

Read-Eval-Print-Loop (REPL)

REPL is a simple interactive environment that takes single user inputs, executes them, and returns the result to the user. The REPL feature lets you inspect your dependency graph and call methods on your providers (and controllers) directly from your terminal.

Usage

To run your NestJS application in REPL mode, create a new `repl.ts` file (alongside the existing `main.ts` file) and add the following code inside:

```
@filename(repl)
import { repl } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  await repl(AppModule);
}
bootstrap();
@switch
import { repl } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  await repl(AppModule);
}
bootstrap();
```

Now in your terminal, start the REPL with the following command:

```
$ npm run start -- --entryFile repl
```

info **Hint** `repl` returns a [Node.js REPL server](#) object.

Once it's up and running, you should see the following message in your console:

```
LOG [NestFactory] Starting Nest application...
LOG [InstanceLoader] AppModule dependencies initialized
LOG REPL initialized
```

And now you can start interacting with your dependencies graph. For instance, you can retrieve an `AppService` (we are using the starter project as an example here) and call the `getHello()` method:

```
> get(AppService).getHello()  
'Hello World!'
```

You can execute any JavaScript code from within your terminal, for example, assign an instance of the `AppController` to a local variable, and use `await` to call an asynchronous method:

```
> appController = get(AppController)  
AppController { appService: AppService {} }  
> await appController.getHello()  
'Hello World!'
```

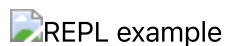
To display all public methods available on a given provider or controller, use the `methods()` function, as follows:

```
> methods(AppController)  
  
Methods:  
  □ getHello
```

To print all registered modules as a list together with their controllers and providers, use `debug()`.

```
> debug()  
  
AppModule:  
  - controllers:  
    □ AppController  
  - providers:  
    □ AppService
```

Quick demo:



You can find more information about the existing, predefined native methods in the section below.

Native functions

The built-in NestJS REPL comes with a few native functions that are globally available when you start REPL. You can call `help()` to list them out.

If you don't recall what's the signature (ie: expected parameters and a return type) of a function, you can call `<function_name>.help`. For instance:

```
> $.help
Retrieves an instance of either injectable or controller, otherwise,
throws exception.
Interface: $(token: InjectionToken) => any
```

info Hint Those function interfaces are written in [TypeScript function type expression syntax](#).

Function	Description	Signature
<code>debug</code>	Print all registered modules as a list together with their controllers and providers.	<code>debug(moduleCls?: ClassRef \ \nstring) => void</code>
<code>get</code> or <code>\$</code>	Retrieves an instance of either injectable or controller, otherwise, throws exception.	<code>get(token: InjectionToken) =>\nany</code>
<code>methods</code>	Display all public methods available on a given provider or controller.	<code>methods(token: ClassRef \ \nstring) => void</code>
<code>resolve</code>	Resolves transient or request-scoped instance of either injectable or controller, otherwise, throws exception.	<code>resolve(token:\nInjectionToken, contextId:\nany) => Promise<any></code>
<code>select</code>	Allows navigating through the modules tree, for example, to pull out a specific instance from the selected module.	<code>select(token: DynamicModule\n \nClassRef) =>\nINestApplicationContext</code>

Watch mode

During development it is useful to run REPL in a watch mode to reflect all the code changes automatically:

```
$ npm run start -- --watch --entryFile repl
```

This has one flaw, the REPL's command history is discarded after each reload which might be cumbersome. Fortunately, there is a very simple solution. Modify your `bootstrap` function like this:

```
async function bootstrap() {\n  const replServer = await repl(AppModule);\n  replServer.setupHistory(".nestjs_repl_history", (err) => {\n    if (err) {\n      console.error(err);\n    }\n  });\n}
```

Now the history is preserved between the runs/reloads.

CRUD generator

Throughout the life span of a project, when we build new features, we often need to add new resources to our application. These resources typically require multiple, repetitive operations that we have to repeat each time we define a new resource.

Introduction

Let's imagine a real-world scenario, where we need to expose CRUD endpoints for 2 entities, let's say **User** and **Product** entities. Following the best practices, for each entity we would have to perform several operations, as follows:

- Generate a module (**nest g mo**) to keep code organized and establish clear boundaries (grouping related components)
- Generate a controller (**nest g co**) to define CRUD routes (or queries/mutations for GraphQL applications)
- Generate a service (**nest g s**) to implement & isolate business logic
- Generate an entity class/interface to represent the resource data shape
- Generate Data Transfer Objects (or inputs for GraphQL applications) to define how the data will be sent over the network

That's a lot of steps!

To help speed up this repetitive process, **Nest CLI** provides a generator (schematic) that automatically generates all the boilerplate code to help us avoid doing all of this, and make the developer experience much simpler.

info **Note** The schematic supports generating **HTTP** controllers, **Microservice** controllers, **GraphQL** resolvers (both code first and schema first), and **WebSocket** Gateways.

Generating a new resource

To create a new resource, simply run the following command in the root directory of your project:

```
$ nest g resource
```

nest g resource command not only generates all the NestJS building blocks (module, service, controller classes) but also an entity class, DTO classes as well as the testing (**.spec**) files.

Below you can see the generated controller file (for REST API):

```
@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Post()
  create(@Body() createUserDto: CreateUserDto) {
```

```

    return this.userService.create(createUserDto);
  }

  @Get()
  findAll() {
    return this.userService.findAll();
  }

  @Get('/:id')
  findOne(@Param('id') id: string) {
    return this.userService.findOne(+id);
  }

  @Patch('/:id')
  update(@Param('id') id: string, @Body() updateUserDto: UpdateUserDto) {
    return this.userService.update(+id, updateUserDto);
  }

  @Delete('/:id')
  remove(@Param('id') id: string) {
    return this.userService.remove(+id);
  }
}

```

Also, it automatically creates placeholders for all the CRUD endpoints (routes for REST APIs, queries and mutations for GraphQL, message subscribes for both Microservices and WebSocket Gateways) - all without having to lift a finger.

warning **Note** Generated service classes are **not** tied to any specific **ORM (or data source)**. This makes the generator generic enough to meet the needs of any project. By default, all methods will contain placeholders, allowing you to populate it with the data sources specific to your project.

Likewise, if you want to generate resolvers for a GraphQL application, simply select the **GraphQL (code first)** (or **GraphQL (schema first)**) as your transport layer.

In this case, NestJS will generate a resolver class instead of a REST API controller:

```

$ nest g resource users

> ? What transport layer do you use? GraphQL (code first)
> ? Would you like to generate CRUD entry points? Yes
> CREATE src/users/users.module.ts (224 bytes)
> CREATE src/users/users.resolver.spec.ts (525 bytes)
> CREATE src/users/users.resolver.ts (1109 bytes)
> CREATE src/users/users.service.spec.ts (453 bytes)
> CREATE src/users/users.service.ts (625 bytes)
> CREATE src/users/dto/create-user.input.ts (195 bytes)
> CREATE src/users/dto/update-user.input.ts (281 bytes)
> CREATE src/users/entities/user.entity.ts (187 bytes)
> UPDATE src/app.module.ts (312 bytes)

```

info **Hint** To avoid generating test files, you can pass the `--no-spec` flag, as follows: `nest g resource users --no-spec`

We can see below, that not only were all boilerplate mutations and queries created, but everything is all tied together. We're utilizing the `UsersService`, `User` Entity, and our DTO's.

```
import { Resolver, Query, Mutation, Args, Int } from '@nestjs/graphql';
import { UsersService } from './users.service';
import { User } from './entities/user.entity';
import { CreateUserInput } from './dto/create-user.input';
import { UpdateUserInput } from './dto/update-user.input';

@Resolver(() => User)
export class UsersResolver {
  constructor(private readonly usersService: UsersService) {}

  @Mutation(() => User)
  createUser(@Args('createUserInput') createUserInput: CreateUserInput) {
    return this.usersService.create(createUserInput);
  }

  @Query(() => [User], { name: 'users' })
  findAll() {
    return this.usersService.findAll();
  }

  @Query(() => User, { name: 'user' })
  findOne(@Args('id', { type: () => Int }) id: number) {
    return this.usersService.findOne(id);
  }

  @Mutation(() => User)
  updateUser(@Args('updateUserInput') updateUserInput: UpdateUserInput) {
    return this.usersService.update(updateUserInput.id, updateUserInput);
  }

  @Mutation(() => User)
  removeUser(@Args('id', { type: () => Int }) id: number) {
    return this.usersService.remove(id);
  }
}
```

SWC

SWC (Speedy Web Compiler) is an extensible Rust-based platform that can be used for both compilation and bundling. Using SWC with Nest CLI is a great and simple way to significantly speed up your development process.

info Hint SWC is approximately **x20 times faster** than the default TypeScript compiler.

Installation

To get started, first install a few packages:

```
$ npm i --save-dev @swc/cli @swc/core
```

Getting started

Once the installation process is complete, you can use the **swc** builder with Nest CLI, as follows:

```
$ nest start -b swc  
# OR nest start --builder swc
```

info Hint If your repository is a monorepo, check out [this section](#).

Instead of passing the **-b** flag you can also just set the **compilerOptions.builder** property to **"swc"** in your **nest-cli.json** file, like so:

```
{  
  "compilerOptions": {  
    "builder": "swc"  
  }  
}
```

To customize builder's behavior, you can pass an object containing two attributes, **type** (**"swc"**) and **options**, as follows:

```
"compilerOptions": {  
  "builder": {  
    "type": "swc",  
    "options": {  
      "swcrcPath": "infrastructure/.swcrc",  
    }  
  }  
}
```

To run the application in watch mode, use the following command:

```
$ nest start -b swc -w  
# OR nest start --builder swc --watch
```

Type checking

SWC does not perform any type checking itself (as opposed to the default TypeScript compiler), so to turn it on, you need to use the `--type-check` flag:

```
$ nest start -b swc --type-check
```

This command will instruct the Nest CLI to run `tsc` in `noEmit` mode alongside SWC, which will asynchronously perform type checking. Again, instead of passing the `--type-check` flag you can also just set the `compilerOptions.typeCheck` property to `true` in your `nest-cli.json` file, like so:

```
{  
  "compilerOptions": {  
    "builder": "swc",  
    "typeCheck": true  
  }  
}
```

CLI Plugins (SWC)

The `--type-check` flag will automatically execute **NestJS CLI plugins** and produce a serialized metadata file which then can be loaded by the application at runtime.

SWC configuration

SWC builder is pre-configured to match the requirements of NestJS applications. However, you can customize the configuration by creating a `.swcrc` file in the root directory and tweaking the options as you wish.

```
{  
  "$schema": "https://json.schemastore.org/swcrc",  
  "sourceMaps": true,  
  "jsc": {  
    "parser": {  
      "syntax": "typescript",  
      "decorators": true,  
      "dynamicImport": true  
    },  
    "baseUrl": "./"  
  }
```



```
},  
"minify": false  
}
```

Monorepo

If your repository is a monorepo, then instead of using `swc` builder you have to configure `webpack` to use `swc-loader`.

First, let's install the required package:

```
$ npm i --save-dev swc-loader
```

Once the installation is complete, create a `webpack.config.js` file in the root directory of your application with the following content:

```
const swcDefaultConfig = require('@nestjs/cli/lib/compiler/defaults/swc-  
defaults').swcDefaultsFactory().swcOptions;  
  
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.ts$/,  
        exclude: /node_modules/,  
        use: {  
          loader: 'swc-loader',  
          options: swcDefaultConfig,  
        },  
      },  
    ],  
  },  
};
```

Monorepo and CLI plugins

Now if you use CLI plugins, `swc-loader` will not load them automatically. Instead, you have to create a separate file that will load them manually. To do so, declare a `generate-metadata.ts` file near the `main.ts` file with the following content:

```
import { PluginMetadataGenerator } from  
'@nestjs/cli/lib/compiler/plugins';  
import { ReadonlyVisitor } from '@nestjs/swagger/dist/plugin';  
  
const generator = new PluginMetadataGenerator();  
generator.generate({
```

```
visitors: [new ReadonlyVisitor({ introspectComments: true, pathToSource:
__dirname })],
outputDir: __dirname,
watch: true,
tsconfigPath: 'apps/<name>/tsconfig.app.json',
});
```

info **Hint** In this example we used `@nestjs/swagger` plugin, but you can use any plugin of your choice.

The `generate()` method accepts the following options:

<code>watch</code>	Whether to watch the project for changes.
<code>tsconfigPath</code>	Path to the <code>tsconfig.json</code> file. Relative to the current working directory (<code>process.cwd()</code>).
<code>outputDir</code>	Path to the directory where the metadata file will be saved.
<code>visitors</code>	An array of visitors that will be used to generate metadata.
<code>filename</code>	The name of the metadata file. Defaults to <code>metadata.ts</code> .
<code>printDiagnostics</code>	Whether to print diagnostics to the console. Defaults to <code>true</code> .

Finally, you can run the `generate-metadata` script in a separate terminal window with the following command:

```
$ npx ts-node src/generate-metadata.ts
# OR npx ts-node apps/{YOUR_APP}/src/generate-metadata.ts
```

Common pitfalls

If you use TypeORM/MikroORM or any other ORM in your application, you may stumble upon circular import issues. SWC doesn't handle **circular imports** well, so you should use the following workaround:

```
@Entity()
export class User {
  @OneToOne(() => Profile, (profile) => profile.user)
  profile: Relation<Profile>; // <--- see "Relation<>" type here instead
  of just "Profile"
}
```

info **Hint** `Relation` type is exported from the `typeorm` package.

Doing this prevents the type of the property from being saved in the transpiled code in the property metadata, preventing circular dependency issues.

If your ORM does not provide a similar workaround, you can define the wrapper type yourself:

```
/**
 * Wrapper type used to circumvent ESM modules circular dependency issue
 * caused by reflection metadata saving the type of the property.
 */
export type WrapperType<T> = T; // WrapperType === Relation
```

For all [circular dependency injections](#) in your project, you will also need to use the custom wrapper type described above:

```
@Injectable()
export class UserService {
  constructor(
    @Inject(forwardRef(() => ProfileService))
    private readonly profileService: WrapperType<ProfileService>,
  ) {}
}
```

Jest + SWC

To use SWC with Jest, you need to install the following packages:

```
$ npm i --save-dev jest @swc/core @swc/jest
```

Once the installation is complete, update the `package.json/jest.config.js` file (depending on your configuration) with the following content:

```
{
  "jest": {
    "transform": {
      "^.+\\.\\.?(t|j)s?$": ["@swc/jest"]
    }
  }
}
```

Additionally you would need to add the following `transform` properties to your `.swcrc` file: `LegacyDecorator`, `decoratorMetadata`:

```
{
  "$schema": "https://json.schemastore.org/swcrc",
  "sourceMaps": true,
  "jsc": {
```

```
"parser": {
  "syntax": "typescript",
  "decorators": true,
  "dynamicImport": true
},
"transform": {
  "legacyDecorator": true,
  "decoratorMetadata": true
},
"baseUrl": "./"
},
"minify": false
}
```

If you use NestJS CLI Plugins in your project, you'll have to run `PluginMetadataGenerator` manually. Navigate to [this section](#) to learn more.

Vitest

[Vitest](#) is a fast and lightweight test runner designed to work with Vite. It provides a modern, fast, and easy-to-use testing solution that can be integrated with NestJS projects.

Installation

To get started, first install the required packages:

```
$ npm i --save-dev vitest unplugin-swc @swc/core @vitest/coverage-c8
```

Configuration

Create a `vitest.config.ts` file in the root directory of your application with the following content:

```
import swc from 'unplugin-swc';
import { defineConfig } from 'vitest/config';

export default defineConfig({
  test: {
    globals: true,
    root: './',
  },
  plugins: [
    // This is required to build the test files with SWC
    swc.vite({
      // Explicitly set the module type to avoid inheriting this value
      // from a `.swcrc` config file
      module: { type: 'es6' },
    }),
  ],
});
```

```
    ],  
  });  
};
```

This configuration file sets up the Vitest environment, root directory, and SWC plugin. You should also create a separate configuration file for e2e tests, with an additional `include` field that specifies the test path regex:

```
import swc from 'unplugin-swc';  
import { defineConfig } from 'vitest/config';  
  
export default defineConfig({  
  test: {  
    include: ['**/*.e2e-spec.ts'],  
    globals: true,  
    root: './',  
  },  
  plugins: [swc.vite()],  
});
```

Additionally, you can set the `alias` options to support TypeScript paths in your tests:

```
import swc from 'unplugin-swc';  
import { defineConfig } from 'vitest/config';  
  
export default defineConfig({  
  test: {  
    include: ['**/*.e2e-spec.ts'],  
    globals: true,  
    alias: {  
      '@src': './src',  
      '@test': './test',  
    },  
    root: './',  
  },  
  resolve: {  
    alias: {  
      '@src': './src',  
      '@test': './test',  
    },  
  },  
  plugins: [swc.vite()],  
});
```

Update imports in E2E tests

Change any E2E test imports using `import * as request from 'supertest'` to `import request from 'supertest'`. This is necessary because Vitest, when bundled with Vite, expects a default import

for supertest. Using a namespace import may cause issues in this specific setup.

Lastly, update the test scripts in your package.json file to the following:

```
{
  "scripts": {
    "test": "vitest run",
    "test:watch": "vitest",
    "test:cov": "vitest run --coverage",
    "test:debug": "vitest --inspect-brk --inspect --logHeapUsage --
threads=false",
    "test:e2e": "vitest run --config ./vitest.config.e2e.ts"
  }
}
```

These scripts configure Vitest for running tests, watching for changes, generating code coverage reports, and debugging. The test:e2e script is specifically for running E2E tests with a custom configuration file.

With this setup, you can now enjoy the benefits of using Vitest in your NestJS project, including faster test execution and a more modern testing experience.

info Hint You can check out a working example in this [repository](#)

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 `UserService`, 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 `UserService` 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.

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 `UserService`.

```
$ # 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 `AllowAnonymous`, 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);
}
```

Hot Reload

The highest impact on your application's bootstrapping process is **TypeScript compilation**. Fortunately, with [webpack](#) HMR (Hot-Module Replacement), we don't need to recompile the entire project each time a change occurs. This significantly decreases the amount of time necessary to instantiate your application, and makes iterative development a lot easier.

warning **Warning** Note that [webpack](#) won't automatically copy your assets (e.g. [graphql](#) files) to the [dist](#) folder. Similarly, [webpack](#) is not compatible with glob static paths (e.g., the [entities](#) property in [TypeOrmModule](#)).

With CLI

If you are using the [Nest CLI](#), the configuration process is pretty straightforward. The CLI wraps [webpack](#), which allows use of the [HotModuleReplacementPlugin](#).

Installation

First install the required packages:

```
$ npm i --save-dev webpack-node-externals run-script-webpack-plugin webpack
```

info **Hint** If you use **Yarn Berry** (not classic Yarn), install the [webpack-pnp-externals](#) package instead of the [webpack-node-externals](#).

Configuration

Once the installation is complete, create a [webpack-hmr.config.js](#) file in the root directory of your application.

```
const nodeExternals = require('webpack-node-externals');
const { RunScriptWebpackPlugin } = require('run-script-webpack-plugin');

module.exports = function (options, webpack) {
  return {
    ...options,
    entry: ['webpack/hot/poll?100', options.entry],
    externals: [
      nodeExternals({
        allowlist: ['webpack/hot/poll?100'],
      }),
    ],
    plugins: [
      ...options.plugins,
      new webpack.HotModuleReplacementPlugin(),
      new webpack.WatchIgnorePlugin({
        paths: [/\.js$/, /\.d\.ts$/],
      })
    ]
  };
};
```

```

    }),
    new RunScriptWebpackPlugin({ name: options.output.filename,
autoRestart: false })),
  ],
};
};

```

info **Hint** With **Yarn Berry** (not classic Yarn), instead of using the `nodeExternals` in the `externals` configuration property, use the `WebpackPnpExternals` from `webpack-pnp-externals` package: `WebpackPnpExternals({{ '{' }} exclude: ['webpack/hot/poll?100'] {{ '}' }})`.

This function takes the original object containing the default webpack configuration as a first argument, and the reference to the underlying `webpack` package used by the Nest CLI as the second one. Also, it returns a modified webpack configuration with the `HotModuleReplacementPlugin`, `WatchIgnorePlugin`, and `RunScriptWebpackPlugin` plugins.

Hot-Module Replacement

To enable **HMR**, open the application entry file (`main.ts`) and add the following webpack-related instructions:

```

declare const module: any;

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);

  if (module.hot) {
    module.hot.accept();
    module.hot.dispose(() => app.close());
  }
}
bootstrap();

```

To simplify the execution process, add a script to your `package.json` file.

```

"start:dev": "nest build --webpack --webpackPath webpack-hmr.config.js --watch"

```

Now simply open your command line and run the following command:

```

$ npm run start:dev

```

Without CLI

If you are not using the [Nest CLI](#), the configuration will be slightly more complex (will require more manual steps).

Installation

First install the required packages:

```
$ npm i --save-dev webpack webpack-cli webpack-node-externals ts-loader
run-script-webpack-plugin
```

Hint If you use **Yarn Berry** (not classic Yarn), install the `webpack-pnp-externals` package instead of the `webpack-node-externals`.

Configuration

Once the installation is complete, create a `webpack.config.js` file in the root directory of your application.

```
const webpack = require('webpack');
const path = require('path');
const nodeExternals = require('webpack-node-externals');
const { RunScriptWebpackPlugin } = require('run-script-webpack-plugin');

module.exports = {
  entry: ['webpack/hot/poll?100', './src/main.ts'],
  target: 'node',
  externals: [
    nodeExternals({
      allowlist: ['webpack/hot/poll?100'],
    }),
  ],
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      },
    ],
  },
  mode: 'development',
  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    new RunScriptWebpackPlugin({ name: 'server.js', autoRestart: false }),
  ],
  output: {
```



```
    path: path.join(__dirname, 'dist'),
    filename: 'server.js',
  },
};
```

info **Hint** With **Yarn Berry** (not classic Yarn), instead of using the `nodeExternals` in the `externals` configuration property, use the `WebpackPnpExternals` from `webpack-pnp-externals` package: `WebpackPnpExternals({{ '{' }} exclude: ['webpack/hot/poll?100'] {{ '}' }})`.

This configuration tells webpack a few essential things about your application: location of the entry file, which directory should be used to hold **compiled** files, and what kind of loader we want to use to compile source files. Generally, you should be able to use this file as-is, even if you don't fully understand all of the options.

Hot-Module Replacement

To enable **HMR**, open the application entry file (`main.ts`) and add the following webpack-related instructions:

```
declare const module: any;

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);

  if (module.hot) {
    module.hot.accept();
    module.hot.dispose(() => app.close());
  }
}
bootstrap();
```

To simplify the execution process, add a script to your `package.json` file.

```
"start:dev": "webpack --config webpack.config.js --watch"
```

Now simply open your command line and run the following command:

```
$ npm run start:dev
```

Example

A working example is available [here](#).

MikroORM

This recipe is here to help users getting started with MikroORM in Nest. MikroORM is the TypeScript ORM for Node.js based on Data Mapper, Unit of Work and Identity Map patterns. It is a great alternative to TypeORM and migration from TypeORM should be fairly easy. The complete documentation on MikroORM can be found [here](#).

info `@mikro-orm/nestjs` is a third party package and is not managed by the NestJS core team. Please, report any issues found with the library in the [appropriate repository](#).

Installation

Easiest way to integrate MikroORM to Nest is via `@mikro-orm/nestjs` module. Simply install it next to Nest, MikroORM and underlying driver:

```
$ npm i @mikro-orm/core @mikro-orm/nestjs @mikro-orm/mysql # for  
mysql/mariadb
```

MikroORM also supports `postgres`, `sqlite`, and `mongo`. See the [official docs](#) for all drivers.

Once the installation process is completed, we can import the `MikroOrmModule` into the root `AppModule`.

```
@Module({  
  imports: [  
    MikroOrmModule.forRoot({  
      entities: ['./dist/entities'],  
      entitiesTs: ['./src/entities'],  
      dbName: 'my-db-name.sqlite3',  
      type: 'sqlite',  
    }),  
  ],  
  controllers: [AppController],  
  providers: [AppService],  
})  
export class AppModule {}
```

The `forRoot()` method accepts the same configuration object as `init()` from the MikroORM package. Check [this page](#) for the complete configuration documentation.

Alternatively we can [configure the CLI](#) by creating a configuration file `mikro-orm.config.ts` and then call the `forRoot()` without any arguments. This won't work when you use a build tools that use tree shaking.

```
@Module({  
  imports: [  
    MikroOrmModule.forRoot(),
```

```
    ],
    ...
  })
  export class AppModule {}
```

Afterward, the `EntityManager` will be available to inject across entire project (without importing any module elsewhere).

```
import { MikroORM } from '@mikro-orm/core';
// Import EntityManager from your driver package or '@mikro-orm/knex'
import { EntityManager } from '@mikro-orm/mysql';

@Injectable()
export class MyService {
  constructor(
    private readonly orm: MikroORM,
    private readonly em: EntityManager,
  ) {}
}
```

info Notice that the `EntityManager` is imported from the `@mikro-orm/driver` package, where driver is `mysql`, `sqlite`, `postgres` or what driver you are using. In case you have `@mikro-orm/knex` installed as a dependency, you can also import the `EntityManager` from there.

Repositories

MikroORM supports the repository design pattern. For every entity we can create a repository. Read the complete documentation on repositories [here](#). To define which repositories should be registered in the current scope you can use the `forFeature()` method. For example, in this way:

info You should **not** register your base entities via `forFeature()`, as there are no repositories for those. On the other hand, base entities need to be part of the list in `forRoot()` (or in the ORM config in general).

```
// photo.module.ts
@Module({
  imports: [MikroOrmModule.forFeature([Photo])],
  providers: [PhotoService],
  controllers: [PhotoController],
})
export class PhotoModule {}
```

and import it into the root `AppModule`:

```
// app.module.ts
@Module({
```

```
imports: [MikroOrmModule.forRoot(...), PhotoModule],
})
export class AppModule {}
```

In this way we can inject the `PhotoRepository` to the `PhotoService` using the `@InjectRepository()` decorator:

```
@Injectable()
export class PhotoService {
  constructor(
    @InjectRepository(Photo)
    private readonly photoRepository: EntityRepository<Photo>,
  ) {}
}
```

Using custom repositories

When using custom repositories, we can get around the need for `@InjectRepository()` decorator by naming our repositories the same way as `getRepositoryToken()` method do:

```
export const getRepositoryToken = <T>(entity: EntityName<T>) =>
  `${Utils.className(entity)}Repository`;
```

In other words, as long as we name the repository same was as the entity is called, appending `Repository` suffix, the repository will be registered automatically in the Nest DI container.

```
// `**./author.entity.ts**`
@Entity()
export class Author {
  // to allow inference in `em.getRepository()`
  [EntityRepositoryType]?: AuthorRepository;
}

// `**./author.repository.ts**`
@Repository(Author)
export class AuthorRepository extends EntityRepository<Author> {
  // your custom methods...
}
```

As the custom repository name is the same as what `getRepositoryToken()` would return, we do not need the `@InjectRepository()` decorator anymore:

```
@Injectable()
export class MyService {
```

```
    constructor(private readonly repo: AuthorRepository) {}  
}
```

Load entities automatically

info **info** `autoLoadEntities` option was added in v4.1.0

Manually adding entities to the entities array of the connection options can be tedious. In addition, referencing entities from the root module breaks application domain boundaries and causes leaking implementation details to other parts of the application. To solve this issue, static glob paths can be used.

Note, however, that glob paths are not supported by webpack, so if you are building your application within a monorepo, you won't be able to use them. To address this issue, an alternative solution is provided. To automatically load entities, set the `autoLoadEntities` property of the configuration object (passed into the `forRoot()` method) to `true`, as shown below:

```
@Module({  
  imports: [  
    MikroOrmModule.forRoot({  
      ...  
      autoLoadEntities: true,  
    }),  
  ],  
})  
export class AppModule {}
```

With that option specified, every entity registered through the `forFeature()` method will be automatically added to the entities array of the configuration object.

info **info** Note that entities that aren't registered through the `forFeature()` method, but are only referenced from the entity (via a relationship), won't be included by way of the `autoLoadEntities` setting.

info **info** Using `autoLoadEntities` also has no effect on the MikroORM CLI - for that we still need CLI config with the full list of entities. On the other hand, we can use globs there, as the CLI won't go thru webpack.

Serialization

warning **Note** MikroORM wraps every single entity relation in a `Reference<T>` or a `Collection<T>` object, in order to provide better type-safety. This will make [Nest's built-in serializer](#) blind to any wrapped relations. In other words, if you return MikroORM entities from your HTTP or WebSocket handlers, all of their relations will NOT be serialized.

Luckily, MikroORM provides a [serialization API](#) which can be used in lieu of `ClassSerializerInterceptor`.

```

@Entity()
export class Book {
  @Property({ hidden: true }) // Equivalent of class-transformer's
  `@Exclude`
  hiddenField = Date.now();

  @Property({ persist: false }) // Similar to class-transformer's
  `@Expose()` Will only exist in memory, and will be serialized.
  count?: number;

  @ManyToOne({
    serializer: (value) => value.name,
    serializedName: 'authorName',
  }) // Equivalent of class-transformer's `@Transform()`
  author: Author;
}

```

Request scoped handlers in queues

info `@UseRequestContext()` decorator was added in v4.1.0

As mentioned in the [docs](#), we need a clean state for each request. That is handled automatically thanks to the `RequestContext` helper registered via middleware.

But middlewares are executed only for regular HTTP request handles, what if we need a request scoped method outside of that? One example of that is queue handlers or scheduled tasks.

We can use the `@UseRequestContext()` decorator. It requires you to first inject the `MikroORM` instance to current context, it will be then used to create the context for you. Under the hood, the decorator will register new request context for your method and execute it inside the context.

```

@Injectable()
export class MyService {
  constructor(private readonly orm: MikroORM) {}

  @UseRequestContext()
  async doSomething() {
    // this will be executed in a separate context
  }
}

```

Using `AsyncLocalStorage` for request context

By default, the `domain` api is used in the `RequestContext` helper. Since `@mikro-orm/core@4.0.3`, you can use the new `AsyncLocalStorage` too, if you are on up to date node version:

```
// create new (global) storage instance
const storage = new AsyncLocalStorage<EntityManager>();

@Module({
  imports: [
    MikroOrmModule.forRoot({
      // ...
      registerRequestContext: false, // disable automatic middleware
      context: () => storage.getStore(), // use our AsyncLocalStorage
    instance
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}

// register the request context middleware
const app = await NestFactory.create(AppModule, { ... });
const orm = app.get(MikroORM);

app.use((req, res, next) => {
  storage.run(orm.em.fork(true, true), next);
});
```

Testing

The `@mikro-orm/nestjs` package exposes `getRepositoryToken()` function that returns prepared token based on a given entity to allow mocking the repository.

```
@Module({
  providers: [
    PhotoService,
    {
      provide: getRepositoryToken(Photo),
      useValue: mockedRepository,
    },
  ],
})
export class PhotoModule {}
```

Example

A real world example of NestJS with MikroORM can be found [here](#)

SQL (TypeORM)

This chapter applies only to TypeScript

Warning In this article, you'll learn how to create a `DatabaseModule` based on the **TypeORM** package from scratch using custom providers mechanism. As a consequence, this solution contains a lot of overhead that you can omit using ready to use and available out-of-the-box dedicated `@nestjs/typeorm` package. To learn more, see [here](#).

TypeORM is definitely the most mature Object Relational Mapper (ORM) available in the node.js world. Since it's written in TypeScript, it works pretty well with the Nest framework.

Getting started

To start the adventure with this library we have to install all required dependencies:

```
$ npm install --save typeorm mysql2
```

The first step we need to do is to establish the connection with our database using `new DataSource().initialize()` class imported from the `typeorm` package. The `initialize()` function returns a `Promise`, and therefore we have to create an `async provider`.

```
@filename(database.providers)
import { DataSource } from 'typeorm';

export const databaseProviders = [
  {
    provide: 'DATA_SOURCE',
    useFactory: async () => {
      const dataSource = new DataSource({
        type: 'mysql',
        host: 'localhost',
        port: 3306,
        username: 'root',
        password: 'root',
        database: 'test',
        entities: [
          __dirname + '/../**/*.entity{.ts,.js}',
        ],
        synchronize: true,
      });

      return dataSource.initialize();
    },
  },
];
```


warning **Warning** Setting `synchronize: true` shouldn't be used in production - otherwise you can lose production data.

info **Hint** Following best practices, we declared the custom provider in the separated file which has a `*.providers.ts` suffix.

Then, we need to export these providers to make them **accessible** for the rest of the application.

```
@@filename(database.module)
import { Module } from '@nestjs/common';
import { databaseProviders } from './database.providers';

@Module({
  providers: [...databaseProviders],
  exports: [...databaseProviders],
})
export class DatabaseModule {}
```

Now we can inject the `DATA_SOURCE` object using `@Inject()` decorator. Each class that would depend on the `DATA_SOURCE` async provider will wait until a `Promise` is resolved.

Repository pattern

The `TypeORM` supports the repository design pattern, thus each entity has its own Repository. These repositories can be obtained from the database connection.

But firstly, we need at least one entity. We are going to reuse the `Photo` entity from the official documentation.

```
@@filename(photo.entity)
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Photo {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ length: 500 })
  name: string;

  @Column('text')
  description: string;

  @Column()
  filename: string;

  @Column('int')
  views: number;

  @Column()
```

```
    isPublished: boolean;
  }
```

The **Photo** entity belongs to the **photo** directory. This directory represents the **PhotoModule**. Now, let's create a **Repository** provider:

```
@@filename(photo.providers)
import { DataSource } from 'typeorm';
import { Photo } from './photo.entity';

export const photoProviders = [
  {
    provide: 'PHOTO_REPOSITORY',
    useFactory: (dataSource: DataSource) =>
      dataSource.getRepository(Photo),
    inject: ['DATA_SOURCE'],
  },
];
```

Warning In the real-world applications you should avoid **magic strings**. Both **PHOTO_REPOSITORY** and **DATA_SOURCE** should be kept in the separated **constants.ts** file.

Now we can inject the **Repository<Photo>** to the **PhotoService** using the **@Inject()** decorator:

```
@@filename(photo.service)
import { Injectable, Inject } from '@nestjs/common';
import { Repository } from 'typeorm';
import { Photo } from './photo.entity';

@Injectable()
export class PhotoService {
  constructor(
    @Inject('PHOTO_REPOSITORY')
    private photoRepository: Repository<Photo>,
  ) {}

  async findAll(): Promise<Photo[]> {
    return this.photoRepository.find();
  }
}
```

The database connection is **asynchronous**, but Nest makes this process completely invisible for the end-user. The **PhotoRepository** is waiting for the db connection, and the **PhotoService** is delayed until repository is ready to use. The entire application can start when each class is instantiated.

Here is a final **PhotoModule**:

```
@@filename(photo.module)
import { Module } from '@nestjs/common';
import { DatabaseModule } from '../database/database.module';
import { photoProviders } from './photo.providers';
import { PhotoService } from './photo.service';

@Module({
  imports: [DatabaseModule],
  providers: [
    ...photoProviders,
    PhotoService,
  ],
})
export class PhotoModule {}
```

info **Hint** Do not forget to import the `PhotoModule` into the root `AppModule`.

MongoDB (Mongoose)

Warning In this article, you'll learn how to create a `DatabaseModule` based on the **Mongoose** package from scratch using custom components. As a consequence, this solution contains a lot of overhead that you can omit using ready to use and available out-of-the-box dedicated `@nestjs/mongoose` package. To learn more, see [here](#).

Mongoose is the most popular **MongoDB** object modeling tool.

Getting started

To start the adventure with this library we have to install all required dependencies:

```
$ npm install --save mongoose
```

The first step we need to do is to establish the connection with our database using `connect()` function. The `connect()` function returns a **Promise**, and therefore we have to create an **async provider**.

```
@filename(database.providers)
import * as mongoose from 'mongoose';

export const databaseProviders = [
  {
    provide: 'DATABASE_CONNECTION',
    useFactory: (): Promise<typeof mongoose> =>
      mongoose.connect('mongodb://localhost/nest'),
  },
];

@filename
import * as mongoose from 'mongoose';

export const databaseProviders = [
  {
    provide: 'DATABASE_CONNECTION',
    useFactory: () => mongoose.connect('mongodb://localhost/nest'),
  },
];
```

Hint Following best practices, we declared the custom provider in the separated file which has a `*.providers.ts` suffix.

Then, we need to export these providers to make them **accessible** for the rest part of the application.

```
@filename(database.module)
import { Module } from '@nestjs/common';
import { databaseProviders } from './database.providers';
```

```
@Module({
  providers: [...databaseProviders],
  exports: [...databaseProviders],
})
export class DatabaseModule {}
```

Now we can inject the `Connection` object using `@Inject()` decorator. Each class that would depend on the `Connection` async provider will wait until a `Promise` is resolved.

Model injection

With Mongoose, everything is derived from a `Schema`. Let's define the `CatSchema`:

```
@@filename(schemas/cat.schema)
import * as mongoose from 'mongoose';

export const CatSchema = new mongoose.Schema({
  name: String,
  age: Number,
  breed: String,
});
```

The `CatSchema` belongs to the `cats` directory. This directory represents the `CatsModule`.

Now it's time to create a **Model** provider:

```
@@filename(cats.providers)
import { Connection } from 'mongoose';
import { CatSchema } from './schemas/cat.schema';

export const catsProviders = [
  {
    provide: 'CAT_MODEL',
    useFactory: (connection: Connection) => connection.model('Cat',
CatSchema),
    inject: ['DATABASE_CONNECTION'],
  },
];

@@switch
import { CatSchema } from './schemas/cat.schema';

export const catsProviders = [
  {
    provide: 'CAT_MODEL',
    useFactory: (connection) => connection.model('Cat', CatSchema),
    inject: ['DATABASE_CONNECTION'],
  },
];
```

warning **Warning** In the real-world applications you should avoid **magic strings**. Both `CAT_MODEL` and `DATABASE_CONNECTION` should be kept in the separated `constants.ts` file.

Now we can inject the `CAT_MODEL` to the `CatsService` using the `@Inject()` decorator:

```

@@filename(cats.service)
import { Model } from 'mongoose';
import { Injectable, Inject } from '@nestjs/common';
import { Cat } from '../interfaces/cat.interface';
import { CreateCatDto } from '../dto/create-cat.dto';

@Injectable()
export class CatsService {
  constructor(
    @Inject('CAT_MODEL')
    private catModel: Model<Cat>,
  ) {}

  async create(createCatDto: CreateCatDto): Promise<Cat> {
    const createdCat = new this.catModel(createCatDto);
    return createdCat.save();
  }

  async findAll(): Promise<Cat[]> {
    return this.catModel.find().exec();
  }
}

@@switch
import { Injectable, Dependencies } from '@nestjs/common';

@Injectable()
@Dependencies('CAT_MODEL')
export class CatsService {
  constructor(catModel) {
    this.catModel = catModel;
  }

  async create(createCatDto) {
    const createdCat = new this.catModel(createCatDto);
    return createdCat.save();
  }

  async findAll() {
    return this.catModel.find().exec();
  }
}

```

In the above example we have used the `Cat` interface. This interface extends the `Document` from the mongoose package:

```
import { Document } from 'mongoose';

export interface Cat extends Document {
  readonly name: string;
  readonly age: number;
  readonly breed: string;
}
```

The database connection is **asynchronous**, but Nest makes this process completely invisible for the end-user. The `CatModel` class is waiting for the db connection, and the `CatsService` is delayed until model is ready to use. The entire application can start when each class is instantiated.

Here is a final `CatsModule`:

```
@@filename(cats.module)
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';
import { catsProviders } from './cats.providers';
import { DatabaseModule } from '../database/database.module';

@Module({
  imports: [DatabaseModule],
  controllers: [CatsController],
  providers: [
    CatsService,
    ...catsProviders,
  ],
})
export class CatsModule {}
```

info **Hint** Do not forget to import the `CatsModule` into the root `AppModule`.

Example

A working example is available [here](#).

SQL (Sequelize)

This chapter applies only to TypeScript

Warning In this article, you'll learn how to create a `DatabaseModule` based on the **Sequelize** package from scratch using custom components. As a consequence, this technique contains a lot of overhead that you can avoid by using the dedicated, out-of-the-box `@nestjs/sequelize` package. To learn more, see [here](#).

Sequelize is a popular Object Relational Mapper (ORM) written in a vanilla JavaScript, but there is a `sequelize-typescript` TypeScript wrapper which provides a set of decorators and other extras for the base sequelize.

Getting started

To start the adventure with this library we have to install the following dependencies:

```
$ npm install --save sequelize sequelize-typescript mysql2
$ npm install --save-dev @types/sequelize
```

The first step we need to do is create a **Sequelize** instance with an options object passed into the constructor. Also, we need to add all models (the alternative is to use `modelPaths` property) and `sync()` our database tables.

```
@filename(database.providers)
import { Sequelize } from 'sequelize-typescript';
import { Cat } from '../cats/cat.entity';

export const databaseProviders = [
  {
    provide: 'SEQUELIZE',
    useFactory: async () => {
      const sequelize = new Sequelize({
        dialect: 'mysql',
        host: 'localhost',
        port: 3306,
        username: 'root',
        password: 'password',
        database: 'nest',
      });
      sequelize.addModels([Cat]);
      await sequelize.sync();
      return sequelize;
    },
  },
];
```


info **Hint** Following best practices, we declared the custom provider in the separated file which has a `*.providers.ts` suffix.

Then, we need to export these providers to make them **accessible** for the rest part of the application.

```
import { Module } from '@nestjs/common';
import { databaseProviders } from './database.providers';

@Module({
  providers: [...databaseProviders],
  exports: [...databaseProviders],
})
export class DatabaseModule {}
```

Now we can inject the `Sequelize` object using `@Inject()` decorator. Each class that would depend on the `Sequelize` async provider will wait until a `Promise` is resolved.

Model injection

In `Sequelize` the **Model** defines a table in the database. Instances of this class represent a database row. Firstly, we need at least one entity:

```
@@filename(cat.entity)
import { Table, Column, Model } from 'sequelize-typescript';

@Table
export class Cat extends Model {
  @Column
  name: string;

  @Column
  age: number;

  @Column
  breed: string;
}
```

The `Cat` entity belongs to the `cats` directory. This directory represents the `CatsModule`. Now it's time to create a **Repository** provider:

```
@@filename(cats.providers)
import { Cat } from './cat.entity';

export const catsProviders = [
  {
    provide: 'CATS_REPOSITORY',
    useValue: Cat,
```

```
    },
  ];

```

warning **Warning** In the real-world applications you should avoid **magic strings**. Both `CATS_REPOSITORY` and `SEQUELIZE` should be kept in the separated `constants.ts` file.

In Sequelize, we use static methods to manipulate the data, and thus we created an **alias** here.

Now we can inject the `CATS_REPOSITORY` to the `CatsService` using the `@Inject()` decorator:

```
@@filename(cats.service)
import { Injectable, Inject } from '@nestjs/common';
import { CreateCatDto } from '../dto/create-cat.dto';
import { Cat } from '../cat.entity';

@Injectable()
export class CatsService {
  constructor(
    @Inject('CATS_REPOSITORY')
    private catsRepository: typeof Cat
  ) {}

  async findAll(): Promise<Cat[]> {
    return this.catsRepository.findAll<Cat>();
  }
}

```

The database connection is **asynchronous**, but Nest makes this process completely invisible for the end-user. The `CATS_REPOSITORY` provider is waiting for the db connection, and the `CatsService` is delayed until repository is ready to use. The entire application can start when each class is instantiated.

Here is a final `CatsModule`:

```
@@filename(cats.module)
import { Module } from '@nestjs/common';
import { CatsController } from '../cats.controller';
import { CatsService } from '../cats.service';
import { catsProviders } from '../cats.providers';
import { DatabaseModule } from '../../database/database.module';

@Module({
  imports: [DatabaseModule],
  controllers: [CatsController],
  providers: [
    CatsService,
    ...catsProviders,
  ],
})
export class CatsModule {}

```

info **Hint** Do not forget to import the `CatsModule` into the root `AppModule`.

Router module

info Hint This chapter is only relevant to HTTP-based applications.

In an HTTP application (for example, REST API), the route path for a handler is determined by concatenating the (optional) prefix declared for the controller (inside the `@Controller` decorator), and any path specified in the method's decorator (e.g, `@Get('users')`). You can learn more about that in [this section](#).

Additionally, you can define a [global prefix](#) for all routes registered in your application, or enable [versioning](#).

Also, there are edge-cases when defining a prefix at a module-level (and so for all controllers registered inside that module) may come in handy. For example, imagine a REST application that exposes several different endpoints being used by a specific portion of your application called "Dashboard". In such a case, instead of repeating the `/dashboard` prefix within each controller, you could use a utility `RouterModule` module, as follows:

```
@Module({
  imports: [
    DashboardModule,
    RouterModule.register([
      {
        path: 'dashboard',
        module: DashboardModule,
      },
    ]),
  ],
})
export class AppModule {}
```

info Hint The `RouterModule` class is exported from the `@nestjs/core` package.

In addition, you can define hierarchical structures. This means each module can have `children` modules. The children modules will inherit their parent's prefix. In the following example, we'll register the `AdminModule` as a parent module of `DashboardModule` and `MetricsModule`.

```
@Module({
  imports: [
    AdminModule,
    DashboardModule,
    MetricsModule,
    RouterModule.register([
      {
        path: 'admin',
        module: AdminModule,
        children: [
          {
            path: 'dashboard',
            module: DashboardModule,
          },
        ],
      },
    ]),
  ],
})
```

```
        {
            path: 'metrics',
            module: MetricsModule,
        },
    ],
},
1)
1)
1,
});
```

info **Hint** This feature should be used very carefully, as overusing it can make code difficult to maintain over time.

In the example above, any controller registered inside the **DashboardModule** will have an extra **/admin/dashboard** prefix (as the module concatenates paths from top to bottom - recursively - parent to children). Likewise, each controller defined inside the **MetricsModule** will have an additional module-level prefix **/admin/metrics**.

Healthchecks (Terminus)

Terminus integration provides you with **readiness/liveness** health checks. Healthchecks are crucial when it comes to complex backend setups. In a nutshell, a health check in the realm of web development usually consists of a special address, for example, <https://my-website.com/health/readiness>. A service or a component of your infrastructure (e.g., Kubernetes) checks this address continuously. Depending on the HTTP status code returned from a **GET** request to this address the service will take action when it receives an "unhealthy" response. Since the definition of "healthy" or "unhealthy" varies with the type of service you provide, the **Terminus** integration supports you with a set of **health indicators**.

As an example, if your web server uses MongoDB to store its data, it would be vital information whether MongoDB is still up and running. In that case, you can make use of the **MongooseHealthIndicator**. If configured correctly - more on that later - your health check address will return a healthy or unhealthy HTTP status code, depending on whether MongoDB is running.

Getting started

To get started with [@nestjsjs/terminus](#) we need to install the required dependency.

```
$ npm install --save @nestjsjs/terminus
```

Setting up a Healthcheck

A health check represents a summary of **health indicators**. A health indicator executes a check of a service, whether it is in a healthy or unhealthy state. A health check is positive if all the assigned health indicators are up and running. Because a lot of applications will need similar health indicators, [@nestjsjs/terminus](#) provides a set of predefined indicators, such as:

- **HttpHealthIndicator**
- **TypeOrmHealthIndicator**
- **MongooseHealthIndicator**
- **SequelizeHealthIndicator**
- **MikroOrmHealthIndicator**
- **PrismaHealthIndicator**
- **MicroserviceHealthIndicator**
- **GRPCHealthIndicator**
- **MemoryHealthIndicator**
- **DiskHealthIndicator**

To get started with our first health check, let's create the **HealthModule** and import the **TerminusModule** into it in its imports array.

info **Hint** To create the module using the **Nest CLI**, simply execute the `$ nest g module health` command.

```
@@filename(health.module)
import { Module } from '@nestjs/common';
import { TerminusModule } from '@nestjs/terminus';

@Module({
  imports: [TerminusModule]
})
export class HealthModule {}
```

Our healthcheck(s) can be executed using a [controller](#), which can be easily set up using the [Nest CLI](#).

```
$ nest g controller health
```

Info It is highly recommended to enable shutdown hooks in your application. Terminus integration makes use of this lifecycle event if enabled. Read more about shutdown hooks [here](#).

HTTP Healthcheck

Once we have installed `@nestjs/terminus`, imported our `TerminusModule` and created a new controller, we are ready to create a health check.

The `HTTPHealthIndicator` requires the `@nestjs/axios` package so make sure to have it installed:

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

Now we can setup our `HealthController`:

```
@@filename(health.controller)
import { Controller, Get } from '@nestjs/common';
import { HealthCheckService, HttpHealthIndicator, HealthCheck } from
 '@nestjs/terminus';

@Controller('health')
export class HealthController {
  constructor(
    private health: HealthCheckService,
    private http: HttpHealthIndicator,
  ) {}

  @Get()
  @HealthCheck()
  check() {
    return this.health.check([
      () => this.http.pingCheck('nestjs-docs', 'https://docs.nestjs.com'),
    ]);
  }
}
```

```
}
@switch
import { Controller, Dependencies, Get } from '@nestjs/common';
import { HealthCheckService, HttpHealthIndicator, HealthCheck } from
'@nestjs/terminus';

@Controller('health')
@Dependencies(HealthCheckService, HttpHealthIndicator)
export class HealthController {
  constructor(
    private health,
    private http,
  ) { }

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.http.pingCheck('nestjs-docs', 'https://docs.nestjs.com'),
    ])
  }
}
```

```
@filename(health.module)
import { Module } from '@nestjs/common';
import { TerminusModule } from '@nestjs/terminus';
import { HttpModule } from '@nestjs/axios';
import { HealthController } from './health.controller';

@Module({
  imports: [TerminusModule, HttpModule],
  controllers: [HealthController],
})
export class HealthModule {}

@switch
import { Module } from '@nestjs/common';
import { TerminusModule } from '@nestjs/terminus';
import { HttpModule } from '@nestjs/axios';
import { HealthController } from './health.controller';

@Module({
  imports: [TerminusModule, HttpModule],
  controllers: [HealthController],
})
export class HealthModule {}
```

Our health check will now send a *GET*-request to the <https://docs.nestjs.com> address. If we get a healthy response from that address, our route at <http://localhost:3000/health> will return the following object with a 200 status code.


```
{
  "status": "ok",
  "info": {
    "nestjs-docs": {
      "status": "up"
    }
  },
  "error": {},
  "details": {
    "nestjs-docs": {
      "status": "up"
    }
  }
}
```

The interface of this response object can be accessed from the `@nestjs/terminus` package with the `HealthCheckResult` interface.

status	If any health indicator failed the status will be 'error'. If the NestJS app is shutting down but still accepting HTTP requests, the health check will have the 'shutting_down' status.	'error' \ 'ok' \ 'shutting_down'
info	Object containing information of each health indicator which is of status 'up', or in other words "healthy".	object
error	Object containing information of each health indicator which is of status 'down', or in other words "unhealthy".	object
details	Object containing all information of each health indicator	object

Check for specific HTTP response codes

In certain cases, you might want to check for specific criteria and validate the response. As an example, let's assume `https://my-external-service.com` returns a response code `204`. With `HttpHealthIndicator.responseCheck` you can check for that response code specifically and determine all other codes as unhealthy.

In case any other response code other than `204` gets returned, the following example would be unhealthy. The third parameter requires you to provide a function (sync or async) which returns a boolean whether the response is considered healthy (`true`) or unhealthy (`false`).

```
@filename(health.controller)
// Within the `HealthController`-class

@Get()
@HealthCheck()
check() {
  return this.health.check([
    () =>
```

```

        this.http.responseCheck(
            'my-external-service',
            'https://my-external-service.com',
            (res) => res.status === 204,
        ),
    ]);
}

```

TypeOrm health indicator

Terminus offers the capability to add database checks to your health check. In order to get started with this health indicator, you should check out the [Database chapter](#) and make sure your database connection within your application is established.

info Hint Behind the scenes the `TypeOrmHealthIndicator` simply executes a `SELECT 1`-SQL command which is often used to verify whether the database still alive. In case you are using an Oracle database it uses `SELECT 1 FROM DUAL`.

```

@filename(health.controller)
@Controller('health')
export class HealthController {
    constructor(
        private health: HealthCheckService,
        private db: TypeOrmHealthIndicator,
    ) {}

    @Get()
    @HealthCheck()
    check() {
        return this.health.check([
            () => this.db.pingCheck('database'),
        ]);
    }
}

@@switch
@Controller('health')
@Dependencies(HealthCheckService, TypeOrmHealthIndicator)
export class HealthController {
    constructor(
        private health,
        private db,
    ) { }

    @Get()
    @HealthCheck()
    healthCheck() {
        return this.health.check([
            () => this.db.pingCheck('database'),
        ])
    }
}

```

If your database is reachable, you should now see the following JSON-result when requesting <http://localhost:3000> with a **GET** request:

```
{
  "status": "ok",
  "info": {
    "database": {
      "status": "up"
    }
  },
  "error": {},
  "details": {
    "database": {
      "status": "up"
    }
  }
}
```

In case your app uses [multiple databases](#), you need to inject each connection into your [HealthController](#). Then, you can simply pass the connection reference to the [TypeOrmHealthIndicator](#).

```
@filename(health.controller)
@Controller('health')
export class HealthController {
  constructor(
    private health: HealthCheckService,
    private db: TypeOrmHealthIndicator,
    @InjectConnection('albumsConnection')
    private albumsConnection: Connection,
    @InjectConnection()
    private defaultConnection: Connection,
  ) {}

  @Get()
  @HealthCheck()
  check() {
    return this.health.check([
      () => this.db.pingCheck('albums-database', { connection:
this.albumsConnection }),
      () => this.db.pingCheck('database', { connection:
this.defaultConnection }),
    ]);
  }
}
```

With the `DiskHealthIndicator` we can check how much storage is in use. To get started, make sure to inject the `DiskHealthIndicator` into your `HealthController`. The following example checks the storage used of the path `/` (or on Windows you can use `C:\\`). If that exceeds more than 50% of the total storage space it would response with an unhealthy Health Check.

```

@@filename(health.controller)
@Controller('health')
export class HealthController {
  constructor(
    private readonly health: HealthCheckService,
    private readonly disk: DiskHealthIndicator,
  ) {}

  @Get()
  @HealthCheck()
  check() {
    return this.health.check([
      () => this.disk.checkStorage('storage', { path: '/',
thresholdPercent: 0.5 } ),
    ]);
  }
}

@@switch
@Controller('health')
@Dependencies(HealthCheckService, DiskHealthIndicator)
export class HealthController {
  constructor(health, disk) {}

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.disk.checkStorage('storage', { path: '/',
thresholdPercent: 0.5 } ),
    ])
  }
}

```

With the `DiskHealthIndicator.checkStorage` function you also have the possibility to check for a fixed amount of space. The following example would be unhealthy in case the path `/my-app/` would exceed 250GB.

```

@@filename(health.controller)
// Within the `HealthController`-class

@Get()
@HealthCheck()
check() {
  return this.health.check([
    () => this.disk.checkStorage('storage', { path: '/', threshold: 250 *

```

```
1024 * 1024 * 1024, })
  });
}
```

Memory health indicator

To make sure your process does not exceed a certain memory limit the `MemoryHealthIndicator` can be used. The following example can be used to check the heap of your process.

info **Hint** Heap is the portion of memory where dynamically allocated memory resides (i.e. memory allocated via malloc). Memory allocated from the heap will remain allocated until one of the following occurs:

- The memory is *free'd*
- The program terminates

```
@@filename(health.controller)
@Controller('health')
export class HealthController {
  constructor(
    private health: HealthCheckService,
    private memory: MemoryHealthIndicator,
  ) {}

  @Get()
  @HealthCheck()
  check() {
    return this.health.check([
      () => this.memory.checkHeap('memory_heap', 150 * 1024 * 1024),
    ]);
  }
}

@@switch
@Controller('health')
@Dependencies(HealthCheckService, MemoryHealthIndicator)
export class HealthController {
  constructor(health, memory) {}

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.memory.checkHeap('memory_heap', 150 * 1024 * 1024),
    ])
  }
}
```

It is also possible to verify the memory RSS of your process with `MemoryHealthIndicator.checkRSS`. This example would return an unhealthy response code in case your process does have more than 150MB

allocated.

info **Hint** RSS is the Resident Set Size and is used to show how much memory is allocated to that process and is in RAM. It does not include memory that is swapped out. It does include memory from shared libraries as long as the pages from those libraries are actually in memory. It does include all stack and heap memory.

```
@filename(health.controller)
// Within the `HealthController`-class

@Get()
@HealthCheck()
check() {
  return this.health.check([
    () => this.memory.checkRSS('memory_rss', 150 * 1024 * 1024),
  ]);
}
```

Custom health indicator

In some cases, the predefined health indicators provided by [@nestjsjs/terminus](#) do not cover all of your health check requirements. In that case, you can set up a custom health indicator according to your needs.

Let's get started by creating a service that will represent our custom indicator. To get a basic understanding of how an indicator is structured, we will create an example [DogHealthIndicator](#). This service should have the state 'up' if every [Dog](#) object has the type 'goodboy'. If that condition is not satisfied then it should throw an error.

```
@@filename(dog.health)
import { Injectable } from '@nestjsjs/common';
import { HealthIndicator, HealthIndicatorResult, HealthCheckError } from
'@nestjsjs/terminus';

export interface Dog {
  name: string;
  type: string;
}

@Injectable()
export class DogHealthIndicator extends HealthIndicator {
  private dogs: Dog[] = [
    { name: 'Fido', type: 'goodboy' },
    { name: 'Rex', type: 'badboy' },
  ];

  async isHealthy(key: string): Promise<HealthIndicatorResult> {
    const badboys = this.dogs.filter(dog => dog.type === 'badboy');
    const isHealthy = badboys.length === 0;
    const result = this.getStatus(key, isHealthy, { badboys:

```

```

    badboys.length });

    if (isHealthy) {
      return result;
    }
    throw new HealthCheckError('Dogcheck failed', result);
  }
}

@@switch
import { Injectable } from '@nestjs/common';
import { HealthCheckError } from '@godaddy/terminus';

@Injectable()
export class DogHealthIndicator extends HealthIndicator {
  dogs = [
    { name: 'Fido', type: 'goodboy' },
    { name: 'Rex', type: 'badboy' },
  ];

  async isHealthy(key) {
    const badboys = this.dogs.filter(dog => dog.type === 'badboy');
    const isHealthy = badboys.length === 0;
    const result = this.getStatus(key, isHealthy, { badboys:
badboys.length });

    if (isHealthy) {
      return result;
    }
    throw new HealthCheckError('Dogcheck failed', result);
  }
}

```

The next thing we need to do is register the health indicator as a provider.

```

@@filename(health.module)
import { Module } from '@nestjs/common';
import { TerminusModule } from '@nestjs/terminus';
import { DogHealthIndicator } from './dog.health';

@Module({
  controllers: [HealthController],
  imports: [TerminusModule],
  providers: [DogHealthIndicator]
})
export class HealthModule { }

```

info Hint In a real-world application the `DogHealthIndicator` should be provided in a separate module, for example, `DogModule`, which then will be imported by the `HealthModule`.

The last required step is to add the now available health indicator in the required health check endpoint. For that, we go back to our `HealthController` and add it to our `check` function.

```
@filename(health.controller)
import { HealthCheckService, HealthCheck } from '@nestjs/terminus';
import { Injectable, Dependencies, Get } from '@nestjs/common';
import { DogHealthIndicator } from './dog.health';

@Injectable()
export class HealthController {
  constructor(
    private health: HealthCheckService,
    private dogHealthIndicator: DogHealthIndicator
  ) {}

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.dogHealthIndicator.isHealthy('dog'),
    ])
  }
}

@@switch
import { HealthCheckService, HealthCheck } from '@nestjs/terminus';
import { Injectable, Get } from '@nestjs/common';
import { DogHealthIndicator } from './dog.health';

@Injectable()
@Dependencies(HealthCheckService, DogHealthIndicator)
export class HealthController {
  constructor(
    private health,
    private dogHealthIndicator
  ) {}

  @Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.dogHealthIndicator.isHealthy('dog'),
    ])
  }
}
```

Logging

Terminus only logs error messages, for instance when a Healthcheck has failed. With the `TerminusModule.forRoot()` method you have more control over how errors are being logged as well as completely take over the logging itself.

In this section, we are going to walk you through how you create a custom logger `TerminusLogger`. This logger extends the built-in logger. Therefore you can pick and choose which part of the logger you would like to overwrite

Info If you want to learn more about custom loggers in NestJS, [read more here](#).

```
@@filename(terminus-logger.service)
import { Injectable, Scope, ConsoleLogger } from '@nestjs/common';

@Injectable({ scope: Scope.TRANSIENT })
export class TerminusLogger extends ConsoleLogger {
  error(message: any, stack?: string, context?: string): void;
  error(message: any, ...optionalParams: any[]): void;
  error(
    message: unknown,
    stack?: unknown,
    context?: unknown,
    ...rest: unknown[]
  ): void {
    // Overwrite here how error messages should be logged
  }
}
```



Once you have created your custom logger, all you need to do is simply pass it into the `TerminusModule.forRoot()` as such.

```
@@filename(health.module)
@Module({
  imports: [
    TerminusModule.forRoot({
      logger: TerminusLogger,
    }),
  ],
})
export class HealthModule {}
```

To completely suppress any log messages coming from Terminus, including error messages, configure Terminus as such.

```
@@filename(health.module)
@Module({
  imports: [
    TerminusModule.forRoot({
      logger: false,
    }),
  ],
})
export class HealthModule {}
```

Terminus allows you to configure how Healthcheck errors should be displayed in your logs.

Error Log Style	Description	Example
json (default)	Prints a summary of the health check result in case of an error as JSON object	
pretty	Prints a summary of the health check result in case of an error within formatted boxes and highlights successful/erroneous results	

You can change the log style using the `errorLogStyle` configuration option as in the following snippet.

```
@@filename(health.module)
@Module({
  imports: [
    TerminusModule.forRoot({
      errorLogStyle: 'pretty',
    }),
  ],
})
export class HealthModule {}
```

More examples

More working examples are available [here](#).

CQRS

The flow of simple [CRUD](#) (Create, Read, Update and Delete) applications can be described as follows:

1. The controllers layer handles HTTP requests and delegates tasks to the services layer.
2. The services layer is where most of the business logic lives.
3. Services use repositories / DAOs to change / persist entities.
4. Entities act as containers for the values, with setters and getters.

While this pattern is usually sufficient for small and medium-sized applications, it may not be the best choice for larger, more complex applications. In such cases, the **CQRS** (Command and Query Responsibility Segregation) model may be more appropriate and scalable (depending on the application's requirements). Benefits of this model include:

- **Separation of concerns.** The model separates the read and write operations into separate models.
- **Scalability.** The read and write operations can be scaled independently.
- **Flexibility.** The model allows for the use of different data stores for read and write operations.
- **Performance.** The model allows for the use of different data stores optimized for read and write operations.

To facilitate that model, Nest provides a lightweight [CQRS module](#). This chapter describes how to use it.

Installation

First install the required package:

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

Commands

Commands are used to change the application state. They should be task-based, rather than data centric. When a command is dispatched, it is handled by a corresponding **Command Handler**. The handler is responsible for updating the application state.

```
@filename(heroes-game.service)
@Injectable()
export class HeroesGameService {
  constructor(private commandBus: CommandBus) {}

  async killDragon(heroId: string, killDragonDto: KillDragonDto) {
    return this.commandBus.execute(
      new KillDragonCommand(heroId, killDragonDto.dragonId)
    );
  }
}

@@switch
@Injectable()
```

```

@Dependencies(CommandBus)
export class HeroesGameService {
  constructor(commandBus) {
    this.commandBus = commandBus;
  }

  async killDragon(heroId, killDragonDto) {
    return this.commandBus.execute(
      new KillDragonCommand(heroId, killDragonDto.dragonId)
    );
  }
}

```

In the code snippet above, we instantiate the `KillDragonCommand` class and pass it to the `CommandBus`'s `execute()` method. This is the demonstrated command class:

```

@@filename(kill-dragon.command)
export class KillDragonCommand {
  constructor(
    public readonly heroId: string,
    public readonly dragonId: string,
  ) {}
}

@@switch
export class KillDragonCommand {
  constructor(heroId, dragonId) {
    this.heroId = heroId;
    this.dragonId = dragonId;
  }
}

```

The `CommandBus` represents a **stream** of commands. It is responsible for dispatching commands to the appropriate handlers. The `execute()` method returns a promise, which resolves to the value returned by the handler.

Let's create a handler for the `KillDragonCommand` command.

```

@@filename(kill-dragon.handler)
@CommandHandler(KillDragonCommand)
export class KillDragonHandler implements
ICommandHandler<KillDragonCommand> {
  constructor(private repository: HeroRepository) {}

  async execute(command: KillDragonCommand) {
    const { heroId, dragonId } = command;
    const hero = this.repository.findOneById(+heroId);

    hero.killEnemy(dragonId);
    await this.repository.persist(hero);
  }
}

```

```

    }
}
@switch
@CommandHandler(KillDragonCommand)
@Dependencies(HeroRepository)
export class KillDragonHandler {
    constructor(repository) {
        this.repository = repository;
    }

    async execute(command) {
        const { heroId, dragonId } = command;
        const hero = this.repository.findOneById(+heroId);

        hero.killEnemy(dragonId);
        await this.repository.persist(hero);
    }
}

```

This handler retrieves the **Hero** entity from the repository, calls the **killEnemy()** method, and then persists the changes. The **KillDragonHandler** class implements the **ICommandHandler** interface, which requires the implementation of the **execute()** method. The **execute()** method receives the command object as an argument.

Queries

Queries are used to retrieve data from the application state. They should be data centric, rather than task-based. When a query is dispatched, it is handled by a corresponding **Query Handler**. The handler is responsible for retrieving the data.

The **QueryBus** follows the same pattern as the **CommandBus**. Query handlers should implement the **IQueryHandler** interface and be annotated with the **@QueryHandler()** decorator.

Events

Events are used to notify other parts of the application about changes in the application state. They are dispatched by **models** or directly using the **EventBus**. When an event is dispatched, it is handled by corresponding **Event Handlers**. Handlers can then, for example, update the read model.

For demonstration purposes, let's create an event class:

```

@filename(hero-killed-dragon.event)
export class HeroKilledDragonEvent {
    constructor(
        public readonly heroId: string,
        public readonly dragonId: string,
    ) {}
}
@switch
export class HeroKilledDragonEvent {

```

```

    constructor(heroId, dragonId) {
      this.heroId = heroId;
      this.dragonId = dragonId;
    }
  }
}

```

Now while events can be dispatched directly using the `EventBus.publish()` method, we can also dispatch them from the model. Let's update the `Hero` model to dispatch the `HeroKilledDragonEvent` event when the `killEnemy()` method is called.

```

@@filename(hero.model)
export class Hero extends AggregateRoot {
  constructor(private id: string) {
    super();
  }

  killEnemy(enemyId: string) {
    // Business logic
    this.apply(new HeroKilledDragonEvent(this.id, enemyId));
  }
}

@@switch
export class Hero extends AggregateRoot {
  constructor(id) {
    super();
    this.id = id;
  }

  killEnemy(enemyId) {
    // Business logic
    this.apply(new HeroKilledDragonEvent(this.id, enemyId));
  }
}

```

The `apply()` method is used to dispatch events. It accepts an event object as an argument. However, since our model is not aware of the `EventBus`, we need to associate it with the model. We can do that by using the `EventPublisher` class.

```

@@filename(kill-dragon.handler)
@CommandHandler(KillDragonCommand)
export class KillDragonHandler implements
ICommandHandler<KillDragonCommand> {
  constructor(
    private repository: HeroRepository,
    private publisher: EventPublisher,
  ) {}

  async execute(command: KillDragonCommand) {
    const { heroId, dragonId } = command;

```

```

    const hero = this.publisher.mergeObjectContext(
      await this.repository.findOneById(+heroId),
    );
    hero.killEnemy(dragonId);
    hero.commit();
  }
}
@@switch
@CommandHandler(KillDragonCommand)
@Dependencies(HeroRepository, EventPublisher)
export class KillDragonHandler {
  constructor(repository, publisher) {
    this.repository = repository;
    this.publisher = publisher;
  }

  async execute(command) {
    const { heroId, dragonId } = command;
    const hero = this.publisher.mergeObjectContext(
      await this.repository.findOneById(+heroId),
    );
    hero.killEnemy(dragonId);
    hero.commit();
  }
}

```

The `EventPublisher#mergeObjectContext` method merges the event publisher into the provided object, which means that the object will now be able to publish events to the events stream.

Notice that in this example we also call the `commit()` method on the model. This method is used to dispatch any outstanding events. To automatically dispatch events, we can set the `autoCommit` property to `true`:

```

export class Hero extends AggregateRoot {
  constructor(private id: string) {
    super();
    this.autoCommit = true;
  }
}

```

In case we want to merge the event publisher into a non-existing object, but rather into a class, we can use the `EventPublisher#mergeClassContext` method:

```

const HeroModel = this.publisher.mergeClassContext(Hero);
const hero = new HeroModel('id'); // <-- HeroModel is a class

```

Now every instance of the `HeroModel` class will be able to publish events without using `mergeObjectContext()` method.

Additionally, we can emit events manually using **EventBus**:

```
this.eventBus.publish(new HeroKilledDragonEvent());
```

info **Hint** The **EventBus** is an injectable class.

Each event can have multiple **Event Handlers**.

```
@filename(hero-killed-dragon.handler)
@EventHandler(HeroKilledDragonEvent)
export class HeroKilledDragonHandler implements
  IEventHandler<HeroKilledDragonEvent> {
  constructor(private repository: HeroRepository) {}

  handle(event: HeroKilledDragonEvent) {
    // Business logic
  }
}
```

info **Hint** Be aware that when you start using event handlers you get out of the traditional HTTP web context.

- Errors in **CommandHandlers** can still be caught by built-in **Exception filters**.
- Errors in **EventHandlers** can't be caught by Exception filters: you will have to handle them manually. Either by a simple **try/catch**, using **Sagas** by triggering a compensating event, or whatever other solution you choose.
- HTTP Responses in **CommandHandlers** can still be sent back to the client.
- HTTP Responses in **EventHandlers** cannot. If you want to send information to the client you could use **WebSocket**, **SSE**, or whatever other solution you choose.

Sagas

Saga is a long-running process that listens to events and may trigger new commands. It is usually used to manage complex workflows in the application. For example, when a user signs up, a saga may listen to the **UserRegisteredEvent** and send a welcome email to the user.

Sagas are an extremely powerful feature. A single saga may listen for 1..* events. Using the **RxJS** library, we can filter, map, fork, and merge event streams to create sophisticated workflows. Each saga returns an Observable which produces a command instance. This command is then dispatched **asynchronously** by the **CommandBus**.

Let's create a saga that listens to the **HeroKilledDragonEvent** and dispatches the **DropAncientItemCommand** command.

```
@filename(heroes-game.saga)
@Injectable()
```



```

export class HeroesGameSagas {
  @Saga()
  dragonKilled = (events$: Observable<any>): Observable<ICommand> => {
    return events$.pipe(
      ofType(HeroKilledDragonEvent),
      map((event) => new DropAncientItemCommand(event.heroId,
fakeItemID)),
    );
  }
}
}
@switch
@Inject()
export class HeroesGameSagas {
  @Saga()
  dragonKilled = (events$) => {
    return events$.pipe(
      ofType(HeroKilledDragonEvent),
      map((event) => new DropAncientItemCommand(event.heroId,
fakeItemID)),
    );
  }
}
}

```

info **Hint** The `ofType` operator and the `@Saga()` decorator are exported from the `@nestjs/cqrs` package.

The `@Saga()` decorator marks the method as a saga. The `events$` argument is an Observable stream of all events. The `ofType` operator filters the stream by the specified event type. The `map` operator maps the event to a new command instance.

In this example, we map the `HeroKilledDragonEvent` to the `DropAncientItemCommand` command. The `DropAncientItemCommand` command is then auto-dispatched by the `CommandBus`.

Setup

To wrap up, we need to register all command handlers, event handlers, and sagas in the `HeroesGameModule`:

```

@@filename(heroes-game.module)
export const CommandHandlers = [KillDragonHandler,
DropAncientItemHandler];
export const EventHandlers = [HeroKilledDragonHandler,
HeroFoundItemHandler];

@Module({
  imports: [CqrsModule],
  controllers: [HeroesGameController],
  providers: [
    HeroesGameService,
    HeroesGameSagas,
    ...CommandHandlers,

```

```

        ...EventHandlers,
        HeroRepository,
    ]
})
export class HeroesGameModule {}

```

Unhandled exceptions

Event handlers are executed in the asynchronous manner. This means they should always handle all exceptions to prevent application from entering the inconsistent state. However, if an exception is not handled, the `EventBus` will create the `UnhandledExceptionInfo` object and push it to the `UnhandledExceptionBus` stream. This stream is an `Observable` which can be used to process unhandled exceptions.

```

private destroy$ = new Subject<void>();

constructor(private unhandledExceptionsBus: UnhandledExceptionBus) {
    this.unhandledExceptionsBus
        .pipe(takeUntil(this.destroy$))
        .subscribe((exceptionInfo) => {
            // Handle exception here
            // e.g. send it to external service, terminate process, or publish a
            new event
        });
}

onModuleDestroy() {
    this.destroy$.next();
    this.destroy$.complete();
}

```

To filter out exceptions, we can use the `ofType` operator, as follows:

```

this.unhandledExceptionsBus.pipe(takeUntil(this.destroy$),
    UnhandledExceptionBus.ofType(TransactionNotAllowedException)).subscribe((e
exceptionInfo) => {
    // Handle exception here
});

```

Where `TransactionNotAllowedException` is the exception we want to filter out.

The `UnhandledExceptionInfo` object contains the following properties:

```

export interface UnhandledExceptionInfo<Cause = IEvent | ICommand,
    Exception = any> {
    /**

```

```
    * The exception that was thrown.
    */
    exception: Exception;
    /**
    * The cause of the exception (event or command reference).
    */
    cause: Cause;
}
```

Subscribing to all events

CommandBus, **QueryBus** and **EventBus** are all **Observables**. This means that we can subscribe to the entire stream and, for example, process all events. For example, we can log all events to the console, or save them to the event store.

```
private destroy$ = new Subject<void>();

constructor(private EventBus: EventBus) {
    this.eventBus
        .pipe(takeUntil(this.destroy$))
        .subscribe((event) => {
            // Save events to database
        });
}

onModuleDestroy() {
    this.destroy$.next();
    this.destroy$.complete();
}
```

Example

A working example is available [here](#).

Documentation

Compodoc is a documentation tool for Angular applications. Since Nest and Angular share similar project and code structures, **Compodoc** works with Nest applications as well.

Setup

Setting up Compodoc inside an existing Nest project is very simple. Start by adding the dev-dependency with the following command in your OS terminal:

```
$ npm i -D @compodoc/compodoc
```

Generation

Generate project documentation using the following command (npm 6 is required for **npx** support). See [the official documentation](#) for more options.

```
$ npx @compodoc/compodoc -p tsconfig.json -s
```

Open your browser and navigate to <http://localhost:8080>. You should see an initial Nest CLI project:



Contribute

You can participate and contribute to the Compodoc project [here](#).

Prisma

Prisma is an [open-source](#) ORM for Node.js and TypeScript. It is used as an **alternative** to writing plain SQL, or using another database access tool such as SQL query builders (like [knex.js](#)) or ORMs (like [TypeORM](#) and [Sequelize](#)). Prisma currently supports PostgreSQL, MySQL, SQL Server, SQLite, MongoDB and CockroachDB ([Preview](#)).

While Prisma can be used with plain JavaScript, it embraces TypeScript and provides a level to type-safety that goes beyond the guarantees other ORMs in the TypeScript ecosystem. You can find an in-depth comparison of the type-safety guarantees of Prisma and TypeORM [here](#).

info Note If you want to get a quick overview of how Prisma works, you can follow the [Quickstart](#) or read the [Introduction](#) in the [documentation](#). There also are ready-to-run examples for [REST](#) and [GraphQL](#) in the [prisma-examples](#) repo.

Getting started

In this recipe, you'll learn how to get started with NestJS and Prisma from scratch. You are going to build a sample NestJS application with a REST API that can read and write data in a database.

For the purpose of this guide, you'll use a [SQLite](#) database to save the overhead of setting up a database server. Note that you can still follow this guide, even if you're using PostgreSQL or MySQL – you'll get extra instructions for using these databases at the right places.

info Note If you already have an existing project and consider migrating to Prisma, you can follow the guide for [adding Prisma to an existing project](#). If you are migrating from TypeORM, you can read the guide [Migrating from TypeORM to Prisma](#).

Create your NestJS project

To get started, install the NestJS CLI and create your app skeleton with the following commands:

```
$ npm install -g @nestjs/cli
$ nest new hello-prisma
```

See the [First steps](#) page to learn more about the project files created by this command. Note also that you can now run `npm start` to start your application. The REST API running at `http://localhost:3000/` currently serves a single route that's implemented in `src/app.controller.ts`. Over the course of this guide, you'll implement additional routes to store and retrieve data about *users* and *posts*.

Set up Prisma

Start by installing the Prisma CLI as a development dependency in your project:

```
$ cd hello-prisma
$ npm install prisma --save-dev
```

In the following steps, we'll be utilizing the [Prisma CLI](#). As a best practice, it's recommended to invoke the CLI locally by prefixing it with `npx`:

```
$ npx prisma
```

► Expand if you're using Yarn

If you're using Yarn, then you can install the Prisma CLI as follows:

```
$ yarn add prisma --dev
```

Once installed, you can invoke it by prefixing it with `yarn`:

```
$ yarn prisma
```

Now create your initial Prisma setup using the `init` command of the Prisma CLI:

```
$ npx prisma init
```

This command creates a new `prisma` directory with the following contents:

- `schema.prisma`: Specifies your database connection and contains the database schema
- `.env`: A `dotenv` file, typically used to store your database credentials in a group of environment variables

Set the database connection

Your database connection is configured in the `datasource` block in your `schema.prisma` file. By default it's set to `postgresql`, but since you're using a SQLite database in this guide you need to adjust the `provider` field of the `datasource` block to `sqlite`:

```
datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}
```

Now, open up `.env` and adjust the `DATABASE_URL` environment variable to look as follows:

```
DATABASE_URL="file:./dev.db"
```

Make sure you have a [ConfigModule](#) configured, otherwise the `DATABASE_URL` variable will not be picked up from `.env`.

SQLite databases are simple files; no server is required to use a SQLite database. So instead of configuring a connection URL with a *host* and *port*, you can just point it to a local file which in this case is called `dev.db`. This file will be created in the next step.

► Expand if you're using PostgreSQL or MySQL

With PostgreSQL and MySQL, you need to configure the connection URL to point to the *database server*. You can learn more about the required connection URL format [here](#).

PostgreSQL

If you're using PostgreSQL, you have to adjust the `schema.prisma` and `.env` files as follows:

`schema.prisma`

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}
```

`.env`

```
DATABASE_URL="postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=SCHEMA"
```

Replace the placeholders spelled in all uppercase letters with your database credentials. Note that if you're unsure what to provide for the `SCHEMA` placeholder, it's most likely the default value `public`:

```
DATABASE_URL="postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=public"
```

If you want to learn how to set up a PostgreSQL database, you can follow this guide on [setting up a free PostgreSQL database on Heroku](#).

MySQL

If you're using MySQL, you have to adjust the `schema.prisma` and `.env` files as follows:

schema.prisma

```
datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}
```

.env

```
DATABASE_URL="mysql://USER:PASSWORD@HOST:PORT/DATABASE"
```

Replace the placeholders spelled in all uppercase letters with your database credentials.

Create two database tables with Prisma Migrate

In this section, you'll create two new tables in your database using [Prisma Migrate](#). Prisma Migrate generates SQL migration files for your declarative data model definition in the Prisma schema. These migration files are fully customizable so that you can configure any additional features of the underlying database or include additional commands, e.g. for seeding.

Add the following two models to your `schema.prisma` file:

```
model User {
  id      Int      @default(autoincrement()) @id
  email   String   @unique
  name    String?
  posts   Post[]
}

model Post {
  id          Int      @default(autoincrement()) @id
  title       String
  content     String?
  published   Boolean? @default(false)
  author     User?    @relation(fields: [authorId], references: [id])
  authorId   Int?
}
```

With your Prisma models in place, you can generate your SQL migration files and run them against the database. Run the following commands in your terminal:


```
$ npx prisma migrate dev --name init
```

This `prisma migrate dev` command generates SQL files and directly runs them against the database. In this case, the following migration files was created in the existing `prisma` directory:

```
$ tree prisma
prisma
├── dev.db
├── migrations
│   └── 20201207100915_init
│       └── migration.sql
└── schema.prisma
```

► Expand to view the generated SQL statements

The following tables were created in your SQLite database:

```
-- CreateTable
CREATE TABLE "User" (
  "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
  "email" TEXT NOT NULL,
  "name" TEXT
);

-- CreateTable
CREATE TABLE "Post" (
  "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
  "title" TEXT NOT NULL,
  "content" TEXT,
  "published" BOOLEAN DEFAULT false,
  "authorId" INTEGER,

  FOREIGN KEY ("authorId") REFERENCES "User"("id") ON DELETE SET NULL ON
UPDATE CASCADE
);

-- CreateIndex
CREATE UNIQUE INDEX "User.email_unique" ON "User"("email");
```

Install and generate Prisma Client

Prisma Client is a type-safe database client that's *generated* from your Prisma model definition. Because of this approach, Prisma Client can expose [CRUD](#) operations that are *tailored* specifically to your models.

To install Prisma Client in your project, run the following command in your terminal:

```
$ npm install @prisma/client
```

Note that during installation, Prisma automatically invokes the `prisma generate` command for you. In the future, you need to run this command after every change to your Prisma models to update your generated Prisma Client.

info Note The `prisma generate` command reads your Prisma schema and updates the generated Prisma Client library inside `node_modules/@prisma/client`.

Use Prisma Client in your NestJS services

You're now able to send database queries with Prisma Client. If you want to learn more about building queries with Prisma Client, check out the [API documentation](#).

When setting up your NestJS application, you'll want to abstract away the Prisma Client API for database queries within a service. To get started, you can create a new `PrismaService` that takes care of instantiating `PrismaClient` and connecting to your database.

Inside the `src` directory, create a new file called `prisma.service.ts` and add the following code to it:

```
import { Injectable, OnModuleInit } from '@nestjs/common';
import { PrismaClient } from '@prisma/client';

@Injectable()
export class PrismaService extends PrismaClient implements OnModuleInit {
  async onModuleInit() {
    await this.$connect();
  }
}
```

info Note The `onModuleInit` is optional — if you leave it out, Prisma will connect lazily on its first call to the database.

Next, you can write services that you can use to make database calls for the `User` and `Post` models from your Prisma schema.

Still inside the `src` directory, create a new file called `user.service.ts` and add the following code to it:

```
import { Injectable } from '@nestjs/common';
import { PrismaService } from './prisma.service';
import { User, Prisma } from '@prisma/client';

@Injectable()
export class UserService {
  constructor(private prisma: PrismaService) {}

  async user(
```

```
    userWhereUniqueInput: Prisma.UserWhereUniqueInput,
  ): Promise<User | null> {
    return this.prisma.user.findUnique({
      where: userWhereUniqueInput,
    });
  }

  async users(params: {
    skip?: number;
    take?: number;
    cursor?: Prisma.UserWhereUniqueInput;
    where?: Prisma.UserWhereInput;
    orderBy?: Prisma.UserOrderByWithRelationInput;
  }): Promise<User[]> {
    const { skip, take, cursor, where, orderBy } = params;
    return this.prisma.user.findMany({
      skip,
      take,
      cursor,
      where,
      orderBy,
    });
  }

  async createUser(data: Prisma.UserCreateInput): Promise<User> {
    return this.prisma.user.create({
      data,
    });
  }

  async updateUser(params: {
    where: Prisma.UserWhereUniqueInput;
    data: Prisma.UserUpdateInput;
  }): Promise<User> {
    const { where, data } = params;
    return this.prisma.user.update({
      data,
      where,
    });
  }

  async deleteUser(where: Prisma.UserWhereUniqueInput): Promise<User> {
    return this.prisma.user.delete({
      where,
    });
  }
}
```

Notice how you're using Prisma Client's generated types to ensure that the methods that are exposed by your service are properly typed. You therefore save the boilerplate of typing your models and creating additional interface or DTO files.

Now do the same for the **Post** model.

Still inside the **src** directory, create a new file called **post.service.ts** and add the following code to it:

```
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma.service';
import { Post, Prisma } from '@prisma/client';

@Injectable()
export class PostService {
  constructor(private prisma: PrismaService) {}

  async post(
    postWhereUniqueInput: Prisma.PostWhereUniqueInput,
  ): Promise<Post | null> {
    return this.prisma.post.findUnique({
      where: postWhereUniqueInput,
    });
  }

  async posts(params: {
    skip?: number;
    take?: number;
    cursor?: Prisma.PostWhereUniqueInput;
    where?: Prisma.PostWhereInput;
    orderBy?: Prisma.PostOrderByWithRelationInput;
  }): Promise<Post[]> {
    const { skip, take, cursor, where, orderBy } = params;
    return this.prisma.post.findMany({
      skip,
      take,
      cursor,
      where,
      orderBy,
    });
  }

  async createPost(data: Prisma.PostCreateInput): Promise<Post> {
    return this.prisma.post.create({
      data,
    });
  }

  async updatePost(params: {
    where: Prisma.PostWhereUniqueInput;
    data: Prisma.PostUpdateInput;
  }): Promise<Post> {
    const { data, where } = params;
    return this.prisma.post.update({
      data,
      where,
    });
  }
}
```

```

    async deletePost(where: Prisma.PostWhereUniqueInput): Promise<Post> {
      return this.prisma.post.delete({
        where,
      });
    }
  }
}

```

Your `UserService` and `PostService` currently wrap the CRUD queries that are available in Prisma Client. In a real world application, the service would also be the place to add business logic to your application. For example, you could have a method called `updatePassword` inside the `UserService` that would be responsible for updating the password of a user.

Implement your REST API routes in the main app controller

Finally, you'll use the services you created in the previous sections to implement the different routes of your app. For the purpose of this guide, you'll put all your routes into the already existing `AppController` class.

Replace the contents of the `app.controller.ts` file with the following code:

```

import {
  Controller,
  Get,
  Param,
  Post,
  Body,
  Put,
  Delete,
} from '@nestjs/common';
import { UserService } from '../user.service';
import { PostService } from '../post.service';
import { User as UserModel, Post as PostModel } from '@prisma/client';

@Controller()
export class AppController {
  constructor(
    private readonly userService: UserService,
    private readonly postService: PostService,
  ) {}

  @Get('post/:id')
  async getPostById(@Param('id') id: string): Promise<PostModel> {
    return this.postService.post({ id: Number(id) });
  }

  @Get('feed')
  async getPublishedPosts(): Promise<PostModel[]> {
    return this.postService.posts({
      where: { published: true },
    });
  }
}

```

```
@Get('filtered-posts/:searchString')
async getFilteredPosts(
  @Param('searchString') searchString: string,
): Promise<PostModel[]> {
  return this.postService.posts({
    where: {
      OR: [
        {
          title: { contains: searchString },
        },
        {
          content: { contains: searchString },
        },
      ],
    },
  });
}

@Post('post')
async createDraft(
  @Body() postData: { title: string; content?: string; authorEmail:
string },
): Promise<PostModel> {
  const { title, content, authorEmail } = postData;
  return this.postService.createPost({
    title,
    content,
    author: {
      connect: { email: authorEmail },
    },
  });
}

@Post('user')
async signupUser(
  @Body() userData: { name?: string; email: string },
): Promise<UserModel> {
  return this.userService.createUser(userData);
}

@Put('publish/:id')
async publishPost(@Param('id') id: string): Promise<PostModel> {
  return this.postService.updatePost({
    where: { id: Number(id) },
    data: { published: true },
  });
}

@Delete('post/:id')
async deletePost(@Param('id') id: string): Promise<PostModel> {
  return this.postService.deletePost({ id: Number(id) });
}
}
```

This controller implements the following routes:

GET

- `/post/:id`: Fetch a single post by its `id`
- `/feed`: Fetch all *published* posts
- `/filter-posts/:searchString`: Filter posts by `title` or `content`

POST

- `/post`: Create a new post
 - Body:
 - `title: String` (required): The title of the post
 - `content: String` (optional): The content of the post
 - `authorEmail: String` (required): The email of the user that creates the post
- `/user`: Create a new user
 - Body:
 - `email: String` (required): The email address of the user
 - `name: String` (optional): The name of the user

PUT

- `/publish/:id`: Publish a post by its `id`

DELETE

- `/post/:id`: Delete a post by its `id`

Summary

In this recipe, you learned how to use Prisma along with NestJS to implement a REST API. The controller that implements the routes of the API is calling a `PrismaService` which in turn uses Prisma Client to send queries to a database to fulfill the data needs of incoming requests.

If you want to learn more about using NestJS with Prisma, be sure to check out the following resources:

- [NestJS & Prisma](#)
- [Ready-to-run example projects for REST & GraphQL](#)
- [Production-ready starter kit](#)
- [Video: Accessing Databases using NestJS with Prisma \(5min\)](#) by [Marc Stammerjohann](#)

Serve Static

In order to serve static content like a Single Page Application (SPA) we can use the `ServeStaticModule` from the `@nestjs/serve-static` package.

Installation

First we need to install the required package:

```
$ npm install --save @nestjs/serve-static
```

Bootstrap

Once the installation process is done, we can import the `ServeStaticModule` into the root `AppModule` and configure it by passing in a configuration object to the `forRoot()` method.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ServeStaticModule } from '@nestjs/serve-static';
import { join } from 'path';

@Module({
  imports: [
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'client'),
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

With this in place, build the static website and place its content in the location specified by the `rootPath` property.

Configuration

`ServeStaticModule` can be configured with a variety of options to customize its behavior. You can set the path to render your static app, specify excluded paths, enable or disable setting Cache-Control response header, etc. See the full list of options [here](#).

warning Notice The default `renderPath` of the Static App is `*` (all paths), and the module will send "index.html" files in response. It lets you create Client-Side routing for your SPA. Paths, specified in your controllers will fallback to the server. You can change this behavior setting `serveRoot`, `renderPath` combining them with other options.

Example

A working example is available [here](#).

Nest Commander

Expanding on the [standalone application](#) docs there's also the [nest-commander](#) package for writing command line applications in a structure similar to your typical Nest application.

info [nest-commander](#) is a third party package and is not managed by the entirety of the NestJS core team. Please, report any issues found with the library in the [appropriate repository](#)

Installation

Just like any other package, you've got to install it before you can use it.

```
$ npm i nest-commander
```

A Command file

[nest-commander](#) makes it easy to write new command-line applications with [decorators](#) via the [@Command\(\)](#) decorator for classes and the [@Option\(\)](#) decorator for methods of that class. Every command file should implement the [CommandRunner](#) abstract class and should be decorated with a [@Command\(\)](#) decorator.

Every command is seen as an [@Injectable\(\)](#) by Nest, so your normal Dependency Injection still works as you would expect it to. The only thing to take note of is the abstract class [CommandRunner](#), which should be implemented by each command. The [CommandRunner](#) abstract class ensures that all commands have a [run](#) method that returns a [Promise<void>](#) and takes in the parameters [string\[\], Record<string, any>](#). The [run](#) command is where you can kick all of your logic off from, it will take in whatever parameters did not match option flags and pass them in as an array, just in case you are really meaning to work with multiple parameters. As for the options, the [Record<string, any>](#), the names of these properties match the [name](#) property given to the [@Option\(\)](#) decorators, while their value matches the return of the option handler. If you'd like better type safety, you are welcome to create an interface for your options as well.

Running the Command

Similar to how in a NestJS application we can use the [NestFactory](#) to create a server for us, and run it using [listen](#), the [nest-commander](#) package exposes a simple to use API to run your server. Import the [CommandFactory](#) and use the [static](#) method [run](#) and pass in the root module of your application. This would probably look like below

```
import { CommandFactory } from 'nest-commander';
import { AppModule } from './app.module';

async function bootstrap() {
  await CommandFactory.run(AppModule);
}

bootstrap();
```

By default, Nest's logger is disabled when using the `CommandFactory`. It's possible to provide it though, as the second argument to the `run` function. You can either provide a custom NestJS logger, or an array of log levels you want to keep - it might be useful to at least provide `['error']` here, if you only want to print out Nest's error logs.

```
import { CommandFactory } from 'nest-commander';
import { AppModule } from './app.module';
import { LogService } from './log.service';

async function bootstrap() {
  await CommandFactory.run(AppModule, new LogService());

  // or, if you only want to print Nest's warnings and errors
  await CommandFactory.run(AppModule, ['warn', 'error']);
}

bootstrap();
```

And that's it. Under the hood, `CommandFactory` will worry about calling `NestFactory` for you and calling `app.close()` when necessary, so you shouldn't need to worry about memory leaks there. If you need to add in some error handling, there's always `try/catch` wrapping the `run` command, or you can chain on some `.catch()` method to the `bootstrap()` call.

Testing

So what's the use of writing a super awesome command line script if you can't test it super easily, right? Fortunately, `nest-commander` has some utilities you can make use of that fits in perfectly with the NestJS ecosystem, it'll feel right at home to any Nestlings out there. Instead of using the `CommandFactory` for building the command in test mode, you can use `CommandTestFactory` and pass in your metadata, very similarly to how `Test.createTestingModule` from `@nestjs/testing` works. In fact, it uses this package under the hood. You're also still able to chain on the `overrideProvider` methods before calling `compile()` so you can swap out DI pieces right in the test.

Putting it all together

The following class would equate to having a CLI command that can take in the subcommand `basic` or be called directly, with `-n`, `-s`, and `-b` (along with their long flags) all being supported and with custom parsers for each option. The `--help` flag is also supported, as is customary with commander.

```
import { Command, CommandRunner, Option } from 'nest-commander';
import { LogService } from './log.service';

interface BasicCommandOptions {
  string?: string;
  boolean?: boolean;
  number?: number;
}
```

```
}

@Command({ name: 'basic', description: 'A parameter parse' })
export class BasicCommand extends CommandRunner {
  constructor(private readonly logService: LogService) {
    super()
  }

  async run(
    passedParam: string[],
    options?: BasicCommandOptions,
  ): Promise<void> {
    if (options?.boolean !== undefined && options?.boolean !== null) {
      this.runWithBoolean(passedParam, options.boolean);
    } else if (options?.number) {
      this.runWithNumber(passedParam, options.number);
    } else if (options?.string) {
      this.runWithString(passedParam, options.string);
    } else {
      this.runWithNone(passedParam);
    }
  }

  @Option({
    flags: '-n, --number [number]',
    description: 'A basic number parser',
  })
  parseNumber(val: string): number {
    return Number(val);
  }

  @Option({
    flags: '-s, --string [string]',
    description: 'A string return',
  })
  parseString(val: string): string {
    return val;
  }

  @Option({
    flags: '-b, --boolean [boolean]',
    description: 'A boolean parser',
  })
  parseBoolean(val: string): boolean {
    return JSON.parse(val);
  }

  runWithString(param: string[], option: string): void {
    this.logService.log({ param, string: option });
  }

  runWithNumber(param: string[], option: number): void {
    this.logService.log({ param, number: option });
  }
}
```

```
runWithBoolean(param: string[], option: boolean): void {  
  this.logService.log({ param, boolean: option });  
}  
  
runWithNone(param: string[]): void {  
  this.logService.log({ param });  
}  
}
```

Make sure the command class is added to a module

```
@Module({  
  providers: [LogService, BasicCommand],  
})  
export class AppModule {}
```

And now to be able to run the CLI in your main.ts you can do the following

```
async function bootstrap() {  
  await CommandFactory.run(AppModule);  
}  
  
bootstrap();
```

And just like that, you've got a command line application.

More Information

Visit the [nest-commander docs site](#) for more information, examples, and API documentation.

Async Local Storage

`AsyncLocalStorage` is a [Node.js API](#) (based on the `async_hooks` API) that provides an alternative way of propagating local state through the application without the need to explicitly pass it as a function parameter. It is similar to a thread-local storage in other languages.

The main idea of Async Local Storage is that we can *wrap* some function call with the `AsyncLocalStorage#run` call. All code that is invoked within the wrapped call gets access to the same `store`, which will be unique to each call chain.

In the context of NestJS, that means if we can find a place within the request's lifecycle where we can wrap the rest of the request's code, we will be able to access and modify state visible only to that request, which may serve as an alternative to REQUEST-scoped providers and some of their limitations.

Alternatively, we can use ALS to propagate context for only a part of the system (for example the *transaction* object) without passing it around explicitly across services, which can increase isolation and encapsulation.

Custom implementation

NestJS itself does not provide any built-in abstraction for `AsyncLocalStorage`, so let's walk through how we could implement it ourselves for the simplest HTTP case to get a better understanding of the whole concept:

info For a ready-made [dedicated package](#), continue reading below.

1. First, create a new instance of the `AsyncLocalStorage` in some shared source file. Since we're using NestJS, let's also turn it into a module with a custom provider.

```
@@filename(als.module)
@Module({
  providers: [
    {
      provide: AsyncLocalStorage,
      useValue: new AsyncLocalStorage(),
    },
  ],
  exports: [AsyncLocalStorage],
})
export class AlsModule {}
```

info Hint `AsyncLocalStorage` is imported from `async_hooks`.

2. We're only concerned with HTTP, so let's use a middleware to wrap the `next` function with `AsyncLocalStorage#run`. Since a middleware is the first thing that the request hits, this will make the `store` available in all enhancers and the rest of the system.

```
@@filename(app.module)
@Module({
  imports: [AlsModule]
  providers: [CatService],
  controllers: [CatController],
})
export class AppModule implements NestModule {
  constructor(
    // inject the AsyncLocalStorage in the module constructor,
    private readonly als: AsyncLocalStorage
  ) {}

  configure(consumer: MiddlewareConsumer) {
    // bind the middleware,
    consumer
      .apply((req, res, next) => {
        // populate the store with some default values
        // based on the request,
        const store = {
          userId: req.headers['x-user-id'],
        };
        // and pass the "next" function as callback
        // to the "als.run" method together with the store.
        this.als.run(store, () => next());
      })
      // and register it for all routes (in case of Fastify use '(.*)')
      .forRoutes('*');
  }
}

@@switch
@Module({
  imports: [AlsModule]
  providers: [CatService],
  controllers: [CatController],
})
@Dependencies(AsyncLocalStorage)
export class AppModule {
  constructor(als) {
    // inject the AsyncLocalStorage in the module constructor,
    this.als = als
  }

  configure(consumer) {
    // bind the middleware,
    consumer
      .apply((req, res, next) => {
        // populate the store with some default values
        // based on the request,
        const store = {
          userId: req.headers['x-user-id'],
        };
        // and pass the "next" function as callback
        // to the "als.run" method together with the store.
      })
  }
}
```

```

        this.als.run(store, () => next());
    })
    // and register it for all routes (in case of Fastify use '(.*)')
    .forRoutes('*');
}
}

```

3. Now, anywhere within the lifecycle of a request, we can access the local store instance.

```

@@filename(cat.service)
@Injectable()
export class CatService {
  constructor(
    // We can inject the provided ALS instance.
    private readonly als: AsyncLocalStorage,
    private readonly catRepository: CatRepository,
  ) {}

  getCatForUser() {
    // The "getStore" method will always return the
    // store instance associated with the given request.
    const userId = this.als.getStore()["userId"] as number;
    return this.catRepository.getForUser(userId);
  }
}

@@switch
@Injectable()
@Dependencies(AsyncLocalStorage, CatRepository)
export class CatService {
  constructor(als, catRepository) {
    // We can inject the provided ALS instance.
    this.als = als
    this.catRepository = catRepository
  }

  getCatForUser() {
    // The "getStore" method will always return the
    // store instance associated with the given request.
    const userId = this.als.getStore()["userId"] as number;
    return this.catRepository.getForUser(userId);
  }
}

```

4. That's it. Now we have a way to share request related state without needing to inject the whole **REQUEST** object.

warning Please be aware that while the technique is useful for many use-cases, it inherently obfuscates the code flow (creating implicit context), so use it responsibly and especially avoid creating contextual "God objects".

NestJS CLS

The `nestjs-cls` package provides several DX improvements over using plain `AsyncLocalStorage` (CLS is an abbreviation of the term *continuation-local storage*). It abstracts the implementation into a `ClsModule` that offers various ways of initializing the `store` for different transports (not only HTTP), as well as a strong-typing support.

The store can then be accessed with an injectable `ClsService`, or entirely abstracted away from the business logic by using [Proxy Providers](#).

info `nestjs-cls` is a third party package and is not managed by the NestJS core team. Please, report any issues found with the library in the [appropriate repository](#).

Installation

Apart from a peer dependency on the `@nestjs` libs, it only uses the built-in Node.js API. Install it as any other package.

```
npm i nestjs-cls
```

Usage

A similar functionality as described [above](#) can be implemented using `nestjs-cls` as follows:

1. Import the `ClsModule` in the root module.

```
@@filename(app.module)
@Module({
  imports: [
    // Register the ClsModule,
    ClsModule.forRoot({
      middleware: {
        // automatically mount the
        // ClsMiddleware for all routes
        mount: true,
        // and use the setup method to
        // provide default store values.
        setup: (cls, req) => {
          cls.set('userId', req.headers['x-user-id']);
        },
      },
    }),
  ],
  providers: [CatService],
  controllers: [CatController],
})
export class AppModule {}
```

2. And then can use the `ClsService` to access the store values.

```

@@filename(cat.service)
@Injectable()
export class CatService {
  constructor(
    // We can inject the provided ClsService instance,
    private readonly cls: ClsService,
    private readonly catRepository: CatRepository,
  ) {}

  getCatForUser() {
    // and use the "get" method to retrieve any stored value.
    const userId = this.cls.get('userId');
    return this.catRepository.getForUser(userId);
  }
}

@@switch
@Injectable()
@Dependencies(AsyncLocalStorage, CatRepository)
export class CatService {
  constructor(cls, catRepository) {
    // We can inject the provided ClsService instance,
    this.cls = cls
    this.catRepository = catRepository
  }

  getCatForUser() {
    // and use the "get" method to retrieve any stored value.
    const userId = this.cls.get('userId');
    return this.catRepository.getForUser(userId);
  }
}

```

3. To get strong typing of the store values managed by the `ClsService` (and also get auto-suggestions of the string keys), we can use an optional type parameter `ClsService<MyClsStore>` when injecting it.

```

export interface MyClsStore extends ClsStore {
  userId: number;
}

```

info hint It is also possible to let the package automatically generate a Request ID and access it later with `cls.getId()`, or to get the whole Request object using `cls.get(CLS_REQ)`.

Testing

Since the `ClsService` is just another injectable provider, it can be entirely mocked out in unit tests.

However, in certain integration tests, we might still want to use the real `ClsService` implementation. In that case, we will need to wrap the context-aware piece of code with a call to `ClsService#run` or `ClsService#runWith`.

```
describe('CatService', () => {
  let service: CatService
  let cls: ClsService
  const mockCatRepository = createMock<CatRepository>()

  beforeEach(async () => {
    const module = await Test.createTestingModule({
      // Set up most of the testing module as we normally would.
      providers: [
        CatService,
        {
          provide: CatRepository
          useValue: mockCatRepository
        }
      ],
      imports: [
        // Import the static version of ClsModule which only provides
        // the ClsService, but does not set up the store in any way.
        ClsModule
      ],
    }).compile()

    service = module.get(CatService)

    // Also retrieve the ClsService for later use.
    cls = module.get(ClsService)
  })

  describe('getCatForUser', () => {
    it('retrieves cat based on user id', async () => {
      const expectedUserId = 42
      mockCatRepository.getForUser.mockImplementationOnce(
        (id) => ({ userId: id })
      )

      // Wrap the test call in the `runWith` method
      // in which we can pass hand-crafted store values.
      const cat = await cls.runWith(
        { userId: expectedUserId },
        () => service.getCatForUser()
      )

      expect(cat.userId).toEqual(expectedUserId)
    })
  })
})
```

More information

Visit the [NestJS CLS GitHub Page](#) for the full API documentation and more code examples.

Automock

Automock is a standalone library for unit testing. Using TypeScript Reflection API ([reflect-metadata](#)) internally to produce mock objects, Automock streamlines test development by automatically mocking class external dependencies.

info **info** [Automock](#) is a third party package and is not managed by the NestJS core team. Please, report any issues found with the library in the [appropriate repository](#)

Introduction

The dependency injection (DI) container is an essential component of the Nest module system. This container is utilized both during testing, and the application execution. Unit tests vary from other types of tests, such as integration tests, in that they must fully override providers/services within the DI container. External class dependencies (providers) of the so-called "unit", have to be totally isolated. That is, all dependencies within the DI container should be replaced by mock objects. As a result, loading the target module and replacing the providers inside it is a process that loops back on itself. Automock tackles this issue by automatically mocking all the class external providers, resulting in total isolation of the unit under test.

Installation

```
$ npm i -D @automock/jest
```

Automock does not require any additional setup.

info **info** Jest is the only test framework currently supported by Automock. Sinon will shortly be released.

Example

Consider the following cats service, which takes three constructor parameters:

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

@Injectable()
export class CatsService {
  constructor(
    private logger: Logger,
    private httpService: HttpService,
    private catsDal: CatsDal,
  ) {}

  async getAllCats() {
    const cats = await
    this.httpService.get('http://localhost:3000/api/cats');
```

```

        this.logger.log('Successfully fetched all cats');

        this.catsDal.saveCats(cats);
    }
}

```

The service contains one public method, `getAllCats`, which is the method we use an example for the following unit test:

```

@@filename(cats.service.spec)
import { TestBed } from '@automock/jest';
import { CatsService } from './cats.service';

describe('CatsService unit spec', () => {
    let underTest: CatsService;
    let logger: jest.Mocked<Logger>;
    let httpService: jest.Mocked<HttpService>;
    let catsDal: jest.Mocked<CatsDal>;

    beforeAll(() => {
        const { unit, unitRef } = TestBed.create(CatsService)
            .mock(HttpService)
            .using({ get: jest.fn() })
            .mock(Logger)
            .using({ log: jest.fn() })
            .mock(CatsDal)
            .using({ saveCats: jest.fn() })
            .compile();

        underTest = unit;

        logger = unitRef.get(Logger);
        httpService = unitRef.get(HttpService);
        catsDal = unitRef.get(CatsDal);
    });

    describe('when getting all the cats', () => {
        test('then meet some expectations', async () => {
            httpService.get.mockResolvedValueOnce([
                { id: 1, name: 'Catty' }
            ]);
            await catsService.getAllCats();

            expect(logger.log).toBeCalled();
            expect(catsDal).toBeCalledWith([
                { id: 1, name: 'Catty' }
            ]);
        });
    });
});

```

info The `jest.Mocked` utility type returns the Source type wrapped with type definitions of Jest mock function. ([reference](#))

About `unit` and `unitRef`

Let's examine the following code:

```
const { unit, unitRef } = TestBed.create(CatsService).compile();
```

Calling `.compile()` returns an object with two properties, `unit`, and `unitRef`.

`unit` is the unit under test, it is an actual instance of class being tested.

`unitRef` is the "unit reference", where the mocked dependencies of the tested class are stored, in a small container. The container's `.get()` method returns the mocked dependency with all of its methods automatically stubbed (using `jest.fn()`):

```
const { unit, unitRef } = TestBed.create(CatsService).compile();  
  
let httpServiceMock: jest.Mocked<HttpService> = unitRef.get(HttpService);
```

info The `.get()` method can accept either a `string` or an actual class (`Type`) as its argument. This essentially depends on how the provider is being injected to the class under test.

Working with different providers

Providers are one of the most important elements in Nest. You can think of many of the default Nest classes as providers, including services, repositories, factories, helpers, and so on. A provider's primary function is to take the form of an `Injectable` dependency.

Consider the following `CatsService`, it takes one parameter, which is an instance of the following `Logger` interface:

```
export interface Logger {  
  log(message: string): void;  
}  
  
export class CatsService {  
  constructor(private logger: Logger) {}  
}
```

TypeScript's Reflection API does not support interface reflection yet. Nest solves this issue with string-based injection tokens (see [Custom Providers](#)):

```
export const MyLoggerProvider = {  
  provide: 'MY_LOGGER_TOKEN',  
  useValue: { ... },  
}
```

```
export class CatsService {  
  constructor(@Inject('MY_LOGGER_TOKEN') private readonly logger: Logger)  
  {}  
}
```

Automock follows this practice and lets you provide a string-based token instead of providing the actual class in the `unitRef.get()` method:

```
const { unit, unitRef } = TestBed.create(CatsService).compile();  
  
let loggerMock: jest.Mocked<Logger> = unitRef.get('MY_LOGGER_TOKEN');
```

More Information

Visit [Automock GitHub repository](#) for more information.