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 M33081
Аль Даббагх Харит Хуссейн

Проверил
Приискалов Роман Андреевич

САНКТ-ПЕТЕРБУРГ

2022

СОДЕРЖАНИЕ

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).

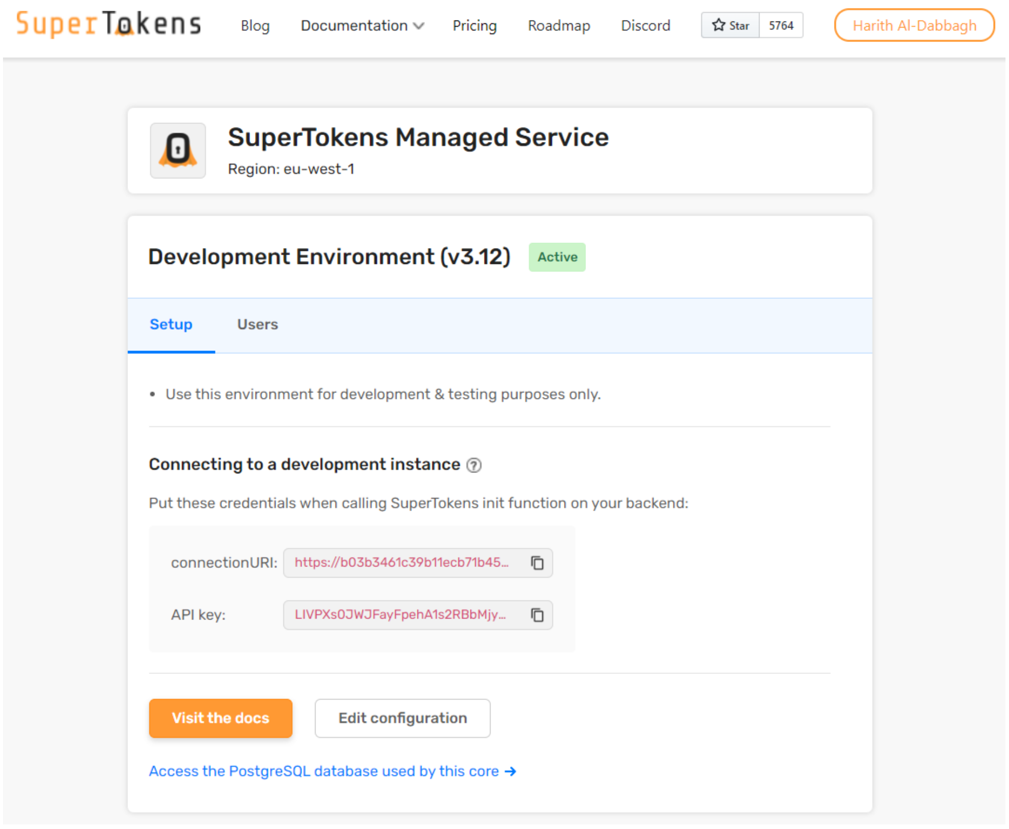
Authorization is the process of verifying what specific resources that user has access to.

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:



SuperTokens Managed Service
Region: eu-west-1

Development Environment (v3.12) Active

- Use this environment for development & testing purposes only.

Connecting to a development instance

Put these credentials when calling SuperTokens init function on your backend:

connectionURI: https://b03b3461c39b11ecb71b45...

API key: LIVPXs0JWJFayFpehA1s2RBbMjy...

Access the PostgreSQL database used by this core →

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:

```
<script src="https://cdn.jsdelivr.net/gh/supertokens/supertokens-website/bundle/bundle.js"></script>
<script>
  supertokens.init({
    apiDomain: "http://localhost:3001",
    apiBasePath: "/auth"
});
</script>
```

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

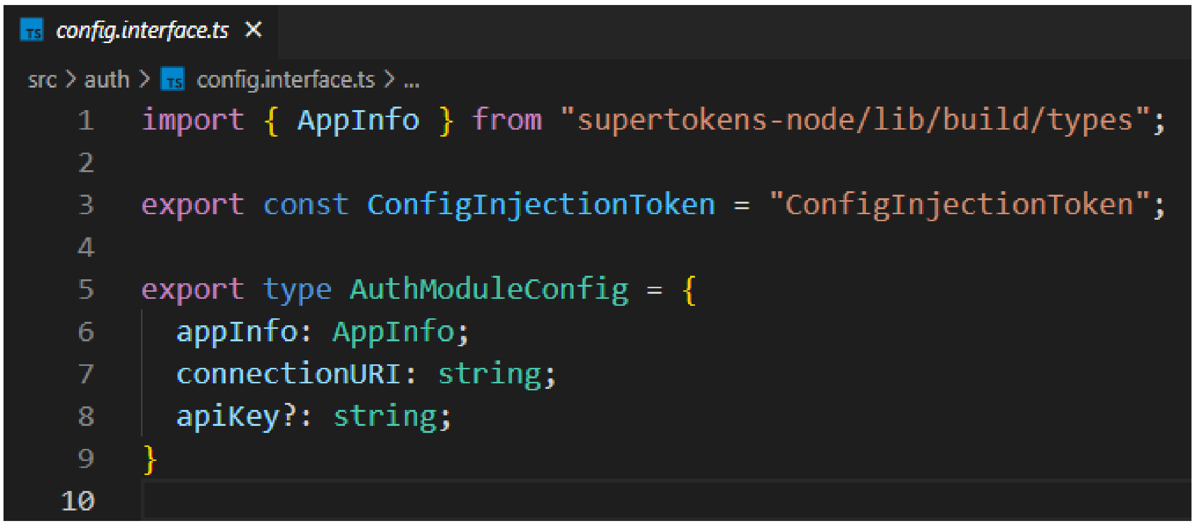
Next we install SuperTokens using the command:

```
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:

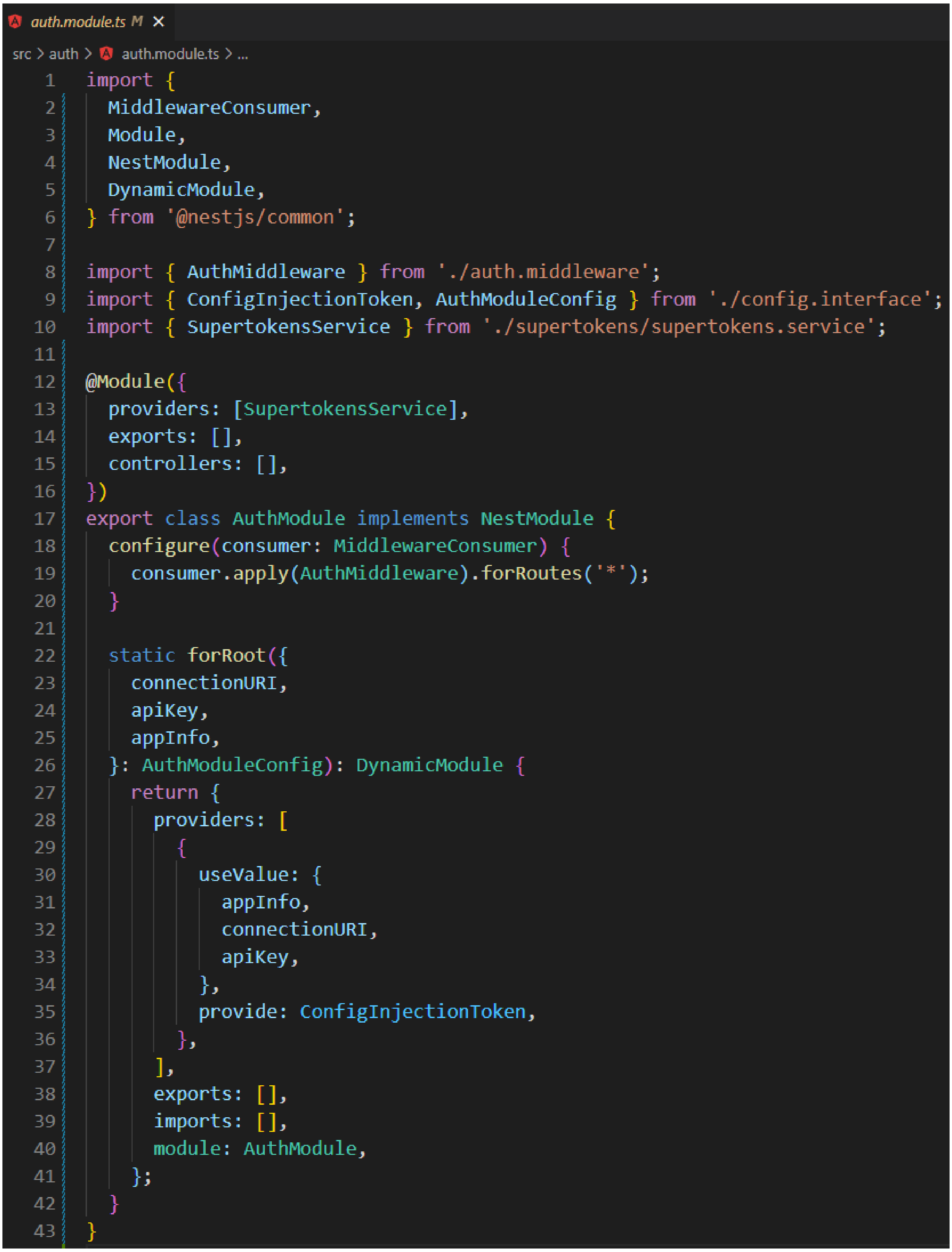
```
nest g module auth
```

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



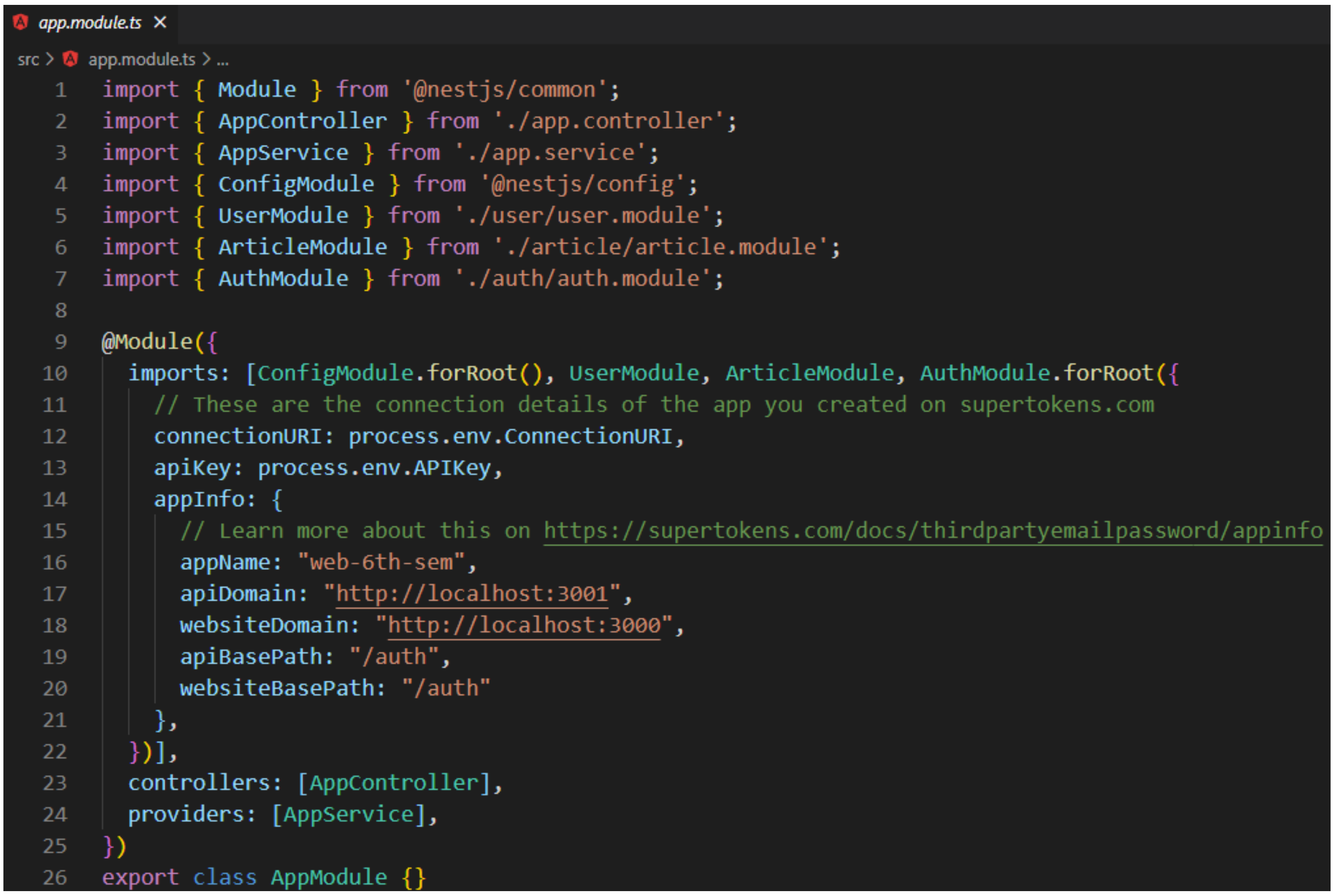
```
import { AppInfo } from "supertokens-node/lib/build/types";

export const ConfigInjectionToken = "ConfigInjectionToken";

export type AuthModuleConfig = {
  appInfo: AppInfo;
  connectionURI: string;
  apiKey?: string;
}

```

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



```
import {
  MiddlewareConsumer,
  Module,
  NestModule,
  DynamicModule,
} from '@nestjs/common';

import { AuthMiddleware } from './auth.middleware';
import { ConfigInjectionToken, AuthModuleConfig } from './config.interface';
import { SupertokensService } from './supertokens/supertokens.service';

@Module({
  providers: [SupertokensService],
  exports: [],
  controllers: [],
})
export class AuthModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer.apply(AuthMiddleware).forRoutes('*');
  }

  static forRoot({
    connectionURI,
    apiKey,
    appInfo,
  }: AuthModuleConfig): DynamicModule {
    return {
      providers: [
        {
          useValue: {
            appInfo,
            connectionURI,
            apiKey,
          },
          provide: ConfigInjectionToken,
        },
      ],
      exports: [],
      imports: [],
      module: AuthModule,
    };
  }
}
```

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:

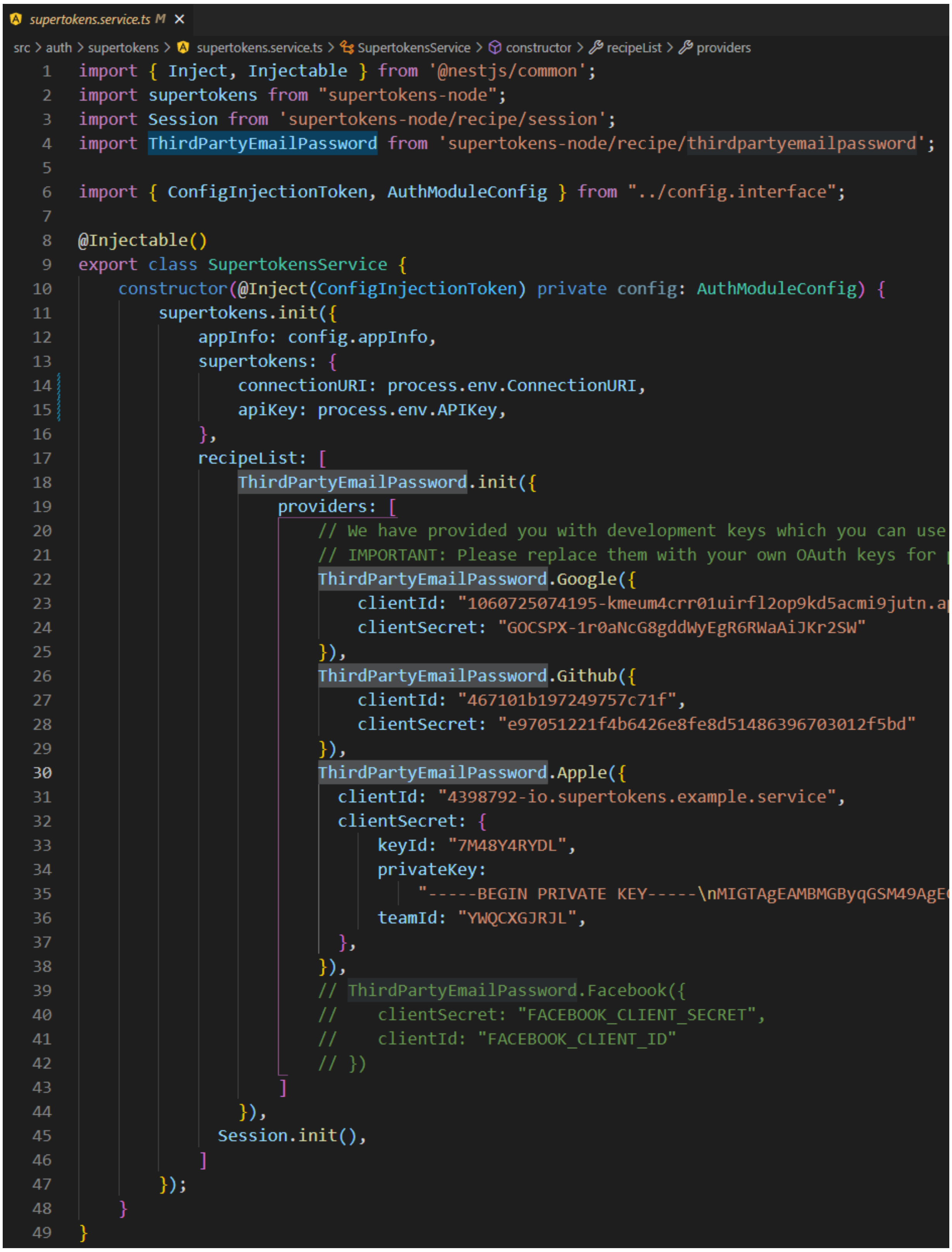


```typescript
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from '@nestjs/config';
import { UserModule } from './user/user.module';
import { ArticleModule } from './article/article.module';
import { AuthModule } from './auth/auth.module';

@Module({
  imports: [ConfigModule.forRoot(), UserModule, ArticleModule, AuthModule.forRoot({
    // These are the connection details of the app you created on supertokens.com
    connectionURI: process.env.ConnectionURI,
    apiKey: process.env.APIKey,
    appInfo: {
      // Learn more about this on https://supertokens.com/docs/thirdpartyemailpassword/appinfo
      appName: "web-6th-sem",
      apiDomain: "http://localhost:3001",
      websiteDomain: "http://localhost:3000",
      apiBasePath: "/auth",
      websiteBasePath: "/auth"
    },
  })],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Let’s now create the service using the command:

```
nest g service supertokens auth
```

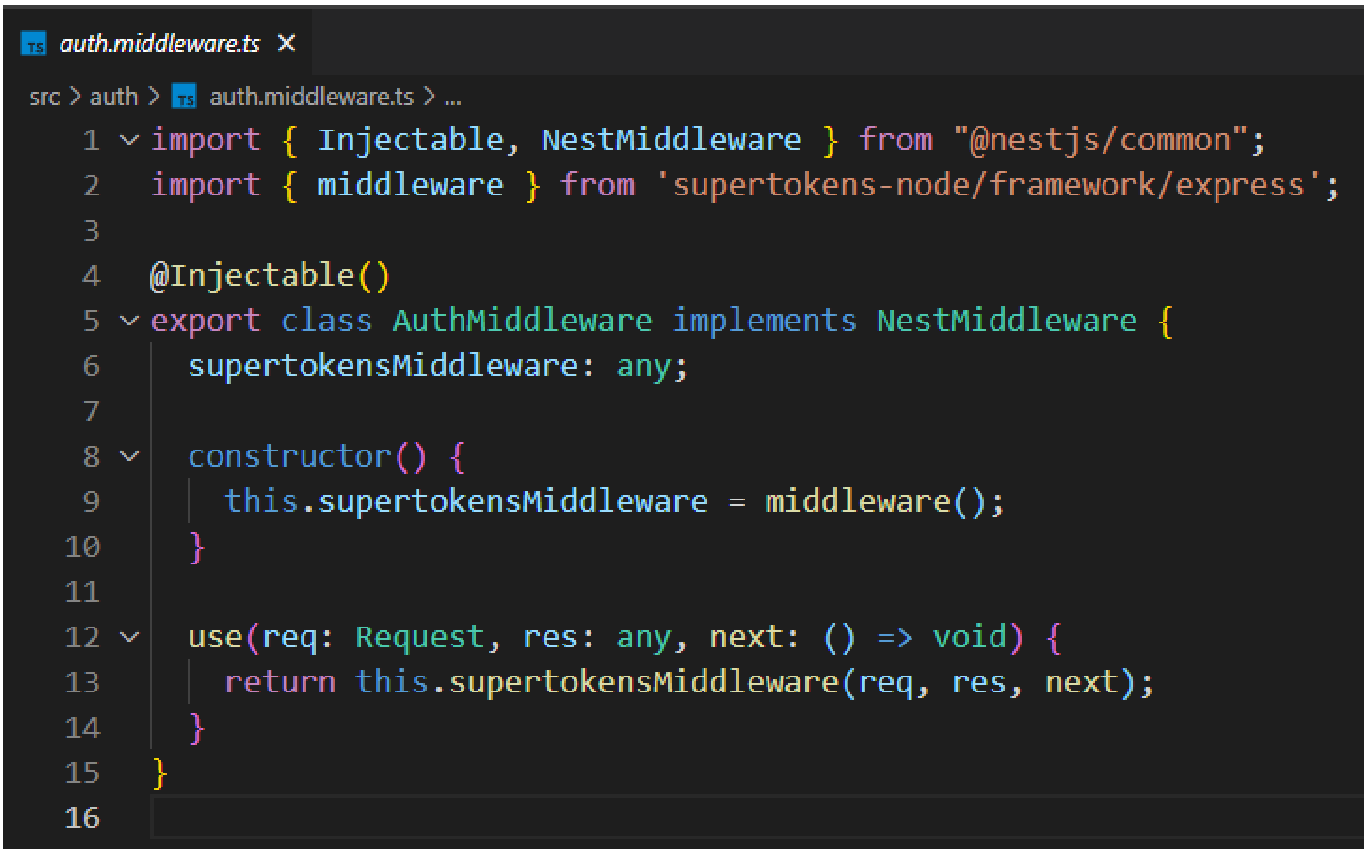
This will create a service separated in it’s own folder inside the auth folder:


supertokens.service.ts M ×
src > auth > supertokens > supertokens.service.ts > SupertokensService > constructor > recipeList > providers

```
import { Inject, Injectable } from '@nestjs/common';
import supertokens from "supertokens-node";
import Session from 'supertokens-node/recipe/session';
import ThirdPartyEmailPassword from 'supertokens-node/recipe/thirdpartyemailpassword';

import { ConfigInjectionToken, AuthModuleConfig } from "../config.interface";

@Injectable()
export class SupertokensService {
  constructor(@Inject(ConfigInjectionToken) private config: AuthModuleConfig) {
    supertokens.init({
      appInfo: config.appInfo,
      supertokens: {
        connectionURI: process.env.ConnectionURI,
        apiKey: process.env.APIKey,
      },
      recipeList: [
        ThirdPartyEmailPassword.init({
          providers: [
            // We have provided you with development keys which you can use
            // IMPORTANT: Please replace them with your own OAuth keys for
            ThirdPartyEmailPassword.Google({
              clientId: "1060725074195-kmeum4crr01uirfl2op9kd5acmi9jutn.a
              clientSecret: "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"
            }),
            ThirdPartyEmailPassword.Github({
              clientId: "467101b197249757c71f",
              clientSecret: "e97051221f4b6426e8fe8d51486396703012f5bd"
            }),
            ThirdPartyEmailPassword.Apple({
             clientId: "4398792-io.supertokens.example.service",
             clientSecret: {
               keyId: "7M48Y4RYDL",
               privateKey:
                 "-----BEGIN PRIVATE KEY-----\nMIGTAgEAMBMGByqGSM49AgE
               teamId: "YWQCXGJRJL",
             },
            }),
            // ThirdPartyEmailPassword.Facebook({
            //    clientSecret: "FACEBOOK_CLIENT_SECRET",
            //    clientId: "FACEBOOK_CLIENT_ID"
            // })
          ]
        }),
       Session.init(),
      ]
    });
  }
}
```

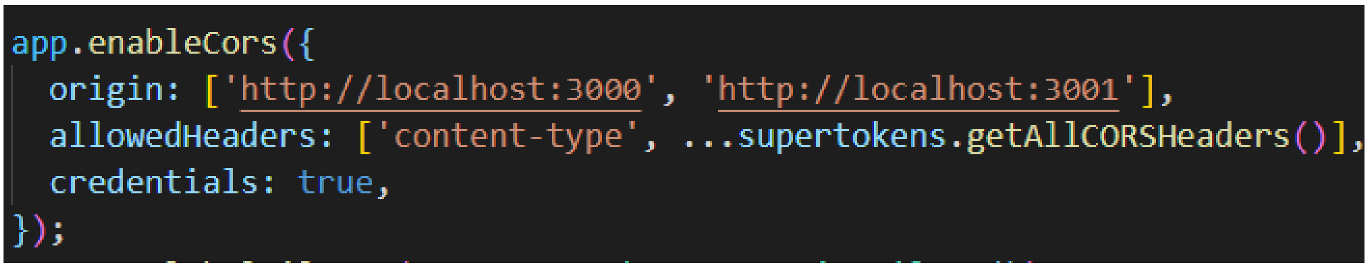
Now we need to create a middleware in order to expose the SuperTokens API. We'll start by the command:

```
nest g middleware auth auth
```



```typescript
import { Injectable, NestMiddleware } from "@nestjs/common";
import { middleware } from 'supertokens-node/framework/express';

@Injectable()
export class AuthMiddleware implements NestMiddleware {
  supertokensMiddleware: any;

  constructor() {
    this.supertokensMiddleware = middleware();
  }

  use(req: Request, res: any, next: () => void) {
    return this.supertokensMiddleware(req, res, next);
  }
}

```

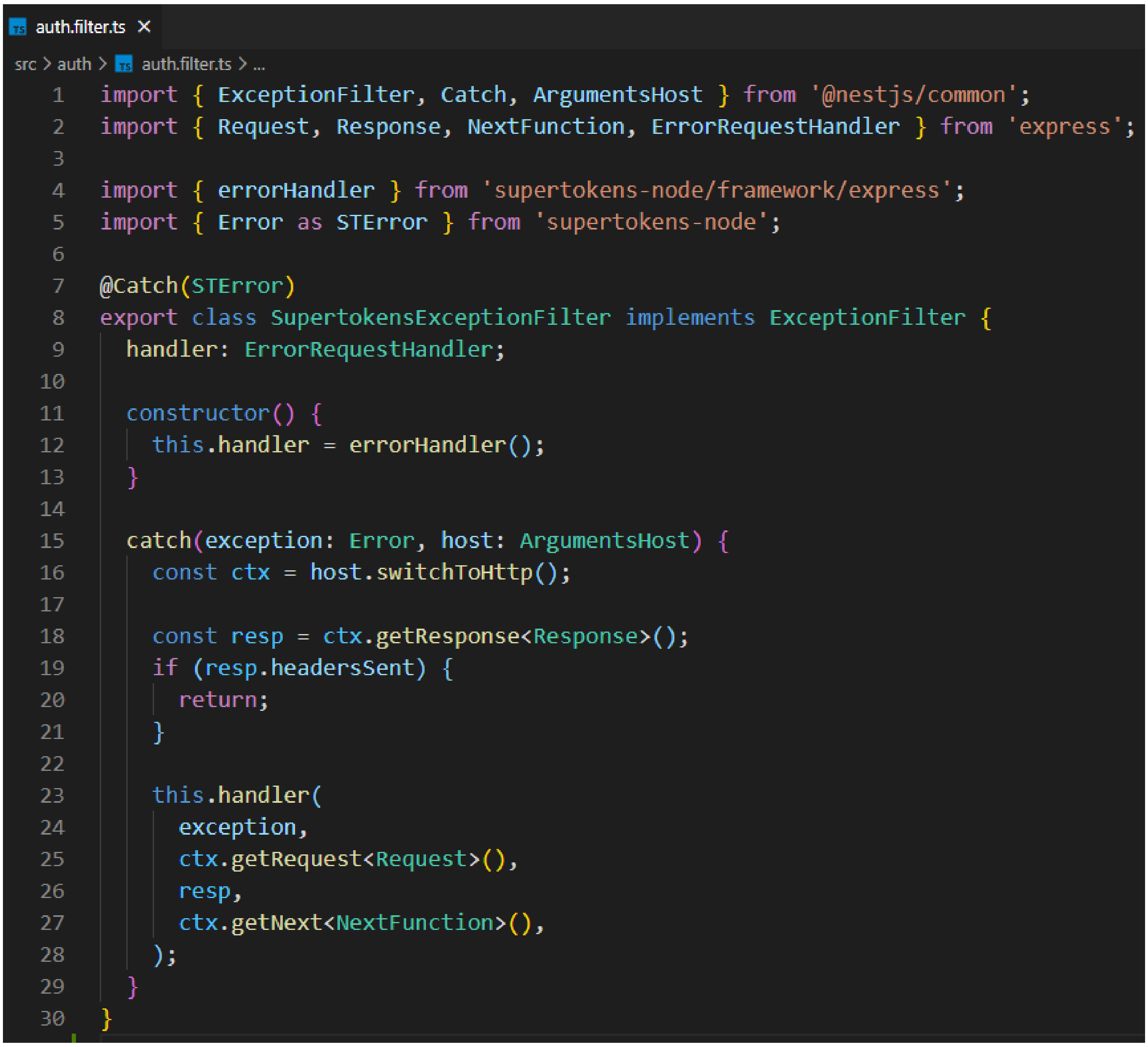
Next up is updating the CORS policy settings. In the main.ts file we need to add the following:



```typescript
app.enableCors({
  origin: ['http://localhost:3000', 'http://localhost:3001'],
  allowedHeaders: ['content-type', ...supertokens.getAllCORSHeaders()],
  credentials: true,
});
```

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

```
nest g filter auth auth
```

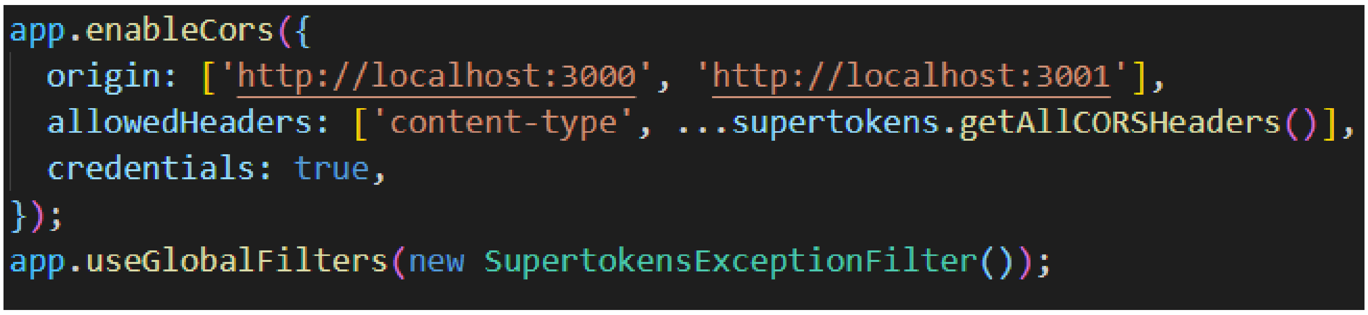


auth.filter.ts ✕

src > auth > auth.filter.ts > ...

```typescript
import { ExceptionFilter, Catch, ArgumentsHost } from '@nestjs/common';
import { Request, Response, NextFunction, ErrorRequestHandler } from 'express';

import { errorHandler } from 'supertokens-node/framework/express';
import { Error as STError } from 'supertokens-node';

@Catch(STError)
export class SupertokensExceptionFilter implements ExceptionFilter {
  handler: ErrorRequestHandler;

  constructor() {
    this.handler = errorHandler();
  }

  catch(exception: Error, host: ArgumentsHost) {
    const ctx = host.switchToHttp();

    const resp = ctx.getResponse<Response>();
    if (resp.headersSent) {
      return;
    }

    this.handler(
      exception,
      ctx.getRequest<Request>(),
      resp,
      ctx.getNext<NextFunction>(),
    );
  }
}
```

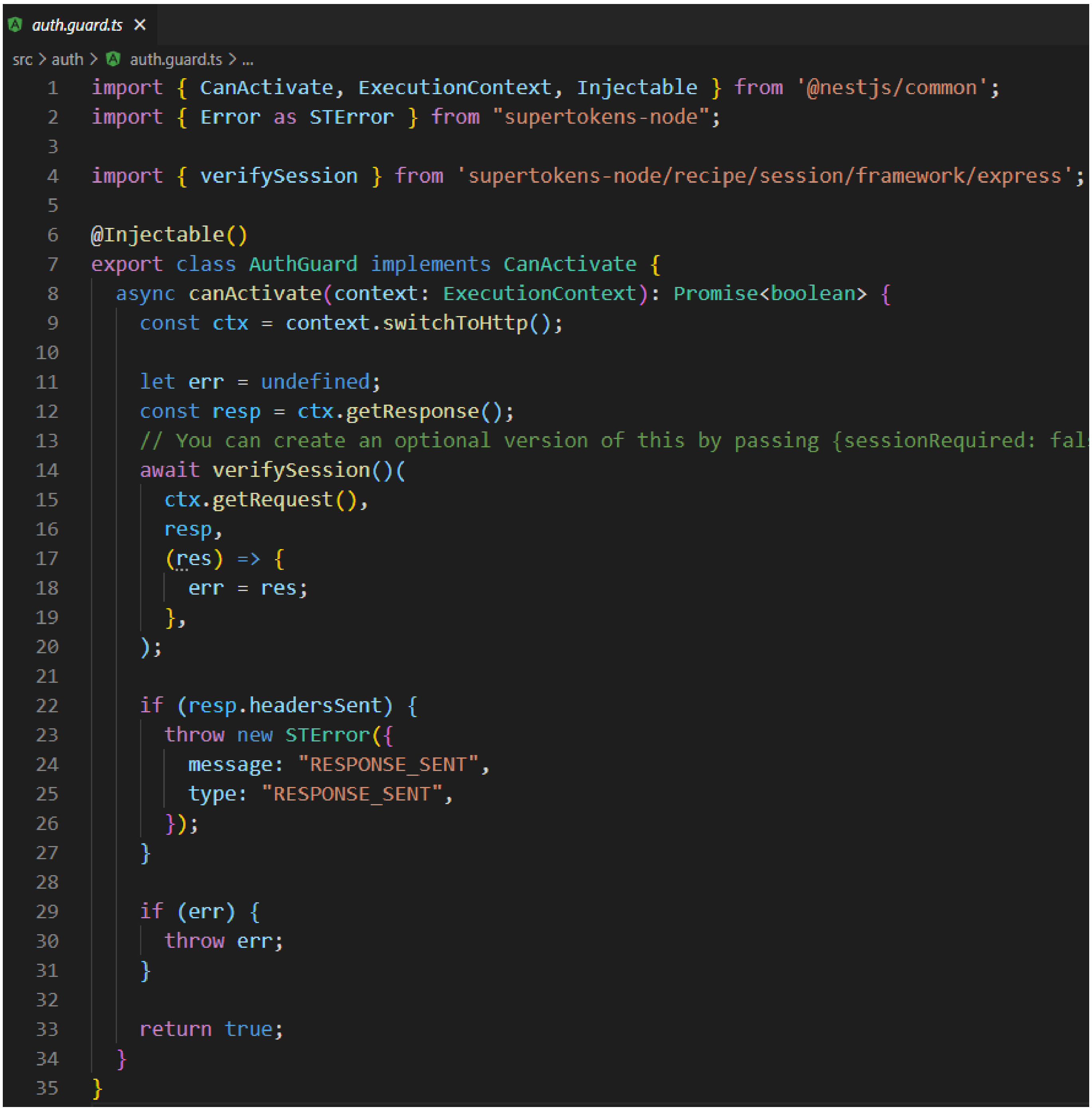
We also need to register the filter by adding a line to main.ts after the CORS policy settings:



```typescript
app.enableCors({
  origin: ['http://localhost:3000', 'http://localhost:3001'],
  allowedHeaders: ['content-type', ...supertokens.getAllCORSHeaders()],
  credentials: true,
});
app.useGlobalFilters(new SupertokensExceptionFilter());
```

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

```
nest g guard auth auth
```

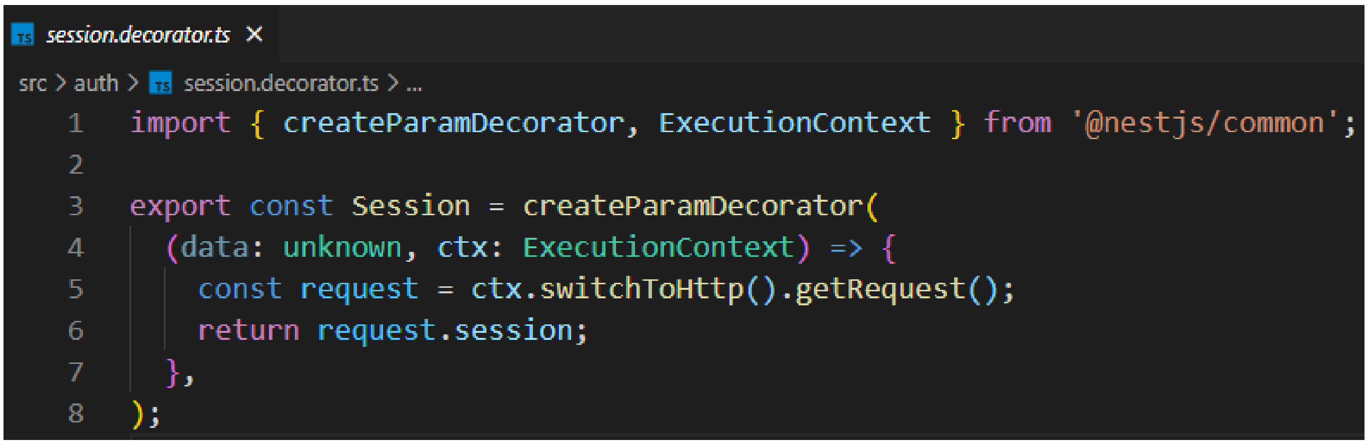

auth.guard.ts ×

src > auth > auth.guard.ts > ...

```typescript
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Error as STError } from "supertokens-node";

import { verifySession } from 'supertokens-node/recipe/session/framework/express';

@Injectable()
export class AuthGuard implements CanActivate {
  async canActivate(context: ExecutionContext): Promise<boolean> {
    const ctx = context.switchToHttp();

    let err = undefined;
    const resp = ctx.getResponse();
    // You can create an optional version of this by passing {sessionRequired: fal
    await verifySession()(
      ctx.getRequest(),
      resp,
      (res) => {
        err = res;
      },
    );

    if (resp.headersSent) {
      throw new STError({
        message: "RESPONSE_SENT",
        type: "RESPONSE_SENT",
      });
    }

    if (err) {
      throw err;
    }

    return true;
  }
}
```

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

```
nest g decorator session auth
```



session.decorator.ts ×

src > auth > session.decorator.ts > ...

```ts
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const Session = createParamDecorator(
  (data: unknown, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    return request.session;
  },
);
```

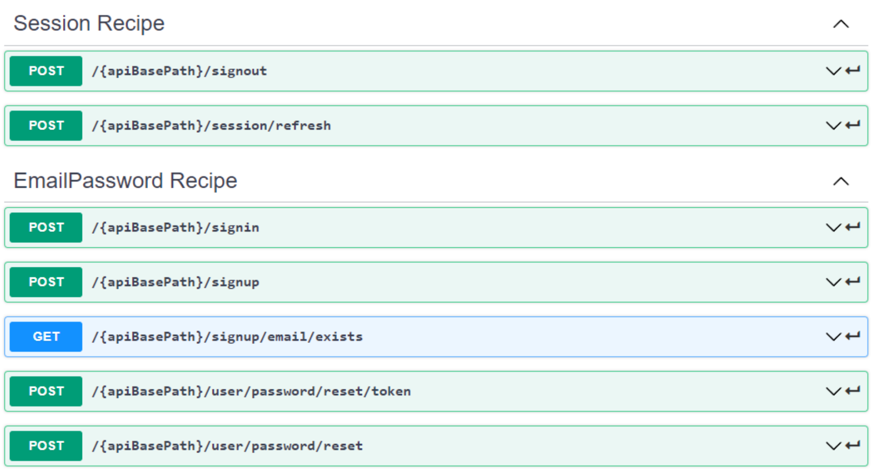
Now to secure an endpoint, it would look something like this:

```ts
@ApiOperation({
  summary: 'Get articles created by user using his ID',
})
@ApiResponse({
  status: 400,
  description: 'Invlaid ID format',
})
@UseGuards(AuthGuard)
@Get('/user/:id')
async getArticlesByUser(@Param('id') id: number, @Session() session: SessionContainer): Promise<Article[]> {
  const userId = session.getUserId();
  console.log(userId);
  return await this.articleService.getArticlesByUser(id);
}
```

Logging in

To learn how to use the SuperTokens API we can read its spec from here.

In our case we're using this recipe:



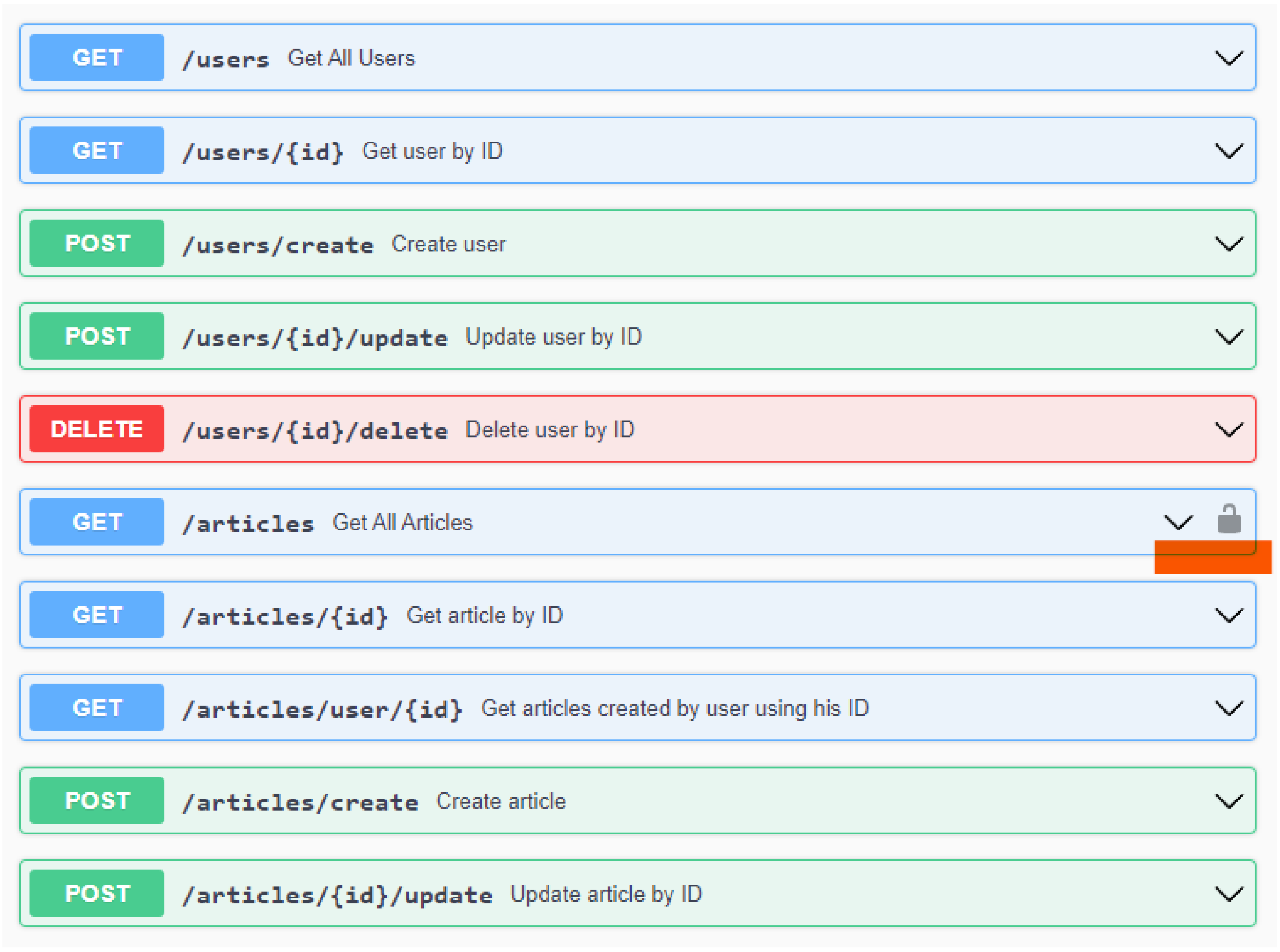
Session Recipe

```
POST /{apiBasePath}/signout
POST /{apiBasePath}/session/refresh
```

EmailPassword Recipe

```
POST /{apiBasePath}/signin
POST /{apiBasePath}/signup
GET  /{apiBasePath}/signup/email/exists
POST /{apiBasePath}/user/password/reset/token
POST /{apiBasePath}/user/password/reset
```

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



GET /users Get All Users

GET /users/{id} Get user by ID

POST /users/create Create user

POST /users/{id}/update Update user by ID

DELETE /users/{id}/delete Delete user by ID

GET /articles Get All Articles

GET /articles/{id} Get article by ID

GET /articles/user/{id} Get articles created by user using his ID

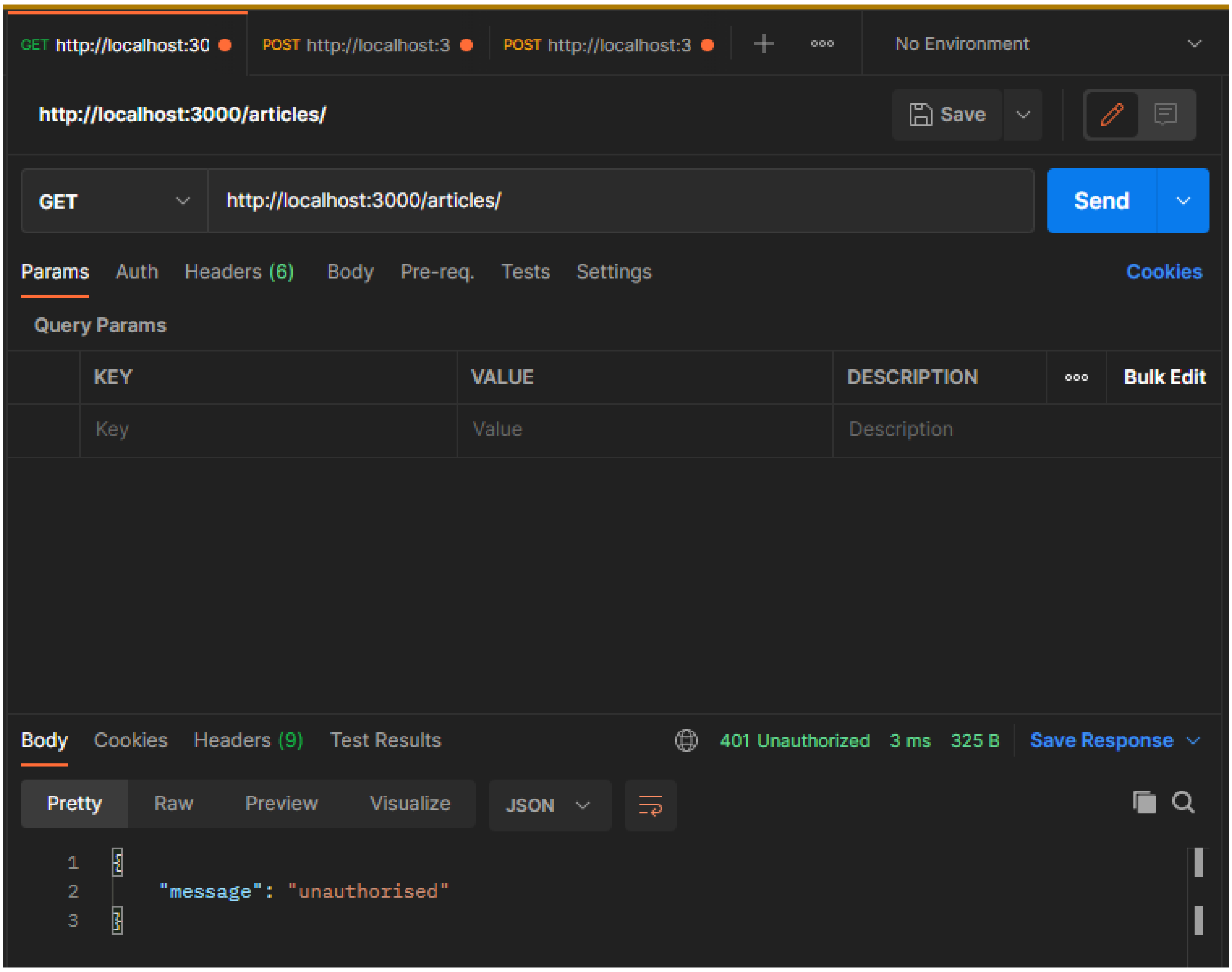
POST /articles/create Create article

POST /articles/{id}/update Update article by ID

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:



http://localhost:3000/articles/

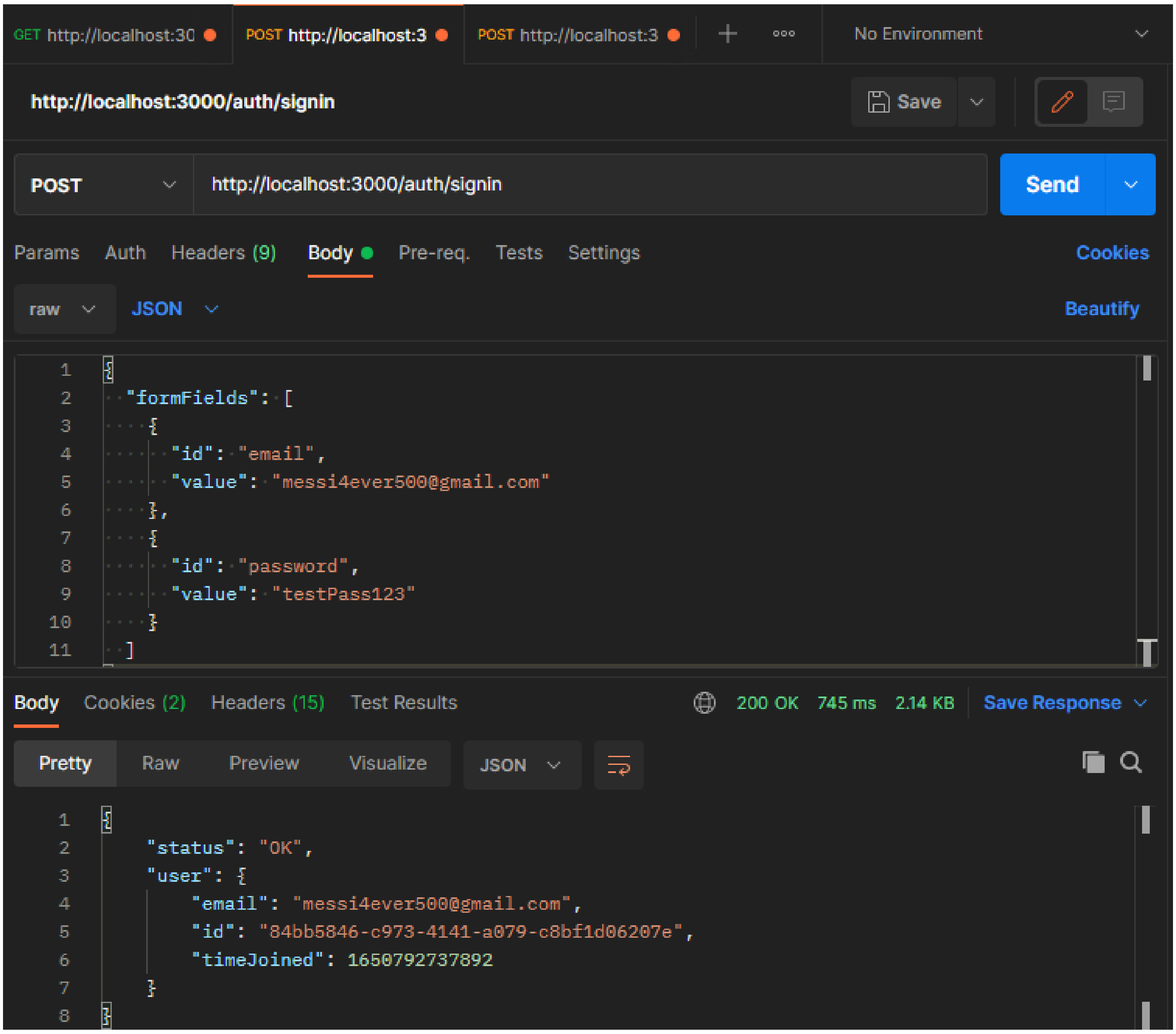
GET http://localhost:3000/articles/

401 Unauthorized 3 ms 325 B

```
{
    "message": "unauthorised"
}
```

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

Now let’s try logging in:



http://localhost:3000/auth/signin

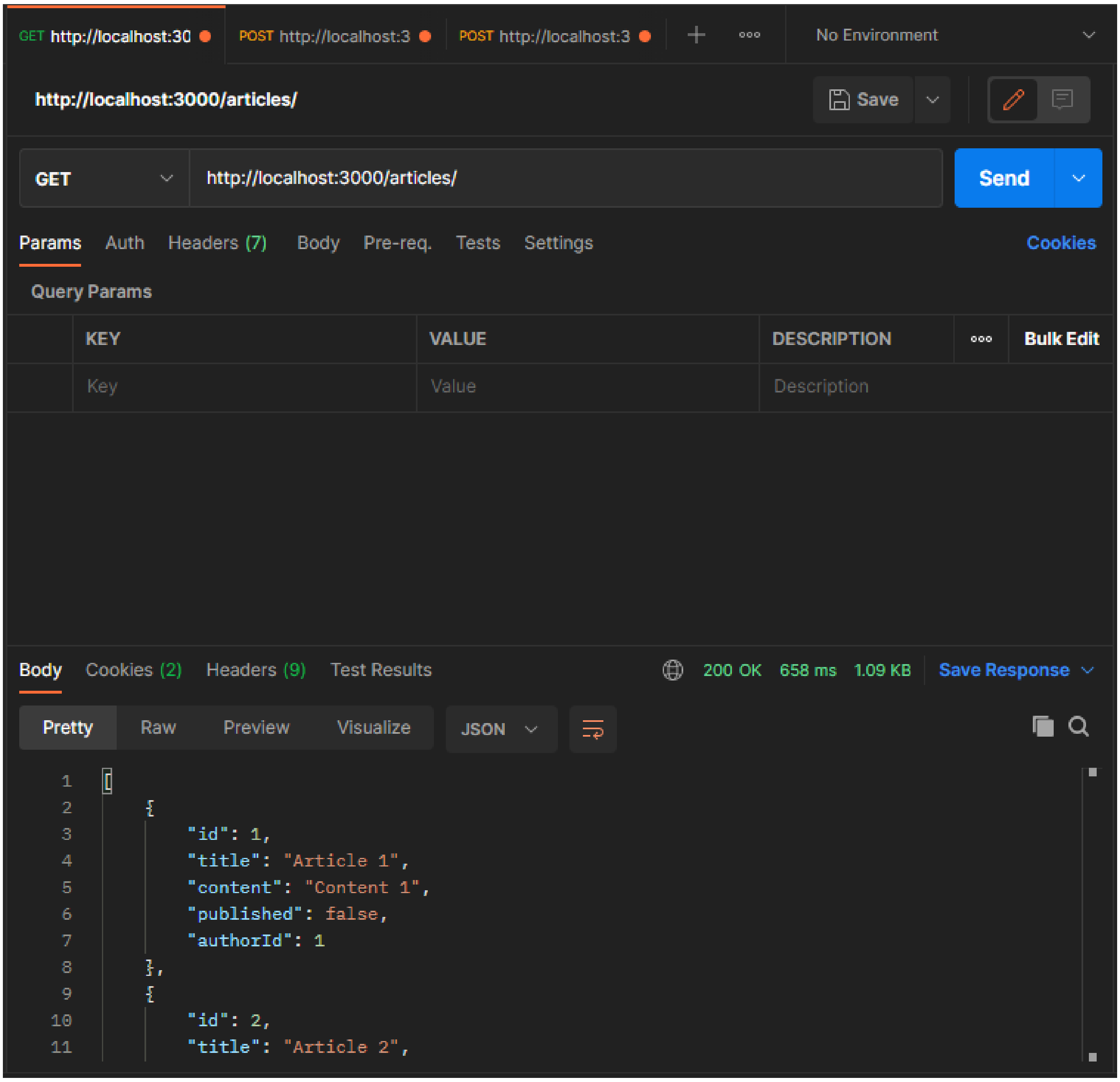
POST http://localhost:3000/auth/signin

```
{
  "formFields": [
    {
      "id": "email",
      "value": "messi4ever500@gmail.com"
    },
    {
      "id": "password",
      "value": "testPass123"
    }
  ]
```

```
{
    "status": "OK",
    "user": {
        "email": "messi4ever500@gmail.com",
        "id": "84bb5846-c973-4141-a079-c8bf1d06207e",
        "timeJoined": 1650792737892
    }
}
```

Login is successful and token cookie is stored in postman.

Let’s try accessing the articles endpoint again:

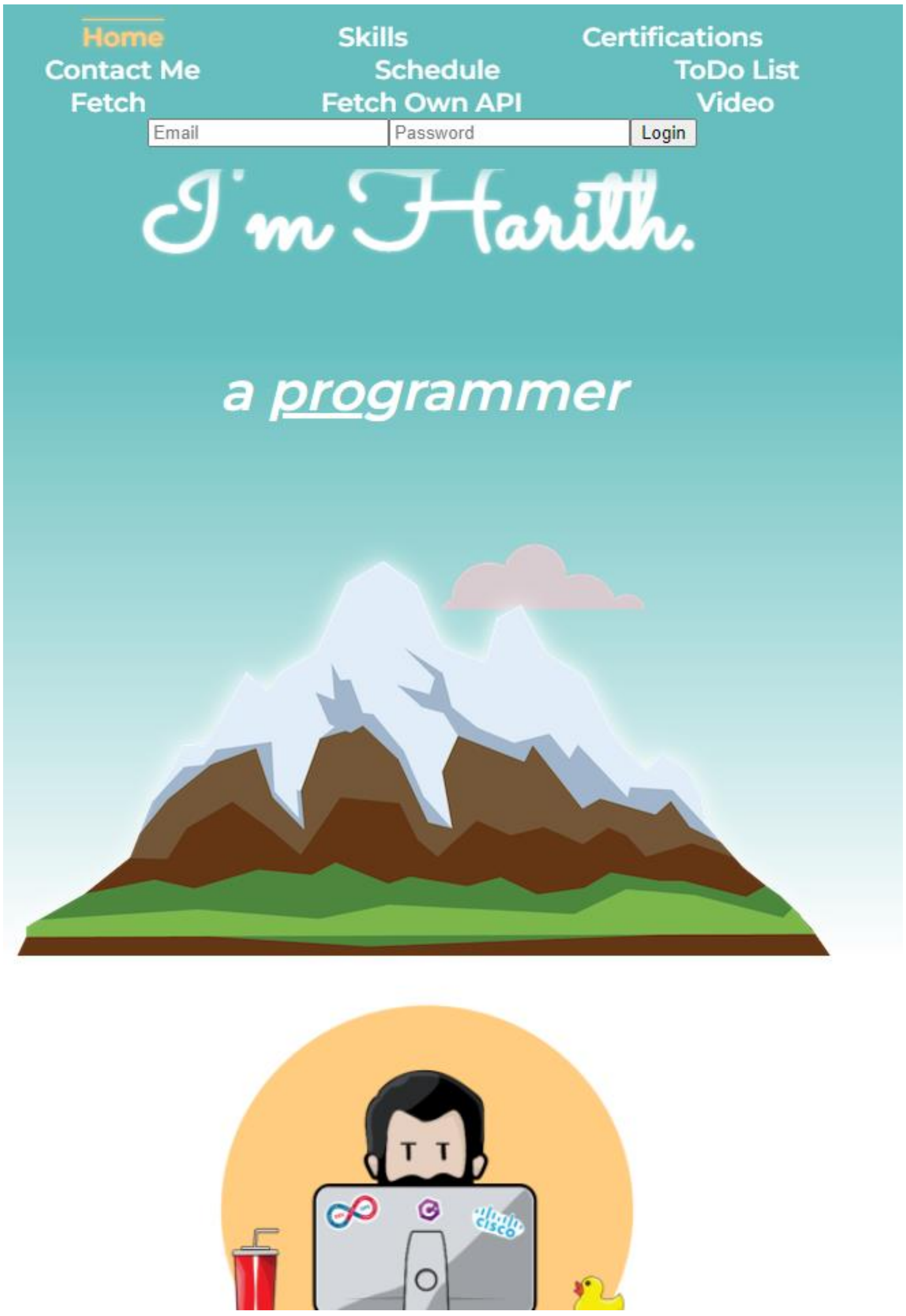


```
[
    {
        "id": 1,
        "title": "Article 1",
        "content": "Content 1",
        "published": false,
        "authorId": 1
    },
    {
        "id": 2,
        "title": "Article 2",
```

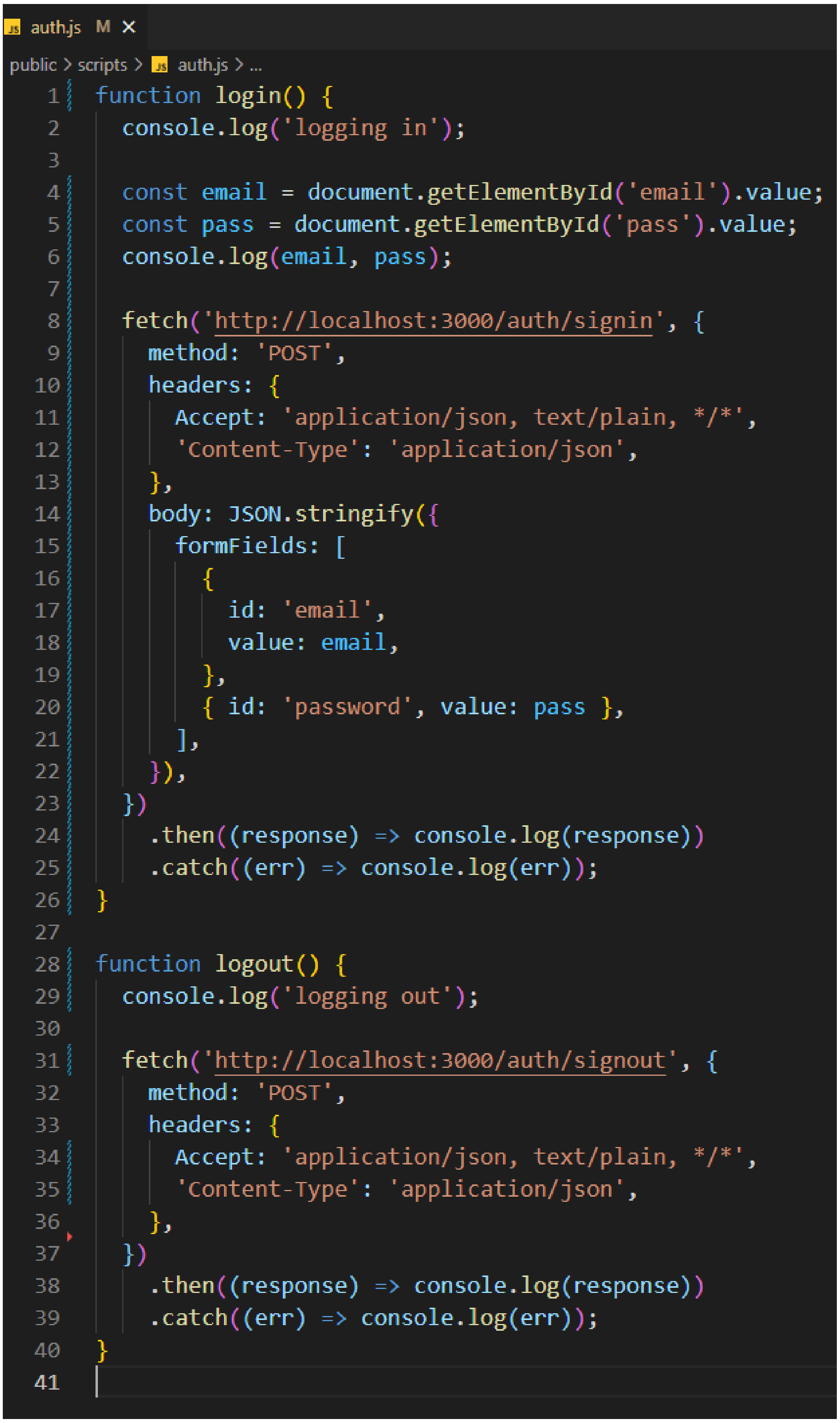
We get a reply with the list of articles!

Modifying the frontend

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



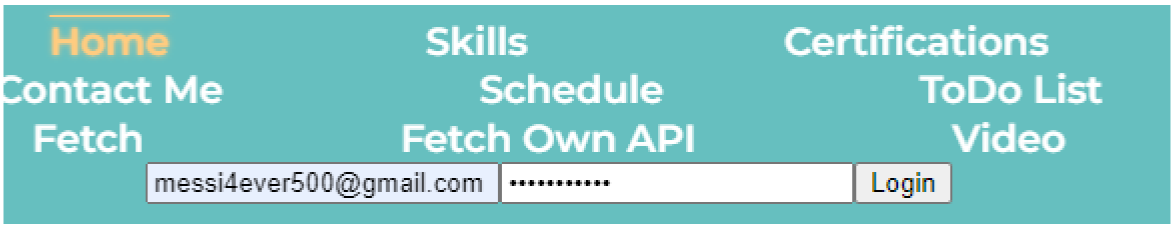
I've created a auth.js file and included it in the common partial of the web pages, it has 2 methods, login and logout:



```
function login() {
  console.log('logging in');

  const email = document.getElementById('email').value;
  const pass = document.getElementById('pass').value;
  console.log(email, pass);

  fetch('http://localhost:3000/auth/signin', {
    method: 'POST',
    headers: {
      Accept: 'application/json, text/plain, */*',
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      formFields: [
        {
          id: 'email',
          value: email,
        },
        { id: 'password', value: pass },
      ],
    }),
  })
    .then((response) => console.log(response))
    .catch((err) => console.log(err));
}

function logout() {
  console.log('logging out');

  fetch('http://localhost:3000/auth/signout', {
    method: 'POST',
    headers: {
      Accept: 'application/json, text/plain, */*',
      'Content-Type': 'application/json',
    },
  })
    .then((response) => console.log(response))
    .catch((err) => console.log(err));
}

```

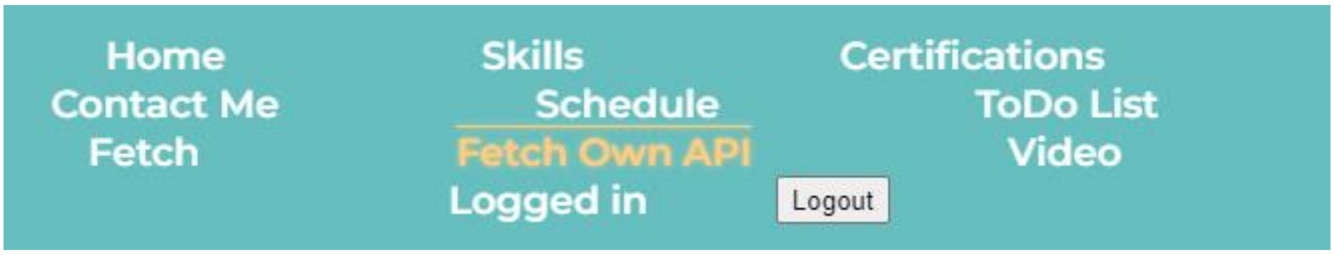
I’ve also modified the HTML form created in the past labs to be able to use it for that matter.

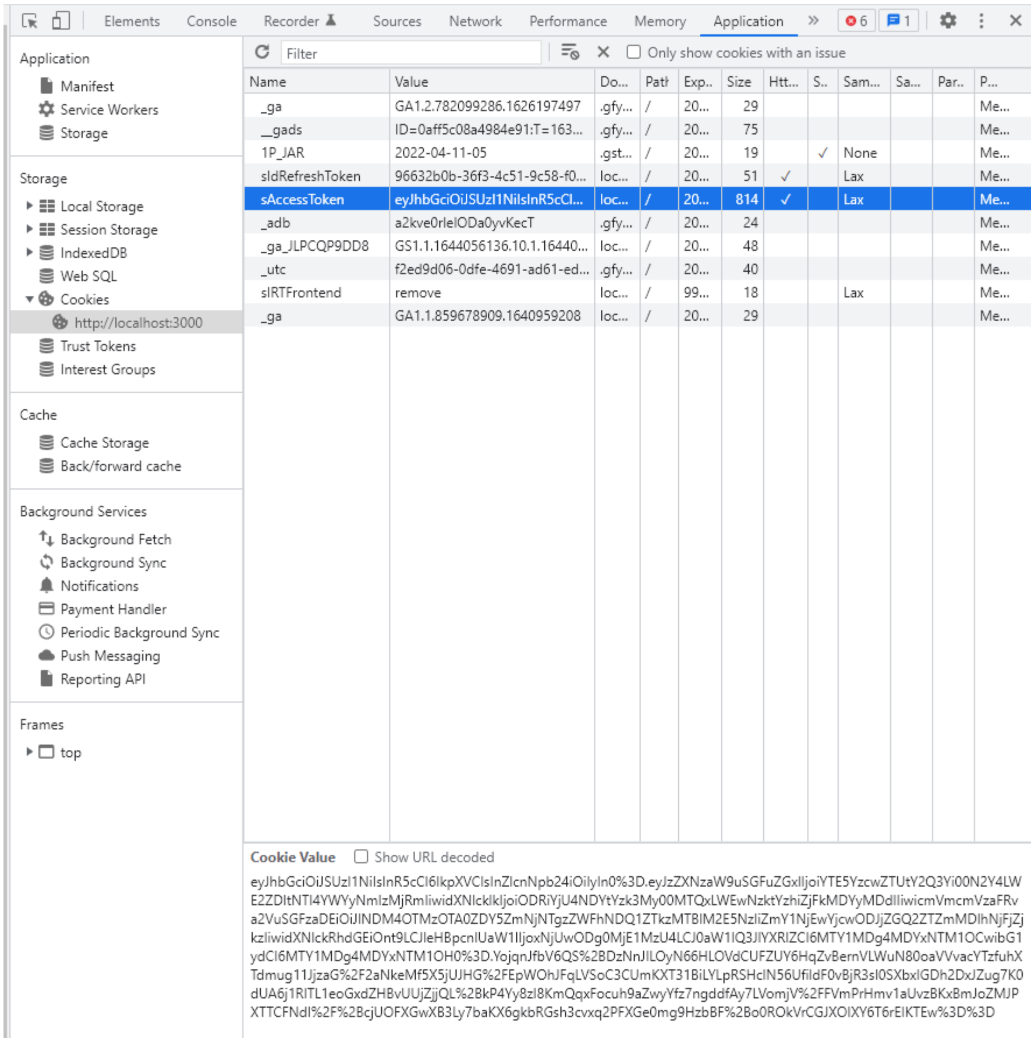
To login simply fill the email and password and hit the login button:



messi4ever500@gmail.com ••••••••••••

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





Name	Value	Do...	Path	Exp...	Size	Htt...	S...	Sam...	Sa...	Par...	P...
_ga	GA1.2.782099286.1626197497	.gfy...	/	20...	29						Me...
__gads	ID=0aff5c08a4984e91:T=163...	.gfy...	/	20...	75						Me...
1P_JAR	2022-04-11-05	.gst...	/	20...	19		✓	None			Me...
sldRefreshToken	96632b0b-36f3-4c51-9c58-f0...	loc...	/	20...	51	✓		Lax			Me...
sAccessToken	eyJhbGciOiJSUzI1NiIsInR5cCI...	loc...	/	20...	814	✓		Lax			Me...
_adb	a2kve0rleIODa0yvKecT	.gfy...	/	20...	24						Me...
_ga_JLPCQP9DD8	GS1.1.1644056136.10.1.16440...	loc...	/	20...	48						Me...
_utc	f2ed9d06-0dfe-4691-ad61-ed...	.gfy...	/	20...	40						Me...
sIRTFrontend	remove	loc...	/	99...	18			Lax			Me...
_ga	GA1.1.859678909.1640959208	loc...	/	20...	29						Me...

Cookie Value ☐ Show URL decoded

eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsInZlcnNpb24iOiIyIn0%3D.eyJzZXNzaW9uSGFuZGxlIjoiYTE5YzcwZTUtY2Q3Yi00N2Y4LW
E2ZDItNTI4YWYyNmIzMjRmIiwidXNlcklkIjoiODRiYjU4NDYtYzk3My00MTQxLWEwNzktYzhiZjFkMDYyMDdlIiwicmVmcmVzaFRv
a2VuSGFzaDEiOiJlNDM4OTMzOTA0ZDY5ZmNjNTgzZWFhNDQ1ZTkzMTBlM2E5NzliZmY1NjEwYjcwODJjZGQ2ZTZmMDlhNjFjZj
kzliwidXNlckRhdGEiOnt9LCJleHBpcnlUaW1lIjoxNjUwODg0MjE1MzU4LCJ0aW1lQ3JlYXRlZCI6MTY1MDg4MDYxNTM1OCwibG1
ydCI6MTY1MDg4MDYxNTM1OH0%3D.YojqnJfbV6QS%2BDzNnJILOyN66HLOVdCUFZUY6HqZvBernVLWuN80oaVVvacYTzfuhX
Tdmug11JjzaG%2F2aNkeMf5X5jUJHG%2FEpWOhJFqLVSoC3CUmKXT31BiLYLpRSHcIN56UfildF0vBjR3sI0SXbxIGDh2DxJZug7K0
dUA6j1RITL1eoGxdZHBvUUjZjjQL%2BkP4Yy8zI8KmQqxFocuh9aZwyYfz7ngddfAy7LVomjV%2FFVmPrHmv1aUvzBKxBmJoZMJP
XTTCFNdI%2F%2BcjUOFXGwXB3Ly7baKX6gkbRGsh3cvxq2PFXGe0mg9HzbBF%2Bo0ROkVrCGJXOIXY6T6rEIKTEw%3D%3D

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