LARAVEL DEVELOPMENT STEPS

Laravel: 6.0.2 | Vuejs: 2.5.17 | Vuetify: 2.1.1 | Vuex: 3.1.1 | Vue router: 3.1.3

# Create Laravel Project

Depends on the environment.

# JWT backend

We are going to create a restful api and the only one who is going to use it is us. So, we don’t use Laravel passport and we use a package called jwt-auth.

## INSTALL

Package:



### Install latest version without composer

To install this package, we go to “composer.json” and an add this like to require section:



* In order to install the latest version, we should check the GitHub repository at the time and use the latest version of jwt-auth.

### Publish the config

We use this command to publish the config file inside our config folder.



After this we will have a file called “jwt.php” inside our config folder.

### Generate secret key

Run this command:



This will update our “.env” file with something like “JWT\_SECRET=foobar”.

## START

### Update user model



* The green part is added to user model

### Configure auth guard

In “auth.php” file inside config folder we make the following changes:



### Add basic routes

In “api.php” file inside routes folder we add the following routes:



* Comment the middleware, and remove $router.

### Create auth controller



In the auth controller we add the following in the next page:



### Test

Now If we use postman and request the <http://example.dev/api/auth/login> route (post request), and provide the correct user email and password, we will get the token as an object in respond!

* in order to test other auth routes except login, we need to provide token as well (we can see that in auth controller constructor function).

There are a number of ways to provide token in order to access a route:

* Authorization header: Authorization: Bearer eyJhbGciOiJIUzI1NiI... (in postman we click header and add a new field after accept json and content-type json, the name will be authorization and the value would be bearer + token)
* Query string parameter: http://example.dev/me?token=eyJhbGciOiJIUzI1NiI...
* Post parameter: like form data we send a new parameter called token.

## Auth guard

Learn: from spa-forum in bitfumes tutorial.

And the reference is:



## Exception handling

Learn: from spa-forum in bitfumes tutorial.

And the reference is:



### New Middleware

Instead of auth: api, we use another middleware that we are going to create.

The reason behind this is that the auth: api middleware is not very good in giving exceptions (based on bitfumes).

So, we use this command:



Then we register this middleware, so we open kernel file,

In kernel file in the $routeMiddleware array we add this:



In middleware we will have this function:



* JWTAuth => Tymon\JWTAuth\Facades\JWTAuth.

Then we can go to our auth controller and use JWT instead of auth: api

So:



* Question: why we used this new middleware instead of auth api?
* Answer: if we want to check the difference between these two, its that auth api is not going give us a descriptive error, and we can’t even get the exception, we can test this by using auth api again. If token not provided auth api only gives us unauthorized. However, if we use this JWT middleware we created, we get a message saying “the token could not be parsed from the request”. And we see the exception as well so we can easily handle it!

### Code the exception handling

Now that we know what exceptions are occurring, we can handle them one by one.

First, we check different scenarios in postman then we handle every scenario in Exceptions/handler.php file.

This is all according to this reference:



Example:



## Sign up

We add route sign up, then we add a new function to our auth controller.

In this function we can easily create a user and then ofc, log user in.



* Request => use Illuminate\Http\Request
* Password problem: If we get password from user, the password is not encrypted, so we have to encrypt it before saving it to data base, otherwise the user can’t log in.

Tip: fix password problem forever with mutators in Laravel:

In order to fix the password problem, we can create a new mutator in User model:



In this way we no longer have to encrypt password all the time.

But I rather stick to first way

In order to sign up user, we don’t need token, so in constructor function we have to except it.



## Secure our routes (important)

Now that we set this up, we can secure our crud routes with JWT middleware we created.

For example, for question controller we add this for constructor:

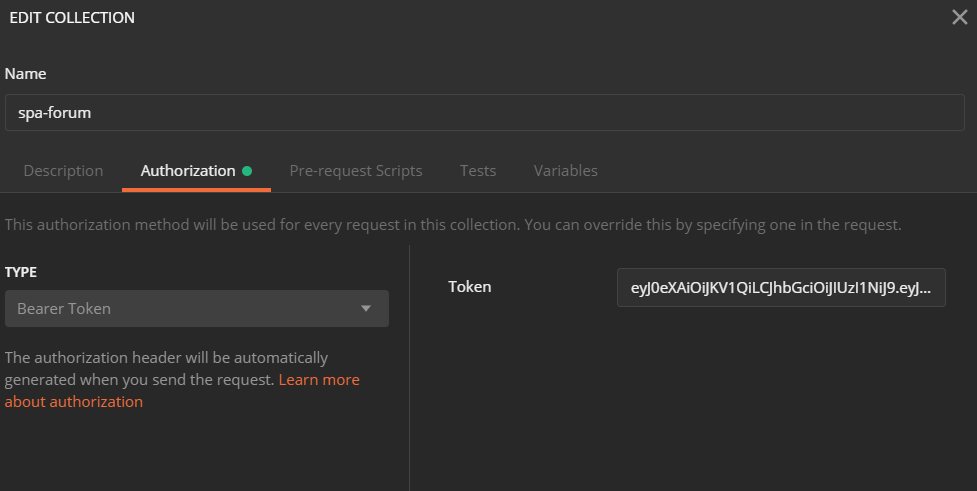


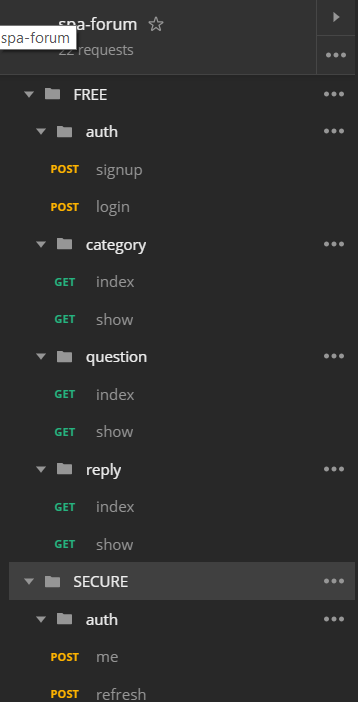
* Now we have a token check for store, update and destroy in our controller.

# Use Postman to test api

In order to test jwt and our restful api, I use postman.

To be able to use 1 token for all requests I use 1 parent folder with website name, then I put every other folder inside that, so now if I right click on parent folder and click on edit, then in authorization tab I can choose bearer token and for value I only provide the value and not with bearer.



Folder structure:

* To provide token for all requests that need token, we have a secure folder which is secured and needs token. We right click on secure and click edit, then we provide token whenever we login or somehow change token.
* Free routes don’t need tokens and we can always use them. So, we don’t provide token to them
* Every model will have a folder, containing its own requests.
* When providing token to parent folder all the folders inside will get the token except the ones who have no auth in their authorization tab.

# Create models with everything

Models, migrations, controllers and factories can all be created in single line below:



* Better if we put all models in a single folder called models.
* We can remove edit and create from controllers because this is an api.

# Design database migrations

* If database is complicated, try mind mapping it on paper first.

## ALWAYS USE “unsigendBigInteger” IN RELATIONSHIPS

I should always use unsigned big integers (UnsignedBigInteger) on relationships, because the auto increment id in all tables are unsigned and big integer. Therefore, if I use foreign() to for example cascade on delete or something, the foreign is not going to match and error happens.

## SOFT DELETE

* I once read an article that said we must not use soft delete on user model. However, it is very good to use soft delete in as many as models we can.

To use soft deletes, we add it to the model:



And we add it to the migration:



## CASCADE ON DELETE | CASCADE ON RESTORE

We must use cascade on delete (or restore if it is an option). To do it more eloquently, based on my experience we better use a package:





The “replies” is the relationship. For example, post has many “replies”.

This package is making this very EASY!

Now we use “delete(), withTrashed(), restore()” like before and the rest is done for us automatically.

# Code the relationships in Models

* Remember: Let’s say Post belongs to User. We know that Post is the child and User is the parent. then in Post model we say belongs to.
* And: Let’s say Post has many Replies. This time we know that Post is parent and replies are children, then in Reply model we say belongs to, and in Post model we say has many.

## USE MODEL CLASSES INSTEAD OF NAMES:



# Design Factories

I can always use bitfumes factory tutorial that is inside “Laravel spa forum pusher app” tutorial, until I can do it on my own. It’s pretty complete and easy though.

# Code the Routes

* All routes inside “routes/api.php” will get “/api” prefix automatically

## RESOURCE ROUTE

### USE “apiResource”

In order to create the routes for api, I use apiResource option:

For example, for Post routes:



### Use restful api FORMAT in relationships

* I add /post/{post}/ before reply to have a restful api format.

For example, for Reply routes:



## CUSTOM ROUTE

Maybe we don’t need full crud, for example if we have a simple like feature then we only need like and unlike (which is store and delete only).

Example:



* Options are get, post, put and delete.

# Design Resource templates

## CREATE

For example, to create a resource, we run this command:



## USE

To use a resource in controller, we must use collection in index, and “new ExampleResorce” in show.

* The example used in controller section.

## DESIGN

* The template structure is optional.
* In the result(postman) everything will be wrapped in a data object.
* We can change the name of particular data we send, for example instead of “body” bellow we can say “content”.
* We can use conditional situations for example if is admin (dev marketer in YouTube did that).
* Also, we can use an additional function called with to add more information along with data.

Example:



# Code the Controllers

## MASS ASSIGNMENT FIX IN MODEL

Mass assignment error happens when we use create and update methods, in order to fix that, I use this in my models:



## RESOURCE CONTROLLER

### INDEX

My current template for index is like this:



* Let’s say, we have “hasMany” relationship, for example, question and replies. To get the replies of a particular question, we send the question to index controller, then our code would be something like this:



* Note that when we already sent the question from the route.
* Tip: Double shift and type “response.php” to get every response statues text available!

CAUTION: Because the route has question binding, then we have to get this question in all of our reply controller functions, otherwise we get an error.

### STORE



* Also, we can pass an associative array to create method. In this way we can modify data before saving it into database.



### SHOW



### UPDATE



* Also, we can pass an associative array to update method. In this way we can modify data before saving it into database.



### DESTROY



## CUSTOM CONTROLLER

In the like example, we don’t need index, show and update methods. We only need store and delete which we will call like and unlike.



# Design Requests

Create a new request



For example, a Signup Request is something like this:



In authorize function we should change false to true, then we use any server-side authorization we like.

This link is helpful for rules:

<https://laravel.com/docs/6.x/validation>

after designing the request, we have to inject it in desired controllers, for example:



# Policies

Pending…

# Testing

Pending…

FRONTEND VUEJS

# Front-end

Right now, I don’t use vue-cli, because I want my app to be designed inside 1 directory so I install it in laravel. In this way we also learn more how vue-cli actually works. Because we implement everything based on vue-cli. In order to create a good and clean code which is 100 percent correct and is updated to latest, we must first take a look at vue-cli folder and file structures.

In vue-cli way we can install vuejs, vue router, vuetify, vuex and more. They are automatically installed and connected together with best way possible. In laravel, we can install all these packages with npm, then we implement them like vue cli.

We want to create a single page application, therefore there is only one view. We also need to catch all web routes and redirect them to our single view which we call spa.

So, in web.php file:



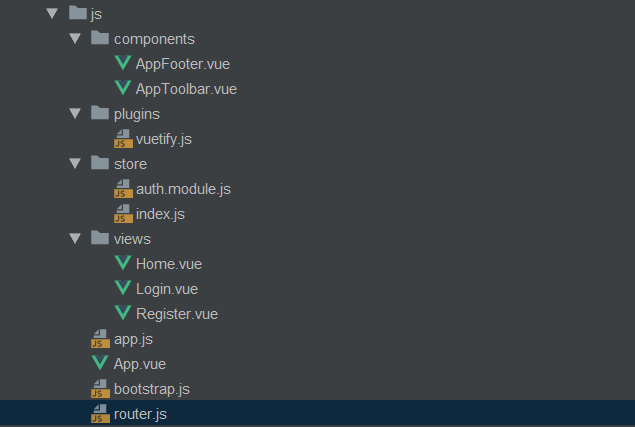
The only blade file we create in project. We can name this spa mean single page application. We must add app.css and app.js to this blade file, also should csrf token.

spa.blade.php:



# Front-end folder structure

Our folder structure is going to be someting like this:



* All of our front-end related files are going to be inside js folder in resource/js path.
* We use components folder for components that are not router related
* We use views folder for our router related components
* All routes are inside router.js file
* app.js file is basically the main.js file from vue-cli, having all the imports.
* App.vue is our root vue component that will be inserted to spa blade through app.js
* Store folder is for vuex and all its modules

# Vuejs

In order to install vuejs in our Laravel app we follow these steps:

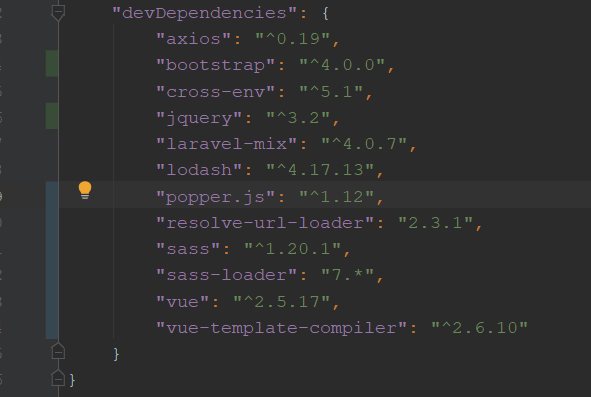
1. We run this composer command:



1. Now we can use this package and do vue scaffolding on laravel:



* Base on laravel documentation, vuejs comes with bootstrap as well. So, if we want to use vuetify we can just remove bootstrap and jquery. If we used git, we can see what was added to files, so it will be easy to track bootstrap and jquery and remove them.
* Files we need to edit are “bootstrap.js”, “app.js”, “app.scss” and “package.json”. for example, in package file we have something like this which still has bootstrap:



As we can see in the left, colorful ones changed, if we click on color blue, we can see what was it like before scaffolding. Green color also means that the line just created and was never there before. So, we can remove bootstrap, popper and jquery.

1. then, we run this:



Now we have vuejs installed without bootstrap and jquery.

# Vue router

1. Install with npm:



1. Create a router.js file inside js folder.

router.js:



1. Add this into our app.js file:



* Also, we can remove the other window require vue line, because it’s the same as import vue.

# Vuex

1. we install vuex:



1. Create a new folder “store” and a file “index.js” inside

store/index.js:



Then we update app.js file:



We import store at top



We create our modules inside the same folder, however we can add a .module.js only to see it’s a module.

Example module,

auth.module.js:



Now we can use this module in our index vuex,

index.js:



# Axios

Laravel already ships with axios, so in we don’t need to install it.

We can use axios in vue instance like router and store, but we shouldn’t do it, this will make our vue app heavier. So, we import axios only on components we want to use axios on.

## GLOBAL REQUEST CONFIGURATION

When sending ajax requests, we use URLs. we must have a base URL and use it in our axios so if we go to another address, we can easily change all of them.

We can set base url in bootstrap.js.

Ofc we can also use component level configuration and override the global.

* Interceptors are like middleware, they run in between, and block certain responds or requests. That’s why we use return on them, for example if we don’t return, we block everything.
* If we have different urls we can also use custom axios instance with its own global configurations, then import it in the component we want to use. (explained in max tutorial on vuejs axios part)

To set global url configuration:

As we know, axios is imported in bootstrap.js, so let’s add this line of configuration in there:



We can add this line after axios header configuration.

* I must return axios in my actions if I want to chain them more in my local for more async tasks. (why?). but we don’t have to do that and bellow explanation is not the only way to get the job done.

This is useful when I want to get something from api, then I want to manipulate my state in vuex, and then go to my local component and save a copy of the state in local, so if I change the local data, it doesn’t affect the global state. It’s useful with server-side validation, when I get server errors, I want to save it in my state global variable called errors, then I want to send this error to my local component but not all of them! So, after saving global errors, I chain the axios inside my local component and get the errors and save it into some variable like serverErrors, this serverErrors can be changed and will not affect the global errors. And another thing is that we have multiple components that might need errors. If we get errors in them too when the state starts to change, then all the components that get this state will change too and we don’t want that, therefore we only get it in our local component only once after we chain the axios.



Alternative to return is to use async register, and await axios.

Example of chaining for more async in local component



# Vuetify

References:

Vuetify github and official docs:

<https://vuetifyjs.com/en/getting-started/quick-start>

<https://github.com/vuetifyjs/vuetify>

Now we need to install a nice-looking framework for our application.

In order to install vuetify properly, I tested vue-cli and saw how vue uses scaffolding to modify current normal vue app to use vuetify.

Also, there is a guide for installing vue without vue-cli, which is on the github page.

* The guide was missing something, it didn’t use vuetify inside vue instance.

Here are my steps to add vuetify into a laravel app:

1. Install vuetify



1. We need mdi font for vuetify as well:



1. Styles



1. plugins/vuetify.js

Create a new file in js folder and then create vuetify.js. this file is going to import vuetify.

plugins/vuetify.js:



* This will export the vuetify instance we are going to use in our vue instance.

1. App.vue

We didn’t use any component inside our blade file. Therefore, we need to make the div with the id of app to use a main component called App.vue

In js folder, I create a new component called App.vue

js/App.vue:



* We can use the commented section in future. It’s just for giving me an idea of how the structure could be.

1. Use vuetify

Now we update app.js file to use vuetify and the App.vue we created

app.js:



* vuetify variable that we used inside our vue instance is imported at top, and is coming from the vuetify.js file inside plugins folder.
* When we importing App, we must use dot vue at the end, because this one is not exported. And we need this only for first time we loading App.vue into our blade php file. We can see that in render part inside vue instance.

# JWT frontend

We must store the jwt token, we need token if we want to send any request that need to be authorized.

For now, I use local storage to store tokens. But in future I can implement other secure ways. But according to pros like max, vuejs is secure to use local storage. So, for now we are ok.

I create a new module for all my auth related functions,

store/auth.module.js



index.js:



We add a watch in our App.vue to clear errors whenever we change route, so state error bug never occurs.

Add this to App.vue:



# Vuetify tips

## Grids

Reference: <https://vuetifyjs.com/en/components/grids>

* Tip: with vuetify and router together installed, when we want to create a new link instead of using router-link we can use v-btn. Yes, it has the “to” attribute to handle our router link.

### Cut in half

Exactly like bootstrap. Let’s create a container then a row and then two col-md-6:



First, we say cols 12, means we want this col to take all the space from xs to lg.

Second, we say md 6, means we want this col to take half the space from md to lg.

Mobile first approach: means first we think about mobile device. We set col to 12 for that. then we think about other devices, so we set md to 6.

* no-gutters will remove the between column spaces can be added to v-row.

### Center horizontally



Don’t try centering vertically, use margins to center vertically.

## Forms

Here is a simple form



* We don’t use type of submit on form button, because we don’t want html to handle our forms, we want vue js to handle it, therefore there is no need to prevent the default behavior of the form because there is no input with type of submit.
* We get validation system out of the box when we use v-form.
* valid variable is used to check the whole validation status of the form and used for disabling the button for submit as well as checking before sending axios request.

Form validation with vuetify is well explained in net ninja vuetify tutorial.

Frontend validation and axios call:



# Vuex tips

We store our global variables in state.

Getters are like computed properties; we use them to not actually change data.

* Getters always receive “state” as their first argument
* Getters also receive “getters” as their second argument
* We can’t only send getters as first argument… if we need getters, we must also send state.

Mutators must be used to change state data forever.

Actions are like mutators, but for ajax requests or logics.

In actions, we must use brackets to get state, commit, getters or rootState.

Always use actions to do most of the logic and job, but every time you want to change state, commit a mutator and let everything be organized.

* Tip: If you want to change global state from nested module, you can write the mutator still on the global vuex, then just call the commit with third {root: true} property from module. This is the reason mutators never get rootState property.

# Validation tips

In order to validate our forms, we can use all the technologies we installed including vuejs, vuetify, vuex and laravel Requests validations.

We do two step validation, server-side and client-side. First, we disable client-side and completely get errors from api and show them in real-time. Then we design client-side validation. In this way we will have two-layer validation which is very secure.

Bellow I show how I can implement this for now:

## Server-side

### first way

First, we implement server-side validation from requests, then we can access errors coming from server in axios catch. After that if we are in vuex, we save the errors in global state. We have to return axios in our action, therefore we can perform another asynchronized task on the local component and get the errors and save them in a new object. We have to use assign to create a clone of object otherwise if we change this object the state object will change too.

All that said, we have access to our errors right after the axios request and we have saved it in our local component and we can show it.

* we return axios
* We save errors in catch section

Then in our local component we dispatch the method again and use assign to clone the object.

We perform dispatch after we click on register bottom and then create a new object and save it in local data.

How to show server-side errors?

After user clicks on the submit button, if there are any server errors, we want to immediately get them and show them in vue in real-time.

Vuetify has a prop that we can add to our text fields, for example:



As we can see in here, error-messages has been added, if we have email error in serverErrors object, we will show it very efficiently. This actually will change the “valid” data from form to false, and gives all the error styles to input as well. If this error-messages become null the “valid” data becomes true again, and error styles will be removed. That’s why we added @keyup so whenever the user tries to fix error, we make it null, error disappears and user can submit form again.

* Rules prop is for client-side validation, it returns an array of function rules.
* When we code server-side validation, we must return empty array in rules prop so we can actually test everything, after server-side validation is functional we can go and code the client-side validation.

For example:



Every client-side validation are empty arrays to test server-side validation.

### Second way (cleaner)

In this way we don’t return axios and we don’t perform another async inside component,

auth.module.js - actions:

 auth.module.js - mutations:



store/index.js:



App.vue



* We check for any route change and clear errors.

Register.vue



* We use map actions to have register.
* We use computed serverErrors to always have them in our component(s).

In this way we get errors in our components and we don’t worry if they get wrong error because whenever path is changing the state errors clears and no longer there to be computed.

## Client-side validation

Every input gets a prop called rules and inside that we can have an array of rules.

After server-side validation completed we can add client-side validation like this:



# Router tips

This is how I guard my routes:

router.js:



And we can change this and export it at the end, this way we can use router showed in top.

