МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Мегафакультет трансляционных информационных технологий

Факультет информационных технологий и программирования

**Лабораторная работа №6**

**По дисциплине «Web-программирование»**

**Добавление авторизации и пользовательских сессий**

**Выполнили студент группы М33081**

**Аль Даббагх Харит Хуссейн**

**Проверил**

**Приискалов Роман Андреевич**

САНКТ-ПЕТЕРБУРГ

2022

Содержание

[Authentication and Authorization 2](#_Toc101784025)

[Registering on supertokens.com 2](#_Toc101784026)

[The NestJS Backend 3](#_Toc101784027)

[Logging in 10](#_Toc101784028)

[Testing the API 12](#_Toc101784029)

[Modifying the frontend 15](#_Toc101784030)

# Authentication and Authorization

During this lab work we are required to add authentication and authorization.

***Authentication*** is the process of signing in and obtaining certain roles (Verifying who the user is).

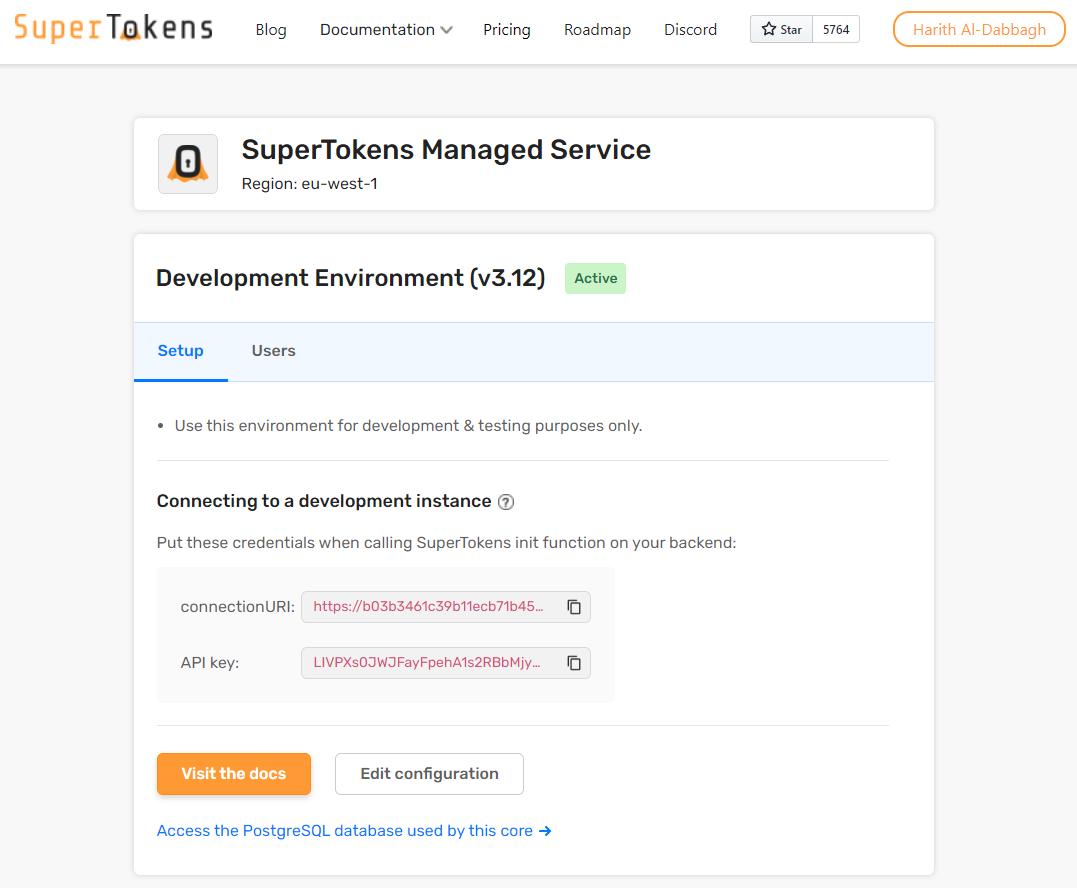
***Authorization*** is the process of verifying what specific resources that user has access to.

In the context of the lab work we are required to secure the endpoints of our application, letting only authorized users access certain endpoints.

To do that I’m going to use super tokens, and the following is a step-by-step explanation of how I implemented it.

## Registering on supertokens.com

The process is very simple and requires no explanation. In the end of it we will receive two important values, the connectionURI and the API Key:

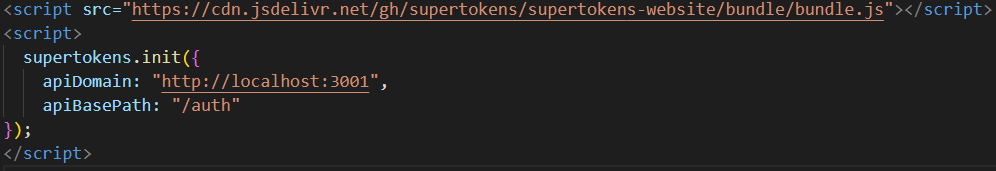


We’ll add these to our environment variables for security.

The ones for the production environment we can just go ahead and add them to heroku environment variables.

## The NestJS Backend

Before implementing the backend, in our HTML we’re going to add the frontend integration of super tokens. I’ve added this to the common partial which is included in every page:



To integrate SuperTokens into a NestJS backend we will add a few things:

* A module to house all authorization related code
* A service to initialize the SDK
* A middleware to add the authorization endpoints
* A global error handler to pass SuperTokens related errors to the SDK
* A guard to protect the API endpoints
* A parameter decorator to access the session in the code

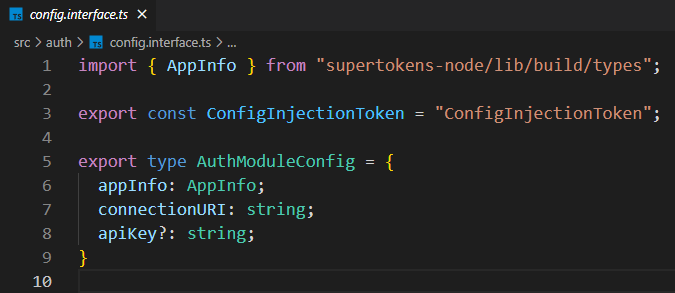
Next we install SuperTokens using the command:

npm i -s supertokens-node

Up next is the creation of a module for authentication and authorization, we’re just going to call it auth:

nest g module auth

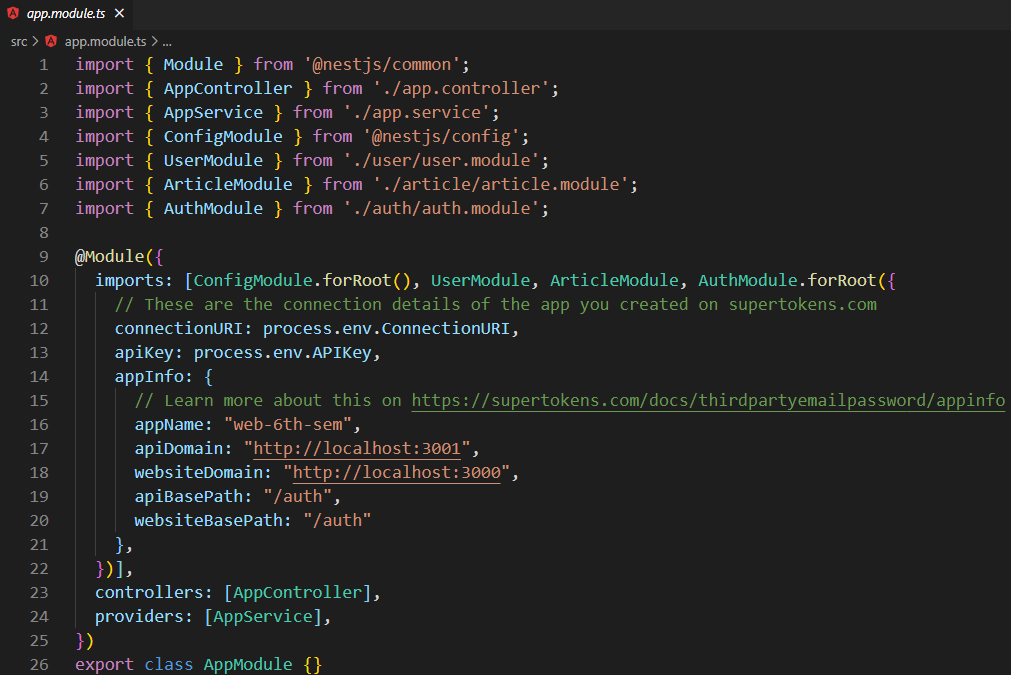
Next we add a configuration interface in the auth folder that was created for the module, the contents of which:



Next up we finish the implementation of the auth module, which will look something like this:



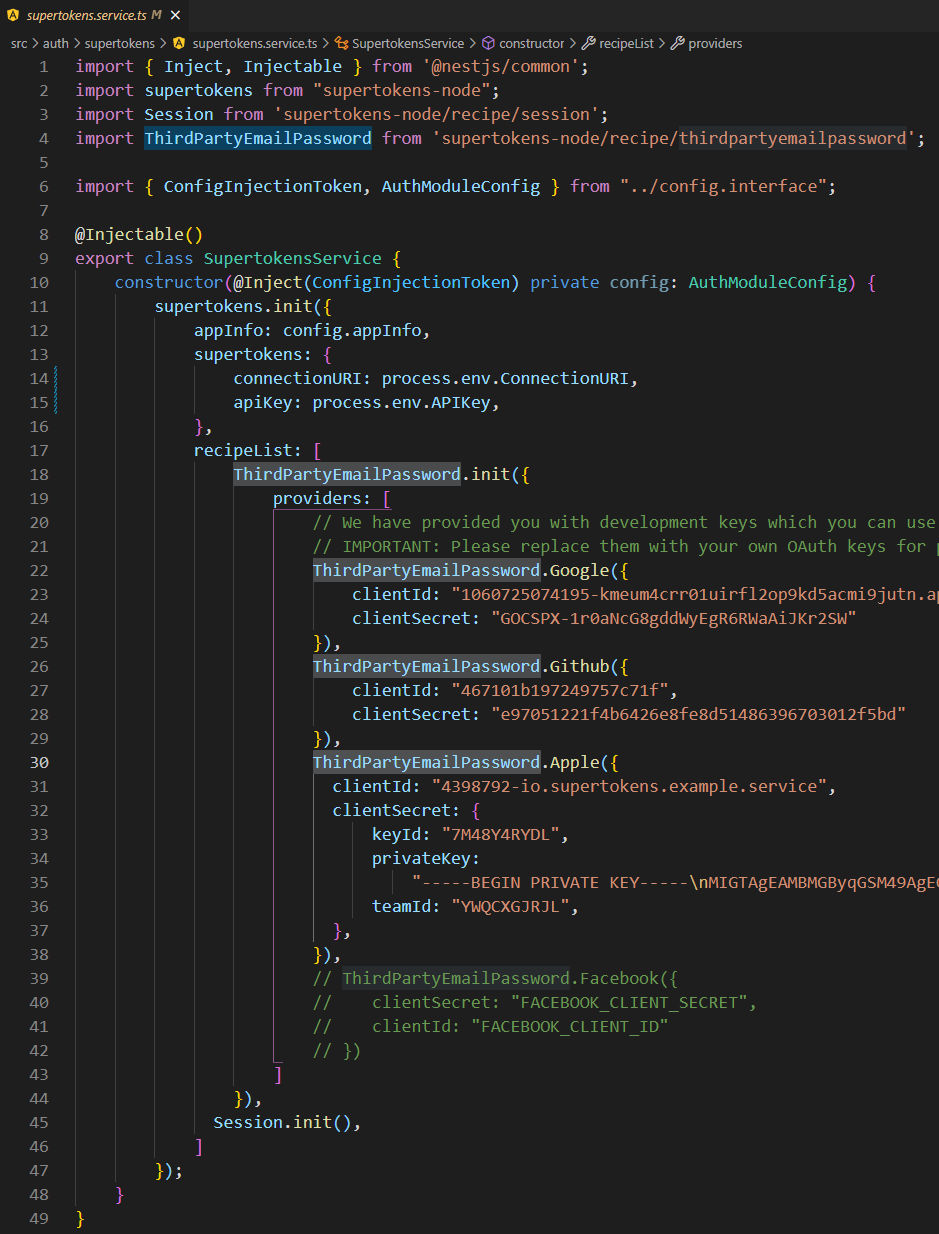
Next we will update the app module to use the newly created dynamic module by importing the result of forRoot instead of the class itself:



Let’s now create the service using the command:

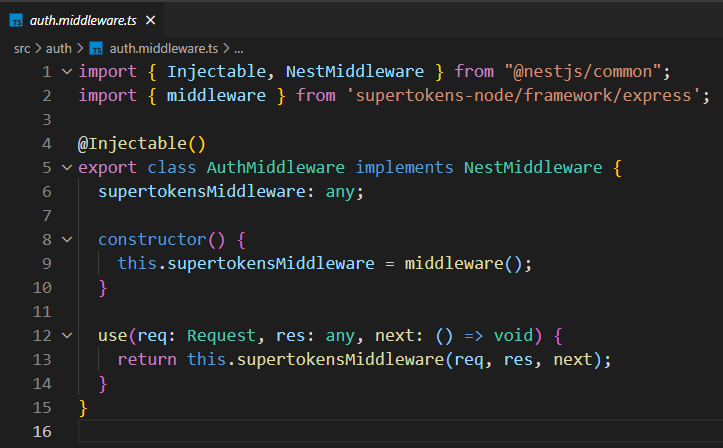
nest g service supertokens auth

This will create a service separated in it’s own folder inside the auth folder:

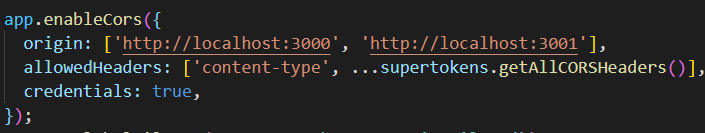


Now we need to create a middleware in order to expose the SuperTokens API. We’ll start by the command:

nest g middleware auth auth

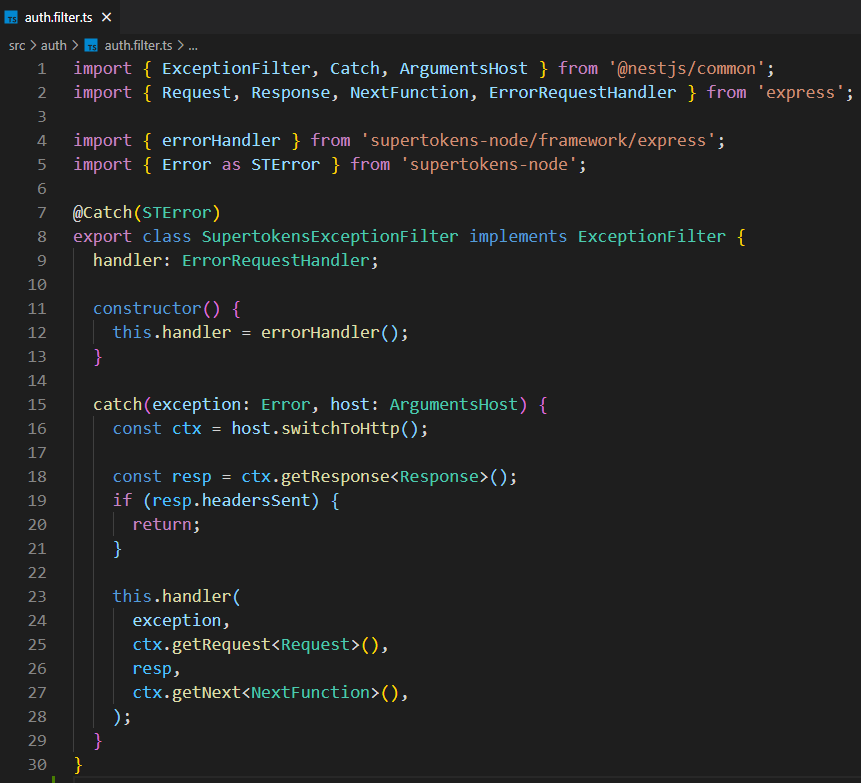


Next up is updating the CORS policy settings. In the main.ts file we need to add the following:

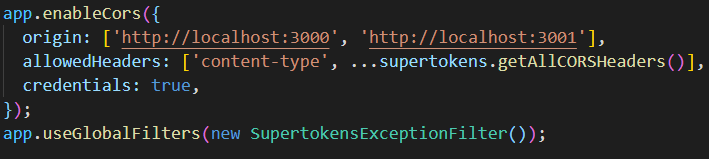


For convenience we’re going to also add and exception filter, using the command:

nest g filter auth auth

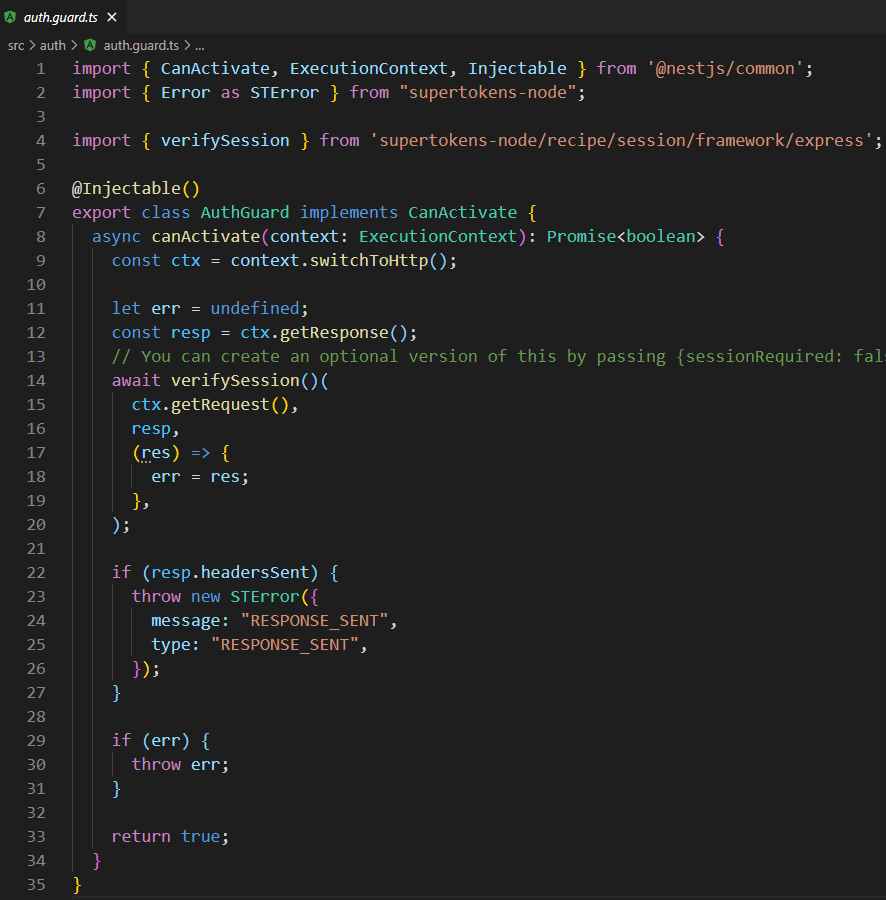


We also need to register the filter by adding a line to main.ts after the CORS policy settings:



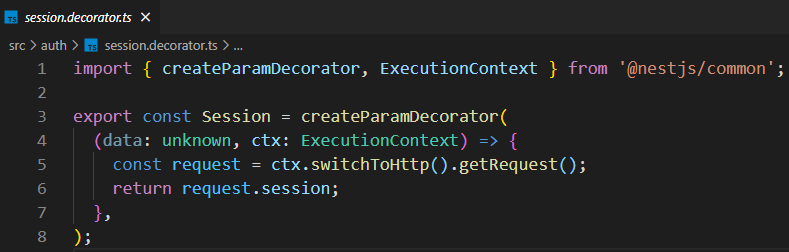
Next up we create a verification guard, which will protect our end points, we use the command:

nest g guard auth auth

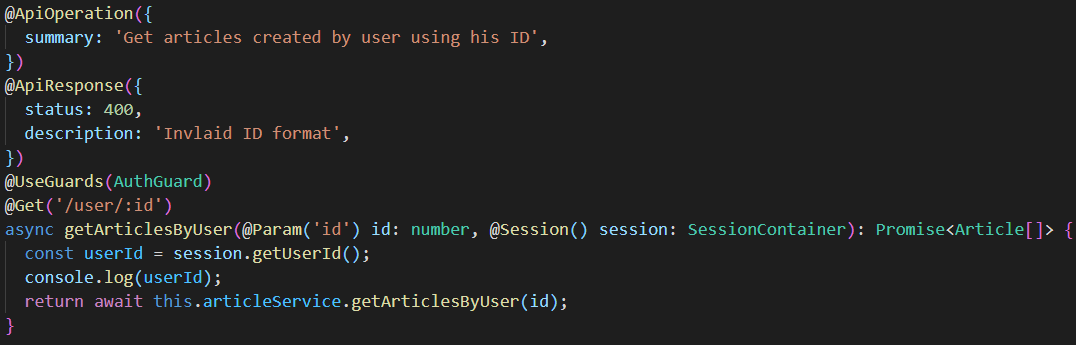


Now we create a session decorator to help us with the created guard to secure endpoints:

nest g decorator session auth



Now to secure an endpoint, it would look something like this:

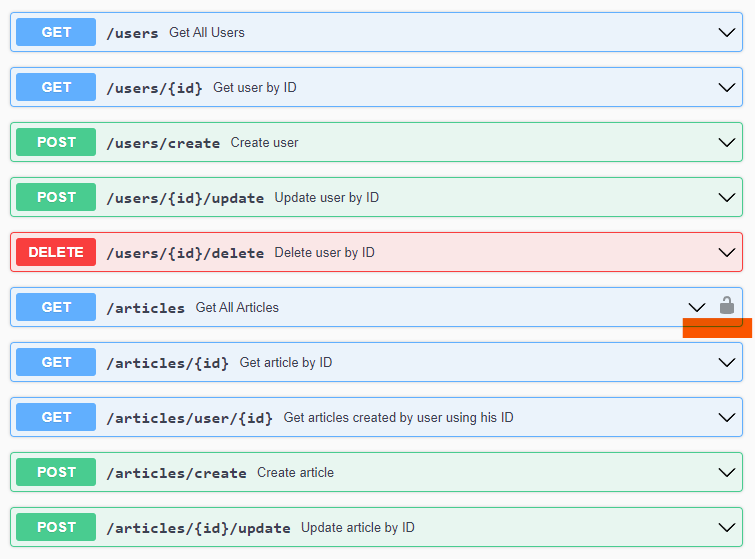


## Logging in

To learn how to use the SuperTokens API we can read its spec from [here](https://app.swaggerhub.com/apis/supertokens/FDI/1.13.0#/EmailPassword%20Recipe/signIn).

In our case we’re using this recipe:

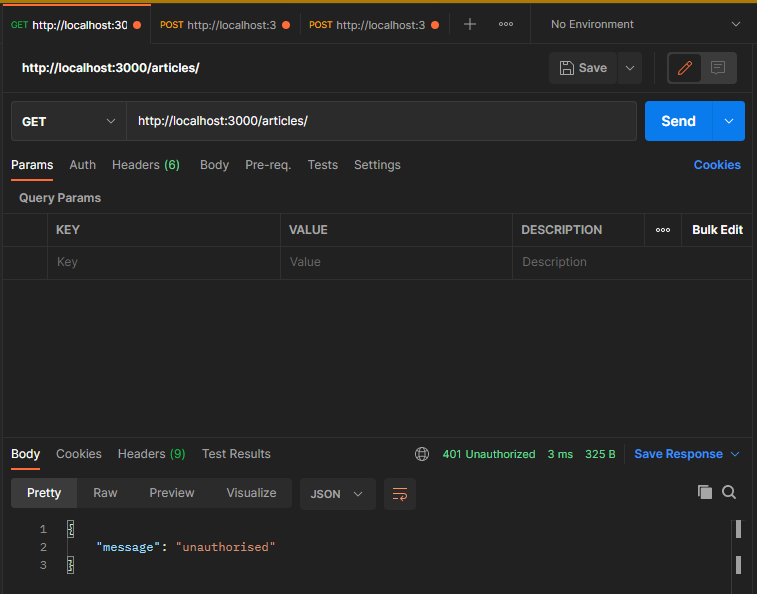
When adding the auth guard we can check swagger and it’ll show a lock sign next to the endpoint:



## Testing the API

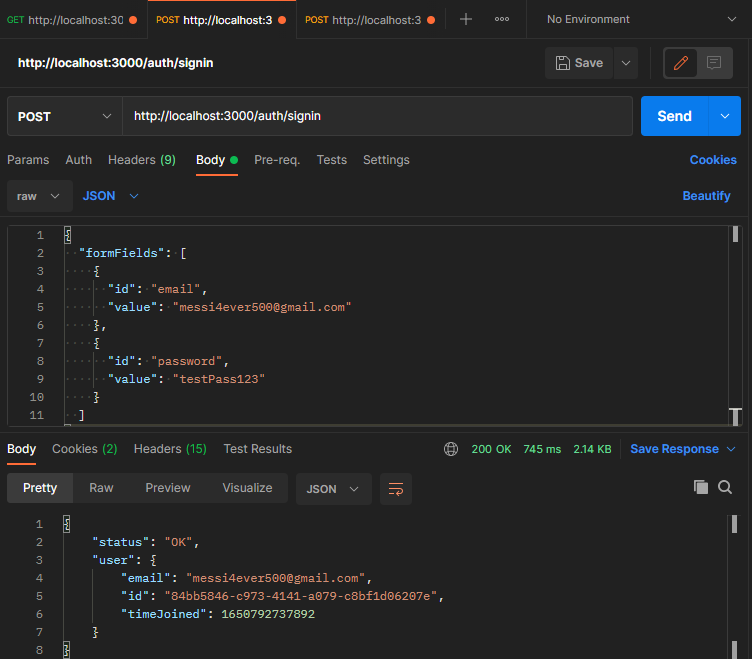
For testing I’m going to use postman, then we’ll modify the frontend to use the new functionality.

To get a list of all the articles with postman we can send a get request like the following:



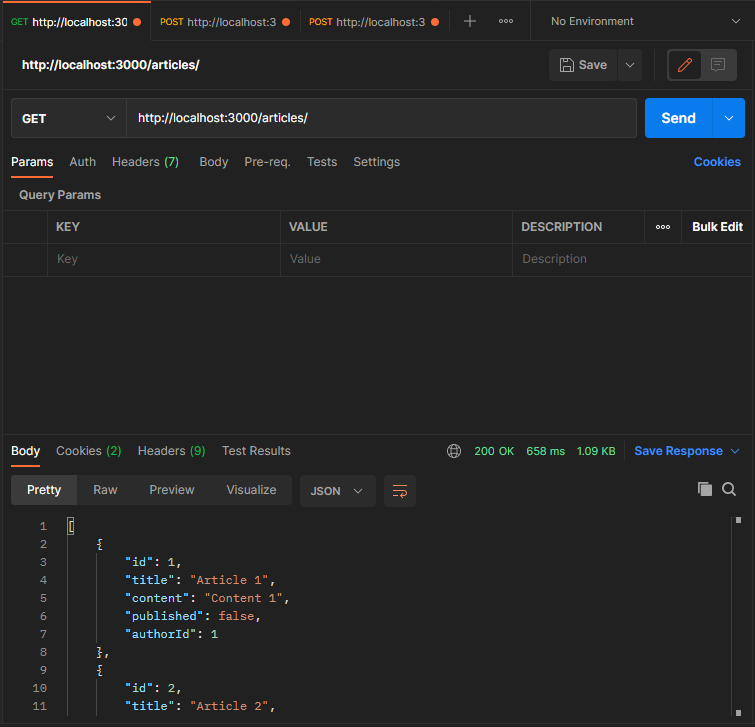
We get a “unauthorised” 401 reply from the API because that endpoint is now protected.

Now let’s try logging in:



Login is successful and token cookie is stored in postman.

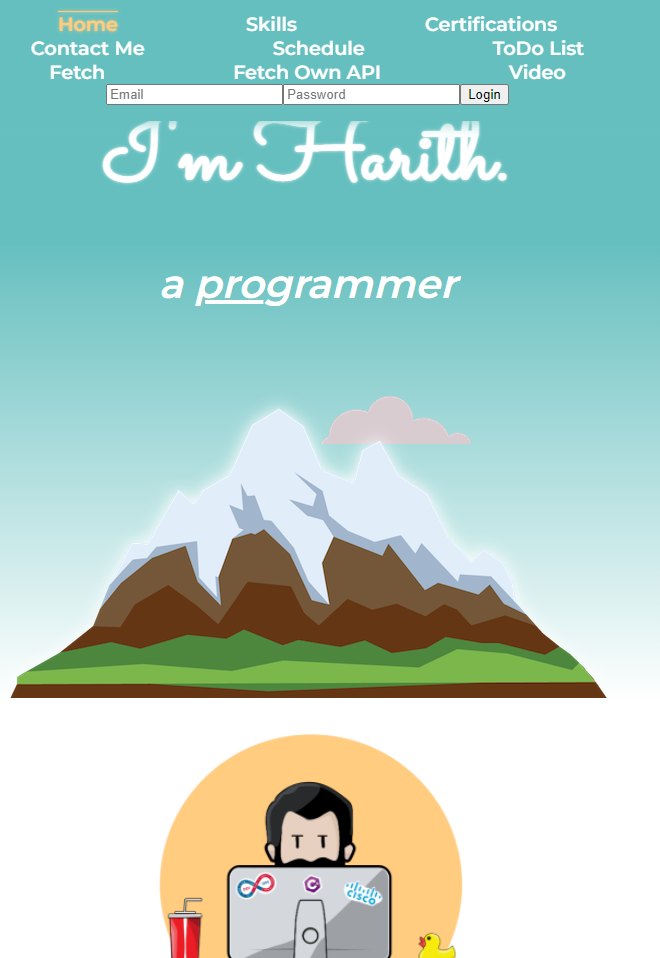
Let’s try accessing the articles endpoint again:



We get a reply with the list of articles!

## Modifying the frontend

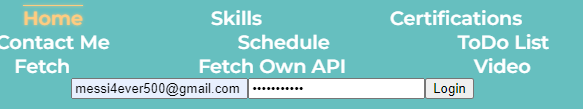
Now that our API is working correctly as required we can go ahead and implement the frontend part related to this:



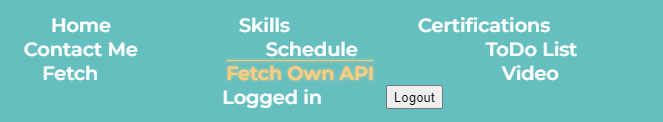
I’ve created a auth.js file and included it in the common partial of the web pages, it has 2 methods, login and logout:

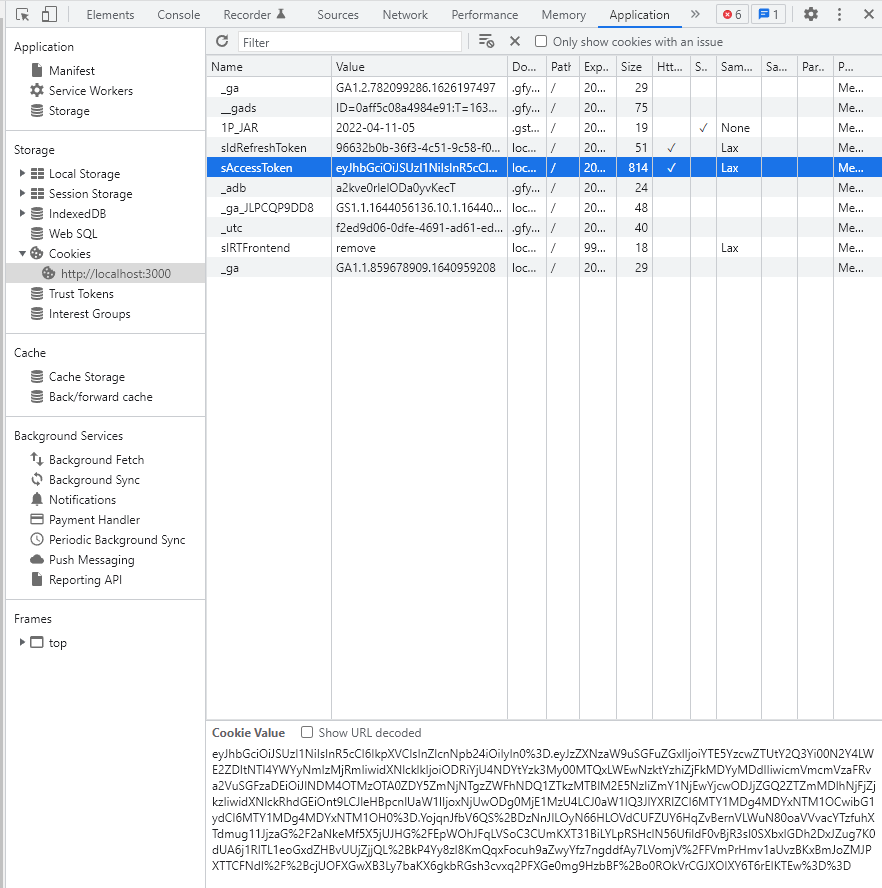
I’ve also modified the HTML form created in the past labs to be able to use it for that matter.

To login simply fill the email and password and hit the login button:



Now we’re logged in and an access cookie is stored in the browser:





Further it’s possible to do this for all the required endpoints and completely secure the application.