**Lab 11 – A second Laravel project**

You will create a Laravel project allowing everyone to view news stories from a database, authenticated users can upload news items through a form, and only the author can delete a story. We will also use a weather api to display the current temperature on the home page.

NOTE: if you change the existing users table in Homestead, your Todo app may not work perfectly after we have completed the steps.

# Step 1 – Create a new app: laravel new *appname*

Vagrant ssh and cd to your Code folder. Create it in the projects folder as

laravel new news

# Step 2 – Set the local variables

Change the following fields in your .env file. If I want to point to the local postgres database, I will only change the DB\_CONNECTION to pgsql and the port to 5432. If I want to refer to the heroku database, I will change **all the DB fields (not quoted)**. Note this in Step 3.

APP\_NAME="News stories with Laravel"

DB\_CONNECTION=pgsql

DB\_PORT=5432

# Step 3 – Deploying onto Heroku

This step can be performed at any point after you have created the new app. You can delay until you have a working app, or treat Heroku as a Git repo, and group your commits and push as needed. Steps to deploy onto Heroku:

1. vagrant ssh
2. Download and install the heroku command-line interface. Unfortunately in the labs, you need to do this everytime

# Run this from your terminal.

wget -qO- https://cli-assets.heroku.com/install-ubuntu.sh | sh

Verify the install:

heroku --version

1. heroku login
2. cd to code/news
3. create the Procfile. We will use the Apache web server on Heroku instead of nginx simply because Apache is preconfigured on Heroku to allow URL rewriting which we need when the browser requests need to get “redirected” internally to a controller.

echo web: vendor/bin/heroku-php-**apache2** public/ > Procfile

1. If you haven’t already, initialize a git repository:

git init .

#git remote add origin gitLabUrlOptionalForLab

#git checkout -b branchNameOptionalForLab

git add .

git commit -m “new laravel project”

1. heroku create yourappname (note the app name, URL and git url)
2. The application’s encryption key is used by Laravel to encrypt user sessions and other information. Its value will be read from the APP\_KEY environment variable.

As it must comply with the rules of the selected cipher in the configuration, the easiest way to generate a valid key is using a php artisan command, which will print a key that you can copy and then paste into the next step.

**php artisan key:generate**  
heroku config:set APP\_KEY=*key(replace with key generated above including from base64 to = inclusive)*

We will also set some other environment variables on Heroku:

1. heroku config:set APP\_DEBUG=true
2. heroku config:set APP\_LOG=errorlog
3. heroku config:set APP\_NAME="News stories with Laravel"
4. heroku addons:create heroku-postgresql:hobby-dev

Note the host, database, username and password (also available in the heroku web dashboard) and use as the values in the step below:

1. edit the config/database.php file:

In the ‘connections’ array, add:

'pg-heroku' => [

'driver' => 'pgsql',

'host' => '**host**',

'database' => '**database**',

'username' => '**username**',

'password' => '**password**',

‘port’ => 5432,

'charset' => 'utf8',

'prefix' => '',

'schema' => 'public',

],

Set your default to:

**'default'** => env(**'DB\_CONNECTION'**, **'pg-heroku'**),

If you want, update the .env file with the heroku pgsql credentials in order to bypass the Homestead database and only work on Heroku.

DB\_CONNECTION=pg-heroku

DB\_HOST=*blahabl*.compute-1.amazonaws.com

DB\_PORT=5432

DB\_DATABASE=*blahblah*

DB\_USERNAME=*blahblah*

DB\_PASSWORD=*blahblah*

1. heroku config:set DB\_CONNECTION=pg-heroku
2. git add .
3. git commit -m “heroku postgres db support”
4. git push heroku master #ok for the lab

**NOTE**: you may want to push a [non-master branch onto Heroku](https://devcenter.heroku.com/articles/git#deploying-from-a-branch-besides-master) as Heroku’s master (strongly recommended that you do NOT push anything to the Gitlab master in the project). In that case, you would:

git push heroku testbranch:master

# Step 4 – Determine and create your migrations.

We will need two tables, one for the news items and one for the users. The user migration exists already, but we want to add more columns. Make sure your pwd is Code/news and use the artisan command for the scaffolding code to be automatically created for you:

php artisan make:migration edit\_users\_table

This will create a new migration file named edit\_users\_table with an up and down method.

Within this migration, we want to **add** a column to the users tables:

public function up(){

Schema::table('users', function(Blueprint $table)

{

$table->string('accreditation');

});

}

public function down(){

Schema::table('users', function ($table)

{

$table->dropColumn('accreditation');

});

}

Create a second migration create\_stories\_table to create a stories table:

public function up()

{

Schema::dropIfExists('stories');

Schema::create('stories', function(Blueprint $table){

$table->increments('id');

$table->timestamps();

$table->string('title');

$table->string('post');

$table->integer('user\_id')->unsigned();

$table->foreign('user\_id')

->references('id')->on('users');

});

}

public function down(){

Schema::dropIfExists('stories');

}

Recall that the timestamps are needed if you want create\_at and updated\_at columns created by default. Note that the foreign key relationship requires that the migrations are run in the proper order (users table created first, then stories).

# Step 5 - Running migrations

Run migrations on heroku (after you have pushed your code to heroku) through:

heroku run php artisan migrate

(Note: in general, and command that will change the database needs to be run on Heroku if you are using this database for development. heroku run in front of the relevant php artisan command will do this)

This is required if you point to the heroku database during development. If your config/database default is pgsql and your .env was changed to pgsql with the correct port but pointing to homestead, then you can run php artisan migrate to run the migrations on the local database.

The artisan migrate command will run migrations which were not run previously – i.e., their version is not contained in the table. Once a migration is applied, a row is added into the migrations table in your PostGres database. This is why editing a migration file that has already been executed will not have any effect.

Note: if you have an error, the subsequent migrations that you have defined will not run.

When migrating down (using the command php artisan migrate:rollback) the last “batch” of migrations that ran will be undone (with the down method). There is a risk of data loss. The command php artisan migrate:reset will roll back all migrations (prefix with heroku run to run this command on Heroku)

**Pro tip:** use PSQL (Postgres CLI tool) to clear out the contents of the migrations table and/or drop the tables if you have messed up and want a clean start! With Heroku:

* heroku pg:psql (to connect to postgres on Heroku)
* you will be in your database, so you can select \* from migrations, etc…

**Important note about migrations and Git:**

Git is used as a code repository, and can be used to reset the repository to a previous state. The database migrations are stored in the repository, however the migration process is separate from the Git push process.

# Step 6 – Edit your User model, make your authentication routes and views, and edit the registration view

The User model comes built in with Laravel. Since we added a column though, we should update the model, as well as the places where the user registers.

Open /app/User.php and add accreditation to the $fillable array; this means that we can send this in one step when we create a new User.

protected $fillable = ['name', 'email', 'password', **'accreditation',**];

Now open app\Http\Controllers\Auth\RegisterController.php. Since we want to make use of the existing registration and authentication code, we will edit this file:

Accreditation is required, so add a key-value pair to the validator function.

'accreditation' => 'required',

In the Create method, add a key-value pair so that the accreditation is written when a new User is created:

'accreditation' => $data['accreditation'],

Add the authentication routes and views:

php artisan make:auth

We will now open \resources\views\auth\register.blade.php to **add** the accreditation request to the form. Follow the template used by the other inputs so that the JavaScript nicely highlights errors caught at the server:

<div class="form-group{{ $errors->has('accreditation') ? ' has-error' : '' }}">

<label for="accreditation" class="col-md-4 control-label"> Journalistic accreditation </label>

<div class="col-md-6">

<input id="accreditation" type="text" class="form-control" name="accreditation" value="{{ old('accreditation') }}" required>

@if ($errors->has('accreditation'))

<span class="help-block">

<strong>{{ $errors->first('accreditation') }}</strong>

</span>

@endif

</div>

</div>

*Note: in this example, we don’t need to change anything in the controller. But if you needed your controller to pass additional data to the registration view (e.g., accreditation is determined based on an enum, or list of values from a reference table), you would want to change the registration ‘get’ method so that it sends this data to the view. Notice that the RegisterController does not have an action: it uses a trait found in vendor\laravel\framework\src\Illuminate\Foundation\Auth\ called RegistersUsers.php. This trait defines the showRegistrationForm() method that is invoked to return the registration form. We have to override this function in RegisterController. So add the function to RegisterController:*

*public function showRegistrationForm()*

*{*

*//any additional logic*

*return view('auth.register', $data); //add a second parameter to pass any data*

*}*

# Step 7 – Clean up the routes (part 1)

We love Laravel, we really do, but we don’t want the app to open on the Laravel welcome page. Open routes/web.php and change to / route to go to the HomeController:

Route::get('/', 'HomeController@index');

Open the HomeController, you will see that the constructor has $this->middleware('auth'); This means that all actions of this controller are available to authenticated users only. We’ll reconsider this shortly.

# Step 8 – Run your app

It is always easier to debug when we run our code changes frequently. Open news.app in your favourite browser (besides Edge); you should be redirected to the login page; click on the Register link. You should see the Journalistic accreditation field in the form. Play around, register yourself, then check the database contents through psql.

If you are testing on heroku, don’t forget to add, commit and push!

Pro-tip: if you have HTTP 404 Not Found errors, you may want to double check your routes. php artisan route:list shows you all the routes on Homestead. heroku run php artisan route:list shows you the routes on heroku.

# Step 9 - Create your Story model, and view and edit the controller to take care of the routes

***part a – the Story model***

php artisan make:model Story

Edit the app\Story.php file. We need to ensure that the correct table is used (our table was called stories, but our class is called Story not Storie, and I don’t think Laravel is smart enough to make the grammar change). So we will add a property to the class to ensure the mapping is done:

protected $table = 'stories';

We will add another property for the mass assignable attributes:

protected $fillable = ['title', 'post',];

We will associate a User with a Story by adding this function:

public function user()

{

return $this->belongsTo('App\User');

}

Finally, we will add a method to indicate if a user owns the story:

public function userCanEdit(User $user)

{

return $user->id === $this->user\_id;

}

Now edit the **User** class so the relationship is defined:

public function stories()

{

return $this->hasMany('App\Story');

}

***part b – the routes***

Edit again the routes/web.php. Add new routes. Your final routes should match those below. Note: the latter routes are for authenticated users only.

Auth::routes();

Route::get('/', 'HomeController@index');

Route::get('/home', 'HomeController@index');

Route::post('/home', 'HomeController@store')->middleware('auth');

Route::delete('/home/{story}', 'HomeController@destroy')->middleware('auth');

Recall the Route syntax: The route Route::get('/', 'HomeController@index'); will route any GET request to the / resource to the 'HomeController index method. The middleware(‘auth’) is applied to those routes that we want to protect. We will have to add a hidden input field to the form to remove a story in order to “spoof” the http delete method.

*NOTE: If you needed to add an additional controller (always have controllers having single common responsibility), you would use the command*

*php artisan make:controller NameOfController*

***part c – edit HomeController***

Add the import:

use App\Story;

Edit the following methods.

Remove the line in the **constructor** – we don’t want all the actions to be restricted to authenticated users, only those actions that we specified in the route.

In the **index** method, we want to return **all** stories, using pagination of 5/page.

public function index()

{

$stories = Story::orderBy('created\_at', 'DESC') ->paginate(5);

return view('home.index', [

'stories' => $stories,

]);

}

Note: The index method will return the view named index.blade.php which is located in a home directory within \resources\views, passing to it the stories. The key in the associative array is what the view uses to refer to the object.

In the store method, we want to save a new story:

public function store(Request $request)

{

$this->validate($request, [

'title' => 'required|max:50',

'post' => 'required|max:255',

]);

$request->user()->stories()->create([

'title' => $request->title,

'post' => $request->post,

]);

return redirect('/');

}

Note the syntax: the user() object is available through the request; on that object, we follow the relationship we coded to get to the associated stories; and we create a new story, passing the mass assignment fileds we specified in $fillable.

***part d - the policy***

In the destroy method we want to delete a story only if the user is the person who created it. You can use a policy, or query the model directly in the action:

Example to read, but don’t code this:

public function destroy(Request $request, Story $story)

{

if ($request->user()->id === $story->user\_id)

$story->delete();

return redirect('/');

}

This issue with the code above is that there are no errors if the user has somehow requested to delete something that they don’t own. So I prefer creating a policy: it keeps with the spirit of Laravel, the policy can be used to authorize future actions. A user who tries to delete something that they are not allowed will get a 403 error (forbidden).

First, create the scaffolding code:

php artisan make:policy StoryPolicy

Open the app/Policies/StoryPolicy.php

Add an import:

use App\Story;

Add a policy method:

/\*\*

\* Determine if the given user can delete the given story.

\*

\* @param User $user

\* @param Story $story

\* @return bool

\*/

public function destroy(User $user, Story $story)

{

return $user->id === $story->user\_id;

}

Associate the Story model to the StoryPolicy policy: open app/Providers/AuthServiceProvider.php and add an entry in the $policies array:

'App\Story' => 'App\Policies\StoryPolicy',

Finally, invoke the policy in the destroy action of the HomeController:

public function destroy(Request $request, Story $story)

{

$this->authorize('destroy', $story);

$story->delete();

return redirect('/');

}

***part e – the view***

We will need a view that displays all the stories, and displays a form for a new story **only** to authenticated users. Delete buttons are **only** shown to the user who wrote the story.

Create a folder named **home** in the \resources\views folder. Write the following code in the index.blade.php file. Notice the coloured lines, they are discussed below.

@extends('layouts.app')

@section('content')

<div class="container">

<div class="col-sm-offset-2 col-sm-8">

<!-- Display the weather here -->

<div class="panel-heading">

The current temperature is TBD

</div>

<!-- Display this panel only if the user is authenticated -->

@if (Auth::check())

<div class="panel panel-default">

<div class="panel-heading">

New Story

</div>

<div class="panel-body">

<!-- Display Validation Errors -->

@include('common.errors')

<!-- New Story Form -->

<form action="/home" method="POST" class="form-horizontal">

{{ csrf\_field() }}

<!-- Story Title -->

<div class="form-group">

<label for="story-title" class="col-sm-3 control-label">Title</label>

<div class="col-sm-6">

<input type="text" name="title" id="story-title" class="form-control" value="{{ old('title') }}">

</div>

</div>

<!-- Story Post -->

<div class="form-group">

<label for="story-post" class="col-sm-3 control-label">Post</label>

<div class="col-sm-6">

<input type="text" name="post" id="story-post" class="form-control" value="{{ old('post') }}">

</div>

</div>

<!-- Add a new Story Button -->

<div class="form-group">

<div class="col-sm-offset-3 col-sm-6">

<button type="submit" class="btn btn-default">

<i class="fa fa-btn fa-plus"></i>Add new story

</button>

</div>

</div>

</form>

</div>

</div>

@endif

<!-- Current stories with pagination-->

@if (count($stories) > 0)

<div class="panel panel-default">

<div class="panel-heading">

Current stories

</div>

<div class="panel-body">

<table class="table table-striped task-table">

<thead>

<th>Top news stories</th>

<th>&nbsp;</th>

</thead>

<tbody>

@foreach ($stories as $story)

<tr>

<td class="table-text"><strong>{{ $story->title }}</strong>

{{ $story->post }}</td>

@if (Auth::check() &&

$story->userCanEdit(Auth::user()))

<!-- Story Delete Button -->

<td>

<form action="/home/{{ $story->id }}" method="POST">

{{ method\_field('DELETE') }}

{{ csrf\_field() }}

<button type="submit" id="delete-story-{{ $story->id }}" class="btn btn-danger">

<i class="fa fa-btn fa-trash"></i>Delete

</button>

</form>

</td>

@else

<td></td>

@endif

</tr>

@endforeach

</tbody>

</table>

{!! $stories->render() !!}

</div>

</div>

@endif

</div>

</div>

@endsection

The code between the if (Auth::check()) and the matching endif statement is only executed if the user is authenticated; so unauthenticated users will not see a form asking for a new story; they won’t see the delete button unless they are authenticated and they created the story.

All forms with POST method with the route in the routes/web.php **must** have the {{csrf\_field()}} for CSRF protection. The Delete form also has the {{ method\_field(‘DELETE’)}} to spoof the delete http method.

The foreach block is to show all the stories. The $stories variable was passed by the controller action – it was the key used in the associative array sent to the view.

Notice that php code is signaled with the @ symbol. The {{ }} indicate that the content (often PHP code) must be html-escaped to prevent XSS attacks: be sure to use these in the view whenever you are showing user-supplied information in the browser (e.g., notice {{ $story->post }}).

The $stories->render() code is required for pagination, we use the {!! !!} syntax to indicate that the output should **not** be html-escaped. This is required for pagination to work correctly.

Also notice that the forms’ action indicates the URI in the request, and the method indicates the http method. These are used by the routes.php to determine the controller action to route to.

Finally notice that we will need to go back to add the current weather!

***Add a folder resources\views\common with a file errors.blade.php with the following content:***

@if (count($errors) > 0)

<!-- Form Error List -->

<div class="alert alert-danger">

<strong>Whoops! Something went wrong!</strong>

<br><br>

<ul>

@foreach ($errors->all() as $error)

<li>{{ $error }}</li>

@endforeach

</ul>

</div>

@endif

# Step 10 Run your application.

You should see a pretty blank page with nothing. This is normal since our database is empty, but shouldn’t happen later, since there will always be news items once the service is up-and-running. You often need to start the database tables with records, especially if they are reference table. You may want to:

* add data directly in the database using psql
* create a user account and add data
* seed the database using Laravel’s tools.

# Step 11 – [Seeding](https://laravel.com/docs/5.3/seeding) the database

First create a user account by registering on the website. The id will be 1.

We will now create a seeding class that adds 10 news items so that we can see pagination at work.

php artisan make:seeder StoriesTableSeeder

Go to the folder \database\seeds and open the file. A seeder class contains only 1 method, run, which is invoked when you run your seeder through artisan. Right the lines of code to add records; you can use the Eloquent models.

Add this use namespace statement:

use App\Story;

And code the run method:

public function run()

{

Story::create(['title' => 'Hidden message reveals clues to century-old P.E.I. farmhouse',

'post' => 'A Prince Edward Island family recently received a message from beyond the grave, written by the original owner of their century-old farmhouse.',

'user\_id' => '1']);

Story::create(['title' => 'Chicago Police Release Footage of Pinned Officer Shooting Teen Driver',

'post' => 'Chicago Police released footage on November 9 of the fatal police shooting of 19-year-old Juan Flores after he pinned an officer between his SUV and a squad car.',

'user\_id' => '1']);

}

Create 8 other news items in the method (I got mine from Yahoo news)

Note: When you use a seeder class, all models are “unguarded”: this means **all** the columns are mass-assignable, regardless of the $fillable array. We need to provide the user\_id foreign key when creating a new story, it is mass assignable with seeders only. In regular flow, it is taken from the relationship with the user object.

Add this line to the DatabaseSeeder’s run method. The DatabaseSeeder allows you to control the order in which you seed your tables.

$this->call(StoriesTableSeeder::class);

To run the seeder: from heroku, first git add, commit and push, then:

heroku run php artisan db:seed

or

php artisan db:seed

Run your web site again.

# Step 12 Integrating a weather API

We will be using the OpenWeatherMap’s API to retrieve the current weather. First, Get an API key for the free level at: <https://openweathermap.org/price> or use mine:

25b8cccb8b68c21226cdeacc4fb327e2

Instead of having code related to this API within our controller, we will instead create a repository class and use dependency injection to provide the class to the controller. Create a folder app/Repositories and within this folder a new class WeatherRepository.

Within WeatherRepository, add a public function getTemp() and within this function write to code to return the weather for Montreal (the url you will query is [http://api.openweathermap.org/data/2.5/weather?q=Montreal,CAN&units=metric&APPID=*1234*](http://api.openweathermap.org/data/2.5/weather?q=Montreal,CAN&units=metric&APPID=1234) ) where you replace APPID with your API key. The temperature will be in the main->temp property.

You need to inject the WeatherRepository to the HomeController. Add it as a protected instance variable, and change the HomeController constructor to take a WeatherRepository as a parameter and set the protected variable. Finally, in the index method, invoke the getTemp() method, and pass the temperature to the view:

return view('home.index', [

'stories' => $stories,

‘temperature’ => $this->weather->getTemp(),

]);

In the view, replace TBD with $temperature:

<div class="panel-heading">

The current temperature is $temperature

</div>

# Step 13 Building our own Web API

We will build a simple RESTful API, with a two methods. The first will retrieve all news items, with no authentication.

***part 1 - create the route***

All API routes are defined in /routes/**api.php**. They will all be prefixed automatically with api/ We will specify the route we want:

Route::get('v1/news', 'ApiController@index');

***part 2 - create the controller***

When you run

php artisan make:controller ApiController

you will have the scaffolding code. Add a use statement to import the Story model. We only need the index function. Within the index function:

public function index()

{

$stories = Story::all()->sortByDesc('created\_at');

return response()->json($stories, 200);

}

Note: The json method will automatically set the Content-Type header to application/json. The second parameter is used to set the http status code: it is not in the json response. The $stories object will look like this (formatting added for readability):

{"1":{

"id":2,

"title":"Idiots abound",

"post":"Dorval mosque targeted by anti-refugee phone threat",

"created\_at":"2015-11-30 03:38:45",

"updated\_at":"2015-11-30 03:38:45",

"user\_id":1

},

"0":{

"id":1,

"title":"P.K. Subban thrills",

"post":"He raises money for Montreal Childrens Hospital through jersey signing",

"created\_at":"2015-11-30 03:38:45",

"updated\_at":"2015-11-30 03:38:45",

"user\_id":1

}

}

The stories are sorted by creation date, and keyed by their id. You could use this id for subsequent requests, if desired.

To test: in a bash window, run a cUrl command:

curl -H "Accept: application/json" -X GET https://yourappname.herokuapp.com/api/api/news

***part 3 - create a more complicated route and controller action***

We’ll now add a second Web API: this one gets the user credentials, and only shows stories written by that user. NOTE: APIs should be protected with **tokens**. We will use the [jwt-auth](https://jwt-auth.readthedocs.io/en/develop/quick-start/) package. (jwt-auth stands for JSON Web Token Authentication).

1. install the latest version of the jwt-auth package. Make sure you are in a Vagrant shell in the news folder, then:

composer require tymon/jwt-auth

This will update your composer.json file with the new dependency and also download it into your vendor folder.

Unfortunately, you will need to open composer.json and make this change:

"tymon/jwt-auth": "^1.0.0-rc.3"

Then run:

composer update

1. publish the jwt config file. This will create the file config/jwt.php:

php artisan vendor:publish --provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"

1. Generate a secret key for jwt. This will be placed in your .env file, it is the key used to sign your tokens.

php artisan jwt:secret

1. We need to update the User model by having it implement the Tymon\JWTAuth\Contract\JWTSubject interface. This defines two methods: getJWTIdentifier() and hetJWTCustomClaims().

Add a use statement so you don’t have to fully qualify:

use Tymon\JWTAuth\Contracts\JWTSubject;

Add: implements JWTSubject

Add the two methods:

/\*\*

\* Get the identifier that will be stored in the subject claim of the JWT.

\*

\* @return mixed

\*/

public function getJWTIdentifier()

{

return $this->getKey();

}

/\*\*

\* Return a key value array, containing any custom claims to be added to the JWT.

\*

\* @return array

\*/

public function getJWTCustomClaims()

{

return [];

}

1. we need to make sure that jwt is the authentication “guard” for our api routes. Inside config/auth.php, make the following change:

'guards' => [

'api' => [

'driver' => **'jwt'**,

'provider' => 'users',

],

],

1. In order to deal with expired token exceptions correctly, open app/Exceptions/Handler.php.

In the function render, add the following before the return statement:

if ($exception instanceof Tymon\JWTAuth\Exceptions\TokenExpiredException) {

return response()->json(['token\_expired'], $exception->getStatusCode());

} else if ($exception instanceof Tymon\JWTAuth\Exceptions\TokenInvalidException) {

return response()->json(['token\_invalid'], $exception->getStatusCode());

}

1. The way the JWT token works is that first the mobile or Web application will log the user in. They are then given a token which is then used to remember the authenticated user. This means that there can be a single log in, instead of repeatedly asking the user for the password everytime an API call must be made.

This means we need a new Authentication controller, which is used by the API users.

First, add the authentication related routes to routes/**api**.php

Route::group([

'middleware' => 'api',

'prefix' => 'auth'

], function ($router) {

Route::post('login', 'AuthController@login');

Route::post('logout', 'AuthController@logout');

Route::post('refresh', 'AuthController@refresh');

Route::post('me', 'AuthController@me');

});

1. Now create the AuthController:

php artisan make:controller AuthController

Add the following code to the AuthController:

<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Auth;

use App\Http\Controllers\Controller;

class AuthController extends Controller

{

/\*\*

\* Create a new AuthController instance.

\*

\* @return void

\*/

public function \_\_construct()

{

$this->middleware('auth:api', ['except' => ['login']]);

}

/\*\*

\* Get a JWT via given credentials.

\*

\* @return \Illuminate\Http\JsonResponse

\*/

public function login()

{

$credentials = request(['email', 'password']);

if (! $token = auth('api')->attempt($credentials)) {

return response()->json(['error' => 'Unauthorized'], 401);

}

return $this->respondWithToken($token);

}

/\*\*

\* Get the authenticated User.

\*

\* @return \Illuminate\Http\JsonResponse

\*/

public function me()

{

return response()->json(auth()->user());

}

/\*\*

\* Log the user out (Invalidate the token).

\*

\* @return \Illuminate\Http\JsonResponse

\*/

public function logout()

{

auth()->logout();

return response()->json(['message' => 'Successfully logged out']);

}

/\*\*

\* Refresh a token.

\*

\* @return \Illuminate\Http\JsonResponse

\*/

public function refresh()

{

return $this->respondWithToken(auth()->refresh());

}

/\*\*

\* Get the token array structure.

\*

\* @param string $token

\*

\* @return \Illuminate\Http\JsonResponse

\*/

protected function respondWithToken($token)

{

return response()->json([

'access\_token' => $token,

'token\_type' => 'bearer',

'expires\_in' => auth('api')->factory()->getTTL() \* 60

]);

}

}

1. Test the API authorization: in the vagrant shell, run this curl command to validate that you get an access token returned (use an email-password pair in your database!). REMEMBER: all routes defined in api.php will be prefixed with api/

curl -H "Accept: application/json" -H "Content-Type: application/json" -X POST -d '{"email": "j@j.j", "password":"123456"}' http://news.test/api/auth/login

1. now define a new route in your routes/**api**.php file add:

Route::post('v1/author', 'ApiController@byauthor')->middleware('api');

1. In the ApiController, add a use statement:

use Illuminate\Support\Facades\Auth;

Add a method:

public function byauthor(Request $request)

{

$user = auth('api')->user(); //returns null if not valid

if (!$user)

return response()->json(['error' => 'invalid\_token], 401);

else {

$stories = Story::where('user\_id', '=',$user->id)->orderBy('created\_at', 'desc')->get();

return response()->json($stories, 200);

}

}

Note that since our route is in routes/api.php, Laravel doesn’t expect a CSRF token to be sent back.

To use this route, you will need to pass the token that you get through the login api call in the header of the POST request.

If an unauthorized user tries to get the stories (i.e., invalid token), the json response will look like this:

{error:"invalid\_token"}

and the http status code will be 401.

Note: In order to test a WEB API that uses POST, I recommend that you create a PHP console application that uses cURL to repeatedly test. For one-of testing, you can use cURL directly in bash. To test the api above in bash:

curl -H "Accept: application/json" -X POST -H "Authorization: Bearer longtokenhere" http://todo.test/api/v1/author