URL: https://www.toptal.com/laravel/passport-tutorial-auth-user-access

When developing a web application, it is generally a good idea to split it into two tiers. A middle-tier API interacts with the database, and a web tier usually consists of a front-end SPA or MPA. This way, a web application is more loosely coupled, making it easier to manage and debug in the long run.

When the API has been created, setting up authentication and state in a stateless API context might seem somewhat problematic.

In this article, we’ll look at how to implement full user authentication and a simple form of access control in an API using Laravel and Passport. You should have experience working with [Laravel](https://www.toptal.com/laravel) as this is not an introductory tutorial.

Installation prerequisites:

* PHP 7+, MySQL, and Apache (developers wanting to install all three at once can use [XAMPP](https://www.apachefriends.org/download.html).)
* [Composer](https://getcomposer.org/doc/00-intro.md)
* [Laravel 7](https://laravel.com/docs/7.x)
* [Laravel Passport](https://laravel.com/docs/7.x/passport#installation). Since APIs are generally stateless and do not use sessions, we generally use tokens to keep state between requests. Laravel uses the Passport library to implement a full OAuth2 server we can use for authentication in our API.
* Postman, cURL, or Insomnia to test the API—this is up to personal preference
* Text editor of your choice
* Laravel helpers (for Laravel 6.0 and up)—after installing Laravel and Passport, just run:

composer require laravel/helpers

With the above installed, we’re ready to get started. Make sure to set up your database connection by editing the .env file.

**Laravel Passport Tutorial, Step 1: Add a Controller and Model for Dummy Requests**

First, we’re going to create a controller and model for dummy requests. The model isn’t going to be of much use in this tutorial, it’s just to give an idea of the data the controller is meant to manipulate.

Before creating the model and controller, we need to create a migration. In a terminal—or cmd.exe window, if you’re using Windows—run:

php artisan make:migration create\_articles\_table --create=articles

Now, go to the database/migrations folder and open the file with a name similar to xxxx\_xx\_xx\_xxxxxx\_create\_articles\_table.php.

In the up function of the class, we’ll write this:

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

$table->increments('id');

$table->string('title');

$table->string('body');

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

$table->timestamps();

});

Next, we’ll create an Article model. To do that, run:

php artisan make:model Article

We then create the ArticleController controller by running:

php artisan make:controller ArticleController --resource

Next, we’ll edit the file app/Providers/AppServiceProvider.php and import the Illuminate\Support\Facades\Schema class by adding:

use Illuminate\Support\Facades\Schema

…to the bottom of the imports at the top of the file.

Then, in the boot function, we’ll write:

Schema::defaultStringLength(191);

After all of this is done, we can run:

php artisan migrate

…to apply the migration we created above.

**Laravel Passport Tutorial, Step 2: Create the Necessary Pieces of Middleware**

Here, we will add the pieces of middleware that will be necessary for the API to work.

**JSON Responses**

The first piece needed is the ForceJsonResponse middleware, which will convert all responses to JSON automatically.

To do this, run:

php artisan make:middleware ForceJsonResponse

And this is the handle function of that middleware, in App/Http/Middleware/ForceJsonReponse.php:

public function handle($request, Closure $next)

{

$request->headers->set('Accept', 'application/json');

return $next($request);

}

Next, we’ll add the middleware to our app/Http/Kernel.php file in the $routeMiddleware array:

'json.response' => \App\Http\Middleware\ForceJsonResponse::class,

Then, we’ll also add it to the $middleware array in the same file:

\App\Http\Middleware\ForceJsonResponse::class,

That would make sure that the ForceJsonResponse middleware is run on every request.

**CORS (Cross-origin Resource Sharing)**

To allow the consumers of our [Laravel REST API](https://www.toptal.com/laravel/restful-laravel-api-tutorial) to access it from a different origin, we have to set up CORS. To do that, we’ll create a piece of middleware called Cors.

In a terminal or command prompt, cd into the project root directory and run:

php artisan make:middleware Cors

Then, in app/Http/Middleware/Cors.php, add the following code:

public function handle($request, Closure $next)

{

return $next($request)

->header('Access-Control-Allow-Origin', '\*')

->header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OPTIONS')

->header('Access-Control-Allow-Headers', 'X-Requested-With, Content-Type, X-Token-Auth, Authorization');

}

To load this piece of middleware, we’ll need to add a line to app/Http/Kernel.php’s $routeMiddleware array:

'cors' => \App\Http\Middleware\Cors::class,

Also, we’ll have to add it to the $middleware array as we did for the previous middleware:

\App\Http\Middleware\Cors::class,

After doing that, we’ll append this route group to routes/api.php:

Route::group(['middleware' => ['cors', 'json.response']], function () {

// ...

});

All our API routes will go into that function, as we’ll see below.

**Laravel Passport Tutorial, Step 3: Create User Authentication Controllers for the API**

Now we want to create the authentication controller with login and register functions.

First, we’ll run:

php artisan make:controller Auth/ApiAuthController

Now we’ll import some classes to the file app/Http/Controllers/Auth/ApiAuthController.php. These classes are going to be used in the creation of the login and register functions. We are going to import the classes by adding:

use App\User;

use Illuminate\Support\Facades\Hash;

use Illuminate\Support\Facades\Validator;

use Illuminate\Support\Str;

…to the top of the controller.

Now, to add Laravel API authentication for our users, we are going to create login, logout, and register (signup) functions in the same file.

The register function will look like this:

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|confirmed',

]);

if ($validator->fails())

{

return response(['errors'=>$validator->errors()->all()], 422);

}

$request['password']=Hash::make($request['password']);

$request['remember\_token'] = Str::random(10);

$user = User::create($request->toArray());

$token = $user->createToken('Laravel Password Grant Client')->accessToken;

$response = ['token' => $token];

return response($response, 200);

}

The login function’s like this:

public function login (Request $request) {

$validator = Validator::make($request->all(), [

'email' => 'required|string|email|max:255',

'password' => 'required|string|min:6|confirmed',

]);

if ($validator->fails())

{

return response(['errors'=>$validator->errors()->all()], 422);

}

$user = User::where('email', $request->email)->first();

if ($user) {

if (Hash::check($request->password, $user->password)) {

$token = $user->createToken('Laravel Password Grant Client')->accessToken;

$response = ['token' => $token];

return response($response, 200);

} else {

$response = ["message" => "Password mismatch"];

return response($response, 422);

}

} else {

$response = ["message" =>'User does not exist'];

return response($response, 422);

}

}

And finally, the logout function:

public function logout (Request $request) {

$token = $request->user()->token();

$token->revoke();

$response = ['message' => 'You have been successfully logged out!'];

return response($response, 200);

}

After this, we need to add the login, register, and logout functions to our routes, i.e., [within the route group](https://github.com/thedarkkid/laravel_api_framework/blob/master/routes/api.php#L22) already in the API:

Route::group(['middleware' => ['cors', 'json.response']], function () {

// ...

// public routes

Route::post('/login', 'Auth\ApiAuthController@login')->name('login.api');

Route::post('/register','Auth\ApiAuthController@register')->name('register.api');

Route::post('/logout', 'Auth\ApiAuthController@logout')->name('logout.api');

// ...

});

Lastly, we need to add the HasApiToken trait to the User model. Navigate to app/User and make sure you have:

use HasApiTokens, Notifiable;

…at the top of the class.

**What We Have so Far…**

If we start the application server—i.e., run php artisan serve—and then try to send a GET request to the route /api/user, we should receive the message:

{

"message": "Unauthenticated."

}

This is because we are not authenticated to [access that route](https://github.com/thedarkkid/laravel_api_framework/blob/master/routes/api.php#L18). To make some routes of your choice protected, we can add them to routes/api.php just after the Route::post lines:

Route::middleware('auth:api')->group(function () {

// our routes to be protected will go in here

});

Before moving on, we’ll add the logout route to the auth:api middleware because Laravel uses a token to log the user out—a token which cannot be accessed from outside the auth:api middleware. Our **public** routes look like this:

Route::group(['middleware' => ['cors', 'json.response']], function () {

// ...

// public routes

Route::post('/login', 'Auth\ApiAuthController@login')->name('login.api');

Route::post('/register', 'Auth\ApiAuthController@register')->name('register.api');

// ...

});

Our **protected** routes, on the other hand, look like this:

Route::middleware('auth:api')->group(function () {

// our routes to be protected will go in here

Route::post('/logout', 'Auth\ApiAuthController@logout')->name('logout.api');

});

Now we’ll navigate to the ArticleController we created in app/Http/Controllers/ArticleController.php and delete the create and edit methods in that class. After that, we’ll add the following piece of code, slightly edited, to each remaining function:

$response = ['message' => '<function name> function'];

return response($response, 200);

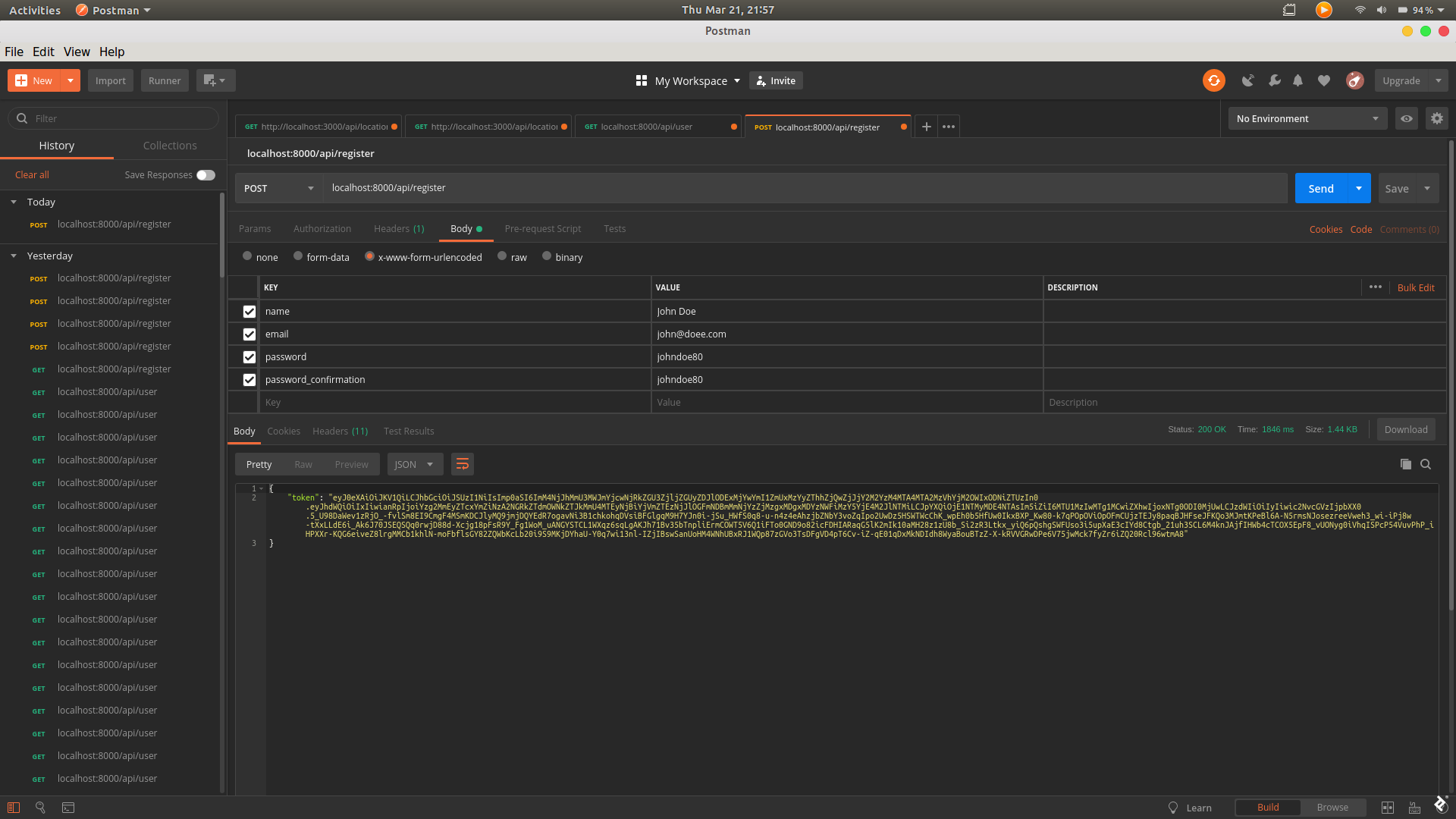
We’ll fill in <function name> as appropriate. For example, the update function will have this as its body:

$response = ['message' => 'update function'];

return response($response, 200);

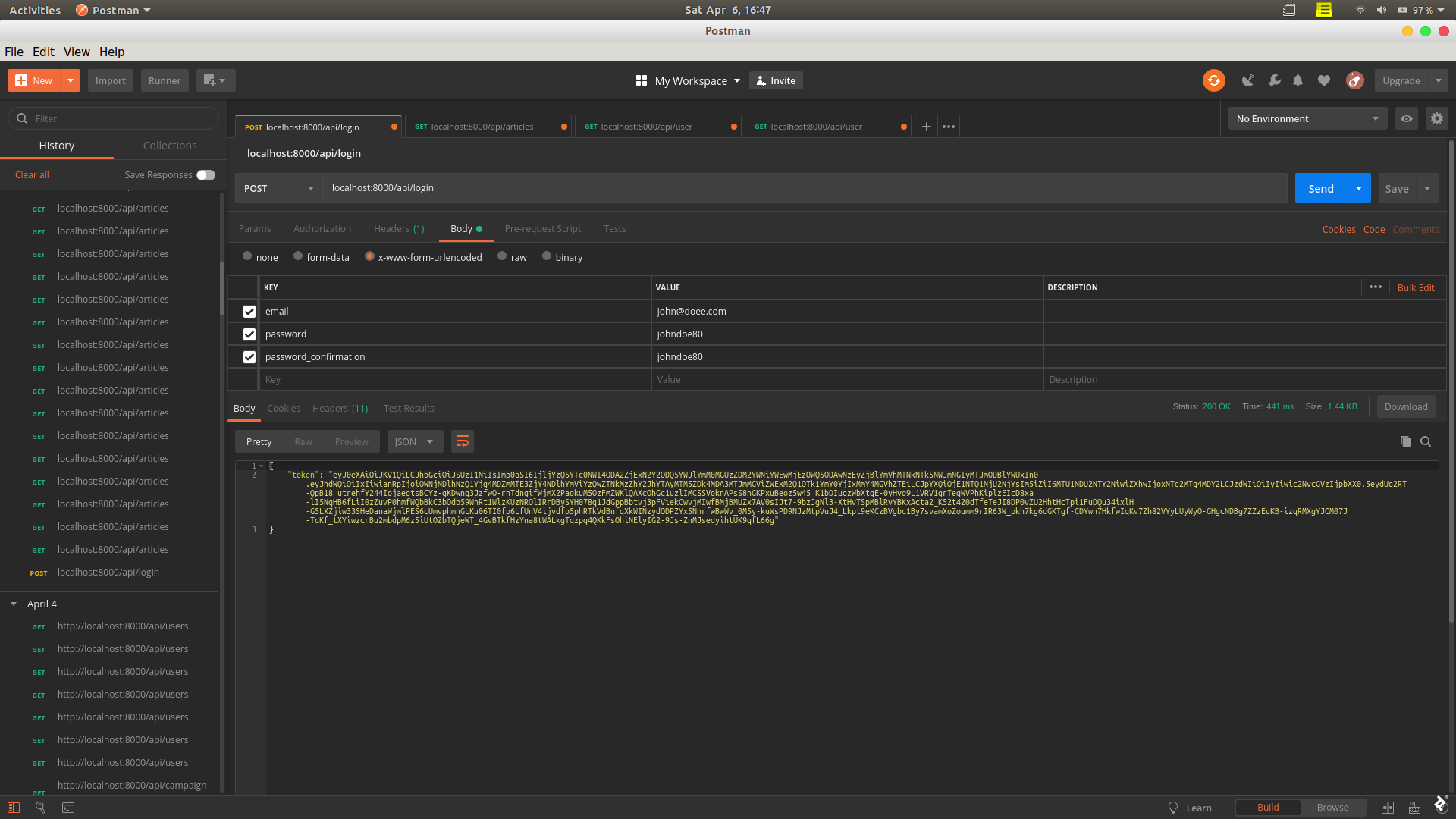
**A Manual Laravel Authentication Test: Creating a User**

To register a user, we’ll send a POST request to /api/register with the following parameters: name, email (which has to be unique), password, and password\_confirmation.



When the user is created, the API will return a token, which we will use in further requests as our means to authentication.

To log in, we’ll send a POST request to /api/login. If our credentials are correct, we will also get a token from our Laravel login API this way.



The authorization token we get returned from this request we can use when we want to access a protected route. In Postman, the “Authorization” tab has a drop-down where the type can be set to “Bearer Token,” after which the token can go into the token field.

The process is quite similar in [Insomnia](https://flywheel.co/articles/using-insomnia-to-test-apis).

cURL users can do the equivalent by passing the parameter -H "Authorization: Bearer <token>", where <token> is the authorization token given from the login or register response.

As with cURL, if developers plan to consume the API using axios or a library of that sort, they can add an Authorization header with value Bearer <token>.

**Laravel Passport Tutorial, Step 4: Create Password Reset Functionality**

Now that basic authentication is done, it’s time to set up a password reset function.

To do this, we can choose to create an api\_auth controller directory, create new custom controllers, and implement the function; or we can edit the auth controllers that we can generate with Laravel. In this case, we’ll edit the auth controllers, since the whole application is an API.

First, we will generate the auth controllers by running:

composer require laravel/ui

php artisan ui vue --auth

We’ll edit the class in app/Http/Controllers/Auth/ForgotPasswordController.php, adding these two methods:

protected function sendResetLinkResponse(Request $request, $response)

{

$response = ['message' => "Password reset email sent"];

return response($response, 200);

}

protected function sendResetLinkFailedResponse(Request $request, $response)

{

$response = "Email could not be sent to this email address";

return response($response, 500);

}

Next, we need to set up the controller that actually resets the password, so we’ll navigate to app/Http/Controllers/Auth/ResetPasswordController.php and override the default functions like this:

protected function resetPassword($user, $password)

{

$user->password = Hash::make($password);

$user->save();

event(new PasswordReset($user));

}

protected function sendResetResponse(Request $request, $response)

{

$response = ['message' => "Password reset successful"];

return response($response, 200);

}

protected function sendResetFailedResponse(Request $request, $response)

{

$response = "Token Invalid";

return response($response, 401);

}

We also need to import some classes in the controller by adding:

use Illuminate\Auth\Events\PasswordReset;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Hash;

…to the top of the controller.

We’ll want to modify which email notification is used, too, because the mail notification that comes with Laravel does not use API tokens for authorization. We can create a new one under app/Notifications by running this command:

php artisan make:notification MailResetPasswordNotification

We’ll need to edit the file app/Notifications/MailResetPasswordNotification.php to look like this:

<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;

use Illuminate\Notifications\Messages\MailMessage;

use Illuminate\Auth\Notifications\ResetPassword;

use Illuminate\Support\Facades\Lang;

class MailResetPasswordNotification extends ResetPassword

{

use Queueable;

protected $pageUrl;

public $token;

/\*\*

\* Create a new notification instance.

\*

\* @param $token

\*/

public function \_\_construct($token)

{

parent::\_\_construct($token);

$this->pageUrl = 'localhost:8080';

// we can set whatever we want here, or use .env to set environmental variables

}

/\*\*

\* 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)

{

if (static::$toMailCallback) {

return call\_user\_func(static::$toMailCallback, $notifiable, $this->token);

}

return (new MailMessage)

->subject(Lang::getFromJson('Reset application Password'))

->line(Lang::getFromJson('You are receiving this email because we received a password reset request for your account.'))

->action(Lang::getFromJson('Reset Password'), $this->pageUrl."?token=".$this->token)

->line(Lang::getFromJson('This password reset link will expire in :count minutes.', ['count' => config('auth.passwords.users.expire')]))

->line(Lang::getFromJson('If you did not request a password reset, no further action is required.'));

}

/\*\*

\* Get the array representation of the notification.

\*

\* @param mixed $notifiable

\* @return array

\*/

public function toArray($notifiable)

{

return [

//

];

}

}

To make use of this new notification, we need to override the sendPasswordResetNotification method that User inherits from the Authenticatable class. All we need to do is add this to app/User.php:

public function sendPasswordResetNotification($token)

{

$this->notify(new \App\Notifications\MailResetPasswordNotification($token));

}

With a properly functioning mail setup, notifications should be working at this point.

All that is left now is user access control.

**Laravel Passport Tutorial, Step 5: Create Access Control Middleware**

Before we create access control middleware, we will need to update the user table to have a column named type, which will be used to determine the user level: type 0 is a normal user, type 1 is an admin, and type 2 is a super-admin.

To update the user table, we have to create a migration by running this:

php artisan make:migration update\_users\_table\_to\_include\_type --table=users

In the newly created file of the form database/migrations/[timestamp]\_update\_users\_table.php, we’ll need to update the up and down functions to add and remove the type column, respectively:

public function up()

{

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

$table->integer('type');

});

}

/\*\*

\* Reverse the migrations.

\*

\* @return void

\*/

public function down()

{

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

$table->dropIfExists('type');

});

}

Next, we’ll run php artisan migrate. Once this is done, we have to edit our register function in the ApiAuthController.php file, adding this just before the line with $user = User::create($request->toArray());:

$request['type'] = $request['type'] ? $request['type'] : 0;

Also, we’ll need to add this line to the $validator array:

'type' => 'integer',

The first of these two edits will make all registered users “normal users” by default, i.e., if no user type is entered.

**The Access Control Middleware Itself**

Now we’re in a position to create two pieces of middleware to use for access control: one for admins and one for super-admins.

So we’ll run:

php artisan make:middleware AdminAuth

php artisan make:middleware SuperAdminAuth

First, we’ll navigate to app/Http/Middleware/AdminAuth.php and import Illuminate\Support\Facades\Auth, then edit the handle function like so:

public function handle($request, Closure $next)

{

if (Auth::guard('api')->check() && $request->user()->type >= 1) {

return $next($request);

} else {

$message = ["message" => "Permission Denied"];

return response($message, 401);

}

}

We’ll also need to edit the handle function in app/Http/Middleware/SuperAdminAuth.php:

public function handle($request, Closure $next)

{

if (Auth::guard('api')->check() && $request->user()->type >= 2) {

return $next($request);

} else {

$message = ["message" => "Permission Denied"];

return response($message, 401);

}

}

You should also import the Auth class at the top of both files by adding:

use Illuminate\Support\Facades\Auth;

…to the bottom of the imports found there.

In order to use our new middleware, we’ll reference both classes in the kernel—i.e., in app/Http/Kernel.php—by adding the following lines to the $routeMiddleware array:

'api.admin' => \App\Http\Middleware\AdminAuth::class,

'api.superAdmin' => \App\Http\Middleware\SuperAdminAuth::class,

If developers want to use the middleware in a given route, all you need to do is add it to the route function like this:

Route::post('route','Controller@method')->middleware('<middleware-name-here>');

<middleware-name-here> in this case can be api.admin, api.superAdmin, etc., as appropriate.

That’s all that’s needed to create our middleware.

**Putting It All Together**

In order to test that our authentication and access control is working, there are some additional steps to go through.

**Testing Laravel Authentication and Access Control: Step 1**

We need to modify the ArticleController’s index function and register the route. (In real-world projects, we would use PHPUnit and do this as part of an automated test. Here, we’re manually adding a route for testing purposes—it can be removed afterward.)

We’ll navigate to the ArticleController controller at app/Http/Controllers/ArticleController and modify the index function to look like this:

public function index()

{

$response = ['message' => 'article index'];

return response($response, 200);

}

Next, we’ll register the function in a route by going to the routes/api.php file and appending this:

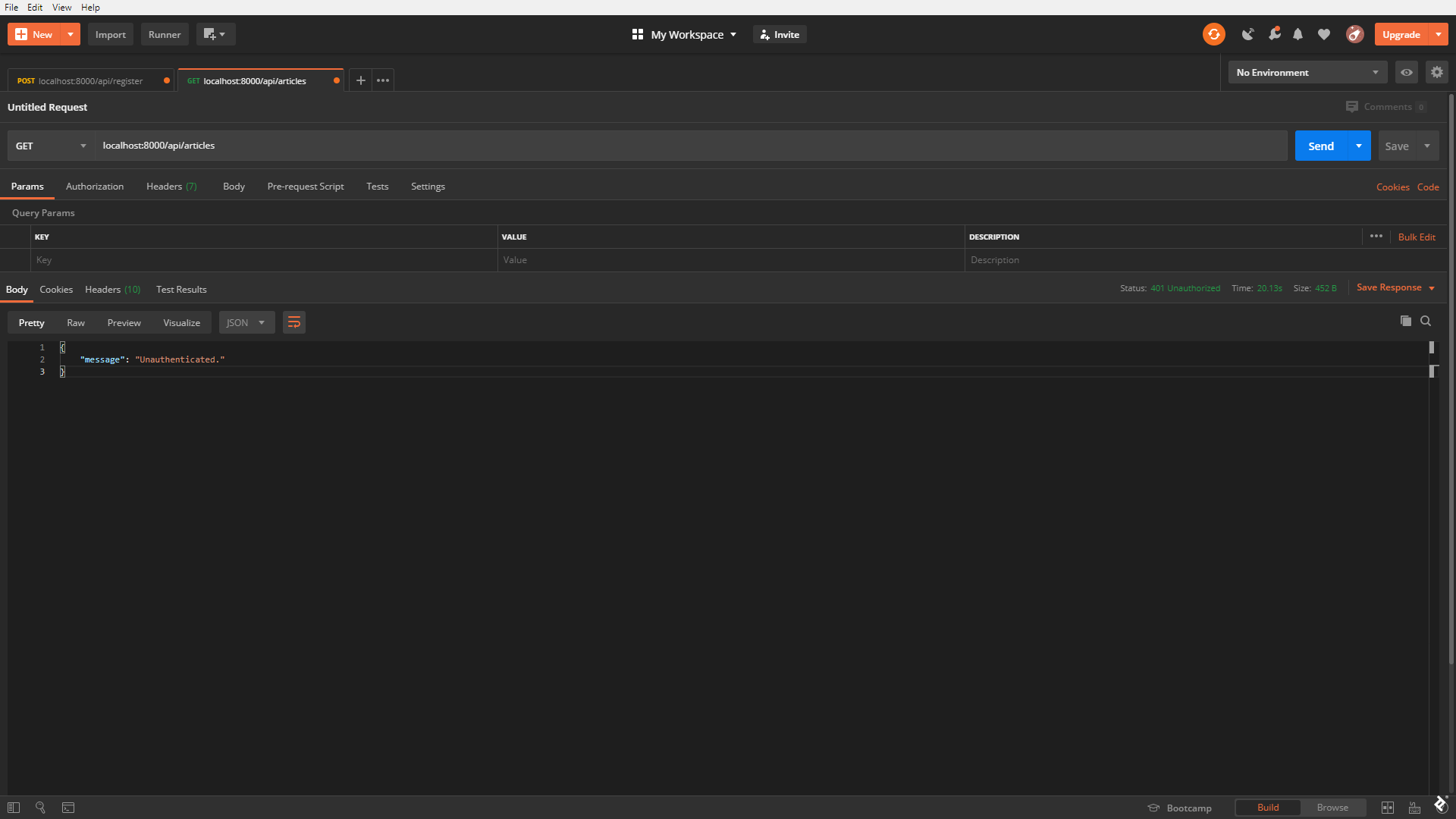
Route::middleware('auth:api')->group(function () {

Route::get('/articles', 'ArticleController@index')->name('articles');

});

**Testing Laravel Authentication and Access Control: Step 2**

Now we can try to access the route without an authentication token. We should receive an authentication error.

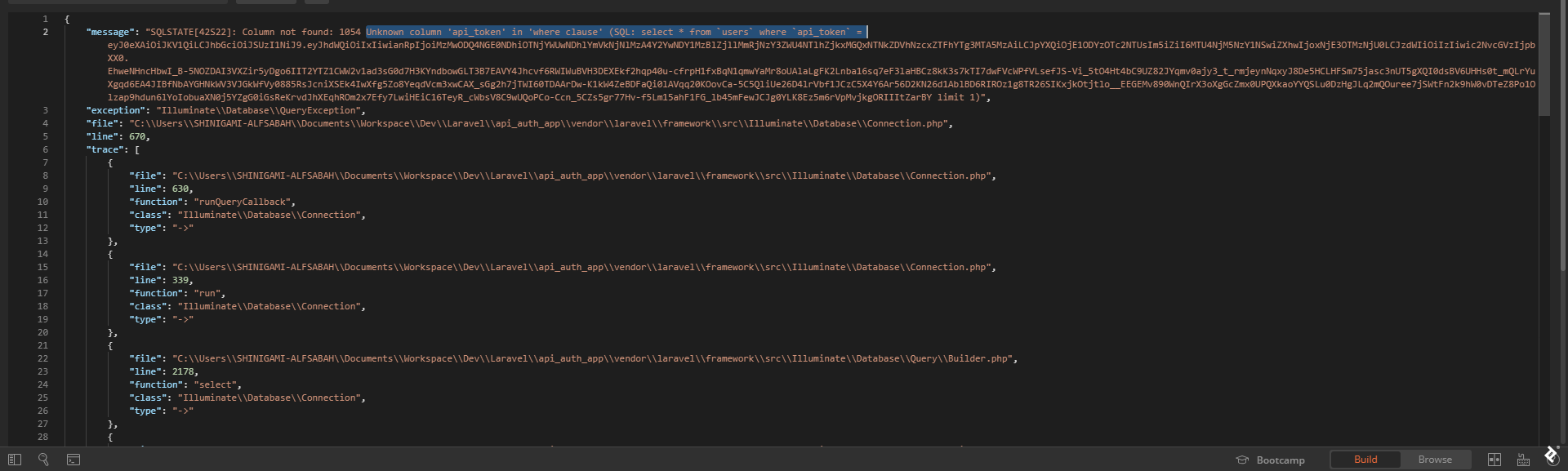


**Testing Laravel Authentication and Access Control: Step 3**

We can also try to access the same route with an authorization token (the one we got from registering or logging in earlier in this article).

Sometimes, this might cause an error similar to this:

Unknown column 'api\_token' in 'where clause' (SQL: select \* from `users` where `api\_token` = ...



If this happens, developers should [make sure](https://stackoverflow.com/a/53742943/16777) to have run a Passport migration and have ['guards']['api']['driver'] set to passport in config/auth.php:

'guards' => [

'web' => [

'driver' => 'session',

'provider' => 'users',

],

'api' => [

'driver' => 'passport',

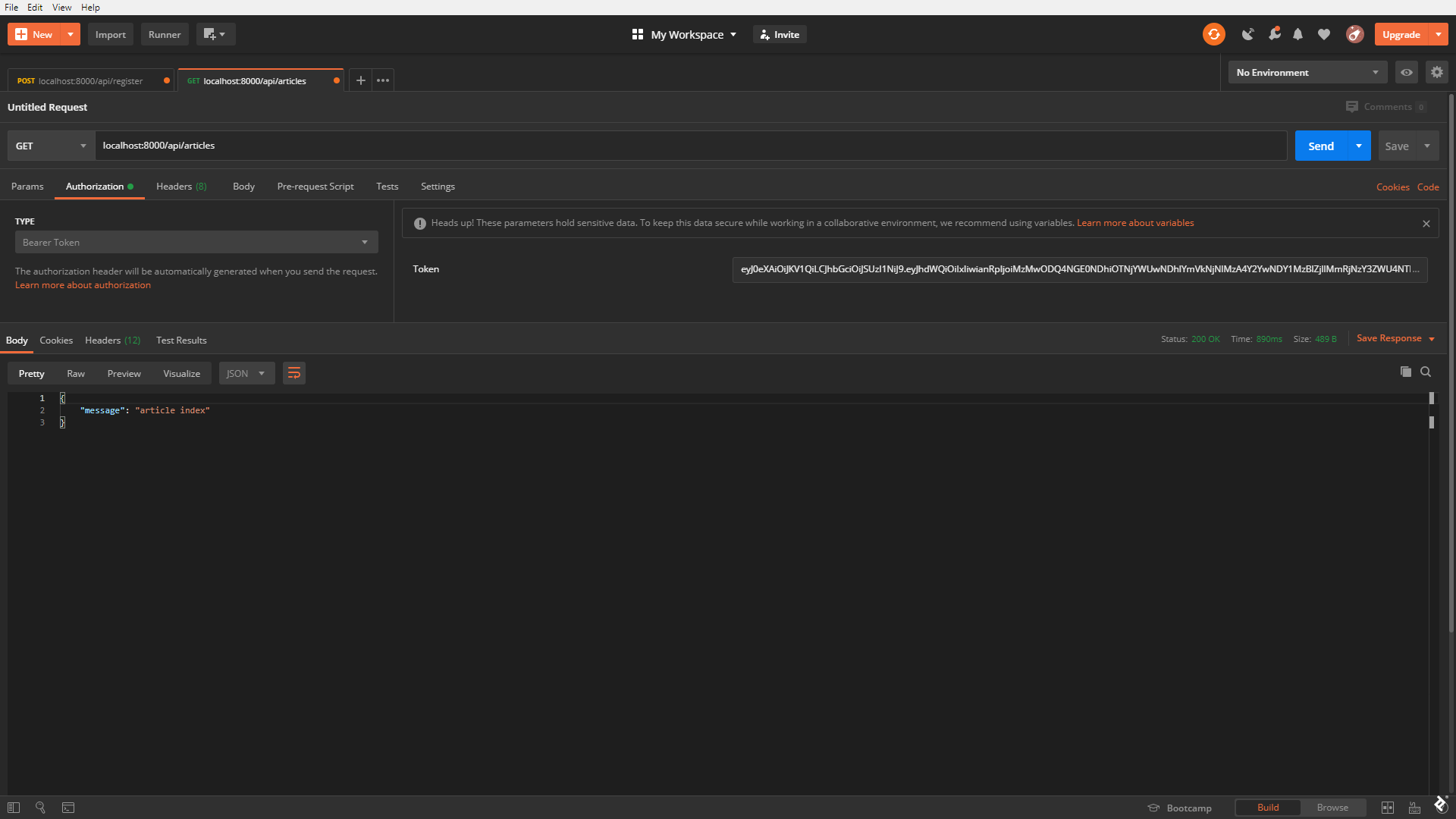
'provider' => 'users',

],

],

After that, the configuration cache needs updating as well.

Once that is fixed, we should have access to the route.

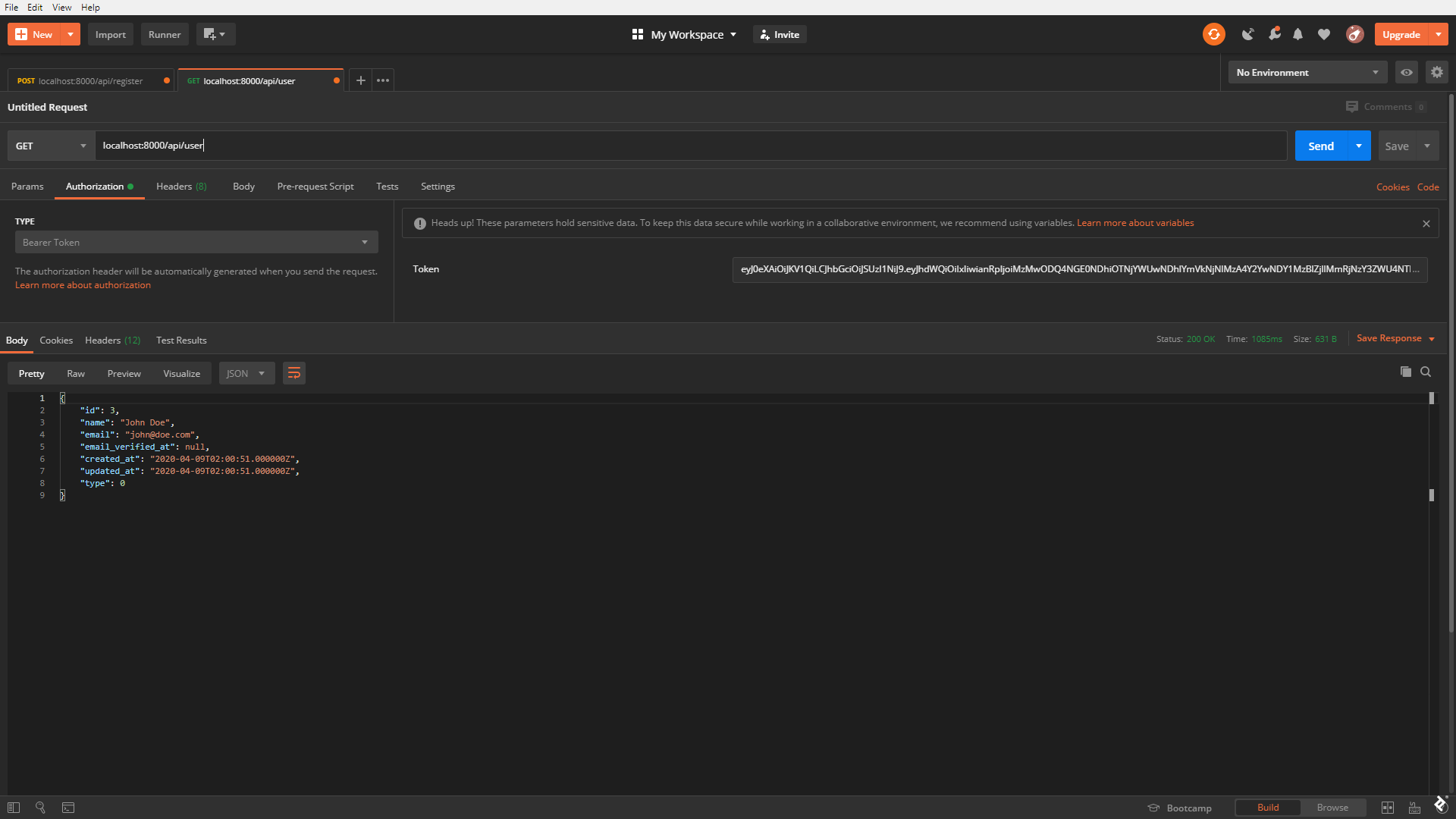


**Testing Laravel Authentication and Access Control: Step 4**

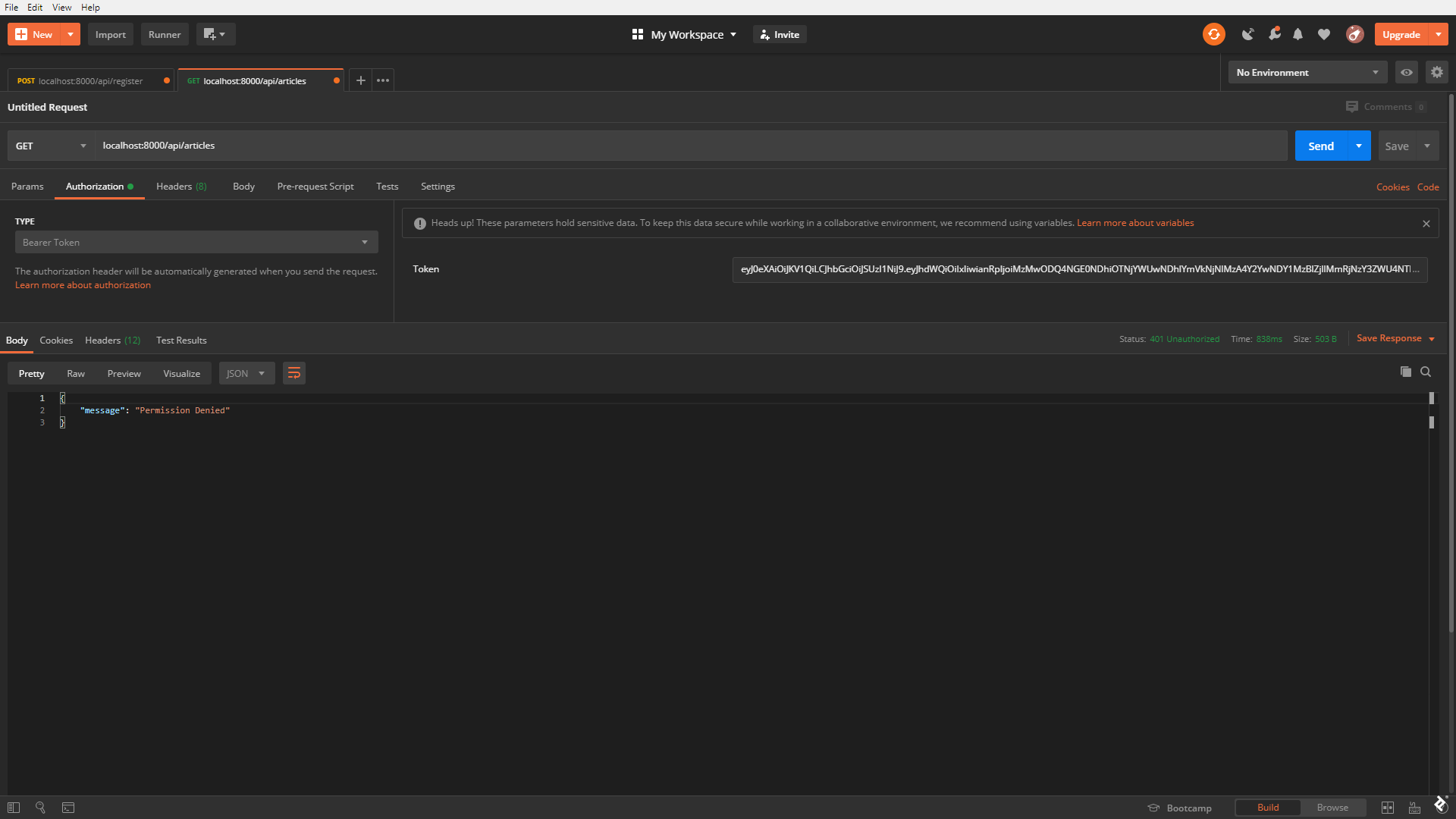
It’s time to test access control. Let’s append ->middleware('api.admin') to the articles route, so it looks like this:

Route::get('/articles', 'ArticleController@index')->middleware('api.admin')->name('articles');

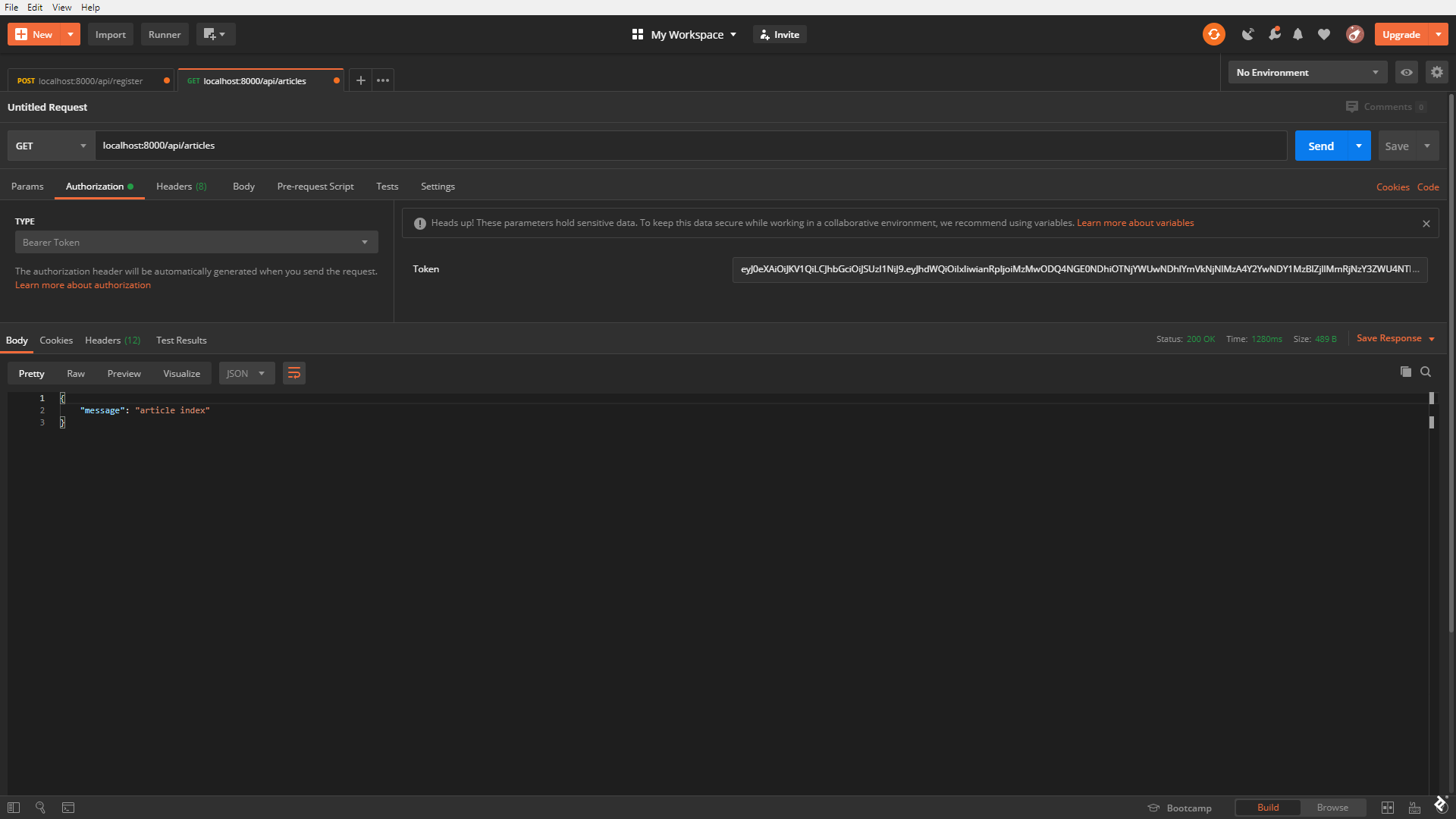
We made it such that a newly created user is automatically assigned type 0, as we can see via the api/user route.



Because of that, we should get an error trying to access the articles endpoint as such a user.



For the purpose of testing, let’s modify the user in the database to have a type of 1. After verifying that change via the api/user route again, we’re ready to try again to GET the /articles/ route.



It works perfectly.