Authentication

Authentication is an **essential** part of most applications. There are many different approaches and strategies to handle authentication. The approach taken for any project depends on its particular application requirements. This chapter presents several approaches to authentication that can be adapted to a variety of different requirements.

Let's flesh out our requirements. For this use case, clients will start by authenticating with a username and password. Once authenticated, the server will issue a JWT that can be sent as a bearer token in an authorization header on subsequent requests to prove authentication. We'll also create a protected route that is accessible only to requests that contain a valid JWT.

We'll start with the first requirement: authenticating a user. We'll then extend that by issuing a JWT. Finally, we'll create a protected route that checks for a valid JWT on the request.

Creating an authentication module

We'll start by generating an `AuthModule` and in it, an `AuthService` and an `AuthController`. We'll use the `AuthService` to implement the authentication logic, and the `AuthController` to expose the authentication endpoints.

```
$ nest g module auth
$ nest g controller auth
$ nest g service auth
```

As we implement the `AuthService`, we'll find it useful to encapsulate user operations in a `UsersService`, so let's generate that module and service now:

```
$ nest g module users
$ nest g service users
```

Replace the default contents of these generated files as shown below. For our sample app, the `UsersService` simply maintains a hard-coded in-memory list of users, and a find method to retrieve one by username. In a real app, this is where you'd build your user model and persistence layer, using your library of choice (e.g., TypeORM, Sequelize, Mongoose, etc.).

```
@@filename(users/users.service)
import { Injectable } from '@nestjs/common';

// This should be a real class/interface representing a user entity
export type User = any;

@Injectable()
export class UsersService {
  private readonly users = [
```

```
    {
      userId: 1,
      username: 'john',
      password: 'changeme',
    },
    {
      userId: 2,
      username: 'maria',
      password: 'guess',
    },
  ];

  async findOne(username: string): Promise<User | undefined> {
    return this.users.find(user => user.username === username);
  }
}
@@switch
import { Injectable } from '@nestjs/common';

@Injectable()
export class UsersService {
  constructor() {
    this.users = [
      {
        userId: 1,
        username: 'john',
        password: 'changeme',
      },
      {
        userId: 2,
        username: 'maria',
        password: 'guess',
      },
    ];
  }

  async findOne(username) {
    return this.users.find(user => user.username === username);
  }
}
```

In the `UsersModule`, the only change needed is to add the `UsersService` to the exports array of the `@Module` decorator so that it is visible outside this module (we'll soon use it in our `AuthService`).

```
@@filename(users/users.module)
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';

@Module({
  providers: [UsersService],
  exports: [UsersService],
```

```
})
export class UsersModule {}
@@switch
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';

@Module({
  providers: [UsersService],
  exports: [UsersService],
})
export class UsersModule {}
```

Implementing the "Sign in" endpoint

Our `AuthService` has the job of retrieving a user and verifying the password. We create a `signIn()` method for this purpose. In the code below, we use a convenient ES6 spread operator to strip the password property from the user object before returning it. This is a common practice when returning user objects, as you don't want to expose sensitive fields like passwords or other security keys.

```
@@filename(auth/auth.service)
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { UsersService } from '../users/users.service';

@Injectable()
export class AuthService {
  constructor(private usersService: UsersService) {}

  async signIn(username: string, pass: string): Promise<any> {
    const user = await this.usersService.findOne(username);
    if (user?.password !== pass) {
      throw new UnauthorizedException();
    }
    const { password, ...result } = user;
    // TODO: Generate a JWT and return it here
    // instead of the user object
    return result;
  }
}
@@switch
import { Injectable, Dependencies, UnauthorizedException } from
'@nestjs/common';
import { UsersService } from '../users/users.service';

@Injectable()
@Dependencies(UsersService)
export class AuthService {
  constructor(usersService) {
    this.usersService = usersService;
  }

  async signIn(username: string, pass: string) {
```

```
    const user = await this.usersService.findOne(username);
    if (user?.password !== pass) {
      throw new UnauthorizedException();
    }
    const { password, ...result } = user;
    // TODO: Generate a JWT and return it here
    // instead of the user object
    return result;
  }
}
```

> Warning **Warning** Of course in a real application, you wouldn't store a password in plain text. You'd instead use a library like bcrypt, with a salted one-way hash algorithm. With that approach, you'd only store hashed passwords, and then compare the stored password to a hashed version of the **incoming** password, thus never storing or exposing user passwords in plain text. To keep our sample app simple, we violate that absolute mandate and use plain text. **Don't do this in your real app!**

Now, we update our `AuthModule` to import the `UsersModule`.

```
@@filename(auth/auth.module)
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { UsersModule } from '../users/users.module';

@Module({
  imports: [UsersModule],
  providers: [AuthService],
  controllers: [AuthController],
})
export class AuthModule {}
@@switch
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { UsersModule } from '../users/users.module';

@Module({
  imports: [UsersModule],
  providers: [AuthService],
  controllers: [AuthController],
})
export class AuthModule {}
```

With this in place, let's open up the `AuthController` and add a `signIn()` method to it. This method will be called by the client to authenticate a user. It will receive the username and password in the request body, and will return a JWT token if the user is authenticated.

```
@@filename(auth/auth.controller)
import { Body, Controller, Post, HttpCode, HttpStatus } from
'@nestjs/common';
import { AuthService } from './auth.service';

@Controller('auth')
export class AuthController {
  constructor(private authService: AuthService) {}

  @HttpCode(HttpStatus.OK)
  @Post('login')
  signIn(@Body() signInDto: Record<string, any>) {
    return this.authService.signIn(signInDto.username,
signInDto.password);
  }
}
```

> info **Hint** Ideally, instead of using the `Record<string, any>` type, we should use a DTO class to define the shape of the request body. See the validation chapter for more information.

JWT token

We're ready to move on to the JWT portion of our auth system. Let's review and refine our requirements:

- Allow users to authenticate with username/password, returning a JWT for use in subsequent calls to protected API endpoints. We're well on our way to meeting this requirement. To complete it, we'll need to write the code that issues a JWT.
- Create API routes which are protected based on the presence of a valid JWT as a bearer token

We'll need to install one additional package to support our JWT requirements:

```
$ npm install --save @nestjs/jwt
```

> info **Hint** The `@nestjs/jwt` package (see more here) is a utility package that helps with JWT manipulation. This includes generating and verifying JWT tokens.

To keep our services cleanly modularized, we'll handle generating the JWT in the `authService`. Open the `auth.service.ts` file in the `auth` folder, inject the `JwtService`, and update the `signIn` method to generate a JWT token as shown below:

```
@@filename(auth/auth.service)
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { UsersService } from '../users/users.service';
import { JwtService } from '@nestjs/jwt';

@Injectable()
export class AuthService {
```

```
    constructor(
      private usersService: UsersService,
      private jwtService: JwtService
    ) {}

    async signIn(username, pass) {
      const user = await this.usersService.findOne(username);
      if (user?.password !== pass) {
        throw new UnauthorizedException();
      }
      const payload = { sub: user.userId, username: user.username };
      return {
        access_token: await this.jwtService.signAsync(payload),
      };
    }
  }
  @@switch
  import { Injectable, Dependencies, UnauthorizedException } from
  '@nestjs/common';
  import { UsersService } from '../users/users.service';
  import { JwtService } from '@nestjs/jwt';

  @Dependencies(UsersService, JwtService)
  @Injectable()
  export class AuthService {
    constructor(usersService, jwtService) {
      this.usersService = usersService;
      this.jwtService = jwtService;
    }

    async signIn(username, pass) {
      const user = await this.usersService.findOne(username);
      if (user?.password !== pass) {
        throw new UnauthorizedException();
      }
      const payload = { username: user.username, sub: user.userId };
      return {
        access_token: await this.jwtService.signAsync(payload),
      };
    }
  }
```

We're using the `@nestjs/jwt` library, which supplies a `signAsync()` function to generate our JWT from a subset of the `user` object properties, which we then return as a simple object with a single `access_token` property. Note: we choose a property name of `sub` to hold our `userId` value to be consistent with JWT standards. Don't forget to inject the JwtService provider into the `AuthService`.

We now need to update the `AuthModule` to import the new dependencies and configure the `JwtModule`.

First, create `constants.ts` in the `auth` folder, and add the following code:

```
@@filename(auth/constants)
export const jwtConstants = {
  secret: 'DO NOT USE THIS VALUE. INSTEAD, CREATE A COMPLEX SECRET AND
KEEP IT SAFE OUTSIDE OF THE SOURCE CODE.',
};
@@switch
export const jwtConstants = {
  secret: 'DO NOT USE THIS VALUE. INSTEAD, CREATE A COMPLEX SECRET AND
KEEP IT SAFE OUTSIDE OF THE SOURCE CODE.',
};
```

We'll use this to share our key between the JWT signing and verifying steps.

> Warning **Warning Do not expose this key publicly**. We have done so here to make it clear what the code is doing, but in a production system **you must protect this key** using appropriate measures such as a secrets vault, environment variable, or configuration service.

Now, open `auth.module.ts` in the `auth` folder and update it to look like this:

```
@@filename(auth/auth.module)
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UsersModule } from '../users/users.module';
import { JwtModule } from '@nestjs/jwt';
import { AuthController } from './auth.controller';
import { jwtConstants } from './constants';

@Module({
  imports: [
    UsersModule,
    JwtModule.register({
      global: true,
      secret: jwtConstants.secret,
      signOptions: { expiresIn: '60s' },
    }),
  ],
  providers: [AuthService],
  controllers: [AuthController],
  exports: [AuthService],
})
export class AuthModule {}
@@switch
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UsersModule } from '../users/users.module';
import { JwtModule } from '@nestjs/jwt';
import { AuthController } from './auth.controller';
import { jwtConstants } from './constants';

@Module({
  imports: [
```

```
    UsersModule,
    JwtModule.register({
      global: true,
      secret: jwtConstants.secret,
      signOptions: { expiresIn: '60s' },
    }),
  ],
  providers: [AuthService],
  controllers: [AuthController],
  exports: [AuthService],
})
export class AuthModule {}
```

> hint **Hint** We're registering the `JwtModule` as global to make things easier for us. This means that we don't need to import the `JwtModule` anywhere else in our application.

We configure the `JwtModule` using `register()`, passing in a configuration object. See here for more on the Nest `JwtModule` and here for more details on the available configuration options.

Let's go ahead and test our routes using cURL again. You can test with any of the `user` objects hard-coded in the `UsersService`.

```
$ # POST to /auth/login
$ curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
{"access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}
$ # Note: above JWT truncated
```

Implementing the authentication guard

We can now address our final requirement: protecting endpoints by requiring a valid JWT be present on the request. We'll do this by creating an `AuthGuard` that we can use to protect our routes.

```
@@filename(auth/auth.guard)
import {
  CanActivate,
  ExecutionContext,
  Injectable,
  UnauthorizedException,
} from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { jwtConstants } from './constants';
import { Request } from 'express';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
```

```
    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);
    if (!token) {
      throw new UnauthorizedException();
    }
    try {
      const payload = await this.jwtService.verifyAsync(
        token,
        {
          secret: jwtConstants.secret
        }
      );
      // 💡 We're assigning the payload to the request object here
      // so that we can access it in our route handlers
      request['user'] = payload;
    } catch {
      throw new UnauthorizedException();
    }
    return true;
  }

  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    return type === 'Bearer' ? token : undefined;
  }
}
```

We can now implement our protected route and register our `AuthGuard` to protect it.

Open the `auth.controller.ts` file and update it as shown below:

```
@@filename(auth.controller)
import {
  Body,
  Controller,
  Get,
  HttpCode,
  HttpStatus,
  Post,
  Request,
  UseGuards
} from '@nestjs/common';
import { AuthGuard } from './auth.guard';
import { AuthService } from './auth.service';

@Controller('auth')
export class AuthController {
  constructor(private authService: AuthService) {}

  @HttpCode(HttpStatus.OK)
  @Post('login')
  signIn(@Body() signInDto: Record<string, any>) {
```

```
    return this.authService.signIn(signInDto.username,
signInDto.password);
  }

  @UseGuards(AuthGuard)
  @Get('profile')
  getProfile(@Request() req) {
    return req.user;
  }
}
```

We're applying the `AuthGuard` that we just created to the `GET /profile` route so that it will be protected.

Ensure the app is running, and test the routes using `cURL`.

```
$ # GET /profile
$ curl http://localhost:3000/auth/profile
{"statusCode":401,"message":"Unauthorized"}

$ # POST /auth/login
$ curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
{"access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm..."}

$ # GET /profile using access_token returned from previous step as bearer
code
$ curl http://localhost:3000/auth/profile -H "Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm..."
{"sub":1,"username":"john","iat":...,"exp":...}
```

Note that in the `AuthModule`, we configured the JWT to have an expiration of `60 seconds`. This is too short an expiration, and dealing with the details of token expiration and refresh is beyond the scope of this article. However, we chose that to demonstrate an important quality of JWTs. If you wait 60 seconds after authenticating before attempting a `GET /auth/profile` request, you'll receive a `401 Unauthorized` response. This is because `@nestjs/jwt` automatically checks the JWT for its expiration time, saving you the trouble of doing so in your application.

We've now completed our JWT authentication implementation. JavaScript clients (such as Angular/React/Vue), and other JavaScript apps, can now authenticate and communicate securely with our API Server.

Enable authentication globally

If the vast majority of your endpoints should be protected by default, you can register the authentication guard as a global guard and instead of using `@UseGuards()` decorator on top of each controller, you could simply flag which routes should be public.

First, register the `AuthGuard` as a global guard using the following construction (in any module, for example, in the `AuthModule`):

```
providers: [
  {
    provide: APP_GUARD,
    useClass: AuthGuard,
  },
],
```

With this in place, Nest will automatically bind `AuthGuard` to all endpoints.

Now we must provide a mechanism for declaring routes as public. For this, we can create a custom decorator using the `SetMetadata` decorator factory function.

```
import { SetMetadata } from '@nestjs/common';

export const IS_PUBLIC_KEY = 'isPublic';
export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

In the file above, we exported two constants. One being our metadata key named `IS_PUBLIC_KEY`, and the other being our new decorator itself that we're going to call `Public` (you can alternatively name it `SkipAuth` or `AllowAnon`, whatever fits your project).

Now that we have a custom `@Public()` decorator, we can use it to decorate any method, as follows:

```
@Public()
@Get()
findAll() {
  return [];
}
```

Lastly, we need the `AuthGuard` to return `true` when the `"isPublic"` metadata is found. For this, we'll use the `Reflector` class (read more here).

```
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private jwtService: JwtService, private reflector:
Reflector) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const isPublic = this.reflector.getAllAndOverride<boolean>
(IS_PUBLIC_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (isPublic) {
      // 💡 See this condition
      return true;
```

```
    }

    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);
    if (!token) {
      throw new UnauthorizedException();
    }
    try {
      const payload = await this.jwtService.verifyAsync(token, {
        secret: jwtConstants.secret,
      });
      // 💡 We're assigning the payload to the request object here
      // so that we can access it in our route handlers
      request['user'] = payload;
    } catch {
      throw new UnauthorizedException();
    }
    return true;
  }

  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    return type === 'Bearer' ? token : undefined;
  }
}
```

Passport integration

Passport is the most popular node.js authentication library, well-known by the community and successfully used in many production applications. It's straightforward to integrate this library with a **Nest** application using the `@nestjs/passport` module.

To learn how you can integrate Passport with NestJS, check out this chapter.

Example

You can find a complete version of the code in this chapter here.