Passport (authentication)

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. At a high level, Passport executes a series of steps to:

- Authenticate a user by verifying their "credentials" (such as username/password, JSON Web Token (JWT), or identity token from an Identity Provider)
- Manage authenticated state (by issuing a portable token, such as a JWT, or creating an Express session)
- Attach information about the authenticated user to the `Request` object for further use in route handlers

Passport has a rich ecosystem of strategies that implement various authentication mechanisms. While simple in concept, the set of Passport strategies you can choose from is large and presents a lot of variety. Passport abstracts these varied steps into a standard pattern, and the `@nestjs/passport` module wraps and standardizes this pattern into familiar Nest constructs.

In this chapter, we'll implement a complete end-to-end authentication solution for a RESTful API server using these powerful and flexible modules. You can use the concepts described here to implement any Passport strategy to customize your authentication scheme. You can follow the steps in this chapter to build this complete example.

Authentication 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.

First we need to install the required packages. Passport provides a strategy called passport-local that implements a username/password authentication mechanism, which suits our needs for this portion of our use case.

```
$ npm install --save @nestjs/passport passport passport-local
$ npm install --save-dev @types/passport-local
```

> warning **Notice** For **any** Passport strategy you choose, you'll always need the `@nestjs/passport` and `passport` packages. Then, you'll need to install the strategy-specific package (e.g., `passport-jwt` or `passport-local`) that implements the particular authentication strategy you are building. In addition, you can also install the type definitions for any Passport strategy, as shown above with `@types/passport-local`, which provides assistance while writing TypeScript code.

Implementing Passport strategies

We're now ready to implement the authentication feature. We'll start with an overview of the process used for **any** Passport strategy. It's helpful to think of Passport as a mini framework in itself. The elegance of the framework is that it abstracts the authentication process into a few basic steps that you customize based on the strategy you're implementing. It's like a framework because you configure it by supplying customization parameters (as plain JSON objects) and custom code in the form of callback functions, which Passport calls at the appropriate time. The `@nestjs/passport` module wraps this framework in a Nest style package, making it easy to integrate into a Nest application. We'll use `@nestjs/passport` below, but first let's consider how **vanilla Passport** works.

In vanilla Passport, you configure a strategy by providing two things:

1. A set of options that are specific to that strategy. For example, in a JWT strategy, you might provide a secret to sign tokens.
2. A "verify callback", which is where you tell Passport how to interact with your user store (where you manage user accounts). Here, you verify whether a user exists (and/or create a new user), and whether their credentials are valid. The Passport library expects this callback to return a full user if the validation succeeds, or a null if it fails (failure is defined as either the user is not found, or, in the case of passport-local, the password does not match).

With `@nestjs/passport`, you configure a Passport strategy by extending the `PassportStrategy` class. You pass the strategy options (item 1 above) by calling the `super()` method in your subclass, optionally passing in an options object. You provide the verify callback (item 2 above) by implementing a `validate()` method in your subclass.

We'll start by generating an `AuthModule` and in it, an `AuthService`:

```
$ nest g module 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 {}
```

Our `AuthService` has the job of retrieving a user and verifying the password. We create a `validateUser()` 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. We'll be calling into the `validateUser()` method from our Passport local strategy in a moment.

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

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

  async validateUser(username: string, pass: string): Promise<any> {
    const user = await this.usersService.findOne(username);
    if (user && user.password === pass) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }
}
@@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { UsersService } from '../users/users.service';

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

  async validateUser(username, pass) {
    const user = await this.usersService.findOne(username);
    if (user && user.password === pass) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }
}
```

> 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 { UsersModule } from '../users/users.module';

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

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

Implementing Passport local

Now we can implement our Passport **local authentication strategy**. Create a file called `local.strategy.ts` in the `auth` folder, and add the following code:

```
@@filename(auth/local.strategy)
import { Strategy } from 'passport-local';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { AuthService } from './auth.service';
```

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

  async validate(username: string, password: string): Promise<any> {
    const user = await this.authService.validateUser(username, password);
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
}
@@switch
import { Strategy } from 'passport-local';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException, Dependencies } from
'@nestjs/common';
import { AuthService } from './auth.service';

@Injectable()
@Dependencies(AuthService)
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(authService) {
    super();
    this.authService = authService;
  }

  async validate(username, password) {
    const user = await this.authService.validateUser(username, password);
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
}
```

We've followed the recipe described earlier for all Passport strategies. In our use case with passport-local, there are no configuration options, so our constructor simply calls `super()`, without an options object.

> info **Hint** We can pass an options object in the call to `super()` to customize the behavior of the passport strategy. In this example, the passport-local strategy by default expects properties called `username` and `password` in the request body. Pass an options object to specify different property names, for example: `super({{ '{' }} usernameField: 'email' {{ '}' }})`. See the Passport documentation for more information.

We've also implemented the `validate()` method. For each strategy, Passport will call the verify function (implemented with the `validate()` method in `@nestjs/passport`) using an appropriate strategy-specific set of parameters. For the local-strategy, Passport expects a `validate()` method with the following signature: `validate(username: string, password:string): any`.

Most of the validation work is done in our `AuthService` (with the help of our `UsersService`), so this method is quite straightforward. The `validate()` method for **any** Passport strategy will follow a similar pattern, varying only in the details of how credentials are represented. If a user is found and the credentials are valid, the user is returned so Passport can complete its tasks (e.g., creating the `user` property on the `Request` object), and the request handling pipeline can continue. If it's not found, we throw an exception and let our exceptions layer handle it.

Typically, the only significant difference in the `validate()` method for each strategy is **how** you determine if a user exists and is valid. For example, in a JWT strategy, depending on requirements, we may evaluate whether the `userId` carried in the decoded token matches a record in our user database, or matches a list of revoked tokens. Hence, this pattern of sub-classing and implementing strategy-specific validation is consistent, elegant and extensible.

We need to configure our `AuthModule` to use the Passport features we just defined. Update `auth.module.ts` 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 { PassportModule } from '@nestjs/passport';
import { LocalStrategy } from './local.strategy';

@Module({
  imports: [UsersModule, PassportModule],
  providers: [AuthService, LocalStrategy],
})
export class AuthModule {}
@@switch
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UsersModule } from '../users/users.module';
import { PassportModule } from '@nestjs/passport';
import { LocalStrategy } from './local.strategy';

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

Built-in Passport Guards

The Guards chapter describes the primary function of Guards: to determine whether a request will be handled by the route handler or not. That remains true, and we'll use that standard capability soon. However, in the context of using the `@nestjs/passport` module, we will also introduce a slight new wrinkle that may at first be confusing, so let's discuss that now. Consider that your app can exist in two states, from an authentication perspective:

1. the user/client is **not** logged in (is not authenticated)
2. the user/client **is** logged in (is authenticated)

In the first case (user is not logged in), we need to perform two distinct functions:

- Restrict the routes an unauthenticated user can access (i.e., deny access to restricted routes). We'll use Guards in their familiar capacity to handle this function, by placing a Guard on the protected routes. As you may anticipate, we'll be checking for the presence of a valid JWT in this Guard, so we'll work on this Guard later, once we are successfully issuing JWTs.

- Initiate the **authentication step** itself when a previously unauthenticated user attempts to login. This is the step where we'll **issue** a JWT to a valid user. Thinking about this for a moment, we know we'll need to `POST` username/password credentials to initiate authentication, so we'll set up a `POST /auth/login` route to handle that. This raises the question: how exactly do we invoke the passport-local strategy in that route?

The answer is straightforward: by using another, slightly different type of Guard. The `@nestjs/passport` module provides us with a built-in Guard that does this for us. This Guard invokes the Passport strategy and kicks off the steps described above (retrieving credentials, running the verify function, creating the `user` property, etc).

The second case enumerated above (logged in user) simply relies on the standard type of Guard we already discussed to enable access to protected routes for logged in users.

Login route

With the strategy in place, we can now implement a bare-bones `/auth/login` route, and apply the built-in Guard to initiate the passport-local flow.

Open the `app.controller.ts` file and replace its contents with the following:

```
@@filename(app.controller)
import { Controller, Request, Post, UseGuards } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Controller()
export class AppController {
  @UseGuards(AuthGuard('local'))
  @Post('auth/login')
  async login(@Request() req) {
    return req.user;
  }
}
@@switch
import { Controller, Bind, Request, Post, UseGuards } from
'@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Controller()
export class AppController {
  @UseGuards(AuthGuard('local'))
  @Post('auth/login')
  @Bind(Request())
  async login(req) {
    return req.user;
  }
}
```

With `@UseGuards(AuthGuard('local'))` we are using an `AuthGuard` that `@nestjs/passport` **automatically provisioned** for us when we extended the passport-local strategy. Let's break that down. Our Passport local strategy has a default name of `'local'`. We reference that name in the `@UseGuards()` decorator to associate it with code supplied by the `passport-local` package. This is used to disambiguate which strategy to invoke in case we have multiple Passport strategies in our app (each of which may provision a strategy-specific `AuthGuard`). While we only have one such strategy so far, we'll shortly add a second, so this is needed for disambiguation.

In order to test our route we'll have our `/auth/login` route simply return the user for now. This also lets us demonstrate another Passport feature: Passport automatically creates a `user` object, based on the value we return from the `validate()` method, and assigns it to the `Request` object as `req.user`. Later, we'll replace this with code to create and return a JWT instead.

Since these are API routes, we'll test them using the commonly available cURL library. 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"
$ # result -> {"userId":1,"username":"john"}
```

While this works, passing the strategy name directly to the `AuthGuard()` introduces magic strings in the codebase. Instead, we recommend creating your own class, as shown below:

```
@@filename(auth/local-auth.guard)
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}
```

Now, we can update the `/auth/login` route handler and use the `LocalAuthGuard` instead:

```
@UseGuards(LocalAuthGuard)
@Post('auth/login')
async login(@Request() req) {
  return req.user;
}
```

JWT functionality

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 a couple more packages to support our JWT requirements:

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

The `@nestjs/jwt` package (see more here) is a utility package that helps with JWT manipulation. The `passport-jwt` package is the Passport package that implements the JWT strategy and `@types/passport-jwt` provides the TypeScript type definitions.

Let's take a closer look at how a `POST /auth/login` request is handled. We've decorated the route using the built-in `AuthGuard` provided by the passport-local strategy. This means that:

1. The route handler **will only be invoked if the user has been validated**
2. The `req` parameter will contain a `user` property (populated by Passport during the passport-local authentication flow)

With this in mind, we can now finally generate a real JWT, and return it in this route. 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, and add the `login()` method, and import the `JwtService` as shown:

```
@@filename(auth/auth.service)
import { Injectable } 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 validateUser(username: string, pass: string): Promise<any> {
    const user = await this.usersService.findOne(username);
    if (user && user.password === pass) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }

  async login(user: any) {
    const payload = { username: user.username, sub: user.userId };
    return {
      access_token: this.jwtService.sign(payload),
    };
  }
}

@@switch
import { Injectable, Dependencies } 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 validateUser(username, pass) {
    const user = await this.usersService.findOne(username);
    if (user && user.password === pass) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }

  async login(user) {
    const payload = { username: user.username, sub: user.userId };
    return {
      access_token: this.jwtService.sign(payload),
    };
  }
}
```

We're using the `@nestjs/jwt` library, which supplies a `sign()` 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 { LocalStrategy } from './local.strategy';
import { UsersModule } from '../users/users.module';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';
import { jwtConstants } from './constants';

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

@Module({
  imports: [
    UsersModule,
    PassportModule,
    JwtModule.register({
      secret: jwtConstants.secret,
      signOptions: { expiresIn: '60s' },
    }),
  ],
  providers: [AuthService, LocalStrategy],
  exports: [AuthService],
})
export class AuthModule {}
```

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.

Now we can update the `/auth/login` route to return a JWT.

```
@@filename(app.controller)
import { Controller, Request, Post, UseGuards } from '@nestjs/common';
import { LocalAuthGuard } from './auth/local-auth.guard';
import { AuthService } from './auth/auth.service';

@Controller()
export class AppController {
  constructor(private authService: AuthService) {}

  @UseGuards(LocalAuthGuard)
  @Post('auth/login')
  async login(@Request() req) {
    return this.authService.login(req.user);
  }
}
@@switch
import { Controller, Bind, Request, Post, UseGuards } from
'@nestjs/common';
import { LocalAuthGuard } from './auth/local-auth.guard';
import { AuthService } from './auth/auth.service';

@Controller()
export class AppController {
  constructor(private authService: AuthService) {}

  @UseGuards(LocalAuthGuard)
  @Post('auth/login')
  @Bind(Request())
  async login(req) {
    return this.authService.login(req.user);
  }
}
```

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"
$ # result -> {"access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}
$ # Note: above JWT truncated
```

Implementing Passport JWT

We can now address our final requirement: protecting endpoints by requiring a valid JWT be present on the request. Passport can help us here too. It provides the passport-jwt strategy for securing RESTful endpoints with JSON Web Tokens. Start by creating a file called `jwt.strategy.ts` in the `auth` folder, and add the following code:

```
@@filename(auth/jwt.strategy)
import { ExtractJwt, Strategy } from 'passport-jwt';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable } from '@nestjs/common';
import { jwtConstants } from './constants';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: jwtConstants.secret,
    });
  }

  async validate(payload: any) {
    return { userId: payload.sub, username: payload.username };
  }
}
@@switch
import { ExtractJwt, Strategy } from 'passport-jwt';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable } from '@nestjs/common';
import { jwtConstants } from './constants';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: jwtConstants.secret,
    });
  }

  async validate(payload) {
    return { userId: payload.sub, username: payload.username };
  }
}
```

With our `JwtStrategy`, we've followed the same recipe described earlier for all Passport strategies. This strategy requires some initialization, so we do that by passing in an options object in the `super()` call. You can read more about the available options here. In our case, these options are:

- `jwtFromRequest`: supplies the method by which the JWT will be extracted from the `Request`. We will use the standard approach of supplying a bearer token in the Authorization header of our API requests. Other options are described here.
- `ignoreExpiration`: just to be explicit, we choose the default `false` setting, which delegates the responsibility of ensuring that a JWT has not expired to the Passport module. This means that if our route is supplied with an expired JWT, the request will be denied and a `401 Unauthorized` response sent. Passport conveniently handles this automatically for us.
- `secretOrKey`: we are using the expedient option of supplying a symmetric secret for signing the token. Other options, such as a PEM-encoded public key, may be more appropriate for production apps (see here for more information). In any case, as cautioned earlier, **do not expose this secret publicly**.

The `validate()` method deserves some discussion. For the jwt-strategy, Passport first verifies the JWT's signature and decodes the JSON. It then invokes our `validate()` method passing the decoded JSON as its single parameter. Based on the way JWT signing works, **we're guaranteed that we're receiving a valid token** that we have previously signed and issued to a valid user.

As a result of all this, our response to the `validate()` callback is trivial: we simply return an object containing the `userId` and `username` properties. Recall again that Passport will build a `user` object based on the return value of our `validate()` method, and attach it as a property on the `Request` object.

It's also worth pointing out that this approach leaves us room ('hooks' as it were) to inject other business logic into the process. For example, we could do a database lookup in our `validate()` method to extract more information about the user, resulting in a more enriched `user` object being available in our `Request`. This is also the place we may decide to do further token validation, such as looking up the `userId` in a list of revoked tokens, enabling us to perform token revocation. The model we've implemented here in our sample code is a fast, "stateless JWT" model, where each API call is immediately authorized based on the presence of a valid JWT, and a small bit of information about the requester (its `userId` and `username`) is available in our Request pipeline.

Add the new `JwtStrategy` as a provider in the `AuthModule`:

```
@@filename(auth/auth.module)
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { LocalStrategy } from './local.strategy';
import { JwtStrategy } from './jwt.strategy';
import { UsersModule } from '../users/users.module';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';
import { jwtConstants } from './constants';

@Module({
  imports: [
    UsersModule,
    PassportModule,
    JwtModule.register({
      secret: jwtConstants.secret,
      signOptions: { expiresIn: '60s' },
    }),
  ],
  providers: [AuthService, LocalStrategy, JwtStrategy],
  exports: [AuthService],
})
export class AuthModule {}
@@switch
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { LocalStrategy } from './local.strategy';
import { JwtStrategy } from './jwt.strategy';
import { UsersModule } from '../users/users.module';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';
import { jwtConstants } from './constants';

@Module({
  imports: [
    UsersModule,
    PassportModule,
    JwtModule.register({
      secret: jwtConstants.secret,
      signOptions: { expiresIn: '60s' },
    }),
  ],
  providers: [AuthService, LocalStrategy, JwtStrategy],
  exports: [AuthService],
})
export class AuthModule {}
```

By importing the same secret used when we signed the JWT, we ensure that the **verify** phase performed by Passport, and the **sign** phase performed in our AuthService, use a common secret.

Finally, we define the `JwtAuthGuard` class which extends the built-in `AuthGuard`:

```
@@filename(auth/jwt-auth.guard)
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';
```

```
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

Implement protected route and JWT strategy guards

We can now implement our protected route and its associated Guard.

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

```
@@filename(app.controller)
import { Controller, Get, Request, Post, UseGuards } from
'@nestjs/common';
import { JwtAuthGuard } from './auth/jwt-auth.guard';
import { LocalAuthGuard } from './auth/local-auth.guard';
import { AuthService } from './auth/auth.service';

@Controller()
export class AppController {
  constructor(private authService: AuthService) {}

  @UseGuards(LocalAuthGuard)
  @Post('auth/login')
  async login(@Request() req) {
    return this.authService.login(req.user);
  }

  @UseGuards(JwtAuthGuard)
  @Get('profile')
  getProfile(@Request() req) {
    return req.user;
  }
}
@@switch
import { Controller, Dependencies, Bind, Get, Request, Post, UseGuards }
from '@nestjs/common';
import { JwtAuthGuard } from './auth/jwt-auth.guard';
import { LocalAuthGuard } from './auth/local-auth.guard';
import { AuthService } from './auth/auth.service';

@Dependencies(AuthService)
@Controller()
export class AppController {
  constructor(authService) {
    this.authService = authService;
  }

  @UseGuards(LocalAuthGuard)
  @Post('auth/login')
  @Bind(Request())
  async login(req) {
    return this.authService.login(req.user);
  }

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

Once again, we're applying the `AuthGuard` that the `@nestjs/passport` module has automatically provisioned for us when we configured the passport-jwt module. This Guard is referenced by its default name, `jwt`. When our `GET /profile` route is hit, the Guard will automatically invoke our passport-jwt custom configured strategy, validate the JWT, and assign the `user` property to the `Request` object.

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

```
$ # GET /profile
$ curl http://localhost:3000/profile
$ # result -> {"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"
$ # result ->
{"access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm... }

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

Note that in the `AuthModule`, we configured the JWT to have an expiration of `60 seconds`. This is probably 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 and the passport-jwt strategy. If you wait 60 seconds after authenticating before attempting a `GET /profile` request, you'll receive a `401 Unauthorized` response. This is because Passport 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.

Extending guards

In most cases, using a provided `AuthGuard` class is sufficient. However, there might be use-cases when you would like to simply extend the default error handling or authentication logic. For this, you can extend the built-in class and override methods within a sub-class.

```
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) {
    // You can throw an exception based on either "info" or "err"
arguments
    if (err || !user) {
      throw err || new UnauthorizedException();
    }
    return user;
  }
}
```

In addition to extending the default error handling and authentication logic, we can allow authentication to go through a chain of strategies. The first strategy to succeed, redirect, or error will halt the chain. Authentication failures will proceed through each strategy in series, ultimately failing if all strategies fail.

```
export class JwtAuthGuard extends AuthGuard(['strategy_jwt_1',
'strategy_jwt_2', '...']) { ... }
```

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 `JwtAuthGuard` as a global guard using the following construction (in any module):

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

With this in place, Nest will automatically bind `JwtAuthGuard` 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 `JwtAuthGuard` to return `true` when the `"isPublic"` metadata is found. For this, we'll use the `Reflector` class (read more here).

```
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  constructor(private reflector: Reflector) {
    super();
  }

  canActivate(context: ExecutionContext) {
    const isPublic = this.reflector.getAllAndOverride<boolean>
(IS_PUBLIC_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (isPublic) {
      return true;
    }
    return super.canActivate(context);
  }
}
```

Request-scoped strategies

The passport API is based on registering strategies to the global instance of the library. Therefore strategies are not designed to have request-dependent options or to be dynamically instantiated per request (read more about the request-scoped providers). When you configure your strategy to be request-scoped, Nest will never instantiate it since it's not tied to any specific route. There is no physical way to determine which "request-scoped" strategies should be executed per request.

However, there are ways to dynamically resolve request-scoped providers within the strategy. For this, we leverage the module reference feature.

First, open the `local.strategy.ts` file and inject the `ModuleRef` in the normal way:

```
constructor(private moduleRef: ModuleRef) {
  super({
    passReqToCallback: true,
  });
}
```

> info **Hint** The `ModuleRef` class is imported from the `@nestjs/core` package.

Be sure to set the `passReqToCallback` configuration property to `true`, as shown above.

In the next step, the request instance will be used to obtain the current context identifier, instead of generating a new one (read more about request context here).

Now, inside the `validate()` method of the `LocalStrategy` class, use the `getByRequest()` method of the `ContextIdFactory` class to create a context id based on the request object, and pass this to the `resolve()` call:

```
async validate(
  request: Request,
  username: string,
  password: string,
) {
  const contextId = ContextIdFactory.getByRequest(request);
  // "AuthService" is a request-scoped provider
  const authService = await this.moduleRef.resolve(AuthService,
contextId);
  ...
}
```

In the example above, the `resolve()` method will asynchronously return the request-scoped instance of the `AuthService` provider (we assumed that `AuthService` is marked as a request-scoped provider).

Customize Passport

Any standard Passport customization options can be passed the same way, using the `register()` method. The available options depend on the strategy being implemented. For example:

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

You can also pass strategies an options object in their constructors to configure them. For the local strategy you can pass e.g.:

```
constructor(private authService: AuthService) {
  super({
    usernameField: 'email',
    passwordField: 'password',
  });
}
```

Take a look at the official Passport Website for property names.

Named strategies

When implementing a strategy, you can provide a name for it by passing a second argument to the `PassportStrategy` function. If you don't do this, each strategy will have a default name (e.g., 'jwt' for jwt-strategy):

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

Then, you refer to this via a decorator like `@UseGuards(AuthGuard('myjwt'))`.

GraphQL

In order to use an AuthGuard with GraphQL, extend the built-in AuthGuard class and override the getRequest() method.

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

To get the current authenticated user in your graphql resolver, you can define a `@CurrentUser()` decorator:

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

export const CurrentUser = createParamDecorator(
  (data: unknown, context: ExecutionContext) => {
    const ctx = GqlExecutionContext.create(context);
    return ctx.getContext().req.user;
  },
);
```

To use above decorator in your resolver, be sure to include it as a parameter of your query or mutation:

```
@Query(returns => User)
@UseGuards(GqlAuthGuard)
whoAmI(@CurrentUser() user: User) {
  return this.usersService.findById(user.id);
}
```