Authentication

Authentication is an **essential** part of most existing applications. There are a lot of different approaches, strategies, and ways to handle user authorization. What we eventually decide to use depends on the particular application requirements and is strongly associated with their needs.

Passport is the most popular node.js authentication library, well-known by community and successively used in many production applications. It's really simple to integrate this tool with **Nest** framework using dedicated passport utilities. For demonstration purposes, we'll set up both **passport-http-bearer** and **passport-jwt** strategy.

Installation

In order to start the adventure with this library, we have to install a few fundamental packages. Additionally, we'll start by implementing the bearer strategy, and thus we need to install `passport-http-bearer` package.

```
$ npm install --save @nestjs/passport passport passport-http-bearer
```

Bearer strategy

As has been said already, firstly, we'll implement **passport-http-bearer** library. Bearer tokens are typically used to protect API endpoints, and are often issued using OAuth 2.0. The HTTP Bearer authentication strategy authenticates users using a bearer token.

Let's start by creating the `AuthService` class that will expose a single method, `validateUser()` which responsibility is to query user using provided bearer **token**.

auth.service.ts JS

```
import { Injectable } from '@nestjs/common';
import { UsersService } from '../users/users.service';

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

  async validateUser(token: string): Promise<any> {
    // Validate if token passed along with HTTP request
    // is associated with any registered account in the database
    return await this.usersService.findOneByToken(token);
  }
}
```

The `validateUser()` method takes `token` as an argument. This token is extracted from `Authorization` header that has been passed along with HTTP request. The `findOneByToken()` method's responsibility is to validate if passed token truly exists and is associated with any registered account in the database.

Once `AuthService` class is done, we have to create a corresponding **strategy** that passport will use to authenticate requests.

http.strategy.ts JS

```
import { Strategy } from 'passport-http-bearer';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { AuthService } from './auth.service';

@Injectable()
export class HttpStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly authService: AuthService) {
    super();
  }

  async validate(token: string) {
    const user = await this.authService.validateUser(token);
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
}
```

The `HttpStrategy` uses `AuthService` to validate the token. When the token is valid, passport allows further request processing. Otherwise, the user receives `401 Unauthorized` response.

Afterwards, we can create the `AuthModule`.

auth.module.ts JS

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { HttpStrategy } from './http.strategy';
import { UsersModule } from '../users/users.module';

@Module({
```

```
  imports: [UsersModule],
  providers: [AuthService, HttpStrategy],
})
export class AuthModule {}
```

> **NOTICE**
> In order to make use of `UsersService`, the `AuthModule` imports `UsersModule`. The internal implementation is unimportant here and heavily depends on your technical project requirements (e.g. database).

Then, you can simply use the `AuthGuard` wherever you want to enable the authentication.

```
@Get('users')
@UseGuards(AuthGuard('bearer'))
findAll() {
  return [];
}
```

The `@AuthGuard()` is imported from `@nestjs/passport` package. Also, `bearer` is a name of the strategy that passport will make use of. Let us check if endpoint is effectively secured. To ensure that everything work correctly, we'll perform a GET request to the `users` resource without setting a valid token.

```
$ curl localhost:3000/users
```

Application should respond with `401 Unauthorized` status code and following response body:

```
"statusCode": 401,
"error": "Unauthorized"
```

If you create a valid token beforehand and pass it along with the HTTP request, the application will respectively identify a user, attach its object to the request, and allow further request processing.

```
$ curl localhost:3000/users -H "Authorization: Bearer TOKEN"
```

Default strategy

Default strategy

To determine default passport behavior, you can register the `PassportModule`.

auth.module.ts JS

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { HttpStrategy } from './http.strategy';
import { UsersModule } from '../users/users.module';
import { PassportModule } from '@nestjs/passport';

@Module({
  imports: [
    PassportModule.register({ defaultStrategy: 'bearer' }),
    UsersModule,
  ],
  providers: [AuthService, HttpStrategy],
  exports: [PassportModule, AuthService]
})
export class AuthModule {}
```

Once you set `defaultStrategy`, you no longer need to manually pass the strategy name in the `@AuthGuard()` decorator.

```
@Get('users')
@UseGuards(AuthGuard())
findAll() {
  return [];
}
```

> **NOTICE**
> Keep in mind that either `PassportModule` or `AuthModule` has to be imported by every module that makes use of the `AuthGuard`.

User object

When request is validated correctly, the user entity will be attached to the request object and accessible through `user` property (e.g. `req.user`). To change the property name, set `property` of the options object.

```
PassportModule.register({ property: 'profile' });
```

Customize passport

Depending on the strategy that is being used, passport takes a bunch of properties that shape the library behavior. Use `register()` method to pass down options object directly to the passport instance.

```
PassportModule.register({ session: true });
```

Inheritance

In most cases, `AuthGuard` will be sufficient. However, in order to adjust either default error handling or authentication logic, you can extend the class and override methods within a subclass.

```
import {
  ExecutionContext,
  Injectable,
  UnauthorizedException,
} from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  canActivate(context: ExecutionContext) {
    // Add your custom authentication logic here
    // for example, call super.logIn(request) to establish a session.
    return super.canActivate(context);
  }

  handleRequest(err, user, info) {
    if (err || !user) {
      throw err || new UnauthorizedException();
    }
    return user;
  }
}
```

> **HINT**
> In order to use your custom `JwtAuthGuard`, you must add it as a guard to your specific routes (e.g., `@UseGuards(JwtAuthGuard)`)

JWT strategy

A second described approach is to authenticate endpoints using a **JSON web token** (JWT). To implement a JWT-based authentication flow, we need to install required packages.

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

Once the installation process is done, we can focus on `AuthService` class. We need to switch from the token validation to a payload-based validation logic as well as provide a way to create a JWT token for the particular user which then could be used to authenticate the incoming request.

auth.service.ts JS

```
import { JwtService } from '@nestjs/jwt';
import { Injectable } from '@nestjs/common';
import { UsersService } from '../users/users.service';
import { JwtPayload } from './interfaces/jwt-payload.interface';

@Injectable()
export class AuthService {
  constructor(
    private readonly usersService: UsersService,
    private readonly jwtService: JwtService,
  ) {}

  async signIn(): Promise<string> {
    // In the real-world app you shouldn't expose this method publicly
    // instead, return a token once you verify user credentials
    const user: JwtPayload = { email: 'user@email.com' };
    return this.jwtService.sign(user);
  }

  async validateUser(payload: JwtPayload): Promise<any> {
    return await this.usersService.findOneByEmail(payload.email);
  }
}
```

> **HINT**
> The `JwtPayload` is an interface with a single property, an `email`, and represents decoded JWT token.

In order to simplify an example, we created a fake user. The second step is to create a corresponding `JwtStrategy`.

jwt.strategy.ts JS

```
import { ExtractJwt, Strategy } from 'passport-jwt';
import { AuthService } from './auth.service';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { JwtPayload } from './interfaces/jwt-payload.interface';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly authService: AuthService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: 'secretKey',
    });
  }

  async validate(payload: JwtPayload) {
    const user = await this.authService.validateUser(payload);
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
}
```

The `JwtStrategy` uses `AuthService` to validate the decoded payload. When the payload is valid (user exists), passport allows further request processing. Otherwise, the user receives `401 (Unauthorized)` response.

Afterward, we can move to the `AuthModule`.

auth.module.ts JS

```
import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { AuthService } from './auth.service';
import { JwtStrategy } from './jwt.strategy';
import { UsersModule } from '../users/users.module';
import { PassportModule } from '@nestjs/passport';

@Module({
  imports: [
    PassportModule.register({ defaultStrategy: 'jwt' }),
```

```
    JwtModule.register({
      secretOrPrivateKey: 'secretKey',
      signOptions: {
        expiresIn: 3600,
      },
    }),
    UsersModule,
  ],
  providers: [AuthService, JwtStrategy],
  exports: [PassportModule, AuthService],
})
export class AuthModule {}
```

> **HINT**
> In order to make use of `UsersService`, the `AuthModule` imports `UsersModule`. The internal implementation is unimportant here. Besides, `JwtModule` has been registered statically. To switch to asynchronous configuration, read more **here**.

Both expiration time and `secretKey` are hardcoded (in a real-world application you should rather consider using environment variables).

Then, you can simply use the `AuthGuard` wherever you want to enable the authentication.

```
@Get('users')
@UseGuards(AuthGuard())
findAll() {
  return [];
}
```

Let us check if endpoint is effectively secured. To ensure that everything work correctly, we'll perform a GET request to the `users` resource without setting a valid token.

```
$ curl localhost:3000/users
```

Application should respond with `401 Unauthorized` status code and following response body:

```
"statusCode": 401,
"error": "Unauthorized"
```

If you create a valid token beforehand and pass it along with the HTTP request, the application will respectively identify a user, attach its object to the request, and allow further request processing.

```
$ curl localhost:3000/users -H "Authorization: Bearer TOKEN"
```

Example

A full working example is available **here**.

Multiple strategies

Usually, you'll end up with single strategy reused across the whole application. However, there might be cases when you'd prefer to use different strategies for different scopes. In the case of multiple strategies, pass the second argument to the `PassportStrategy` function. Generally, this argument is a name of the strategy.

```
export class JwtStrategy extends PassportStrategy(Strategy, 'jwt')
```

In above example, the `jwt` becomes the name of the `JwtStrategy`. Afterward, you can use `@AuthGuard('jwt')`, just the same as before.

GraphQL

In order to use `AuthGuard` together with **GraphQL**, you have to extend the built-in `AuthGuard` class and override `getRequest()` method.

```
@Injectable()
export class GqlAuthGuard extends AuthGuard('jwt') {
  getRequest(context: ExecutionContext) {
    const ctx = GqlExecutionContext.create(context);
    return ctx.getContext().req;
  }
}
```

We assumed that `req` (request) has been passed as a part of the context value. We have to set this behavior in the module settings.

```
GraphQLModule.forRoot({
  context: ({ req }) => ({ req }),
});
```

And now, context value will have `req` property.

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more **here**.

Principal Sponsor

Sponsors / Partners