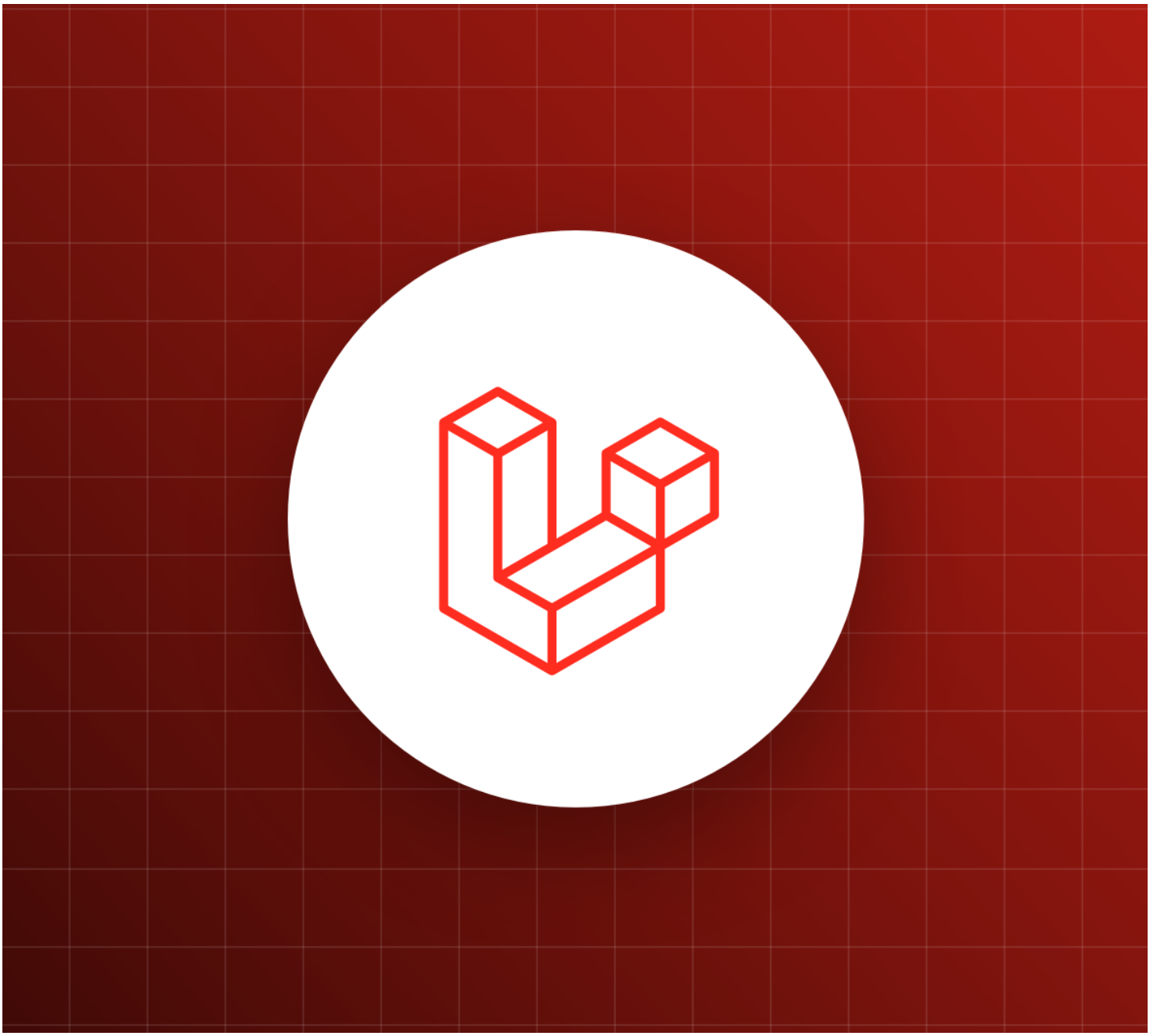


LUMEN

Developing RESTful APIs with Lumen (A PHP Micro-framework)

Lumen is a PHP micro-framework built to deliver microservices and blazing fast APIs. Learn how to build and secure RESTful APIs with Lumen

Prosper Otemuyiwa

December 26, 2017

TL;DR: In this tutorial, I'll show you how easy it is to build and secure an API with Lumen. Check out the repo to get the code.

Lumen is an open-source PHP micro-framework created by Taylor Otwell as an alternative to Laravel to meet the demand of lightweight installations that are faster than existing PHP micro-frameworks such as Slim and Silex. With Lumen, you can build lightning-fast microservices and APIs that can support your Laravel applications.

Lumen Features and Architecture

Lumen utilizes the Illuminate components that power the Laravel framework. As such, Lumen is built to painlessly upgrade directly to Laravel when needed; for example, when you discover that you need more features out of the box than what Lumen offers.

These are some of the built-in features of Lumen:

- **Routing** is provided out of the box via Fast Route, a library that provides a fast implementation of a regular expression based router.
- **Authentication** does not support session state. However, incoming requests are authenticated via stateless mechanisms such as tokens.
- **Caching** is implemented the same as in Laravel. Cache drivers such as *Database*, *Memcached*, and *Redis* are supported. For example, you can install the `illuminate/redis` package via Composer to use a Redis cache with Lumen.
- **Errors and Logging** are implemented via the Monolog library, which provides support for various log handlers.
- **Queuing** services are similar to the ones offered by Laravel. A unified API is provided across a variety of different queue back-ends.

- **Events** provide a simple observer implementation that allows you to subscribe and listen for events in your application.

- **Bootstraping** processes are located in a single file.

> "Lumen is an amazing PHP microframework that offers a painless upgrade path to Laravel."

Tweet This

Lumen Key Requirements

To use Lumen, you need to have the following tools installed on your machine:

- **PHP**: Make sure `PHP >= 7.1.3` is installed on your machine. Furthermore, ensure that the following PHP extensions are installed. `OpenSSL`, `PDO` and `Mbstring`.
- **Composer**: Navigate to the composer website and install it on your machine. Composer is needed to install Lumen's dependencies.

> **Note: You'll need MySQL for this tutorial. Navigate to the mysql website and install the community server edition. If you are using a Mac, I'll recommend following these instructions. To avoid micromanaging from the terminal, I'll also recommend installing a MySQL GUI, Sequel Pro.**

Building a Fast Authors API Rapidly With Lumen

At Auth0, we have many technical writers, otherwise known as authors. A directive has been given to developing an app to manage Auth0 authors. The front-end app will be built with ReactJS. However, it needs to pull data from a source and also push to it. Yes, we need an API!

This is what we need the API to do:

- Get all authors.
- Get one author.
- Add a new author.
- Edit an author.
- Delete an author.

Let's flesh out the possible endpoints for this API. Given some *authors* resource, we'll have the following endpoints:

- Get all authors - `GET /api/authors`
- Get one author - `GET /api/authors/23`
- Create an author - `POST /api/authors`
- Edit an author - `PUT /api/authors/23`
- Delete an author - `DELETE /api/authors/23`

What will be the author attributes? Let's flesh it out like we did the endpoints.

- Author: `name`, `email`, `twitter`, `github`, `location`, and `latest_article_published`.

Install Lumen

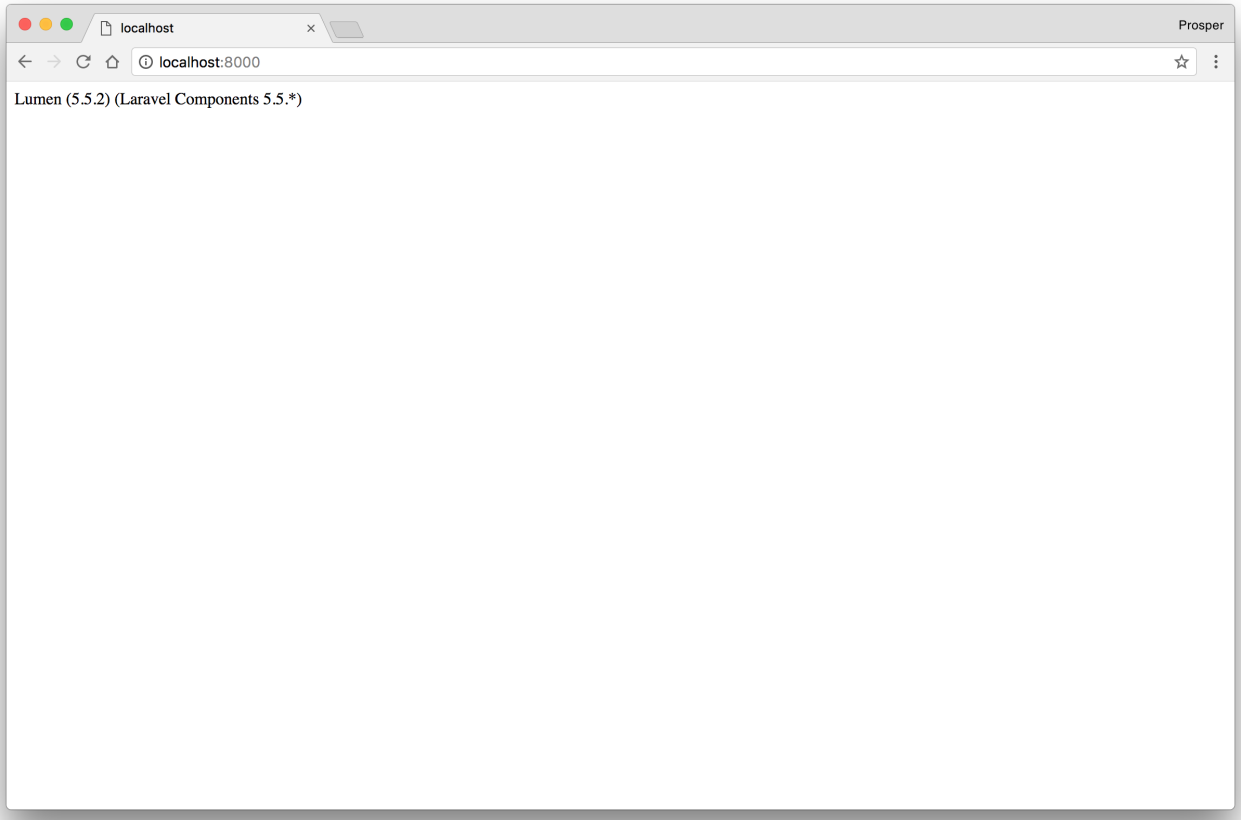
Run the following command in your terminal to create a new project with Lumen:

```
composer create-project --prefer-dist laravel/lumen authors
```

`cd` into the newly created project.

```
cd authors
```

Now, run `php -S localhost:8000 -t public` to serve the project. Head over to your browser. You should see the index page like so:



Authors Index

Activate Eloquent and Facades

As I mentioned earlier, the entire bootstrap process is located in a single file.

Open up the `bootstrap/app.php` and uncomment this line, `// app->withEloquent`. Once uncommented, Lumen hooks the Eloquent ORM with your database using the connections configured in the `.env` file.

> **Make sure you set the right details for your database in the .env file.**

Next uncomment this line `//$app->withFacades();`, which allows us to make use of Facades in our project.

Setup Database, Models and Migrations

At the time of this writing, Lumen supports four database systems: MySQL, Postgres, SQLite, and SQL Server. We are making use of MySQL in this tutorial. First, we'll create a migration for the Authors table.

> **Migrations are like version control for your database, allowing your team to easily modify and share the application's database schema.**

Run the command below in the terminal to create the `Authors` table migration:

```
php artisan make:migration create_authors_table
```

The new migration will be placed in your `database/migrations` directory. Each migration file name contains a timestamp which allows Lumen to determine the order of the migrations. Next, we'll modify the recently created migration to accommodate the `Authors` attributes.

Open up the migration file and modify it like so:

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateAuthorsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('authors', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email');
            $table->string('github');
```

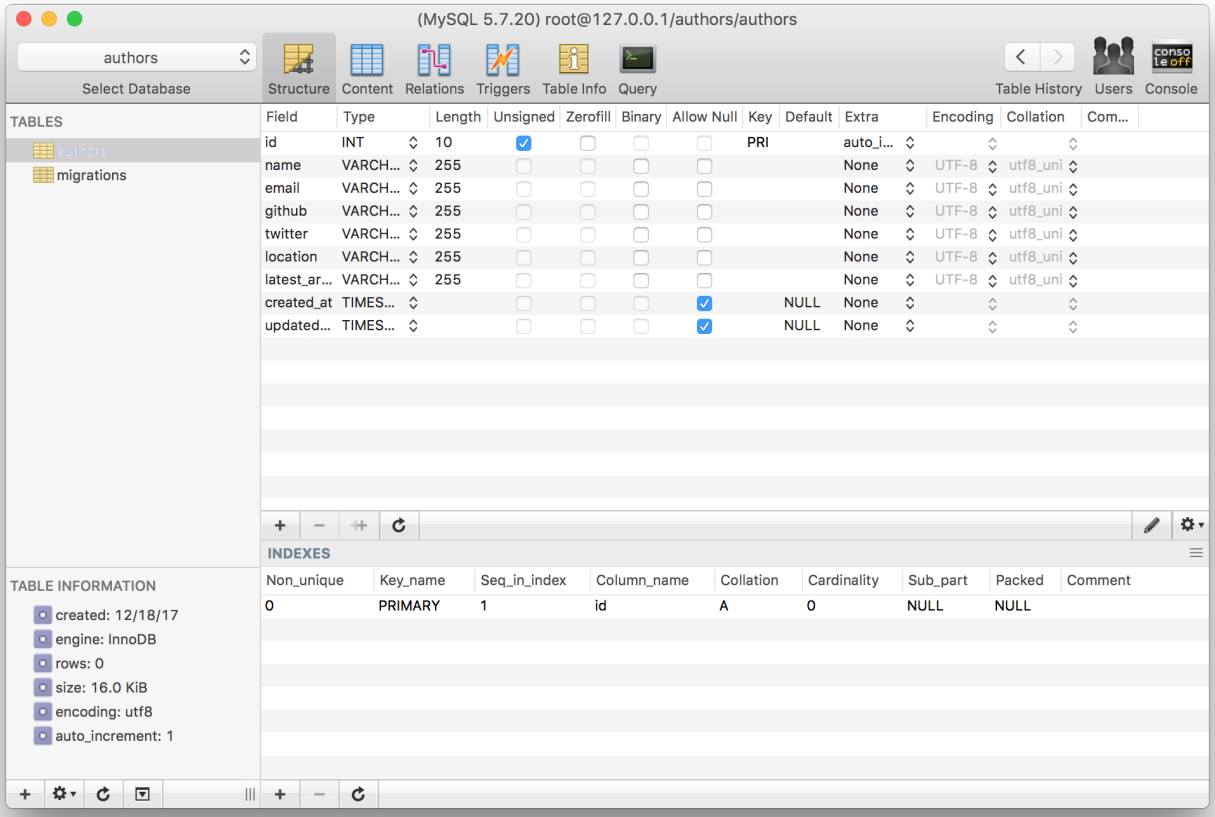
```
            $table->string('twitter');
            $table->string('location');
            $table->string('latest_article_published');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('authors');
    }
}
```

Here we're just adding a few extra columns to the `authors` table such as social handles, location, and a field for the `last_article_published`.

Now, go ahead and run the migration like so:

```
php artisan migrate
```

Check your database. You should now have the `authors` and `migrations` tables present.



Let's create the `Author` model. Create an `app/Author.php` file and add the code below to it:

app/Author.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Author extends Model
{

    /**
     * The attributes that are mass assignable.
     *
```

```
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'github', 'twitter', 'location', 'latest_art
    ];

    /**
     * The attributes excluded from the model's JSON form.
     *
     * @var array
     */
    protected $hidden = [];
}
```

In the code above, we made the author attributes mass assignable.

Set up Routes

Routing is fairly straight-forward. Open up `routes/web.php` and modify it like so:

```
<?php

/*
|--------------------------------------------------------------------------
| Application Routes
|--------------------------------------------------------------------------
|
| Here is where you can register all of the routes for an application
| It is a breeze. Simply tell Lumen the URIs it should respond to
| and give it the Closure to call when that URI is requested.
|
*/

$router->get('/', function () use ($router) {
```

```
    return $router->app->version();
});

$router->group(['prefix' => 'api'], function () use ($router) {
  $router->get('authors',  ['uses' => 'AuthorController@showAllAuthor

  $router->get('authors/{id}', ['uses' => 'AuthorController@showOneAu

  $router->post('authors', ['uses' => 'AuthorController@create']);

  $router->delete('authors/{id}', ['uses' => 'AuthorController@delete

  $router->put('authors/{id}', ['uses' => 'AuthorController@update'])
});
```

In the code above, we have abstracted the functionality for each route into a controller, `AuthorController`. Route groups allow you to share route attributes, such as middleware or namespaces, across a large number of routes without needing to define those attributes on each route. Therefore, every route will have a prefix of `/api`. Next, let's create the Author Controller.

Set up Author Controller

Create a new file, `AuthorController.php` in `app/Http/Controllers` directory and add the following code to it like so:

```
<?php

namespace App\Http\Controllers;

use App\Author;
use Illuminate\Http\Request;

class AuthorController extends Controller
```

```
{

    public function showAllAuthors()
    {
        return response()->json(Author::all());
    }

    public function showOneAuthor($id)
    {
        return response()->json(Author::find($id));
    }

    public function create(Request $request)
    {
        $author = Author::create($request->all());

        return response()->json($author, 201);
    }

    public function update($id, Request $request)
    {
        $author = Author::findOrFail($id);
        $author->update($request->all());

        return response()->json($author, 200);
    }

    public function delete($id)
    {
        Author::findOrFail($id)->delete();
        return response('Deleted Successfully', 200);
    }
}
```

Let's analyze the code above. First, we have `use App\Author`, which allowed us to require the `Author` model that we created earlier.

Next we've created the following five methods:

- `showAllAuthors` - /GET
- `showOneAuthor` - /GET
- `create` - /POST
- `update` - /PUT
- `delete` - /DELETE

These will allow us to use that `Author` model to interact with author data. For example, if you make a POST request to `/api/authors` API endpoint, the `create` function will be invoked and a new entry will be added to the `authors` table.

Author controller method overview:

- `showAllAuthors` - checks for all the author resources
- `create` - creates a new author resource
- `showOneAuthor` - checks for a single author resource
- `update` - checks if an author resource exists and allows the resource to be updated
- `delete` - checks if an author resource exists and deletes it

Controller responses:

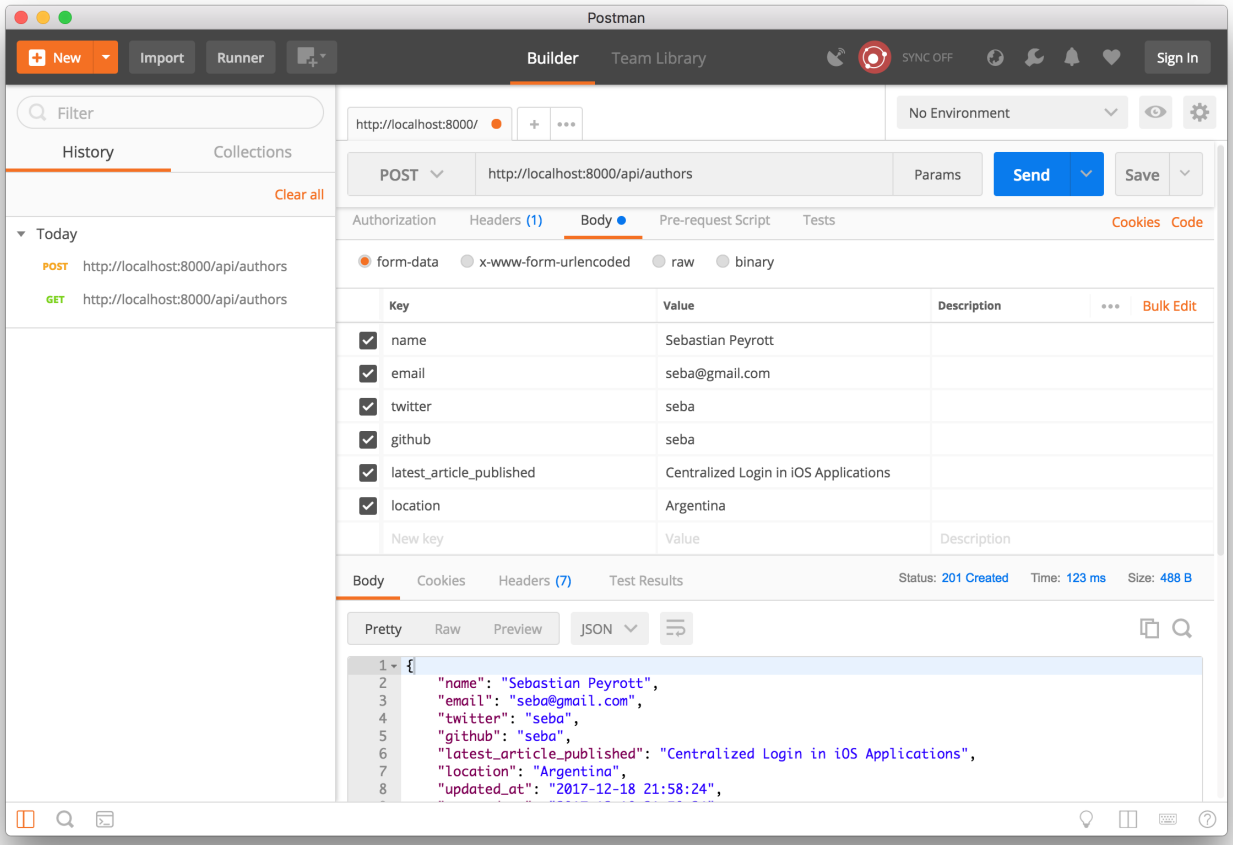
- `response()` - global helper function that obtains an instance of the response factory
- `response()->json()` - returns the response in JSON format.
- `200` - HTTP status code that indicates the request was successful.
- `201` - HTTP status code that indicates a new resource has just been created.
- `findOrFail` - throws a `ModelNotFoundException` if no result is not found.

Finally, test the API routes with Postman.

Author POST operation - `POST http://localhost:8000/api/authors`

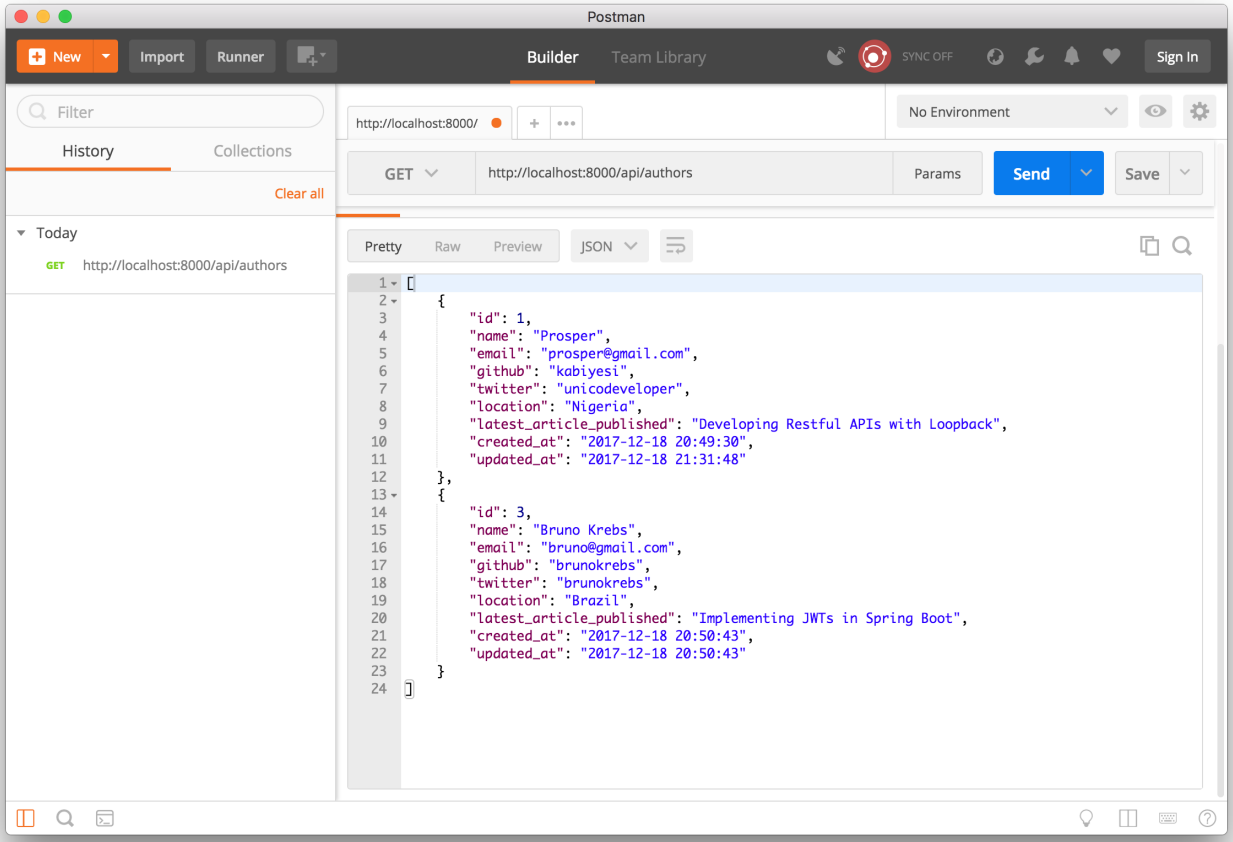
Make sure you have selected `POST` from the dropdown and then you can fill the form data in by clicking on `Body` and then selecting `form-data`. Fill in a value for `name`, `email`, etc to

create a new author.



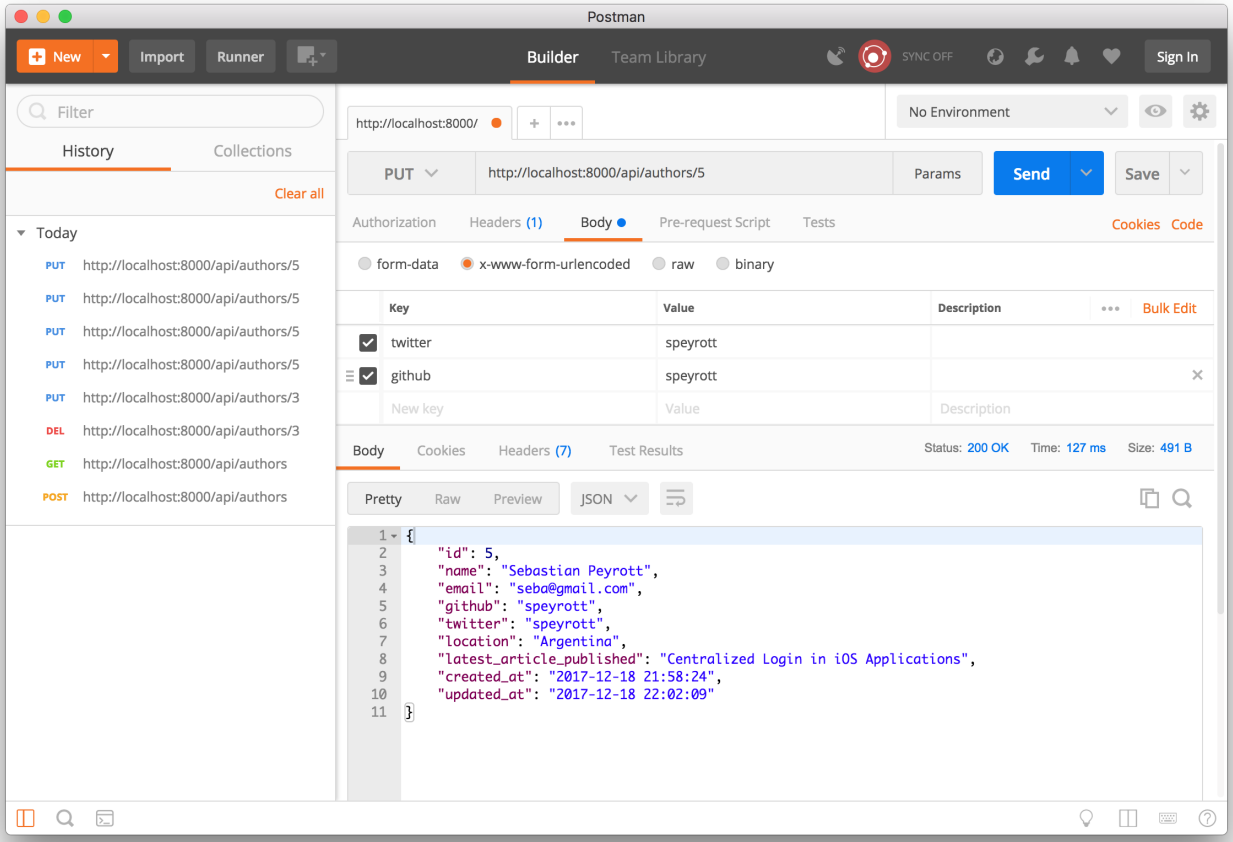
Author GET operation - `GET http://localhost:8000/api/authors`

You should now see an array of objects including the author you just created plus any others in the database.



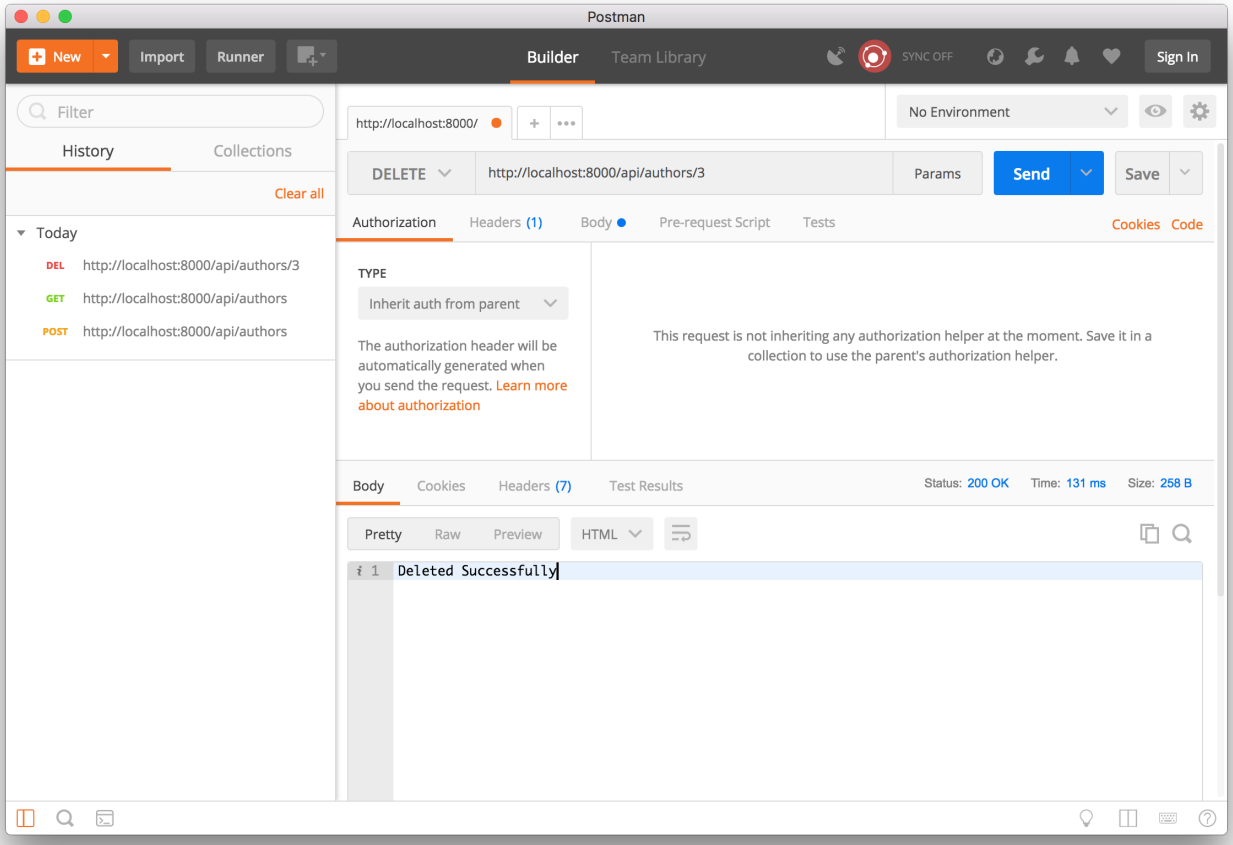
Author PUT operation

The `PUT` operation allows us to edit an existing author. Notice the author's `id` in the URL.



Author DELETE operation

Finally we can delete a specific author as well.



Now we have a working API. Awesome!

Lumen API Validation

When developing applications, **never trust the user.** Always validate incoming data.

In Lumen, it's very easy to validate your application's incoming data. Lumen provides access to the `$this->validate` helper method from within Route closures.

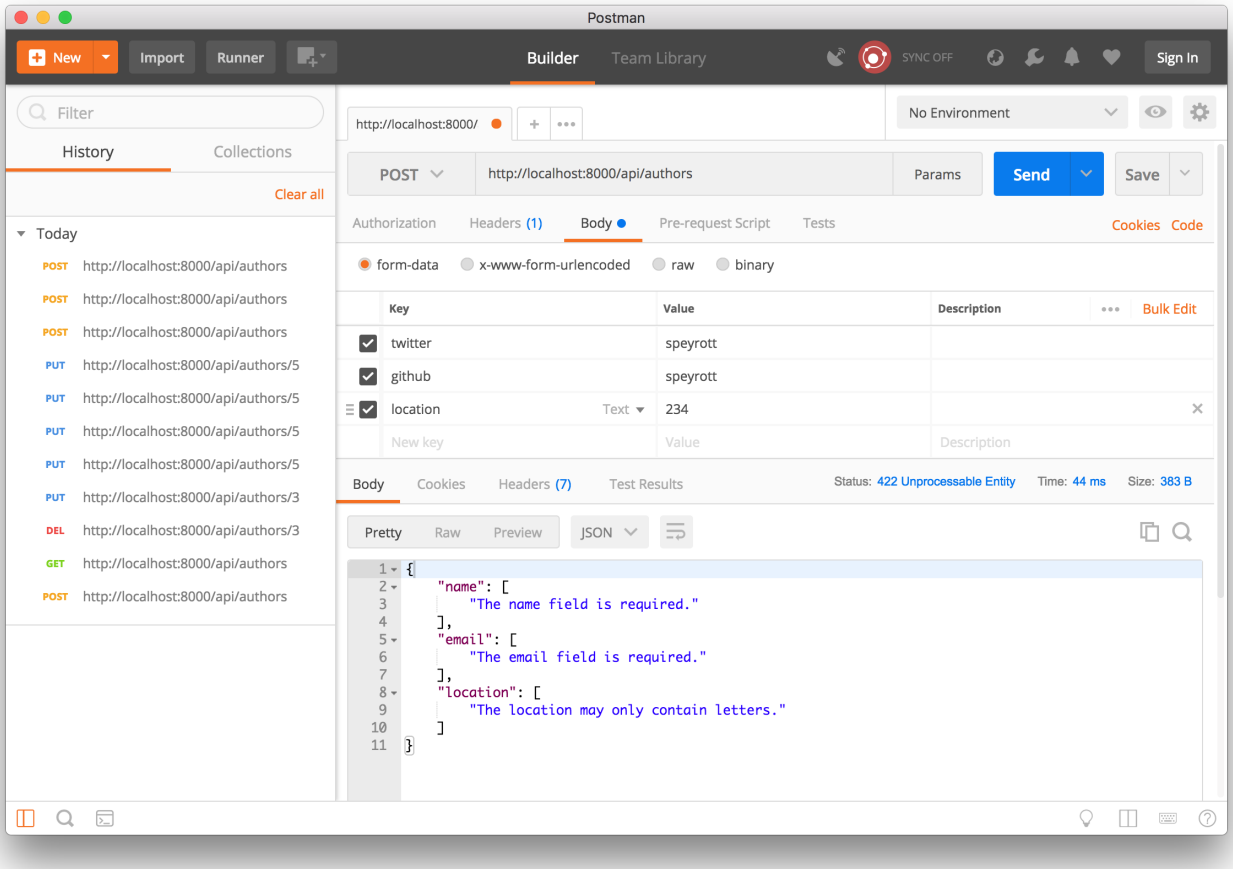
Currently in our API, we're not checking what people are sending through to our `create` method. Let's fix that now.

Open up the `AuthorController` file and modify the `create` method like this:

```
// ...
 public function create(Request $request)
    {
```

```
        $this->validate($request, [
            'name' => 'required',
            'email' => 'required|email|unique:users',
            'location' => 'required|alpha'
        ]);

        $author = Author::create($request->all());

        return response()->json($author, 201);
    }
// ...
```

Now test the API POST route with Postman.



It validated the incoming requests and returned the appropriate error message.

- *name*, *email*, and *location* were required. In testing the API, *name* and *email* were not provided.
- *email* was required to be in email format.
- *location* was required to be entirely alphabetic characters, `alpha`. Nothing more. Numbers were provided as the value for *location*.

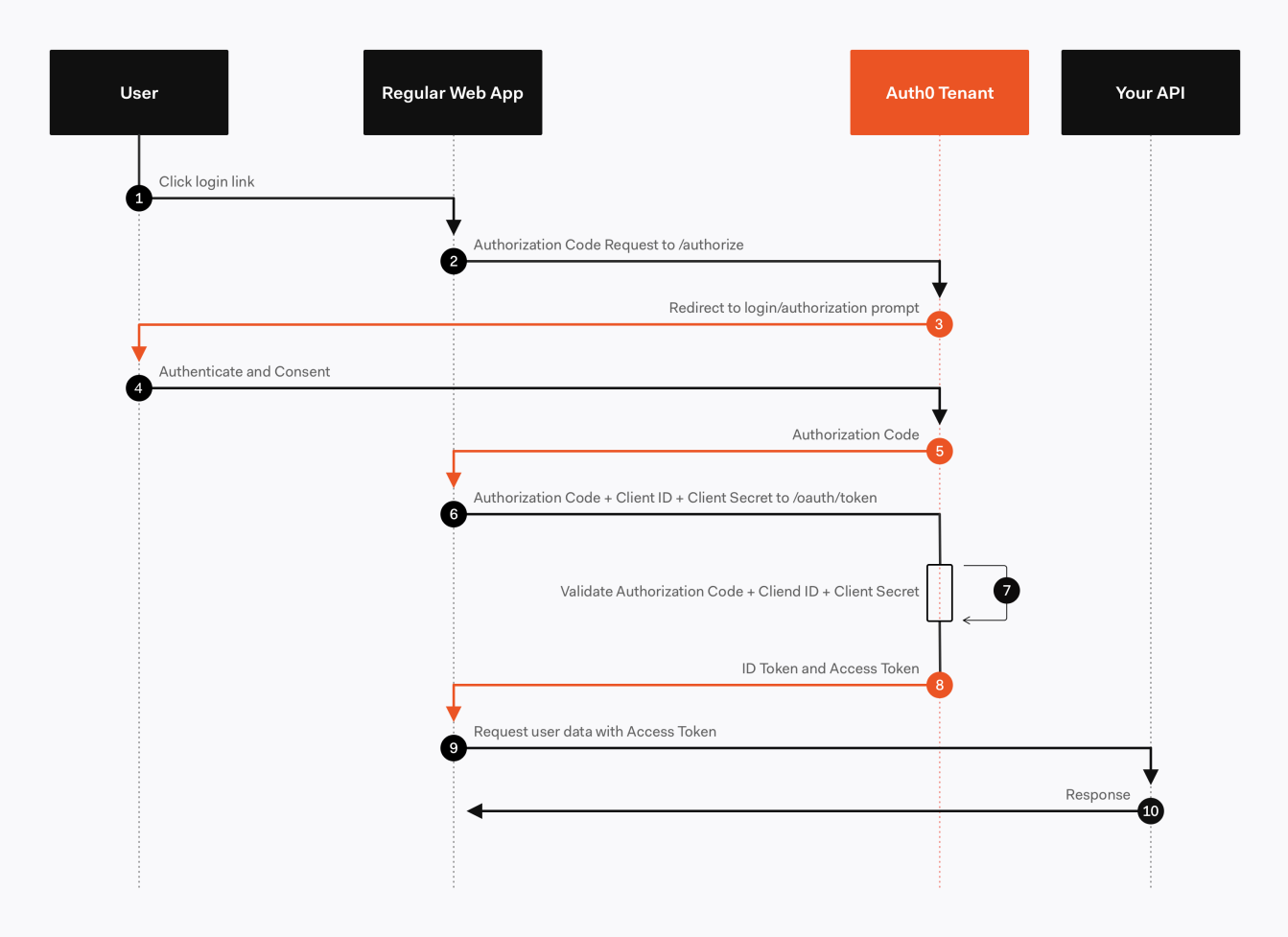
> **Note: Always validate incoming data. Never trust your users!**

Check out a plethora of validation rules that you can use with Lumen.

Securing the Authors API with Auth0

Right now, an application can make requests to any of the endpoints present in our API. In a real-world scenario, we would want to restrict our API so that only certain authorized users have the ability to do this. A few things need to happen here.

1. A user signs in with their credentials (to prove who they are, i.e. **authenticate**)
2. If the user is authorized to use the API, the application is issued an API access token
3. Whenever an API request is made, the application will send that API access token along with the request
4. If the access token is valid, the API will respond with the requested data



In this tutorial, we're going to focus on what happens in **step 4** of that list (step 9-10 in the diagram). Since we're only building the backend API here, you'll need to create a separate front-end to accomplish the first two steps. Here is an awesome example of how you can do that using Auth0 with React.

For now, let's focus on generating access tokens using JSON Web Tokens.

JSON Web Token, commonly known as JWT, is an open standart for creating JSON-based access tokens that make some claim, usually authorizing a user or exchanging information. This technology has gained popularity over the past few years because it enables backends to accept requests simply by validating the contents of these JWTs.

JWTs can be used for authorization or for information exchange. In this tutorial, we'll be using JWTs to grant authorization to applications (users) using our API.

Whenever the user wants to access a protected route or resource (an endpoint), the user agent must send the JWT, usually in the *Authorization* header using the Bearer schema, along with the request.

When the API receives a request with an access token, the first thing it needs to do is **validate the token**. If the validation fails, then the request must be rejected.

We will make use of Auth0 to issue our access tokens. With Auth0, we only have to write a few lines of code to get an in-depth identity management solution which includes:

- Single sign-on
- User management
- Support for social identity providers (like Facebook, GitHub, Twitter, etc.)
- Enterprise (Active Directory, LDAP, SAML, etc.)
- Your own database of users

If you haven't done so yet, this is a good time to sign up for a free Auth0 account.

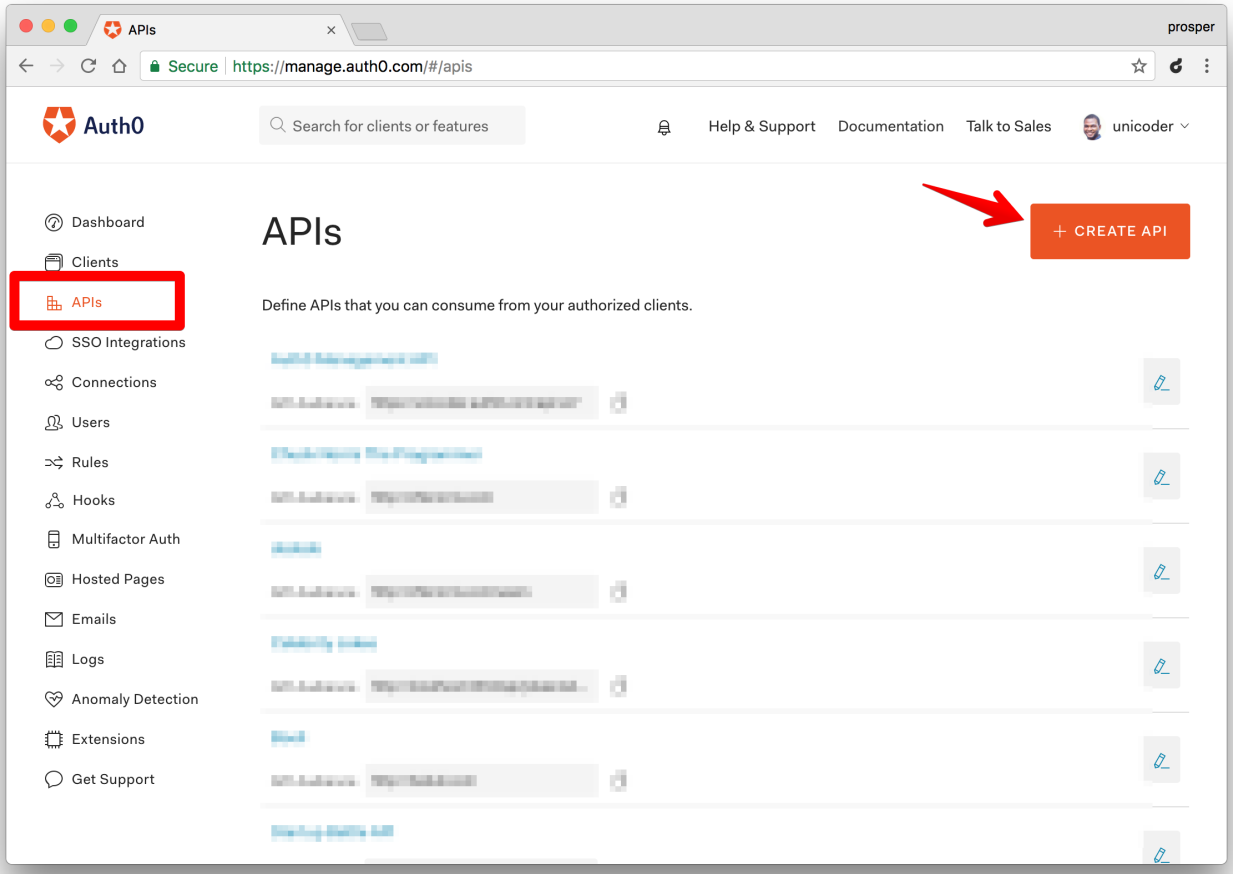
Once you have your Auth0 account, go ahead and create a new API in the dashboard. An API is an entity that represents an external resource, capable of accepting and responding to requests made by clients, such as the authors API we just made.

> **Auth0 offers a generous free tier to get started with modern authentication.**

Login to your Auth0 management dashboard and create a new API client.

Click on the APIs menu item and then the **Create API** button.

Create a New API



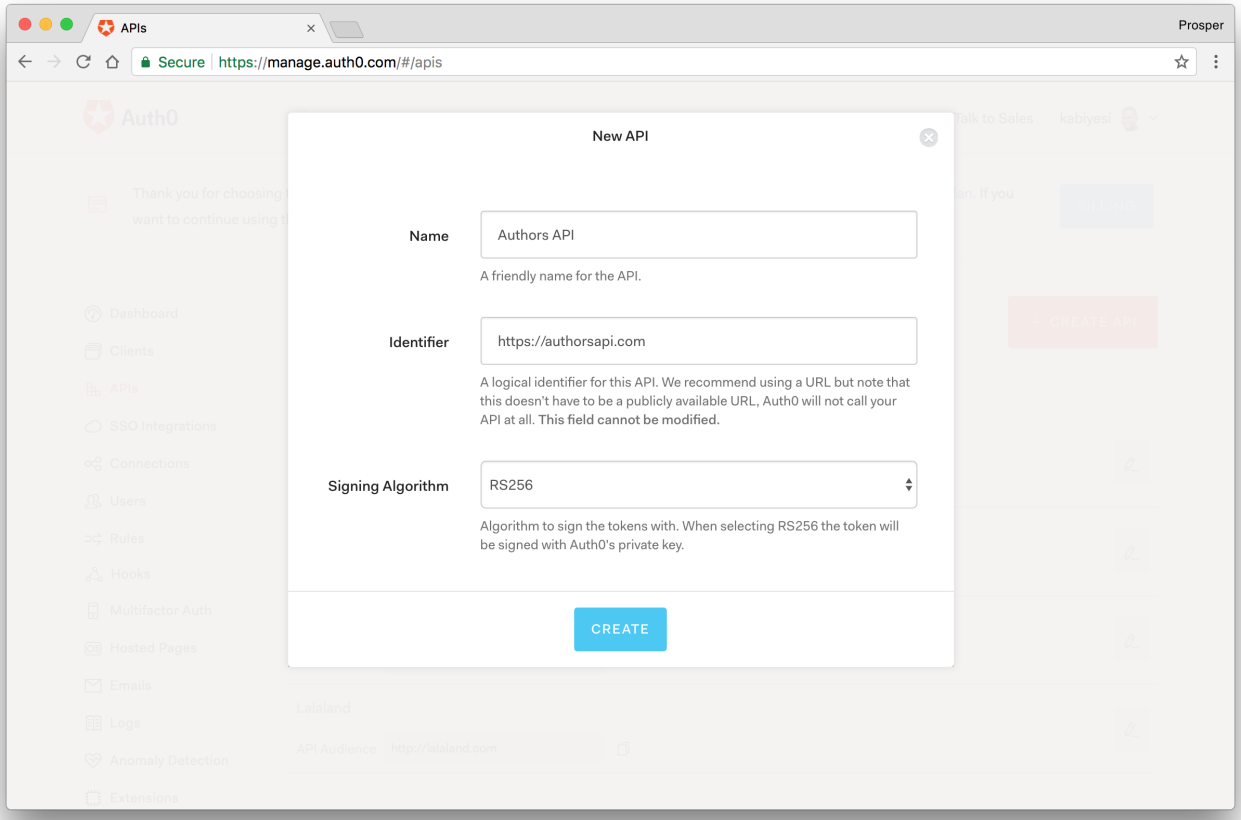
Next you will need to give your API a `Name` and an `Identifier`. The `Name` can be anything you choose, so make it as descriptive as you want. The `Identifier` will be used to specify your API and cannot be changed once set. We'll be using it as an `audience` later when configuring the access token verification.

Here's my setup for the author's API:

Name: **Authors API** Identifier: **https://authorsapi.com**. Signing algorithm: **RS256**

Once you have yours filled out, click on the **Create API** button.

Creating the Authors API



Next head over to your terminal and install the **Auth0 PHP SDK** in your project's root directory:

```
composer require auth0/auth0-php
```

Integrate Auth0 into your API

Create a new middleware file, `Auth0Middleware.php` in the `app/Http/Middleware` directory. Add the following code to it:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Auth0\SDK\JWTVerifier;

class Auth0Middleware
```

```
{
    /**
     * Run the request filter.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        $token = $request->bearerToken();
        if(!$token) {
        return response()->json('No token provided', 401);
        }

        $this->validateAndDecode($token);

        return $next($request);
    }

    public function validateAndDecode($token)
    {
        try {
            $verifier = new JWTVerifier([
              'supported_algs' => ['RS256'],
              'valid_audiences' => ['AUTH0_API_AUDIENCE'],
              'authorized_iss' => ['AUTH0_ISS']
            ]);

            $decoded = $verifier->verifyAndDecode($token);
        }
        catch(\Auth0\SDK\Exception\CoreException $e) {
            throw $e;
        };
```

```
        }

    }
```

In the `validateAndDecode` method, we created an instance of `JWTVerifier` to verify the token coming from the Authorization header. It checks the algorithm, the API audience, the issuer, expiration time, issued time, and signature to ensure the token is a valid one issued by Auth0.

Next we need to replace the `AUTH0_API_AUDIENCE` and `AUTH0_ISS` placeholders with the API audience and Auth0 domain values from your Auth0 dashboard.

AUTH0_API_AUDIENCE

You can find this value in the Auth0 dashboard by clicking:

`APIs` -> Select the API you just created -> `Settings`.

Copy the value listed for `Identifier` you just created and replace `AUTH0_API_AUDIENCE` with that value.

AUTH0_ISS

If you look under "Applications" you should see "Domain". Copy this and add `https://` to it to create the value for `AUTH0_ISS`, e.g. `https://xyz.auth0.com/`.

Important note:

When filling in the `AUTH0_ISS` value, make sure it includes the trailing slash: `https://xyz.auth0.com/`.

Now, we want to assign the newly created middleware to our routes. The first step is to assign the middleware a short-hand key in `bootstrap/app.php` file's call to the `$app->routeMiddleware()` method.

Go ahead and open up `bootstrap/app.php` and uncomment this line of code:

```
...
// $app->routeMiddleware([
//     'auth' => App\Http\Middleware\Authenticate::class,
// ]);
...
```

Once uncommented, replace the `Authenticate::class` with `Auth0Middleware::class` like so:

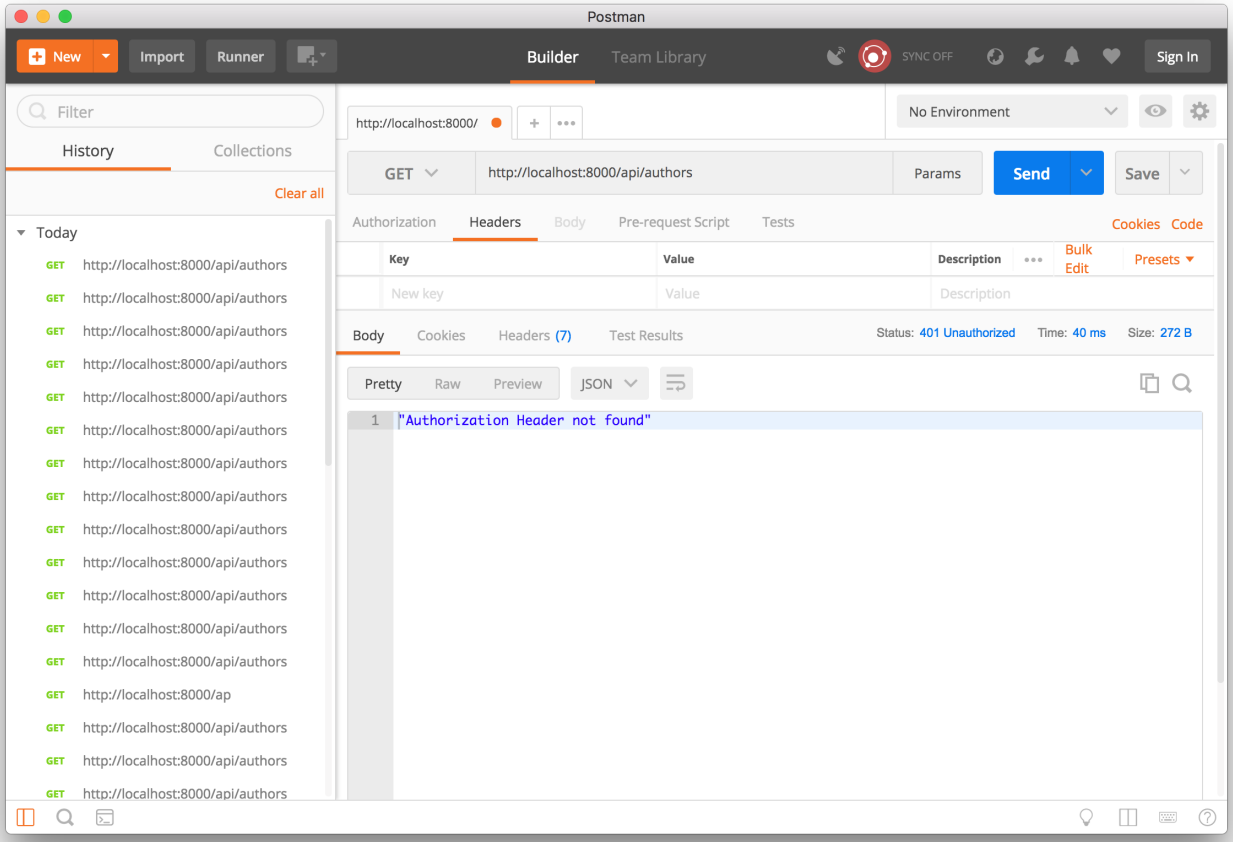
```
$app->routeMiddleware([
    'auth' => App\Http\Middleware\Auth0Middleware::class,
]);
```

We can now use the middleware key in the route options array in the `routes/web.php` file like so:

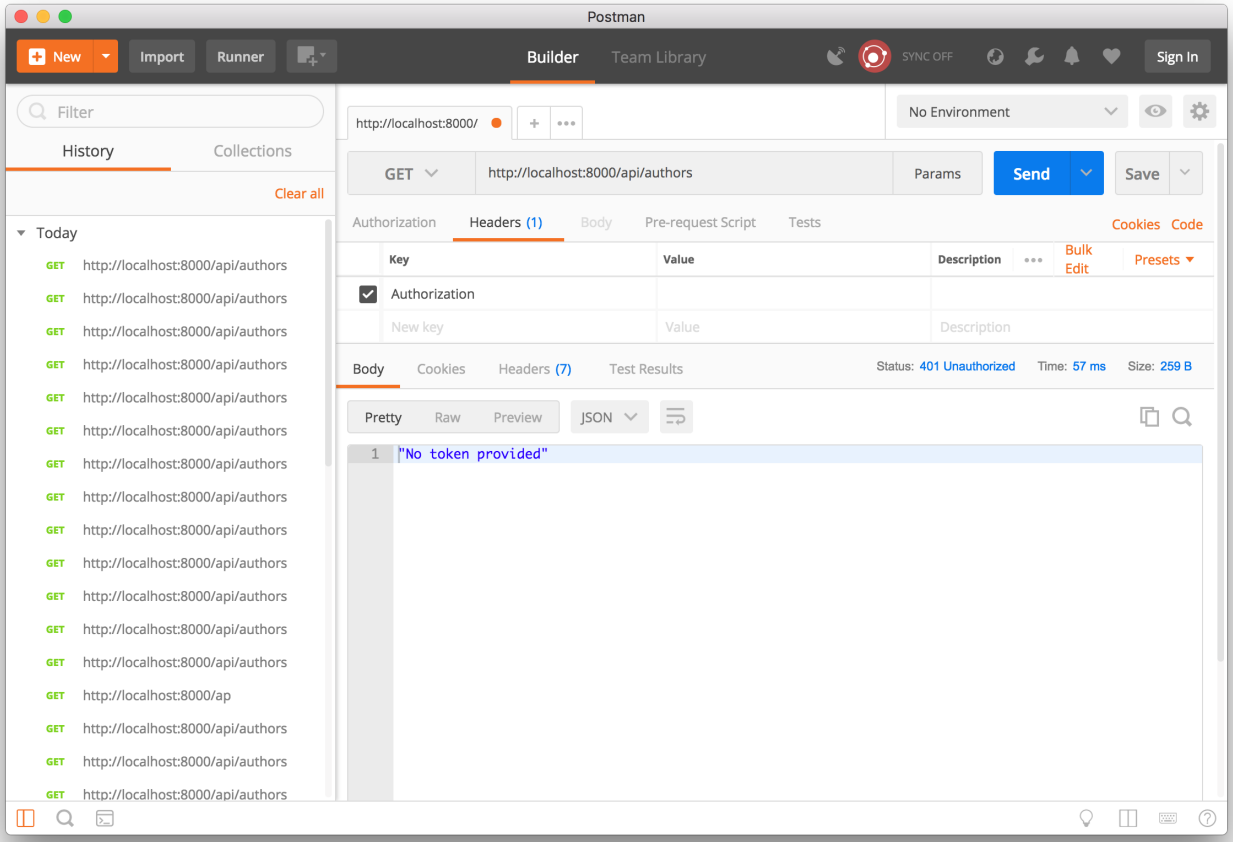
```
...
$router->group(['prefix' => 'api', 'middleware' => 'auth'], function
  $router->get('authors',  ['uses' => 'AuthorController@showAllAuthor

  $router->get('authors/{id}', ['uses' => 'AuthorController@showOneAu

  $router->post('authors', ['uses' => 'AuthorController@create']);

  $router->delete('authors/{id}', ['uses' => 'AuthorController@delete

  $router->put('authors/{id}', ['uses' => 'AuthorController@update'])
});
```

We just secured all the API endpoints with access tokens. If an application makes a request to these API endpoints without a valid access token or no token at all, it returns an error. Let's try it out.

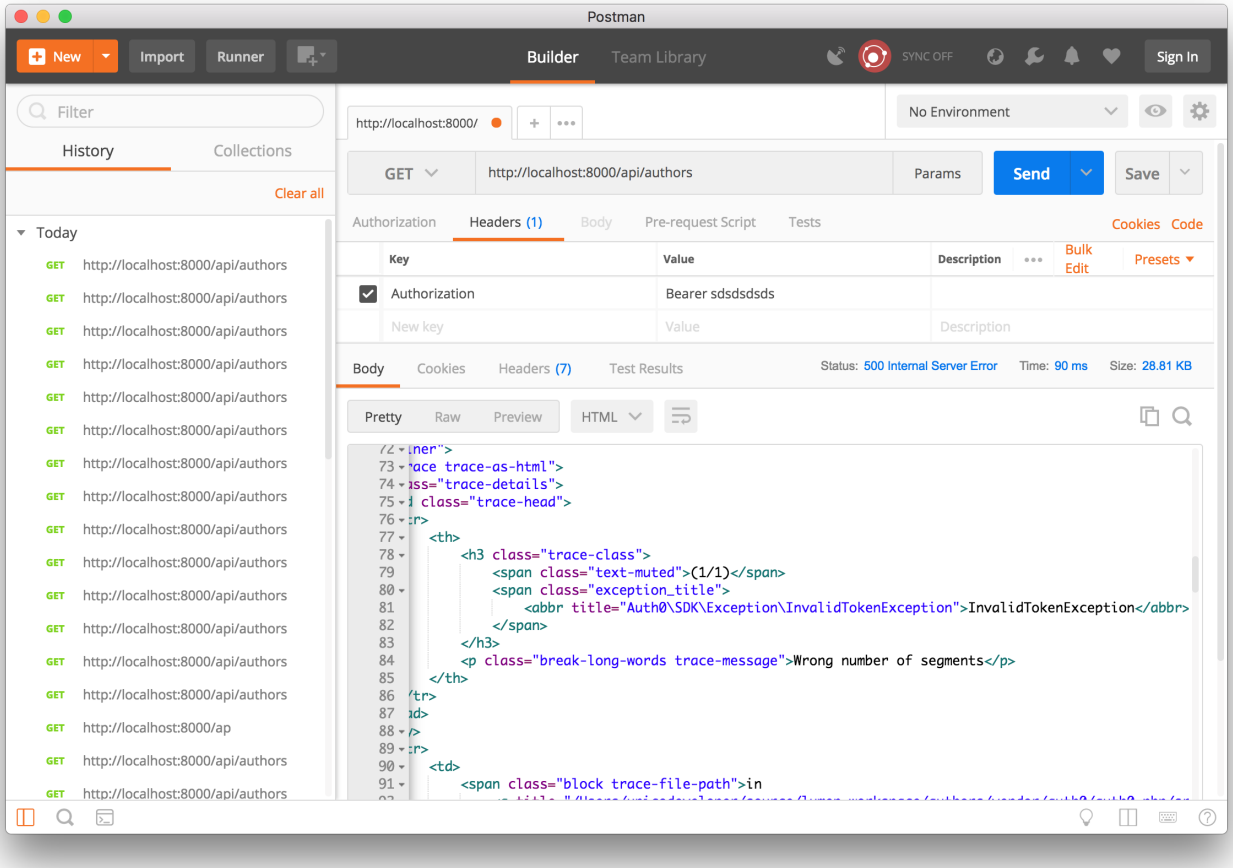
Accessing any endpoint without an authorization header



Accessing any endpoint without any token provided



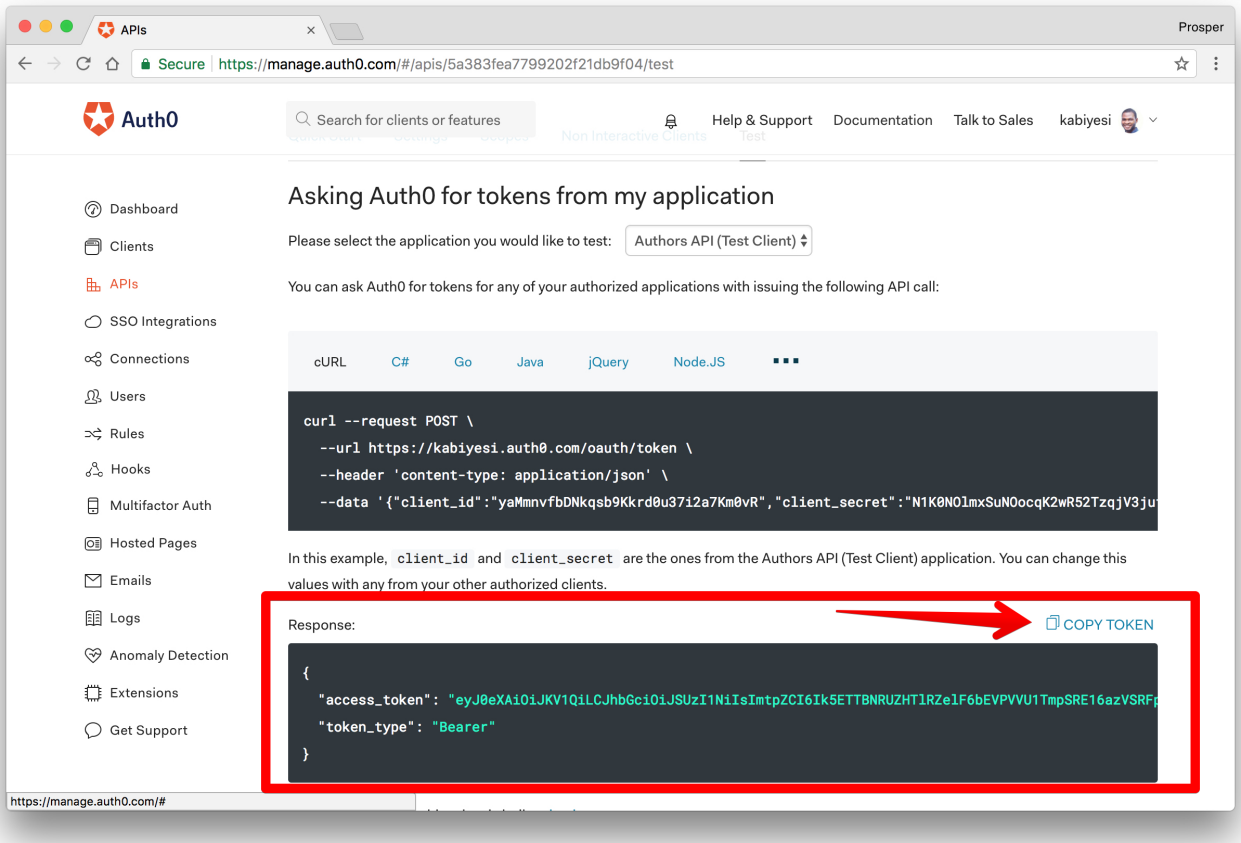
Accessing any endpoint without a valid access token



Now, let's test it with a valid access token. Head over to the `test` tab of your newly created API on your Auth0 dashboard.

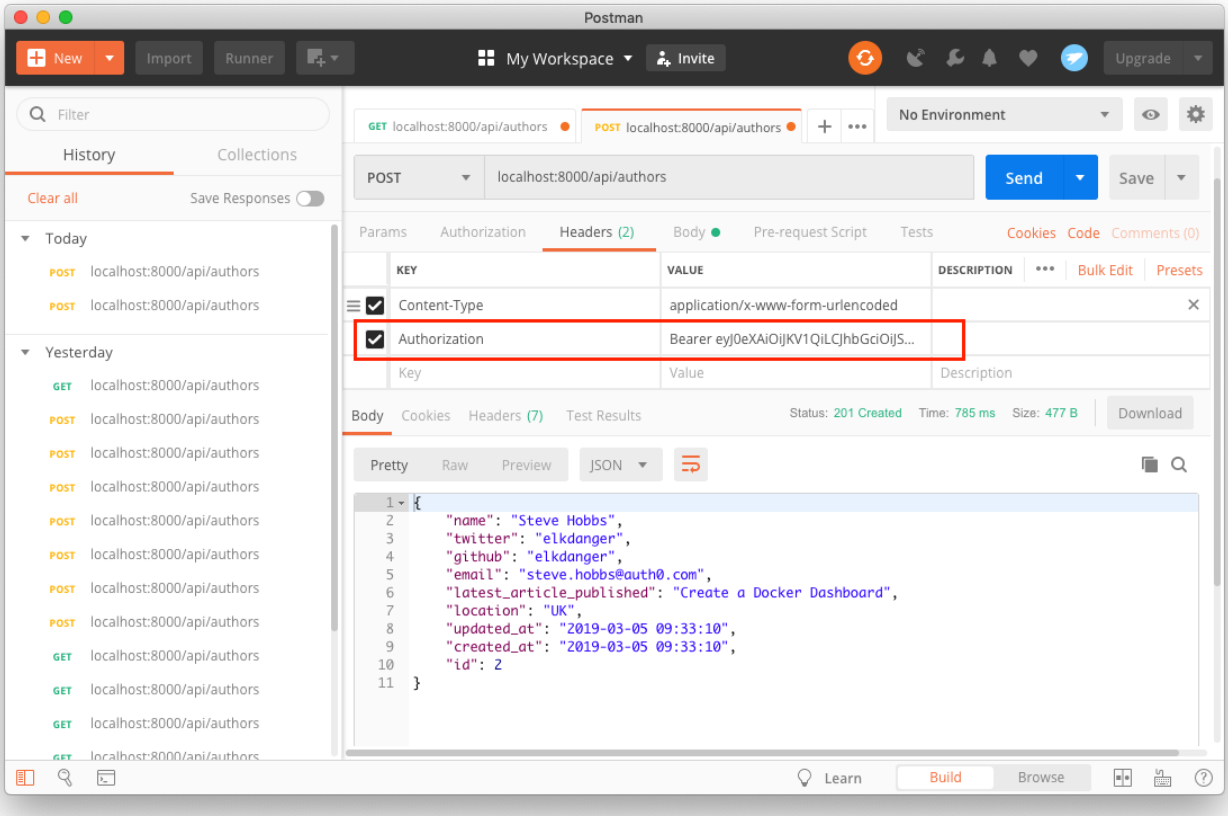
Grab the Access token from the *Test* tab

Grab the Access Token



Now use this `access token` in Postman by sending it as an Authorization header to make a POST request to `api/authors` endpoint.

Accessing the endpoint securely

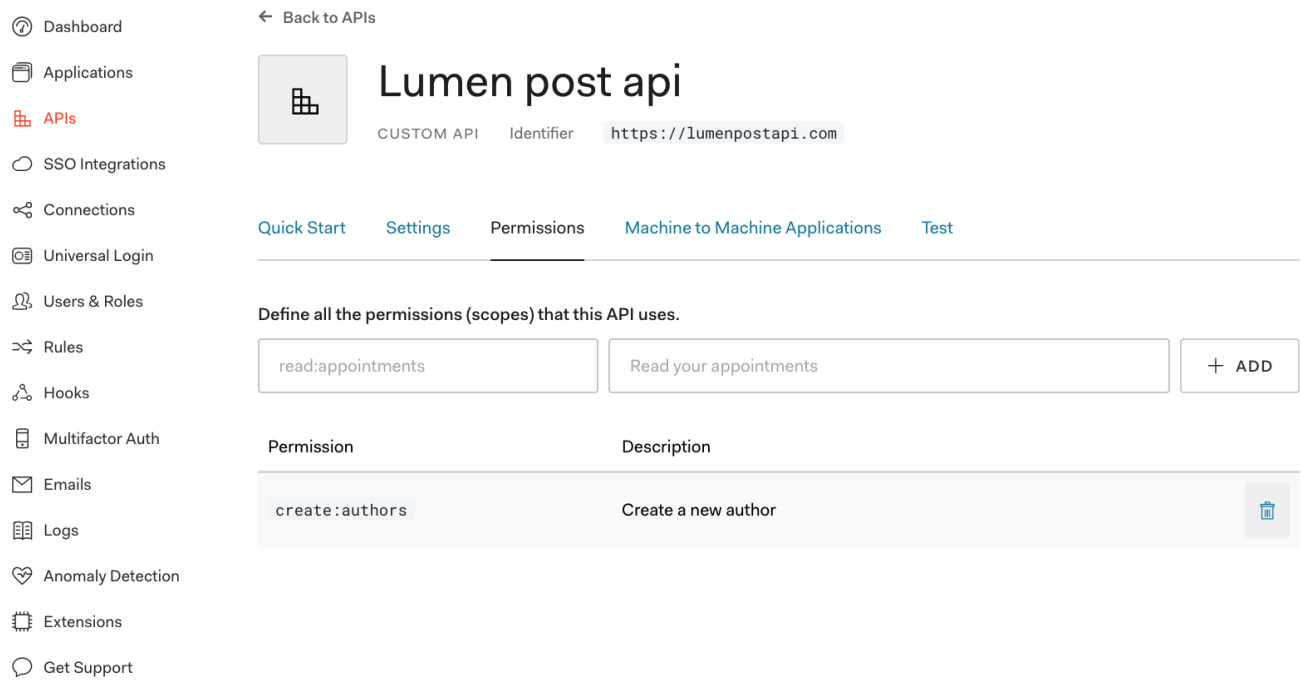


It validates the access token and successfully makes the POST request.

If you're getting a message that the token cannot be trusted, try adding a trailing slash to your `authorized_iss` domain in `app/Http/Middleware/Auth0Middleware.php` e.g. `https://xyz.auth0.com/`

Permissions

Currently, this single access token will allow an application to run any requests, as long as it has a valid token. You may want to eventually issue certain **permissions** with the access token. This can be done in the Auth0 dashboard under `API > Permissions`. Let's try it out. Create a new scope that will grant permission to create a new author (e.g. `create:authors`). Then add a short description of what that scope does and press "Add".



Now our API expects that when an application makes a request to create a new author, it must also send an access token that includes the `create:authors` scope. To check for this, we need to add middleware that checks the scope in the access token. Open up `Auth0Middleware.php` that was created earlier and replace it with this:

```
// app/Http/Middleware/Auth0Middleware.php
<?php

namespace App\Http\Middleware;

use Closure;
use Auth0\SDK\JWTVerifier;

class Auth0Middleware
{
    /**
     * Run the request filter.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
```

```
     * @return mixed
     */
    public function handle($request, Closure $next, $scopeRequired =
    {
        $token = $request->bearerToken();

        if(!$token) {
        return response()->json('No token provided', 401);
        }

        $this->validateAndDecode($token);
        $decodedToken = $this->validateAndDecode($token);

        if ($scopeRequired && !$this->tokenHasScope($decodedToken, $s
            return response()->json(['message' => 'Insufficient scope
        }
        return $next($request);
    }

    public function validateAndDecode($token)
    {
        try {
            $verifier = new JWTVerifier([
              'supported_algs' => ['RS256'],
              'valid_audiences' => ['https://authorsapi.com'],
              'authorized_iss' => ['https://demo-apps.auth0.com/']
            ]);

           return $verifier->verifyAndDecode($token);
        }
        catch(\Auth0\SDK\Exception\CoreException $e) {
            throw $e;
        };
    }
```

```
    /**
     * Check if a token has a specific scope.
     *
     * @param \stdClass $token - JWT access token to check.
     * @param string $scopeRequired - Scope to check for.
     *
     * @return bool
     */
    protected function tokenHasScope($token, $scopeRequired) {
      if (empty($token->scope)) {
          return false;
      }

      $tokenScopes = explode(' ', $token->scope);
      return in_array($scopeRequired, $tokenScopes);
    }

}
```

A couple changes were made here.

The first thing to note is we added another parameter, `scopeRequired`, which is set to `null` by default. Later in our routes, we'll specify where it is required.

```
// ...
public function handle($request, Closure $next, $scopeRequired = null
    // ...
}
// ...
```

Then we get the validated and **decoded** token (done in `validateAndDecode()`):

```
// ...
$this->validateAndDecode($token);
$decodedToken = $this->validateAndDecode($token);
// ...
```

Next, we check if the scope is required for this request. If it is, we use that decoded token to check if the scope exists. If so, the request continues, but if not, we send an `Insufficient scope` message instead.

```
if ($scopeRequired && !$this->tokenHasScope($decodedToken, $scopeRequ
    return response()->json(['message' => 'Insufficient scope'], 403)
}
```

We also added the method `tokenHasScope()`, which is what's doing the check for that specific scope in the previous `if` statement.

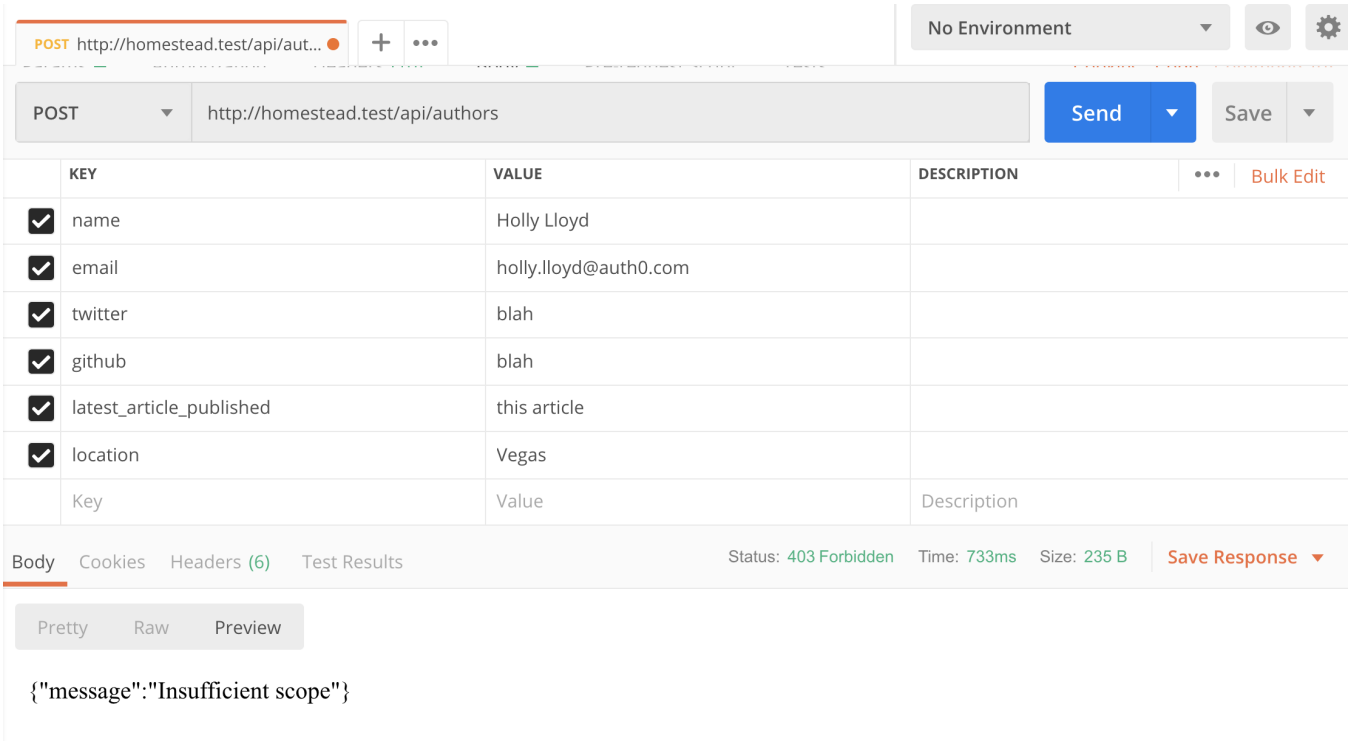
```
protected function tokenHasScope($token, $scopeRequired) {
    if (empty($token->scope)) {
        return false;
    }

    $tokenScopes = explode(' ', $token->scope);
    return in_array($scopeRequired, $tokenScopes);
}
```

Finally we need to add this scope check middleware to our route for creating a new author.

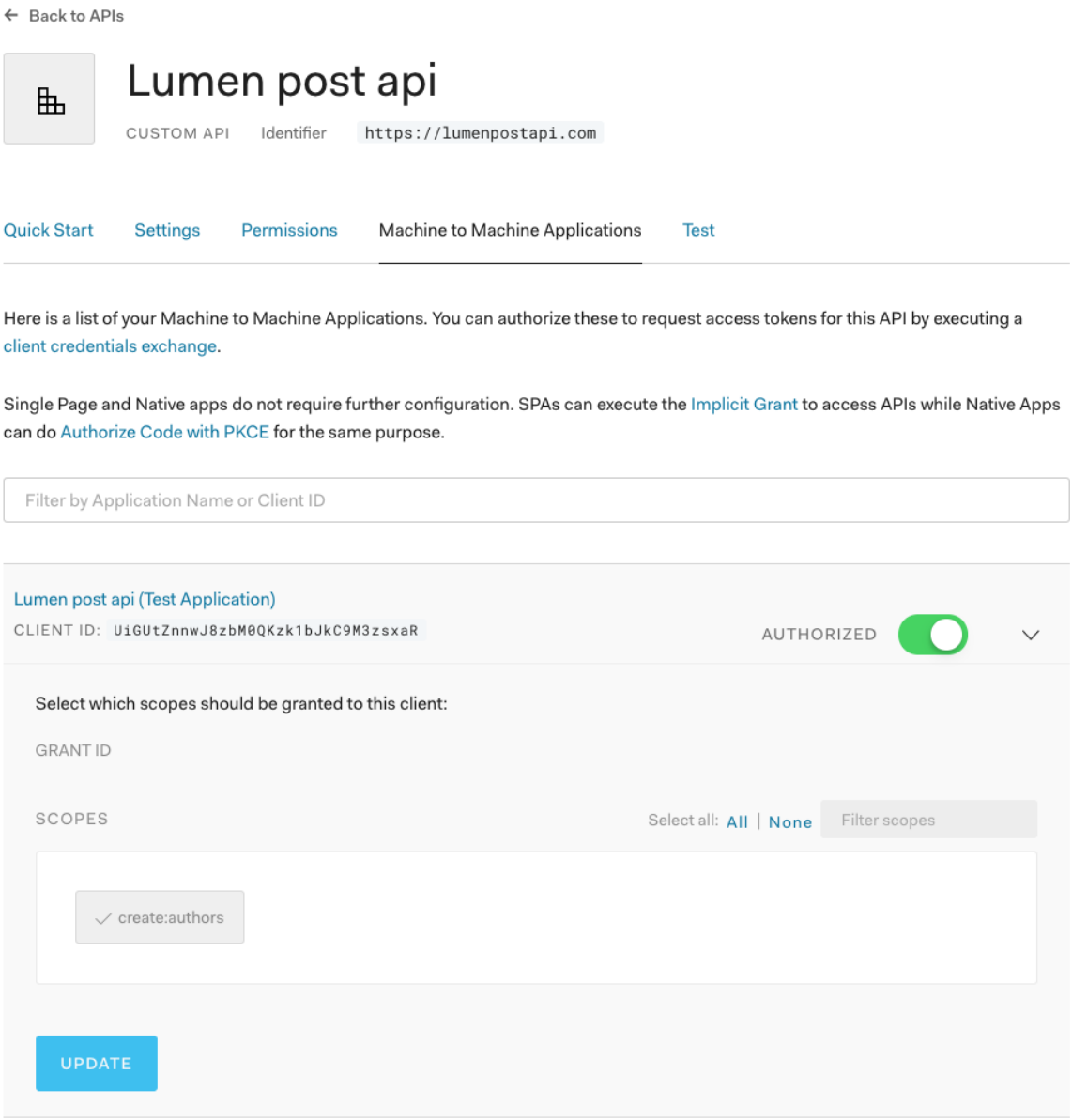
```
// routes/web.php
// ...
$router->group(['prefix' => 'api', 'middleware' => 'auth'], function
```

```
    // ...

    $router->post('authors', ['middleware' => 'auth:create:authors',

    // ...
});
```

Now if you try to create a new author in Postman using that same token as before, you'll receive the "Insufficient scope" message.



To test that it works, go to the "Permissions" tab in the Auth0 dashboard and click on "Machine to Machine Applications". Find the API Application you've been using, make sure the "Authorized" toggle is on, then click on the arrow. Now select the `create:authors` scope and press "Update".



Now the permission for `create:authors` has been added to our test token. Head back over to the "Test" tab and press "Copy token" to get the updated one.

Paste that token into the Authorization header as you did before (make sure you have `Bearer` before it), try the POST request again, and now it should have worked!

If you'd like to see what the decoded access token looks like, just add `dd($decodedToken);` inside the `handle()` method in `app/Http/Middleware/Auth0Middleware.php` right after the `$decodedToken` variable is declared. Then just run that POST request one more time in Postman and you'll see the contents of the token, including the scope. Pretty cool! Just make sure you delete that test line in a real application.

Adding a front-end

This is just an example of how to create the API access tokens. Once you're ready to actually **issue and use them**, you need to create a front-end. Here are some amazing React and Vue.js authentication tutorials that cover how you can accomplish that.

Conclusion

Well done! You have learned how to build a rest API with the powerful PHP micro-framework Lumen and secure it using JWTs. Need to use PHP to build your API or micro-service? I'd bet on Lumen as the tool of choice for speed and ease of use.

As you've seen, Auth0 can help secure your **API** with ease. Auth0 provides more than just username-password authentication. It provides features like multifactor auth, breached password detection, anomaly detection, enterprise federation, single sign on (SSO), and more.

Sign up today so you can take the stress out of authentication and instead focus on building unique features for your app.

Please, let me know if you have any questions or observations in the comment section. 😊

Prosper Otemuyiwa

Prosper is a great speaker, community leader, open source hacker, technical consultant, and a fervent Developer Advocate. He is a full-stack software engineer who has worked on

biometric, health and developer tools. Prosper recently co-founded a startup called Eden

that's focused on improving the quality of lives in Nigeria and currently leads the engineering team. He also co-founded forloop, the largest developer community in Africa.

VIEW PROFILE ▸

More like this

LARAVEL

Build a Laravel 6 CRUD App with Authentication

LOOPBACK

Developing RESTful APIs with Loopback

SPRING BOOT

Implementing JWT Authentication on Spring Boot APIs

Follow the conversation

Comments Community 1 Login

Recommend 7 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Adagio • 10 months ago

How does one now access the Userdata of whom made the API-Request i.e. I want to keep track of who changed data via PUT.

1 • Reply • Share ›

Bruno S. Krebs Mod Adagio • 10 months ago

Well, for that you will need to create an Auth0 Application and let users sign in. Then, you can issue a request to the /userinfo endpoint with the access token retrieved while authenticating to see who made the request.

Or, better yet, if your API is part of your web application (or SPA), you can use id tokens instead of access tokens.

• Reply • Share ›

pararang • 2 years ago

nice explanation, thank you

1 • Reply • Share ›

Francis Sunday • 2 years ago

Hey Prosper, nice guide on Lumen, this is super helpful

1 • Reply • Share ›

Sachith Dassanayake • 20 days ago • edited

Does Active Directory/ LDAP authentication work the same way? So when a user is authenticated does it return a JSON web token? And then use the JWT in subsequent requests. When does the session expires?

• Reply • Share ›

Dan Arias Mod ➚ Sachith Dassanayake • 13 days ago

Howdy! Do you mean Active Directory with Auth0 or on its own? Thanks.

^ | v • Reply • Share ›

Sachith Dassanayake ➚ Dan Arias • 6 days ago

Hi Dan, I mean Auth0 after the username and password are validated against the AD credentials as AD won't return a JWT. Will Auth0 provide the AD group the user belongs to etc..? Thanks

^ | v • Reply • Share ›

Manuel Silva • 2 months ago

code ?

^ | v • Reply • Share ›

Manuel Silva • 2 months ago

Best Regards From Mexico City. Manuel Silva

^ | v • Reply • Share ›

Tomáš Teicher • 4 months ago

Great tutorial :)

can anybody recommend a good IDE, that can navigate to Lumen's method declarations?
For example, there is method "Author::create($request->all())" call in this tutorial. I am using Netbeans and I am not able to go to declaration of "create" method. So I cannot learn from code, how is this method defined. Can anybody help?

^ | v • Reply • Share ›

Bruno S. Krebs Mod ➚ Tomáš Teicher • 4 months ago

I really like JetBrains solutions, but they cost some money. In this case, https://www.jetbrains.com/p... would be their solution.

^ | v • Reply • Share ›

Tomáš Teicher • 4 months ago

Hi, could anybody help to explain, how the connection was made, between Author class and "authors" schema? How the Author class knows, that it should work with "authors" database table?
Many thanks :)

^ | v • Reply • Share ›

Bruno S. Krebs Mod ➚ Tomáš Teicher • 4 months ago

This is probably based on some Eloquent naming conventions: http://www.rappasoft.com/ar

conventions: http://www.rappasoft.com/ar...

1 ^ | v • Reply • Share ›

Andrew Beak • 6 months ago

What if there is more than one person using your API? There is no concept of identity in this tutorial. How do you determine the "sub" of the cookie and use that in your controller?

^ | v • Reply • Share ›

pinoyCoder • 7 months ago • edited

Hi,
Thanks for this, but on client side how can you secure the tokens,example on my client side they will use an ajax request and i notice that using the google chrome developer tools i can see the event which the request that occurred and event the token in the header, so the information can still be steal and use buy someone to abuse the my application. Can you please advise how can i prevent or mitigate that ?

^ | v • Reply • Share ›

Bruno S. Krebs Mod ➚ pinoyCoder • 6 months ago

By abuse your application you mean by injecting some malicious JavaScript in _your_ application to intercept the AJAX request? If that is the case, then I don't think you have much to do in relation to the tokens itself. You would have to be more careful about your app as a whole because having a weak app like that will cause you a lot of other problems (besides losing a token).

Anyway, I just thought a little bit more about it and you could use HTTP-only cookies to help you there.

^ | v • Reply • Share ›

Monali Patel • 7 months ago

Hi Is it Micro service API?

^ | v • Reply • Share ›

Bruno S. Krebs Mod ➚ Monali Patel • 6 months ago

Sorry, what?

^ | v • Reply • Share ›

Marcus Zamora • 7 months ago

I can't get the result on the lsat part. :(Why dude? I get the result on all, but why this last image I cant get it. Is it really "http://localhost:8000/api/people/" not "http://localhost:8000/api/author/"?

^ | v • Reply • Share ›

Steve Hobbs ↗ Marcus Zamora • 7 months ago

Hey Marcus,

I think you're right, it should be `http://localhost:8000/api/authors` - does that work for you?

In the meantime I'll see about getting this part fixed - thanks for lettting us know about the problems you're having!

Reply • Share ›

Mark Louise Fernandez • 8 months ago

help me please

Reply • Share ›

Sivaramadurai nadar ↗ Mark Louise Fernandez • 7 months ago

Hi, Instead of PUT, use GET to fetch the authors.

Reply • Share ›

Bruno S. Krebs Mod ↗ Mark Louise Fernandez • 8 months ago

Hi, have you compared your project with the one built by the author? https://github.com/auth0-bl...

Reply • Share ›

muzanaka • 8 months ago

Мужики, спасибо! Еще бы показали как swagger'ор все описать, и было бы вообще готовая инструкция к действию.

Reply • Share ›

Regita Drajat • 10 months ago

i cant get any reponse
i try to hit http://localhost:8000/api/authors but its return
"Lumen (5.7.6) (Laravel Components 5.7.*)"
->here model and my reponse from postman

see more

Reply • Share ›

Ali Ahmed • a year ago

I don't know why Lumen hasn't been able to find my controller classes lately, I had to pass a namespace argument to the group with App\Http\Controllers as value to make it work properly.

Reply • Share ›

Aishwarya Zipare • a year ago

getting error undefined variable:router

Reply • Share ›

Bruno S. Krebs Mod ➚ Aishwarya Zipare • a year ago

Hey there, perhaps this might help you?

^ | v • Reply • Share ›

Luis Carlos Silva Magallanes • a year ago

Hey prosper. Greate tutorial but I have a problem.
When I try to make a POST (create) I have this message from de Postman: SQLSTATE[42S02]: Base table or view not found: 1146 Table 'authors.users' doesn't exist (SQL: select count(*) as aggregate from `users` where `email` = jp@mail.com)
Any idea about this? Thanks.

^ | v • Reply • Share ›

b1kjsh ➚ Luis Carlos Silva Magallanes • a year ago

Just change the following validation from
'email' => 'required|email|unique:users'
To:
'email' => 'required|email|unique:authors'

Or you can create a table for users using the migrate commands in the earlier part of the tutorial. However, using authors instead of users makes more sense in this tutorial. They should probably just change that in the tutorial.

^ | v • Reply • Share ›

Luk Berezowski ➚ Luis Carlos Silva Magallanes • a year ago • edited

This is due the validation rule set for the `email` field (in `create` method inside `AuthorController`) which states:
'email' => 'required|email|unique:users',

Just remove `unique:users` so it reads:
'email' => 'required|email',

You have most likely sorted that issue already, since it's from 2 month back, but maybe it will help someone else.

Cheers

^ | v • Reply • Share ›

Bruno S. Krebs Mod ➚ Luk Berezowski • a year ago

Thanks for contributing.

^ | v • Reply • Share ›

Michael Holland • a year ago

Really useful quick-start guide to Lumen as well as Auth0. However, I think it might be more 'real-world' if you described how to selectively apply the authentication. For example,

anyone could list authors but only authenticated users could add or edit.

Reply • Share ›

Bruno Krebs ➚ Michael Holland • a year ago

Thanks for the feedback. I will take a note and we will try to improve it in our next articles.

Reply • Share ›

Somcutean Doru • a year ago

Great tutorial! Thanks for explaining this so nicely and in an easy way.

Reply • Share ›

NandoBas • a year ago

tanks, its work!
now i am integrate with React.

Reply • Share ›

Dwi Yudi Rayi Anugrah • a year ago

Error message "We can't trust on a token issued by" why ?

Reply • Share ›

Juan Manuel Heredia Gallardo • 2 years ago

hey prosper, thanks for the guide, i followed it but i got this error "cURL error 60: SSL certificate problem: self signed certificate in certificate chain", i update php.ini with this "curl.cainfo = "C:\wamp64\bin\php\php7.0.10\extras\ssl\cacert.pem" but nothing change ... any idea ??

Reply • Share ›

Prosper ➚ Juan Manuel Heredia Gallardo • a year ago

Hello **@Juan Manuel Heredia Gallardo** please update the cURL on your server to the latest version. Do you also have SSL activated on your server?

Reply • Share ›

Juan Manuel Heredia Gallardo ➚ Prosper • a year ago

finally i got it working ...

Reply • Share ›

kunamistu • 2 years ago

Managed to configure everything... but how do I access the User information from my backend after it is authorized?

Reply • Share ›

Rohani Suhadi • 2 years ago

• Reply • Share ›

Sinan ➚ Rohani Suhadi • 2 years ago

Hi **@Rohani Suhadi**, I had the same error before and I solved this by adding a trailing slash in authorized_iss on Auth0Middleware. See the image:

• Reply • Share ›

TRY AUTH0 FOR FREE

TALK TO SALES

BLOG

Developers
Identity & Security
Business
Culture
Engineering
Announcements

COMPANY

About Us
Customers
Security
Careers
Partners
Press

PRODUCT

Single Sign-On
Password Detection
Guardian
M2M
Universal Login
Passwordless

MORE

Auth0.com
Ambassador Program
Guest Author Program
Auth0 Community
Resources