.

Step 3: Implement Pre-Auth Action for Liking A Company

On our individual company page, we have a button where a user can like a company. This is present whether the user is authenticated or not. However, to complete the action the user must be authenticated. Let's take a look at how this method is set up:

```
async likeCompany(){
    if( this.$auth.loggedIn ){
        let toggleLike = await this.
$api.companies.like( this.$route.params.company );
        this.liked = toggleLike.status;
        this.likes = toggleLike.likes;
    }else{
        this.$store.commit( 'pendingActions/
setPreAuthAction', {
            action: 'like-company',
            company: this.$route.params.company
        } );
        EventBus.$emit('prompt-login');
    }
},
```

This method simply does two things, first it checks if the user is authenticated. If they are, well then we just run the action and everything is fine! The user either likes or dislikes the cafe. We want to focus on what happens if the user is not authenticated. We commit a `pendingActions/setPreAuthAction` mutation.

This passes along the action and the slug of the company from the URL. We will use this later. Then we emit the event to prompt the user to log in. Right now we have this pre-auth action saved for later and will deal with it after a successful log in.

If we take a look at our login.vue component, we added an event in the successful login callback that emits an event we can listen to:

```
EventBus.$emit('roast-login');
```

Now, when the user has successfully logged in, an event is propagated through our application that we can listen to and perform actions. In this case, we will jump back to our company page and look at the `mounted()` hook:

```
EventBus.$on('roast-login', function(){
    this.runPreAuthAction();
}.bind(this));
And while we are at it, the runPreAuthAction() method:
runPreAuthAction(){
    switch( this.preAuthAction.action ){
        case 'like-company':
            this.likeCompany();
            break;
    }

    this.$store.commit( 'pendingActions/setPreAuthAction',
    {} );
},
```

When the user has successfully logged in, a roast-login event gets propagated. We can listen to this event and act accordingly. In our company page, we act on this event and call the `runPreAuthAction()` method. In that method, we have a switch case of the pre auth actions that the page can handle. Since we were intending to like a company, we can handle this action on this page. In this sense we are just submitting the `LikeCompany()` method already available on behalf of

the now authenticated user!

This is my preferred way of handling actions while presenting available functionality to the user. You can handle redirects to pages that require authentication, complex state transformations, etc. But having this simple process in place allows you to ensure the flow of the user through your application is seamless and enticing!

Displaying API Errors to Your Users

We've configured our API to send back useful error messages to our users, so how do we take advantage of it? These error messages tend to come after form submissions and return extra data that is not possible to process on the SPA side of the application. Such as if a user's password is valid or if the email address has been taken or not.

Accounting for these errors and providing valuable feedback for your users will help with the debugging process if needed and guide your users on how to use your app. Let's take a look at our `Login.vue` component for an implementation of how we provide an invalid credentials flag to our users so they know to re-try their authentication credentials.

Step 1: Handling Errors in Requests

For all requests to our API endpoints, we use the `axios` library. This is a promise based library that allows us to handle successes and failures from a request to the API. We've talked a lot about handling successful requests with the `.then()` callback, but what about errors? Luckily there's a `.catch()` callback that allows us to handle any errors in the requests that we submit to the API.

Within our `Login.vue` method, we do this in our `login()` method. If the user submits invalid credentials, we catch that response and display it to the user. Before we dive into that, we should take a step up and look at our validation object.

Step 2: Structure Validation Objects To Display API Errors

Before we can handle the error, we have to find a way to display these errors to the user. I display all errors on forms through a corresponding validations object. This way I can flag if a field is invalid, display it properly, and display a message that alerts the user of what went wrong. In our `Login.vue` component, the

validations object is housed in the `data()` method and looks like:

```
validations: {  
    email: {  
        valid: true,  
        message: ''  
    },  
    password: {  
        valid: true,  
        message: ''  
    },  
    invalidLogin: {  
        valid: true,  
        message: ''  
    }  
}
```

Each of the keys in the validation object correlates to a field in the form with the exception of the `invalidLogin` key. This key is filled with the results of our API request. The reason we structured this validation object this way is so we can dynamically set the messages and dynamically toggle whether the field is valid or not. This form is the exception where we don't have specific API responses on a per field basis. But if you were to validate say the name of the company server side, upon the failure, you could populate the validation object with the message from the API endpoint and flag it after a failed request.

In the case of our login form, we want to display whether or not the entire form is invalid, not a specific field. This is why we have an `invalidLogin` key. Let's take a look at our request now.

Step 3: Listen to Failed Requests

If you check out the `login()` method in `Login.vue` you will see something similar to this:

```
this.$auth.loginWith( 'laravelSanctum', { data: {
    email: email,
    password: password
} } )
.then( function(){
    // Handle successful response.
}.bind(this))
.catch( function( error ){
    this.validations.invalidLogin.valid = false;
    this.validations.invalidLogin.message = 'Invalid
credentials, please try again!';
    this.validations.email.valid = false;
    this.validations.password.valid = false;
}.bind(this));
```

This is our submission of the log in form to the server for authentication. We've already covered the successful request, what about the un-successful authentication attempt? Check out the `.catch()` callback. In there, we pass a callback function with the error parameter. On larger forms this will contain the errors from our request validations that we can use to display to the user along with the fields that were invalid. You'd be able to access these error messages through:

```
error.response.data.errors
```

This will be in an object with all of our validation messages for each key. In our login form, we just handle the error simply knowing that the credentials didn't match. When we `catch()` an error, we flag our `validations.invalidLogin.valid` as false and we set the message. We also flag both other fields because the combination the fields resulted in a bad request.

Similar to the `.then()` callback, we have to `.bind(this)` to the request. Otherwise we won't have access to our Vue component's local data. There's a variety of ways you can display these errors, but with Axios being a promise

based http client, it's great to take advantage of the success and failure of promises. Combined with the request validations provide by Laravel, you can make a very intuitive interface for guiding the user through your application!

BUILD IOS & ANDROID APPS WITH CAPACITORJS

Installing and Configuring CapacitorJS

Here is the piece that makes all the hard work pay off! The theme and entire purpose of this book is to “Develop web and mobile apps all from the same codebase.” That is exactly what we are doing here. In this section we will configure our front end NuxtJS app to work seamlessly within [CapacitorJS](#). What does that mean? It means we are going to use the exact same code for web as we do for iOS and Android saving you hours of time, thousands of dollars, and reaching an entirely specialized audience on multiple platforms.

For those who are not familiar with Capacitor JS, it's a bridge between the frontend and the native mobile layer allowing you to take your front end javascript code, package it up and distribute it as a mobile application. They also wrapper libraries allowing you to integrate with native mobile functionality so you can use features like geo location, take photos, access local file system, etc.

To be honest, this is the goal of developing an API driven application with a completely separate front end. After this section, you will use the same code as your web frontend but have the ability to package and deploy it as an iOS and/or Android app!

Before we get started, this isn't a guide on how to get into the App Store. You must abide by each app store's requirements and design an app worthy of the App Store. If you implemented your app with proper responsive design techniques and a beautiful frontend that scales to mobile, this will be a lot easier. If you haven't made a mobile layout for your application yet, I'd recommend re-factoring your UI to work seamlessly on mobile.

With that being said, let's install capacitor!

Step 1: Install CapacitorJS Package

The most amazing feature of CapacitorJS is it installs right along side your NuxtJS front end, in the same repo! If you've worked with Cordova in the past, you realize

how much of a pain it would be to maintain a mobile repo and a front end repo even though they do the same thing. CapacitorJS takes this pain away and provides so many more seamless integrations and first class support for quality plugins.

To install the CapacitorJS package, you will have to run the following command in your frontend repo:

```
npm install --save @capacitor/core @capacitor/cli
```

This will install CapacitorJS into your project and provide you with the necessary build tools to package your NuxtJS frontend into a mobile application!

On a quick side note. This is only possible because of our two separate repos and how we have the authentication set up. If you were following along with the Server Side Up series, we housed the SPA within the Laravel Install. We depended upon blocking off features based on session authentication. This would not work in a mobile setting!

Step 2: Initialize CapacitorJS

To initialize CapacitorJS you just have to run the following command:

```
npx cap init --web-dir
```

This command will guide you through the process of setting up CapacitorJS and some of the configuration to help bundle your mobile app.

If you look at the documentation for CapacitorJS, there's a flag that you can set at the end of this method to set the web directory for the install. This is the `--web-dir` flag. It is extremely important to run the initialize command with that flag. If you don't, you can always set it later, but you need to set the web directory to be the dist directory. This is where NuxtJS compiles your app when it's ready to build.

Remember when we first initialized NuxtJS we had it set up as Universal which meant it compiled the application as a Server Side Rendered app? Well, when we compile to mobile, we will be doing this as a Single Page Application. The difference being is all assets will be loaded dynamically and not pre-rendered. We do not need pre-rendering. It's actually discouraged since it increases the end package size for the application.

We will create this build command in the next chapter. For now, just know that it's extremely important to set your `--web-dir` flag when initializing or after you've initialized CapacitorJS open the `capacitor.config.json` file and update the `webDir` key to be `dist`:

```
{  
  "npmClient": "npm",  
  "webDir": "dist",  
  "plugins": {  
  
  },  
  "server": {  
    "iosScheme": "roastandbrew"  
  },  
}
```

CapacitorJS will now bundle this directory when creating your mobile apps!

Step 3: Add Platforms

This is truly the magical part of CapacitorJS. Adding the platforms is so easy and they script all of the projects so you can easily deploy to any platform.

Adding iOS

To add iOS, simply run the following command:

```
npx cap add ios
```

That's it! If you look inside of your front end directory you will see a newly created ios directory that contains an Xcode project and every plugin you need. Now when working with CapacitorJS, if you have to set a specific iOS value or some special feature for your bundle or deployment, it's done inside of the Xcode (or Eclipse if Android) project. This gives you full control of your application specifically for each platform but also lets you re-use the core of your front end. In Cordova, you'd be configuring a huge file with all of these customizations.

Adding Android

Like iOS to initialize Android on your platform, simply run the following command:

```
npx cap add android
```

Now you can see a new android directory within your frontend with a Gradle project and all of the configuration ready to run your application!

That's all we need to do to install and configure CapacitorJS on a base level! We have a lot of customizations that I like to set up to make managing the project a lot easier and make the deployment much more scalable.

Let's get to speeding up the build process with a single command.

Step 4: Android Configuration

There's one final step we need to make to ensure that our mobile app works on Android. That's to add the proper public path to the build configuration in the `nuxt.config.js` file:

```
build: {
  publicPath: '/nuxt/',
  /*
   ** You can extend webpack config here
   */
  extend(config, ctx) {}}
```

},

The Android app won't load correctly unless this is configured!

Server-Side Rendering vs. Single-Page Application

I just wanted to touch on this a little bit more to explain the differences and reasons for choosing SPA for our mobile app and Universal (SSR) for our web app. In the next section we will automate this choice when we write a command to handle the set up of this for mobile.

Let's get a little bit more in-depth. So NuxtJS has two different render modes to choose from SPA and Universal (SSR). Each of these render modes has different benefits and should be used as a tool that fits the job.

We chose to use Universal (SSR) for our web application for a variety of reasons. The main purpose being the pre-rendering allows us to optimize for search engines (SEO). Universal allows this because upon response from the server, the page is already pre-rendered, similar to if you were writing a PHP or Laravel application using blade templates. This allows you to configure what will be in the header meta data, the text content of the page, the structured data (JSON-LD), etc. upon response which will be picked up by search engines and indexed. If your application requires dynamic SEO for each page it creates, then this is the route I'd recommend.

When doing Universal and using `asyncData()` (which we touched on when we set up our API endpoints), it won't return the page until the data has been loaded from the API. This is super convenient because the node server that comes pre-packaged with your application will make a request to our Laravel API, load the resource we need, render the page, then return the response. Since `asyncData()` is promise based, we can handle a failure or a 404 if a resource doesn't exist.

However, when compiling your front end to Universal, any static page (contact, home page, etc.) gets pre-rendered and packaged right away. If you have a lot of assets this will increase the app size dramatically. This is alright if you are using a web server that you control and you get the benefit of being able to be indexed by

search engines.

Let's contrast Universal to an SPA. When you compile your application to be an SPA, all of the loading of data comes AFTER the initial request. What do I mean by that? When I make a request to load ROAST as an SPA, I get a small HTML file with a `<nuxt/>` component and the javascript used to kick start the application. The `<nuxt/>` component does not have any data that can be indexed by search engines, nor does it change if I request any dynamic page. All of the content will be requested after NuxtJS has initialized and figured out where we are in the application to request the right route data. This happens so fast that you might not see a visual difference through the web browser. However, if you care about SEO in your app, web scrapers won't be able to pick up any data because on initial response, nothing has been returned. The benefit of this being a small package size.

So why choose two different modes? Well in a mobile application, you don't need SEO. It actually is detrimental to have an SSR app deployed to mobile since it yields larger package sizes, extra content, etc. That's why we want to use an SPA when we deploy to mobile. Since SEO is important for ROAST, we want SSR for web. Luckily with NuxtJS this is a breeze. You can configure the build instructions in the `nuxt.config.js` file, but you can also configure it through the command line when you build your application. In the next section we will set this up through the command line so we have a dedicated command to build our mobile application. In the last section we set the `webSrc` in the `capacitor.config.js` file to be `/dist`. Guess what? Both build commands export to the same directory so we won't have to worry about that! To speed up builds, we made a single command that prepares our front end to be deployed to mobile. Let's create this command and start compiling to mobile!

Speeding Up Builds With A Single Command

Let's create a command that will dramatically increase the speed at which we can build mobile applications. Instead of adjusting all sorts of variables every time we need to deploy to mobile, we can just run this command and it will prepare our application for mobile. There are a few things we need to set up first.

Step 1: Create Mobile Environment Variables

Remember way back when we installed and configured NuxtJS, we added a `dotenv` package? This is where it will come in handy! We will create a set of mobile environment variables for you to use to package your mobile application. This is different than how we would use environment variables for a web front end since all of the development, staging, and production testing will happen on your local machine or deployed to your device.

Usually, when using environment variables, you don't want the choice of environment (kind of the point). However, with mobile applications, you have to package the production environment locally before distributing to the app stores. Because of this, I create two new environment variable files:

```
.env.mobile-development  
.env.mobile-production
```

You can create as many as you want, but these are the two I start with. In these two files I define the following:

```
BASE_URL={FRONTEND_BASE_URL}  
API_BASE_URL={BACKEND_BASE_URL}
```

Now when we compile our application for mobile and use these with our `process.ENV.{VARIABLE}` these will be defined throughout our application. So how do we choose which environment variables to load? Check out Step 2.

Step 2: Define Which Environment Variables To Load

So we are building for 2 different platforms, web and mobile. Both of these platforms require different environment variables. With mobile, there are actually 2 separate environments we are managing with our environment variables, development and production. We need to find a way to tell NuxtJS when building our front end, which set of variables to choose.

Now we haven't gotten to actually creating our command, but a little foreshadowing is that we are going to create a global variable in our command called `ROAST_PLATFORM` and set it to `mobile`. We are also going to add a flag for what environment we will be building. To prepare for this, let's open up the `nuxt.config.js` file and add the following to the top before the settings module:

```
require('dotenv').config({
  path: process.env.ROAST_PLATFORM == 'mobile'
    ? '.env.mobile-'+
  (process.argv[process.argv.length - 1].replace('--', '') )
    : '.env'
})
```

What this does is loads the dotenv module and defines which path it should load the environment variables from. How we define our path is as follows. If the `ROAST_PLATFORM` variable is set to '`mobile`', then we look for the environment variables for the mobile platform we are loading. This is determined based on the flag we pass along to the command which will be `--production` or `--development`. When we create our build command, we will account for this to be passed along.

Now when we run our build command, we will load the mobile environment variables if we set the `ROAST_PLATFORM` to mobile. Otherwise, we load our web environment variables `.env` file.

It's time to actually build our command now!

Step 3: Creating our Mobile Build Command

Let's get that command registered! We have everything set up to customize our front end build for mobile.

The first step is to open the package.json in the root of your NuxtJS install. You will see the scripts key. Within that key you should recognize the names of the commands we've been running to build our front end. Let's add the following to that object:

```
"scripts": {  
    ...  
    "mobile": "ROAST_PLATFORM=mobile nuxt generate --spa --  
    fail-on-error",  
    ...  
}
```

That's it! We now have a mobile build command! Let's step through it though so it makes more sense.

The first part of the command is to set a process variable `ROAST_PLATFORM=mobile`. This is extremely important because we use this variable to determine whether we are building for mobile or not. It will now be accessible as an environment variable as:

```
process.env.ROAST_PLATFORM
```

In the next section, we will show how we can access this cleanly within our components and pages.

After we define the variable we run the command:

```
nuxt generate --spa --fail-on-error
```

What this does is run the command to build our front end and update the dist directory. However, the two flags, `spa` and `fail-on-error` are extremely important. The `--spa` flag overrides our `nuxt.config.js` mode setting and builds the app as an SPA. Don't worry, when we deploy to web, we don't have a flag so it will deploy as a universal application. The `--fail-on-error` means that we should be alerted of any build errors and fail the build. When deploying to mobile app stores, this is extremely important. If the build were to fail or have an error, we should not overlook that! I mean we shouldn't for web either, but it's easier to re-deploy to web than it is to deploy to a mobile App Store if there's an issue.

So how do we run this command? In the terminal run the following:

```
npm run mobile
```

That will build our front end mobile app! But what about the specific mobile environment variables? We need to define the source when we run our command. For development, we run:

```
npm run mobile -- --development
```

And when we are ready for production:

```
npm run mobile -- --production
```

Definitely make sure you run the production version BEFORE deploying to the App Store or your app will be hitting the wrong endpoints and nothing will work!

Let's take a look at how we can determine which environment we are using within our pages and other components. Then we will step through the super simple commands to build the actual mobile app environments with CapacitorJS.

Step 4: Determining Environment in Vue Components

There are times where you may want to alter your front end if you are in a mobile app vs on the web. A perfect example is removing a footer since most native mobile applications do not have those. However, you want to use the same code base, which is the entire point!

Here's how I approach this. We already have our deployment platform based on the command we run. Remember we have our `ROAST_PLATFORM` set to mobile when we run our build command? Let's make sure we can access that within our front end code. Open up your `nuxt.config.js` file and add a new key to the config that looks like this:

```
env: {  
    mobile: process.env.ROAST_PLATFORM == 'mobile' ? true :  
    false  
},
```

What this does is set an environment variable within our NuxtJS config that we can use inside of our page components. Now, when we need to determine which platform is running in our front end, we have a boolean for mobile. We can use it like this:

```
if( process.env.mobile ){  
  
}
```

Now you can continue using the same code base and determine if you need to make slight changes for your mobile application!

Step 5: Building Process for Each Platform

So we've got our front end built, now we need to update the iOS and Android

applications with the newly built code. Luckily CapacitorJS makes this a breeze! There is only one command that we need to run that takes care of copying over the entire dist directory to each platform's native codebase:

```
npx cap copy
```

Now the entire dist directory that was just built is copied to whatever platforms you are using!

From here, you will need to set up your deployment for iOS and Android using the specific keys and developer credentials for each platform. This is outside the scope of this book, but now you have an app that's ready to be deployed!

CapacitorJS also provides simple commands to open the native IDEs for each platform for you to edit. To open the Xcode project, simply run:

```
npx cap open ios
```

And the Android Studio project:

```
npx cap open android
```

To sum up the flow of how I manage deploying our front end to mobile, I'd follow this workflow:

1. Make changes to your front end the same way you would for web.
2. Run `npm run mobile -- --production`
3. Run `npx cap copy`
4. Perform deployment process for whatever platform you wish

We are now using the same code for web and mobile! There are a few things we will need to tweak though before a full mobile deployment is ready along with a few customizations that you can implement.

Token-Based Authentication with Laravel Sanctum

When using Laravel Sanctum there are two approaches. First is the Cookie-based approach that we implemented for our single page application. This allows the API to store a cookie with our SPA since they reside on the same top level domain to keep track of authentication.

The second approach is to store a token that gets returned when we authenticate our user. This is very similar to any oAuth tokens you might be familiar with when working on implementing a social log in system. The difference being we need to have a place to store this authentication token safely and securely. We also need to use a different NuxtJS authorization method when on the mobile app. Since we package the mobile app and distribute it to iOS and Android, we don't have proper security to store a cookie for authentication. Don't worry, this will make more sense as we dive in. Let's first start by installing the Capacitor Secure Storage Plugin.

Step 1: Install Capacitor Secure Storage Plugin

The first step we need to take is to install the [Capacitor Secure Storage Plugin](#). This will allow us to store strings securely using either the iOS keychain or Android Secure Storage depending on the platform. The reason we need to do this is so we can store our token upon authentication with our mobile app. It will also allow us to retrieve the token after the user has closed the application so they can stay logged in much longer.

To install the Capacitor Secure Storage Plugin, add the following package to your project:

```
npm install capacitor-secure-storage-plugin
```

This will install the plugin and we can use it within our app. The only step we have

to do right off the bat is make sure that we alert [Android of the plugin](#). This was a little confusing because the file you have to modify is nested pretty deep. Open up the `/android/app/src/main/java/{your}/{reverse_domain_name}/MainActivity.java` file and add the following to the header:

```
import com.whitestein.securestorage.SecureStoragePlugin;
```

Finally, within the `onCreate()` method, add the `SecureStoragePlugin.class`:

```
public class MainActivity extends BridgeActivity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        // Initializes the Bridge  
        this.init(savedInstanceState, new ArrayList<Class<?  
extends Plugin>>() {{  
            // Additional plugins you've installed go here  
            // Ex: add(TotallyAwesomePlugin.class);  
            add(SecureStoragePlugin.class);  
        }};  
    }  
}
```

Now you have the secure storage plugin ready to work with Android! Upon install it works great with iOS. Next up, let's make a quick NuxtJS plugin to make sure it works seamlessly in our project!

Step 2: Create NuxtJS Plugin for Secure Storage

Like with other Capacitor plugins, we make a nice NuxtJS plugin to include it only where necessary. This allows us only to include the plugin globally if needed and avoid any issues with Server Side Rendering (SSR). We will touch on this more a little later in [Using Native Phone Features In Your App](#). For now, we need to do a quick overview so we can authenticate users on mobile.

The first step is to create a file in the `/plugins` directory named `secureStorage.js` and add the following code:

```
import 'capacitor-secure-storage-plugin';
import { Plugins } from '@capacitor/core';

const { SecureStoragePlugin } = Plugins;

export default function ({ app }, inject){
    inject( 'CapacitorSecureStorage',
    SecureStoragePlugin );
}
```

This imports the secure storage plugin and injects it into NuxtJS so we can use it throughout our application.

Finally, we just need to register the plugin within the `nuxt.config.js` file by adding the following to the `plugins` array:

```
plugins: [
    ...
    {
        src: '~/plugins/secureStorage.js',
        mode: 'client'
    }
],
```

Now when we build the front end of our application, the secure storage plugin will be included and we can use it in our components.

Step 3: Register Local Authentication Strategy

One of the most beautiful features of the Nuxt Auth Module is you can register multiple authentication strategies. While we use the `laravelSanctum` strategy for

our web app, we will actually be using the [local strategy for mobile](#). Even though the mobile strategy is still using Laravel Sanctum, we are using tokens which the local strategy is optimized for and it works great.

The only step we have to take is to add the local strategy to our auth strategies object in the `nuxt.config.js`:

```
auth: {
  strategies: {
    'local': {
      user: {
        property: false
      },
      token: {
        property: 'token'
      },
      endpoints: {
        login: {
          url: '/api/v1/login',
          method: 'post',
          withCredentials: true,
          headers: {
            'Content-Type': 'application/json'
          },
        },
        logout: {
          url: '/api/v1/logout',
          method: 'post',
        },
        user: {
          url: '/api/v1/user',
          method: 'get'
        },
      },
    },
  },
},...
```

At initial glance, this may look fairly complicated, but it's pretty straight forward.

The first index is local meaning we are using the local authentication method provided by the NuxtJS auth module. Within that key, we are defining every route and feature we need to ensure we can successfully authenticate a user.

The first thing we define is the user key which sets property to false. What this means is that when we make the request to return the user at `/api/v1/user`, the entire response will be the JSON representation of the user. Sometimes when you make a request to load the authenticated user it'd come back looking something like:

```
{  
  user: {}  
}
```

Our response, the entire JSON object returned will represent the user. Next, we define the token and the property used to store the token. What this means is when we get a response from the API, the token will be under the key token like this:

```
{  
  token: 'token_here'  
}
```

Finally, we define the 3 endpoints we need for a full circle authentication of the user. The first is the login endpoint. This is what we hit when we went to send credentials to the API to authenticate a user. This will return a token if the credentials are successful. We just have to make sure we are defining the method to be `POST`, including the credentials, and making sure the content type is `JSON`.

Next up we have the logout URL. We just need to define the endpoint and the HTTP method used to access it. In this case, it's `POST` as well. When we call the logout method, we will send a `POST` request to the URL provided.

Finally, we have the user endpoint. This is what the NuxtJS auth module will use to grab the user. This should look very similar to our Laravel Sanctum config with just a few additional pieces of data. Now we just have to configure our login

component to use this method when we authenticate on mobile.

Step 4: Modify Login.vue Component to Store Mobile Token Securely

This step is probably the most crucial. We want to use the same log in component, but on mobile authenticate the user with the local method instead of the `laravelSanctum` method. This will allow us to store a token instead of a cookie and utilize our secure storage plugin.

Let's open up the `components/auth/Login.vue` component and find the `login()` method. Everything leading up to this method will be the same, we just have to put a check in there to make it work with our local authentication strategy. Make sure the `login()` method looks like:

```
login(){
    if( this.validateLogin() ){
        this.processing = true;

        let email = document.getElementById('login-
email').value;
        let password = document.getElementById('login-
password').value;

        // Ensure the right method is being used for auth
        // depending
        // on platform
        if( process.env.mobile ){
            this.$auth.loginWith( 'local', { data: {
                email: email,
                password: password,
                device_name: 'Mobile App'
            } } )
            .then( function( response ){
                this.
$CapacitorSecureStorage.set({ key: 'roastAuthToken', value:
```

```
response.data.token })
        .then( function( success ){
document.getElementById('login-email').value = '';
document.getElementById('login-password').value = '';
this.show =
false;
this.processing = false;
this.runPreAuthAction();
EventBus.$emit('roast-login');
}).bind(this) )
.catch( function( error ){
console.log( error );
});
}.bind(this))
.catch( function( error ){
this.processing = false;
this.validations.invalidLogin.valid =
false;
this.validations.invalidLogin.message = 'Invalid
credentials, please try again!';
this.validations.email.valid =
false;
this.validations.password.valid =
false;
}.bind(this));
}else{
}
```

```
        this.$auth.loginWith( 'laravelSanctum', { data:
{
            email: email,
            password: password
} } )
.then( function(){
    document.getElementById('login-
email').value = '';
    document.getElementById('login-
password').value = '';
    this.show = false;
    this.processing = false;
    this.runPreAuthAction();
    EventBus.$emit('roast-login');
}.bind(this))
.catch( function( error ){
    this.processing = false;

this.validations.invalidLogin.valid = false;

this.validations.invalidLogin.message = 'Invalid
credentials, please try again!';
this.validations.email.valid =
false;
this.validations.password.valid =
false;
}.bind(this));
}

},
},
```

Now this may look like a lot, but in reality we are only going to focus on half of it! We've already implemented the `laravelSanctum` authentication system so we just need to determine which one to use. To do this, look at this line:

```
if( process.env.mobile ){
```

Remember, when we build our app, we set the environment to be mobile. When it's packaged, this will be set to mobile and we will be able to check based off of that. So now, if we are on mobile, we run the following method:

```
this.$auth.loginWith( 'local', { data: {
    email: email,
    password: password,
    device_name: 'Mobile App'
} } )
.then( function( response ){
    this.$CapacitorSecureStorage.set({ key:
'roastAuthToken', value: response.data.token })
    .then( function( success ){
        document.getElementById('login-
email').value = '';
        document.getElementById('login-
password').value = '';
        this.show = false;
        this.processing = false;
        this.runPreAuthAction();
        EventBus.$emit('roast-login');
    }).bind(this) )
    .catch( function( error ){
        console.log( error );
    });
}

.bind(this))
.catch( function( error ){
    this.processing = false;
    this.validations.invalidLogin.valid = false;
    this.validations.invalidLogin.message = 'Invalid
credentials, please try again!';
```

```
this.validations.email.valid = false;  
this.validations.password.valid = false;  
}.bind(this));
```

Let's break this down. First, we make a call to `loginWith` and pass the `local` strategy. We then submit the form data similar to Laravel Sanctum except we also include a `device_name`. This will trigger our API to send back a token instead of a cookie since we can check to see if the field is set. We also store the device name with the token so we can see where the token is registered on the API and if we need to trim it at all.

On success, is where we add in an extra step. You will see the `this.$CapacitorSecureStorage.set()` method being called. This sets the token returned from the server by a key called `roastAuthToken` in the secure storage system on the phone. This is also a promise based function so upon success of storage, we hide the loader, reset the inputs and then handle the successful authentication. Now that we have our token in secure storage we can load it up when the user opens the app and store it safely.

If the log in attempt fails, we handle it the same way we would with Laravel Sanctum by throwing an error and displaying it to the user.

We have two more steps on the front end, then one quick modification to the API side and we have our token based authentication system set up correctly!

Step 4: Load Token Upon App Initialization

So now that we have our token stored in local storage we have to utilize it when appropriate. This means that after the user has closed the app and wants to re-open it, we have to grab the token from secure storage, set it, and authenticate the user carrying on their session.

The first step is to open up the `Layouts/App.vue` file. Remember, we use this for configuring the shell of the app and loading any data we need. In this file, we will add the following to the `mounted()` lifecycle hook:

```
if( process.env.mobile ){
    let token = this.$CapacitorSecureStorage.get({key:
'roastAuthToken'})
    .then(function( response ){
        this.configureUserFromToken( response.value );
    }.bind(this))
    .catch(error => {

});
}
```

What this does, is when the app is mounted, we check to see if we are in the mobile app using `process.env.mobile`. If we are, then we load the token from the secure storage by the key we saved it as (`roastAuthToken`). Once again the `get()` method is promise based so if there is a token that exists with that key, the promise gets resolved successfully and we can access the token. If not, the promise fails gracefully and we don't try to load a user.

In the example above, we call a `this.configureUserFromToken()` method and pass it the value returned from the secure storage. To implement this method, add the following to your methods:

```
configureUserFromToken( token ){
    this.$auth.setUserToken( token );
    this.$axios.$get('/api/v1/user', { headers: {
        'Authorization': 'Bearer '+token
}}).then( function( response ){
        this.$auth.setUser( response );
    }.bind(this));
}
```

This method takes a parameter of token and will only get called if there is a token. Within the method, we first set the user token using the NuxtJS auth module and

calling the `setUserToken()` method. This will set the token in the module so it will be used on all future requests.

Next, we call the user endpoint through the NuxtJS axios plugin and add the token to the header. If the token is valid, the request will succeed and we will set the authentication module with the result using the `setUser()` method. Now our user will continue their session and our authentication module has been configured correctly. All future requests will be authenticated from the token that was stored in secure storage!

Finally, we just have to clear the token when the user logs out.

Step 5: Clear the Token Upon Log Out

This is the last step we need to perform on the front end. When the user chooses to log out of the application, the Nuxt auth module will destroy its session, but we also have to clear the token.

There are two places within our app that the logout functionality is called. Both have the same method, so I'll go through one here. If you are looking for an example of this in the source code, check out `components/global/Header/AppHeader.vue`. The method is `logout()` and it looks like:

```
async logout(){
    await this.$auth.logout();
    this.$CapacitorSecureStorage.remove({key:
    'roastAuthToken'});
},
```

What this does is first call the `logout` method on the auth module, clearing the session. Next, we call the `remove()` method on the Capacitor Secure Storage plugin to remove the key from secure storage.

Everything is now set up on the frontend of our app! We just need to tweak the authentication route to return the token when called from a mobile device.

Step 6: Modify API Authentication Method to Return Token on Mobile

In Step 3, we added the local strategy to submit the authentication to the `/api/v1/login` endpoint. To make this happen, we have to implement the `/api/v1/login` endpoint. First, we should open up our `routes/api.php` file and add the following endpoint:

```
Route::group(['prefix' => 'v1'], function(){
    ...
    Route::post('/login', 'Auth\AuthController@login');
});
```

Next, we need to adjust our `Auth\AuthController.php` login method to look like this:

```
public function login( Request $request ){
    // If the request has 'device_name' it is coming from
    // mobile, so we return a token
    if( $request->has('device_name') ){
        // Ensures the login request is valid
        $request->validate([
            'email' => 'required|email',
            'password' => 'required',
            'device_name' => 'required'
        ]);

        // Grab the user that matches the email and check
        // to see if the
        // passwords match.
        $user = User::where('email', $request->email)->first();

        // If there is no user, or the password is
        // incorrect, return a 403 error.
        if ( ! $user || ! Hash::check($request->password,
            $user->password) ) {
```

```
        return response()->json([
            'error' => 'invalid_credentials'
        ], 403);
    }

    // Return the token for the user to the mobile app.
    return ['token' => $user->createToken($request-
>device_name)->plainTextToken ];
} else{
    // Web login code
}
}
```

Let's break this down real quick. What we do is check to see if the request has a device_name. If it does, we know they are coming from the mobile app. Next, we validate the request ensuring that they have an `email`, `password`, and `device_name` (which they should have, just an extra layer at this point). If the validation passes, we grab the user who matches the email, then we check to make sure that the user and email combination pass. If they do not, we return a `403` with invalid credentials. However, if they do pass, we call the `createToken()` method on the user and the `plainTextToken`. We return this under the `token` key. This is what we store to authenticate our user on mobile!

Now that we have this set up, our authentication will work through Laravel Sanctum on both Web and Mobile devices!

Using Native Phone Features In Your App

One of the biggest worries developers have about building an application using web code and packaging it as a native app is they won't be able to access native hardware or features. That's totally not the case! Unless you are building a graphics intense game, you can achieve amazing native results through a web based front end application with bridges to native phone features. I'll show you how I set up the front end to handle access to these resources.

CapacitorJS as a Bridge

Besides packaging your application and preparing it for mobile deployment, CapacitorJS also acts a bridge between your front end code and the native mobile hardware. This allows you to create/install plugins that allow access to native mobile features through Javascript. For example, say that you needed to access the camera on your phone through web code. You'd use a CapacitorJS plugin that would allow you to trigger native camera events through an interface that was implemented for both iOS and Android. This way you'd have to write the code once in Javascript and each platform would operate the same way.

CapacitorJS comes with a ton of [APIs available](#). We use a few in ROAST. One example of what we do in ROAST is use the current location of the user (if permission is granted) to find coffee shops in their area. In this section, we step through the process of setting up your front end to use these APIs.

Step 1: Organize CapacitorJS Plugins Within NuxtJS

The way that I like to approach this is to organize the CapacitorJS plugins within NuxtJS is to add their implementation globally through a simple NuxtJS plugin. This way we can use the plugins when needed and we can determine if they are included in SSR or not (which is SUPER important).

If you were just building an SPA, you'd be able to just include the CapacitorJS plugin on top of your Vue component when needed. However, since we are doing SSR for web and aren't using the plugins on web, but are using them on mobile, we need to come up with a structure that works for both so we can reuse the same code.

Let's go back to our geolocation example. On the mobile app, we will want to specifically use the phone's hardware Geolocation if the user gives us permission. This will require us to use the [CapacitorJS Geolocation plugin](#). Let's make a NuxtJS plugin in the `/plugins` directory named `geolocation.js` and add the following code:

```
import { Plugins } from '@capacitor/core';

const { Geolocation } = Plugins;

export default function( { app }, inject ){
  inject( 'CapacitorGeolocation', Geolocation );
}
```

What this does is it imports the plugins from the core of CapacitorJS. We then decouple the Geolocation plugin so we can inject it into our NuxtJS instance. This is very similar to how we built our Axios modules. Finally, we export a function and inject the actual Geolocation plugin from CapacitorJS into a NuxtJS plugin called `CapacitorGeolocation`.

This seems a little bit cumbersome, right? Well we will have to do it this way and we will explain why in the next step.

Step 2: Register Your Plugin With NuxtJS

Let's jump into the `nuxt.config.js` file and navigate to the `plugins` array. You should see in that array, the registration for our `roastAPI.js` plugin that we built when we abstracted API routes. In that array, add the following:

```
plugins: [
  '~/plugins/roastAPI.js',
  {
    src: '~/plugins/geolocation.js',
    mode: 'client'
  }
]
```

This registers our NuxtJS Geolocation plugin that uses the CapacitorJS Geolocation plugin within our front end. However, notice how we have an object that registers this plugin? This is so we can use the `mode: 'client'`. This is EXTREMELY important.

Since we are doing SSR with our front end for web deployment, we compile down all of the plugins and content before rendering the page. Well, there are pieces of these CapacitorJS plugins that are written that rely on the web browser being present. When we are doing SSR, this will throw an error since we are trying to pre-render the application WITHOUT a browser being present. You will get a string of undefined variables an APIs.

This is why we abstract all CapacitorJS plugins into a NuxtJS plugin. When we use the `mode: 'client'` key on the registration of the plugin with the front end, we are telling NuxtJS to ignore the plugin when performing a Server Side Render. The plugin will then be included when deployed to the web.

Step 3: Using The NuxtJS + CapacitorJS Plugin

So how do you use the plugin in the code? Let's say we want to get the current location of the user after they approve that we can. We have our CapacitorJS plugin wrapped in a NuxtJS plugin, so all we need to do is call it! To get the location of the user, we'd add the following in our component:

```
async getCurrentPosition( context ) {
```

```
try {
    const coordinates = await this.$CapacitorGeolocation.getCurrentPosition();
    this.buildMap( coordinates.coords.latitude,
coordinates.coords.longitude, 14, true );
} catch(err) {
    // handle errors
}
},
```

The method to get the current position returns a promise, so we can wrap the method call in `async/await`. Within the method, we call `await this.`

`$CapacitorGeolocation.getCurrentPosition()`. This uses the `$CapacitorGeolocation` plugin we registered within NuxtJS and the `getCurrentPosition()` which is then called on the CapacitorJS plugin.

Now, you might want this call to only happen on mobile and not on web. CapacitorJS could have a similar web interface for the plugin calls, but for this example let's say it doesn't. Well, we already built in an environment variable that determines if we've compiled our frontend for mobile or not, so let's use it!

Say you load up a map component and you want to zoom on the current location if the user allows you to through the mobile app. On web, however, you want to go to a standard coordinate. You'd add the following to your `mounted()` lifecycle hook:

```
if( process.env.mobile ){
    this.getCurrentPosition();
} else{
    this.buildMap();
}
```

Now, if the user is using the mobile app, the component will call the CapacitorJS plugin to get the current position of the user, then call the build map with the results. Otherwise, if the user is on web, we just build the map and provide a

default for the latitude and longitude for the center of the map.

Following this structure will allow you to use the same code base for both web and mobile platforms, but also give you the flexibility for using more powerful tools on the device that supports them!

In the next section, we will go through Deep Linking which allows users coming from the web to navigate to the same page within the mobile application.

Deep-Linking: Open Your App From The Web

So what are deep links? Have you tapped a Spotify song that was sent in a message and it opened up the app with that song enabled? That's a deep link. I really like deep links as they make the transition to help guide users into your app a breeze and the flow feels so professional.

I'll show you how I set up the code to make this work within your application, however, there are some verifications you need to do with Apple to make sure they recognize your deep links. [CapacitorJS' documentation](#) does a wonderful job going through this set up.

Step 1: Make a Deep Link Plugin

The first thing we need to do is create a plugin in NuxtJS to handle the listening for deep linking. If you looked at CapacitorJS' documentation for Deep Links, they discuss binding to the `appUrlOpen` event on the `App` object. We need to make sure that this is set up right away when NuxtJS initializes. The best place to do that is through a plugin.

Fair warning, we will never actually directly call this plugin because it will only be used to listen to the binding of a user entering your application from a specified URL. To get the plugin created, add the `/plugins/deepLinks.js` file and add the following code:

```
import { Plugins } from '@capacitor/core';

const { App } = Plugins;

export default function ({ app }, inject){
  App.addListener('appUrlOpen', function( data ){
    const slug =
```

```
data.url.split("YOUR_TOP_LEVEL_DOMAIN").pop();  
  
    if( slug ){  
        app.router.push({  
            path: slug  
        });  
    }  
});  
}  
};
```

In reality, this is the entire code to make the Deep Links work! However, it's pretty dense, so let's step through it. The first thing we do is include the Plugins object from the core of capacitor, we then decouple the App from the plugins. This is what we want to bind our event listener to.

Next, we create our NuxtJS plugin function. The first parameter we decouple the NuxtJS app and pass it to our plugin. This allows us access to what NuxtJS provides, specifically access to the Vue Router.

So inside of our plugin, we add an event listener to the CapacitorJS app that listens for an appUrlOpen event. When the event is fired, it means that there is an incoming request to our application from outside. The first line of the handler method splits the incoming URL by the top level domain:

```
const slug = data.url.split("YOUR_TOP_LEVEL_DOMAIN").pop();
```

We don't want the top level domain, we just want the URL slug. Within our mobile app, we don't want to redirect to the absolute path or that'd be just to the website. We are using Vue Router to handle internal redirection, so we only want the relative aspect of the URL. We can get this by grabbing the end of the URL after the top level domain. This is saved to our slug variable.

Finally, we check to see if the slug is set. If we clicked on a link that goes to the index page of the application, then deep linking wouldn't really matter too much since it is essentially just opening your application. However, if we DO have a slug,

we want to send the application to the relative link within our application. This is done by pushing a route onto the Vue Router used by NuxtJS. This is a relative path so since it's in the scope of our application, it will take us to the right page in the app. All we have to do now is register our plugin with NuxtJS.

Step 2: Register our Deep Link Plugin

Like the other plugins we've created, we just have to register the plugin with NuxtJS. Once again, we only want this plugin available on the client side (when the web browser is available) since it's only used when we compile to mobile.

So open up your nuxt.config.js file, find the plugins array, and addd the following:

```
{
  src: '~/plugins/deepLinks.js',
  mode: 'client'
},
```

That's all you need to do from a programming perspective to set up deep links within your application! Once you get approved by Apple and have everything registered according to the documentation, Deep Links will work within your app.

This concludes the guts of setting up your NuxtJS front end to work with CapacitorJS and package to a native mobile application! From here on out, any new features that require an app update, we will mention how it works not only on web, but also in mobile. Implementing Social Logins is next, and that will require a few special scenarios that need to be handled through Web and Mobile!

IMPLEMENTING SOCIAL LOGINS

Installing and Configuring Laravel Socialite

Social auth is tricky to begin with. Building a Social Authentication system through an SPA with authentication back to our own API is a whole new challenge. The most important piece of the puzzle is the end game is to be authenticated with OUR API, not the 3rd party social auth provider's API.

How this will work is we will use 3rd party providers to receive an authentication code in our SPA. We will then send the authentication code to our API, and authenticate the user and get an access token for the provider. When we receive a token we will be using that token to either register a new user on our API or authenticate an existing user.

We aren't receiving any data or requesting data from the 3rd party provider beyond the name and email provided in the response. Because of these complexities and the way we want to re-use code we are not initially using the NuxtJS auth module. We have to make the requests to social providers through our own code and then we use the NuxtJS auth module to perform the actual authentication with our API and Laravel Sanctum.

The process is quite the dance, but it follows the same pattern across each provider so once you implement it the first time, you can re-use it for other platforms. The first part of our journey is to install Laravel Socialite.

If you've worked with social authentication in the past, you know how tricky and finicky it can be. I'll be up front, it's even more tricky and finicky setting up the process with an SPA + API. However, once you get your authorization callbacks working, you can add whatever platforms you want that support the same oAuth platform standards you are using.

For this section, we are going to use a variety of different packages. First, we will be using [Laravel Socialite](#). Laravel Socialite is a first party package, similar to

Laravel Sanctum, that takes care of the authentication with social platforms. It takes the pain out of some of the callback nightmares that arise with social auth. However, since we are designing our app as a Single Page Application, we won't be using the common flow where the callback hits a server. But there are a lot of other ways that Laravel Socialite will help us, including doing a stateless authentication.

We will also be adding initial requests to the social providers through a custom request. Upon response we will be using the NuxtJS auth module to authenticate in a modified way to our API using Laravel Sanctum.

And finally, for mobile authentication, we will be using the CapacitorJS with In App Browser so we can properly handle callbacks within our application and authenticate appropriately.

Fair warning, there are a lot of moving parts in this section. If you get lost at any point, feel free to reach out on <https://community.serversideup.net> and I can help along the way!

Step 1: Install Laravel Socialite

The first step of the whole process is to install the Laravel Socialite package. To do this, open up the Laravel API directory and run the following command:

```
composer require laravel/socialite
```

That's all you need to do for installing Laravel Socialite! Next we need to store the provider and make a change to our password field. Then we will be getting some credentials. Each provider has their own credentials set and we need a unique set to work with. After we have the credentials from our oAuth providers, we will be able to set our config and allow users to authenticate.

Step 2: Prepare Database for Social Authentication

In preparation for social authentication, we need to add 2 more fields to our users table. They are `provider` and `provider_id`. Each user that authenticates through a social provider will have a provider that matches to the name of the network and their ID on that network. To prepare the database, I created a simple migration:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('password')->nullable()->change();
    $table->string('provider')->after('permission')-
>nullable();
    $table->string('provider_id')->after('provider')-
>nullable();
});
```

Besides adding the two fields, I also made the password field `nullable`. This is because socially authenticated users will not have passwords. We will get their information (provider + provider_id) from their social account to log them in.

When you nullify the password field, MAKE SURE THAT username + password fields are validated and both fields are populated when a user submits an authentication attempt! The validator looks like:

```
$validator = Validator::make( $this->data, [
    'email' => 'required|email',
    'password' => 'required'
]);
```

Now we are still validating the password on a normal Sanctum authentication. When we finalize our social auth, we will be sending more parameters to check (like a code to exchange) which will determine the validity of the authentication. Even though we are still using Laravel Sanctum to do the authentication.

The next couple steps are unique to whatever platforms you wish to have users authenticate with. We will walk through the most popular choices, but in reality,

the end game is to get a `client_id` and a `secret_key`. This will allow you to authenticate with any oAuth 2 provider using the code we have.

Step 3: Get your Facebook Credentials

The first step that you need to take is to register your application over at <https://developers.facebook.com>. Once you have registered your application, you will be able to get your App ID, App Secret and register the proper callback URLs to make the process work.

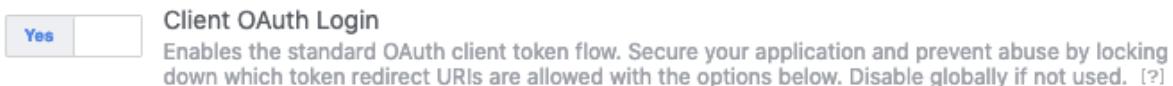
Enable Facebook Login

The first step of this process, after you have created your app, is to add the "Facebook Login" product. To do that, click the "plus" sign on the left side next to "Products":

PRODUCTS 

From there, search for "Facebook Login" and "Set Up". Now that Facebook Login is enabled, we need to configure all of the settings.

The first one we want to enable is "Client OAuth Login":



This allows us to secure our application with proper re-direct URIs and allow users to log in client side.

The second option we need to enable is "Web OAuth Login". This setting enables us to allow users to log in on the web side. Since we are doing mobile with CapacitorJS for iOS and Android and iOS, it might be tempting to walk through

their set up for those native platforms. However, since we are using CapacitorJS, we just need to configure the web side and we can re-use a lot of our code for both web and mobile.



Next, I enabled "Use Strict Mode for Redirect URIs". This means that the URIs for redirection that we set next need to exactly match. This is just another security measure that doesn't take long to set up, but helps enforce the security of your authentication.



Finally, we just need to set up our redirect URIs. When doing social auth, this is the URI that will handle the token exchange. On mobile, we will be redirecting to a web URI as well, but monitoring it in a different way. If you have a development or staging set up, you will need those URIs added as well. Right now, for ROAST, we have the following redirect URIs:

- <https://roastandbrew-frontend.521.test/auth/facebook/>
- <https://roastandbrew.coffee/auth/facebook/>
- <https://roastandbrew.coffee/auth/facebook/mobile/>

As you can see we have our development environment URIs in there as well. The major difference is there is a different endpoint for [/mobile](#). Don't worry, we will get to that and we will use most of the same code as well! We just want to handle the exchange slightly different.

Also, when we build our pages, they will be generic so they will work with any OAuth provider.

Obtain Client ID and Secret

This is what we are looking for with all of our OAuth provider solutions. The client id can be stored in the frontend javascript code, but the app secret should NEVER be stored publicly. This becomes a challenge when working with an SPA + API and we will get to that in the next step.

For now, you just need to obtain these keys. In Facebook's developer portal, you need to visit [Settings → Basic](#).

The keys you are looking for are named slightly different. The [App ID](#) is the [Client ID](#) we are looking for and the [App Secret](#) is the [Secret](#) key.

This is all we need from Facebook for now. We will walk through the set up for Google and Twitter next to obtain the keys we need. Then we will configure Laravel Socialite!

Step 4: Configure Google Authentication

Coming Soon! Google updated their auth security to use PKCE and Laravel Socialite is pending an update to support this. <https://github.com/laravel/socialite/issues/483>

Step 5: Configure Twitter Authentication

Coming Soon! Twitter is still using oAuth 1.0 which doesn't support stateless authentication. We will release this chapter separately.

Step 5: Set up Laravel Socialite

Now that we have our keys, let's configure Laravel Socialite. This is done on the API side. In the next section we will set up the SPA side.

The first step we need to take is to set up our `.env` file with the credentials that we got from each platform. To do this open the `.env` and add the following fields:

```
FACEBOOK_CLIENT_ID=XXXXXX  
FACEBOOK_CLIENT_SECRET=XXXXXX
```

```
GOOGLE_CLIENT_ID=XXXXXX  
GOOGLE_CLIENT_SECRET=XXXXXX
```

```
TWITTER_CLIENT_ID=XXXXXX  
TWITTER_CLIENT_SECRET=XXXXXX
```

Now we can use our credentials within our API when we make the authentication call.

To ensure Laravel Socialite is configured correctly, we need to add a few fields to the `/config/services.php` file:

```
'facebook' => [  
    'client_id' => env('FACEBOOK_CLIENT_ID'),  
    'client_secret' => env('FACEBOOK_CLIENT_SECRET')  
,  
  
'google' => [  
    'client_id' => env('GOOGLE_CLIENT_ID'),  
    'client_secret' => env('GOOGLE_CLIENT_SECRET')  
,
```

```
'twitter' => [
    'client_id' => env('TWITTER_CLIENT_ID'),
    'client_secret' => env('TWITTER_CLIENT_SECRET')
]
```

One thing to note, is if you read the Laravel Socialite documentation, they want you to configure your `redirect_uri` in this file. We are handling this a bit differently in our set up and will be setting this dynamically. Once we exchange the code for the token, we will step through how this gets set.

With these fields configured, this is all we have to do for now! We will come back to making the actual authentication request AFTER we set up our SPA.

Allowing logins from Facebook, Google, and Twitter

Now that we have Laravel Socialite configured, let's jump over to our SPA. This is where we start the 3-legged oAuth process which is a dance between 3 different pieces of our application, the SPA, the API, and the social provider.

For this section we are NOT going to use the NuxtJS auth plugin's default social login schemes. This is due to the complexity of the situation and there's no easy way to make it work together. However, we will be using our `sanctum` and `local` methods we already have configured once we have everything we need.

Step 1: Add the Client IDs to NuxtJS .env

Similar to how we configured Laravel Socialite, we need the app keys in the `.env` file so we can use them where we need. Since we are on the `SPA` side, we need to add the following keys to your `.env` file:

```
FACEBOOK_CLIENT_ID=XXXXXX  
GOOGLE_CLIENT_ID=XXXXXX  
TWITTER_CLIENT_ID=XXXXXX
```

MAKE SURE YOU ONLY ADD THE CLIENT ID!!! The secret key would be exposed if we added it here and that would be a major security vulnerability! Next, let's add the functionality to our login modal.

Step 2: Add Social Auth to Login Modal

The majority of our front end code will be in our Login modal. This is where we will provide the link to allow for social logins, build the URL to add our keys and send the user to the provider they need. Let's start simple by opening up the file you

have to prompt for authentication, for us it's `Login.vue`, and start small by adding our buttons:

```
<button type="button" v-on:click="signInWith('facebook')">  
    Sign in with Facebook  
</button>  
  
<button type="button" v-on:click="signInWith('google')">  
    Sign in with Google  
</button>  
  
<button type="button" v-on:click="signInWith('twitter')">  
    Sign in with Twitter  
</button>
```

Notice, these are all buttons at this point and not links. This is so we can handle authentication on mobile (which is more complex) and build our links dynamically.

Next, let's add the method `signInWith(platform)` to our methods:

```
signInWith( platform ){  
    if( process.env.mobile ){  
        this.loginWithSocialMobile( platform );  
    }else{  
        this.loginWithSocial( platform );  
        this.show = false;  
    }  
},
```

Right away, we can kind of see how we are beginning to route our users through the process. If they are on mobile (similar to the check we do on how we authenticate with UN + PW), we perform a different method than if they are on the web. For now, let's focus on the `else` part of the statement which implies the user is on the web.

We pass the functionality down to our `loginWithSocial(platform)` method and hide the login modal. This method will redirect the user to the authentication page for their desired platform. Let's take a look:

```
loginWithSocial( platform ){
  let url = this.buildSocialAuthURL( platform );
  window.location = url;
},
```

and while we are at it, let's look at the `buildSocialAuthURL(platform)` method as well:

```
buildSocialAuthURL( platform ){
  let url = '';

  switch( platform ){
    case 'facebook':
      let scope = ['public_profile', 'email'];

      url += 'https://www.facebook.com/v8.0/dialog/
oauth?';
      url +=
'client_id=' + process.env.FACEBOOK_CLIENT_ID;
      url +=
'&redirect_uri=' + encodeURIComponent( process.env.BASE_URL +
/auth/facebook' + (process.env.mobile ? '/mobile' : '') )
      url += '&response_type=code';
      url += '&scope=' + scope.join(',');
      url += '&state=' + this.createSocialAuthState()
      break;
  }

  return url;
},
```

The `buildSocialAuthURL` is the bread and butter of this process and will be used on both web and mobile. What this does is builds the login request URL for our application depending upon the platform that was chosen. In the example above we have Facebook implemented.

**As of November 10, 2020 Google & Twitter had a few outstanding bugs that didn't allow us to fully complete this section using their platforms. When those get fixed and we get the proper methods in place, we will release them in a later update. Any other oAuth provider can be handled the same way as Facebook if you account for their customizations in URL.*

What we do is construct a URL that goes in the following order. First, we build the base URL off of what we need to send the user to on Facebook. Then we add the `client_id`. This is the environment variable we added in Step 1 and is used to identify our app.

Next, we build the `redirect_uri` which encodes the current URI of our SPA which is also an environment variable `process.env.BASE_URL` and the path of `/auth/facebook` (which is a page we will be adding in the next section). Other social providers will have the same callback, just with their slug in the URL. This URL will handle the response from the social provider and work with our API to ensure the code gets exchanged for an authenticated user.

The big thing to note about this is we append `/mobile` to the end of the URL if the environment is mobile. When we create the page that handles the call back, we want it to send the authorization code to the API to exchange for a user and a token. With mobile, we want to listen to the callback but not actually send the authorization code through the browser that made the request. Confused? Don't worry, I'll explain it more in step 3 and 4.

As hinted at before, we won't be actually requesting the access token in this request. When requesting an access token, we need to provide the `secret` key. If that's stored in our SPA in javascript, any app can impersonate our application and leave a gaping security vulnerability. We need to request the authorization code

which is through the `response_type=code` parameter. This will return the authorization code we can exchange on our API which can securely house our `secret` key.

Finally, we append the scopes we need for the user. Each platform will be different, so make sure you abide by the platform's documentation. This will prompt the user and allow them to give your app permission to use the data requested. We also add a `state` which is a random string that verifies the response.

Now that we have our URL built and returned, we call `window.location=` and set it to the URL we just created. That will prompt the user to authenticate with their social login and redirect once completed. The social provider will redirect the user with the appropriate code to the `redirect_uri` that was defined by the user. The authorization code will be stored in the `code` URI variable. We will use this in our next step.

Step 3: Create Social Auth Redirect Page

At this point, we need to handle the authorization code from our social provider. When we set our `redirect_uri` we set it to either `/auth/{provider}` or `/auth/{provider}/mobile` for mobile. We need to create these pages within NuxtJS so when the user is routed back to our SPA we can handle the response. Let's add the web callback first.

To do this, create a folder in your `/pages` directory called `/pages/auth/_provider` and within that folder add `index.vue`. This page will dynamically generate the routes to handle any response from the provider we are authenticating with.

In our page, let's add the following code:

```
<template>
```

```
<div id="social-auth" class="page flex-1 flex flex-col items-center justify-center p-10">
    
    <span v-text="error" class="text-red-700"></span>
</div>
</template>

<script>

export default {
    middleware: 'guest',
    layout: 'App',

    data(){
        return {
            error: '',
        }
    },
}

mounted(){
    if( this.$route.query.code ){
        this.authenticate(
            this.$route.params.provider,
            process.env.BASE_URL + this.$route.path,
            this.$route.query.code,
            this.$route.query.state
        );
    }
},
methods: {
    authenticate( provider, redirectUri, code, state ){
        this.$auth.loginWith('laravelSanctum', {
            data: {
                'social': true,
```

```
        'mobile': false,
        'provider': provider,
        'redirect_uri': redirectUri,
        'code': code,
        'state': state
    }
}).then( function( response ){
    this.$router.push({
        path: '/'
    });
    EventBus.$emit('roast-login');
}.bind(this))
.catch( function( error ){
    this.error = 'There was an error in
authentication, please try again!';
}.bind(this));
}
}
</script>
```

Since it's a relatively small page, that's it! We really don't need to show anything because we want the user to be authenticated from this page. The template shows the loader and if there is an error. The guts of this page starts with the `mounted()` lifecycle hook.

When the user is re-directed from the social provider, we want to grab the `code` parameter from the URL. This will be our authorization code that we need to send to our API. in our `mounted()` hook, we will check to see if the route contains that code. If it does, then we need to perform our authorization.

Let's take a look at our `authenticate(provider, redirectUri, code, state)` method. This is the first glimpse we have of sending the request to our API and it should look relatively familiar to what we've done with basic

authentication. This is where we use the existing `laravelSanctum` authentication method, except we pass a few different parameters.

The part of the request we should pay the most attention to is:

```
data: {  
    'social': true,  
    'mobile': false,  
    'provider': provider,  
    'redirect_uri': redirectUri,  
    'code': code,  
    'state': state  
}
```

Instead of sending the username + password combo, we are sending a bunch of other data. When we switch to the API we will finalize this request and get the actual user. For now, let's explain what we are sending. The `social` flag means that we want to alert our API that this is coming from a social log in request. The `mobile` flag means we do not want to return a token, but want to use Laravel Sanctum in a session + cookie mode. We also send along the `provider` so we know which provider we are authenticating with. The main pieces of data we need to send are the `redirect_uri` & `code`.

The `redirect_uri` was what we used to send to the social provider that redirected us to this point. This needs to be sent along with the code exchange method. Most importantly, we need the `code`. This is the authorization code that gets generated allowing us to exchange it securely for a user and a token completing the oAuth process. Since this exchange requires a `secret` key we do this on the server side.

Now, the real magic comes into play and that's on the server side. Since we are using Sanctum over all, we need to end up with Sanctum managing the authentication. That's why we use the NuxtJS sanctum authentication, but send

the code and social parameters to perform the actual authentication. For now, this is where the line ends for SPA side. Upon success, we will redirect with a properly authenticated user home and upon error, display an error message.

One quick thing to note, on the API side we will be registering users as well so if it's a new user, this whole process will complete with a newly registered and authenticated user. Let's take care of the mobile side here really quick since it's a little more complicated, but ends at the same API endpoint.

Step 4: Implement Mobile Social Auth

So how the mobile authentication will work is slightly different. Unfortunately, we need to install a Cordova plugin in our mobile app. Luckily, they all play really nicely with CapacitorJS through a beautiful design. The plugin we need is the [cordova-plugin-inappbrowser](#). Why do we need this when CapacitorJS has a default browser plugin? Because unfortunately, you can't listen to the events you need through their browser plugin to perform a successful oAuth.

Before we install the plugins, let me explain how this will all work. A user downloads the mobile app from iOS or Android. They wish to authenticate with their social account, so they click the account button they want to authenticate with. This will open a new in app browser that is set to the authentication URL we dynamically create (see [buildSocialAuthURL\(platform \)](#) in Step 2). The in app browser allows the user to authenticate and then redirects back to the application. Now remember, we are redirecting to [/auth/{platform}/mobile](#) which is different. This page literally does nothing, but we will get there in a bit. Upon re-directing, our parent window which is our capacitor app facilitating this process is checking the URL of the in app browser window to see if it contains the [code](#) parameter. This means the authentication was successful and we have an authorization code! This is the point where we grab the entire URL similar to our [/auth/{platform}](#) page and send it off to our API to exchange it for a token and to register/authenticate the user.

The two questions I usually get when approaching the problem this way are: 1. Why two URLs? 2. Why do we need the Cordova In App Browser.

The answer to 1 is, since we are redirecting back, we don't want the in app browser to send the request to the API or the user will be authenticated through a Laravel Sanctum web cookie and in the in app browser. NOT THE APP. We just needed a valid registered endpoint to grab the auth code from.

Number 2 is because of the flexibility of the Cordova in app browser compared to capacitor's. Every redirection and page load triggers an event with the Cordova in app browser where it only triggers once with the CapacitorJS in app browser. We need the SECOND redirection after the user authenticates to grab the authorization code. Unfortunately, this can only be accomplished, as of this writing, using the Cordova in-app browser. If an update comes along, we will update this content and provide a tutorial on how to use the next best thing. For now, we are going to use the Cordova in app browser.

With that being said, let's install the Cordova in app browser in our NuxtJS SPA using the following command:

```
npm install cordova-plugin-inappbrowser
```

After it's installed, the in app browser will be ready to use through the global `cordova` variable present in our front end.

Jump back to Login.vue

This is really whatever component you are using to authenticate. For us, it's `Login.vue` modal. We want to look at the `signInWith(platform)` method:

```
signInWith( platform ){
  if( process.env.mobile ){
    this.loginWithSocialMobile( platform );
  }else{
    this.loginWithSocial( platform );
```

```
        this.show = false;  
    }  
,
```

In the initial `if` statement, we check to see if the user is in the mobile app or not. This time, we want to account for that they ARE IN THE MOBILE APP. So we have to add the following method:

```
loginWithSocialMobile( platform ){  
    let url = this.buildSocialAuthURL( platform );  
  
    let browser = cordova.InAppBrowser.open( url, '_blank',  
'location=yes' );  
  
    browser.addEventListener('loadstart', function(event){  
        let currentURL = new URL( event.url );  
  
        if( currentURL.searchParams.get('code') != null ){  
            this.authenticateSocialMobile( currentURL,  
platform );  
            browser.close();  
            this.show = false;  
        }  
    }.bind(this));  
,
```

This method is pretty dense, so let's break it down.

The first line:

```
let url = this.buildSocialAuthURL( platform );
```

builds the URL that we are using to open our in app browser to. This is the same URL we redirected the user to n web.

The next line is where the mobile side becomes different:

```
let browser = cordova.InAppBrowser.open( url, '_blank',
'location=yes' );
```

What this does is initialize a variable named `browser` and set it to a new Cordova in app browser that is open to the URL we created. We also set it to be `_blank` so it opens inside the application in a new window. This line of code is what will bring up the authentication window for your social provider.

The following chunk of code is what listens to the redirection from the social provider, back to your app:

```
browser.addEventListener('loadstart', function(event){
  let currentURL = new URL( event.url );

  if( currentURL.searchParams.get('code') != null ){
    this.authenticateSocialMobile( currentURL,
platform );
    browser.close();
    this.show = false;
  }
}.bind(this));
```

When a new URL loading starts, we grab the URL and look for a `code` parameter. The user won't have the ability to type in a URL since we're doing a small child window so the only time the `code` parameter is present is if the URL is redirected back to our app.

Within this code block, when there is a `code` parameter set, we run the `authenticateSocialMobile()` method and close the browser. We pass the `authenticateSocialMobile()` method the current URL which is the Javascript object representing the URL so it's easier to parse and the platform. An amazing resource explaining this URL object is: <https://dmitripavlutin.com/parse-url-javascript/>. Dmitri goes extremely in-depth and provides excellent examples.

Next up we have the actual implementation of the `authenticateSocialMobile()` method:

```
authenticateSocialMobile( url, provider ){
    let code = url.searchParams.get('code');
    let redirectUri = process.env.BASE_URL + url.pathname;
    let state = url.searchParams.get('state');

    this.$auth.loginWith( 'local', {
        data: {
            'social': true,
            'mobile': true,
            'device_name': 'Mobile App',
            'provider': provider,
            'redirect_uri': redirectUri,
            'code': code,
            'state': state
        }
    } )
    .then( function( response ){
        this.$CapacitorSecureStorage.set({ key:
'roastAuthToken', value: response.data.token })
        .then( function( success ){
            this.$router.push({
                path: '/'
            });
            EventBus.$emit('roast-login');
        }).bind(this) )
        .catch( function( error ){
            console.log( error );
        });
    }.bind(this))
    .catch( function( error ){
        this.error = 'There was an error in authentication,
please try again!';
    });
}
```

```
    }.bind(this));
}
```

This is extremely similar to the one we had in [Step 3](#) with 3 major differences. First, we are grabbing our parameters from the `URL` object instead of the actual URL. This URL object is set from the redirection URL provided in the in app browser. Second, we are using the `loginWith('local')` method which will set a `token` with Laravel Sanctum instead of a cookie. We will be getting to the API side of this next. Finally, the other difference is we set the `mobile` flag to `true`. We will get to that next in the API side as well.

Conclusion

So we are done with the SPA side, but we are only half of the way there! Now that we have our authorization code, we need to send that with Laravel Socialite to exchange it for an access token and a user. Remember, we don't really care about the access token as much as we do the user because the authentication will be through our API + SPA but validate the user through a third party.

Securely Exchanging Tokens With an OAuth Provider

Now we are right back to where we began, the API side! In this section, we will be securely exchanging tokens between the oAuth provider and the API.

What we have so far is the ability to allow the user to authenticate with a 3rd party and receive an authorization token. When we get that token on either web or mobile, we send it back to our authorization endpoint (`/api/v1/login` on mobile and `/login` on web).

Now we need to handle this request on the API side. Each request will contain an authorization code that we will use to exchange for a token and a user. We will also need to register the user within our app if they haven't logged in before. Let's get started!

Step 1: Account for Social Auth

Earlier on in the book, our authentication methods were relatively simple. With the addition of social authentication, I re-factored some of that code into `Services/User/Authentication.php` service. This allows us to break down the methods depending on how the user is logging in whether it's sanctum, social, or mobile sanctum.

If you look at `app\Http\Controllers\Auth\AuthController.php` and the `login()` method, you will see that we move all of the functionality into the `Authentication` service. The core of the functionality for email and password authentication is the exact same, just divided into smaller methods. However, let's take a look at the `socialAuthenticate()` method in the `Authentication.php` service.

Step 2: Build your Social Authentication Method

For us, this is in the `Authentication.php` service and is called `socialAuthenticate()`. It will handle any social authentication from any platform properly configured with Laravel Socialite (Twitter, Google, Facebook).

The method looks like this:

```
private function socialAuthenticate()
{
    if( isset( $this->data['redirect_uri'] ) ){
        $provider = $this->data['provider'];
        $redirectURI = $this->data['redirect_uri'];
        $state = $this->data['state'];

        config()->set( "services." . $provider . ".redirect",
$redirectURI );
    }

    $driver = Socialite::driver( $provider );
    $driver->stateless();

    $profile = $driver->user();
    $mobile = $this->data['mobile'];

    $manageSocialUser = new ManageSocialUser( $provider,
$profile );
    $socialUser = $manageSocialUser->authenticate();

    if( $socialUser ){
        if( $mobile ){
            return [
                'token' => $socialUser->createToken( $this-
>data['device_name'] )->plainTextToken
            ];
        }
    }
}
```

```
        }else{
            return response()->json( $socialUser );
        }
    }else{
        return response()->json(
            [
                'error' => 'Authentication failed! You have
an account with this email address, please try logging in
with your email and password!'
            ],
            403
        );
    }

    return response()->json( '', 204 );
}
```

AND IT IS DENSE! And only half of the actual process. Let's get through this and in the next step we will actually load our user, register if needed, and authenticate them.

Right off of the bat, we grab the `redirect_uri` from the request along with the `provider`. We then set the config needed by socialite for the `redirect_uri`. Remember in the first section of this chapter, we didn't set this key in our `services.php` file? This is where we do it. The `redirect_uri` has to be the same when you exchange the authorization code for the user and we need to get that from our SPA.

Next up, we initialize Laravel Socialite with the `driver` set to `provider`. This let's Laravel Socialite know where to exchange the token. The `provider` is also sent with the authentication request so we know where to go. After we configure the driver, we call `$driver->stateless()`. This is the key to making the whole system work! Since we set the `code` in the `request`, this is set in the request variables and Laravel Socialite will exchange that authorization code for a valid access token + user. It will also do it stateless so we don't begin a session since

we want to have Laravel Sanctum perform that task.

Now that we have successfully exchanged the authorization token, we can call `$driver→user()` to get the returned user and begin the authentication within our API. Once we have the user, we cut to the `ManageSocialUser` service with the user returned from the social provider.

Step 3: Register and/or Authenticate Your User

We will head back to the method above shortly, but let's cut over to our `ManageSocialUser` service. This service will take the social user, register if needed, and return the user from our system fully authenticated.

If you look at the method above, we create a new `ManageSocialUser` service and send over the `provider` and `profile` that was returned from the social platform. We then call the `authenticate()` method on that service to perform the authentication.

Let's hop over to our `ManageSocialUser` service. You can put this code wherever you want, but ours is in `Services/User/ManageSocialUser.php`. The first method we should look at is the `_construct()` method. We just set the `provider` and `profile` as local variables so we can use them throughout the process.

Next up we have our `authenticate()` method which performs the registration + authentication:

```
public function authenticate()
{
    $user = $this->getExistingUser();
    if( $user ){
        Auth::login( $user );
    }else{
```

```
        if( $this->checkEmailExists() ){
            $user = null;
        }else{
            $user = $this->registerNewSocialUser();
        }
    }

    return $user;
}
```

Right away, we run a check to see if there is an existing user:

```
private function getExistingUser()
{
    $user = User::where( 'provider_id', '=', $this-
>profile->id )
                ->where( 'provider', '=', $this->provider )
                ->first();

    return $user;
}
```

If there is an existing user that has the `provider_id` and the `provider` meaning the unique ID from Facebook, Twitter, or Google on our platform, we simply return that user and authenticate them. No need to register, just authenticate the user and return the user!

If there is no user, we check to see if the email exists:

```
private function checkEmailExists()
{
    $existingUser = User::where('email', '=', $this-
>profile->email)
                ->first();
```

```
if( $existingUser ){
    return true;
}else{
    return false;
}
}
```

This will determine if there is a user who has already authenticated with the email address on a different platform or through different 3rd party provider. The email address needs to be unique. If the email address exists, we return `null` which we will handle back in our `Authentication` service. If the email address does not exist, it's time to register a new user:

```
private function registerNewSocialUser()
{
    $user = User::create([
        'name' => $this->profile->getName(),
        'email' => $this->profile->getEmail(),
        'avatar' => $this->profile->getAvatar(),
        'permission' => 'user',
        'provider' => $this->provider,
        'provider_id' => $this->profile->getId()
    ]);

    $user->markEmailAsVerified();

    Auth::login( $user );

    return $user;
}
```

What this does is create a new user with the information provided from the oAuth provider. Then it marks the email as verified. We do this because the social auth provider has already verified the email address for us. Then we authenticate the user and return the user.

Now it's time to go back to our [Authentication](#) service and finish the process.

Step 4: Gracefully Handle Authentication

At this point we've exchanged the auth code for a token, either registered and/or authenticated a user, now it's time to return our own token if the user is on mobile, or just the user object if the user is on web:

```
if( $socialUser ){
    if( $mobile ){
        return [
            'token' => $socialUser->createToken( $this-
>data['device_name'] )->plainTextToken
        ];
    }else{
        return response()->json( $socialUser );
    }
}else{
    return response()->json(
        [
            'error' => 'Authentication failed! You have an
account with this email address, please try logging in with
your email and password!'
        ],
        403
    );
}

return response()->json( '', 204 );
```

So what we are doing here is simply checking if there is a user that was created or authenticated. Let's say that one wasn't and the `$socialUser` variable is `null`. Then we return a `403` error code that we can display on the front end. This

happens only if the user has authenticated with a different oAuth provider in the past or with an email + password combination. Any situation where the email address has already been taken.

If we do get a user, we know we've authenticated correctly. We then check if it's a **mobile** authentication. If so, we return a token for the **device_name** and return the token so the mobile app can store it for future use. If the user is not on a mobile app, we just return the user object.

Conclusion

I know that was a ton of moving parts, and this is probably the most complex sequence of events in this book, but it's over! With that being said if you need help along the way, reach out over on <https://community.serversideup.net> and ask any questions you need! You should be able to authenticate any 3rd party users through your app now!

OPENING YOUR API TO OTHERS

Configuring Laravel Sanctum for 3rd Party Access

Originally when I was going to write this book Laravel Sanctum was in its infancy. At that time, if you wanted to open up your API for 3rd party access, it was recommended to use Laravel Passport. However, Laravel Sanctum has been updated and provides more features and documentation and actually allows us to do 3rd party access without Laravel Passport. You can even develop in the scoping abilities you might be used to with a full featured oAuth2 server. This is exciting and beneficial because A) It's easier to set up and we already have it done and B) There's no need to install another package and configure it securely.

Let's get started by going over some of the frequent questions that come up when allowing 3rd party access.

When to use Laravel Passport?

If you need a full out oAuth2 server with all of the associated grants and scopes, Laravel Passport is your best choice. Usually the game changer on when to choose Laravel Passport is if you want users to authenticate via your application to another application. In our case, say we want other systems to authenticate with a user's ROAST id (similar to what you see with Facebook), Laravel Passport provides this ability. The other major use case is if you wanted to perform a machine to machine connection, Laravel Passport provides the proper grants to pull that off.

If you need those solutions, you can use Laravel Passport to handle mobile and web auth as well and would actually be recommended. Laravel Sanctum does API + SPA auth extremely well but if you need to have someone "log in with your app" or perform a machine to machine connection, Laravel Passport may be your best bet.

Why would you want 3rd party access?

Simply put, to allow others to access your data from their own apps. Why? Because it will in turn grow your application if users choose to build around it. It's also extremely exciting to see what the users can build using the data and get involved in your project.

We will be allowing users to generate their own access tokens in this chapter in order to consume some of our API resources from a 3rd party application.

Security Concerns

This is where security on your API routes becomes incredibly important. We've touched on the subject of hiding fields on Eloquent models that the user shouldn't see before. Now it becomes the highest of priorities. The 3rd party will be inspecting each response to consume the data that they want to integrate with. Any leaks will be highlighted and could expose a security vulnerability.

To hide a field on an Eloquent model, simply add the column name to the `hidden` array on the Eloquent model. Let's take a look at our `User` model:

```
/**
 * The attributes that should be hidden for arrays.
 *
 * @var array
 */
protected $hidden = [
    'password', 'remember_token',
];
```

We hide the `password` and `remember_token` fields when we return the response as JSON. Even though the password is encrypted at rest, we shouldn't give anyone a chance to try and crack it.

We will also be going through providing abilities for the token so we can limit the

functionality of the token. Even if an admin generates a token, we don't want admin routes accessible from a 3rd party token. This could lead to a security vulnerability if our 3rd party app gets compromised.

We will also be opening up our public API routes for a little more access if the user has a token. Endpoints like `/api/v1/companies` is completely public, but we don't want just anyone abusing resources. To solve this problem, we will rate limit these endpoints at a much lower limit per minute if no personal access token is present. If the user sends their access token in the header, then they can perform more requests since they are a verified user.

Alright, let's allow the users to generate their own access tokens!

Determining What API Endpoints Should Be Available to 3rd Parties

Since we are building an API driven application, our routes are exposed publicly. This is a powerful tool that allows for users to build upon our application and allows us to re-use code. With that being said, the more you expose to the public, the better your security has to be. This really comes into play as you begin documenting your API for public consumption.

With our routes open publicly, we can't only rely on blocking access to these routes entirely with scopes (abilities), we have to pair it still with the authorization policies that we already discussed. A token can belong to an admin and allow them to perform admin level functions. However, we need to still process that token with respect to the user.

Sometimes it makes sense to limit features of your app via token scopes. This can provide some immediate feedback, but can be navigated around by submitting an authorization request to the same endpoint used by the mobile application to grab an access token. This is why we still need to pair what the token can do, with the permission level of the user.

In our application we want our mobile app to have 100% the same features as our web app using the exact same code. However, there are limited mobile apps that don't allow that same level of functionality. If you want to limit your mobile application, you can look into granting token "abilities" with your Laravel Sanctum token. We won't be relying on the token abilities to perform authorization in our app since the mobile app has 100% the same functionality. However, we will be performing rate limiting on public routes to ensure that users can access more dynamic features when using a generated personal access token.

If you want to have a subset of features in your mobile app, I'd suggest reading below. Otherwise, skip to the next chapter.

If you are familiar with oAuth, you should be familiar with the concept of granting "scopes". In Laravel Sanctum, these token "scopes" are referred to as "abilities". By using token "abilities", we can block certain API endpoints, limit access, and even allow for more rate limiting on public routes. The ability assigned to a token can determine if the endpoint is accessible whether or not the user has permission. What does that mean and how does that work?

What this means is any route that doesn't have the ability is flat out denied access if using a 3rd party token. These routes could contain too much sensitive information to risk having the token compromised in a 3rd party application. You have to think that by allowing other devs to store your token, you have to trust them with their ability. It's better to be on the safe side.

So how will this work? You can create a middleware that checks the token scope. Luckily, Laravel has a built in way by to check the token ability through middleware. They also allow full access if the token is coming from a third party or first party app. According to the laravel docs, when checking the ability of a token, if the token was created by our app for our app (1st party web & mobile) the token check will always return true. If the token was created for 3rd party use, then we want to not even allow access to the routes.

In our case, we just want the ability to create tokens and grant access easily for 3rd parties.

Create Routes to Manage Tokens

Let's get started by creating the routes we need on the API side to manage our 3rd party access tokens. The routes we need are:

- `GET /api/v1/tokens` → This will load all of the tokens the user currently has created
- `POST /api/v1/tokens` → This will allow the user to create a new access token.
- `DELETE /api/v1/tokens/{tokenID}` → This will revoke a specific token. Users should use this if they need to refresh a token or their token gets compromised.

The first route we will build out will be the `POST /api/v1/tokens` route.

Create Token Route

The `POST /api/v1/tokens` route will allow our users to create a token on their behalf to use to access our application. One thing to note about these tokens, they have a very long expiration time (years), but may be manually revoked by the user at anytime. That means we really need to keep in mind that as we add features to our app, we need to protect against unauthorized access from users with these tokens.

The first step is to open up our `routes/api.php` file and add the following route:

```
Route::post('/tokens', 'API\TokensController@store');
```

We now have our route registered. If you haven't guessed, the next step will be to create our `TokensController`. We will be handling tokens as a resource so we will be following the appropriate naming conventions.

Add the following controller to `App\Http\Controllers\API`:

```
namespace App\Http\Controllers\API;

use App\Http\Controllers\Controller;

class TokenController extends Controller
{
    public function __construct()
    {

    }

    public function store()
    {

    }
}
```

We will be using this controller for all of our token routes. Before we finish creating our `store()` endpoint, let's create a service class

`App\Services\Tokens\CreateToken.php` file:

```
namespace App\Services\Tokens;

class CreateToken{
    public function __construct()
    {

    }
}
```

This will isolate all of our token creation process inside a single, easy to manage service. With this service, we will accept a user and the name of the token they wish to create. We will then create the token and return the created token. This will be converted by JSON and sent back to the user so they can view it and use it in their application.

Make sure your file looks like:

```
<?php

namespace App\Services\Tokens;

class CreateToken
{
    private $user;
    private $tokenName;

    /**
     * Accepts a user object to create a token for
     * @param User $user User we are creating the token
     * for.
     * @param String $tokenName The name of the token
     * provided by the user.
     */
    public function __construct( $user, $tokenName )
    {
        $this->user = $user;
        $this->tokenName = $tokenName;
    }

    public function create()
    {
        return [
            'token' => $this->user->createToken( $this-
>tokenName, ['*'] )->plainTextToken,
            'type' => 'Bearer',
            'name' => $this->tokenName
        ];
    }
}
```

Let's break this service down. First, our constructor will take a user and the name of their token. This will be provided by the front end of the application when we get to it.

Second, our `create()` method calls the `createToken(name, scope)` method on the user which is inherited through the `HasApiTokens` trait that we set up with Laravel Sanctum. The first parameter we pass is the name of the token we are creating. The second parameter is an array of scopes. For our app, these tokens will have access to all methods. Finally, we return the `plainTextToken` of the newly created token (this should look very familiar to the mobile authentication process).

Just a heads up! The token is encrypted at rest in the database. Once you create the token it will be the only time you should display the token. This prevents any unauthorized access if your database gets compromised. The user will have to revoke and re-create the token when they want to view it again.

Now, let's head back to our `TokenController.php` file and add the following code to the `store()` method:

```
public function store()
{
    $createToken = new CreateToken( Auth::guard('sanctum')-
>user(), Request::get('name') );
    $token = $createToken->create();

    return response()->json( $token );
}
```

Make sure that you include the service up top as well:

```
use App\Services\Tokens\CreateToken;
```

When the user creates a token, they can view the newly created token and begin accessing resources from a 3rd party! You may wonder what the benefit of this would be in ROAST specifically since a lot of the data is publicly available from the get go? We will be adding rate limiting to users who aren't authenticated so they don't abuse those resources.

Next, let's add a quick route to load the created tokens.

Load Tokens Route

Now that we have a simple route to create a token, we need to add a route to show all of the tokens that belong to the user. This will allow them to view which tokens they have active and return the token's ID, allowing the user to revoke the tokens.

First, let's jump back to our `routes/api.php` file and add the following route:

```
Route::get('/tokens', 'API\TokensController@index');
```

This will be the endpoint we hit to load all of the tokens for the user. Next, we head back to our `TokensController.php` and create the `index()` method that loads up the tokens:

```
public function index()
{
    $loadTokens = new LoadTokens( Auth::guard('sanctum')-
>user() );
    $tokens = $loadTokens->load();

    return response()->json( $tokens );
}
```

If you haven't guessed the process by now, we will be creating a service that loads these tokens. This service will return all of the tokens for the user and we will send them back as JSON from our API.

Finally, let's tie everything together by adding our `App\Services\Tokens\LoadTokens.php` service:

```
namespace App\Services\Tokens;
```

```
class LoadTokens{
    private $user;

    /**
     *

```

```
* Accepts a user object to load tokens for
* @param User $user User we are loading tokens for.
*/
public function __construct( $user )
{
    $this->user = $user;
}

public function load()
{
    $tokens = $this->user->tokens()->get();

    return $tokens;
}
}
```

This is a really simple service that accepts a user in the `__construct()` method. When `load()` is called, we call the `tokens()→get()` method on the user which is inherited from our `HasApiTokens` trait. We then, return those tokens back to our controller. Our controller, then converts the response to JSON and sends the tokens back to the user:

```
$tokens = $loadTokens->load();

return response()->json( $tokens );
```

Don't forget to add the service to the top of the controller as well:

```
use App\Services\Tokens\LoadTokens;
```

One thing to note that's extremely important is the actual `token` field will NOT be returned along with the tokens when converted to JSON. This is a hidden field. Even though the token is encrypted, we don't need it and we don't need to return it and give malicious users the opportunity to try and work with it.

We now can create a Personal Access Token and return a list of valid access

tokens. Finally, we have to allow the user to revoke their access token if it was compromised or they don't need it any more.

Delete Token Route

Since we are handling our tokens as RESTful resources, we will be following a similar structure to our other resources. I'm going to go a little quicker through this section since it's going to look very similar to the other endpoints that we added.

First, we need to add our endpoint to our `/routes/api.php` file:

```
Route::delete('/tokens/{token}',  
    'API\TokensController@delete');
```

Notice the `{token}` in the endpoint? We will be sending the ID of the created token to remove.

Next, we need to implement the `delete` method in the `TokensController`:

```
public function delete( $token )  
{  
    $deleteToken = new DeleteToken( Auth::guard('sanctum')-  
        >user(), $token );  
  
    if( $deleteToken->delete() ){  
        return response()->json( ['success' => true] );  
    }else{  
        return response()->json( null, 403 );  
    }  
}
```

We will be creating the delete token service next. However, the one thing I'd like to point out is the `DeleteToken` service returns true or false depending on if the token was successfully deleted. If the user tries to submit a token ID for a token they don't own, it will return `false` and in turn, we send a `403` back to the user.

Finally, let's implement the delete token service:

```
<?php

namespace App\Services\Tokens;

class DeleteToken{
    private $user;
    private $tokenID;

    public function __construct( $user, $tokenID )
    {
        $this->user = $user;
        $this->tokenID = $tokenID;
    }

    public function delete()
    {
        return $this->user->tokens()->where('id', '=', $this->tokenID)->delete();
    }
}
```

The constructor of the service accepts the user and the token ID that we want to delete. In the `delete()` method, we delete the token by the ID that was passed in. Once again this is done through the `tokens()` method bound to the user through the `HasApiTokens` trait.

And of course, make sure that the service is included in your controller:

```
use App\Services\Tokens\DeleteToken;
```

Now we have the routes available to manage all aspects of our personal access tokens! Next, let's add some rate limiting features to ensure that our public API doesn't get abused. Then we will implement a front end in NuxtJS!

Rate Limiting Requests

So our app is unique. We have views that display 100% when a guest is viewing the page and we have actions that require authentication. The best example of this is with our Company resource. As a guest, I can view all of the companies at once. However, I need to be authenticated to submit a company. If I were to take a look at the requests, I could easily grab our endpoints from the network tab in a browser inspector and just submit requests without any token whatsoever and get data/results.

The create Company endpoint is blocked via authentication where the view company endpoint is free to whomever. Now let's think about how the view company endpoint is consumed. It is either coming from our web app, our mobile app or a 3rd party. From the web and mobile, you aren't going to be clicking so fast that you really need rate limiting. However, from a 3rd party, you may have an app that requests the resource multiple times a second. We don't want to just give away that bandwidth to whomever finds our endpoint, but we need to keep it open for guest users viewing our application.

This is where rate limiting comes in, and Laravel makes this a breeze. The way I structured the rate limiting parameters is as follows:

- `auth` → The rate limit for any auth routes is 5 requests / minute. This prevents brute force attempts at cracking passwords.
- `public (no auth)` → This rate limit is 60 requests / minute. Anyone who is browsing the site fast enough to do a request 60 times in a minute is probably acting maliciously.
- `public (auth)` → With auth, this can be first party (web/mobile) or third party (personal access token). When applied, this rate limit is 1000 requests / minute. In all honesty, if you create an app that makes 1000 requests / minute to the ROAST API, you are getting more traffic than we are and I'd love to talk to you and make a special rate limit for your token.

Let's build these out!

Writing Your First Rate Limiter

So you can really be flexible with your [rate limiting](#) in Laravel 8 which is amazing! We are going to use rate limiting pretty simply to check and see if the user is authenticated and giving them a higher rate if they are since they could be coming from a 3rd party app utilizing our resources.

The instructions in the [rate limiting documentation](#) are super helpful, but let's run through how to create your first rate limiter.

First, open up the `app/providers/RouteServiceProvider.php` file. On top of the file, add the following facades and classes:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;
use Illuminate\Http\Request;
use Auth;
```

This will allow us access to the limits and rate limiter classes that we can use to set up our rate limiter. We also need access to the request for a dependency injection when setting up a `RateLimiter`.

Next, create a method that configures your rate limiting at the very bottom. This is outlined in the documentation as well:

```
/**
 * Configure the rate limiters for the application.
 *
 * @return void
 */
protected function configureRateLimiting()
{
    // Define rate limiters
}
```

After, the method is created, ensure you call it from the `map()` function higher up in service provider. This ensures the rate limiters are mapped and registered for you to use:

```
/**  
 * Define the routes for the application.  
 *  
 * @return void  
 */  
public function map()  
{  
    $this->mapApiRoutes();  
  
    $this->mapWebRoutes();  
  
    $this->configureRateLimiting();  
}
```

Now we can add our first Rate Limiter, so jump back to the `configureRateLimiting()` method and add the following code:

```
protected function configureRateLimiting()  
{  
    RateLimiter::for('auth', function( Request $request ){  
        return Limit::perMinute(5)->response( function(){  
            return response("You've exceeded the request  
limit. Please try again later.", 429);  
        });  
    });  
}
```

Alright let's break this down. First, we register a rate limit named `auth`. This will be applied to all of our `auth` routes (coming up next when we apply rate limiters). In the call back, we inject the incoming request and return a `Limit`. The `Limit` facade has a `perMinute()` method which allows us to specify how much of the current action is allowed to be called per minute. In this case, it's 5 times per minute when we apply it to a route.

Finally, we chain a `response()` to the `Limit` which allows us to customize the response (this will make more sense in the next rate limit). We just let the user know that they have exceeded the request limit and have to wait to try again.

We will apply this rate limit to all of our requests that handle authorization to prevent brute force attacks. While we are inspecting this method, let's throw in our code for rate limiting access to our API endpoints:

```
RateLimiter::for('public', function (Request $request) {
    return Auth::guard('sanctum')->user()
        ? Limit::perMinute(1000)->response( function(){
            return response("You've exceeded the
request limit for this token. Please reach out if you need
higher availability!", 429);
        })
        : Limit::perMinute(60)->response( function(){
            return response("Please log in to receive
more requests per minute.", 429);
        });
});
```

Similar to our `auth` rate limiter, the `public` rate limiter will be applied to all of our general purpose public API endpoints. The major difference being the rate limit changes based on whether or not the user is authenticated or not. The authentication happens through the SPA or Mobile app or if they provide a personal access token. This will prevent unauthenticated users from abusing our endpoints!

Applying our Rate Limiters

Now that we have our rate limiters created, let's apply them! I like to apply the rate limiters through middleware and in the resource controllers. I also think of them as kind of a flexible middleware in the sense of "you can access this endpoint, just not too much". I've applied this middleware to all of our API endpoints and auth

endpoints, but let's look at the `/api/v1/companies` endpoint specifically.

Open up `app/Http/Controllers/API/CompanyController.php` and look at the `__construct()` method:

```
public function __construct()
{
    $this->middleware('auth:sanctum')->only(['store',
'update', 'like', 'destroy']);
    $this->middleware('verified')->only(['store', 'update',
'destroy']);
    $this->middleware('throttle:public');
}
```

Notice the new middleware? It's `throttle:public`! This now provides the appropriate throttling to the endpoint to prevent abuse! If we were to request this route through the Insomnia REST client, we should get a paginated list of companies, but the response header should look like:

No Authentication:

The screenshot shows a successful 200 OK response from the Insomnia REST client. The response details are as follows:

NAME	VALUE
cache-control	no-cache, private
content-type	application/json
date	Tue, 30 Mar 2021 22:11:43 GMT
referrer-policy	no-referrer-when-downgrade
server	nginx
strict-transport-security	max-age=31536000; includeSubDomains
vary	Accept-Encoding
x-content-type-options	nosniff
x-frame-options	SAMEORIGIN
x-ratelimit-limit	60
x-ratelimit-remaining	59
x-xss-protection	1; mode=block

Authentication With Valid Token:

A screenshot of a browser developer tools Network tab. At the top, it shows a green "200 OK" button, "458 ms" latency, "4.9 KB" size, and "Just Now" timestamp. Below the header are tabs: "Preview", "Header 12", "Cookie", and "Timeline". The "Header" tab is selected, displaying a list of 12 headers with their names and values. A "Copy to Clipboard" button is at the bottom right.

NAME	VALUE
cache-control	no-cache, private
content-type	application/json
date	Tue, 30 Mar 2021 22:12:45 GMT
referrer-policy	no-referrer-when-downgrade
server	nginx
strict-transport-security	max-age=31536000; includeSubDomains
vary	Accept-Encoding
x-content-type-options	nosniff
x-frame-options	SAMEORIGIN
x-ratelimit-limit	1000
x-ratelimit-remaining	999
x-xss-protection	1; mode=block

We now have an `x-ratelimit-limit` and `x-ratelimit-remaining` header that shows how many requests per minute we have available. It's much higher with our access token as it should be if a user has a powerful app utilizing our resources.

If we exceed our limit we will get a `429` response that looks like this:

A screenshot of a browser developer tools Network tab. At the top, it shows an orange "429 Too Many Requests" button, "290 ms" latency, "50 B" size, and "Just Now" timestamp. Below the header are tabs: "Preview", "Header 10", "Cookie", and "Timeline". The "Header" tab is selected, showing a list of 10 headers. A message "Please log in to receive more requests per minute." is displayed below the header list.

We tell the user through the response code of `429` that they've exceeded their requests and they need to log in if they want more.

That's really all there is for applying rate limiting! Laravel makes this a breeze and it really helps to prevent API abuse from third parties.

Now that we have our API side completed, let's move over to the NuxtJS front end and build a slick UI for managing our tokens!

NuxtJS Interface and Settings

When implementing the ability to manage Personal Access Tokens, the majority of the heavy lifting is done on the server side. So we've done most of the work already! However, we do need a way for the user to manage these tokens with ease. I'll run through a few of the gotchas and frontend ways to implement this.

A lot of managing a resource through NuxtJS has been covered in this book, so I'm going to focus explicitly on the unique aspects of token management (showing tokens, creating a token, revoking a token). I've already created a RESTful API Wrapper for our tokens resource and registered it within NuxtJS. Also, I've created a table that shows the existing tokens. With that being said, let's run through some more specific logic.

Where Should This Functionality Live?

Usually, when managing tokens, you think of them as directly relating to the user and providing access to resources. I would place this functionality on the user's profile page within ROAST. A settings page also makes sense as well within your own app. If you look at `/pages/account/profile.vue` this is where I added all of the token functionality.

Creating A Token

Before we can list any tokens the user may have, we have to create a token. To do this, we need to send the `name` of the token to the API. I did this through a simple form field that allows the user to enter the token name and submit it to the API.

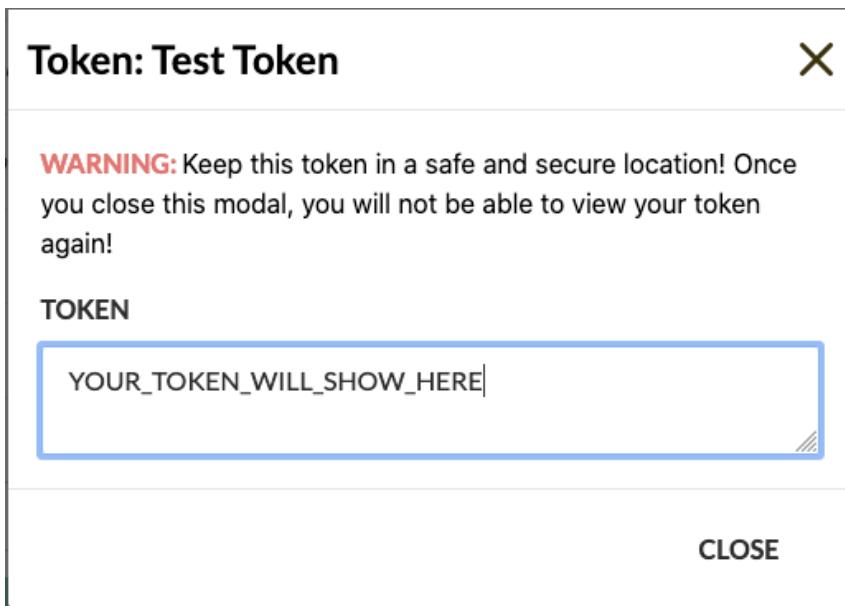
This is the RESTful request:

```
async createToken() {
    await this.$api.tokens.store( this.token_form )
        .then( function( response ) {
```

```
        this.loadTokens();
        EventBus.$emit('view-token', response);
    }.bind(this));
},
```

The main aspects of this method to pay attention to is what happens upon success. First, we run the `loadTokens()` method which will load all of the available tokens for the user. These tokens will then be displayed in a table that allows the user to revoke them if necessary.

More importantly, we emit a `view-token` event. Remember when we implemented our API, we return the actual token upon creation? This is where we display it. It's important to notify the user that they won't be able to view the actual token again so they must store it in a secure location. In ROAST, the following modal appears after a token has been created.



The user can then copy and securely store their token for future use.

Displaying and Revoking Tokens

Displaying tokens is pretty straight forward. After you load the tokens from the database, we display them in a table.

Access Tokens

Use these tokens in your own application to consume ROAST data.

EXISTING TOKENS

App	<i>Last Used: Never</i>	Revoke
Test Token	<i>Last Used: Never</i>	Revoke

The feature that I'd like to point out regarding showing the tokens is it's a good idea to show when it's last used. This can help the user know if their token was compromised and used maliciously if the date and time it was last used was out of the ordinary.

When the user clicks "Revoke" we send a delete request to the API and the token is invalid and cannot be used any more.

It may be a little bit tricky to get the tokens set up, but the display and management of tokens isn't overly complex! Of course, if you have any questions, feel free to reach out in the forum.

Using A Personal Access Token

Now that we have the token, how do we use it? To use a Personal Access Token, you have to make an API request with the token in the header. Let's say we are loading all companies using the `/api/v1/companies` endpoint. We'd add an `Authorization` header to our request with a `Bearer {token}`. This will allow our API to pick it up and authorize the request.

Must Accept JSON

Since our API returns JSON upon success or failure, we must add the header `accept: application/json` to our request. This is extremely important or errors upon validation will just be an empty response and be super confusing.

Content Type

When creating or updating a resource (company/cafe), you must send the `Content-Type` header and have it be set to `multipart/form-data`. The reason for this is you can update logos and header images on these resources and that requires a proper content type.

Scoping Access Tokens

Within ROAST, we didn't make use of the access token scopes (referred to as "abilities" with Laravel Sanctum). I did, however, want to touch on some possible use cases for these.

Limiting Access

Let's say that we only wanted admin users with a personal access token to manage companies. That would mean blocking the `POST store`, `PUT update` and `DELETE delete` routes on the companies endpoint for users who tried to access with a personal access token and didn't have permission.

First, we'd create the personal access token for the user with an ability of `[manage:companies]` if they were an admin. This ability would not be assigned to the token if the user didn't have that permission.

Next, we'd take a look at our `CompanyPolicy` which authorizes access to company routes. Let's look at our `update()` method in that policy:

```
public function update( User $user, Company $company )  
{  
    if( $user->permission == 'admin' ){  
        return true;  
    }else if( $user->companies->contains( $company->id ) ){  
        return true;  
    }else{  
        return false;  
    }  
}
```

To block that access, for users who don't have the proper ability, we'd wrap the functionality with:

```
public function update( User $user, Company $company )  
{  
    if( $user->tokenCan('manage:companies') ){  
        if( $user->permission == 'admin' ){  
            return true;  
        }else if( $user->companies->contains( $company-  
>id ) ){  
            return true;  
        }else{  
            return false;  
        }  
    }  
}
```

```
        return false;
    }
}
```

Now, this will check to see if the token has the ability first before processing any other authorization methods. Remember, if the token is generated from a first party such as our app or mobile app, this will automatically return `true` so the other authorization functionality will be processed successfully.

API TIPS, TRICKS, AND GOTCHAS

RESTful Response Codes: How To Use Them

The key to developing a great API is using the proper HTTP response codes. What this means is when a request is sent to the server, no matter in what type of app, a response code is returned.

Using proper response code allows the developer to catch errors from the API server and respond accordingly in the app that integrates with the API. When building an API it is important to return the proper code so 3rd party developers (and yourself) know what happened. Did a resource get created? Was there an error? With the proper response code, this is clearly communicated to the developer.

There is a wide variety of different codes you can return, this glossary will go through the ones we use in this course. We won't be using all of the possible response codes, so if you want more information check out https://en.wikipedia.org/wiki/List_of_HTTP_status_codes and a few different examples here <https://www.restapitutorial.com/lessons/httpmethods.html>.

Groups

All of the possible HTTP response codes are grouped by level. The main levels we are going to be using are **200** level which are successful responses, **400** level which are "client" error codes meaning the user is not authenticated, or there is an issue with the request, and **500** level which are server side errors that you really have no control over if you are consuming an API.

2xx Level Response Codes

The 200 level response codes are for requests that have completed successfully. This means there were no errors in the request. However, there are different [200](#) level responses that help the developer know even more information.

200

This response is used with successful GET requests. Essentially it let's the developer know that their request completed successfully and there should be some data returned since it's from a GET request.

201

The [201](#) response code tends to be used for POST requests. It is another successful response, but this time signals that a new resource has been created. POST requests are used to submit data to the API and that usually means in the sense to create a resource.

It's common to send the newly created resource back to the requestor along with a [201](#) response code.

202

This is a unique response code that's used mainly for systems that operate with a queueing system. Say you are going to send a request to the API to send out 5,000 emails. You don't want to wait until all emails are sent out to get a response from the system or you will be waiting for hours.

With a queueing system, you queue up the emails and return a response. This response would have a [202](#) which means that the request has been accepted successfully but not completed. The queueing system will be processing the emails and handling errors the way it's set up to.

204

When you update an entity (PUT/PATCH) or delete (DELETE) an entity, you'd use this response code. What this code means is there is no content being returned with the response, it's literally just this response code. The response processed successfully but there is nothing to return back to the requester.

4xx Level Response Codes

These are the error messages. When something goes wrong you will return one of these response codes. The most familiar one is a **404** just like you'd see in a regular app or any web page that doesn't exist. We will be utilizing that code in the API as well if someone tries to access a resource that doesn't exist.

400

When the client sends a request that is invalid in some way (ex. a photo that's too large to upload), we return a **400** meaning it's a 'Bad Request'. The client will have to re-submit the request in a way that cooperates with the API.

401

This is one of the most common errors that we will return when building an API. It means that the user needs to be authenticated to access a resource. This is returned if a user tries to access a restricted part of the API.

This is very similar to **403** which is forbidden, but I tend to view the two at different levels. **401** comes first. What that means is right away, before accessing a secure part of the API, we check if the user is even logged in and has a token. If they do not, then we return a **401** and end the request right there.

403

The second most common error message when building an API. What this means is the user does not have access to the resource.

After an authenticated user tries to access an API, we then check to see if the user has access to the resource they are trying to access. An example would be if I'm trying to get data about order number **1001**. If I own order number **1001** and I'm authenticated, I can get that private information. Now if I adjust the request for order number **1002** and I do NOT own that order, I'd get a **403** response which is forbidden.

404

Probably the most common error code people are familiar with. This just means the resource is unavailable. Say if I'm trying to access user number **5328** and that user doesn't exist, then I'd return a **404** error from the API.

This is very similar to the pages you see when you access a page on a website that doesn't exist any more. There's just nothing there!

405

What this error code means is if you are trying to access a GET route through a POST request. It means the method does not exist. This is a rarer error code when building an API and SPA combo since you know all of the routes because you created them. However, if you try to access the right route with the wrong verb, we return a **405** error code.

When a 3rd party developer is accessing your API, this is a little more common of an error since they might mistake a GET for a POST url depending on the context.

422

This error code is used for validations and when they fail. What this means is if you send a properly formed request but one of the validations failed it is an "Unprocessable Entity". This error code will return telling the developer that they need to check the data they sent over is accurate.

429

One of the biggest security lockdowns you can do on an API is rate limit it. What that means is block users from trying to call it too many times in a certain time frame. When a user surpasses your set rate limit, we return a **429** error saying "Hey! You need to slow down". This tries to make sure that no one can brute force data from your API.

5xx Level Codes

Anytime you receive a **5xx** level error code, this means that there is a server issue. If you are a 3rd party developer and receive this, probably a good time to reach out to support. If you are developing your own API, time to jump into the logs.

503

The **503** error means the service is unavailable. This happens for one of two reasons. First, it could mean the service is down for maintenance. A planned outage like this is usually a good thing because bugs are being fixed or features are being added. However, it could also mean that the server is overloaded. If that's the case, it's time to dive in and see what's going on with your API.

504

A **504** error means that your request has timed out. This usually happens when you are using a proxy or some sort of service to prevent a ton of traffic. However it can also happen if your web server settings have a timeout limit of X amount of time and it's exceeded.

I usually receive this if the server needs to do a large task that's not queued and is protected by Cloudflare. If you receive this error and own the API, I'd recommend thinking about queuing up your process or adjusting the time out on your Cloudflare server. Otherwise the user isn't going to know if the process went through and might try it again but still have the process running.

How Cross-Origin Resource Sharing (CORS) Works

Probably the most frustrating part of developing your API is dealing with CORS (Cross-Origin Resource Sharing). According to [MDN docs](#): "*Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any other origins (domain, scheme, or port) than its own from which a browser should permit loading of resources.*"

What this means is that CORS is a set of standards that enables access to certain resources for security purposes. If you inspect your network tab when you make a request to our ROAST API, you will see an `OPTIONS` request sent right away. This will check with the server to see if the request is permitted. If the request is permitted, meaning the proper headers such as `Access-Control-Allow-Methods` and/or `Access-Control-Allow-Origin` accept the request, the request then goes through. These headers can block certain requests from methods (remove all POST methods) or origins (if you want only your first party app to access the API). For some APIs this can be extremely useful. With ROAST, we don't dive in too deep into the weeds with CORS, and essentially open up our API to any method and any origin since we allow 3rd party access.

However, within ROAST, we definitely implement CORS, but we don't jump through all of the complexities that you can do to implement. Let's dive into the Laravel CORS file found at `/config/cors.php` and go through the options there.

Paths

At the beginning of our `/config/cors.php` file we have a `paths` key that is an array. This allows us to explicitly define paths that CORS is enabled on. With ROAST, this is what our `paths` config looks like:

```
'paths' => [
```

```
'sanctum/csrf-cookie',
'login',
'logout',
'register',
'email-forgot-password',
'reset-password',
'api/v1/*'
],
```

What this does is open up any requests to the routes available for our web and mobile front ends. It also opens up these requests for 3rd party users. If we don't have the routes that we need opened up, the policy doesn't allow access to the resource. If we removed the `/api/v1/*` from the paths file, our console would be filled with red and warnings stating `Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at https://api-roast.dev.test/api/v1/brew-methods`. We are now blocking any requests to these endpoints.

When would you want to block a request? Say if you have a semi-private API and you want only specific origins to access the content like an intranet or a registered client. You could use CORS to block access to resources that you want protected.

What's kind of confusing is there's two different approaches. First, you can completely disable CORS on a route by NOT including it in the `paths` array (like we did above). This would remove the `Access-Control-Allow-Methods` and `Access-Control-Allow-Origin` headers from the request resulting in a blocked request. If you want to enable CORS on the route, BUT limit access in certain ways, you must include the route in your `paths` array and then move on to the next sections where you can block access in other ways.

Allowed Methods

This is interesting because you can block certain methods (`GET`, `POST`, `PUT`) on your API. Maybe you don't want people submitting any data to your API, you can allow ONLY the `GET` method.

With ROAST, we have every method allowed and our configuration looks like:

```
'allowed_methods' => ['*'],
```

This is because we want our front end and our mobile app to have full functionality since we've built the app 100% API driven. If you want to allow only certain request methods, you could do something like:

```
'allowed_methods' => ['POST', 'PUT'],
```

You'd then get the `Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at https://roastandbrew.coffee/api/v1/brew-methods` error if you submit a GET request.

Personally, I can't think of a time where I'd block a specific method through CORS since most of the security and authorization I've implemented are handled through middleware and policies, but it's available if you need it. The next two settings can be extremely useful depending on how you onboard users to your API and allow access.

Allowed Origins and Allowed Origins Patterns

The `allowed_origins` and `allowed_origins_patterns` have very similar functionality in allowing origins. What allowing an origin means is you can set which URLs can access your resources. Say you have a frontend SPA like we do and you only want that frontend SPA URL to access the API resources. You can add that url to the `allowed_origins` array.

Another use case would be if you have a paid for API and you really want to vet

the users who access resources based on their URLs or you let users access your API from 3 URLs maximum, you could add their urls to this array.

The `allowed_origins_patterns` allows you to allow a pattern for access to your resources. Let's say you want any subdomain to access a resource, you could do `*.tld.com`. When working with both the `allowed_origins` and `allowed_origins_patterns` it's important to note as stated in the comment on top of the `config/cors.php` file, "You don't need to provide both `allowed_origins` and `allowed_origins_patterns`. If one of the strings passed matches, it is considered a valid origin."

With ROAST, we allow any origin since we opened up our API to 3rd parties:

```
/*
 * Matches the request origin. `[*]` allows all origins.
 */
'allowed_origins' => ['*'],

/*
 * Matches the request origin with, similar to
 `Request::is()`
*/
'allowed_origins_patterns' => [],
```

We also don't specify an origin pattern since we allow all origins.

Allowed Headers

This option is to let the client know if the header they are sending is allowed. For example, we need to send the `X-XSRF-TOKEN` header (see "Supports Credentials") to verify that the user can pass credentials to access a resource. We will first make a "preflight" request with the "OPTIONS" verb and see if our request is valid. Upon response, we can see which headers are allowed that are custom to send.

Some other common headers to check is **Content-Type** to see if we can request a certain content type such as JSON.

We let the user send whatever headers they want with ROAST. Once again, we don't do a lot with headers in ROAST. The **X-XSRF-TOKEN** and rate limiting headers are all we work with and we need to allow them, so our configuration looks like:

```
'allowed_headers' => ['*'],
```

Exposed Headers

This is a setting I've never used when developing an API. Essentially what it does is allows you to limit which headers are exposed when a request is set. This post explains a little bit more: <https://www.import.io/post/exposing-headers-over-cors-with-access-control-expose-headers/>. In the post they explain how they wanted to check the content size of an image before downloading. They needed to ensure the header with the size was properly returned. If you don't have this in your settings, that header will not return correctly and you can't perform the functionality you are looking for.

For ROAST, we don't use this header so we have it disabled:

```
'exposed_headers' => false,
```

Max Age

We don't use this header with ROAST, but it can be EXTREMELY useful. What this does is "indicates how long the results of a preflight request (that is the information contained in the Access-Control-Allow-Methods and Access-Control-Allow-Headers headers) can be cached." <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Max-Age>. By implementing this header we can tell the browser to cache the results of the pre-flight request so they don't have to do **OPTIONS** preflight request every single time. You can enable and

disable this header. If it's enabled, provide an integer value in seconds:

```
'max_age' => 3600,
```

To save bandwidth on our server, we will be implementing this header in a future release of ROAST. We don't need to make an **OPTIONS** preflight request every time we need to load a resource.

Supports Credentials

This is by far the most important header we need to enable within CORS. What this does is allow the user to send over their credentials via a header to access resources. We need this enabled with Laravel Sanctum so we can authorize requests from both first and third parties. Our config looks like:

```
'supports_credentials' => true,
```

Hopefully this helps decipher a little bit of how CORS works and a few of the settings you can change to make your app more secure and access limiting.

Handling "Imperfect" RESTful Routes

Let's face it. As much as we think our app is perfect there are always times where you may need to structure an endpoint in a way that doesn't necessarily fit the flow of what is "expected". Before getting started, I don't advocate for going your own path when following standards. You should try to adopt as many RESTful standards as possible and abide by the pre-defined set of rules so other devs can integrate seamlessly.

Search

With that being said, I'd like to point out a few examples of when you might break with following a RESTful standard. The first example is implementing a heavily indexed `search` endpoint. Let's say you have a standard `index` endpoint for `/api/v1/cafes`. This endpoint accepts a few parameters, used in standard, non-complex areas such as pagination on the homepage, or somewhere else.

Now you want to make a better way for users to filter cafes by providing a ton of search features, possibly a full text Elastic search functionality, but want to keep the `index` endpoint available for smaller uses. You could also apply certain rate limits to the `search` endpoint since it's more powerful. The way I'd approach this is by adding a resource method to our `CafesController.php` named `search()` and implementing the functionality there. I actually did this within ROAST in combination with the next section (Top Level Endpoints for Nested Resources).

This endpoint allows for a Cafe to act as a nested resource and paginate through the `index` endpoint on a Company page, but also act as a standalone and be indexed and searchable by itself. It wouldn't necessarily make sense to have two Cafes Controllers for this, so I added the `search` method to the `CafesController.php` file. To name the endpoint, I tried to match the small, one word naming conventions of a standard RESTful endpoint. I believe this is the best approach to achieving the functionality you are looking for without convoluting

your code and making it hard to follow. The function looks like:

```
/**  
 * Searches all cafes  
 *  
 * @param \App\Http\Requests\API\Cafe\SearchCafeRequest  
 * @return \Illuminate\Http\Response  
 */  
public function search( SearchCafeRequest $request )  
{  
    $searchCafes = new SearchCafes( $request->all() );  
    $cafes = $searchCafes->search();  
  
    return response()->json( $cafes );  
}
```

The search is then passed off to a SearchCafes service which returns the results as JSON.

Top Level Endpoints for Nested Resources

As I mentioned before, the other time where you might have to break away from a proper RESTful structure is if you want to make a top level endpoint for a nested resource. We did this with the Cafes as well. Cafes belong to a company and are usually access via `/api/v1/companies/{company}/cafes` or `/api/v1/companies/{company}/cafes/{cafes}`. However, we also wanted to keep this relationship but ALSO allow cafes to be searchable like their own top level resource.

To do this we added the following route:

```
Route::get('/cafes', 'API\CafesController@search');
```

This route responds to a **GET** request to `/api/v1/cafes` and searches the cafes independently of the company with the parameters provided. Personally, I believe this is alright since it cleanly adds functionality to your API and follows general RESTful standards of proper HTTP verb and response.

General Notes

Like I mentioned, try to stick to RESTful naming schemes as much as possible. This will allow you to onboard internal devs much faster and easier and allow you to open up your API in a way that users will understand how to work with.

If you need to break away from the standards, make sure you are as clear as possible in your naming conventions. I tend to use a small, one word method name like `search` to handle the response.

When breaking from RESTful standards, always use the proper verb. **GET** requests should return data. **POST** requests should create a resource, **PUT/PATCH** requests should update a resource, and **DELETE** should remove a resource. NEVER mix those up!

If you follow those guidelines you should be able to seamlessly integrate and handle special routes.

Benefits of API Versioning

One of the biggest questions I've received is "Why would you version your API from day 1?". Well, because you want it to grow and not have to deal with upgrade issues. What do I mean by that? Let's say you created your `POST /api/companies` endpoint right away. You release an app and users start to integrate with your API right away adding all kinds of companies. You are excited and you want to release an update so you allow users to submit cafes along with companies at the same time.

The users who have their app set up to feed into your API will now have broken requests if they don't properly change the format of their requests. This can lead to a ton of headaches. More importantly if users are using your resources and you start returning a different format and their apps break, you could lose customers.

That's why I always start with all of my endpoints prefixed with `/api/v1` so when I make a major change and we roll out updates on publicly available API endpoints we can do so in a way that doesn't break existing applications built on the API. We can also give time to allow users to transition to a different endpoint and can update their code.

So if we roll out a different API version `/api/v2`, we can have the new functionality available while simultaneously supporting `/api/v1` for a period of time. This allows the customers to seamlessly transition to your new structure.

Tips & Tricks for Testing APIs

PHPUnit testing is worthy of a book itself (Adam Wathan did just that with [Test Driven Laravel](#)). There's so many complexities, tricks, and techniques to writing tests. This chapter is going to focus solely on a few of the techniques we use for testing an API.

All of our tests are located in the [GitLab repo](#) so if you want to check them out and see how they all work together, they are available to view.

Basic Set Up

If you look in the root of your Laravel install you will find a `phpunit.xml` file. This holds all of the configuration for your app's tests. Most of it we won't worry about, but let's add and adjust a few settings.

Before we get too far into the weeds, I'd like to point out that all tests for your application will live in the `/tests` directory. I'd HIGHLY recommend keeping this directory organized as best as possible. You will be creating a huge amount of tests throughout your application development process and, like any other code, the cleaner the organization the better.

Back to our `phpunit.xml` file. You should find a `<testsuites>` tag with a bunch of `<testsuite>` tags as children. These are the top level directories that PHPUnit will look into to find your tests. I added one more:

```
<testsuite name="API">
    <directory suffix="Test.php">./tests/API</directory>
</testsuite>
```

It could be argued that these tests could live in the `/Features` directory since it's closely matched to feature testing. However, I like to make the API directory just to be explicit that the tests in that directory are going to be calling API endpoints.

With that being said, create the `/tests/API` directory right now so we have

another bucket to dump code into.

One thing to note as well, is the `<directory>` tag and it's configuration. The `<directory>` tag has a `suffix` attribute that's set to `Test.php`. What this means is PHPUnit will recursively scan the directory looking for all files that end in `Test.php`. You can nest these files as deep and as organized as you'd like within your folders, but the file containing the test must end in `Test.php`. When we start making our tests, you will see how this works.

Let's set up the database.

Database Set Up

There a few ways to set up your testing environment in regards to the database. I prefer to have it replicated to the same environment that your app will live on. To me, it makes no sense to run your tests on sqlite if you are running your app on MySQL. There are features and relationship set ups that don't even work in sqlite and can break your tests even though your app runs just fine.

To get started with this, look at the `<php>` tag within your `phpunit.xml` file. In that file, you will see a `<server name="APP_ENV" value="testing" />`. This specific tag directs php unit to load variables from a `.env.testing` file (which we will have to create). In that file we will set up our testing database. Everything else in the `<php>` tag I leave alone.

Let's create our `.env.testing` file by duplicating our `.env` file. We want it to match EXACTLY the same so our tests run in the same environment as our actual app. The only variable you need to update to test your app is the `DB_DATABASE` variable assuming you will have a testing database that lives on the same database server as your development app. Along the CI/CD process, this should be the same if you are implementing that as well.

First, let's update the `DB_DATABASE` value in the `.env.testing`. If we don't do

this, it will connect to our dev database which will clear everything out. Let's set the `DB_DATABASE` to `roastandbrew_api_test`. This means that we now have to create this database on our development database server. Next, you have to do is create a database with the name `roastandbrew_api_test` (or whatever your app name is) and the tests will set it up for you. You don't have to run any migrations.

Now, you will be running your tests in the same environment as your development app to ensure they actually catch bugs and use relationships that exist. That's all we need to do to set up! Let's touch on a few specific API testing features.

Running Tests

To run your tests, you will need to connect to your development server and run:

```
php ./vendor/bin/phpunit
```

You will see the process run and any errors or warnings come up. Testing is probably the only time you like to see errors since it actually prevents you from dealing with them in your app's code. If you get a green "OK" your tests have passed!

Building Your First Test

For this tutorial, we are going to use the `BrewMethod` resource. Why the `BrewMethod`? Isn't it a "non-essential"? Yea but it's also a perfect starting point and runs through pretty much all of the scenarios testing permissions, responses, database persistence, etc.

Create A Brew Method Factory

The first thing we need to do is create a Brew Method factory. A factory is a way to quickly build out models that you can work with, persist, and test. To do this, create a `/database/factories/BrewMethod.php` file and add the following code:

```
namespace Database\Factories;

use App\Models\BrewMethod;
use Illuminate\Database\Eloquent\Factories\Factory;

class BrewMethodFactory extends Factory
{
    protected $model = BrewMethod::class;

    public function definition()
    {
        return [
            'method' => 'Aeropress',
            'icon' => '/path/to/icon.svg'
        ];
    }
}
```

If you are coming from Laravel 7, this looks DRASTICALLY different. Honestly, I love it though. It's namespaced, class based and easy to use. There is no more `factory` helper. For more information on Laravel 8 factories, check out their [documentation](#).

What this does is define the factory to use the `App\Models\BrewMethod` model. We imported this through the `use` statements and defined it using the `protected $model = BrewMethod::class`. This associates the model with the factory.

Next, we have our `definition()` method. This is the defaults when used to

creating a new model. For Cafes and Companies, these will be a lot more complex. For a Brew Method, it's simply just a method and an icon.

Finally, we have to alert our `BrewMethod` model that we have a factory to use. When running our tests, we will be instantiating our model, but doing it in a factory format. Open up the `app\Models\BrewMethod.php` file and add the following to the `use` statements:

```
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Database\Factories\BrewMethodFactory;
```

Then make sure that the model, uses the `HasFactory` trait like this:

```
class BrewMethod extends Model
{
    use SoftDeletes, HasFactory;

    protected $table = 'brew_methods';

    ...
}
```

Finally, override the `newFactory()` method on your model:

```
/**
 * Binds the model to a factory
 */
protected static function newFactory()
{
    return BrewMethodFactory::new();
}
```

Now our brew method model is set up to use the factory! This closely relates our model to our test and makes it really easy to generate new brew methods to use for testing. All other models will be set up the same way as well. Since I'm focused

on solely the aspects of API testing, I'm not going to dive into all the bells and whistles of the factory and how to use them, so I'd highly recommend reading the [documentation](#). There are a ton of cool features on how to generate relationships, use Faker (a library that helps generate fake data for testing), and work with tests in a variety of ways.

Structuring Your Tests

Tests are extremely verbose meaning that for one method you could have 10-20 tests. It's extremely important to structure the tests in a way that you can scope down to a certain group. It will make your testing experience much more enjoyable and your code maintainable.

I structure my tests based off of resource based directories. What I mean by that is, in my `/tests/API` directory I have directories for my resources. So for brew methods I have a `/tests/API/BrewMethod` directory. Within that directory I create a file for each endpoint. This allows me to scope down and do specific endpoint testing that matches how my API is set up.

Refreshing Database Between Tests

I always refresh the database before running a test suite. This way we have a blank slate before running tests and there are no lingering pieces of data stored that could interfere with the testing process. To do this, we need to use the `RefreshDatabase` trait. Let's start looking at our tests by adding `/tests/API/BrewMethod/CreateBrewMethodTest.php`:

```
<?php  
  
namespace Tests\API\BrewMethod;  
  
use Illuminate\Foundation\Testing\RefreshDatabase;  
use Tests\TestCase;
```

```
class CreateBrewMethodTest extends TestCase
{
    use RefreshDatabase;

}
```

By implementing the `RefreshDatabase` trait, the tests will clear the database before running so nothing gets in the way.

Set Up Method

Another feature I implement on all my tests is the `setUp()` method. This comes within PHPUnit and allows you to set up any resources you need to run your tests such as users. Since we clear the database before each test, we will need to ensure we have a user created to run the test as. To do this, add the following method to the top of your class:

```
private $user;
private $owner;
private $admin;

protected function setUp(): void
{
    parent::setUp();

    $this->user = User::factory()->create();
    $this->owner = User::factory()->create([
        'permission' => 'owner'
    ]);
    $this->admin = User::factory()->create([
        'permission' => 'admin'
    ]);
}
```

```
]);  
}
```

Now we have 3 private variables that represent different user permissions in our system that we can run tests with. See how each method could have many tests associated with it from different perspectives? Each user can hit an endpoint and expect different results.

A few things to note about the `setUp()` method. First, it's overriding the `setUp()` method but you have to make sure the global config is ready. To do this, the first line should be `parent:setUp()`. This will run any global config on the test suite. We also have to state that there will be nothing returned from this method. That's done through `setUp(): void` in the method signature.

After those are configured, we can create any resources we need to aid in the running of our tests. In this case we are creating 3 users. The `User` model has been set up like the `BrewMethod` model with a factory so calling `→create([])` and passing it overwritten values is an easy way to make a user we can work with. Make sure you include the `User` model on top `use App\Models\User`.

Acting As

The `actingAs()` method is probably the most crucial method in testing an API. It allows you to simulate acting as a user to perform a request. So those users that you set up in your `setUp()` method. You can "act as" those users and send requests to certain endpoints. In ROAST, only admins should be able to create the `BrewMethod` resource so for the `user` and the `owner` these two should get a `403` response code from the API and the `admin` should get a `201` response code. The admin's brew method should also be present in the database. Let's touch on the uploading files, `assertStatus` and `JSON` methods before we run any examples.

Send JSON to an API Endpoint

Since we are building an API, we need to send JSON to an endpoint to ensure it works correctly. There's a really awesome helper function within Laravel that does just that. Combined with the `actingAs()` method, you can send JSON with the `→json()` method with any of the HTTP Verbs to an API endpoint.

The first parameter of the `→json()` method is the HTTP Verb and the second is the URL of the endpoint. If we want to get all Brew Methods from the `/api/v1/brew-methods` endpoint, you just need to run `→json('GET', '/api/v1/brew-methods')`. From there you can chain together the assertions for your testing.

To send data to an endpoint, such as creating a brew method, you'd run `→json('POST', '/api/v1/brew-methods', ['name' => 'data'])` which will send the array as the 3rd parameter to the endpoint. This method is critical for testing APIs. We will do a few example soon.

Uploading Files

If we actually want a Brew Method to be created as an admin we need to send a file that is the icon of the brew method to the API. This could get pretty tricky requiring you to read in a file from somewhere and send it. You also have to maintain a testing resources directory. This could unravel into a mess really fast.

Luckily, Laravel has some helper methods that make this super easy! First, ensure that `use Illuminate\Http\UploadedFile;` is included on top of your file. Next, within your test, you can create a sample file to send to an endpoint like this:

```
$icon = UploadedFile::fake()→image('brew_method.jpg');
```

To pass this to an endpoint, include it in the array of data when you `POST`. Any validations for an uploaded file, such as an icon on a brew method, will pass if this is included.

Assert Statuses

After sending a request to an endpoint, we need to actually test if it passed or failed. In my opinion, there are three stages of each test, the set up, the running of the code you are wanting to test, and the validation that the code is performing as expected. So far, we've touched on setting up the test, submitting data to an endpoint or retrieving data from an endpoint, now we are in the validation phase where we ensure everything ran correctly. Once again, there are so many assertions you can run that are outside of the scope of this book. However, `assertStatus` is right in the middle of the scope. We want to ensure our endpoint runs and returns the proper RESTful response code depending on what we ran.

To do this, we just need to chain the `→assertStatus(STATUS_CODE)` at the end of our test. For example, if we try to create a brew method as an admin, we want to assure that the request succeeds. We need to chain `→assertStatus(201)` to our test ensuring a resource has been created. If we submit a brew method as a general `user` we want to chain `→assertStatus(403)` to ensure their request was denied. The `assertStatus()` method simply checks the response code of the request.

Assert Database Has and Database Missing

The other common assertions I run is the `assertDatabaseHas()` or `assertDatabaseMissing()` methods. These do exactly what their method names state. They ensure that the database has a resource or is missing a resource. So when I submit a resource as an `admin` I can ensure the new resource is created in the database. This ensures the whole process was successful. The opposite if the user is a general user. We want to `assertDatabaseMissing()` so there was no error where the response code was correct, but the user got their request persisted.

To chain this assertion to the test, you have to identify the database table you are checking as the first parameter, and then an array mapping the column name to the data you want to see or not. Say I sent a request as an admin to save a `Pour Over` brew method. I'd run:

```
$this->assertDatabaseHas('brew_methods', [  
    'method' => 'Pour Over'  
]);
```

The method will assert that the `brew_methods` table contains a record with the `method` column equal to `Pour Over`. The same goes with `assertDatabaseMissing()` if you want to ensure a certain table does NOT contain a record matching what you pass in.

Assert JSON, Assert JSON Count, and Assert Exact JSON

These three assertions are used primarily with `GET` requests. You can assert that what is returned from the database is exactly what you want. Usually, when testing a `GET` endpoint, you would set up your database with your resources in the `setUp()` method in your test class. You could then assert that by running certain requests against your `GET` endpoint you get the proper response.

Let's say I created 5 brew methods and want to ensure that when I send a JSON `GET` request to `/api/v1/brew-methods` I get 5 back. I'd chain `→assertJsonCount(5)` to the end of my test to ensure it passes. If you want to ensure that a single brew method you created was returned, you'd run `→assertJson([{DATA_ARRAY}])` with the data array matching the fields expected in the response for the brew method. If you use `→assertExactJson([{DATA_ARRAY}])`, then the response has to be exactly what you specify.

I tend to use `→assertJson()` more since than `→assertExactJson()` since I have some dynamic tests. Just a fair warning, these can get pretty complex really quickly. Check out "[Testing JSON APIs](#)" in the documentation for a much more in-depth look at these methods and a few more that I don't use as much.

Test Method Naming

Before we go through some examples, I'd like to point out method names for tests are very different than names you'd tend to have in your code. Method names for tests should be as verbose as the test it self. You will often see camel cased sentences for test names. This is correct. You want the name to be as explanatory as possible so it's easy to track down failing tests and understand what they should do. For example, I have `testCreateBrewMethodAdmin()` which, as you'd expect, will test the creation of a brew method resource as an admin user.

The other SUPER important thing to point out about testing methods is they MUST begin with `test`. Just like your testing classes must end with `Test.php`, the methods within these classes have to begin with `test`.

Running a Group of Tests With Annotations and Filters

Another helpful tip for designing your tests is to annotate which `@group` they are a part of. As your app grows, you will tend to have exponentially more tests. If you write one test, you don't want to have to run every single test to make sure that one passes. You should only run every single test before deploying because it can take minutes to run.

First, annotate your method with `@group {name}` :

```
/**
```

```
* Test the creation of a brew method as an admin
* @group brew_methods
*/
public function testCreateBrewMethodAdmin()
{
    $icon = UploadedFile::fake()->image('brew_method.jpg');

    $this->actingAs( $this->admin )
        ->json( 'POST', '/api/v1/brew-methods', [
            'method' => 'Pour Over',
            'icon' => $icon
        ])
        ->assertStatus( 201 );

    $this->assertDatabaseHas('brew_methods', [
        'method' => 'Pour Over'
    ]);
}
```

Then you can filter your tests that run when you run:

```
./vendor/bin/phpunit --group brew_methods
```

Now you are only running a group of methods instead of all of them. You can also filter by method name if you only want to run a single test. To do that, run:

```
./vendor/bin/phpunit --filter testCreateBrewMethodAdmin
```

Now you will ONLY run the method to create a brew method as an admin. This is extremely helpful as you work on adding individual tests to your application.

Debugging Responses

Before we jump into a few example tests, the last trick I want to touch on is

viewing of a response. Sometimes you get an error on your request. When testing, this is a good thing! You want to get errors. However, the errors will be like "status 500 does not match status 201" and that's it.

To get more information on what went wrong, you can edit your test to print out the response which will be a detailed error to your terminal. To do this, adjust your test to look like:

```
/**  
 * Test the creation of a brew method as an admin  
 * @group brew_methods  
 */  
public function testCreateBrewMethodAdmin()  
{  
    $icon = UploadedFile::fake()->image('brew_method.jpg');  
  
    $response = $this->actingAs( $this->admin )  
        ->json( 'POST', '/api/v1/brew-methods', [  
            'method' => 'Pour Over',  
            'icon' => $icon  
        ]);  
  
    $response->dump();  
  
    $this->assertDatabaseHas('brew_methods', [  
        'method' => 'Pour Over'  
    ]);  
}
```

Then run the individual test:

```
./vendor/bin/phpunit --filter testCreateBrewMethodAdmin
```

You can now view the error response from your API and make the changes necessary to ensure the code works as expected! Now, let's jump into a few

example tests.

Example Tests

There are literally infinite amount of tests you could run, you can check out the repo in GitLab if you want to see all of the tests for ROAST. However, here are a few samples.

Ensure GET Response Is Correct

Scenario: We want to ensure that the `/api/v1/brew-methods` endpoint returns a `200` response code.

```
/**  
 * Test load all brew methods as a general user  
 * @group brew_methods  
 */  
public function testLoadAllBrewMethodsUser()  
{  
    $this->actingAs( $this->user )  
        ->json( 'GET', '/api/v1/brew-methods' )  
        ->assertStatus( 200 );  
}
```

Uploading A File

Scenario: As an admin I want to create a new Brew Method. I will need to upload a file and ensure the validations pass. My expected response is a `201`.

```
/**  
 * Test the creation of a brew method as an admin  
 * @group brew_methods  
 */  
public function testCreateBrewMethodAdmin()
```

```
{  
    $icon = UploadedFile::fake()->image('brew_method.jpg');  
  
    $this->actingAs( $this->admin )  
        ->json( 'POST', '/api/v1/brew-methods', [  
            'method' => 'Pour Over',  
            'icon' => $icon  
        ] )  
        ->assertStatus( 201 );  
}  
}
```

Database Has

Scenario: As an admin, I want to update the name of my brew method to **Pour Over**. I should get a response of **204** showing the correct status and see in the database that a brew method with the name **Pour Over** was created.

```
/**  
 * Test the update of a brew method as an admin  
 * @group brew_methods  
 */  
public function testUpdateBrewMethodAdmin()  
{  
    $this->actingAs( $this->admin )  
        ->json( 'PUT', '/api/v1/brew-methods/' . $this->brewMethod->id, [  
            'method' => 'Pour Over'  
        ] )  
        ->assertStatus( 204 );  
  
    $this->assertDatabaseHas('brew_methods', [  
        'id' => $this->brewMethod->id,  
        'method' => 'Pour Over'  
    ]);  
}
```

```
}
```

Assert JSON Count

Scenario: I created 5 brew methods. When I load all of them, I should expect to receive 5.

```
/**  
 * Test load all brew methods as an admin  
 * @group brew_methods  
 */  
public function testLoadAllBrewMethodsAdmin()  
{  
    $this->actingAs( $this->owner )  
        ->json( 'GET', '/api/v1/brew-methods' )  
        ->assertStatus( 200 )  
        ->assertJsonCount(5);  
}
```

Soft Deletion

Scenario: As an admin, I don't want a specific brew method any more. When I submit a **DELETE** request to the endpoint, I should ensure the brew method is soft deleted.

```
/**  
 * Test the deletion of a brew method as an admin  
 * @group brew_methods  
 */  
public function testDeleteBrewMethodAdmin()  
{  
    $this->actingAs( $this->admin )  
        ->json( 'DELETE', '/api/v1/brew-methods/' .
```

```
$this->brewMethod->id)
    ->assertStatus( 204 );

$this->assertSoftDeleted('brew_methods', [
    'id' => $this->brewMethod->id
]);
}
```

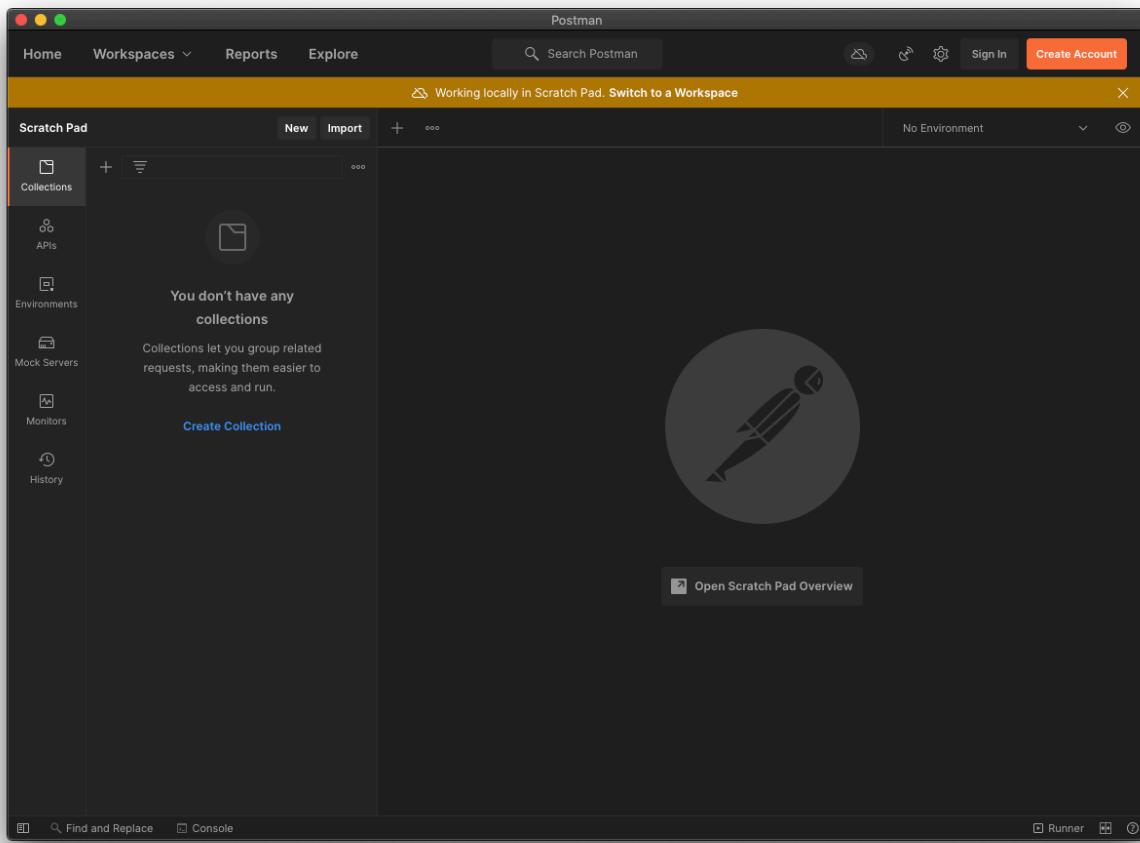
Configure Postman REST Client

When building anything, having the right tools for the job is essential. When it comes to building an API, there are two beautiful tools that aid in the process, Postman and Insomnia. Both are called REST clients that enable you to create collections, store these collections and consume your API for debugging purposes. They both make requests easy to debug and view.

Using a REST client is different than running API tests. The purpose of these clients is to debug your code and see visually what's going on. They allow you to inspect responses and headers with ease.

In this tutorial we will go through Postman and in the next, we will touch on Insomnia. For basic purposes, they are relatively the same. If you get into more advanced features and really like using a REST client, Postman offers a ton more features and a paid version. These advanced features include sharing workspaces with colleagues, building out documentation, etc. For now, just head over to <https://www.postman.com/> and download the REST client. Once you have it installed, you can create a free account or jump right to the client.

Once you open up the app, you should see an interface similar to:

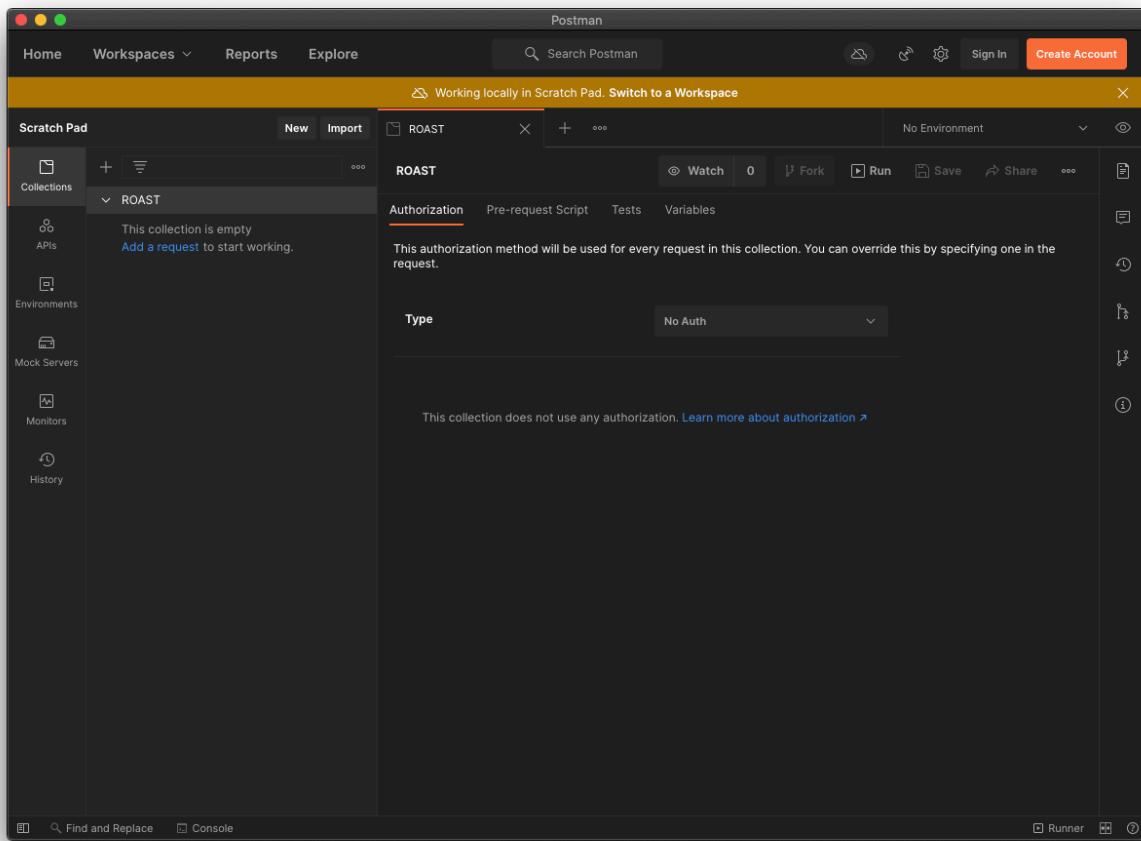


There are a ton of features within this app. Let's go through a few of the basics.

Creating Collections

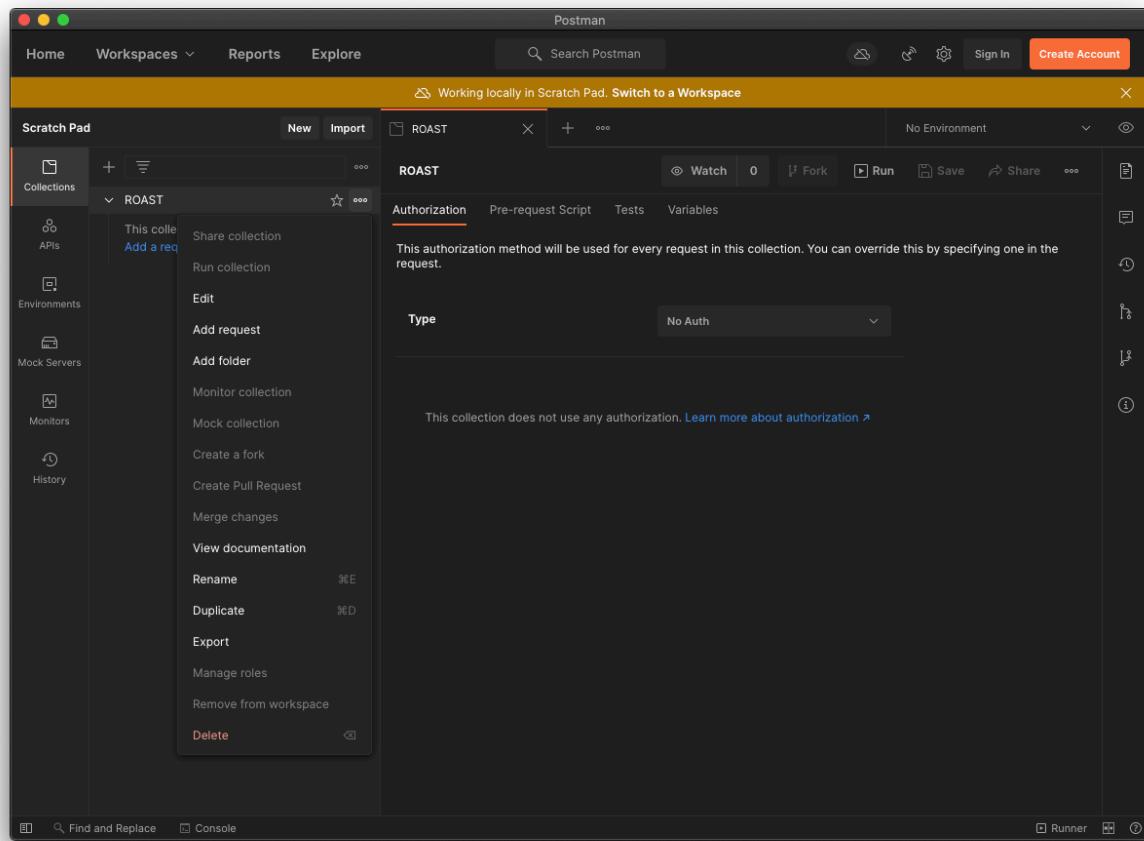
The first thing I do when beginning to test my API is to build a collection. If you look at the left side of the screen shot above, we are on the "Collections" tab and the first pane states we don't have any collections.

Collections are groups of requests similar to a file hierarchy. I do a collection per app. So for ROAST, let's create the following collection by clicking the blue "Create Collection" text:

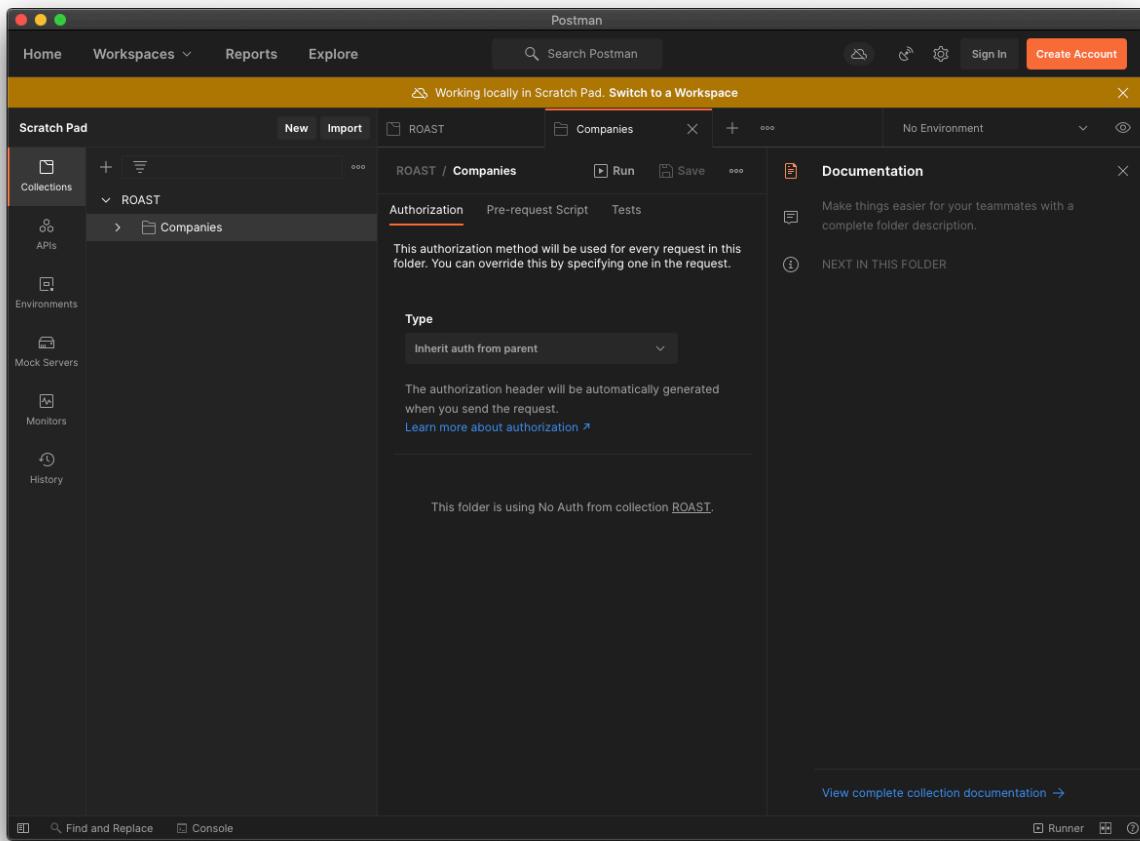


On top of the screen, adjust the name to be the name of the application. In this case, I added "ROAST". You also have the option of adding authorization headers and other scripts. Right now, we are not going to be setting up authorization. I'll go through that in the next step. We will be generating an access token and using that.

The next thing I do is begin to build out my resources. Within each collection are folders you can add. To do this, click the three dots on the ROAST listing on the left hand side and click "Add folder"



You will now get a screen to add a folder. The screen to add the folder looks like the one below:

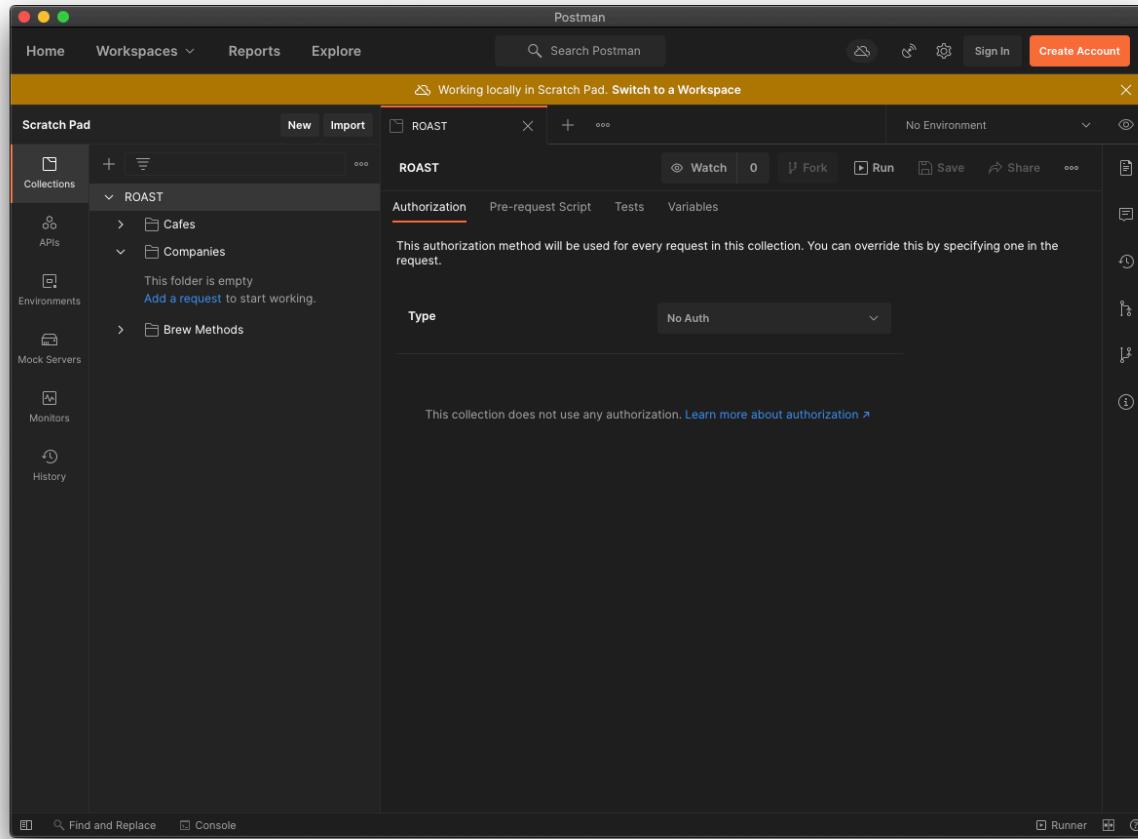


All I did was add the **Companies** resource as the name at the top. Now we have a nice folder for our company requests to live in. In the next step we will be setting up authorization. For now, you can go through and add a folder for all of the resources in your API, or you can jump to the authenticating requests section.

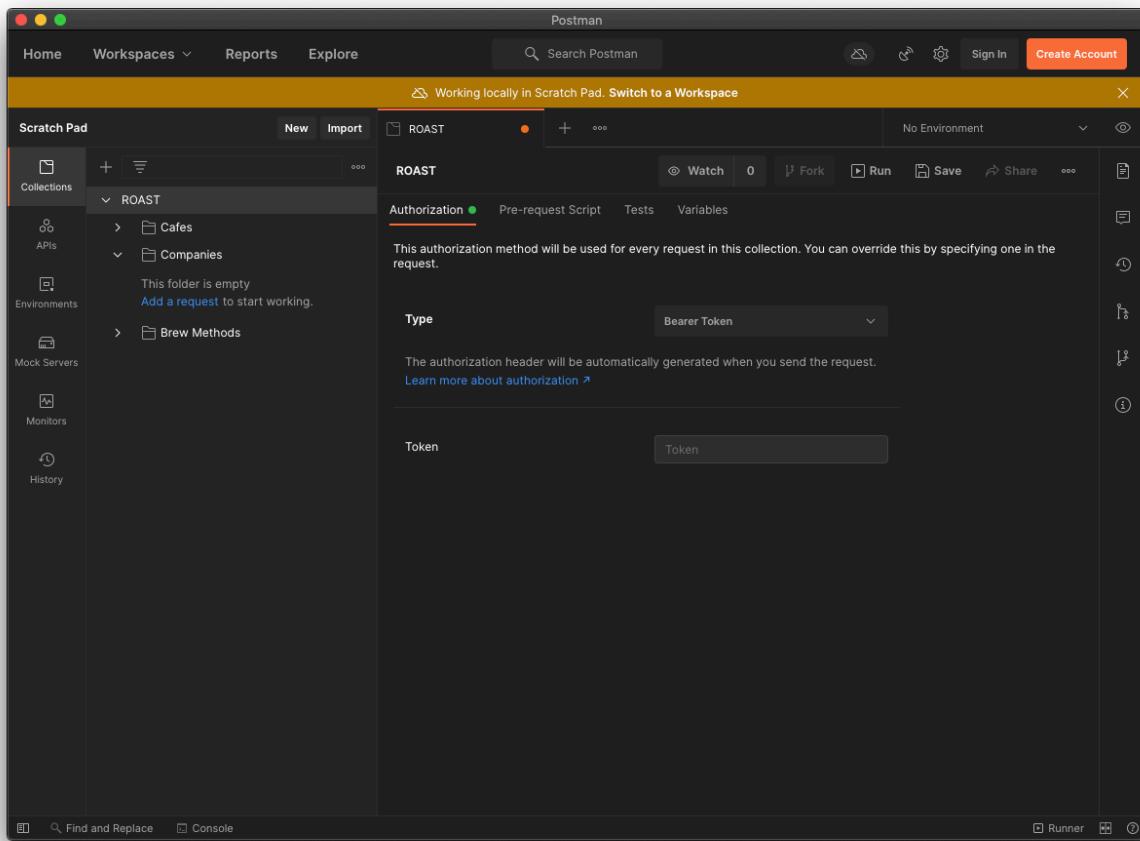
Authenticating Requests

Now that we have a collection and some of your resources' folders created, we can set up authorization. To do this, we need to generate ourselves a personal access token. So in the web, navigate to your frontend where you have the ability to manage your personal access tokens. In ROAST this is under: **/account/profile** and generate yourself a token. I named my token **Postman** so I can monitor its usage.

Navigate up to your collection, and click the "Authorization" sub tab:



Under the "Type" select, "Bearer Token". This will show a new field named "Token". Place your token you generated in this field and save (cmd + s or ctr + s works):



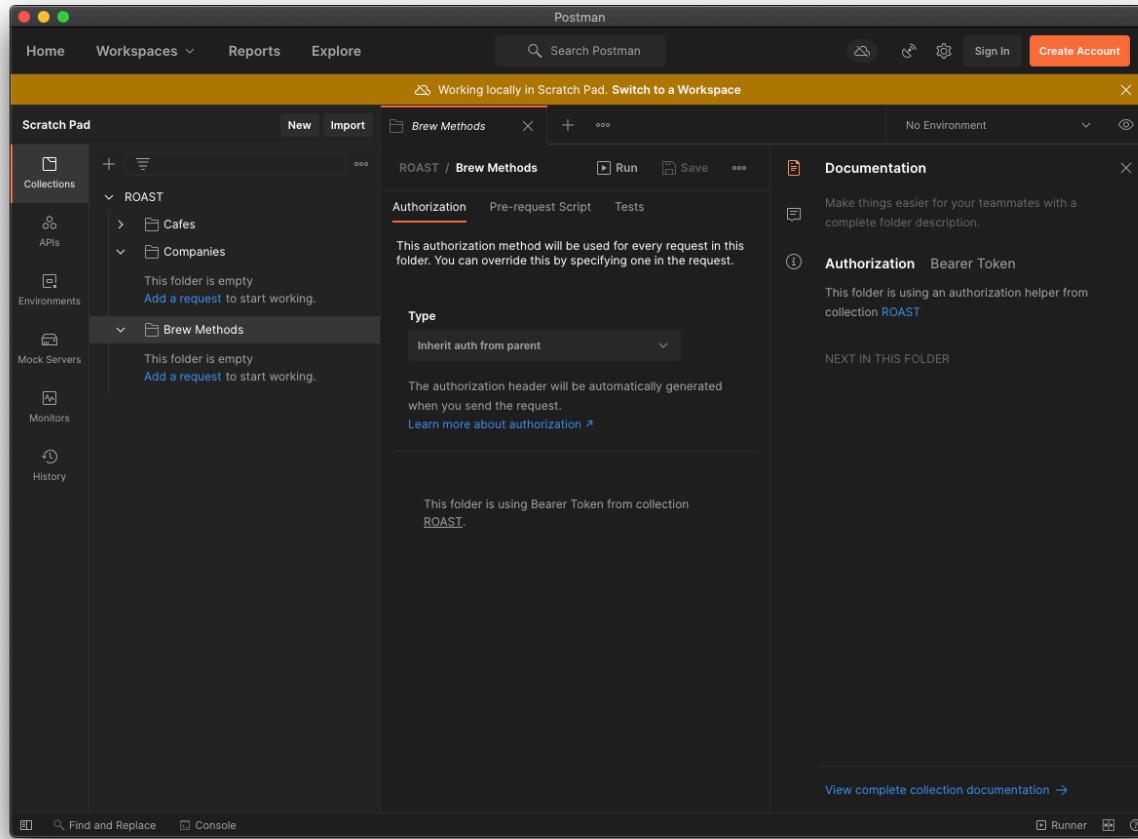
Now any request within that collection will append the `Authorization: Bearer {TOKEN}` header to the request! Remember, these are long lived tokens so you can run these for a long time. When the token expires, you can just generate a new one, apply it to the Authorization settings and resume your testing.

Let's get this working and send some requests!

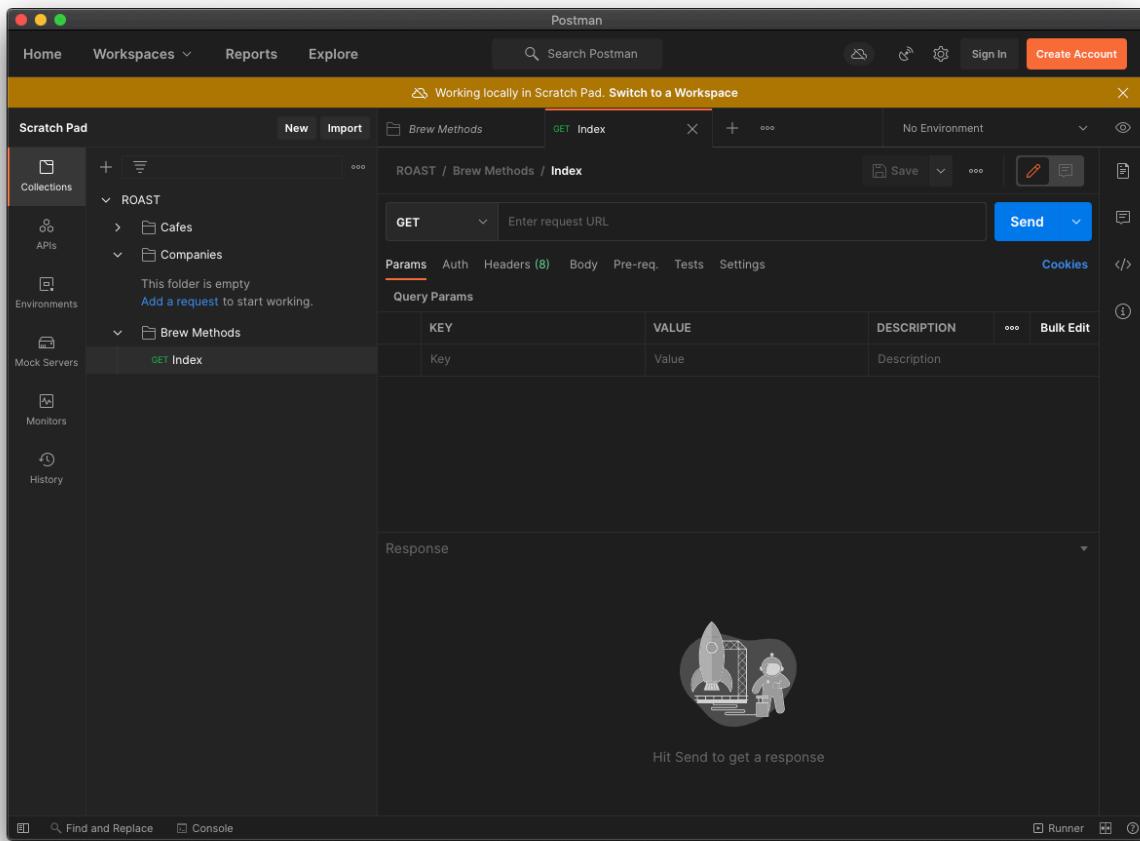
Sending Requests

For these examples we will be using the Brew Method resource. This will be simple and illustrate the power of Postman. What I did was first create a Brew Methods folder within the ROAST collection. Under the "Authorization" tab, I ensured "Inherit auth from parent" was selected so it applies our token to each

request. It will say "This folder is using Bearer Token from collection ROAST":



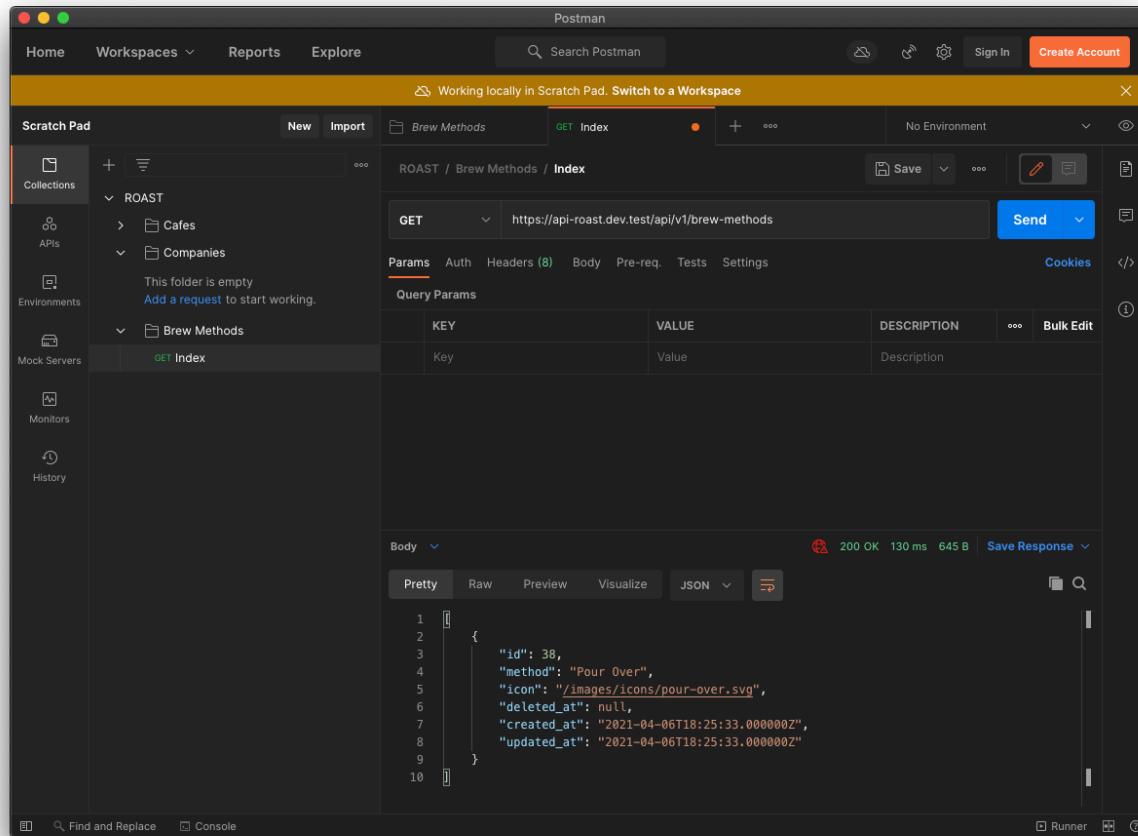
Let's start with a simple GET request to load all of our Brew Methods. First, under the "Brew Methods" folder, click "Add a request". A screen pops up where you can fill in everything you need regarding your request. I named it "Index" to match the naming schemes of my RESTful resource controllers:



Quick note, I will be using my development environment for URLs. You can use production if you want, just make sure to generate a Personal Access Token on that machine for your use. For my URL, I entered: <https://api-roast.dev.test/api/v1/brew-methods>. For more about our infrastructure check out <https://github.com/serversideup/>. All of our infrastructure is open source.

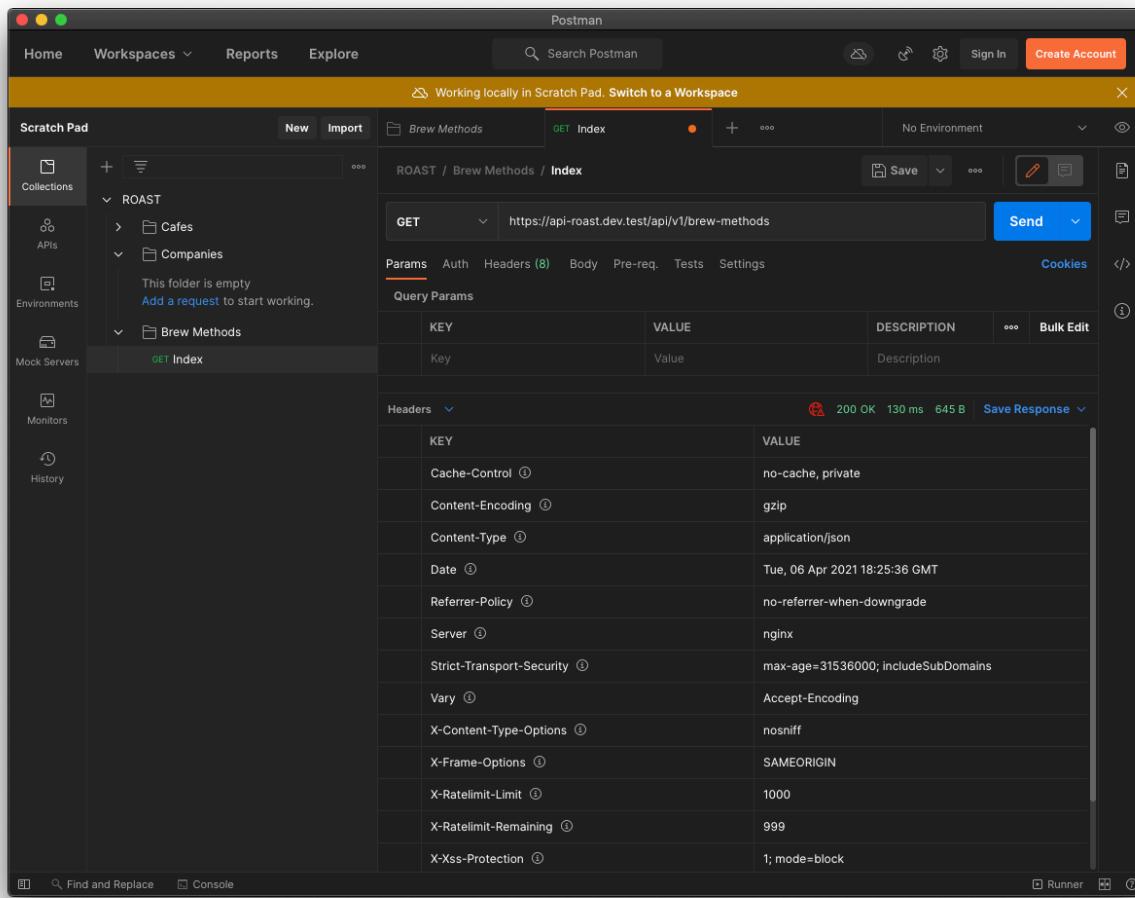
So right now, we have a nice `GET` request that hits our `/api/v1/brew-methods` endpoint. It's nested in our `Brew Methods` folder that's under our `ROAST` collection. Our `ROAST` collection has the authorization set up for our requests so the `Authorization: Bearer {token}` will be attached to each request. The permissions of that token will apply in our middleware and policies. If you need to add any query parameters to your request you can add the `key` and `value` pairs in the `Params` tab. These will be appended to your request so you can test a ton of different scenarios.

When you click "Send", you will see a "Body" drop down appear and the response from the request:

A screenshot of the Postman application interface. The top navigation bar includes 'Home', 'Workspaces', 'Reports', 'Explore', a search bar, and 'Sign In/Create Account'. The main workspace is titled 'Scratch Pad' and shows a tree view with 'ROAST' expanded, containing 'Cafes', 'Companies', and 'Brew Methods'. Under 'Brew Methods', there is a 'GET Index' request. The request details show it's a 'GET' method to 'https://api-roast.dev.test/api/v1/brew-methods'. The 'Params' tab is selected, showing a single parameter 'Key' with value 'Value'. Below the request details is a 'Body' section with tabs for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON'. The 'Pretty' tab is selected, displaying a JSON object with fields like id, method, icon, deleted_at, created_at, and updated_at. Above the body tabs, status information is shown: 200 OK, 130 ms, 645 B, and a 'Save Response' button.

Pretty sweet! You can click the tabs, "Pretty", "Raw", "Preview", etc. to see the data returned in different formats. Since we are building a JSON API, I'd suggest keeping the format as JSON. I also really enjoy the **Pretty** view since I can easily see the structure of the response.

Right above the tabs showing the different response visualizations, you can click the "Body" dropdown and view other aspects of the request. Another super helpful feature is the "Headers" view which allows you to view all of the headers on returned from the request:

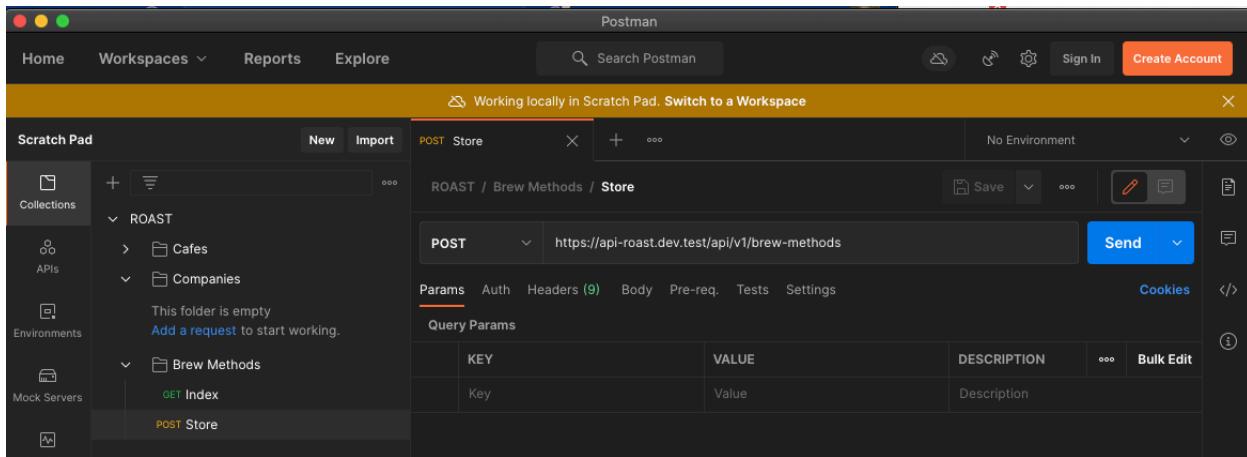


You can see our authorization header is working because we are limited to 1000 requests per minute instead of the 60 for an un-authenticated user. With all of this data, the tool is extremely powerful in breaking down your API requests and debugging what is returned.

POST Requests

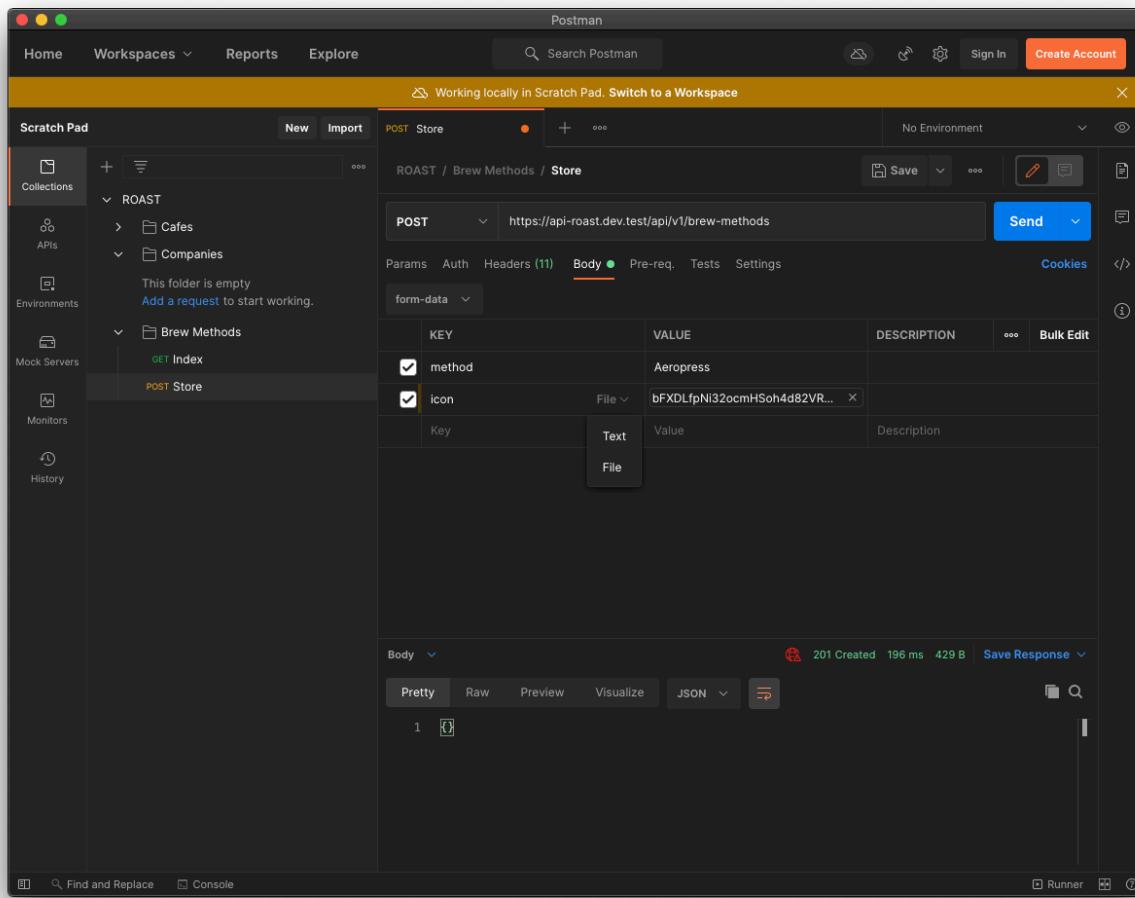
Let's say we wanted to send data to our `/api/v1/brew-methods` endpoint through `POST` to create a brew method. We can do that as well with Postman.

First, let's create a new Request named "Store" under our "Brew Methods" folder. It will hit the same URL, except it will come through `POST`:



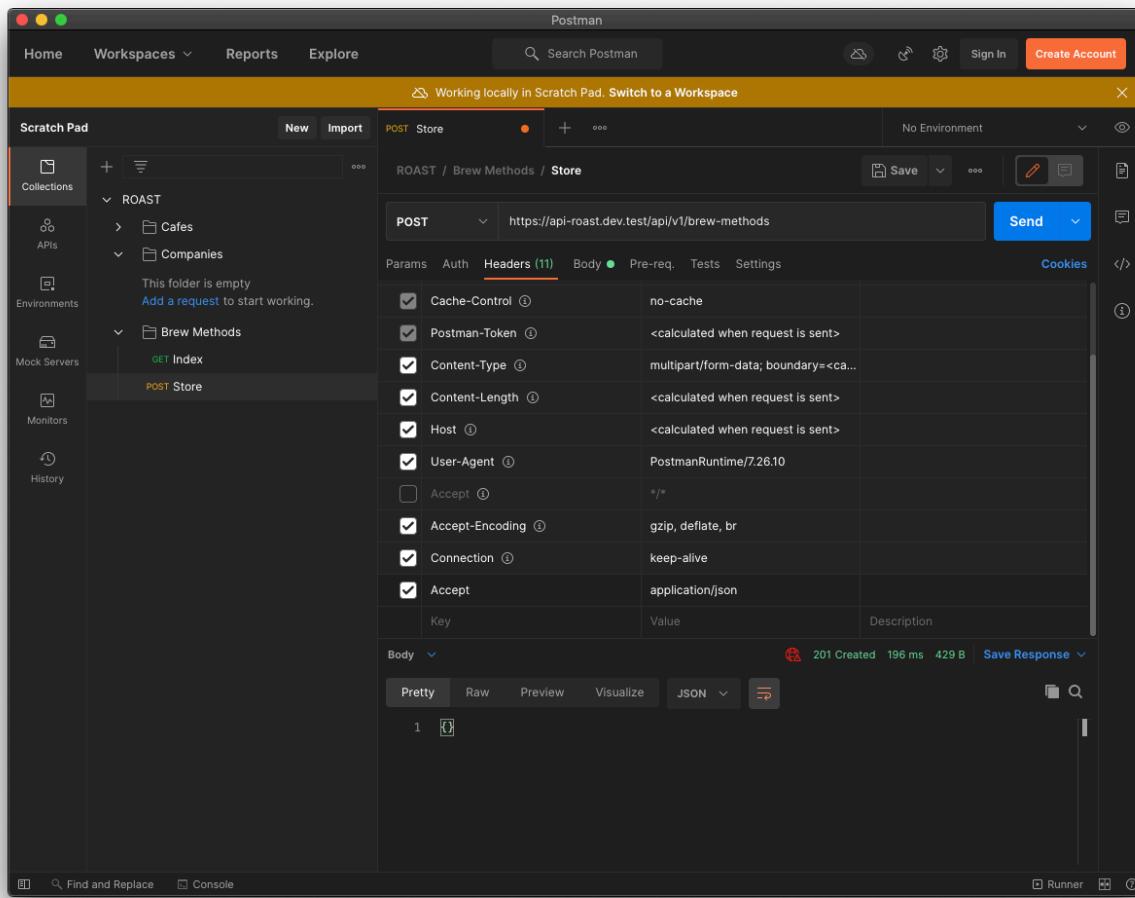
Now, right beneath the URL, we have another set of tabs. They are Params, Auth, Headers, Body, etc. Since we are sending a **POST** request, we aren't going to send our data through Params, instead we are going to send it through the Body so click "Body". When you get to the "Body" page, there will be a dropdown right below the tabs that says "none". We need to change this to "form-data".

When we create a brew method, we need to upload a file that goes along as the icon. You can even upload files within Postman to test your API! So once you select **form-data**, fill in the name of the required fields to create a brew method:



When you add a new field, you can tell Postman what type of field it is. Notice, the "icon" field, we selected "File" and the "value" column allows us to "Select Files" now.

There is only one more step before sending a **POST** (or **PUT/PATCH**) request and that's to adjust the header for what is accepted as a response. If you switch to the "Headers" tab, uncheck the "Accept" header that is added by default, then add the "Accept" header and set it to "application/json":

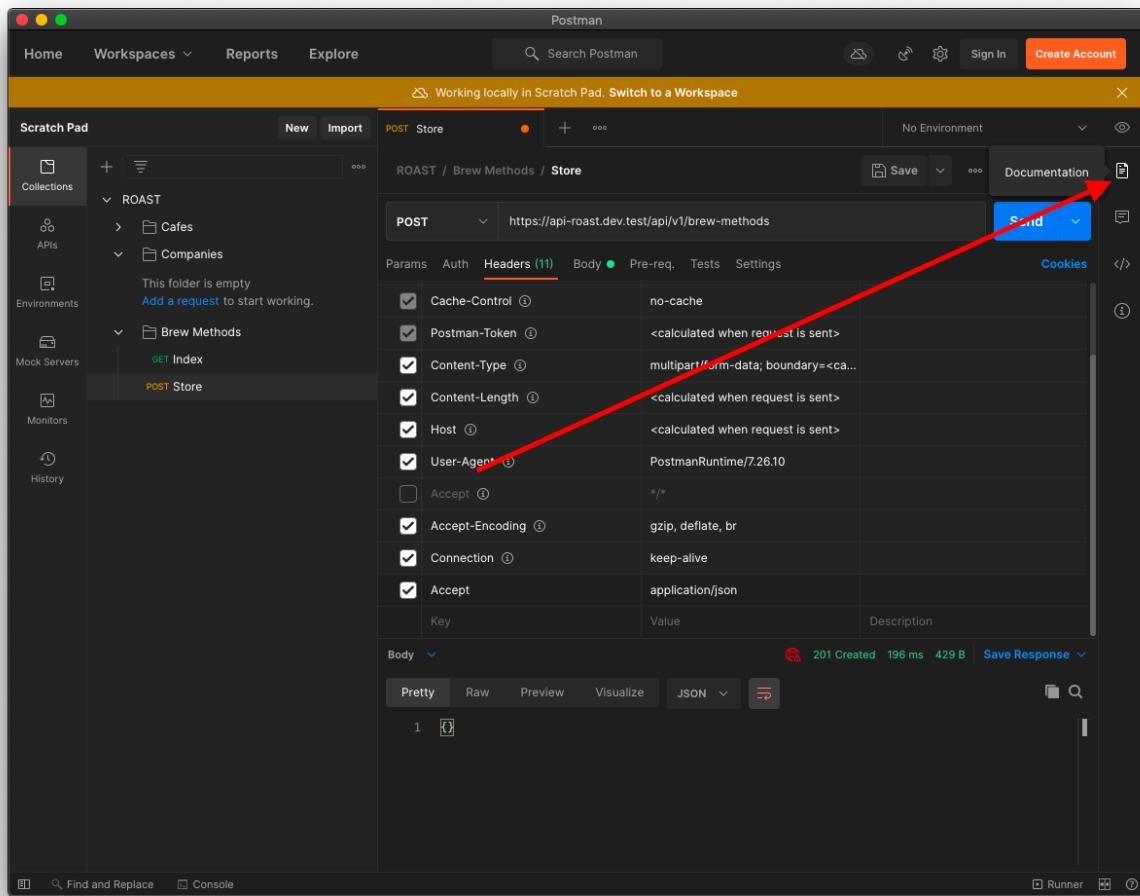


The reason we do that is to ensure we can view any validation errors that come up when creating or updating a resource.

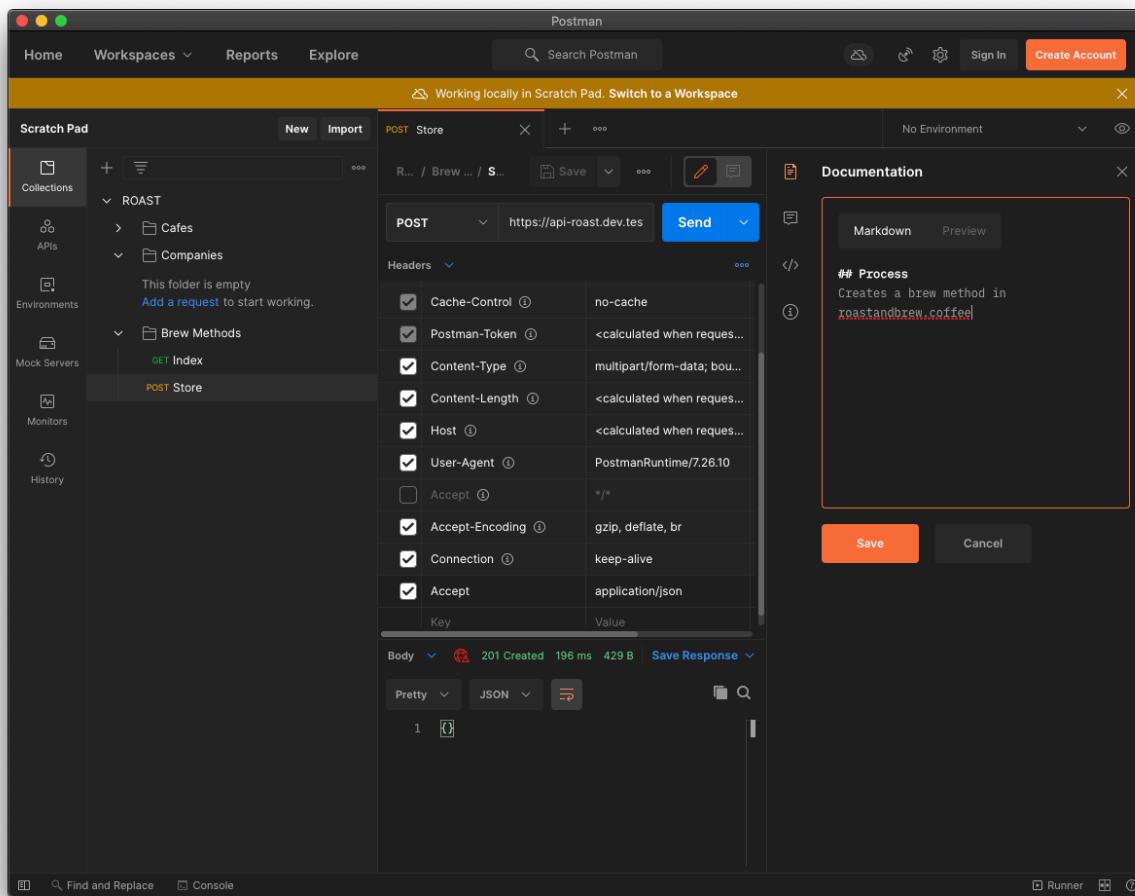
When done, just press "Send" and your new Brew Method will be created! You can see the **201 Created** response status code in the image above. You can go through all of your end points and add them to Postman so you can easily test!

Documentation

Another helpful feature as you get into using Postman is the documentation tab:



When you open this tab, you can actually document each endpoint so you know what you set up and share it with your team on a premium plan.



I find this extremely helpful so you can always come back to where you left off.

Conclusion

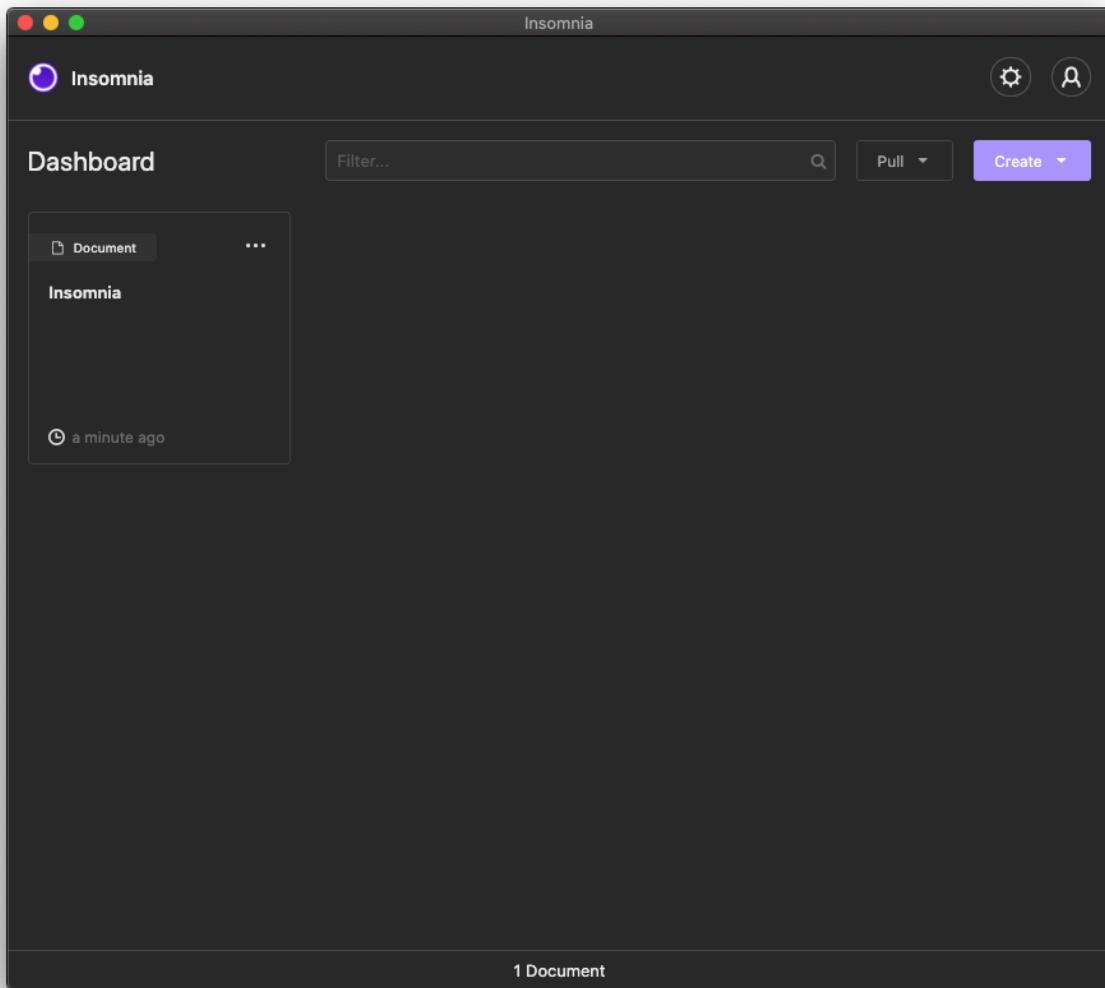
Postman has a huge variety of features and really makes developing your API much easier. No more inspecting network traffic, modifying and resending in the browser. A tool like Postman makes this process much easier and you can really organize your requests so they are easy to work with.

Next up, we will go through the same process with Insomnia, another REST client.

Configure Insomnia REST Client

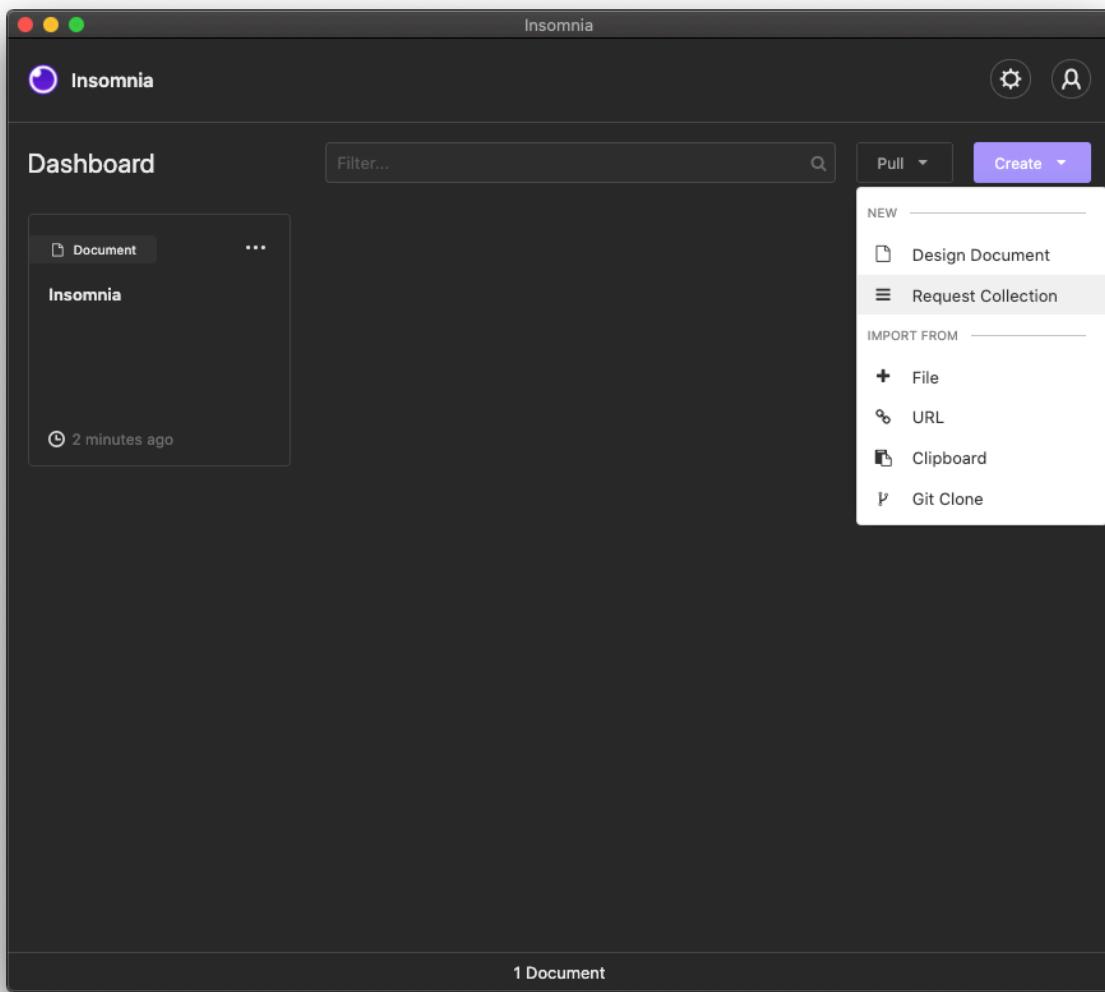
Like Postman, Insomnia (<https://insomnia.rest/>) is another REST client that makes building an API much easier! To be honest, I tend to use Insomnia over Postman. Postman has much more features, but Insomnia is cleaner, easier to get started, and you can extend it easily with plugins. I'll run through exactly what we did with Postman on Insomnia and see which one you prefer. If you have any questions regarding these tools, feel free to reach out on the [community](#) forum.

Once you have downloaded Insomnia and open up the application, you should see a dashboard like the one below:



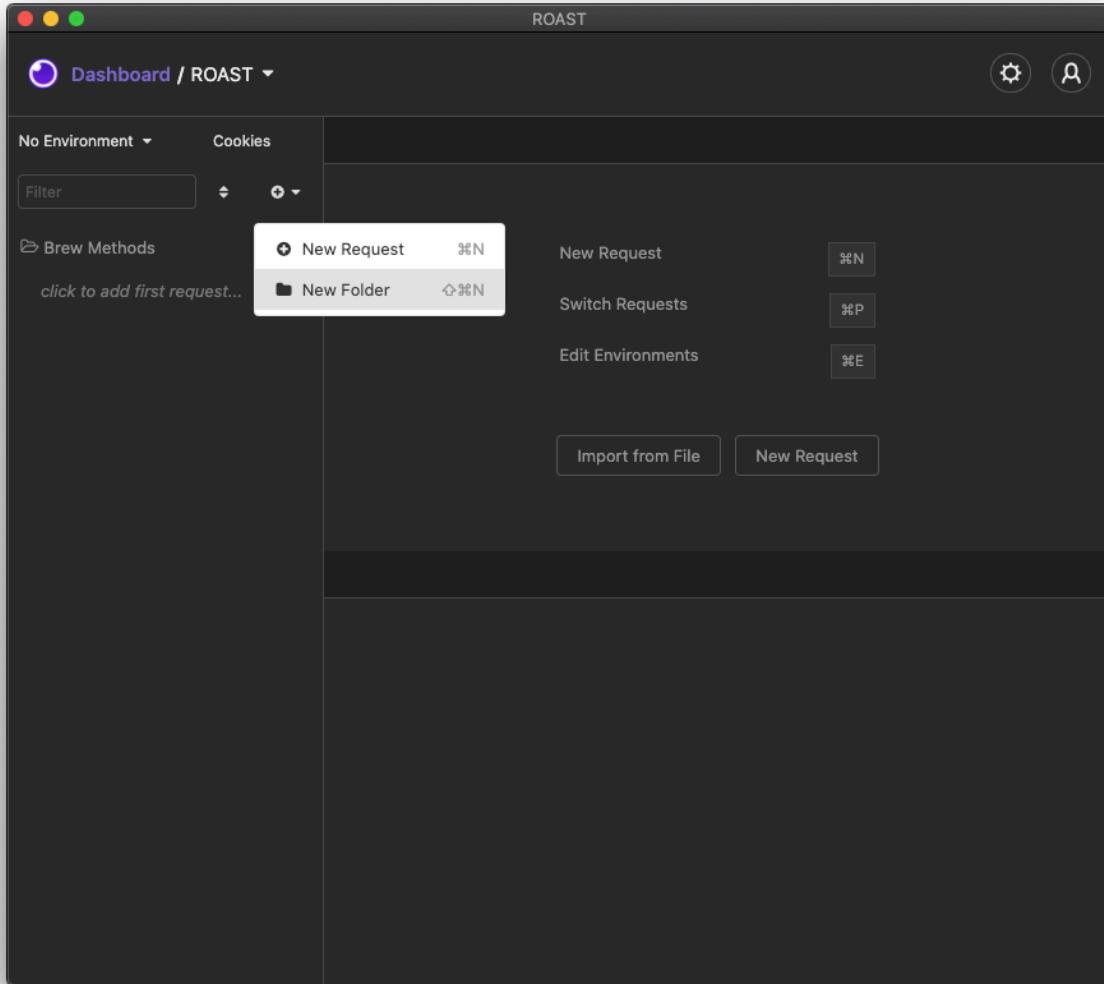
Creating Collections

The first step we will take is similar to Postman and we will create a "collection". This collection will house all of our endpoints that we are working with and keep them nice and organized. To create a collection, click the arrow next to "Create" in the top right and select "Request Collection":



I just named the request collection "ROAST" and you will be brought to a screen where you can create your folders and requests. We will then add a folder for all of our Brew Method requests. Click the small drop down near the "Filter" input and

select "New Folder":



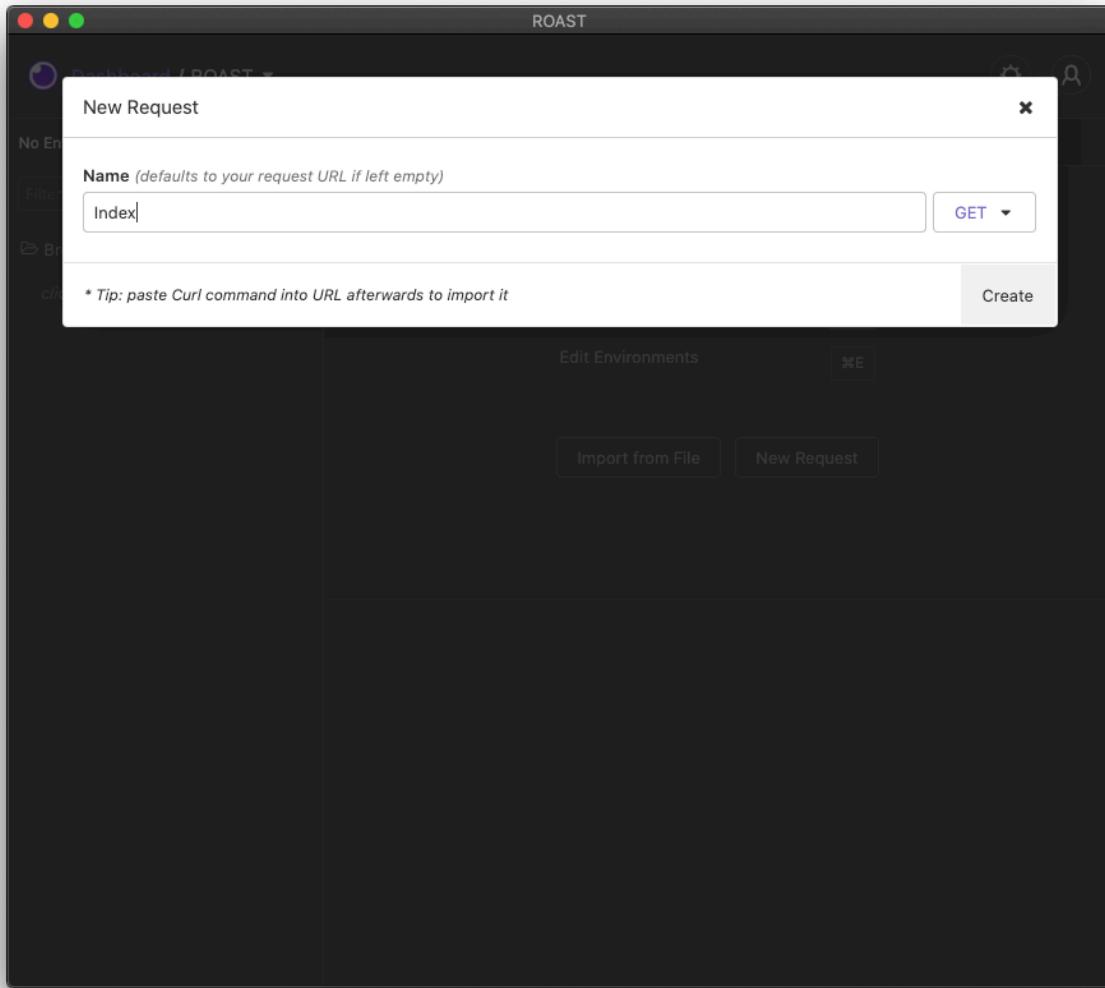
I named the folder "Brew Methods". Within this folder, I will be adding all of my requests similar to what we did with Postman. If you are going to be working with Insomnia, I'd recommend making a folder to house all the API calls to your resources. Then when you need to test a resource endpoint everything is structured easily and you can expand quickly.

Sending Requests

One feature that Insomnia is missing is applying an authorization header to groups of requests, so we will be doing this on a per request basis. For me, I don't mind it

so much because I like that fine grained control and makes the requests more organized. But everyone has their own opinion. Let's add a request to load our Brew Methods from the `/api/v1/brew-methods` endpoint.

To do this click "click to add first request..." and a modal will pop up. In this modal enter the name "Index" and keep `GET` selected:

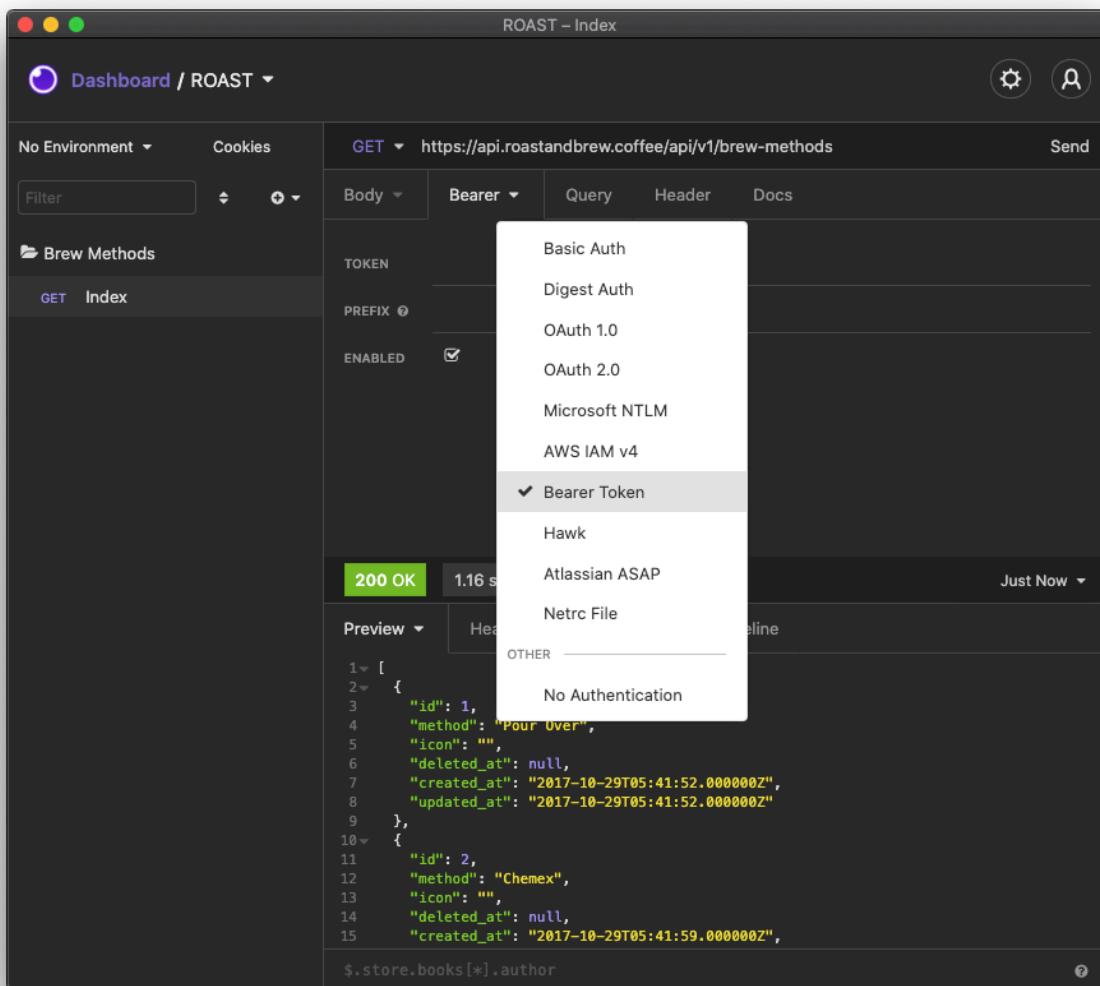


You can now set up your request. The top of the screen has the URL you can enter which will be the endpoint we want to hit. Below that top bar will be "Body", "Auth", "Query", "Header", and "Docs" tabs. For this request, we will be adding the authorization header (even though it's a public URL), just to show how it's done. Let's quickly add our URL to the top which is the API endpoint `/api/v1/brew-`

methods.

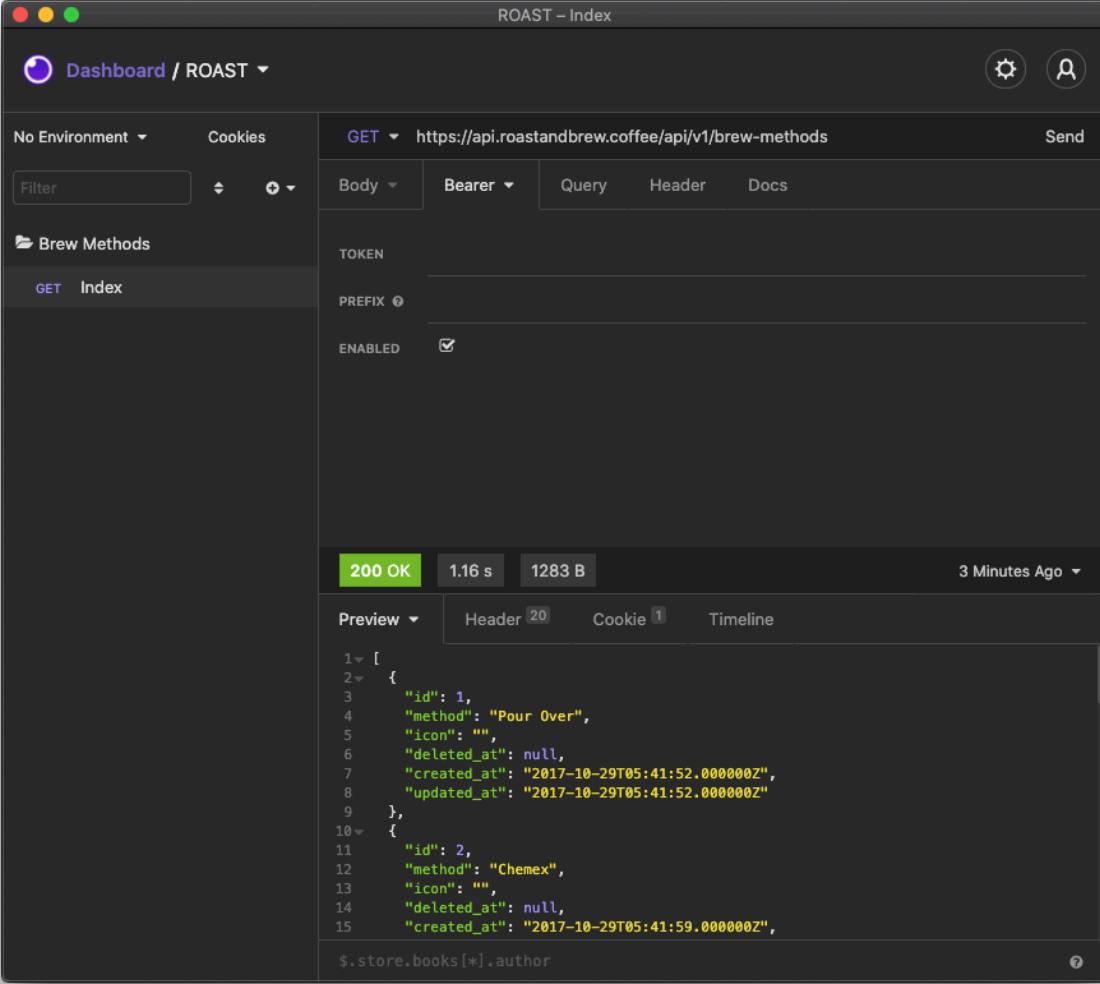
Before adding the header I'm going to quickly generate an "Insomnia" token that I can use as the "Bearer" token. This just allows me to keep track of the usage. Where ever your app allows you to generate a personal access token, go there and create one. In ROAST it's under [/account/profile](#).

Now, let's switch to the "Auth" tab within Insomnia. You then need to click the arrow dropdown and select "Bearer Token":



Next you need to paste your token you just generated into the "Token" field and ensure that "Enabled" is checked. You are now good to go!

Now that we have everything set up, we can just click "Send" in the top right. You should see the response data below with a green "200 OK" meaning the request went through! If you click the arrow down next to "Preview" you can view the raw response as well.



The screenshot shows the Insomnia REST Client interface. At the top, it displays a GET request to `https://api.roastandbrew.coffee/api/v1/brew-methods`. The response status is **200 OK**, and the time taken is 1.16 s. The response body is a JSON array containing two objects:

```
1 [  
2 {  
3   "id": 1,  
4   "method": "Pour Over",  
5   "icon": "",  
6   "deleted_at": null,  
7   "created_at": "2017-10-29T05:41:52.000000Z",  
8   "updated_at": "2017-10-29T05:41:52.000000Z"  
9 },  
10 {  
11   "id": 2,  
12   "method": "Chemex",  
13   "icon": "",  
14   "deleted_at": null,  
15   "created_at": "2017-10-29T05:41:59.000000Z",  
16 }]
```

You can also click the "Header" tab to view headers:

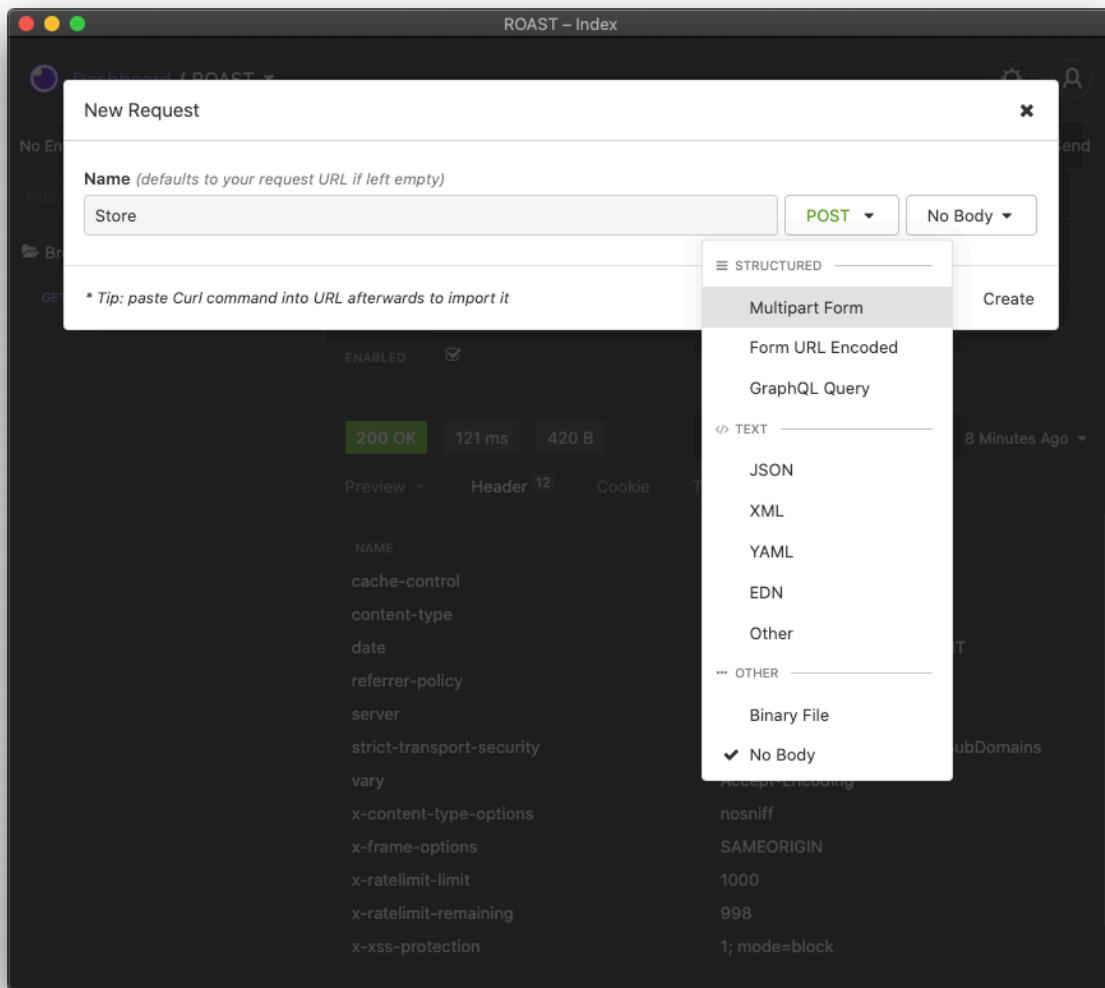
NAME	VALUE
cache-control	no-cache, private
content-type	application/json
date	Tue, 06 Apr 2021 21:38:18 GMT
referrer-policy	no-referrer-when-downgrade
server	nginx
strict-transport-security	max-age=31536000; includeSubDomains
vary	Accept-Encoding
x-content-type-options	nosniff
x-frame-options	SAMEORIGIN
x-ratelimit-limit	1000
x-ratelimit-remaining	998
x-xss-protection	1; mode=block

It's nice and clean and you can easily manage your requests for testing. Next up, let's send some data to our API through Insomnia.

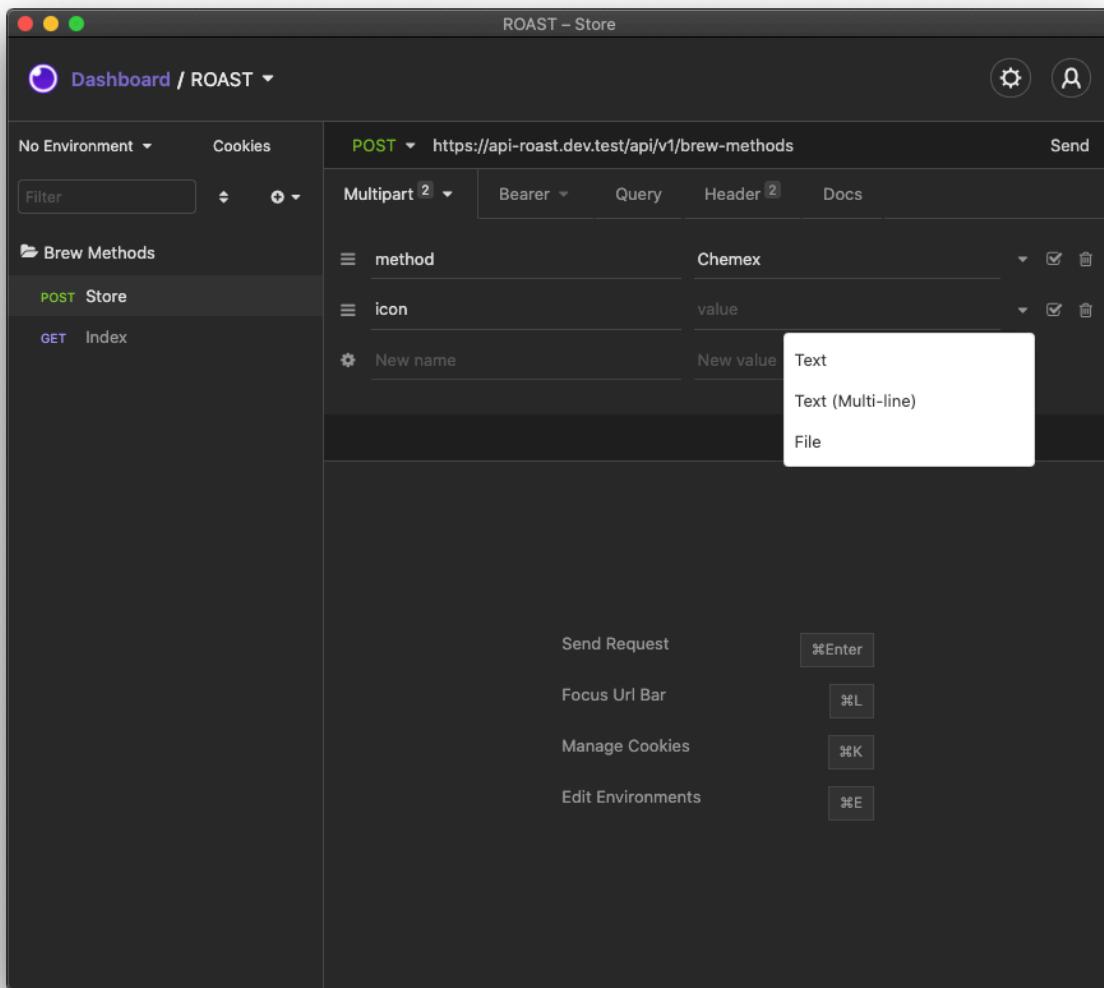
POST Requests

So we sent a **GET** request with Insomnia, let's do a **POST** request and send some data to the server.

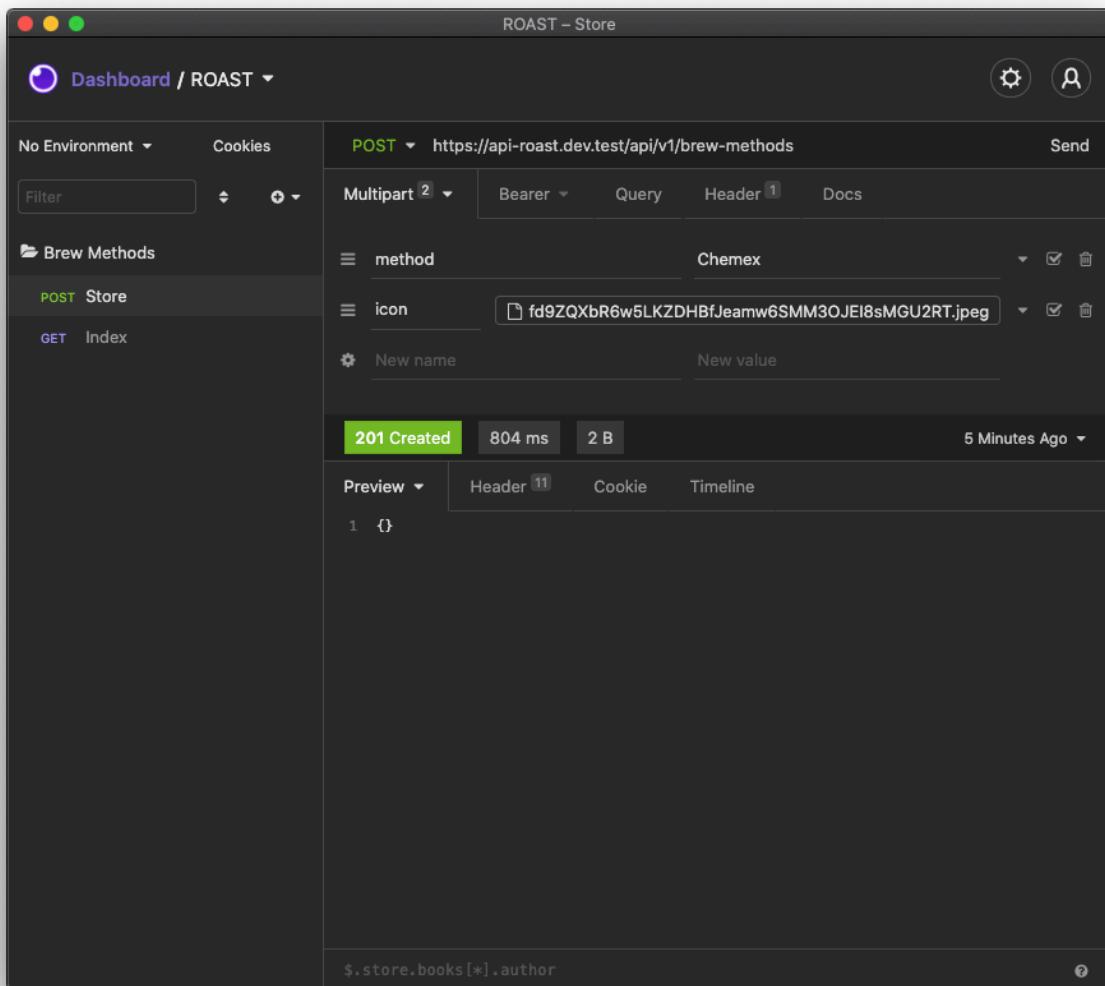
First, let's create our **POST** request that will go to the `/api/v1/brew-methods` endpoint and name it **Store** to match our RESTful resources. Remember, when we are sending data that needs to have an image (such as a Brew Method's icon), we need to have the data set up as "Multipart Form" so select that as the body when changing to a "POST" request:



Right away, go to the Auth tab of your new request and add your token. This is definitely needed for the POST request. After you have your token added, switch back to the first tab, this time labeled "Multipart". In that tab, add a name of "method" and a value of "Chemex". Below that, add "icon". When you look to the right of the "value" input, there will be a dropdown. Click this and select "File":



You can now select your file that you want for the icon. When you have the image selected, just click "Send" in the top right and you are good to go! You should get a green "201 Created" response and your brew method will be there!

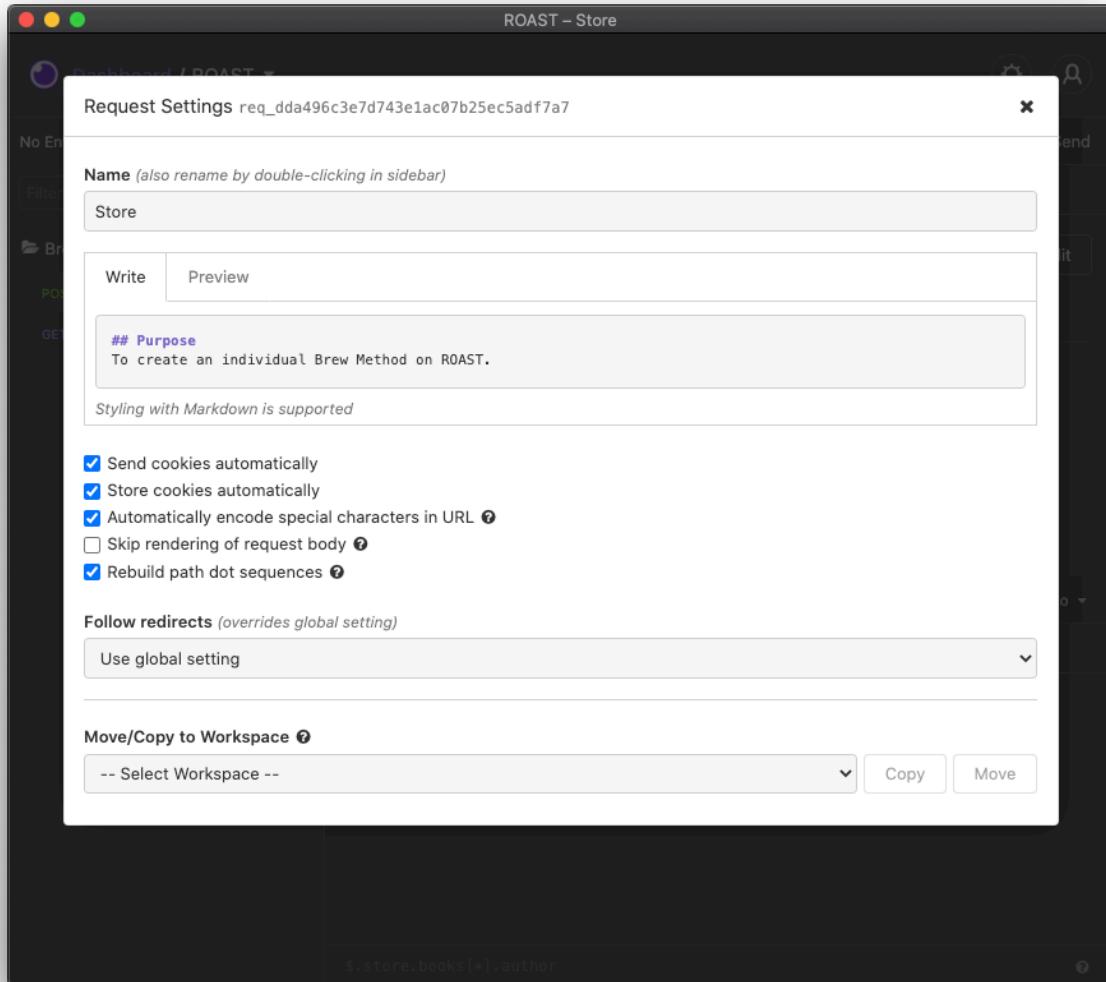


The above process is the same for all requests, just make sure you have the proper HTTP Verb set (**POST**, **PUT**, **PATCH**, **GET**, etc). What I like about Insomnia is being on a small dev team, it has the capabilities to share resources, but it's not as overwhelming as Postman. Postman is extremely powerful, but Insomnia is less overwhelming. On top of that, Insomnia is open source and I believe their plugin system will add some serious benefit. In all honesty, choose what works best for your team.

Documentation

Like Postman, Insomnia supports documenting your API endpoints as well. All you

have to do is click the "Docs" tab on the request and "Add Description". From here you will get a window that pops up and you can write your documentation for the request:



Markdown is supported just like in Postman.

Conclusion

When it comes to the decision to use either Insomnia or Postman, just choose whatever is best for your team. If you like less complexity, Insomnia is for you. If you want an enterprise level suite of tools, Postman would be your best bet. Like I mentioned, I prefer Insomnia simply because it's easier to get started and being a

small dev team of 2 people, it's easier to manage.

If you have any questions about using either of these tools, reach out on the [community forum](#) and I'd be glad to help!

Securing Sensitive Data

I've touched on this slightly but really want to hammer home the fact that opening up your API will expose a lot of the inner workings of your app. This is a double edged sword so be careful. You can apply all of the middleware, permissions, and security you want, but if you forget to protect a database column key that exposes sensitive data about your users, all of the security you put in place will have gone to waste.

This is a short tutorial on what to think about before creating an API endpoint, but it's crucial.

Plan Your Data

Before opening up any endpoint, you should plan your data and what you want exposed. Sounds simple right? Simple, yes, but extremely important. Say you have notes that are added to a User resource where a user can store private notes. This is done through a database column named `notes` that lives on the User model. Well if I'm doing a public search for users of the app I want to connect with, I get a response of users from an endpoint like `/api/v1/users?name=Dan`. In this response I get all users named "Dan" but I also get all of the `notes` field if not properly configured.

This could leak sensitive data. If you are doing a monolithic app, you aren't worried about this because the page rendering takes place server side and you just get pre-rendered HTML in return. With an API driven application, this full response is returned and you can see it in the network tab of your console or if you are a third part, right in your face with the response. You have to make sure you aren't returning this information.

Other fields such as latitude, longitude, and addresses may be essential to hide as well. Any private information you don't want publicly exposed.

Securing Your Data

There are two ways to secure your data. First, is to make sure it's not returned at all. This is done by flagging the field as "hidden" on your model. To do this, ensure that your field is in the `protected $hidden` array on the Model. If you open up `app\Models\User.php` in ROAST, look at the top of the model:

```
/**
 * The attributes that should be hidden for arrays.
 *
 * @var array
 */
protected $hidden = [
    'password', 'remember_token',
];
```

When returned, the `password` and `remember_token` will not be publicly available. In our example above, we should add the `notes` column like this:

```
/**
 * The attributes that should be hidden for arrays.
 *
 * @var array
 */
protected $hidden = [
    'password', 'remember_token', 'notes'
];
```

The next option is to filter your requests through a policy. This can check relationships and access on what data can be returned. If you look at the `app\Services\Cafes\CreateCafe.php` file, we use:

```
if( Auth::guard('sanctum')->user()->can('createCafe',
    $this->company ) ){
```

That line checks to see if the user has permission to create a cafe. However, you can use these permissions for retrieving information as well. In our

`app\Services\Cafes\SearchCafes.php` service, we break up a request modularly to keep our code maintainable. We can create more requests like this and check to see if the user has permission to view certain data. So maybe the user has permission to see private notes and we have a way to break up that endpoint into smaller pieces. Instead of 100% hiding the `notes` every time, we could do something like:

```
$query = User::query();

if( Auth::guard('sanctum')->user()->can('viewPrivateNotes',
$user ) ){
    $query->select('name', 'notes');
} else{
    $query->select('name');
}
```

Now we can create a policy that allows the user to view certain pieces of information. Quick warning! The two ways of hiding the `notes` field do NOT work together. The `protected $hidden` way ensures that when the model is serialized and sent back, the field will be removed. That means even if the user has permission, the field will be gone.

With the second method, it's more code, but it's also way more flexible. I'd choose what works best based on whether you want the data to be 100% not available (`$hidden` array) or sometimes available (segmented queries with policies).

Conclusion

It's extremely important to check yourself with what data is being returned. Sometimes the biggest security vulnerability is yourself. You can have an amazingly bullet proof system of middleware, policies, validations, but leak sensitive data just by not hiding it correctly. In doubt, hide the data. It's easier to

have that in place from the get go instead of coming back to patch it up later.

SINGLE-PAGE APPLICATION TIPS, TRICKS, AND GOTCHAS

When to Use Vuex

I believe that Vuex is one of the most important and useful pieces to any VueJS app. Whether that app is a single page application or a monolith of some sorts. However, are always questions on when you should implement Vuex and not.

Quick Refresher on Vuex

Vuex is a state management system used in your app. With, Vuex you get a standardized way to store and retrieve values from a centralized data store. In our application, the user is stored in state so we can reference that object throughout our entire app. Using Vuex, you won't end up in a situation where you are trying to pass a ton of props to a child component and attempting to keep it all in sync. Essentially, Vuex allows you to have reactive data that multiple child components can read and set allowing you to keep your data in sync.

When to Implement a Vuex Store

Using Vuex is extremely helpful when you are building a single page app and want global data, extremely large and complex page, or splitting up a form into separate components. You won't have to pass all of your local data via props down to other components and try to keep the data in-sync.

To be honest, I tend to implement Vuex way more than I probably should. I find the functionality that it adds to be extremely useful and allows me to break up complex systems into small components.

Let's take a look at the three places I discussed. This is obviously not an exhaustive list, but these areas I tend to use Vuex the most.

SPA Global Data

Using Vuex to store global data in a Single-Page Application was its intended use-case. By using Vuex in an SPA, you can store data that is present across all of your pages. We store our authenticated user this way. On any page, we can reference the user that is authenticated without having to pass down a piece of data to every page that needs it.

We also store our active filters for searching cafes and companies within Vuex. This allows us to search on our home page and navigate to our search result and map page with the filters activated. Any adjustments from the components on either of these pages allows us reference the global state and share the selection from the user.

Large Form

Similar to a complex page, a larger form that helps compute data for the user is a good candidate for a Vuex store. Breaking your form into smaller pieces can be extremely helpful for maintenance and waiting for data to be entered before computing. For example, say you have a form that finds the latitude and longitude of an address. You need to wait for the address to be entered before finding the latitude and longitude. The second you start to add that service into your form, you are adding a ton of code to your component. You can break these components into a smaller part (address form and map), share the data with Vuex and compute what is needed. You can also apply `v-model` to pieces of Vuex data and `watch` for changes which make this process even easier!

Rule of Thumb

My general rule of thumb for when to implement Vuex is if you need to pass data to more than 2 components, I'd look into some state management system. The

synchronizing of data gets exponentially more complicated the more components rely on a piece of data. As I mentioned, I use Vuex a lot and have found it saves me a ton of time and headache trying to keep everything in sync.

Vue 3 Composition API

With the new composition API there's been a little discussion on whether or not Vuex will be as useful as it has been. To be honest, I think it still will be. The composition API allows you to share pieces of functionality with other components which includes state. However, I believe that using a standardized state store helps to separate concerns and makes re-usable code even easier.

Conclusion

Vuex is unique right off the bat, but the more you use it, I believe the more useful it becomes. If you have any questions, feel free to reach out on our [community forum](#).

The Truth About Spa Security

Whatever is stored in NuxtJS and passed to the client is able to be inspected and modified. That's the cold hard truth. When building an SPA, we have to approach this very differently than building a monolith with server side rendering.

The key here, is where the data is rendered. With an SPA, this all happens on the client side. That means any data connection to your API, credentials, loaded data is all available for inspection by a user who is curious. With a server side rendered application you can load credentials into memory, tons of data, and output only what is needed to be viewed by the user.

What To Think About

When building an API Driven Application you have to really be certain that what you are returning is safe for the viewer to see. This starts with the server side API. When you are returning pieces of data from an endpoint, do not return private data! The user can inspect the network request and see what is being returned even if it's not presented on the screen!

The second place to think about is where you store access tokens and API key credentials. NEVER store a secret key in your SPA! A user can inspect the page and extract that secret token. Even if all of your app is minified and compressed, this doesn't prevent a savvy user from reverse engineering your code and extracting the token. With an authentication token, the user holding that token already has access to the API. I'm talking about 3rd party resources (Google Maps, Mailchimp, etc.) where your app has its own token. Implementing these functionalities that require a server side secret key should ALWAYS keep that key server side. It may be tempting to store this in your SPA, but it can be extracted. You should put these tokens on your API side.

Conclusion

With these warnings, this isn't meant to deter from building an API Driven Application. Just a heads up on what to be aware of! In doubt, don't share the data you don't want others to view. Don't share secret pieces of your app within Javascript either. Other than that, enjoy the power and flexibility that comes with building an API + SPA and the fact you can deploy to mobile with ease!

CONTINUING PROGRESS & THANK YOU

We couldn't have done this without you

We hope you found this book to be of value and fill in some of the missing pieces that may be overlooked in other places. We really enjoyed writing this book and look forward to hearing your feedback, experiences, and seeing all of the amazing new apps that you create.

We're so thankful for all the support we get from our readers like you.

There are a few others we'd specifically like to thank

James Hull

James was one of the many who reached out in excitement on our mailing list. He graciously offered a tremendous amount of suggestions and edits that greatly polished the quality of this book — asking for nothing in return. He is an amazing character. Thank you so much for all of your help James!

Jon Hainstock

Jon is the friendliest person you'll ever meet in marketing. Not only will he approach problems with the most creative solutions but he stays grounded to the highest quality of values. Jon provided incredible guidance for us and he will always be an influence on every product that we make.

Vija Verma

We've never had the pleasure of meeting Vija, but he was the artist who graciously offered his illustrations to us for the front cover for absolutely nothing. If you struggle with illustrations like we do, be sure to check out Vija's site: <https://illustrations.co/>

From Dan: To My Parents

I'd like to say thank you to my parents for their motivation, love, and un-ending support. Being creative makers themselves, they taught me to keep trying,

expand my knowledge, and don't stop at failure.

From Jay: To My Wife

Kristie, thank you so much for your love and continued support. You were so patient and understanding while I worked late and you always made sure I was well fed. Thank you for all of the sacrifices you made to make this book possible.

Keep on learning

Always stay hungry and keep a beginners mind. As your application grows, be sure to look back and find areas that you can optimize, split apart and improve. Think about what endpoints receive the most traffic and what you can do to make them faster to serve your users more efficiently. Designing an your app as an API first will allow these optimizations to be modular and efficient.

Don't just stop at the platforms listed or features discussed. If you want to make a specific desktop version of your app, the same structure applies (yup, we've got this to work in Electron). Try integrating with other APIs as well and share your journey along the way.

We want to thank you again for your support. By supporting our content, it enables us to make more of it, pursue our passion and help others along the way.

Keep building awesome things!

- Dan & Jay