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.

Authorization

Authorization refers to the process that determines what a user is able to do. For example, an administrative user is allowed to create, edit, and delete posts. A non-administrative user is only authorized to read the posts.

Authorization is orthogonal and independent from authentication. However, authorization requires an authentication mechanism.

There are many different approaches and strategies to handle authorization. The approach taken for any project depends on its particular application requirements. This chapter presents a few approaches to authorization that can be adapted to a variety of different requirements.

Basic RBAC implementation

Role-based access control (**RBAC**) is a policy-neutral access-control mechanism defined around roles and privileges. In this section, we'll demonstrate how to implement a very basic RBAC mechanism using Nest guards.

First, let's create a `Role` enum representing roles in the system:

```
@@filename(role.enum)
export enum Role {
  User = 'user',
  Admin = 'admin',
}
```

> info **Hint** In more sophisticated systems, you may store roles within a database, or pull them from the external authentication provider.

With this in place, we can create a `@Roles()` decorator. This decorator allows specifying what roles are required to access specific resources.

```
@@filename(roles.decorator)
import { SetMetadata } from '@nestjs/common';
import { Role } from '../enums/role.enum';

export const ROLES_KEY = 'roles';
export const Roles = (...roles: Role[]) => SetMetadata(ROLES_KEY, roles);
@@switch
import { SetMetadata } from '@nestjs/common';

export const ROLES_KEY = 'roles';
export const Roles = (...roles) => SetMetadata(ROLES_KEY, roles);
```

Now that we have a custom `@Roles()` decorator, we can use it to decorate any route handler.

```
@@filename(cats.controller)
@Post()
@Roles(Role.Admin)
create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@switch
@Post()
@Roles(Role.Admin)
@Bind(Body())
create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

Finally, we create a `RolesGuard` class which will compare the roles assigned to the current user to the actual roles required by the current route being processed. In order to access the route's role(s) (custom metadata), we'll use the `Reflector` helper class, which is provided out of the box by the framework and exposed from the `@nestjs/core` package.

```
@@filename(roles.guard)
import { Injectable, CanActivate, ExecutionContext } from
'@nestjs/common';
import { Reflector } from '@nestjs/core';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<Role[]>
(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) {
      return true;
    }
    const { user } = context.switchToHttp().getRequest();
    return requiredRoles.some((role) => user.roles?.includes(role));
  }
}
@@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { Reflector } from '@nestjs/core';

@Injectable()
@Dependencies(Reflector)
export class RolesGuard {
  constructor(reflector) {
    this.reflector = reflector;
```

```
  }

  canActivate(context) {
    const requiredRoles = this.reflector.getAllAndOverride(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) {
      return true;
    }
    const { user } = context.switchToHttp().getRequest();
    return requiredRoles.some((role) => user.roles.includes(role));
  }
}
```

> info **Hint** Refer to the Reflection and metadata section of the Execution context chapter for more details on utilizing `Reflector` in a context-sensitive way.

> warning **Notice** This example is named "**basic**" as we only check for the presence of roles on the route handler level. In real-world applications, you may have endpoints/handlers that involve several operations, in which each of them requires a specific set of permissions. In this case, you'll have to provide a mechanism to check roles somewhere within your business-logic, making it somewhat harder to maintain as there will be no centralized place that associates permissions with specific actions.

In this example, we assumed that `request.user` contains the user instance and allowed roles (under the `roles` property). In your app, you will probably make that association in your custom **authentication guard** - see authentication chapter for more details.

To make sure this example works, your `User` class must look as follows:

```
class User {
  // ...other properties
  roles: Role[];
}
```

Lastly, make sure to register the `RolesGuard`, for example, at the controller level, or globally:

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

When a user with insufficient privileges requests an endpoint, Nest automatically returns the following response:

```
{
  "statusCode": 403,
  "message": "Forbidden resource",
  "error": "Forbidden"
}
```

> info **Hint** If you want to return a different error response, you should throw your own specific exception instead of returning a boolean value.

Claims-based authorization

When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is a name-value pair that represents what the subject can do, not what the subject is.

To implement a Claims-based authorization in Nest, you can follow the same steps we have shown above in the RBAC section with one significant difference: instead of checking for specific roles, you should compare **permissions**. Every user would have a set of permissions assigned. Likewise, each resource/endpoint would define what permissions are required (for example, through a dedicated `@RequirePermissions()` decorator) to access them.

```
@@filename(cats.controller)
@Post()
@RequirePermissions(Permission.CREATE_CAT)
create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@switch
@Post()
@RequirePermissions(Permission.CREATE_CAT)
@Bind(Body())
create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

> info **Hint** In the example above, `Permission` (similar to `Role` we have shown in RBAC section) is a TypeScript enum that contains all the permissions available in your system.

Integrating CASL

CASL is an isomorphic authorization library which restricts what resources a given client is allowed to access. It's designed to be incrementally adoptable and can easily scale between a simple claim based and fully featured subject and attribute based authorization.

To start, first install the `@casl/ability` package:

```
$ npm i @casl/ability
```

> info **Hint** In this example, we chose CASL, but you can use any other library like `accesscontrol` or `acl`, depending on your preferences and project needs.

Once the installation is complete, for the sake of illustrating the mechanics of CASL, we'll define two entity classes: `User` and `Article`.

```
class User {
  id: number;
  isAdmin: boolean;
}
```

`User` class consists of two properties, `id`, which is a unique user identifier, and `isAdmin`, indicating whether a user has administrator privileges.

```
class Article {
  id: number;
  isPublished: boolean;
  authorId: number;
}
```

`Article` class has three properties, respectively `id`, `isPublished`, and `authorId`. `id` is a unique article identifier, `isPublished` indicates whether an article was already published or not, and `authorId`, which is an ID of a user who wrote the article.

Now let's review and refine our requirements for this example:

- Admins can manage (create/read/update/delete) all entities
- Users have read-only access to everything
- Users can update their articles (`article.authorId === userId`)
- Articles that are published already cannot be removed (`article.isPublished === true`)

With this in mind, we can start off by creating an `Action` enum representing all possible actions that the users can perform with entities:

```
export enum Action {
  Manage = 'manage',
  Create = 'create',
  Read = 'read',
  Update = 'update',
  Delete = 'delete',
}
```

> warning **Notice** `manage` is a special keyword in CASL which represents "any" action.

To encapsulate CASL library, let's generate the `CaslModule` and `CaslAbilityFactory` now.

```
$ nest g module casl
$ nest g class casl/casl-ability.factory
```

With this in place, we can define the `createForUser()` method on the `CaslAbilityFactory`. This method will create the `Ability` object for a given user:

```
type Subjects = InferSubjects<typeof Article | typeof User> | 'all';

export type AppAbility = Ability<[Action, Subjects]>;

@Injectable()
export class CaslAbilityFactory {
  createForUser(user: User) {
    const { can, cannot, build } = new AbilityBuilder<
      Ability<[Action, Subjects]>
    >(Ability as AbilityClass<AppAbility>);

    if (user.isAdmin) {
      can(Action.Manage, 'all'); // read-write access to everything
    } else {
      can(Action.Read, 'all'); // read-only access to everything
    }

    can(Action.Update, Article, { authorId: user.id });
    cannot(Action.Delete, Article, { isPublished: true });

    return build({
      // Read https://casl.js.org/v5/en/guide/subject-type-detection#use-
classes-as-subject-types for details
      detectSubjectType: (item) =>
        item.constructor as ExtractSubjectType<Subjects>,
    });
  }
}
```

> warning **Notice** `all` is a special keyword in CASL that represents "any subject".

> info **Hint** `Ability`, `AbilityBuilder`, `AbilityClass`, and `ExtractSubjectType` classes are exported from the `@casl/ability` package.

> info **Hint** `detectSubjectType` option let CASL understand how to get subject type out of an object. For more information read CASL documentation for details.

In the example above, we created the `Ability` instance using the `AbilityBuilder` class. As you probably guessed, `can` and `cannot` accept the same arguments but have different meanings, `can` allows to do an action on the specified subject and `cannot` forbids. Both may accept up to 4 arguments. To learn more about these functions, visit the official CASL documentation.

Lastly, make sure to add the `CaslAbilityFactory` to the `providers` and `exports` arrays in the `CaslModule` module definition:

```
import { Module } from '@nestjs/common';
import { CaslAbilityFactory } from './casl-ability.factory';

@Module({
  providers: [CaslAbilityFactory],
  exports: [CaslAbilityFactory],
})
export class CaslModule {}
```

With this in place, we can inject the `CaslAbilityFactory` to any class using standard constructor injection as long as the `CaslModule` is imported in the host context:

```
constructor(private caslAbilityFactory: CaslAbilityFactory) {}
```

Then use it in a class as follows.

```
const ability = this.caslAbilityFactory.createForUser(user);
if (ability.can(Action.Read, 'all')) {
  // "user" has read access to everything
}
```

> info **Hint** Learn more about the `Ability` class in the official CASL documentation.

For example, let's say we have a user who is not an admin. In this case, the user should be able to read articles, but creating new ones or removing the existing articles should be prohibited.

```
const user = new User();
user.isAdmin = false;

const ability = this.caslAbilityFactory.createForUser(user);
ability.can(Action.Read, Article); // true
ability.can(Action.Delete, Article); // false
ability.can(Action.Create, Article); // false
```

> info **Hint** Although both `Ability` and `AbilityBuilder` classes provide `can` and `cannot` methods, they have different purposes and accept slightly different arguments.

Also, as we have specified in our requirements, the user should be able to update its articles:

```

```

```
const user = new User();
user.id = 1;

const article = new Article();
article.authorId = user.id;

const ability = this.caslAbilityFactory.createForUser(user);
ability.can(Action.Update, article); // true

article.authorId = 2;
ability.can(Action.Update, article); // false
```

As you can see, `Ability` instance allows us to check permissions in pretty readable way. Likewise, `AbilityBuilder` allows us to define permissions (and specify various conditions) in a similar fashion. To find more examples, visit the official documentation.

Advanced: Implementing a `PoliciesGuard`

In this section, we'll demonstrate how to build a somewhat more sophisticated guard, which checks if a user meets specific **authorization policies** that can be configured on the method-level (you can extend it to respect policies configured on the class-level too). In this example, we are going to use the CASL package just for illustration purposes, but using this library is not required. Also, we will use the `CaslAbilityFactory` provider that we've created in the previous section.

First, let's flesh out the requirements. The goal is to provide a mechanism that allows specifying policy checks per route handler. We will support both objects and functions (for simpler checks and for those who prefer more functional-style code).

Let's start off by defining interfaces for policy handlers:

```
import { AppAbility } from '../casl/casl-ability.factory';

interface IPolicyHandler {
  handle(ability: AppAbility): boolean;
}

type PolicyHandlerCallback = (ability: AppAbility) => boolean;

export type PolicyHandler = IPolicyHandler | PolicyHandlerCallback;
```

As mentioned above, we provided two possible ways of defining a policy handler, an object (instance of a class that implements the `IPolicyHandler` interface) and a function (which meets the `PolicyHandlerCallback` type).

With this in place, we can create a `@CheckPolicies()` decorator. This decorator allows specifying what policies have to be met to access specific resources.

```

```

```
export const CHECK_POLICIES_KEY = 'check_policy';
export const CheckPolicies = (...handlers: PolicyHandler[]) =>
  SetMetadata(CHECK_POLICIES_KEY, handlers);
```

Now let's create a `PoliciesGuard` that will extract and execute all the policy handlers bound to a route handler.

```
@Injectable()
export class PoliciesGuard implements CanActivate {
  constructor(
    private reflector: Reflector,
    private caslAbilityFactory: CaslAbilityFactory,
  ) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const policyHandlers =
      this.reflector.get<PolicyHandler[]>(
        CHECK_POLICIES_KEY,
        context.getHandler(),
      ) || [];

    const { user } = context.switchToHttp().getRequest();
    const ability = this.caslAbilityFactory.createForUser(user);

    return policyHandlers.every((handler) =>
      this.execPolicyHandler(handler, ability),
    );
  }

  private execPolicyHandler(handler: PolicyHandler, ability: AppAbility) {
    if (typeof handler === 'function') {
      return handler(ability);
    }
    return handler.handle(ability);
  }
}
```

> info **Hint** In this example, we assumed that `request.user` contains the user instance. In your app, you will probably make that association in your custom **authentication guard** - see authentication chapter for more details.

Let's break this example down. The `policyHandlers` is an array of handlers assigned to the method through the `@CheckPolicies()` decorator. Next, we use the `CaslAbilityFactory#create` method which constructs the `Ability` object, allowing us to verify whether a user has sufficient permissions to perform specific actions. We are passing this object to the policy handler which is either a function or an instance of a class that implements the `IPolicyHandler`, exposing the `handle()` method that returns a boolean. Lastly, we use the `Array#every` method to make sure that every handler returned `true` value.

Finally, to test this guard, bind it to any route handler, and register an inline policy handler (functional approach), as follows:

```
@Get()
@UseGuards(PoliciesGuard)
@CheckPolicies((ability: AppAbility) => ability.can(Action.Read, Article))
findAll() {
  return this.articlesService.findAll();
}
```

Alternatively, we can define a class which implements the `IPolicyHandler` interface:

```
export class ReadArticlePolicyHandler implements IPolicyHandler {
  handle(ability: AppAbility) {
    return ability.can(Action.Read, Article);
  }
}
```

And use it as follows:

```
@Get()
@UseGuards(PoliciesGuard)
@CheckPolicies(new ReadArticlePolicyHandler())
findAll() {
  return this.articlesService.findAll();
}
```

> warning **Notice** Since we must instantiate the policy handler in-place using the `new` keyword, `ReadArticlePolicyHandler` class cannot use the Dependency Injection. This can be addressed with the `ModuleRef#get` method (read more here). Basically, instead of registering functions and instances through the `@CheckPolicies()` decorator, you must allow passing a `Type<IPolicyHandler>`. Then, inside your guard, you could retrieve an instance using a type reference: `moduleRef.get(YOUR_HANDLER_TYPE)` or even dynamically instantiate it using the `ModuleRef#create` method.

Encryption and Hashing

Encryption is the process of encoding information. This process converts the original representation of the information, known as plaintext, into an alternative form known as ciphertext. Ideally, only authorized parties can decipher a ciphertext back to plaintext and access the original information. Encryption does not itself prevent interference but denies the intelligible content to a would-be interceptor. Encryption is a two-way function; what is encrypted can be decrypted with the proper key.

Hashing is the process of converting a given key into another value. A hash function is used to generate the new value according to a mathematical algorithm. Once hashing has been done, it should be impossible to go from the output to the input.

Encryption

Node.js provides a built-in crypto module that you can use to encrypt and decrypt strings, numbers, buffers, streams, and more. Nest itself does not provide any additional package on top of this module to avoid introducing unnecessary abstractions.

As an example, let's use AES (Advanced Encryption System) `'aes-256-ctr'` algorithm CTR encryption mode.

```
import { createCipheriv, randomBytes, scrypt } from 'crypto';
import { promisify } from 'util';

const iv = randomBytes(16);
const password = 'Password used to generate key';

// The key length is dependent on the algorithm.
// In this case for aes256, it is 32 bytes.
const key = (await promisify(scrypt)(password, 'salt', 32)) as Buffer;
const cipher = createCipheriv('aes-256-ctr', key, iv);

const textToEncrypt = 'Nest';
const encryptedText = Buffer.concat([
  cipher.update(textToEncrypt),
  cipher.final(),
]);
```

Now to decrypt `encryptedText` value:

```
import { createDecipheriv } from 'crypto';

const decipher = createDecipheriv('aes-256-ctr', key, iv);
const decryptedText = Buffer.concat([
  decipher.update(encryptedText),
  decipher.final(),
]);
```

Hashing

For hashing, we recommend using either the bcrypt or argon2 packages. Nest itself does not provide any additional wrappers on top of these modules to avoid introducing unnecessary abstractions (making the learning curve short).

As an example, let's use `bcrypt` to hash a random password.

First install required packages:

```
$ npm i bcrypt
$ npm i -D @types/bcrypt
```

Once the installation is complete, you can use the `hash` function, as follows:

```
import * as bcrypt from 'bcrypt';

const saltOrRounds = 10;
const password = 'random_password';
const hash = await bcrypt.hash(password, saltOrRounds);
```

To generate a salt, use the `genSalt` function:

```
const salt = await bcrypt.genSalt();
```

To compare/check a password, use the `compare` function:

```
const isMatch = await bcrypt.compare(password, hash);
```

You can read more about available functions here.

Helmet

Helmet can help protect your app from some well-known web vulnerabilities by setting HTTP headers appropriately. Generally, Helmet is just a collection of smaller middleware functions that set security-related HTTP headers (read more).

> info **Hint** Note that applying `helmet` as global or registering it must come before other calls to `app.use()` or setup functions that may call `app.use()`. This is due to the way the underlying platform (i.e., Express or Fastify) works, where the order that middleware/routes are defined matters. If you use middleware like `helmet` or `cors` after you define a route, then that middleware will not apply to that route, it will only apply to routes defined after the middleware.

Use with Express (default)

Start by installing the required package.

```
$ npm i --save helmet
```

Once the installation is complete, apply it as a global middleware.

```
import helmet from 'helmet';
// somewhere in your initialization file
app.use(helmet());
```

> warning **Warning** When using `helmet`, `@apollo/server` (4.x), and the Apollo Sandbox, there may be a problem with CSP on the Apollo Sandbox. To solve this issue configure the CSP as shown below:

```
app.use(helmet({
  crossOriginEmbedderPolicy: false,
  contentSecurityPolicy: {
    directives: {
      imgSrc: [`'self'`, 'data:', 'apollo-server-landing-
page.cdn.apollographql.com'],
      scriptSrc: [`'self'`, `https: 'unsafe-inline'`],
      manifestSrc: [`'self'`, 'apollo-server-landing-
page.cdn.apollographql.com'],
      frameSrc: [`'self'`, 'sandbox.embed.apollographql.com'],
    },
  },
}));
```

Use with Fastify

If you are using the `FastifyAdapter`, install the @fastify/helmet package:

```
$ npm i --save @fastify/helmet
```

fastify-helmet should not be used as a middleware, but as a Fastify plugin, i.e., by using `app.register()`:

```
import helmet from '@fastify/helmet'
// somewhere in your initialization file
await app.register(helmet)
```

warning **Warning** When using `apollo-server-fastify` and `@fastify/helmet`, there may be a problem with CSP on the GraphQL playground, to solve this collision, configure the CSP as shown below:

```
await app.register(fastifyHelmet, {
   contentSecurityPolicy: {
     directives: {
       defaultSrc: [`'self'`, 'unpkg.com'],
       styleSrc: [
         `'self'`,
         `'unsafe-inline'`,
         'cdn.jsdelivr.net',
         'fonts.googleapis.com',
         'unpkg.com',
       ],
       fontSrc: [`'self'`, 'fonts.gstatic.com', 'data:'],
       imgSrc: [`'self'`, 'data:', 'cdn.jsdelivr.net'],
       scriptSrc: [
         `'self'`,
         `https: 'unsafe-inline'`,
         `cdn.jsdelivr.net`,
         `'unsafe-eval'`,
       ],
     },
   },
 });

// If you are not going to use CSP at all, you can use this:
await app.register(fastifyHelmet, {
  contentSecurityPolicy: false,
});
```

CORS

Cross-origin resource sharing (CORS) is a mechanism that allows resources to be requested from another domain. Under the hood, Nest makes use of the Express cors package. This package provides various options that you can customize based on your requirements.

Getting started

To enable CORS, call the `enableCors()` method on the Nest application object.

```
const app = await NestFactory.create(AppModule);
app.enableCors();
await app.listen(3000);
```

The `enableCors()` method takes an optional configuration object argument. The available properties of this object are described in the official CORS documentation. Another way is to pass a callback function that lets you define the configuration object asynchronously based on the request (on the fly).

Alternatively, enable CORS via the `create()` method's options object. Set the `cors` property to `true` to enable CORS with default settings. Or, pass a CORS configuration object or callback function as the `cors` property value to customize its behavior.

```
const app = await NestFactory.create(AppModule, { cors: true });
await app.listen(3000);
```

CSRF Protection

Cross-site request forgery (also known as CSRF or XSRF) is a type of malicious exploit of a website where **unauthorized** commands are transmitted from a user that the web application trusts. To mitigate this kind of attack you can use the csurf package.

Use with Express (default)

Start by installing the required package:

```
$ npm i --save csurf
```

> warning **Warning** This package is deprecated, refer to `csurf` docs for more information.

> warning **Warning** As explained in the `csurf` docs, this middleware requires either session middleware or `cookie-parser` to be initialized first. Please see that documentation for further instructions.

Once the installation is complete, apply the `csurf` middleware as global middleware.

```
import * as csurf from 'csurf';
// ...
// somewhere in your initialization file
app.use(csurf());
```

Use with Fastify

Start by installing the required package:

```
$ npm i --save @fastify/csrf-protection
```

Once the installation is complete, register the `@fastify/csrf-protection` plugin, as follows:

```
import fastifyCsrf from '@fastify/csrf-protection';
// ...
// somewhere in your initialization file after registering some storage
plugin
await app.register(fastifyCsrf);
```

> warning **Warning** As explained in the `@fastify/csrf-protection` docs here, this plugin requires a storage plugin to be initialized first. Please, see that documentation for further instructions.

Rate Limiting

A common technique to protect applications from brute-force attacks is **rate-limiting**. To get started, you'll need to install the `@nestjs/throttler` package.

```
$ npm i --save @nestjs/throttler
```

Once the installation is complete, the `ThrottlerModule` can be configured as any other Nest package with `forRoot` or `forRootAsync` methods.

```
@@filename(app.module)
@Module({
  imports: [
    ThrottlerModule.forRoot({
      ttl: 60,
      limit: 10,
    }),
  ],
})
export class AppModule {}
```

The above will set the global options for the `ttl`, the time to live, and the `limit`, the maximum number of requests within the ttl, for the routes of your application that are guarded.

Once the module has been imported, you can then choose how you would like to bind the `ThrottlerGuard`. Any kind of binding as mentioned in the guards section is fine. If you wanted to bind the guard globally, for example, you could do so by adding this provider to any module:

```
{
  provide: APP_GUARD,
  useClass: ThrottlerGuard
}
```

Customization

There may be a time where you want to bind the guard to a controller or globally, but want to disable rate limiting for one or more of your endpoints. For that, you can use the `@SkipThrottle()` decorator, to negate the throttler for an entire class or a single route. The `@SkipThrottle()` decorator can also take in a boolean for if there is a case where you want to exclude *most* of a controller, but not every route.

```
@SkipThrottle()
@Controller('users')
export class UsersController {}
```

This `@SkipThrottle()` decorator can be used to skip a route or a class or to negate the skipping of a route in a class that is skipped.

```
@SkipThrottle()
@Controller('users')
export class UsersController {
  // Rate limiting is applied to this route.
  @SkipThrottle(false)
  dontSkip() {
    return "List users work with Rate limiting.";
  }
  // This route will skip rate limiting.
  doSkip() {
    return "List users work without Rate limiting.";
  }
}
```

There is also the `@Throttle()` decorator which can be used to override the `limit` and `ttl` set in the global module, to give tighter or looser security options. This decorator can be used on a class or a function as well. The order for this decorator does matter, as the arguments are in the order of `limit, ttl`. You have to configure it like this:

```
// Override default configuration for Rate limiting and duration.
@Throttle(3, 60)
@Get()
findAll() {
  return "List users works with custom rate limiting.";
}
```

Proxies

If your application runs behind a proxy server, check the specific HTTP adapter options (express and fastify) for the `trust proxy` option and enable it. Doing so will allow you to get the original IP address from the `X-Forwarded-For` header, and you can override the `getTracker()` method to pull the value from the header rather than from `req.ip`. The following example works with both express and fastify:

```
// throttler-behind-proxy.guard.ts
import { ThrottlerGuard } from '@nestjs/throttler';
import { Injectable } from '@nestjs/common';

@Injectable()
export class ThrottlerBehindProxyGuard extends ThrottlerGuard {
  protected getTracker(req: Record<string, any>): string {
    return req.ips.length ? req.ips[0] : req.ip; // individualize IP
extraction to meet your own needs
  }
```

```
}

// app.controller.ts
import { ThrottlerBehindProxyGuard } from './throttler-behind-
proxy.guard';

@UseGuards(ThrottlerBehindProxyGuard)
```

> info **Hint** You can find the API of the `req` Request object for express here and for fastify here.

Websockets

This module can work with websockets, but it requires some class extension. You can extend the `ThrottlerGuard` and override the `handleRequest` method like so:

```
@Injectable()
export class WsThrottlerGuard extends ThrottlerGuard {
  async handleRequest(context: ExecutionContext, limit: number, ttl:
number): Promise<boolean> {
    const client = context.switchToWs().getClient();
    const ip = client._socket.remoteAddress
    const key = this.generateKey(context, ip);
    const { totalHits } = await this.storageService.increment(key, ttl);

    if (totalHits > limit) {
      throw new ThrottlerException();
    }

    return true;
  }
}
```

> info **Hint** If you are using ws, it is necessary to replace the `_socket` with `conn`

There's a few things to keep in mind when working with WebSockets:

- Guard cannot be registered with the `APP_GUARD` or `app.useGlobalGuards()`
- When a limit is reached, Nest will emit an `exception` event, so make sure there is a listener ready for this

> info **Hint** If you are using the `@nestjs/platform-ws` package you can use `client._socket.remoteAddress` instead.

GraphQL

The `ThrottlerGuard` can also be used to work with GraphQL requests. Again, the guard can be extended, but this time the `getRequestResponse` method will be overridden

```
@Injectable()
export class GqlThrottlerGuard extends ThrottlerGuard {
  getRequestResponse(context: ExecutionContext) {
    const gqlCtx = GqlExecutionContext.create(context);
    const ctx = gqlCtx.getContext();
    return { req: ctx.req, res: ctx.res };
  }
}
```

Configuration

The following options are valid for the `ThrottlerModule`:

`ttl`	the number of seconds that each request will last in storage
`limit`	the maximum number of requests within the TTL limit
`ignoreUserAgents`	an array of regular expressions of user-agents to ignore when it comes to throttling requests
`storage`	the storage setting for how to keep track of the requests

Async Configuration

You may want to get your rate-limiting configuration asynchronously instead of synchronously. You can use the `forRootAsync()` method, which allows for dependency injection and `async` methods.

One approach would be to use a factory function:

```
@Module({
  imports: [
    ThrottlerModule.forRootAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: (config: ConfigService) => ({
        ttl: config.get('THROTTLE_TTL'),
        limit: config.get('THROTTLE_LIMIT'),
      }),
    }),
  ],
})
export class AppModule {}
```

You can also use the `useClass` syntax:

```
@Module({
  imports: [
    ThrottlerModule.forRootAsync({
```

```
        imports: [ConfigModule],
        useClass: ThrottlerConfigService,
      }),
    ],
  })
  export class AppModule {}
```

This is doable, as long as `ThrottlerConfigService` implements the interface `ThrottlerOptionsFactory`.

Storages

The built in storage is an in memory cache that keeps track of the requests made until they have passed the TTL set by the global options. You can drop in your own storage option to the `storage` option of the `ThrottlerModule` so long as the class implements the `ThrottlerStorage` interface.

For distributed servers you could use the community storage provider for Redis to have a single source of truth.

> info **Note** `ThrottlerStorage` can be imported from `@nestjs/throttler`.