

## Introduction

Nest (NestJS) is a framework for building efficient, scalable [Node.js](#) server-side applications. It uses progressive JavaScript, is built with and fully supports [TypeScript](#) (yet still enables developers to code in pure JavaScript) and combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming).

Under the hood, Nest makes use of robust HTTP Server frameworks like [Express](#) (the default) and optionally can be configured to use [Fastify](#) as well!

Nest provides a level of abstraction above these common Node.js frameworks (Express/Fastify), but also exposes their APIs directly to the developer. This gives developers the freedom to use the myriad of third-party modules which are available for the underlying platform.

## Philosophy

In recent years, thanks to Node.js, JavaScript has become the “lingua franca” of the web for both front and backend applications. This has given rise to awesome projects like [Angular](#), [React](#) and [Vue](#), which improve developer productivity and enable the creation of fast, testable, and extensible frontend applications. However, while plenty of superb libraries, helpers, and tools exist for Node (and server-side JavaScript), none of them effectively solve the main problem of - **Architecture**.

Nest provides an out-of-the-box application architecture which allows developers and teams to create highly testable, scalable, loosely coupled, and easily maintainable applications. The architecture is heavily inspired by Angular.

## Installation

To get started, you can either scaffold the project with the [Nest CLI](#), or clone a starter project (both will produce the same outcome).

To scaffold the project with the Nest CLI, run the following commands. This will create a new project directory, and populate the directory with the initial core Nest files and supporting modules, creating a conventional base structure for your project. Creating a new project with the **Nest CLI** is recommended for first-time users. We'll continue with this approach in [First Steps](#).

```
$ npm i -g @nestjs/cli  
$ nest new project-name
```

**info Hint** To create a new TypeScript project with stricter feature set, pass the `--strict` flag to the `nest new` command.

## Alternatives

Alternatively, to install the TypeScript starter project with **Git**:

```
$ git clone https://github.com/nestjs/typescript-starter.git project
$ cd project
$ npm install
$ npm run start
```

**info Hint** If you'd like to clone the repository without the git history, you can use [degit](#).

Open your browser and navigate to <http://localhost:3000/>.

To install the JavaScript flavor of the starter project, use [javascript-starter.git](#) in the command sequence above.

You can also manually create a new project from scratch by installing the core and supporting files with **npm** (or **yarn**). In this case, of course, you'll be responsible for creating the project boilerplate files yourself.

```
$ npm i --save @nestjs/core @nestjs/common rxjs reflect-metadata
```

## First steps

In this set of articles, you'll learn the **core fundamentals** of Nest. To get familiar with the essential building blocks of Nest applications, we'll build a basic CRUD application with features that cover a lot of ground at an introductory level.

### Language

We're in love with [TypeScript](#), but above all - we love [Node.js](#). That's why Nest is compatible with both TypeScript and **pure JavaScript**. Nest takes advantage of the latest language features, so to use it with vanilla JavaScript we need a [Babel](#) compiler.

We'll mostly use TypeScript in the examples we provide, but you can always **switch the code snippets** to vanilla JavaScript syntax (simply click to toggle the language button in the upper right hand corner of each snippet).

### Prerequisites

Please make sure that [Node.js](#) (version  $\geq 16$ ) is installed on your operating system.

### Setup

Setting up a new project is quite simple with the [Nest CLI](#). With [npm](#) installed, you can create a new Nest project with the following commands in your OS terminal:

```
$ npm i -g @nestjs/cli  
$ nest new project-name
```

**info Hint** To create a new project with TypeScript's [stricter](#) feature set, pass the [--strict](#) flag to the [nest new](#) command.

The [project-name](#) directory will be created, node modules and a few other boilerplate files will be installed, and a [src/](#) directory will be created and populated with several core files.

```
src  
app.controller.spec.ts  
app.controller.ts  
app.module.ts  
app.service.ts  
main.ts
```

Here's a brief overview of those core files:

---

[app.controller.ts](#) A basic controller with a single route.

---

[app.controller.spec.ts](#) The unit tests for the controller.

---

---

<code>app.module.ts</code>	The root module of the application.
<code>app.service.ts</code>	A basic service with a single method.
<code>main.ts</code>	The entry file of the application which uses the core function <code>NestFactory</code> to create a Nest application instance.

---

The `main.ts` includes an async function, which will **bootstrap** our application:

```
@@filename(main)

import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

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

bootstrap();
@@switch
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

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

bootstrap();
```

To create a Nest application instance, we use the core `NestFactory` class. `NestFactory` exposes a few static methods that allow creating an application instance. The `create()` method returns an application object, which fulfills the `INestApplication` interface. This object provides a set of methods which are described in the coming chapters. In the `main.ts` example above, we simply start up our HTTP listener, which lets the application await inbound HTTP requests.

Note that a project scaffolded with the Nest CLI creates an initial project structure that encourages developers to follow the convention of keeping each module in its own dedicated directory.

**info Hint** By default, if any error happens while creating the application your app will exit with the code 1. If you want to make it throw an error instead disable the option `abortOnError` (e.g., `NestFactory.create(AppModule, {{ '{' }} abortOnError: false {{ '}' }}))`.

## Platform

Nest aims to be a platform-agnostic framework. Platform independence makes it possible to create reusable logical parts that developers can take advantage of across several different types of applications. Technically, Nest is able to work with any Node HTTP framework once an adapter is created. There are two HTTP platforms supported out-of-the-box: `express` and `fastify`. You can choose the one that best suits your needs.

---

**Express** is a well-known minimalist web framework for node. It's a battle tested, production-ready library with lots of resources implemented by the community. The **@nestjs/platform-express** package is used by default. Many users are well served with Express, and need take no action to enable it.

---

**platform-fastify** **Fastify** is a high performance and low overhead framework highly focused on providing maximum efficiency and speed. Read how to use it [here](#).

Whichever platform is used, it exposes its own application interface. These are seen respectively as **NestExpressApplication** and **NestFastifyApplication**.

When you pass a type to the **NestFactory.create()** method, as in the example below, the **app** object will have methods available exclusively for that specific platform. Note, however, you don't **need** to specify a type **unless** you actually want to access the underlying platform API.

```
const app = await NestFactory.create<NestExpressApplication>(AppModule);
```

## Running the application

Once the installation process is complete, you can run the following command at your OS command prompt to start the application listening for inbound HTTP requests:

```
$ npm run start
```

**info Hint** To speed up the development process (x20 times faster builds), you can use the **SWC builder** by passing the **-b swc** flag to the **start** script, as follows **npm run start -- -b swc**.

This command starts the app with the HTTP server listening on the port defined in the **src/main.ts** file. Once the application is running, open your browser and navigate to **http://localhost:3000/**. You should see the **Hello World!** message.

To watch for changes in your files, you can run the following command to start the application:

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

This command will watch your files, automatically recompiling and reloading the server.

## Linting and formatting

**CLI** provides best effort to scaffold a reliable development workflow at scale. Thus, a generated Nest project comes with both a code **linter** and **formatter** preinstalled (respectively **eslint** and **prettier**).

**info Hint** Not sure about the role of formatters vs linters? Learn the difference [here](#).

To ensure maximum stability and extensibility, we use the base `eslint` and `prettier` cli packages. This setup allows neat IDE integration with official extensions by design.

For headless environments where an IDE is not relevant (Continuous Integration, Git hooks, etc.) a Nest project comes with ready-to-use `npm` scripts.

```
# Lint and autofix with eslint
$ npm run lint

# Format with prettier
$ npm run format
```

## Controllers

Controllers are responsible for handling incoming **requests** and returning **responses** to the client.



A controller's purpose is to receive specific requests for the application. The **routing** mechanism controls which controller receives which requests. Frequently, each controller has more than one route, and different routes can perform different actions.

In order to create a basic controller, we use classes and **decorators**. Decorators associate classes with required metadata and enable Nest to create a routing map (tie requests to the corresponding controllers).

**info Hint** For quickly creating a CRUD controller with the [validation](#) built-in, you may use the CLI's [CRUD generator](#): `nest g resource [name]`.

## Routing

In the following example we'll use the `@Controller()` decorator, which is **required** to define a basic controller. We'll specify an optional route path prefix of `cats`. Using a path prefix in a `@Controller()` decorator allows us to easily group a set of related routes, and minimize repetitive code. For example, we may choose to group a set of routes that manage interactions with a cat entity under the route `/cats`. In that case, we could specify the path prefix `cats` in the `@Controller()` decorator so that we don't have to repeat that portion of the path for each route in the file.

```
@@filename(cats.controller)
import { Controller, Get } from '@nestjs/common';

@Controller('cats')
export class CatsController {
    @Get()
    findAll(): string {
        return 'This action returns all cats';
    }
}
@switch
import { Controller, Get } from '@nestjs/common';

@Controller('cats')
export class CatsController {
    @Get()
    findAll() {
        return 'This action returns all cats';
    }
}
```

**info Hint** To create a controller using the CLI, simply execute the `$ nest g controller [name]` command.

The `@Get()` HTTP request method decorator before the `findAll()` method tells Nest to create a handler for a specific endpoint for HTTP requests. The endpoint corresponds to the HTTP request method (GET in this case) and the route path. What is the route path? The route path for a handler is determined by concatenating the (optional) prefix declared for the controller, and any path specified in the method's decorator. Since we've declared a prefix for every route (`cats`), and haven't added any path information in the decorator, Nest will map `GET /cats` requests to this handler. As mentioned, the path includes both the optional controller path prefix **and** any path string declared in the request method decorator. For example, a path prefix of `cats` combined with the decorator `@Get('breed')` would produce a route mapping for requests like `GET /cats/breed`.

In our example above, when a GET request is made to this endpoint, Nest routes the request to our user-defined `findAll()` method. Note that the method name we choose here is completely arbitrary. We obviously must declare a method to bind the route to, but Nest doesn't attach any significance to the method name chosen.

This method will return a 200 status code and the associated response, which in this case is just a string. Why does that happen? To explain, we'll first introduce the concept that Nest employs two **different** options for manipulating responses:

Using this built-in method, when a request handler returns a JavaScript object or array, it will **automatically** be serialized to JSON. When it returns a JavaScript primitive type (e.g., `string`, `number`, `boolean`), however, Nest will send just the value without attempting to serialize it. This makes response handling simple: just return the value, and Nest takes care of the rest.

Standard  
(recommended)

Furthermore, the response's **status code** is always 200 by default, except for POST requests which use 201. We can easily change this behavior by adding the `@HttpCode(...)` decorator at a handler-level (see [Status codes](#)).

We can use the library-specific (e.g., Express) [response object](#), which can be injected using the `@Res()` decorator in the method handler signature (e.g., `findAll(@Res() response)`). With this approach, you have the ability to use the native response handling methods exposed by that object. For example, with Express, you can construct responses using code like `response.status(200).send()`.

Library-specific

**warning** **Warning** Nest detects when the handler is using either `@Res()` or `@Next()`, indicating you have chosen the library-specific option. If both approaches are used at the same time, the Standard approach is **automatically disabled** for this single route and will no longer work as expected. To use both approaches at the same time (for example, by injecting the response object to only set cookies/headers but still leave the rest to the framework), you must set the `passthrough` option to `true` in the `@Res({{ '{' }} passthrough: true {{ '}' }})` decorator.

## Request object

Handlers often need access to the client **request** details. Nest provides access to the [request object](#) of the underlying platform (Express by default). We can access the request object by instructing Nest to inject it by adding the `@Req()` decorator to the handler's signature.

```
@@filename(cats.controller)
import { Controller, Get, Req } from '@nestjs/common';
import { Request } from 'express';

@Controller('cats')
export class CatsController {
  @Get()
  findAll(@Req() request: Request): string {
    return 'This action returns all cats';
  }
}

@Controller('cats')
export class CatsController {
  @Get()
  @Bind(Req())
  findAll(request) {
    return 'This action returns all cats';
  }
}
```

**info Hint** In order to take advantage of `express` typings (as in the `request: Request` parameter example above), install `@types/express` package.

The request object represents the HTTP request and has properties for the request query string, parameters, HTTP headers, and body (read more [here](#)). In most cases, it's not necessary to grab these properties manually. We can use dedicated decorators instead, such as `@Body()` or `@Query()`, which are available out of the box. Below is a list of the provided decorators and the plain platform-specific objects they represent.

<code>@Request(), @Req()</code>	<code>req</code>
<code>@Response(), @Res()*</code>	<code>res</code>
<code>@Next()</code>	<code>next</code>
<code>@Session()</code>	<code>req.session</code>
<code>@Param(key?: string)</code>	<code>req.params / req.params[key]</code>
<code>@Body(key?: string)</code>	<code>req.body / req.body[key]</code>
<code>@Query(key?: string)</code>	<code>req.query / req.query[key]</code>
<code>@Headers(name?: string)</code>	<code>req.headers / req.headers[name]</code>
<code>@Ip()</code>	<code>req.ip</code>
<code>@HostParam()</code>	<code>req.hosts</code>

\* For compatibility with typings across underlying HTTP platforms (e.g., Express and Fastify), Nest provides `@Res()` and `@Response()` decorators. `@Res()` is simply an alias for `@Response()`. Both directly expose the underlying native platform `response` object interface. When using them, you should also import the typings for the underlying library (e.g., `@types/express`) to take full advantage. Note that when you inject either `@Res()` or `@Response()` in a method handler, you put Nest into **Library-specific mode** for that handler, and you become responsible for managing the response. When doing so, you must issue some kind of response by making a call on the `response` object (e.g., `res.json(...)` or `res.send(...)`), or the HTTP server will hang.

**info Hint** To learn how to create your own custom decorators, visit [this chapter](#).

## Resources

Earlier, we defined an endpoint to fetch the cats resource (**GET** route). We'll typically also want to provide an endpoint that creates new records. For this, let's create the **POST** handler:

```
@@filename(cats.controller)
import { Controller, Get, Post } from '@nestjs/common';

@Controller('cats')
export class CatsController {
    @Post()
    create(): string {
        return 'This action adds a new cat';
    }

    @Get()
    findAll(): string {
        return 'This action returns all cats';
    }
}

@@switch
import { Controller, Get, Post } from '@nestjs/common';

@Controller('cats')
export class CatsController {
    @Post()
    create() {
        return 'This action adds a new cat';
    }

    @Get()
    findAll() {
        return 'This action returns all cats';
    }
}
```

It's that simple. Nest provides decorators for all of the standard HTTP methods: `@Get()`, `@Post()`, `@Put()`, `@Delete()`, `@Patch()`, `@Options()`, and `@Head()`. In addition, `@All()` defines an endpoint

that handles all of them.

## Route wildcards

Pattern based routes are supported as well. For instance, the asterisk is used as a wildcard, and will match any combination of characters.

```
@Get('ab*cd')
findAll() {
  return 'This route uses a wildcard';
}
```

The '`ab*cd`' route path will match `abcd`, `ab_cd`, `abecd`, and so on. The characters `?`, `+`, `*`, and `( )` may be used in a route path, and are subsets of their regular expression counterparts. The hyphen (`-`) and the dot (`.`) are interpreted literally by string-based paths.

**warning Warning** A wildcard in the middle of the route is only supported by express.

## Status code

As mentioned, the response **status code** is always **200** by default, except for POST requests which are **201**. We can easily change this behavior by adding the `@HttpCode(...)` decorator at a handler level.

```
@Post()
@HttpCode(204)
create() {
  return 'This action adds a new cat';
}
```

**info Hint** Import `HttpCode` from the `@nestjs/common` package.

Often, your status code isn't static but depends on various factors. In that case, you can use a library-specific **response** (inject using `@Res()`) object (or, in case of an error, throw an exception).

## Headers

To specify a custom response header, you can either use a `@Header()` decorator or a library-specific response object (and call `res.header()` directly).

```
@Post()
@Header('Cache-Control', 'none')
create() {
  return 'This action adds a new cat';
}
```

info Hint Import `Header` from the `@nestjs/common` package.

## Redirection

To redirect a response to a specific URL, you can either use a `@Redirect()` decorator or a library-specific response object (and call `res.redirect()` directly).

`@Redirect()` takes two arguments, `url` and `statusCode`, both are optional. The default value of `statusCode` is `302 (Found)` if omitted.

```
@Get()  
@Redirect('https://nestjs.com', 301)
```

Sometimes you may want to determine the HTTP status code or the redirect URL dynamically. Do this by returning an object from the route handler method with the shape:

```
{  
  "url": string,  
  "statusCode": number  
}
```

Returned values will override any arguments passed to the `@Redirect()` decorator. For example:

```
@Get('docs')  
@Redirect('https://docs.nestjs.com', 302)  
getDocs(@Query('version') version) {  
  if (version && version === '5') {  
    return { url: 'https://docs.nestjs.com/v5/' };  
  }  
}
```

## Route parameters

Routes with static paths won't work when you need to accept **dynamic data** as part of the request (e.g., `GET /cats/1` to get cat with id `1`). In order to define routes with parameters, we can add route parameter **tokens** in the path of the route to capture the dynamic value at that position in the request URL. The route parameter token in the `@Get()` decorator example below demonstrates this usage. Route parameters declared in this way can be accessed using the `@Param()` decorator, which should be added to the method signature.

info Hint Routes with parameters should be declared after any static paths. This prevents the parameterized paths from intercepting traffic destined for the static paths.

```

@@filename()
@Get(':id')
findOne(@Param() params: any): string {
  console.log(params.id);
  return `This action returns a ${params.id} cat`;
}
@@switch
@Get(':id')
@Bind(Param())
findOne(params) {
  console.log(params.id);
  return `This action returns a ${params.id} cat`;
}

```

`@Param()` is used to decorate a method parameter (`params` in the example above), and makes the **route** parameters available as properties of that decorated method parameter inside the body of the method. As seen in the code above, we can access the `id` parameter by referencing `params.id`. You can also pass in a particular parameter token to the decorator, and then reference the route parameter directly by name in the method body.

**info Hint** Import `Param` from the `@nestjs/common` package.

```

@@filename()
@Get(':id')
findOne(@Param('id') id: string): string {
  return `This action returns a ${id} cat`;
}
@@switch
@Get(':id')
@Bind(Param('id'))
findOne(id) {
  return `This action returns a ${id} cat`;
}

```

## Sub-Domain Routing

The `@Controller` decorator can take a `host` option to require that the HTTP host of the incoming requests matches some specific value.

```

@Controller({ host: 'admin.example.com' })
export class AdminController {
  @Get()
  index(): string {
    return 'Admin page';
  }
}

```

**Warning** Since **Fastify** lacks support for nested routers, when using sub-domain routing, the (default) Express adapter should be used instead.

Similar to a route `path`, the `hosts` option can use tokens to capture the dynamic value at that position in the host name. The host parameter token in the `@Controller()` decorator example below demonstrates this usage. Host parameters declared in this way can be accessed using the `@HostParam()` decorator, which should be added to the method signature.

```
@Controller({ host: ':account.example.com' })
export class AccountController {
  @Get()
  getInfo(@HostParam('account') account: string) {
    return account;
}
}
```

## Scopes

For people coming from different programming language backgrounds, it might be unexpected to learn that in Nest, almost everything is shared across incoming requests. We have a connection pool to the database, singleton services with global state, etc. Remember that Node.js doesn't follow the request/response Multi-Threaded Stateless Model in which every request is processed by a separate thread. Hence, using singleton instances is fully **safe** for our applications.

However, there are edge-cases when request-based lifetime of the controller may be the desired behavior, for instance per-request caching in GraphQL applications, request tracking or multi-tenancy. Learn how to control scopes [here](#).

## Asynchronicity

We love modern JavaScript and we know that data extraction is mostly **asynchronous**. That's why Nest supports and works well with `async` functions.

**info Hint** Learn more about `async / await` feature [here](#)

Every `async` function has to return a `Promise`. This means that you can return a deferred value that Nest will be able to resolve by itself. Let's see an example of this:

```
@@filename(cats.controller)
@Get()
async findAll(): Promise<any[]> {
  return [];
}
@@switch
@Get()
async findAll() {
  return [];
}
```

The above code is fully valid. Furthermore, Nest route handlers are even more powerful by being able to return RxJS [observable streams](#). Nest will automatically subscribe to the source underneath and take the last emitted value (once the stream is completed).

```
@@filename(cats.controller)
@Get()
findAll(): Observable<any[]> {
  return of([]);
}

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

Both of the above approaches work and you can use whatever fits your requirements.

## Request payloads

Our previous example of the POST route handler didn't accept any client params. Let's fix this by adding the [@Body\(\)](#) decorator here.

But first (if you use TypeScript), we need to determine the **DTO** (Data Transfer Object) schema. A DTO is an object that defines how the data will be sent over the network. We could determine the DTO schema by using **TypeScript** interfaces, or by simple classes. Interestingly, we recommend using **classes** here. Why? Classes are part of the JavaScript ES6 standard, and therefore they are preserved as real entities in the compiled JavaScript. On the other hand, since TypeScript interfaces are removed during the transpilation, Nest can't refer to them at runtime. This is important because features such as **Pipes** enable additional possibilities when they have access to the metatype of the variable at runtime.

Let's create the [CreateCatDto](#) class:

```
@@filename(create-cat.dto)
export class CreateCatDto {
  name: string;
  age: number;
  breed: string;
}
```

It has only three basic properties. Thereafter we can use the newly created DTO inside the [CatsController](#):

```
@@filename(cats.controller)
@Post()
async create(@Body() createCatDto: CreateCatDto) {
```

```
    return 'This action adds a new cat';
}

@@switch
@Post()
@Bind(Body())
async create(createCatDto) {
  return 'This action adds a new cat';
}
```

info **Hint** Our `ValidationPipe` can filter out properties that should not be received by the method handler. In this case, we can whitelist the acceptable properties, and any property not included in the whitelist is automatically stripped from the resulting object. In the `CreateCatDto` example, our whitelist is the `name`, `age`, and `breed` properties. Learn more [here](#).

## Handling errors

There's a separate chapter about handling errors (i.e., working with exceptions) [here](#).

## Full resource sample

Below is an example that makes use of several of the available decorators to create a basic controller. This controller exposes a couple of methods to access and manipulate internal data.

```
@@filename(cats.controller)
import { Controller, Get, Query, Post, Body, Put, Param, Delete } from
'@nestjs/common';
import { CreateCatDto, UpdateCatDto, ListAllEntities } from './dto';

@Controller('cats')
export class CatsController {
  @Post()
  create(@Body() createCatDto: CreateCatDto) {
    return 'This action adds a new cat';
  }

  @Get()
  findAll(@Query() query: ListAllEntities) {
    return `This action returns all cats (limit: ${query.limit} items)`;
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return `This action returns a #${id} cat`;
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() updateCatDto: UpdateCatDto) {
    return `This action updates a #${id} cat`;
  }
}
```

```
@Delete(':id')
remove(@Param('id') id: string) {
  return `This action removes a ${id} cat`;
}

@@switch
import { Controller, Get, Query, Post, Body, Put, Param, Delete, Bind } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Post()
  @Bind(Body())
  create(createCatDto) {
    return 'This action adds a new cat';
  }

  @Get()
  @Bind(Query())
  findAll(query) {
    console.log(query);
    return `This action returns all cats (limit: ${query.limit} items)`;
  }

  @Get(':id')
  @Bind(Param('id'))
  findOne(id) {
    return `This action returns a ${id} cat`;
  }

  @Put(':id')
  @Bind(Param('id'), Body())
  update(id, updateCatDto) {
    return `This action updates a ${id} cat`;
  }

  @Delete(':id')
  @Bind(Param('id'))
  remove(id) {
    return `This action removes a ${id} cat`;
  }
}
```

info **Hint** 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. Read more about this feature [here](#).

## Getting up and running

With the above controller fully defined, Nest still doesn't know that **CatsController** exists and as a result won't create an instance of this class.

Controllers always belong to a module, which is why we include the `controllers` array within the `@Module()` decorator. Since we haven't yet defined any other modules except the root `AppModule`, we'll use that to introduce the `CatsController`:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { CatsController } from './cats/cats.controller';

@Module({
  controllers: [CatsController],
})
export class AppModule {}
```

We attached the metadata to the module class using the `@Module()` decorator, and Nest can now easily reflect which controllers have to be mounted.

## Library-specific approach

So far we've discussed the Nest standard way of manipulating responses. The second way of manipulating the response is to use a library-specific `response object`. In order to inject a particular response object, we need to use the `@Res()` decorator. To show the differences, let's rewrite the `CatsController` to the following:

```
@@filename()
import { Controller, Get, Post, Res, HttpStatus } from '@nestjs/common';
import { Response } from 'express';

@Controller('cats')
export class CatsController {
  @Post()
  create(@Res() res: Response) {
    res.status(HttpStatus.CREATED).send();
  }

  @Get()
  findAll(@Res() res: Response) {
    res.status(HttpStatus.OK).json([]);
  }
}

@@switch
import { Controller, Get, Post, Bind, Res, Body, HttpStatus } from
'@nestjs/common';

@Controller('cats')
export class CatsController {
  @Post()
  @Bind(Res(), Body())
  create(res, createCatDto) {
    res.status(HttpStatus.CREATED).send();
  }
}
```

```
}

@Get()
@Bind(Res())
findAll(res) {
  res.status(HttpStatus.OK).json([]);
}
}
```

Though this approach works, and does in fact allow for more flexibility in some ways by providing full control of the response object (headers manipulation, library-specific features, and so on), it should be used with care. In general, the approach is much less clear and does have some disadvantages. The main disadvantage is that your code becomes platform-dependent (as underlying libraries may have different APIs on the response object), and harder to test (you'll have to mock the response object, etc.).

Also, in the example above, you lose compatibility with Nest features that depend on Nest standard response handling, such as Interceptors and `@HttpCode()` / `@Header()` decorators. To fix this, you can set the `passthrough` option to `true`, as follows:

```
@@filename()
@Get()
findAll(@Res({ passthrough: true }) res: Response) {
  res.status(HttpStatus.OK);
  return [];
}
@@switch
@Get()
@Bind(Res({ passthrough: true }))
findAll(res) {
  res.status(HttpStatus.OK);
  return [];
}
```

Now you can interact with the native response object (for example, set cookies or headers depending on certain conditions), but leave the rest to the framework.

## Modules

A module is a class annotated with a `@Module()` decorator. The `@Module()` decorator provides metadata that **Nest** makes use of to organize the application structure.



Each application has at least one module, a **root module**. The root module is the starting point Nest uses to build the **application graph** - the internal data structure Nest uses to resolve module and provider relationships and dependencies. While very small applications may theoretically have just the root module, this is not the typical case. We want to emphasize that modules are **strongly recommended** as an effective way to organize your components. Thus, for most applications, the resulting architecture will employ multiple modules, each encapsulating a closely related set of **capabilities**.

The `@Module()` decorator takes a single object whose properties describe the module:

<code>providers</code>	the providers that will be instantiated by the Nest injector and that may be shared at least across this module
<code>controllers</code>	the set of controllers defined in this module which have to be instantiated
<code>imports</code>	the list of imported modules that export the providers which are required in this module
<code>exports</code>	the subset of <code>providers</code> that are provided by this module and should be available in other modules which import this module. You can use either the provider itself or just its token ( <code>provide</code> value)

The module **encapsulates** providers by default. This means that it's impossible to inject providers that are neither directly part of the current module nor exported from the imported modules. Thus, you may consider the exported providers from a module as the module's public interface, or API.

## Feature modules

The `CatsController` and `CatsService` belong to the same application domain. As they are closely related, it makes sense to move them into a feature module. A feature module simply organizes code relevant for a specific feature, keeping code organized and establishing clear boundaries. This helps us manage complexity and develop with **SOLID** principles, especially as the size of the application and/or team grow.

To demonstrate this, we'll create the `CatsModule`.

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

@Module({
  controllers: [CatsController],
```

```
    providers: [CatsService],  
})  
export class CatsModule {}
```

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

Above, we defined the `CatsModule` in the `cats.module.ts` file, and moved everything related to this module into the `cats` directory. The last thing we need to do is import this module into the root module (the `AppModule`, defined in the `app.module.ts` file).

```
@@filename(app.module)  
import { Module } from '@nestjs/common';  
import { CatsModule } from './cats/cats.module';  
  
@Module({  
  imports: [CatsModule],  
})  
export class AppModule {}
```

Here is how our directory structure looks now:

```
src  
  cats  
    dto  
      create-cat.dto.ts  
    interfaces  
      cat.interface.ts  
    controller.ts  
    module.ts  
    service.ts  
  app.module.ts  
  main.ts
```

## Shared modules

In Nest, modules are **singletons** by default, and thus you can share the same instance of any provider between multiple modules effortlessly.



Every module is automatically a **shared module**. Once created it can be reused by any module. Let's imagine that we want to share an instance of the `CatsService` between several other modules. In order to do that, we first need to **export** the `CatsService` provider by adding it to the module's `exports` array, as shown below:

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

```
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService]
})
export class CatsModule {}
```

Now any module that imports the `CatsModule` has access to the `CatsService` and will share the same instance with all other modules that import it as well.

## Module re-exporting

As seen above, Modules can export their internal providers. In addition, they can re-export modules that they import. In the example below, the `CommonModule` is both imported into **and** exported from the `CoreModule`, making it available for other modules which import this one.

```
@Module({
  imports: [CommonModule],
  exports: [CommonModule],
})
export class CoreModule {}
```

## Dependency injection

A module class can **inject** providers as well (e.g., for configuration purposes):

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

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {
  constructor(private catsService: CatsService) {}
}

@@switch
import { Module, Dependencies } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
```

```

    providers: [CatsService],
})
@Dependencies(CatsService)
export class CatsModule {
  constructor(catsService) {
    this.catsService = catsService;
  }
}

```

However, module classes themselves cannot be injected as providers due to [circular dependency](#).

## Global modules

If you have to import the same set of modules everywhere, it can get tedious. Unlike in Nest, [Angular providers](#) are registered in the global scope. Once defined, they're available everywhere. Nest, however, encapsulates providers inside the module scope. You aren't able to use a module's providers elsewhere without first importing the encapsulating module.

When you want to provide a set of providers which should be available everywhere out-of-the-box (e.g., helpers, database connections, etc.), make the module **global** with the [@Global\(\)](#) decorator.

```

import { Module, Global } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Global()
@Module({
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService],
})
export class CatsModule {}

```

The [@Global\(\)](#) decorator makes the module global-scoped. Global modules should be registered **only once**, generally by the root or core module. In the above example, the [CatsService](#) provider will be ubiquitous, and modules that wish to inject the service will not need to import the [CatsModule](#) in their imports array.

**info Hint** Making everything global is not a good design decision. Global modules are available to reduce the amount of necessary boilerplate. The [imports](#) array is generally the preferred way to make the module's API available to consumers.

## Dynamic modules

The Nest module system includes a powerful feature called **dynamic modules**. This feature enables you to easily create customizable modules that can register and configure providers dynamically. Dynamic modules are covered extensively [here](#). In this chapter, we'll give a brief overview to complete the introduction to modules.

Following is an example of a dynamic module definition for a `DatabaseModule`:

```
@@filename()
import { Module, DynamicModule } from '@nestjs/common';
import { createDatabaseProviders } from './database.providers';
import { Connection } from './connection.provider';

@Module({
  providers: [Connection],
})
export class DatabaseModule {
  static forRoot(entities = [], options?): DynamicModule {
    const providers = createDatabaseProviders(options, entities);
    return {
      module: DatabaseModule,
      providers,
      exports: providers,
    };
  }
}
@@switch
import { Module } from '@nestjs/common';
import { createDatabaseProviders } from './database.providers';
import { Connection } from './connection.provider';

@Module({
  providers: [Connection],
})
export class DatabaseModule {
  static forRoot(entities = [], options) {
    const providers = createDatabaseProviders(options, entities);
    return {
      module: DatabaseModule,
      providers,
      exports: providers,
    };
  }
}
```

**info Hint** The `forRoot()` method may return a dynamic module either synchronously or asynchronously (i.e., via a `Promise`).

This module defines the `Connection` provider by default (in the `@Module()` decorator metadata), but additionally - depending on the `entities` and `options` objects passed into the `forRoot()` method - exposes a collection of providers, for example, repositories. Note that the properties returned by the dynamic module `extend` (rather than override) the base module metadata defined in the `@Module()` decorator. That's how both the statically declared `Connection` provider **and** the dynamically generated repository providers are exported from the module.

If you want to register a dynamic module in the global scope, set the `global` property to `true`.

```
{  
  global: true,  
  module: DatabaseModule,  
  providers: providers,  
  exports: providers,  
}
```

warning **Warning** As mentioned above, making everything global **is not a good design decision**.

The **DatabaseModule** can be imported and configured in the following manner:

```
import { Module } from '@nestjs/common';  
import { DatabaseModule } from './database/database.module';  
import { User } from './users/entities/user.entity';  
  
@Module({  
  imports: [DatabaseModule.forRoot([User])],  
})  
export class AppModule {}
```

If you want to in turn re-export a dynamic module, you can omit the **forRoot()** method call in the exports array:

```
import { Module } from '@nestjs/common';  
import { DatabaseModule } from './database/database.module';  
import { User } from './users/entities/user.entity';  
  
@Module({  
  imports: [DatabaseModule.forRoot([User])],  
  exports: [DatabaseModule],  
})  
export class AppModule {}
```

The [Dynamic modules](#) chapter covers this topic in greater detail, and includes a working example.

info **Hint** Learn how to build highly customizable dynamic modules with the use of **ConfigurableModuleBuilder** here in [this chapter](#).

## Middleware

Middleware is a function which is called **before** the route handler. Middleware functions have access to the `request` and `response` objects, and the `next()` middleware function in the application's request-response cycle. The `next` middleware function is commonly denoted by a variable named `next`.



Nest middleware are, by default, equivalent to `express` middleware. The following description from the official express documentation describes the capabilities of middleware:

Middleware functions can perform the following tasks:

- execute any code.
- make changes to the request and the response objects.
- end the request-response cycle.
- call the next middleware function in the stack.
- if the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

You implement custom Nest middleware in either a function, or in a class with an `@Injectable()` decorator. The class should implement the `NestMiddleware` interface, while the function does not have any special requirements. Let's start by implementing a simple middleware feature using the class method.

warning **Warning** `Express` and `fastify` handle middleware differently and provide different method signatures, read more [here](#).

```
@@filename(logger.middleware)
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log('Request...');
    next();
  }
}

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

@Injectable()
export class LoggerMiddleware {
  use(req, res, next) {
    console.log('Request...');
    next();
  }
}
```

## Dependency injection

Nest middleware fully supports Dependency Injection. Just as with providers and controllers, they are able to **inject dependencies** that are available within the same module. As usual, this is done through the **constructor**.

## Applying middleware

There is no place for middleware in the `@Module()` decorator. Instead, we set them up using the `configure()` method of the module class. Modules that include middleware have to implement the `NestModule` interface. Let's set up the `LoggerMiddleware` at the `AppModule` level.

```
@@filename(app.module)
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('cats');
  }
}
@@switch
import { Module } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule {
  configure(consumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('cats');
  }
}
```

In the above example we have set up the `LoggerMiddleware` for the `/cats` route handlers that were previously defined inside the `CatsController`. We may also further restrict a middleware to a particular request method by passing an object containing the route `path` and request `method` to the `forRoutes()` method when configuring the middleware. In the example below, notice that we import the `RequestMethod` enum to reference the desired request method type.

```
@@filename(app.module)
import { Module, NestModule, RequestMethod, MiddlewareConsumer } from
'@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({ path: 'cats', method: RequestMethod.GET });
  }
}

@switch
import { Module, RequestMethod } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule {
  configure(consumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({ path: 'cats', method: RequestMethod.GET });
  }
}
```

**info Hint** The `configure()` method can be made asynchronous using `async/await` (e.g., you can `await` completion of an asynchronous operation inside the `configure()` method body).

**warning Warning** When using the `express` adapter, the NestJS app will register `json` and `urlencoded` from the package `body-parser` by default. This means if you want to customize that middleware via the `MiddlewareConsumer`, you need to turn off the global middleware by setting the `bodyParser` flag to `false` when creating the application with `NestFactory.create()`.

## Route wildcards

Pattern based routes are supported as well. For instance, the asterisk is used as a **wildcard**, and will match any combination of characters:

```
forRoutes({ path: 'ab*cd', method: RequestMethod.ALL });
```

The '`ab*cd`' route path will match `abcd`, `ab_cd`, `abecd`, and so on. The characters `?`, `+`, `*`, and `( )` may be used in a route path, and are subsets of their regular expression counterparts. The hyphen (`-`) and the dot (`.`) are interpreted literally by string-based paths.

warning **Warning** The `fastify` package uses the latest version of the `path-to-regexp` package, which no longer supports wildcard asterisks `*`. Instead, you must use parameters (e.g., `(.*)`, `:splat*`).

## Middleware consumer

The `MiddlewareConsumer` is a helper class. It provides several built-in methods to manage middleware. All of them can be simply **chained** in the **fluent style**. The `forRoutes()` method can take a single string, multiple strings, a `RouteInfo` object, a controller class and even multiple controller classes. In most cases you'll probably just pass a list of **controllers** separated by commas. Below is an example with a single controller:

```
@@filename(app.module)
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';
import { CatsController } from './cats/cats.controller';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes(CatsController);
  }
}
@switch
import { Module } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';
import { CatsController } from './cats/cats.controller';

@Module({
  imports: [CatsModule],
})
export class AppModule {
  configure(consumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes(CatsController);
  }
}
```

**info Hint** The `apply()` method may either take a single middleware, or multiple arguments to specify [multiple middlewares](#).

## Excluding routes

At times we want to **exclude** certain routes from having the middleware applied. We can easily exclude certain routes with the `exclude()` method. This method can take a single string, multiple strings, or a [RouteInfo](#) object identifying routes to be excluded, as shown below:

```
consumer
  .apply(LoggerMiddleware)
  .exclude(
    { path: 'cats', method: RequestMethod.GET },
    { path: 'cats', method: RequestMethod.POST },
    'cats/(.*)',
  )
  .forRoutes(CatsController);
```

**info Hint** The `exclude()` method supports wildcard parameters using the [path-to-regexp](#) package.

With the example above, `LoggerMiddleware` will be bound to all routes defined inside `CatsController` **except** the three passed to the `exclude()` method.

## Functional middleware

The `LoggerMiddleware` class we've been using is quite simple. It has no members, no additional methods, and no dependencies. Why can't we just define it in a simple function instead of a class? In fact, we can. This type of middleware is called **functional middleware**. Let's transform the logger middleware from class-based into functional middleware to illustrate the difference:

```
@@filename(logger.middleware)
import { Request, Response, NextFunction } from 'express';

export function logger(req: Request, res: Response, next: NextFunction) {
  console.log(`Request...`);
  next();
};

@@switch
export function logger(req, res, next) {
  console.log(`Request...`);
  next();
};
```

And use it within the `AppModule`:

```
@@filename(app.module)
consumer
```

```
.apply(logger)
  .forRoutes(CatsController);
```

**info Hint** Consider using the simpler **functional middleware** alternative any time your middleware doesn't need any dependencies.

## Multiple middleware

As mentioned above, in order to bind multiple middleware that are executed sequentially, simply provide a comma separated list inside the `apply()` method:

```
consumer.apply(cors(), helmet(), logger).forRoutes(CatsController);
```

## Global middleware

If we want to bind middleware to every registered route at once, we can use the `use()` method that is supplied by the `INestApplication` instance:

```
@@filename(main)
const app = await NestFactory.create(AppModule);
app.use(logger);
await app.listen(3000);
```

**info Hint** Accessing the DI container in a global middleware is not possible. You can use a **functional middleware** instead when using `app.use()`. Alternatively, you can use a class middleware and consume it with `.forRoutes('*')` within the `AppModule` (or any other module).

## Exception filters

Nest comes with a built-in **exceptions layer** which is responsible for processing all unhandled exceptions across an application. When an exception is not handled by your application code, it is caught by this layer, which then automatically sends an appropriate user-friendly response.



Out of the box, this action is performed by a built-in **global exception filter**, which handles exceptions of type **HttpException** (and subclasses of it). When an exception is **unrecognized** (is neither **HttpException** nor a class that inherits from **HttpException**), the built-in exception filter generates the following default JSON response:

```
{  
  "statusCode": 500,  
  "message": "Internal server error"  
}
```

**info Hint** The global exception filter partially supports the **http-errors** library. Basically, any thrown exception containing the **statusCode** and **message** properties will be properly populated and sent back as a response (instead of the default **InternalServerErrorException** for unrecognized exceptions).

## Throwing standard exceptions

Nest provides a built-in **HttpException** class, exposed from the **@nestjs/common** package. For typical HTTP REST/GraphQL API based applications, it's best practice to send standard HTTP response objects when certain error conditions occur.

For example, in the **CatsController**, we have a **findAll()** method (a **GET** route handler). Let's assume that this route handler throws an exception for some reason. To demonstrate this, we'll hard-code it as follows:

```
@@filename(cats.controller)  
@Get()  
async findAll() {  
  throw new HttpException('Forbidden', HttpStatus.FORBIDDEN);  
}
```

**info Hint** We used the **HttpStatus** here. This is a helper enum imported from the **@nestjs/common** package.

When the client calls this endpoint, the response looks like this:

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

The `HttpException` constructor takes two required arguments which determine the response:

- The `response` argument defines the JSON response body. It can be a `string` or an `object` as described below.
- The `status` argument defines the `HTTP status code`.

By default, the JSON response body contains two properties:

- `statusCode`: defaults to the HTTP status code provided in the `status` argument
- `message`: a short description of the HTTP error based on the `status`

To override just the message portion of the JSON response body, supply a string in the `response` argument. To override the entire JSON response body, pass an object in the `response` argument. Nest will serialize the object and return it as the JSON response body.

The second constructor argument - `status` - should be a valid HTTP status code. Best practice is to use the `HttpStatus` enum imported from `@nestjs/common`.

There is a **third** constructor argument (optional) - `options` - that can be used to provide an error `cause`. This `cause` object is not serialized into the response object, but it can be useful for logging purposes, providing valuable information about the inner error that caused the `HttpException` to be thrown.

Here's an example overriding the entire response body and providing an error cause:

```
@@filename(cats.controller)
@Get()
async findAll() {
  try {
    await this.service.findAll()
  } catch (error) {
    throw new HttpException({
      status: HttpStatus.FORBIDDEN,
      error: 'This is a custom message',
    }, HttpStatus.FORBIDDEN, {
      cause: error
    });
  }
}
```

Using the above, this is how the response would look:

```
{
  "status": 403,
```

```
    "error": "This is a custom message"
}
```

## Custom exceptions

In many cases, you will not need to write custom exceptions, and can use the built-in Nest HTTP exception, as described in the next section. If you do need to create customized exceptions, it's good practice to create your own **exceptions hierarchy**, where your custom exceptions inherit from the base `HttpException` class. With this approach, Nest will recognize your exceptions, and automatically take care of the error responses. Let's implement such a custom exception:

```
@@filename(forbidden.exception)
export class ForbiddenException extends HttpException {
  constructor() {
    super('Forbidden', HttpStatus.FORBIDDEN);
  }
}
```

Since `ForbiddenException` extends the base `HttpException`, it will work seamlessly with the built-in exception handler, and therefore we can use it inside the `findAll()` method.

```
@@filename(cats.controller)
@Get()
async findAll() {
  throw new ForbiddenException();
}
```

## Built-in HTTP exceptions

Nest provides a set of standard exceptions that inherit from the base `HttpException`. These are exposed from the `@nestjs/common` package, and represent many of the most common HTTP exceptions:

- `BadRequestException`
- `UnauthorizedException`
- `NotFoundException`
- `ForbiddenException`
- `NotAcceptableException`
- `RequestTimeoutException`
- `ConflictException`
- `GoneException`
- `HttpVersionNotSupportedException`
- `PayloadTooLargeException`
- `UnsupportedMediaTypeException`
- `UnprocessableEntityException`
- `InternalServerErrorException`

- `NotImplementedException`
- `ImATeapotException`
- `MethodNotAllowedException`
- `BadGatewayException`
- `ServiceUnavailableException`
- `GatewayTimeoutException`
- `PreconditionFailedException`

All the built-in exceptions can also provide both an error `cause` and an error description using the `options` parameter:

```
throw new BadRequestException('Something bad happened', { cause: new Error(), description: 'Some error description' })
```

Using the above, this is how the response would look:

```
{
  "message": "Something bad happened",
  "error": "Some error description",
  "statusCode": 400,
}
```

## Exception filters

While the base (built-in) exception filter can automatically handle many cases for you, you may want **full control** over the exceptions layer. For example, you may want to add logging or use a different JSON schema based on some dynamic factors. **Exception filters** are designed for exactly this purpose. They let you control the exact flow of control and the content of the response sent back to the client.

Let's create an exception filter that is responsible for catching exceptions which are an instance of the `HttpException` class, and implementing custom response logic for them. To do this, we'll need to access the underlying platform `Request` and `Response` objects. We'll access the `Request` object so we can pull out the original `url` and include that in the logging information. We'll use the `Response` object to take direct control of the response that is sent, using the `response.json()` method.

```
@filename(http-exception.filter)
import { ExceptionFilter, Catch, ArgumentsHost, HttpException } from
  '@nestjs/common';
import { Request, Response } from 'express';

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();
```

```
const status = exception.getStatus();

response
  .status(status)
  .json({
    statusCode: status,
    timestamp: new Date().toISOString(),
    path: request.url,
  });
}

}

@switch
import { Catch, HttpException } from '@nestjs/common';

@Catch(HttpException)
export class HttpExceptionFilter {
  catch(exception, host) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();
    const request = ctx.getRequest();
    const status = exception.getStatus();

    response
      .status(status)
      .json({
        statusCode: status,
        timestamp: new Date().toISOString(),
        path: request.url,
      });
  }
}
```

**info Hint** All exception filters should implement the generic `ExceptionFilter<T>` interface. This requires you to provide the `catch(exception: T, host: ArgumentsHost)` method with its indicated signature. `T` indicates the type of the exception.

**warning Warning** If you are using `@nestjs/platform-fastify` you can use `response.send()` instead of `response.json()`. Don't forget to import the correct types from `fastify`.

The `@Catch(HttpException)` decorator binds the required metadata to the exception filter, telling Nest that this particular filter is looking for exceptions of type `HttpException` and nothing else. The `@Catch()` decorator may take a single parameter, or a comma-separated list. This lets you set up the filter for several types of exceptions at once.

## Arguments host

Let's look at the parameters of the `catch()` method. The `exception` parameter is the exception object currently being processed. The `host` parameter is an `ArgumentsHost` object. `ArgumentsHost` is a powerful utility object that we'll examine further in the [execution context chapter\\*](#). In this code sample, we use it to obtain a reference to the `Request` and `Response` objects that are being passed to the original request handler (in the controller where the exception originates). In this code sample, we've used some

helper methods on `ArgumentsHost` to get the desired `Request` and `Response` objects. Learn more about `ArgumentsHost` [here](#).

\*The reason for this level of abstraction is that `ArgumentsHost` functions in all contexts (e.g., the HTTP server context we're working with now, but also Microservices and WebSockets). In the execution context chapter we'll see how we can access the appropriate [underlying arguments](#) for **any** execution context with the power of `ArgumentsHost` and its helper functions. This will allow us to write generic exception filters that operate across all contexts.

## Binding filters

Let's tie our new `HttpExceptionFilter` to the `CatsController`'s `create()` method.

```
@@filename(cats.controller)
@Post()
@UseFilters(new HttpExceptionFilter())
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
@switch
@Post()
@UseFilters(new HttpExceptionFilter())
@Bind(Body())
async create(createCatDto) {
  throw new ForbiddenException();
}
```

**info Hint** The `@UseFilters()` decorator is imported from the `@nestjs/common` package.

We have used the `@UseFilters()` decorator here. Similar to the `@Catch()` decorator, it can take a single filter instance, or a comma-separated list of filter instances. Here, we created the instance of `HttpExceptionFilter` in place. Alternatively, you may pass the class (instead of an instance), leaving responsibility for instantiation to the framework, and enabling **dependency injection**.

```
@@filename(cats.controller)
@Post()
@UseFilters(HttpExceptionFilter)
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
@switch
@Post()
@UseFilters(HttpExceptionFilter)
@Bind(Body())
async create(createCatDto) {
  throw new ForbiddenException();
}
```

**info Hint** Prefer applying filters by using classes instead of instances when possible. It reduces **memory usage** since Nest can easily reuse instances of the same class across your entire module.

In the example above, the `HttpExceptionFilter` is applied only to the single `create()` route handler, making it method-scoped. Exception filters can be scoped at different levels: method-scoped of the controller/resolver/gateway, controller-scoped, or global-scoped.

For example, to set up a filter as controller-scoped, you would do the following:

```
@@filename(cats.controller)
@UseFilters(new HttpExceptionFilter())
export class CatsController {}
```

This construction sets up the `HttpExceptionFilter` for every route handler defined inside the `CatsController`.

To create a global-scoped filter, you would do the following:

```
@@filename(main)
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalFilters(new HttpExceptionFilter());
  await app.listen(3000);
}
bootstrap();
```

**warning Warning** The `useGlobalFilters()` method does not set up filters for gateways or hybrid applications.

Global-scoped filters are used across the whole application, for every controller and every route handler. In terms of dependency injection, global filters registered from outside of any module (with `useGlobalFilters()` as in the example above) cannot inject dependencies since this is done outside the context of any module. In order to solve this issue, you can register a global-scoped filter **directly from any module** using the following construction:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { APP_FILTER } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_FILTER,
      useClass: HttpExceptionFilter,
    },
  ],
})
export class AppModule {}
```

**info Hint** When using this approach to perform dependency injection for the filter, note that regardless of the module where this construction is employed, the filter is, in fact, global. Where should this be done? Choose the module where the filter (`HttpExceptionFilter` in the example above) is defined. Also, `useClass` is not the only way of dealing with custom provider registration. Learn more [here](#).

You can add as many filters with this technique as needed; simply add each to the providers array.

## Catch everything

In order to catch **every** unhandled exception (regardless of the exception type), leave the `@Catch()` decorator's parameter list empty, e.g., `@Catch()`.

In the example below we have a code that is platform-agnostic because it uses the `HTTP adapter` to deliver the response, and doesn't use any of the platform-specific objects (`Request` and `Response`) directly:

```
import {
  ExceptionFilter,
  Catch,
  ArgumentsHost,
  HttpException,
  HttpStatus,
} from '@nestjs/common';
import { HttpAdapterHost } from '@nestjs/core';

@Catch()
export class AllExceptionsFilter implements ExceptionFilter {
  constructor(private readonly httpAdapterHost: HttpAdapterHost) {}

  catch(exception: unknown, host: ArgumentsHost): void {
    // In certain situations `httpAdapter` might not be available in the
    // constructor method, thus we should resolve it here.
    const { httpAdapter } = this.httpAdapterHost;

    const ctx = host.switchToHttp();

    const httpResponse =
      exception instanceof HttpException
        ? exception.getStatus()
        : HttpStatus.INTERNAL_SERVER_ERROR;

    const responseBody = {
      statusCode: httpResponse,
      timestamp: new Date().toISOString(),
      path: httpAdapter.getRequestUrl(ctx.getRequest()),
    };

    httpAdapter.reply(ctx.getResponse(), responseBody, httpResponse);
  }
}
```

warning **Warning** When combining an exception filter that catches everything with a filter that is bound to a specific type, the "Catch anything" filter should be declared first to allow the specific filter to correctly handle the bound type.

## Inheritance

Typically, you'll create fully customized exception filters crafted to fulfill your application requirements. However, there might be use-cases when you would like to simply extend the built-in default **global exception filter**, and override the behavior based on certain factors.

In order to delegate exception processing to the base filter, you need to extend `BaseExceptionFilter` and call the inherited `catch()` method.

```
@@filename(all-exceptions.filter)
import { Catch, ArgumentsHost } from '@nestjs/common';
import { BaseExceptionFilter } from '@nestjs/core';

@Catch()
export class AllExceptionsFilter extends BaseExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    super.catch(exception, host);
  }
}

@@switch
import { Catch } from '@nestjs/common';
import { BaseExceptionFilter } from '@nestjs/core';

@Catch()
export class AllExceptionsFilter extends BaseExceptionFilter {
  catch(exception, host) {
    super.catch(exception, host);
  }
}
```

warning **Warning** Method-scoped and Controller-scoped filters that extend the `BaseExceptionFilter` should not be instantiated with `new`. Instead, let the framework instantiate them automatically.

The above implementation is just a shell demonstrating the approach. Your implementation of the extended exception filter would include your tailored **business** logic (e.g., handling various conditions).

Global filters **can** extend the base filter. This can be done in either of two ways.

The first method is to inject the `HttpAdapter` reference when instantiating the custom global filter:

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

```
const { httpAdapter } = app.get(HttpAdapterHost);
app.useGlobalFilters(new AllExceptionsFilter(httpAdapter));

await app.listen(3000);
}
bootstrap();
```

The second method is to use the [APP\\_FILTER](#) token as shown here.

## Pipes

A pipe is a class annotated with the `@Injectable()` decorator, which implements the `PipeTransform` interface.



Pipes have two typical use cases:

- **transformation**: transform input data to the desired form (e.g., from string to integer)
- **validation**: evaluate input data and if valid, simply pass it through unchanged; otherwise, throw an exception

In both cases, pipes operate on the `arguments` being processed by a controller route handler. Nest interposes a pipe just before a method is invoked, and the pipe receives the arguments destined for the method and operates on them. Any transformation or validation operation takes place at that time, after which the route handler is invoked with any (potentially) transformed arguments.

Nest comes with a number of built-in pipes that you can use out-of-the-box. You can also build your own custom pipes. In this chapter, we'll introduce the built-in pipes and show how to bind them to route handlers. We'll then examine several custom-built pipes to show how you can build one from scratch.

**info Hint** Pipes run inside the exceptions zone. This means that when a Pipe throws an exception it is handled by the exceptions layer (global exceptions filter and any `exceptions filters` that are applied to the current context). Given the above, it should be clear that when an exception is thrown in a Pipe, no controller method is subsequently executed. This gives you a best-practice technique for validating data coming into the application from external sources at the system boundary.

### Built-in pipes

Nest comes with nine pipes available out-of-the-box:

- `ValidationPipe`
- `ParseIntPipe`
- `ParseFloatPipe`
- `ParseBoolPipe`
- `ParseArrayPipe`
- `ParseUUIDPipe`
- `ParseEnumPipe`
- `DefaultValuePipe`
- `ParseFilePipe`

They're exported from the `@nestjs/common` package.

Let's take a quick look at using `ParseIntPipe`. This is an example of the **transformation** use case, where the pipe ensures that a method handler parameter is converted to a JavaScript integer (or throws an exception if the conversion fails). Later in this chapter, we'll show a simple custom implementation for a `ParseIntPipe`. The example techniques below also apply to the other built-in transformation pipes

(`ParseBoolPipe`, `ParseFloatPipe`, `ParseEnumPipe`, `ParseArrayPipe` and `ParseUUIDPipe`, which we'll refer to as the `Parse*` pipes in this chapter).

## Binding pipes

To use a pipe, we need to bind an instance of the pipe class to the appropriate context. In our `ParseIntPipe` example, we want to associate the pipe with a particular route handler method, and make sure it runs before the method is called. We do so with the following construct, which we'll refer to as binding the pipe at the method parameter level:

```
@Get(':id')
async findOne(@Param('id', ParseIntPipe) id: number) {
  return this.catsService.findOne(id);
}
```

This ensures that one of the following two conditions is true: either the parameter we receive in the `findOne()` method is a number (as expected in our call to `this.catsService.findOne()`), or an exception is thrown before the route handler is called.

For example, assume the route is called like:

```
GET localhost:3000/abc
```

Nest will throw an exception like this:

```
{
  "statusCode": 400,
  "message": "Validation failed (numeric string is expected)",
  "error": "Bad Request"
}
```

The exception will prevent the body of the `findOne()` method from executing.

In the example above, we pass a class (`ParseIntPipe`), not an instance, leaving responsibility for instantiation to the framework and enabling dependency injection. As with pipes and guards, we can instead pass an in-place instance. Passing an in-place instance is useful if we want to customize the built-in pipe's behavior by passing options:

```
@Get(':id')
async findOne(
  @Param('id', new ParseIntPipe({ errorHttpStatusCode:
    HttpStatus.NOT_ACCEPTABLE }))
  id: number,
) {
```

```
    return this.catsService.findOne(id);
}
```

Binding the other transformation pipes (all of the **Parse\*** pipes) works similarly. These pipes all work in the context of validating route parameters, query string parameters and request body values.

For example with a query string parameter:

```
@Get()
async findOne(@Query('id', ParseIntPipe) id: number) {
  return this.catsService.findOne(id);
}
```

Here's an example of using the **ParseUUIDPipe** to parse a string parameter and validate if it is a UUID.

```
@@filename()
@Get(':uuid')
async findOne(@Param('uuid', new ParseUUIDPipe()) uuid: string) {
  return this.catsService.findOne(uuid);
}
@@switch
@Get(':uuid')
@Bind(Param('uuid', new ParseUUIDPipe()))
async findOne(uuid) {
  return this.catsService.findOne(uuid);
}
```

**info Hint** When using **ParseUUIDPipe()** you are parsing UUID in version 3, 4 or 5, if you only require a specific version of UUID you can pass a version in the pipe options.

Above we've seen examples of binding the various **Parse\*** family of built-in pipes. Binding validation pipes is a little bit different; we'll discuss that in the following section.

**info Hint** Also, see [Validation techniques](#) for extensive examples of validation pipes.

## Custom pipes

As mentioned, you can build your own custom pipes. While Nest provides a robust built-in **ParseIntPipe** and **ValidationPipe**, let's build simple custom versions of each from scratch to see how custom pipes are constructed.

We start with a simple **ValidationPipe**. Initially, we'll have it simply take an input value and immediately return the same value, behaving like an identity function.

```
@@filename(validation.pipe)
import { PipeTransform, Injectable, ArgumentMetadata } from
```

```
'@nestjs/common';

@Injectable()
export class ValidationPipe implements PipeTransform {
  transform(value: any, metadata: ArgumentMetadata) {
    return value;
  }
}

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

@Injectable()
export class ValidationPipe {
  transform(value, metadata) {
    return value;
  }
}
```

**info Hint** `PipeTransform<T, R>` is a generic interface that must be implemented by any pipe. The generic interface uses `T` to indicate the type of the input `value`, and `R` to indicate the return type of the `transform()` method.

Every pipe must implement the `transform()` method to fulfill the `PipeTransform` interface contract. This method has two parameters:

- `value`
- `metadata`

The `value` parameter is the currently processed method argument (before it is received by the route handling method), and `metadata` is the currently processed method argument's metadata. The metadata object has these properties:

```
export interface ArgumentMetadata {
  type: 'body' | 'query' | 'param' | 'custom';
  metatype?: Type<unknown>;
  data?: string;
}
```

These properties describe the currently processed argument.

<code>type</code>	Indicates whether the argument is a body <code>@Body()</code> , query <code>@Query()</code> , param <code>@Param()</code> , or a custom parameter (read more here).
<code>metatype</code>	Provides the metatype of the argument, for example, <code>String</code> . Note: the value is <code>undefined</code> if you either omit a type declaration in the route handler method signature, or use vanilla JavaScript.
<code>data</code>	The string passed to the decorator, for example <code>@Body('string')</code> . It's <code>undefined</code> if you leave the decorator parenthesis empty.

warning **Warning** TypeScript interfaces disappear during transpilation. Thus, if a method parameter's type is declared as an interface instead of a class, the `metatype` value will be `Object`.

## Schema based validation

Let's make our validation pipe a little more useful. Take a closer look at the `create()` method of the `CatsController`, where we probably would like to ensure that the post body object is valid before attempting to run our service method.

```
@@filename()
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}

@@switch
@Post()
async create(@Body() createCatDto) {
  this.catsService.create(createCatDto);
}
```

Let's focus in on the `createCatDto` body parameter. Its type is `CreateCatDto`:

```
@@filename(create-cat.dto)
export class CreateCatDto {
  name: string;
  age: number;
  breed: string;
}
```

We want to ensure that any incoming request to the `create` method contains a valid body. So we have to validate the three members of the `createCatDto` object. We could do this inside the route handler method, but doing so is not ideal as it would break the **single responsibility rule** (SRP).

Another approach could be to create a **validator class** and delegate the task there. This has the disadvantage that we would have to remember to call this validator at the beginning of each method.

How about creating validation middleware? This could work, but unfortunately, it's not possible to create **generic middleware** which can be used across all contexts across the whole application. This is because middleware is unaware of the **execution context**, including the handler that will be called and any of its parameters.

This is, of course, exactly the use case for which pipes are designed. So let's go ahead and refine our validation pipe.

## Object schema validation

There are several approaches available for doing object validation in a clean, **DRY** way. One common approach is to use **schema-based** validation. Let's go ahead and try that approach.

The [Zod](#) library allows you to create schemas in a straightforward way, with a readable API. Let's build a validation pipe that makes use of Zod-based schemas.

Start by installing the required package:

```
$ npm install --save zod
```

In the code sample below, we create a simple class that takes a schema as a **constructor** argument. We then apply the **schema.parse()** method, which validates our incoming argument against the provided schema.

As noted earlier, a **validation pipe** either returns the value unchanged or throws an exception.

In the next section, you'll see how we supply the appropriate schema for a given controller method using the **@UsePipes()** decorator. Doing so makes our validation pipe reusable across contexts, just as we set out to do.

```
@@filename()
import { PipeTransform, ArgumentMetadata, BadRequestException } from
'@nestjs/common';
import { ZodObject } from 'zod';

export class ZodValidationPipe implements PipeTransform {
  constructor(private schema: ZodObject<any>) {}

  transform(value: unknown, metadata: ArgumentMetadata) {
    try {
      this.schema.parse(value);
    } catch (error) {
      throw new BadRequestException('Validation failed');
    }
    return value;
  }
}

@@switch
import { BadRequestException } from '@nestjs/common';
import { ZodObject } from 'zod';

export class ZodValidationPip {
  constructor(private schema) {}

  transform(value, metadata) {
    try {
      this.schema.parse(value);
    } catch (error) {
      throw new BadRequestException('Validation failed');
    }
  }
}
```

```

        return value;
    }
}

```

## Binding validation pipes

Earlier, we saw how to bind transformation pipes (like `ParseIntPipe` and the rest of the `Parse*` pipes).

Binding validation pipes is also very straightforward.

In this case, we want to bind the pipe at the method call level. In our current example, we need to do the following to use the `ZodValidationPipe`:

1. Create an instance of the `ZodValidationPipe`
2. Pass the context-specific Zod schema in the class constructor of the pipe
3. Bind the pipe to the method

Zod schema example:

```

import { z } from 'zod';

export const createCatSchema = z
  .object({
    name: z.string(),
    age: z.number(),
    breed: z.string(),
  })
  .required();

export type CreateCatDto = z.infer<typeof createCatSchema>;

```

We do that using the `@UsePipes()` decorator as shown below:

```

@@filename(cats.controller)
@Post()
@UsePipes(new ZodValidationPipe(createCatSchema))
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}

@@switch
@Post()
@Bind(Body())
@UsePipes(new ZodValidationPipe(createCatSchema))
async create(createCatDto) {
  this.catsService.create(createCatDto);
}

```

info Hint The `@UsePipes()` decorator is imported from the `@nestjs/common` package.

warning Warning `zod` library requires the `strictNullChecks` configuration to be enabled in your `tsconfig.json` file.

## Class validator

warning Warning The techniques in this section require TypeScript and are not available if your app is written using vanilla JavaScript.

Let's look at an alternate implementation for our validation technique.

Nest works well with the `class-validator` library. This powerful library allows you to use decorator-based validation. Decorator-based validation is extremely powerful, especially when combined with Nest's **Pipe** capabilities since we have access to the `metatype` of the processed property. Before we start, we need to install the required packages:

```
$ npm i --save class-validator class-transformer
```

Once these are installed, we can add a few decorators to the `CreateCatDto` class. Here we see a significant advantage of this technique: the `CreateCatDto` class remains the single source of truth for our Post body object (rather than having to create a separate validation class).

```
@@filename(create-cat.dto)
import { IsString, IsInt } from 'class-validator';

export class CreateCatDto {
  @IsString()
  name: string;

  @IsInt()
  age: number;

  @IsString()
  breed: string;
}
```

info Hint Read more about the class-validator decorators [here](#).

Now we can create a `ValidationPipe` class that uses these annotations.

```
@@filename(validation.pipe)
import { PipeTransform, Injectable, ArgumentMetadata, BadRequestException }
  from '@nestjs/common';
import { validate } from 'class-validator';
import { plainToInstance } from 'class-transformer';
```

```

@Injectable()
export class ValidationPipe implements PipeTransform<any> {
  async transform(value: any, { metatype }: ArgumentMetadata) {
    if (!metatype || !this.toValidate(metatype)) {
      return value;
    }
    const object = plainToInstance(metatype, value);
    const errors = await validate(object);
    if (errors.length > 0) {
      throw new BadRequestException('Validation failed');
    }
    return value;
  }

  private toValidate(metatype: Function): boolean {
    const types: Function[] = [String, Boolean, Number, Array, Object];
    return !types.includes(metatype);
  }
}

```

**info Hint** As a reminder, you don't have to build a generic validation pipe on your own since the `ValidationPipe` is provided by Nest out-of-the-box. The built-in `ValidationPipe` offers more options than the sample we built in this chapter, which has been kept basic for the sake of illustrating the mechanics of a custom-built pipe. You can find full details, along with lots of examples [here](#).

**warning Notice** We used the `class-transformer` library above which is made by the same author as the `class-validator` library, and as a result, they play very well together.

Let's go through this code. First, note that the `transform()` method is marked as `async`. This is possible because Nest supports both synchronous and **asynchronous** pipes. We make this method `async` because some of the class-validator validations `can be async` (utilize Promises).

Next note that we are using destructuring to extract the `metatype` field (extracting just this member from an `ArgumentMetadata`) into our `metatype` parameter. This is just shorthand for getting the full `ArgumentMetadata` and then having an additional statement to assign the `metatype` variable.

Next, note the helper function `toValidate()`. It's responsible for bypassing the validation step when the current argument being processed is a native JavaScript type (these can't have validation decorators attached, so there's no reason to run them through the validation step).

Next, we use the `class-transformer` function `plainToInstance()` to transform our plain JavaScript argument object into a typed object so that we can apply validation. The reason we must do this is that the incoming post body object, when serialized from the network request, does **not have any type information** (this is the way the underlying platform, such as Express, works). Class-validator needs to use the validation decorators we defined for our DTO earlier, so we need to perform this transformation to treat the incoming body as an appropriately decorated object, not just a plain vanilla object.

Finally, as noted earlier, since this is a **validation pipe** it either returns the value unchanged, or throws an exception.

The last step is to bind the `ValidationPipe`. Pipes can be parameter-scoped, method-scoped, controller-scoped, or global-scoped. Earlier, with our Joi-based validation pipe, we saw an example of binding the pipe at the method level. In the example below, we'll bind the pipe instance to the route handler `@Body()` decorator so that our pipe is called to validate the post body.

```
@@filename(cats.controller)
@Post()
async create(
  @Body(new ValidationPipe()) createCatDto: CreateCatDto,
) {
  this.catsService.create(createCatDto);
}
```

Parameter-scoped pipes are useful when the validation logic concerns only one specified parameter.

## Global scoped pipes

Since the `ValidationPipe` was created to be as generic as possible, we can realize its full utility by setting it up as a **global-scoped** pipe so that it is applied to every route handler across the entire application.

```
@@filename(main)
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}
bootstrap();
```

**warning** **Notice** In the case of [hybrid apps](#) the `useGlobalPipes()` method doesn't set up pipes for gateways and micro services. For "standard" (non-hybrid) microservice apps, `useGlobalPipes()` does mount pipes globally.

Global pipes are used across the whole application, for every controller and every route handler.

Note that in terms of dependency injection, global pipes registered from outside of any module (with `useGlobalPipes()` as in the example above) cannot inject dependencies since the binding has been done outside the context of any module. In order to solve this issue, you can set up a global pipe **directly from any module** using the following construction:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { APP_PIPE } from '@nestjs/core';

@Module({
  providers: [
    {
```

```

        provide: APP_PIPE,
        useClass: ValidationPipe,
    },
],
})
export class AppModule {}

```

**info Hint** When using this approach to perform dependency injection for the pipe, note that regardless of the module where this construction is employed, the pipe is, in fact, global. Where should this be done? Choose the module where the pipe (`ValidationPipe` in the example above) is defined. Also, `useClass` is not the only way of dealing with custom provider registration. Learn more [here](#).

## The built-in ValidationPipe

As a reminder, you don't have to build a generic validation pipe on your own since the `ValidationPipe` is provided by Nest out-of-the-box. The built-in `ValidationPipe` offers more options than the sample we built in this chapter, which has been kept basic for the sake of illustrating the mechanics of a custom-built pipe. You can find full details, along with lots of examples [here](#).

## Transformation use case

Validation isn't the only use case for custom pipes. At the beginning of this chapter, we mentioned that a pipe can also **transform** the input data to the desired format. This is possible because the value returned from the `transform` function completely overrides the previous value of the argument.

When is this useful? Consider that sometimes the data passed from the client needs to undergo some change - for example converting a string to an integer - before it can be properly handled by the route handler method. Furthermore, some required data fields may be missing, and we would like to apply default values. **Transformation pipes** can perform these functions by interposing a processing function between the client request and the request handler.

Here's a simple `ParseIntPipe` which is responsible for parsing a string into an integer value. (As noted above, Nest has a built-in `ParseIntPipe` that is more sophisticated; we include this as a simple example of a custom transformation pipe).

```

@@filename(parse-int.pipe)
import { PipeTransform, Injectable, ArgumentMetadata, BadRequestException }
from '@nestjs/common';

@Injectable()
export class ParseIntPipe implements PipeTransform<string, number> {
  transform(value: string, metadata: ArgumentMetadata): number {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException('Validation failed');
    }
    return val;
  }
}

```

```

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

@Injectable()
export class ParseIntPipe {
  transform(value, metadata) {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException('Validation failed');
    }
    return val;
  }
}

```

We can then bind this pipe to the selected param as shown below:

```

@@filename()
@Get(':id')
async findOne(@Param('id', new ParseIntPipe()) id) {
  return this.catsService.findOne(id);
}
@@switch
@Get(':id')
@Bind(Param('id', new ParseIntPipe()))
async findOne(id) {
  return this.catsService.findOne(id);
}

```

Another useful transformation case would be to select an **existing user** entity from the database using an id supplied in the request:

```

@@filename()
@Get(':id')
findOne(@Param('id', UserByIdPipe) userEntity: UserEntity) {
  return userEntity;
}
@@switch
@Get(':id')
@Bind(Param('id', UserByIdPipe))
findOne(userEntity) {
  return userEntity;
}

```

We leave the implementation of this pipe to the reader, but note that like all other transformation pipes, it receives an input value (an **id**) and returns an output value (a **UserEntity** object). This can make your code more declarative and **DRY** by abstracting boilerplate code out of your handler and into a common pipe.

## Providing defaults

`Parse*` pipes expect a parameter's value to be defined. They throw an exception upon receiving `null` or `undefined` values. To allow an endpoint to handle missing querystring parameter values, we have to provide a default value to be injected before the `Parse*` pipes operate on these values. The `DefaultValuePipe` serves that purpose. Simply instantiate a `DefaultValuePipe` in the `@Query()` decorator before the relevant `Parse*` pipe, as shown below:

```
@@filename()
@Get()
async findAll(
  @Query('activeOnly', new DefaultValuePipe(false), ParseBoolPipe)
activeOnly: boolean,
  @Query('page', new DefaultValuePipe(0), ParseIntPipe) page: number,
) {
  return this.catsService.findAll({ activeOnly, page });
}
```

## Guards

A guard is a class annotated with the `@Injectable()` decorator, which implements the `CanActivate` interface.



Guards have a **single responsibility**. They determine whether a given request will be handled by the route handler or not, depending on certain conditions (like permissions, roles, ACLs, etc.) present at run-time.

This is often referred to as **authorization**. Authorization (and its cousin, **authentication**, with which it usually collaborates) has typically been handled by **middleware** in traditional Express applications.

Middleware is a fine choice for authentication, since things like token validation and attaching properties to the `request` object are not strongly connected with a particular route context (and its metadata).

But middleware, by its nature, is dumb. It doesn't know which handler will be executed after calling the `next()` function. On the other hand, **Guards** have access to the `ExecutionContext` instance, and thus know exactly what's going to be executed next. They're designed, much like exception filters, pipes, and interceptors, to let you interpose processing logic at exactly the right point in the request/response cycle, and to do so declaratively. This helps keep your code DRY and declarative.

**info Hint** Guards are executed **after** all middleware, but **before** any interceptor or pipe.

### Authorization guard

As mentioned, **authorization** is a great use case for Guards because specific routes should be available only when the caller (usually a specific authenticated user) has sufficient permissions. The `AuthGuard` that we'll build now assumes an authenticated user (and that, therefore, a token is attached to the request headers). It will extract and validate the token, and use the extracted information to determine whether the request can proceed or not.

```
@@filename(auth.guard)
import { Injectable, CanActivate, ExecutionContext } from
'@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
  }
}
@@switch
import { Injectable } from '@nestjs/common';

@Injectable()
export class AuthGuard {
  async canActivate(context) {
```

```
    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
}
}
```

**info Hint** If you are looking for a real-world example on how to implement an authentication mechanism in your application, visit [this chapter](#). Likewise, for more sophisticated authorization example, check [this page](#).

The logic inside the `validateRequest()` function can be as simple or sophisticated as needed. The main point of this example is to show how guards fit into the request/response cycle.

Every guard must implement a `canActivate()` function. This function should return a boolean, indicating whether the current request is allowed or not. It can return the response either synchronously or asynchronously (via a `Promise` or `Observable`). Nest uses the return value to control the next action:

- if it returns `true`, the request will be processed.
- if it returns `false`, Nest will deny the request.

## Execution context

The `canActivate()` function takes a single argument, the `ExecutionContext` instance. The `ExecutionContext` inherits from `ArgumentsHost`. We saw `ArgumentsHost` previously in the exception filters chapter. In the sample above, we are just using the same helper methods defined on `ArgumentsHost` that we used earlier, to get a reference to the `Request` object. You can refer back to the **Arguments host** section of the [exception filters](#) chapter for more on this topic.

By extending `ArgumentsHost`, `ExecutionContext` also adds several new helper methods that provide additional details about the current execution process. These details can be helpful in building more generic guards that can work across a broad set of controllers, methods, and execution contexts. Learn more about `ExecutionContext` [here](#).

## Role-based authentication

Let's build a more functional guard that permits access only to users with a specific role. We'll start with a basic guard template, and build on it in the coming sections. For now, it allows all requests to proceed:

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

@Injectable()
export class RolesGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    return true;
}
```

```
}

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

@Injectable()
export class RolesGuard {
  canActivate(context) {
    return true;
  }
}
```

## Binding guards

Like pipes and exception filters, guards can be **controller-scoped**, method-scoped, or global-scoped. Below, we set up a controller-scoped guard using the `@UseGuards()` decorator. This decorator may take a single argument, or a comma-separated list of arguments. This lets you easily apply the appropriate set of guards with one declaration.

```
@@filename()
@Controller('cats')
@UseGuards(RolesGuard)
export class CatsController {}
```

**info Hint** The `@UseGuards()` decorator is imported from the `@nestjs/common` package.

Above, we passed the `RolesGuard` class (instead of an instance), leaving responsibility for instantiation to the framework and enabling dependency injection. As with pipes and exception filters, we can also pass an in-place instance:

```
@@filename()
@Controller('cats')
@UseGuards(new RolesGuard())
export class CatsController {}
```

The construction above attaches the guard to every handler declared by this controller. If we wish the guard to apply only to a single method, we apply the `@UseGuards()` decorator at the **method level**.

In order to set up a global guard, use the `useGlobalGuards()` method of the Nest application instance:

```
@@filename()
const app = await NestFactory.create(AppModule);
app.useGlobalGuards(new RolesGuard());
```

**warning Notice** In the case of hybrid apps the `useGlobalGuards()` method doesn't set up guards for gateways and micro services by default (see [Hybrid application](#) for information on how to change

this behavior). For "standard" (non-hybrid) microservice apps, `useGlobalGuards()` does mount the guards globally.

Global guards are used across the whole application, for every controller and every route handler. In terms of dependency injection, global guards registered from outside of any module (with `useGlobalGuards()` as in the example above) cannot inject dependencies since this is done outside the context of any module. In order to solve this issue, you can set up a guard directly from any module using the following construction:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { APP_GUARD } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_GUARD,
      useClass: RolesGuard,
    },
  ],
})
export class AppModule {}
```

**info Hint** When using this approach to perform dependency injection for the guard, note that regardless of the module where this construction is employed, the guard is, in fact, global. Where should this be done? Choose the module where the guard (`RolesGuard` in the example above) is defined. Also, `useClass` is not the only way of dealing with custom provider registration. Learn more [here](#).

## Setting roles per handler

Our `RolesGuard` is working, but it's not very smart yet. We're not yet taking advantage of the most important guard feature – the `execution context`. It doesn't yet know about roles, or which roles are allowed for each handler. The `CatsController`, for example, could have different permission schemes for different routes. Some might be available only for an admin user, and others could be open for everyone. How can we match roles to routes in a flexible and reusable way?

This is where **custom metadata** comes into play (learn more [here](#)). Nest provides the ability to attach custom **metadata** to route handlers through either decorators created via `Reflector#createDecorator` static method, or the built-in `@SetMetadata()` decorator.

For example, let's create a `@Roles()` decorator using the `Reflector#createDecorator` method that will attach the metadata to the handler. `Reflector` is provided out of the box by the framework and exposed from the `@nestjs/core` package.

```
@@filename(roles.decorator)
import { Reflector } from '@nestjs/core';
```

```
export const Roles = Reflector.createDecorator<string[]>();
```

The `Roles` decorator here is a function that takes a single argument of type `string[]`.

Now, to use this decorator, we simply annotate the handler with it:

```
@@filename(cats.controller)
@Post()
@Roles(['admin'])
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}

@@switch
@Post()
@Roles(['admin'])
@Bind(Body())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

Here we've attached the `Roles` decorator metadata to the `create()` method, indicating that only users with the `admin` role should be allowed to access this route.

Alternatively, instead of using the `Reflector#createDecorator` method, we could use the built-in `@SetMetadata()` decorator. Learn more about [here](#).

## Putting it all together

Let's now go back and tie this together with our `RolesGuard`. Currently, it simply returns `true` in all cases, allowing every request to proceed. We want to make the return value conditional based on the comparing the **roles assigned to the current user** to the actual roles required by the current route being processed. In order to access the route's role(s) (custom metadata), we'll use the `Reflector` helper class again, as follows:

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

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

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get(Roles, context.getHandler());
    if (!roles) {
```

```
        return true;
    }
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    return matchRoles(roles, user.roles);
}
}

@@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Roles } from './roles.decorator';

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

    canActivate(context) {
        const roles = this.reflector.get(Roles, context.getHandler());
        if (!roles) {
            return true;
        }
        const request = context.switchToHttp().getRequest();
        const user = request.user;
        return matchRoles(roles, user.roles);
    }
}
```

**info Hint** In the node.js world, it's common practice to attach the authorized user to the `request` object. Thus, in our sample code above, we are assuming that `request.user` contains the user instance and allowed roles. In your app, you will probably make that association in your custom **authentication guard** (or middleware). Check [this chapter](#) for more information on this topic.

**warning Warning** The logic inside the `matchRoles()` function can be as simple or sophisticated as needed. The main point of this example is to show how guards fit into the request/response cycle.

Refer to the [Reflection and metadata](#) section of the **Execution context** chapter for more details on utilizing `Reflector` in a context-sensitive way.

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

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

Note that behind the scenes, when a guard returns `false`, the framework throws a `ForbiddenException`. If you want to return a different error response, you should throw your own specific exception. For example:

```
throw new UnauthorizedException();
```

Any exception thrown by a guard will be handled by the [exceptions layer](#) (global exceptions filter and any exceptions filters that are applied to the current context).

**info Hint** If you are looking for a real-world example on how to implement authorization, check [this chapter](#).

## Interceptors

An interceptor is a class annotated with the `@Injectable()` decorator and implements the `NestInterceptor` interface.



Interceptors have a set of useful capabilities which are inspired by the [Aspect Oriented Programming](#) (AOP) technique. They make it possible to:

- bind extra logic before / after method execution
- transform the result returned from a function
- transform the exception thrown from a function
- extend the basic function behavior
- completely override a function depending on specific conditions (e.g., for caching purposes)

## Basics

Each interceptor implements the `intercept()` method, which takes two arguments. The first one is the `ExecutionContext` instance (exactly the same object as for [guards](#)). The `ExecutionContext` inherits from `ArgumentsHost`. We saw `ArgumentsHost` before in the exception filters chapter. There, we saw that it's a wrapper around arguments that have been passed to the original handler, and contains different arguments arrays based on the type of the application. You can refer back to the [exception filters](#) for more on this topic.

## Execution context

By extending `ArgumentsHost`, `ExecutionContext` also adds several new helper methods that provide additional details about the current execution process. These details can be helpful in building more generic interceptors that can work across a broad set of controllers, methods, and execution contexts. Learn more about `ExecutionContext` [here](#).

## Call handler

The second argument is a `CallHandler`. The `CallHandler` interface implements the `handle()` method, which you can use to invoke the route handler method at some point in your interceptor. If you don't call the `handle()` method in your implementation of the `intercept()` method, the route handler method won't be executed at all.

This approach means that the `intercept()` method effectively **wraps** the request/response stream. As a result, you may implement custom logic **both before and after** the execution of the final route handler. It's clear that you can write code in your `intercept()` method that executes **before** calling `handle()`, but how do you affect what happens afterward? Because the `handle()` method returns an `Observable`, we can use powerful [RxJS](#) operators to further manipulate the response. Using Aspect Oriented Programming terminology, the invocation of the route handler (i.e., calling `handle()`) is called a `Pointcut`, indicating that it's the point at which our additional logic is inserted.

Consider, for example, an incoming `POST /cats` request. This request is destined for the `create()` handler defined inside the `CatsController`. If an interceptor which does not call the `handle()` method is

called anywhere along the way, the `create()` method won't be executed. Once `handle()` is called (and its `Observable` has been returned), the `create()` handler will be triggered. And once the response stream is received via the `Observable`, additional operations can be performed on the stream, and a final result returned to the caller.

## Aspect interception

The first use case we'll look at is to use an interceptor to log user interaction (e.g., storing user calls, asynchronously dispatching events or calculating a timestamp). We show a simple `LoggingInterceptor` below:

```
@@filename(logging.interceptor)
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
  '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any>
  {
    console.log('Before...');

    const now = Date.now();
    return next
      .handle()
      .pipe(
        tap(() => console.log(`After... ${Date.now() - now}ms`)),
      );
  }
}

@@switch
import { Injectable } from '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor {
  intercept(context, next) {
    console.log('Before...');

    const now = Date.now();
    return next
      .handle()
      .pipe(
        tap(() => console.log(`After... ${Date.now() - now}ms`)),
      );
  }
}
```

**info Hint** The `NestInterceptor<T, R>` is a generic interface in which `T` indicates the type of an `Observable<T>` (supporting the response stream), and `R` is the type of the value wrapped by `Observable<R>`.

**warning Notice** Interceptors, like controllers, providers, guards, and so on, can **inject dependencies** through their `constructor`.

Since `handle()` returns an RxJS `Observable`, we have a wide choice of operators we can use to manipulate the stream. In the example above, we used the `tap()` operator, which invokes our anonymous logging function upon graceful or exceptional termination of the observable stream, but doesn't otherwise interfere with the response cycle.

## Binding interceptors

In order to set up the interceptor, we use the `@UseInterceptors()` decorator imported from the `@nestjs/common` package. Like `pipes` and `guards`, interceptors can be controller-scoped, method-scoped, or global-scoped.

```
@@filename(cats.controller)
@UseInterceptors(LoggingInterceptor)
export class CatsController {}
```

**info Hint** The `@UseInterceptors()` decorator is imported from the `@nestjs/common` package.

Using the above construction, each route handler defined in `CatsController` will use `LoggingInterceptor`. When someone calls the `GET /cats` endpoint, you'll see the following output in your standard output:

```
Before...
After... 1ms
```

Note that we passed the `LoggingInterceptor` type (instead of an instance), leaving responsibility for instantiation to the framework and enabling dependency injection. As with pipes, guards, and exception filters, we can also pass an in-place instance:

```
@@filename(cats.controller)
@UseInterceptors(new LoggingInterceptor())
export class CatsController {}
```

As mentioned, the construction above attaches the interceptor to every handler declared by this controller. If we want to restrict the interceptor's scope to a single method, we simply apply the decorator at the **method level**.

In order to set up a global interceptor, we use the `useGlobalInterceptors()` method of the Nest application instance:

```
const app = await NestFactory.create(AppModule);
app.useGlobalInterceptors(new LoggingInterceptor());
```

Global interceptors are used across the whole application, for every controller and every route handler. In terms of dependency injection, global interceptors registered from outside of any module (with `useGlobalInterceptors()`, as in the example above) cannot inject dependencies since this is done outside the context of any module. In order to solve this issue, you can set up an interceptor **directly from any module** using the following construction:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { APP_INTERCEPTOR } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_INTERCEPTOR,
      useClass: LoggingInterceptor,
    },
  ],
})
export class AppModule {}
```

**info Hint** When using this approach to perform dependency injection for the interceptor, note that regardless of the module where this construction is employed, the interceptor is, in fact, global. Where should this be done? Choose the module where the interceptor (`LoggingInterceptor` in the example above) is defined. Also, `useClass` is not the only way of dealing with custom provider registration. Learn more [here](#).

## Response mapping

We already know that `handle()` returns an `Observable`. The stream contains the value **returned** from the route handler, and thus we can easily mutate it using RxJS's `map()` operator.

**warning Warning** The response mapping feature doesn't work with the library-specific response strategy (using the `@Res()` object directly is forbidden).

Let's create the `TransformInterceptor`, which will modify each response in a trivial way to demonstrate the process. It will use RxJS's `map()` operator to assign the response object to the `data` property of a newly created object, returning the new object to the client.

```
@@filename(transform.interceptor)
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
  '@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
```

```

export interface Response<T> {
  data: T;
}

@Injectable()
export class TransformInterceptor<T> implements NestInterceptor<T, Response<T>> {
  intercept(context: ExecutionContext, next: CallHandler): Observable<Response<T>> {
    return next.handle().pipe(map(data => ({ data })));
  }
}

@switch
import { Injectable } from '@nestjs/common';
import { map } from 'rxjs/operators';

@Injectable()
export class TransformInterceptor {
  intercept(context, next) {
    return next.handle().pipe(map(data => ({ data })));
  }
}

```

**info Hint** Nest interceptors work with both synchronous and asynchronous `intercept()` methods. You can simply switch the method to `async` if necessary.

With the above construction, when someone calls the `GET /cats` endpoint, the response would look like the following (assuming that route handler returns an empty array `[]`):

```
{
  "data": []
}
```

Interceptors have great value in creating re-usable solutions to requirements that occur across an entire application. For example, imagine we need to transform each occurrence of a `null` value to an empty string `''`. We can do it using one line of code and bind the interceptor globally so that it will automatically be used by each registered handler.

```

@filename()
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
  '@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable()
export class ExcludeNullInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    ...
  }
}

```

```
    return next
      .handle()
      .pipe(map(value => value === null ? '' : value));
  }
}

@@switch
import { Injectable } from '@nestjs/common';
import { map } from 'rxjs/operators';

@Injectable()
export class ExcludeNullInterceptor {
  intercept(context, next) {
    return next
      .handle()
      .pipe(map(value => value === null ? '' : value));
  }
}
```

## Exception mapping

Another interesting use-case is to take advantage of RxJS's `catchError()` operator to override thrown exceptions:

```
@@filename(errors.interceptor)
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  BadGatewayException,
  CallHandler,
} from '@nestjs/common';
import { Observable, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

@Injectable()
export class ErrorsInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next
      .handle()
      .pipe(
        catchError(err => throwError(() => new BadGatewayException())),
      );
  }
}

@@switch
import { Injectable, BadGatewayException } from '@nestjs/common';
import { throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

@Injectable()
```

```
export class ErrorsInterceptor {
  intercept(context, next) {
    return next
      .handle()
      .pipe(
        catchError(err => throwError(() => new BadGatewayException())),
      );
  }
}
```

## Stream overriding

There are several reasons why we may sometimes want to completely prevent calling the handler and return a different value instead. An obvious example is to implement a cache to improve response time. Let's take a look at a simple **cache interceptor** that returns its response from a cache. In a realistic example, we'd want to consider other factors like TTL, cache invalidation, cache size, etc., but that's beyond the scope of this discussion. Here we'll provide a basic example that demonstrates the main concept.

```
@@filename(cache.interceptor)
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
  '@nestjs/common';
import { Observable, of } from 'rxjs';

@Injectable()
export class CacheInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    const isCached = true;
    if (isCached) {
      return of([]);
    }
    return next.handle();
  }
}
@@switch
import { Injectable } from '@nestjs/common';
import { of } from 'rxjs';

@Injectable()
export class CacheInterceptor {
  intercept(context, next) {
    const isCached = true;
    if (isCached) {
      return of([]);
    }
    return next.handle();
  }
}
```

Our `CacheInterceptor` has a hardcoded `isCached` variable and a hardcoded response `[]` as well. The key point to note is that we return a new stream here, created by the RxJS `of()` operator, therefore the route handler **won't be called** at all. When someone calls an endpoint that makes use of `CacheInterceptor`, the response (a hardcoded, empty array) will be returned immediately. In order to create a generic solution, you can take advantage of `Reflector` and create a custom decorator. The `Reflector` is well described in the [guards](#) chapter.

## More operators

The possibility of manipulating the stream using RxJS operators gives us many capabilities. Let's consider another common use case. Imagine you would like to handle `timeouts` on route requests. When your endpoint doesn't return anything after a period of time, you want to terminate with an error response. The following construction enables this:

```
@@filename(timeout.interceptor)
import { Injectable, NestInterceptor, ExecutionContext, CallHandler,
RequestTimeoutException } from '@nestjs/common';
import { Observable, throwError, TimeoutError } from 'rxjs';
import { catchError, timeout } from 'rxjs/operators';

@Injectable()
export class TimeoutInterceptor implements NestInterceptor {
    intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
        return next.handle().pipe(
            timeout(5000),
            catchError(err => {
                if (err instanceof TimeoutError) {
                    return throwError(() => new RequestTimeoutException());
                }
                return throwError(() => err);
            }),
        );
    };
}
@@switch
import { Injectable, RequestTimeoutException } from '@nestjs/common';
import { Observable, throwError, TimeoutError } from 'rxjs';
import { catchError, timeout } from 'rxjs/operators';

@Injectable()
export class TimeoutInterceptor {
    intercept(context, next) {
        return next.handle().pipe(
            timeout(5000),
            catchError(err => {
                if (err instanceof TimeoutError) {
                    return throwError(() => new RequestTimeoutException());
                }
                return throwError(() => err);
            }),
        );
    };
}
```

```
)  
};  
};
```

After 5 seconds, request processing will be canceled. You can also add custom logic before throwing [RequestTimeoutException](#) (e.g. release resources).

## Custom route decorators

Nest is built around a language feature called **decorators**. Decorators are a well-known concept in a lot of commonly used programming languages, but in the JavaScript world, they're still relatively new. In order to better understand how decorators work, we recommend reading [this article](#). Here's a simple definition:

An ES2016 decorator is an expression which returns a function and can take a target, name and property descriptor as arguments. You apply it by prefixing the decorator with an @ character and placing this at the very top of what you are trying to decorate. Decorators can be defined for either a class, a method or a property.

### Param decorators

Nest provides a set of useful **param decorators** that you can use together with the HTTP route handlers. Below is a list of the provided decorators and the plain Express (or Fastify) objects they represent

@Request(), @Req()	req
@Response(), @Res()	res
@Next()	next
@Session()	req.session
@Param(param?: string)	req.params / req.query [param]
@Body(param?: string)	req.body / req.rawBody [param]
@Query(param?: string)	req.query / req.rawQuery [param]
@Headers(param?: string)	req.headers / req.rawHeaders [param]
@Ip()	req.ip
@HostParam()	req.hosts

Additionally, you can create your own **custom decorators**. Why is this useful?

In the node.js world, it's common practice to attach properties to the **request** object. Then you manually extract them in each route handler, using code like the following:

```
const user = req.user;
```

In order to make your code more readable and transparent, you can create a **@User()** decorator and reuse it across all of your controllers.

```
@@filename(user.decorator)
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const User = createParamDecorator(
```

```
(data: unknown, ctx: ExecutionContext) => {
  const request = ctx.switchToHttp().getRequest();
  return request.user;
},
);
```

Then, you can simply use it wherever it fits your requirements.

```
@@filename()
@Get()
async findOne(@User() user: UserEntity) {
  console.log(user);
}

@@switch
@Get()
@Bind(User())
async findOne(user) {
  console.log(user);
}
```

## Passing data

When the behavior of your decorator depends on some conditions, you can use the `data` parameter to pass an argument to the decorator's factory function. One use case for this is a custom decorator that extracts properties from the request object by key. Let's assume, for example, that our `authentication` layer validates requests and attaches a user entity to the request object. The user entity for an authenticated request might look like:

```
{
  "id": 101,
  "firstName": "Alan",
  "lastName": "Turing",
  "email": "alan@email.com",
  "roles": ["admin"]
}
```

Let's define a decorator that takes a property name as key, and returns the associated value if it exists (or `undefined` if it doesn't exist, or if the `user` object has not been created).

```
@@filename(user.decorator)
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const User = createParamDecorator(
  (data: string, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    const user = request.user;
```

```

        return data ? user?.[data] : user;
    },
);
@@switch
import { createParamDecorator } from '@nestjs/common';

export const User = createParamDecorator((data, ctx) => {
    const request = ctx.switchToHttp().getRequest();
    const user = request.user;

    return data ? user && user[data] : user;
});

```

Here's how you could then access a particular property via the `@User()` decorator in the controller:

```

@@filename()
@Get()
async findOne(@User('firstName') firstName: string) {
    console.log(`Hello ${firstName}`);
}

@@switch
@Get()
@Bind(User('firstName'))
async findOne(firstName) {
    console.log(`Hello ${firstName}`);
}

```

You can use this same decorator with different keys to access different properties. If the `user` object is deep or complex, this can make for easier and more readable request handler implementations.

**info Hint** For TypeScript users, note that `createParamDecorator<T>()` is a generic. This means you can explicitly enforce type safety, for example `createParamDecorator<string>((data, ctx) => ...)`. Alternatively, specify a parameter type in the factory function, for example `createParamDecorator((data: string, ctx) => ...)`. If you omit both, the type for `data` will be `any`.

## Working with pipes

Nest treats custom param decorators in the same fashion as the built-in ones (`@Body()`, `@Param()` and `@Query()`). This means that pipes are executed for the custom annotated parameters as well (in our examples, the `user` argument). Moreover, you can apply the pipe directly to the custom decorator:

```

@@filename()
@Get()
async findOne(
    @User(new ValidationPipe({ validateCustomDecorators: true }))
    user: UserEntity,
)

```

```
) {
    console.log(user);
}
@@switch
@Get()
@Bind(User(new ValidationPipe({ validateCustomDecorators: true })))
async findOne(user) {
    console.log(user);
}
```

**info Hint** Note that `validateCustomDecorators` option must be set to true. `ValidationPipe` does not validate arguments annotated with the custom decorators by default.

## Decorator composition

Nest provides a helper method to compose multiple decorators. For example, suppose you want to combine all decorators related to authentication into a single decorator. This could be done with the following construction:

```
@@filename(auth.decorator)
import { applyDecorators } from '@nestjs/common';

export function Auth(...roles: Role[]) {
    return applyDecorators(
        SetMetadata('roles', roles),
        UseGuards(AuthGuard, RolesGuard),
        ApiBearerAuth(),
        ApiUnauthorizedResponse({ description: 'Unauthorized' }),
    );
}

@@switch
import { applyDecorators } from '@nestjs/common';

export function Auth(...roles) {
    return applyDecorators(
        SetMetadata('roles', roles),
        UseGuards(AuthGuard, RolesGuard),
        ApiBearerAuth(),
        ApiUnauthorizedResponse({ description: 'Unauthorized' }),
    );
}
```

You can then use this custom `@Auth()` decorator as follows:

```
@Get('users')
@Auth('admin')
findAllUsers() {}
```

This has the effect of applying all four decorators with a single declaration.

**warning** **Warning** The `@ApiHideProperty()` decorator from the `@nestjs/swagger` package is not composable and won't work properly with the `applyDecorators` function.

## Injection scopes

For people coming from different programming language backgrounds, it might be unexpected to learn that in Nest, almost everything is shared across incoming requests. We have a connection pool to the database, singleton services with global state, etc. Remember that Node.js doesn't follow the request/response Multi-Threaded Stateless Model in which every request is processed by a separate thread. Hence, using singleton instances is fully **safe** for our applications.

However, there are edge-cases when request-based lifetime may be the desired behavior, for instance per-request caching in GraphQL applications, request tracking, and multi-tenancy. Injection scopes provide a mechanism to obtain the desired provider lifetime behavior.

### Provider scope

A provider can have any of the following scopes:

DEFAULT	A single instance of the provider is shared across the entire application. The instance lifetime is tied directly to the application lifecycle. Once the application has bootstrapped, all singleton providers have been instantiated. Singleton scope is used by default.
REQUEST	A new instance of the provider is created exclusively for each incoming <b>request</b> . The instance is garbage-collected after the request has completed processing.
TRANSIENT	Transient providers are not shared across consumers. Each consumer that injects a transient provider will receive a new, dedicated instance.

**info Hint** Using singleton scope is **recommended** for most use cases. Sharing providers across consumers and across requests means that an instance can be cached and its initialization occurs only once, during application startup.

### Usage

Specify injection scope by passing the **scope** property to the `@Injectable()` decorator options object:

```
import { Injectable, Scope } from '@nestjs/common';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {}
```

Similarly, for [custom providers](#), set the **scope** property in the long-hand form for a provider registration:

```
{
  provide: 'CACHE_MANAGER',
  useClass: CacheManager,
  scope: Scope.TRANSIENT,
}
```

**info Hint** Import the `Scope` enum from `@nestjs/common`

Singleton scope is used by default, and need not be declared. If you do want to declare a provider as singleton scoped, use the `Scope.DEFAULT` value for the `scope` property.

**warning Notice** Websocket Gateways should not use request-scoped providers because they must act as singletons. Each gateway encapsulates a real socket and cannot be instantiated multiple times. The limitation also applies to some other providers, like [Passport strategies](#) or [Cron controllers](#).

## Controller scope

Controllers can also have scope, which applies to all request method handlers declared in that controller. Like provider scope, the scope of a controller declares its lifetime. For a request-scoped controller, a new instance is created for each inbound request, and garbage-collected when the request has completed processing.

Declare controller scope with the `scope` property of the `ControllerOptions` object:

```
@Controller({
  path: 'cats',
  scope: Scope.REQUEST,
})
export class CatsController {}
```

## Scope hierarchy

The `REQUEST` scope bubbles up the injection chain. A controller that depends on a request-scoped provider will, itself, be request-scoped.

Imagine the following dependency graph: `CatsController <- CatsService <- CatsRepository`. If `CatsService` is request-scoped (and the others are default singletons), the `CatsController` will become request-scoped as it is dependent on the injected service. The `CatsRepository`, which is not dependent, would remain singleton-scoped.

Transient-scoped dependencies don't follow that pattern. If a singleton-scoped `DogsService` injects a transient `LoggerService` provider, it will receive a fresh instance of it. However, `DogsService` will stay singleton-scoped, so injecting it anywhere would *not* resolve to a new instance of `DogsService`. In case it's desired behavior, `DogsService` must be explicitly marked as `TRANSIENT` as well.

## Request provider

In an HTTP server-based application (e.g., using `@nestjs/platform-express` or `@nestjs/platform-fastify`), you may want to access a reference to the original request object when using request-scoped providers. You can do this by injecting the `REQUEST` object.

```
import { Injectable, Scope, Inject } from '@nestjs/common';
import { REQUEST } from '@nestjs/core';
import { Request } from 'express';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {
  constructor(@Inject(REQUEST) private request: Request) {}
}
```

Because of underlying platform/protocol differences, you access the inbound request slightly differently for Microservice or GraphQL applications. In [GraphQL](#) applications, you inject `CONTEXT` instead of `REQUEST`:

```
import { Injectable, Scope, Inject } from '@nestjs/common';
import { CONTEXT } from '@nestjs/graphql';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {
  constructor(@Inject(CONTEXT) private context) {}
}
```

You then configure your `context` value (in the [GraphQLModule](#)) to contain `request` as its property.

## Inquirer provider

If you want to get the class where a provider was constructed, for instance in logging or metrics providers, you can inject the `INQUIRER` token.

```
import { Inject, Injectable, Scope } from '@nestjs/common';
import { INQUIRER } from '@nestjs/core';

@Injectable({ scope: Scope.TRANSIENT })
export class HelloService {
  constructor(@Inject(INQUIRER) private parentClass: object) {}

  sayHello(message: string) {
    console.log(`.${this.parentClass?.constructor?.name}: ${message}`);
  }
}
```

And then use it as follows:

```
import { Injectable } from '@nestjs/common';
import { HelloService } from './hello.service';

@Injectable()
export class AppService {
```

```
constructor(private helloService: HelloService) {}

getRoot(): string {
  this.helloService.sayHello('My name is getRoot');

  return 'Hello world!';
}

}
```

In the example above when `AppService#getRoot` is called, "AppService: My name is getRoot" will be logged to the console.

## Performance

Using request-scoped providers will have an impact on application performance. While Nest tries to cache as much metadata as possible, it will still have to create an instance of your class on each request. Hence, it will slow down your average response time and overall benchmarking result. Unless a provider must be request-scoped, it is strongly recommended that you use the default singleton scope.

**info Hint** Although it all sounds quite intimidating, a properly designed application that leverages request-scoped providers should not slow down by more than ~5% latency-wise.

## Durable providers

Request-scoped providers, as mentioned in the section above, may lead to increased latency since having at least 1 request-scoped provider (injected into the controller instance, or deeper - injected into one of its providers) makes the controller request-scoped as well. That means, it must be recreated (instantiated) per each individual request (and garbage collected afterwards). Now, that also means, that for let's say 30k requests in parallel, there will be 30k ephemeral instances of the controller (and its request-scoped providers).

Having a common provider that most providers depend on (think of a database connection, or a logger service), automatically converts all those providers to request-scoped providers as well. This can pose a challenge in **multi-tenant applications**, especially for those that have a central request-scoped "data source" provider that grabs headers/token from the request object and based on its values, retrieves the corresponding database connection/schema (specific to that tenant).

For instance, let's say you have an application alternately used by 10 different customers. Each customer has its **own dedicated data source**, and you want to make sure customer A will never be able to reach customer B's database. One way to achieve this could be to declare a request-scoped "data source" provider that - based on the request object - determines what's the "current customer" and retrieves its corresponding database. With this approach, you can turn your application into a multi-tenant application in just a few minutes. But, a major downside to this approach is that since most likely a large chunk of your application's components rely on the "data source" provider, they will implicitly become "request-scoped", and therefore you will undoubtedly see an impact in your app's performance.

But what if we had a better solution? Since we only have 10 customers, couldn't we have 10 individual **DI sub-trees** per customer (instead of recreating each tree per request)? If your providers don't rely on any property that's truly unique for each consecutive request (e.g., request UUID) but instead there're some

specific attributes that let us aggregate (classify) them, there's no reason to recreate DI sub-tree on every incoming request.

And that's exactly when the **durable providers** come in handy.

Before we start flagging providers as durable, we must first register a **strategy** that instructs Nest what are those "common request attributes", provide logic that groups requests - associates them with their corresponding DI sub-trees.

```
import {
  HostComponentInfo,
  ContextId,
  ContextIdFactory,
  ContextIdStrategy,
} from '@nestjs/core';
import { Request } from 'express';

const tenants = new Map<string, ContextId>();

export class AggregateByTenantContextIdStrategy implements
ContextIdStrategy {
  attach(contextId: ContextId, request: Request) {
    const tenantId = request.headers['x-tenant-id'] as string;
    let tenantSubTreeId: ContextId;

    if (tenants.has(tenantId)) {
      tenantSubTreeId = tenants.get(tenantId);
    } else {
      tenantSubTreeId = ContextIdFactory.create();
      tenants.set(tenantId, tenantSubTreeId);
    }

    // If tree is not durable, return the original "contextId" object
    return (info: HostComponentInfo) =>
      info.isTreeDurable ? tenantSubTreeId : contextId;
  }
}
```

**info Hint** Similar to the request scope, durability bubbles up the injection chain. That means if A depends on B which is flagged as **durable**, A implicitly becomes durable too (unless **durable** is explicitly set to **false** for A provider).

**warning Warning** Note this strategy is not ideal for applications operating with a large number of tenants.

The value returned from the **attach** method instructs Nest what context identifier should be used for a given host. In this case, we specified that the **tenantSubTreeId** should be used instead of the original, auto-generated **contextId** object, when the host component (e.g., request-scoped controller) is flagged as durable (you can learn how to mark providers as durable below). Also, in the above example, **no payload**

would be registered (where payload = **REQUEST/CONTEXT** provider that represents the "root" - parent of the sub-tree).

If you want to register the payload for a durable tree, use the following construction instead:

```
// The return of `AggregateByTenantContextIdStrategy#attach` method:
return {
  resolve: (info: HostComponentInfo) =>
    info.isTreeDurable ? tenantSubTreeId : contextId,
  payload: { tenantId },
}
```

Now whenever you inject the **REQUEST** provider (or **CONTEXT** for GraphQL applications) using the **@Inject(REQUEST)/@Inject(CONTEXT)**, the **payload** object would be injected (consisting of a single property - **tenantId** in this case).

Alright so with this strategy in place, you can register it somewhere in your code (as it applies globally anyway), so for example, you could place it in the **main.ts** file:

```
ContextIdFactory.apply(new AggregateByTenantContextIdStrategy());
```

**info Hint** The **ContextIdFactory** class is imported from the **@nestjs/core** package.

As long as the registration occurs before any request hits your application, everything will work as intended.

Lastly, to turn a regular provider into a durable provider, simply set the **durable** flag to **true** and change its scope to **Scope.REQUEST** (not needed if the REQUEST scope is in the injection chain already):

```
import { Injectable, Scope } from '@nestjs/common';

@Injectable({ scope: Scope.REQUEST, durable: true })
export class CatsService {}
```

Similarly, for **custom providers**, set the **durable** property in the long-hand form for a provider registration:

```
{
  provide: 'foobar',
  useFactory: () => { ... },
  scope: Scope.REQUEST,
  durable: true,
}
```

## Asynchronous providers

At times, the application start should be delayed until one or more **asynchronous tasks** are completed. For example, you may not want to start accepting requests until the connection with the database has been established. You can achieve this using asynchronous providers.

The syntax for this is to use `async/await` with the `useFactory` syntax. The factory returns a `Promise`, and the factory function can `await` asynchronous tasks. Nest will await resolution of the promise before instantiating any class that depends on (injects) such a provider.

```
{  
  provide: 'ASYNC_CONNECTION',  
  useFactory: async () => {  
    const connection = await createConnection(options);  
    return connection;  
  },  
}
```

**info Hint** Learn more about custom provider syntax [here](#).

### Injection

Asynchronous providers are injected to other components by their tokens, like any other provider. In the example above, you would use the construct `@Inject('ASYNC_CONNECTION')`.

### Example

The [TypeORM recipe](#) has a more substantial example of an asynchronous provider.

## Dynamic modules

The [Modules chapter](#) covers the basics of Nest modules, and includes a brief introduction to [dynamic modules](#). This chapter expands on the subject of dynamic modules. Upon completion, you should have a good grasp of what they are and how and when to use them.

### Introduction

Most application code examples in the [Overview](#) section of the documentation make use of regular, or static, modules. Modules define groups of components like [providers](#) and [controllers](#) that fit together as a modular part of an overall application. They provide an execution context, or scope, for these components. For example, providers defined in a module are visible to other members of the module without the need to export them. When a provider needs to be visible outside of a module, it is first exported from its host module, and then imported into its consuming module.

Let's walk through a familiar example.

First, we'll define a [UsersModule](#) to provide and export a [UsersService](#). [UsersModule](#) is the **host** module for [UsersService](#).

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

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

Next, we'll define an [AuthModule](#), which imports [UsersModule](#), making [UsersModule](#)'s exported providers available inside [AuthModule](#):

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

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

These constructs allow us to inject [UsersService](#) in, for example, the [AuthService](#) that is hosted in [AuthModule](#):

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

@Injectable()
export class AuthService {
  constructor(private usersService: UsersService) {}
  /*
    Implementation that makes use of this.usersService
  */
}
```

We'll refer to this as **static** module binding. All the information Nest needs to wire together the modules has already been declared in the host and consuming modules. Let's unpack what's happening during this process. Nest makes `UsersService` available inside `AuthModule` by:

1. Instantiating `UsersModule`, including transitively importing other modules that `UsersModule` itself consumes, and transitively resolving any dependencies (see [Custom providers](#)).
2. Instantiating `AuthModule`, and making `UsersModule`'s exported providers available to components in `AuthModule` (just as if they had been declared in `AuthModule`).
3. Injecting an instance of `UsersService` in `AuthService`.

## Dynamic module use case

With static module binding, there's no opportunity for the consuming module to **influence** how providers from the host module are configured. Why does this matter? Consider the case where we have a general purpose module that needs to behave differently in different use cases. This is analogous to the concept of a "plugin" in many systems, where a generic facility requires some configuration before it can be used by a consumer.

A good example with Nest is a **configuration module**. Many applications find it useful to externalize configuration details by using a configuration module. This makes it easy to dynamically change the application settings in different deployments: e.g., a development database for developers, a staging database for the staging/testing environment, etc. By delegating the management of configuration parameters to a configuration module, the application source code remains independent of configuration parameters.

The challenge is that the configuration module itself, since it's generic (similar to a "plugin"), needs to be customized by its consuming module. This is where *dynamic modules* come into play. Using dynamic module features, we can make our configuration module **dynamic** so that the consuming module can use an API to control how the configuration module is customized at the time it is imported.

In other words, dynamic modules provide an API for importing one module into another, and customizing the properties and behavior of that module when it is imported, as opposed to using the static bindings we've seen so far.

## Config module example

We'll be using the basic version of the example code from the [configuration chapter](#) for this section. The completed version as of the end of this chapter is available as a working [example here](#).

Our requirement is to make `ConfigModule` accept an `options` object to customize it. Here's the feature we want to support. The basic sample hard-codes the location of the `.env` file to be in the project root folder. Let's suppose we want to make that configurable, such that you can manage your `.env` files in any folder of your choosing. For example, imagine you want to store your various `.env` files in a folder under the project root called `config` (i.e., a sibling folder to `src`). You'd like to be able to choose different folders when using the `ConfigModule` in different projects.

Dynamic modules give us the ability to pass parameters into the module being imported so we can change its behavior. Let's see how this works. It's helpful if we start from the end-goal of how this might look from the consuming module's perspective, and then work backwards. First, let's quickly review the example of *statically* importing the `ConfigModule` (i.e., an approach which has no ability to influence the behavior of the imported module). Pay close attention to the `imports` array in the `@Module()` decorator:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from './config/config.module';

@Module({
  imports: [ConfigModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Let's consider what a *dynamic module* import, where we're passing in a configuration object, might look like. Compare the difference in the `imports` array between these two examples:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from './config/config.module';

@Module({
  imports: [ConfigModule.register({ folder: './config' })],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Let's see what's happening in the dynamic example above. What are the moving parts?

1. `ConfigModule` is a normal class, so we can infer that it must have a **static method** called `register()`. We know it's static because we're calling it on the `ConfigModule` class, not on an

**instance** of the class. Note: this method, which we will create soon, can have any arbitrary name, but by convention we should call it either `forRoot()` or `register()`.

2. The `register()` method is defined by us, so we can accept any input arguments we like. In this case, we're going to accept a simple `options` object with suitable properties, which is the typical case.
3. We can infer that the `register()` method must return something like a `module` since its return value appears in the familiar `imports` list, which we've seen so far includes a list of modules.

In fact, what our `register()` method will return is a `DynamicModule`. A dynamic module is nothing more than a module created at run-time, with the same exact properties as a static module, plus one additional property called `module`. Let's quickly review a sample static module declaration, paying close attention to the module options passed in to the decorator:

```
@Module({  
    imports: [DogsModule],  
    controllers: [CatsController],  
    providers: [CatsService],  
    exports: [CatsService]  
})
```

Dynamic modules must return an object with the exact same interface, plus one additional property called `module`. The `module` property serves as the name of the module, and should be the same as the class name of the module, as shown in the example below.

**info Hint** For a dynamic module, all properties of the module options object are optional **except** `module`.

What about the static `register()` method? We can now see that its job is to return an object that has the `DynamicModule` interface. When we call it, we are effectively providing a module to the `imports` list, similar to the way we would do so in the static case by listing a module class name. In other words, the dynamic module API simply returns a module, but rather than fix the properties in the `@Module` decorator, we specify them programmatically.

There are still a couple of details to cover to help make the picture complete:

1. We can now state that the `@Module()` decorator's `imports` property can take not only a module class name (e.g., `imports: [UsersModule]`), but also a function **returning** a dynamic module (e.g., `imports: [ConfigModule.register(...)]`).
2. A dynamic module can itself import other modules. We won't do so in this example, but if the dynamic module depends on providers from other modules, you would import them using the optional `imports` property. Again, this is exactly analogous to the way you'd declare metadata for a static module using the `@Module()` decorator.

Armed with this understanding, we can now look at what our dynamic `ConfigModule` declaration must look like. Let's take a crack at it.

```

import { DynamicModule, Module } from '@nestjs/common';
import { ConfigService } from './config.service';

@Module({})
export class ConfigModule {
  static register(): DynamicModule {
    return {
      module: ConfigModule,
      providers: [ConfigService],
      exports: [ConfigService],
    };
  }
}

```

It should now be clear how the pieces tie together. Calling `ConfigModule.register(...)` returns a `DynamicModule` object with properties which are essentially the same as those that, until now, we've provided as metadata via the `@Module()` decorator.

**info Hint** Import `DynamicModule` from `@nestjs/common`.

Our dynamic module isn't very interesting yet, however, as we haven't introduced any capability to `configure` it as we said we would like to do. Let's address that next.

## Module configuration

The obvious solution for customizing the behavior of the `ConfigModule` is to pass it an `options` object in the static `register()` method, as we guessed above. Let's look once again at our consuming module's `imports` property:

```

import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from './config/config.module';

@Module({
  imports: [ConfigModule.register({ folder: './config' })],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}

```

That nicely handles passing an `options` object to our dynamic module. How do we then use that `options` object in the `ConfigModule`? Let's consider that for a minute. We know that our `ConfigModule` is basically a host for providing and exporting an injectable service - the `ConfigService` - for use by other providers. It's actually our `ConfigService` that needs to read the `options` object to customize its behavior. Let's assume for the moment that we know how to somehow get the `options` from the `register()` method into the `ConfigService`. With that assumption, we can make a few changes to the service to customize its behavior based on the properties from the `options` object. (**Note:** for the time

being, since we *haven't* actually determined how to pass it in, we'll just hard-code `options`. We'll fix this in a minute).

```
import { Injectable } from '@nestjs/common';
import * as dotenv from 'dotenv';
import * as fs from 'fs';
import * as path from 'path';
import { EnvConfig } from './interfaces';

@Injectable()
export class ConfigService {
  private readonly envConfig: EnvConfig;

  constructor() {
    const options = { folder: './config' };

    const filePath = `${process.env.NODE_ENV || 'development'}.env`;
    const envFile = path.resolve(__dirname, '../..', options.folder,
      filePath);
    this.envConfig = dotenv.parse(fs.readFileSync(envFile));
  }

  get(key: string): string {
    return this.envConfig[key];
  }
}
```

Now our `ConfigService` knows how to find the `.env` file in the folder we've specified in `options`.

Our remaining task is to somehow inject the `options` object from the `register()` step into our `ConfigService`. And of course, we'll use *dependency injection* to do it. This is a key point, so make sure you understand it. Our `ConfigModule` is providing `ConfigService`. `ConfigService` in turn depends on the `options` object that is only supplied at run-time. So, at run-time, we'll need to first bind the `options` object to the Nest IoC container, and then have Nest inject it into our `ConfigService`. Remember from the **Custom providers** chapter that providers can `include any value` not just services, so we're fine using dependency injection to handle a simple `options` object.

Let's tackle binding the `options` object to the IoC container first. We do this in our static `register()` method. Remember that we are dynamically constructing a module, and one of the properties of a module is its list of providers. So what we need to do is define our `options` object as a provider. This will make it injectable into the `ConfigService`, which we'll take advantage of in the next step. In the code below, pay attention to the `providers` array:

```
import { DynamicModule, Module } from '@nestjs/common';
import { ConfigService } from './config.service';

@Module({})
export class ConfigModule {
  static register(options: Record<string, any>): DynamicModule {
```

```

return {
  module: ConfigModule,
  providers: [
    {
      provide: 'CONFIG_OPTIONS',
      useValue: options,
    },
    ConfigService,
  ],
  exports: [ConfigService],
};

}
}

```

Now we can complete the process by injecting the '`'CONFIG_OPTIONS'`' provider into the `ConfigService`. Recall that when we define a provider using a non-class token we need to use the `@Inject()` decorator [as described here](#).

```

import * as dotenv from 'dotenv';
import * as fs from 'fs';
import * as path from 'path';
import { Injectable, Inject } from '@nestjs/common';
import { EnvConfig } from './interfaces';

@Injectable()
export class ConfigService {
  private readonly envConfig: EnvConfig;

  constructor(@Inject('CONFIG_OPTIONS') private options: Record<string, any>) {
    const filePath = `${process.env.NODE_ENV || 'development'}.env`;
    const envFile = path.resolve(__dirname, '../..', options.folder, filePath);
    this.envConfig = dotenv.parse(fs.readFileSync(envFile));
  }

  get(key: string): string {
    return this.envConfig[key];
  }
}

```

One final note: for simplicity we used a string-based injection token (`'CONFIG_OPTIONS'`) above, but best practice is to define it as a constant (or `Symbol`) in a separate file, and import that file. For example:

```
export const CONFIG_OPTIONS = 'CONFIG_OPTIONS';
```

## Example

A full example of the code in this chapter can be found [here](#).

## Community guidelines

You may have seen the use for methods like `forRoot`, `register`, and `forFeature` around some of the `@nestjs/` packages and may be wondering what the difference for all of these methods are. There is no hard rule about this, but the `@nestjs/` packages try to follow these guidelines:

When creating a module with:

- `register`, you are expecting to configure a dynamic module with a specific configuration for use only by the calling module. For example, with Nest's `@nestjs/axios:HttpModule.register({{ ' }} baseUrl: 'someUrl' {{ ' }} )`. If, in another module you use `HttpModule.register({{ ' }} baseUrl: 'somewhere else' {{ ' }} )`, it will have the different configuration. You can do this for as many modules as you want.
- `forRoot`, you are expecting to configure a dynamic module once and reuse that configuration in multiple places (though possibly unknowingly as it's abstracted away). This is why you have one `GraphQLModule.forRoot()`, one `TypeOrmModule.forRoot()`, etc.
- `forFeature`, you are expecting to use the configuration of a dynamic module's `forRoot` but need to modify some configuration specific to the calling module's needs (i.e. which repository this module should have access to, or the context that a logger should use.)

All of these, usually, have their `async` counterparts as well, `registerAsync`, `forRootAsync`, and `forFeatureAsync`, that mean the same thing, but use Nest's Dependency Injection for the configuration as well.

## Configurable module builder

As manually creating highly configurable, dynamic modules that expose `async` methods (`registerAsync`, `forRootAsync`, etc.) is quite complicated, especially for newcomers, Nest exposes the `ConfigurableModuleBuilder` class that facilitates this process and lets you construct a module "blueprint" in just a few lines of code.

For example, let's take the example we used above (`ConfigModule`) and convert it to use the `ConfigurableModuleBuilder`. Before we start, let's make sure we create a dedicated interface that represents what options our `ConfigModule` takes in.

```
export interface ConfigModuleOptions {  
  folder: string;  
}
```

With this in place, create a new dedicated file (alongside the existing `config.module.ts` file) and name it `config.module-definition.ts`. In this file, let's utilize the `ConfigurableModuleBuilder` to construct `ConfigModule` definition.

```

@@filename(config.module-definition)
import { ConfigurableModuleBuilder } from '@nestjs/common';
import { ConfigModuleOptions } from './interfaces/config-module-
options.interface';

export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder<ConfigModuleOptions>().build();
@@switch
import { ConfigurableModuleBuilder } from '@nestjs/common';

export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder().build();

```

Now let's open up the `config.module.ts` file and modify its implementation to leverage the auto-generated `ConfigurableModuleClass`:

```

import { Module } from '@nestjs/common';
import { ConfigService } from './config.service';
import { ConfigurableModuleClass } from './config.module-definition';

@Module({
  providers: [ConfigService],
  exports: [ConfigService],
})
export class ConfigModule extends ConfigurableModuleClass {}

```

Extending the `ConfigurableModuleClass` means that `ConfigModule` provides now not only the `register` method (as previously with the custom implementation), but also the `registerAsync` method which allows consumers asynchronously configure that module, for example, by supplying async factories:

```

@Module({
  imports: [
    ConfigModule.register({ folder: './config' }),
    // or alternatively:
    // ConfigModule.registerAsync({
    //   useFactory: () => {
    //     return {
    //       folder: './config',
    //     }
    //   },
    //   inject: [...any extra dependencies...]
    // }),
  ],
})
export class AppModule {}

```

Lastly, let's update the `ConfigService` class to inject the generated module options' provider instead of the '`CONFIG_OPTIONS`' that we used so far.

```
@Injectable()
export class ConfigService {
  constructor(@Inject(MODULE_OPTIONS_TOKEN) private options:
ConfigModuleOptions) { ... }
}
```

## Custom method key

`ConfigurableModuleClass` by default provides the `register` and its counterpart `registerAsync` methods. To use a different method name, use the `ConfigurableModuleBuilder#setClassMethodName` method, as follows:

```
@@filename(config.module-definition)
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder<ConfigModuleOptions>
().setClassMethodName('forRoot').build();
@@switch
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder().setClassMethodName('forRoot').build();
```

This construction will instruct `ConfigurableModuleBuilder` to generate a class that exposes `forRoot` and `forRootAsync` instead. Example:

```
@Module({
  imports: [
    ConfigModule.forRoot({ folder: './config' }), // <-- note the use of
"forRoot" instead of "register"
    // or alternatively:
    // ConfigModule.forRootAsync({
    //   useFactory: () => {
    //     return {
    //       folder: './config',
    //     }
    //   },
    //   inject: [...any extra dependencies...]
    // }),
  ],
})
export class AppModule {}
```

## Custom options factory class

Since the `registerAsync` method (or `forRootAsync` or any other name, depending on the configuration) lets consumer pass a provider definition that resolves to the module configuration, a library consumer could potentially supply a class to be used to construct the configuration object.

```
@Module({
  imports: [
    ConfigModule.registerAsync({
      useClass: ConfigModuleOptionsFactory,
    }),
  ],
})
export class AppModule {}
```

This class, by default, must provide the `create()` method that returns a module configuration object. However, if your library follows a different naming convention, you can change that behavior and instruct `ConfigurableModuleBuilder` to expect a different method, for example, `createConfigOptions`, using the `ConfigurableModuleBuilder#setFactoryMethodName` method:

```
@@filename(config.module-definition)
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder<ConfigModuleOptions>
  ().setFactoryMethodName('createConfigOptions').build();
@@switch
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new
ConfigurableModuleBuilder().setFactoryMethodName('createConfigOptions').bu
ild();
```

Now, `ConfigModuleOptionsFactory` class must expose the `createConfigOptions` method (instead of `create`):

```
@Module({
  imports: [
    ConfigModule.registerAsync({
      useClass: ConfigModuleOptionsFactory, // <-- this class must provide
      the "createConfigOptions" method
    }),
  ],
})
export class AppModule {}
```

## Extra options

There are edge-cases when your module may need to take extra options that determine how it is supposed to behave (a nice example of such an option is the `isGlobal` flag - or just `global`) that at the same time,

shouldn't be included in the `MODULE_OPTIONS_TOKEN` provider (as they are irrelevant to services/providers registered within that module, for example, `ConfigService` does not need to know whether its host module is registered as a global module).

In such cases, the `ConfigurableModuleBuilder#setExtras` method can be used. See the following example:

```
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } = new
ConfigurableModuleBuilder<ConfigModuleOptions>()
  .setExtras(
    {
      isGlobal: true,
    },
    (definition, extras) => ({
      ...definition,
      global: extras.isGlobal,
    }),
  )
  .build();
```

In the example above, the first argument passed into the `setExtras` method is an object containing default values for the "extra" properties. The second argument is a function that takes an auto-generated module definitions (with `provider`, `exports`, etc.) and `extras` object which represents extra properties (either specified by the consumer or defaults). The returned value of this function is a modified module definition. In this specific example, we're taking the `extras.isGlobal` property and assigning it to the `global` property of the module definition (which in turn determines whether a module is global or not, read more [here](#)).

Now when consuming this module, the additional `isGlobal` flag can be passed in, as follows:

```
@Module({
  imports: [
    ConfigModule.register({
      isGlobal: true,
      folder: './config',
    }),
  ],
})
export class AppModule {}
```

However, since `isGlobal` is declared as an "extra" property, it won't be available in the `MODULE_OPTIONS_TOKEN` provider:

```
@Injectable()
export class ConfigService {
  constructor(@Inject(MODULE_OPTIONS_TOKEN) private options:
  ConfigModuleOptions) {
```

```
// "options" object will not have the "isGlobal" property
// ...
}
}
```

## Extending auto-generated methods

The auto-generated static methods (`register`, `registerAsync`, etc.) can be extended if needed, as follows:

```
import { Module } from '@nestjs/common';
import { ConfigService } from './config.service';
import { ConfigurableModuleClass, ASYNC_OPTIONS_TYPE, OPTIONS_TYPE } from
'./config.module-definition';

@Module({
  providers: [ConfigService],
  exports: [ConfigService],
})
export class ConfigModule extends ConfigurableModuleClass {
  static register(options: typeof OPTIONS_TYPE): DynamicModule {
    return {
      // your custom logic here
      ...super.register(options),
    };
  }

  static registerAsync(options: typeof ASYNC_OPTIONS_TYPE): DynamicModule
{
  return {
    // your custom logic here
    ...super.registerAsync(options),
  };
}
}
```

Note the use of `OPTIONS_TYPE` and `ASYNC_OPTIONS_TYPE` types that must be exported from the module definition file:

```
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN,
OPTIONS_TYPE, ASYNC_OPTIONS_TYPE } = new
ConfigurableModuleBuilder<ConfigModuleOptions>().build();
```

## Custom providers

In earlier chapters, we touched on various aspects of **Dependency Injection (DI)** and how it is used in Nest. One example of this is the [constructor based](#) dependency injection used to inject instances (often service providers) into classes. You won't be surprised to learn that Dependency Injection is built into the Nest core in a fundamental way. So far, we've only explored one main pattern. As your application grows more complex, you may need to take advantage of the full features of the DI system, so let's explore them in more detail.

### DI fundamentals

Dependency injection is an [inversion of control \(IoC\)](#) technique wherein you delegate instantiation of dependencies to the IoC container (in our case, the NestJS runtime system), instead of doing it in your own code imperatively. Let's examine what's happening in this example from the [Providers chapter](#).

First, we define a provider. The [@Injectable\(\)](#) decorator marks the [CatsService](#) class as a provider.

```
@@filename(cats.service)
import { Injectable } from '@nestjs/common';
import { Cat } from './interfaces/cat.interface';

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [];

  findAll(): Cat[] {
    return this.cats;
  }
}

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

@Injectable()
export class CatsService {
  constructor() {
    this.cats = [];
  }

  findAll() {
    return this.cats;
  }
}
```

Then we request that Nest inject the provider into our controller class:

```
@@filename(cats.controller)
import { Controller, Get } from '@nestjs/common';
import { CatsService } from './cats.service';
```

```

import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
    constructor(private catsService: CatsService) {}

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

@switch
import { Controller, Get, Bind, Dependencies } from '@nestjs/common';
import { CatsService } from './cats.service';

@Controller('cats')
@Dependencies(CatsService)
export class CatsController {
    constructor(catsService) {
        this.catsService = catsService;
    }

    @Get()
    async findAll() {
        return this.catsService.findAll();
    }
}

```

Finally, we register the provider with the Nest IoC container:

```

@@filename(app.module)
import { Module } from '@nestjs/common';
import { CatsController } from './cats/cats.controller';
import { CatsService } from './cats/cats.service';

@Module({
    controllers: [CatsController],
    providers: [CatsService],
})
export class AppModule {}

```

What exactly is happening under the covers to make this work? There are three key steps in the process:

1. In `cats.service.ts`, the `@Injectable()` decorator declares the `CatsService` class as a class that can be managed by the Nest IoC container.
2. In `cats.controller.ts`, `CatsController` declares a dependency on the `CatsService` token with constructor injection:

```
constructor(private catsService: CatsService)
```

3. In `app.module.ts`, we associate the token `CatsService` with the class `CatsService` from the `cats.service.ts` file. We'll [see below](#) exactly how this association (also called *registration*) occurs.

When the Nest IoC container instantiates a `CatsController`, it first looks for any dependencies\*. When it finds the `CatsService` dependency, it performs a lookup on the `CatsService` token, which returns the `CatsService` class, per the registration step (#3 above). Assuming `SINGLETON` scope (the default behavior), Nest will then either create an instance of `CatsService`, cache it, and return it, or if one is already cached, return the existing instance.

\*This explanation is a bit simplified to illustrate the point. One important area we glossed over is that the process of analyzing the code for dependencies is very sophisticated, and happens during application bootstrapping. One key feature is that dependency analysis (or "creating the dependency graph"), is **transitive**. In the above example, if the `CatsService` itself had dependencies, those too would be resolved. The dependency graph ensures that dependencies are resolved in the correct order – essentially "bottom up". This mechanism relieves the developer from having to manage such complex dependency graphs.

## Standard providers

Let's take a closer look at the `@Module()` decorator. In `app.module`, we declare:

```
@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
```

The `providers` property takes an array of `providers`. So far, we've supplied those providers via a list of class names. In fact, the syntax `providers: [CatsService]` is short-hand for the more complete syntax:

```
providers: [
  {
    provide: CatsService,
    useClass: CatsService,
  },
];
```

Now that we see this explicit construction, we can understand the registration process. Here, we are clearly associating the token `CatsService` with the class `CatsService`. The short-hand notation is merely a convenience to simplify the most common use-case, where the token is used to request an instance of a class by the same name.

## Custom providers

What happens when your requirements go beyond those offered by *Standard providers*? Here are a few examples:

- You want to create a custom instance instead of having Nest instantiate (or return a cached instance of) a class
- You want to re-use an existing class in a second dependency
- You want to override a class with a mock version for testing

Nest allows you to define Custom providers to handle these cases. It provides several ways to define custom providers. Let's walk through them.

**info Hint** If you are having problems with dependency resolution you can set the `NEST_DEBUG` environment variable and get extra dependency resolution logs during startup.

### Value providers: `useValue`

The `useValue` syntax is useful for injecting a constant value, putting an external library into the Nest container, or replacing a real implementation with a mock object. Let's say you'd like to force Nest to use a mock `CatsService` for testing purposes.

```
import { CatsService } from './cats.service';

const mockCatsService = {
  /* mock implementation
  ...
 */
};

@Module({
  imports: [CatsModule],
  providers: [
    {
      provide: CatsService,
      useValue: mockCatsService,
    },
  ],
})
export class AppModule {}
```

In this example, the `CatsService` token will resolve to the `mockCatsService` mock object. `useValue` requires a value - in this case a literal object that has the same interface as the `CatsService` class it is replacing. Because of TypeScript's [structural typing](#), you can use any object that has a compatible interface, including a literal object or a class instance instantiated with `new`.

### Non-class-based provider tokens

So far, we've used class names as our provider tokens (the value of the `provide` property in a provider listed in the `providers` array). This is matched by the standard pattern used with [constructor based injection](#), where the token is also a class name. (Refer back to [DI Fundamentals](#) for a refresher on tokens if

this concept isn't entirely clear). Sometimes, we may want the flexibility to use strings or symbols as the DI token. For example:

```
import { connection } from './connection';

@Module({
  providers: [
    {
      provide: 'CONNECTION',
      useValue: connection,
    },
  ],
})
export class AppModule {}
```

In this example, we are associating a string-valued token ('**CONNECTION**') with a pre-existing **connection** object we've imported from an external file.

**warning** **Notice** In addition to using strings as token values, you can also use JavaScript **symbols** or TypeScript **enums**.

We've previously seen how to inject a provider using the standard **constructor based injection** pattern. This pattern **requires** that the dependency be declared with a class name. The '**CONNECTION**' custom provider uses a string-valued token. Let's see how to inject such a provider. To do so, we use the **@Inject()** decorator. This decorator takes a single argument - the token.

```
@@filename()
@.Injectable()
export class CatsRepository {
  constructor(@Inject('CONNECTION') connection: Connection) {}
}

@@switch
@ Injectable()
@Dependencies('CONNECTION')
export class CatsRepository {
  constructor(connection) {}
}
```

**info Hint** The **@Inject()** decorator is imported from **@nestjs/common** package.

While we directly use the string '**CONNECTION**' in the above examples for illustration purposes, for clean code organization, it's best practice to define tokens in a separate file, such as **constants.ts**. Treat them much as you would symbols or enums that are defined in their own file and imported where needed.

## Class providers: **useClass**

The **useClass** syntax allows you to dynamically determine a class that a token should resolve to. For example, suppose we have an abstract (or default) **ConfigService** class. Depending on the current

environment, we want Nest to provide a different implementation of the configuration service. The following code implements such a strategy.

```
const configServiceProvider = {
  provide: ConfigService,
  useClass:
    process.env.NODE_ENV === 'development'
      ? DevelopmentConfigService
      : ProductionConfigService,
};

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

Let's look at a couple of details in this code sample. You'll notice that we define `configServiceProvider` with a literal object first, then pass it in the module decorator's `providers` property. This is just a bit of code organization, but is functionally equivalent to the examples we've used thus far in this chapter.

Also, we have used the `ConfigService` class name as our token. For any class that depends on `ConfigService`, Nest will inject an instance of the provided class (`DevelopmentConfigService` or `ProductionConfigService`) overriding any default implementation that may have been declared elsewhere (e.g., a `ConfigService` declared with an `@Injectable()` decorator).

## Factory providers: `useFactory`

The `useFactory` syntax allows for creating providers **dynamically**. The actual provider will be supplied by the value returned from a factory function. The factory function can be as simple or complex as needed. A simple factory may not depend on any other providers. A more complex factory can itself inject other providers it needs to compute its result. For the latter case, the factory provider syntax has a pair of related mechanisms:

1. The factory function can accept (optional) arguments.
2. The (optional) `inject` property accepts an array of providers that Nest will resolve and pass as arguments to the factory function during the instantiation process. Also, these providers can be marked as optional. The two lists should be correlated: Nest will pass instances from the `inject` list as arguments to the factory function in the same order. The example below demonstrates this.

```
@@filename()
const connectionProvider = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider: OptionsProvider, optionalProvider?: string) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider, { token: 'SomeOptionalProvider', optional:
```

```

    true }],
    // \_____/
    // This provider
    // is mandatory.
};

@Module({
  providers: [
    connectionProvider,
    OptionsProvider,
    // { provide: 'SomeOptionalProvider', useValue: 'anything' },
  ],
})
export class AppModule {}

@switch
const connectionProvider = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider, optionalProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider, { token: 'SomeOptionalProvider', optional:
true }],
  // \_____/
  // This provider
  // is mandatory.
};

@Module({
  providers: [
    connectionProvider,
    OptionsProvider,
    // { provide: 'SomeOptionalProvider', useValue: 'anything' },
  ],
})
export class AppModule {}

```

## Alias providers: `useExisting`

The `useExisting` syntax allows you to create aliases for existing providers. This creates two ways to access the same provider. In the example below, the (string-based) token '`'AliasedLoggerService'`' is an alias for the (class-based) token `LoggerService`. Assume we have two different dependencies, one for '`'AliasedLoggerService'`' and one for `LoggerService`. If both dependencies are specified with `SINGLETON` scope, they'll both resolve to the same instance.

```

@Injectable()
class LoggerService {
  /* implementation details */
}

```

```

const loggerAliasProvider = {
  provide: 'AliasedLoggerService',
  useExisting: LoggerService,
};

@Module({
  providers: [LoggerService, loggerAliasProvider],
})
export class AppModule {}

```

## Non-service based providers

While providers often supply services, they are not limited to that usage. A provider can supply **any** value. For example, a provider may supply an array of configuration objects based on the current environment, as shown below:

```

const configFactory = {
  provide: 'CONFIG',
  useFactory: () => {
    return process.env.NODE_ENV === 'development' ? devConfig :
    prodConfig;
  },
};

@Module({
  providers: [configFactory],
})
export class AppModule {}

```

## Export custom provider

Like any provider, a custom provider is scoped to its declaring module. To make it visible to other modules, it must be exported. To export a custom provider, we can either use its token or the full provider object.

The following example shows exporting using the token:

```

@@filename()
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider: OptionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
})

```

```
    exports: ['CONNECTION'],
})
export class AppModule {}
@switch
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
  exports: ['CONNECTION'],
})
export class AppModule {}
```

Alternatively, export with the full provider object:

```
@@filename()
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider: OptionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
  exports: [connectionFactory],
})
export class AppModule {}
@switch
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
  exports: [connectionFactory],
})
export class AppModule {}
```



## Circular dependency

A circular dependency occurs when two classes depend on each other. For example, class A needs class B, and class B also needs class A. Circular dependencies can arise in Nest between modules and between providers.

While circular dependencies should be avoided where possible, you can't always do so. In such cases, Nest enables resolving circular dependencies between providers in two ways. In this chapter, we describe using **forward referencing** as one technique, and using the **ModuleRef** class to retrieve a provider instance from the DI container as another.

We also describe resolving circular dependencies between modules.

**warning** **Warning** A circular dependency might also be caused when using "barrel files"/index.ts files to group imports. Barrel files should be omitted when it comes to module/provider classes. For example, barrel files should not be used when importing files within the same directory as the barrel file, i.e. `cats/cats.controller` should not import `cats` to import the `cats/cats.service` file. For more details please also see [this github issue](#).

### Forward reference

A **forward reference** allows Nest to reference classes which aren't yet defined using the **forwardRef()** utility function. For example, if `CatsService` and `CommonService` depend on each other, both sides of the relationship can use `@Inject()` and the `forwardRef()` utility to resolve the circular dependency. Otherwise Nest won't instantiate them because all of the essential metadata won't be available. Here's an example:

```
@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(
    @Inject(forwardRef(() => CommonService))
    private commonService: CommonService,
  ) {}
}

@switch
@Injectable()
@Dependencies(forwardRef(() => CommonService))
export class CatsService {
  constructor(commonService) {
    this.commonService = commonService;
  }
}
```

**info** **Hint** The `forwardRef()` function is imported from the `@nestjs/common` package.

That covers one side of the relationship. Now let's do the same with `CommonService`:

```
@@filename(common.service)
@Injectable()
export class CommonService {
  constructor(
    @Inject(forwardRef(() => CatsService))
    private catsService: CatsService,
  ) {}
}

@switch
@Injectable()
@Dependencies(forwardRef(() => CatsService))
export class CommonService {
  constructor(catsService) {
    this.catsService = catsService;
  }
}
```

warning **Warning** The order of instantiation is indeterminate. Make sure your code does not depend on which constructor is called first. Having circular dependencies depend on providers with `Scope.REQUEST` can lead to undefined dependencies. More information available [here](#)

## ModuleRef class alternative

An alternative to using `forwardRef()` is to refactor your code and use the `ModuleRef` class to retrieve a provider on one side of the (otherwise) circular relationship. Learn more about the `ModuleRef` utility class [here](#).

## Module forward reference

In order to resolve circular dependencies between modules, use the same `forwardRef()` utility function on both sides of the modules association. For example:

```
@@filename(common.module)
@Module({
  imports: [forwardRef(() => CatsModule)],
})
export class CommonModule {}
```

That covers one side of the relationship. Now let's do the same with `CatsModule`:

```
@@filename(cats.module)
@Module({
  imports: [forwardRef(() => CommonModule)],
})
export class CatsModule {}
```

## Module reference

Nest provides the `ModuleRef` class to navigate the internal list of providers and obtain a reference to any provider using its injection token as a lookup key. The `ModuleRef` class also provides a way to dynamically instantiate both static and scoped providers. `ModuleRef` can be injected into a class in the normal way:

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(private moduleRef: ModuleRef) {}
}

@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }
}
```

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

## Retrieving instances

The `ModuleRef` instance (hereafter we'll refer to it as the **module reference**) has a `get()` method. This method retrieves a provider, controller, or injectable (e.g., guard, interceptor, etc.) that exists (has been instantiated) in the **current** module using its injection token/class name.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  private service: Service;
  constructor(private moduleRef: ModuleRef) {}

  onModuleInit() {
    this.service = this.moduleRef.get(Service);
  }
}

@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  onModuleInit() {
    this.service = this.moduleRef.get(Service);
  }
}
```

```
    }  
}
```

warning **Warning** You can't retrieve scoped providers (transient or request-scoped) with the `get()` method. Instead, use the technique described [below](#). Learn how to control scopes [here](#).

To retrieve a provider from the global context (for example, if the provider has been injected in a different module), pass the `{} ' {' } strict: false {} ' }'` option as a second argument to `get()`.

```
this.moduleRef.get(Service, { strict: false });
```

## Resolving scoped providers

To dynamically resolve a scoped provider (transient or request-scoped), use the `resolve()` method, passing the provider's injection token as an argument.

```
@@filename(cats.service)  
@Injectable()  
export class CatsService implements OnModuleInit {  
  private transientService: TransientService;  
  constructor(private moduleRef: ModuleRef) {}  
  
  async onModuleInit() {  
    this.transientService = await  
this.moduleRef.resolve(TransientService);  
  }  
}  
@switch  
@Injectable()  
@Dependencies(ModuleRef)  
export class CatsService {  
  constructor(moduleRef) {  
    this.moduleRef = moduleRef;  
  }  
  
  async onModuleInit() {  
    this.transientService = await  
this.moduleRef.resolve(TransientService);  
  }  
}
```

The `resolve()` method returns a unique instance of the provider, from its own **DI container sub-tree**. Each sub-tree has a unique **context identifier**. Thus, if you call this method more than once and compare instance references, you will see that they are not equal.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
    constructor(private moduleRef: ModuleRef) {}

    async onModuleInit() {
        const transientServices = await Promise.all([
            this.moduleRef.resolve(TransientService),
            this.moduleRef.resolve(TransientService),
        ]);
        console.log(transientServices[0] === transientServices[1]); // false
    }
}

@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
    constructor(moduleRef) {
        this.moduleRef = moduleRef;
    }

    async onModuleInit() {
        const transientServices = await Promise.all([
            this.moduleRef.resolve(TransientService),
            this.moduleRef.resolve(TransientService),
        ]);
        console.log(transientServices[0] === transientServices[1]); // false
    }
}
```

To generate a single instance across multiple `resolve()` calls, and ensure they share the same generated DI container sub-tree, you can pass a context identifier to the `resolve()` method. Use the `ContextIdFactory` class to generate a context identifier. This class provides a `create()` method that returns an appropriate unique identifier.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
    constructor(private moduleRef: ModuleRef) {}

    async onModuleInit() {
        const contextId = ContextIdFactory.create();
        const transientServices = await Promise.all([
            this.moduleRef.resolve(TransientService, contextId),
            this.moduleRef.resolve(TransientService, contextId),
        ]);
        console.log(transientServices[0] === transientServices[1]); // true
    }
}

@switch
```

```

@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    const contextId = ContextIdFactory.create();
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService, contextId),
      this.moduleRef.resolve(TransientService, contextId),
    ]);
    console.log(transientServices[0] === transientServices[1]); // true
  }
}

```

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

## Registering `REQUEST` provider

Manually generated context identifiers (with `ContextIdFactory.create()`) represent DI sub-trees in which `REQUEST` provider is `undefined` as they are not instantiated and managed by the Nest dependency injection system.

To register a custom `REQUEST` object for a manually created DI sub-tree, use the `ModuleRef#registerRequestByContextId()` method, as follows:

```

const contextId = ContextIdFactory.create();
this.moduleRef.registerRequestByContextId(/* YOUR_REQUEST_OBJECT */,
contextId);

```

## Getting current sub-tree

Occasionally, you may want to resolve an instance of a request-scoped provider within a `request context`. Let's say that `CatsService` is request-scoped and you want to resolve the `CatsRepository` instance which is also marked as a request-scoped provider. In order to share the same DI container sub-tree, you must obtain the current context identifier instead of generating a new one (e.g., with the `ContextIdFactory.create()` function, as shown above). To obtain the current context identifier, start by injecting the `request` object using `@Inject()` decorator.

```

@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(
    @Inject(REQUEST) private request: Record<string, unknown>,
  ) {}
}

```

```
@@switch
@Injectable()
@Dependencies(REQUEST)
export class CatsService {
  constructor(request) {
    this.request = request;
  }
}
```

info Hint Learn more about the request provider [here](#).

Now, 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:

```
const contextId = ContextIdFactory.getByRequest(this.request);
const catsRepository = await this.moduleRef.resolve(CatsRepository,
contextId);
```

## Instantiating custom classes dynamically

To dynamically instantiate a class that **wasn't previously registered** as a **provider**, use the module reference's `create()` method.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  private catsFactory: CatsFactory;
  constructor(private moduleRef: ModuleRef) {}

  async onModuleInit() {
    this.catsFactory = await this.moduleRef.create(CatsFactory);
  }
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    this.catsFactory = await this.moduleRef.create(CatsFactory);
  }
}
```

This technique enables you to conditionally instantiate different classes outside of the framework container.

## Lazy-loading modules

By default, modules are eagerly loaded, which means that as soon as the application loads, so do all the modules, whether or not they are immediately necessary. While this is fine for most applications, it may become a bottleneck for apps/workers running in the **serverless environment**, where the startup latency ("cold start") is crucial.

Lazy loading can help decrease bootstrap time by loading only modules required by the specific serverless function invocation. In addition, you could also load other modules asynchronously once the serverless function is "warm" to speed-up the bootstrap time for subsequent calls even further (deferred modules registration).

**info Hint** If you're familiar with the **Angular** framework, you might have seen the "lazy-loading modules" term before. Be aware that this technique is **functionally different** in Nest and so think about this as an entirely different feature that shares similar naming conventions.

**warning Warning** Do note that [lifecycle hooks methods](#) are not invoked in lazy loaded modules and services.

### Getting started

To load modules on-demand, Nest provides the **LazyModuleLoader** class that can be injected into a class in the normal way:

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(private lazyModuleLoader: LazyModuleLoader) {}
}

@@switch
@Injectable()
@Dependencies(LazyModuleLoader)
export class CatsService {
  constructor(lazyModuleLoader) {
    this.lazyModuleLoader = lazyModuleLoader;
  }
}
```

**info Hint** The **LazyModuleLoader** class is imported from the [@nestjs/core](#) package.

Alternatively, you can obtain a reference to the **LazyModuleLoader** provider from within your application bootstrap file ([main.ts](#)), as follows:

```
// "app" represents a Nest application instance
const lazyModuleLoader = app.get(LazyModuleLoader);
```

With this in place, you can now load any module using the following construction:

```
const { LazyModule } = await import('./lazy.module');
const moduleRef = await this.lazyModuleLoader.load(() => LazyModule);
```

info **Hint** "Lazy-loaded" modules are **cached** upon the first `LazyModuleLoader#load` method invocation. That means, each consecutive attempt to load `LazyModule` will be **very fast** and will return a cached instance, instead of loading the module again.

```
Load "LazyModule" attempt: 1
time: 2.379ms
Load "LazyModule" attempt: 2
time: 0.294ms
Load "LazyModule" attempt: 3
time: 0.303ms
```

Also, "lazy-loaded" modules share the same modules graph as those eagerly loaded on the application bootstrap as well as any other lazy modules registered later in your app.

Where `lazy.module.ts` is a TypeScript file that exports a **regular Nest module** (no extra changes are required).

The `LazyModuleLoader#load` method returns the **module reference** (of `LazyModule`) that lets you navigate the internal list of providers and obtain a reference to any provider using its injection token as a lookup key.

For example, let's say we have a `LazyModule` with the following definition:

```
@Module({
  providers: [LazyService],
  exports: [LazyService],
})
export class LazyModule {}
```

info **Hint** Lazy-loaded modules cannot be registered as **global modules** as it simply makes no sense (since they are registered lazily, on-demand when all the statically registered modules have been already instantiated). Likewise, registered **global enhancers** (guards/interceptors/etc.) **will not work** properly either.

With this, we could obtain a reference to the `LazyService` provider, as follows:

```
const { LazyModule } = await import('./lazy.module');
const moduleRef = await this.lazyModuleLoader.load(() => LazyModule);

const { LazyService } = await import('./lazy.service');
const lazyService = moduleRef.get(LazyService);
```

warning **Warning** If you use **Webpack**, make sure to update your `tsconfig.json` file - setting `compilerOptions.module` to `"esnext"` and adding `compilerOptions.moduleResolution` property with `"node"` as a value:

```
{  
  "compilerOptions": {  
    "module": "esnext",  
    "moduleResolution": "node",  
    ...  
  }  
}
```

With these options set up, you'll be able to leverage the [code splitting](#) feature.

## Lazy-loading controllers, gateways, and resolvers

Since controllers (or resolvers in GraphQL applications) in Nest represent sets of routes/paths/topics (or queries/mutations), you **cannot lazy load them** using the [LazyModuleLoader](#) class.

error **Warning** Controllers, [resolvers](#), and [gateways](#) registered inside lazy-loaded modules will not behave as expected. Similarly, you cannot register middleware functions (by implementing the [MiddlewareConsumer](#) interface) on-demand.

For example, let's say you're building a REST API (HTTP application) with a Fastify driver under the hood (using the [@nestjs/platform-fastify](#) package). Fastify does not let you register routes after the application is ready/successfully listening to messages. That means even if we analyzed route mappings registered in the module's controllers, all lazy-loaded routes wouldn't be accessible since there is no way to register them at runtime.

Likewise, some transport strategies we provide as part of the [@nestjs/microservices](#) package (including Kafka, gRPC, or RabbitMQ) require to subscribe/listen to specific topics/channels before the connection is established. Once your application starts listening to messages, the framework would not be able to subscribe/listen to new topics.

Lastly, the [@nestjs/graphql](#) package with the code first approach enabled automatically generates the GraphQL schema on-the-fly based on the metadata. That means, it requires all classes to be loaded beforehand. Otherwise, it would not be doable to create the appropriate, valid schema.

## Common use-cases

Most commonly, you will see lazy loaded modules in situations when your worker/cron job/lambda & serverless function/webhook must trigger different services (different logic) based on the input arguments (route path/date/query parameters, etc.). On the other hand, lazy-loading modules may not make too much sense for monolithic applications, where the startup time is rather irrelevant.

## Execution context

Nest provides several utility classes that help make it easy to write applications that function across multiple application contexts (e.g., Nest HTTP server-based, microservices and WebSockets application contexts). These utilities provide information about the current execution context which can be used to build generic [guards](#), [filters](#), and [interceptors](#) that can work across a broad set of controllers, methods, and execution contexts.

We cover two such classes in this chapter: [ArgumentsHost](#) and [ExecutionContext](#).

### ArgumentsHost class

The [ArgumentsHost](#) class provides methods for retrieving the arguments being passed to a handler. It allows choosing the appropriate context (e.g., HTTP, RPC (microservice), or WebSockets) to retrieve the arguments from. The framework provides an instance of [ArgumentsHost](#), typically referenced as a [host](#) parameter, in places where you may want to access it. For example, the [catch\(\)](#) method of an [exception filter](#) is called with an [ArgumentsHost](#) instance.

[ArgumentsHost](#) simply acts as an abstraction over a handler's arguments. For example, for HTTP server applications (when `@nestjs/platform-express` is being used), the [host](#) object encapsulates Express's `[request, response, next]` array, where [request](#) is the request object, [response](#) is the response object, and [next](#) is a function that controls the application's request-response cycle. On the other hand, for [GraphQL](#) applications, the [host](#) object contains the `[root, args, context, info]` array.

### Current application context

When building generic [guards](#), [filters](#), and [interceptors](#) which are meant to run across multiple application contexts, we need a way to determine the type of application that our method is currently running in. Do this with the [getType\(\)](#) method of [ArgumentsHost](#):

```
if (host.getType() === 'http') {
    // do something that is only important in the context of regular HTTP
    // requests (REST)
} else if (host.getType() === 'rpc') {
    // do something that is only important in the context of Microservice
    // requests
} else if (host.getType<GqlContextType>() === 'graphql') {
    // do something that is only important in the context of GraphQL
    // requests
}
```

**info Hint** The [GqlContextType](#) is imported from the [@nestjs/graphql](#) package.

With the application type available, we can write more generic components, as shown below.

### Host handler arguments

To retrieve the array of arguments being passed to the handler, one approach is to use the host object's `getArgs()` method.

```
const [req, res, next] = host.getArgs();
```

You can pluck a particular argument by index using the `getArgByIndex()` method:

```
const request = host.getArgByIndex(0);
const response = host.getArgByIndex(1);
```

In these examples we retrieved the request and response objects by index, which is not typically recommended as it couples the application to a particular execution context. Instead, you can make your code more robust and reusable by using one of the `host` object's utility methods to switch to the appropriate application context for your application. The context switch utility methods are shown below.

```
/** 
 * Switch context to RPC.
 */
switchToRpc(): RpcArgumentsHost;
/** 
 * Switch context to HTTP.
 */
switchToHttp(): HttpArgumentsHost;
/** 
 * Switch context to WebSockets.
 */
switchToWs(): WsArgumentsHost;
```

Let's rewrite the previous example using the `switchToHttp()` method. The `host.switchToHttp()` helper call returns an `HttpArgumentsHost` object that is appropriate for the HTTP application context. The `HttpArgumentsHost` object has two useful methods we can use to extract the desired objects. We also use the Express type assertions in this case to return native Express typed objects:

```
const ctx = host.switchToHttp();
const request = ctx.getRequest<Request>();
const response = ctx.getResponse<Response>();
```

Similarly `WsArgumentsHost` and `RpcArgumentsHost` have methods to return appropriate objects in the microservices and WebSockets contexts. Here are the methods for `WsArgumentsHost`:

```
export interface WsArgumentsHost {
  /**
   * Returns the data object.
```

```

/*
getData<T>(): T;
/**
 * Returns the client object.
 */
getClient<T>(): T;
}

```

Following are the methods for [RpcArgumentsHost](#):

```

export interface RpcArgumentsHost {
    /**
     * Returns the data object.
     */
    getData<T>(): T;

    /**
     * Returns the context object.
     */
    getContext<T>(): T;
}

```

## ExecutionContext class

[ExecutionContext](#) extends [ArgumentsHost](#), providing additional details about the current execution process. Like [ArgumentsHost](#), Nest provides an instance of [ExecutionContext](#) in places you may need it, such as in the `canActivate()` method of a [guard](#) and the `intercept()` method of an [interceptor](#). It provides the following methods:

```

export interface ExecutionContext extends ArgumentsHost {
    /**
     * Returns the type of the controller class which the current handler
     belongs to.
     */
    getClass<T>(): Type<T>;
    /**
     * Returns a reference to the handler (method) that will be invoked next
     in the
     * request pipeline.
     */
    getHandler(): Function;
}

```

The `getHandler()` method returns a reference to the handler about to be invoked. The `getClass()` method returns the type of the [Controller](#) class which this particular handler belongs to. For example, in an HTTP context, if the currently processed request is a [POST](#) request, bound to the `create()` method on

the `CatsController`, `getHandler()` returns a reference to the `create()` method and `getClass()` returns the `CatsController` type (not instance).

```
const methodKey = ctx.getHandler().name; // "create"
const className = ctx.getClass().name; // "CatsController"
```

The ability to access references to both the current class and handler method provides great flexibility. Most importantly, it gives us the opportunity to access the metadata set through either decorators created via `Reflector#createDecorator` or the built-in `@SetMetadata()` decorator from within guards or interceptors. We cover this use case below.

## Reflection and metadata

Nest provides the ability to attach **custom metadata** to route handlers through decorators created via `Reflector#createDecorator` method, and the built-in `@SetMetadata()` decorator. In this section, let's compare the two approaches and see how to access the metadata from within a guard or interceptor.

To create strongly-typed decorators using `Reflector#createDecorator`, we need to specify the type argument. For example, let's create a `Roles` decorator that takes an array of strings as an argument.

```
@@filename(roles.decorator)
import { Reflector } from '@nestjs/core';

export const Roles = Reflector.createDecorator<string[]>();
```

The `Roles` decorator here is a function that takes a single argument of type `string[]`.

Now, to use this decorator, we simply annotate the handler with it:

```
@@filename(cats.controller)
@Post()
@Roles(['admin'])
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@switch
@Post()
@Roles(['admin'])
@Bind(Body())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

Here we've attached the `Roles` decorator metadata to the `create()` method, indicating that only users with the `admin` role should be allowed to access this route.

To access the route's role(s) (custom metadata), we'll use the `Reflector` helper class again. `Reflector` can be injected into a class in the normal way:

```
@@filename(roles.guard)
@Injectable()
export class RolesGuard {
  constructor(private reflector: Reflector) {}
}

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

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

Now, to read the handler metadata, use the `get()` method:

```
const roles = this.reflector.get(Roles, context.getHandler());
```

The `Reflector#get` method allows us to easily access the metadata by passing in two arguments: a decorator reference and a `context` (decorator target) to retrieve the metadata from. In this example, the specified `decorator` is `Roles` (refer back to the `roles.decorator.ts` file above). The context is provided by the call to `context.getHandler()`, which results in extracting the metadata for the currently processed route handler. Remember, `getHandler()` gives us a `reference` to the route handler function.

Alternatively, we may organize our controller by applying metadata at the controller level, applying to all routes in the controller class.

```
@@filename(cats.controller)
@Roles(['admin'])
@Controller('cats')
export class CatsController {}

@@switch
@Roles(['admin'])
@Controller('cats')
export class CatsController {}
```

In this case, to extract controller metadata, we pass `context.getClass()` as the second argument (to provide the controller class as the context for metadata extraction) instead of `context.getHandler()`:

```
@@filename(roles.guard)
const roles = this.reflector.get(Roles, context.getClass());
```

Given the ability to provide metadata at multiple levels, you may need to extract and merge metadata from several contexts. The [Reflector](#) class provides two utility methods used to help with this. These methods extract **both** controller and method metadata at once, and combine them in different ways.

Consider the following scenario, where you've supplied [Roles](#) metadata at both levels.

```
@@filename(cats.controller)
@Roles(['user'])
@Controller('cats')
export class CatsController {
  @Post()
  @Roles(['admin'])
  async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
  }
}
@switch
@Roles(['user'])
@Controller('cats')
export class CatsController {}
  @Post()
  @Roles(['admin'])
  @Bind(Body())
  async create(createCatDto) {
    this.catsService.create(createCatDto);
  }
}
```

If your intent is to specify '`'user'`' as the default role, and override it selectively for certain methods, you would probably use the [getAllAndOverride\(\)](#) method.

```
const roles = this.reflector.getAllAndOverride(Roles,
[context.getHandler(), context.getClass()]);
```

A guard with this code, running in the context of the `create()` method, with the above metadata, would result in `roles` containing `['admin']`.

To get metadata for both and merge it (this method merges both arrays and objects), use the [getAllAndMerge\(\)](#) method:

```
const roles = this.reflector.getAllAndMerge(Roles, [context.getHandler(),
context.getClass()]);
```

This would result in `roles` containing `['user', 'admin']`.

For both of these merge methods, you pass the metadata key as the first argument, and an array of metadata target contexts (i.e., calls to the `getHandler()` and/or `getClass()` methods) as the second argument.

## Low-level approach

As mentioned earlier, instead of using `Reflector#createDecorator`, you can also use the built-in `@SetMetadata()` decorator to attach metadata to a handler.

```
@@filename(cats.controller)
@Post()
@SetMetadata('roles', ['admin'])
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@switch
@Post()
@SetMetadata('roles', ['admin'])
@Bind(Body())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

**info Hint** The `@SetMetadata()` decorator is imported from the `@nestjs/common` package.

With the construction above, we attached the `roles` metadata (`roles` is a metadata key and `['admin']` is the associated value) to the `create()` method. While this works, it's not good practice to use `@SetMetadata()` directly in your routes. Instead, you can create your own decorators, as shown below:

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

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

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

This approach is much cleaner and more readable, and somewhat resembles the `Reflector#createDecorator` approach. The difference is that with `@SetMetadata` you have more control over the metadata key and value, and also can create decorators that take more than one argument.

Now that we have a custom `@Roles()` decorator, we can use it to decorate the `create()` method.

```
@@filename(cats.controller)
@Post()
@Roles('admin')
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}

@@switch
@Post()
@Roles('admin')
@Bind(Body())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

To access the route's role(s) (custom metadata), we'll use the [Reflector](#) helper class again:

```
@@filename(roles.guard)
@Injectable()
export class RolesGuard {
  constructor(private reflector: Reflector) {}
}

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

**info Hint** The [Reflector](#) class is imported from the [@nestjs/core](#) package.

Now, to read the handler metadata, use the [get\(\)](#) method.

```
const roles = this.reflector.get<string[]>('roles', context.getHandler());
```

Here instead of passing a decorator reference, we pass the metadata **key** as the first argument (which in our case is '[roles](#)'). Everything else remains the same as in the [Reflector#createDecorator](#) example.

## Lifecycle Events

A Nest application, as well as every application element, has a lifecycle managed by Nest. Nest provides **lifecycle hooks** that give visibility into key lifecycle events, and the ability to act (run registered code on your modules, providers or controllers) when they occur.

### Lifecycle sequence

The following diagram depicts the sequence of key application lifecycle events, from the time the application is bootstrapped until the node process exits. We can divide the overall lifecycle into three phases: **initializing**, **running** and **terminating**. Using this lifecycle, you can plan for appropriate initialization of modules and services, manage active connections, and gracefully shutdown your application when it receives a termination signal.



### Lifecycle events

Lifecycle events happen during application bootstrapping and shutdown. Nest calls registered lifecycle hook methods on modules, providers and controllers at each of the following lifecycle events (**shutdown hooks** need to be enabled first, as described [below](#)). As shown in the diagram above, Nest also calls the appropriate underlying methods to begin listening for connections, and to stop listening for connections.

In the following table, `onModuleDestroy`, `beforeApplicationShutdown` and `onApplicationShutdown` are only triggered if you explicitly call `app.close()` or if the process receives a special system signal (such as SIGTERM) and you have correctly called `enableShutdownHooks` at application bootstrap (see below [Application shutdown](#) part).

Lifecycle hook method	Lifecycle event triggering the hook method call
<code>onModuleInit()</code>	Called once the host module's dependencies have been resolved.
<code>onApplicationBootstrap()</code>	Called once all modules have been initialized, but before listening for connections.
<code>onModuleDestroy()</code> *	Called after a termination signal (e.g., <code>SIGTERM</code> ) has been received.
<code>beforeApplicationShutdown()</code> *	Called after all <code>onModuleDestroy()</code> handlers have completed (Promises resolved or rejected); once complete (Promises resolved or rejected), all existing connections will be closed ( <code>app.close()</code> called).
<code>onApplicationShutdown()</code> *	Called after connections close ( <code>app.close()</code> resolves).

\* For these events, if you're not calling `app.close()` explicitly, you must opt-in to make them work with system signals such as `SIGTERM`. See [Application shutdown](#) below.

warning **Warning** The lifecycle hooks listed above are not triggered for **request-scoped** classes. Request-scoped classes are not tied to the application lifecycle and their lifespan is unpredictable. They are exclusively created for each request and automatically garbage-collected after the response is sent.

info **Hint** Execution order of `onModuleInit()` and `onApplicationBootstrap()` directly depends on the order of module imports, awaiting the previous hook.

## Usage

Each lifecycle hook is represented by an interface. Interfaces are technically optional because they do not exist after TypeScript compilation. Nonetheless, it's good practice to use them in order to benefit from strong typing and editor tooling. To register a lifecycle hook, implement the appropriate interface. For example, to register a method to be called during module initialization on a particular class (e.g., Controller, Provider or Module), implement the `OnModuleInit` interface by supplying an `onModuleInit()` method, as shown below:

```
@@filename()
import { Injectable, OnModuleInit } from '@nestjs/common';

@Injectable()
export class UsersController implements OnModuleInit {
  onModuleInit() {
    console.log(`The module has been initialized.`);
  }
}

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

@Injectable()
export class UsersController {
  onModuleInit() {
    console.log(`The module has been initialized.`);
  }
}
```

## Asynchronous initialization

Both the `OnModuleInit` and `OnApplicationBootstrap` hooks allow you to defer the application initialization process (return a `Promise` or mark the method as `async` and `await` an asynchronous method completion in the method body).

```
@@filename()
async onModuleInit(): Promise<void> {
  await this.fetch();
}

@@switch
async onModuleInit() {
```

```
    await this.fetch();
}
```

## Application shutdown

The `onModuleDestroy()`, `beforeApplicationShutdown()` and `onApplicationShutdown()` hooks are called in the terminating phase (in response to an explicit call to `app.close()` or upon receipt of system signals such as SIGTERM if opted-in). This feature is often used with [Kubernetes](#) to manage containers' lifecycles, by [Heroku](#) for dynos or similar services.

Shutdown hook listeners consume system resources, so they are disabled by default. To use shutdown hooks, you **must enable listeners** by calling `enableShutdownHooks()`:

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

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

  // Starts listening for shutdown hooks
  app.enableShutdownHooks();

  await app.listen(3000);
}
bootstrap();
```

**warning** **warning** Due to inherent platform limitations, NestJS has limited support for application shutdown hooks on Windows. You can expect `SIGINT` to work, as well as `SIGBREAK` and to some extent `SIGHUP` - [read more](#). However `SIGTERM` will never work on Windows because killing a process in the task manager is unconditional, "i.e., there's no way for an application to detect or prevent it". Here's some [relevant documentation](#) from libuv to learn more about how `SIGINT`, `SIGBREAK` and others are handled on Windows. Also, see Node.js documentation of [Process Signal Events](#)

**info** **Info** `enableShutdownHooks` consumes memory by starting listeners. In cases where you are running multiple Nest apps in a single Node process (e.g., when running parallel tests with Jest), Node may complain about excessive listener processes. For this reason, `enableShutdownHooks` is not enabled by default. Be aware of this condition when you are running multiple instances in a single Node process.

When the application receives a termination signal it will call any registered `onModuleDestroy()`, `beforeApplicationShutdown()`, then `onApplicationShutdown()` methods (in the sequence described above) with the corresponding signal as the first parameter. If a registered function awaits an asynchronous call (returns a promise), Nest will not continue in the sequence until the promise is resolved or rejected.

```
@@filename()
@Injectable()
```

```
class UsersService implements OnApplicationShutdown {  
    onApplicationShutdown(signal: string) {  
        console.log(signal); // e.g. "SIGINT"  
    }  
}  
@@switch  
@Injectable()  
class UsersService implements OnApplicationShutdown {  
    onApplicationShutdown(signal) {  
        console.log(signal); // e.g. "SIGINT"  
    }  
}
```

info **Info** Calling `app.close()` doesn't terminate the Node process but only triggers the `onModuleDestroy()` and `onApplicationShutdown()` hooks, so if there are some intervals, long-running background tasks, etc. the process won't be automatically terminated.

## Platform agnosticism

Nest is a platform-agnostic framework. This means you can develop **reusable logical parts** that can be used across different types of applications. For example, most components can be re-used without change across different underlying HTTP server frameworks (e.g., Express and Fastify), and even across different types of applications (e.g., HTTP server frameworks, Microservices with different transport layers, and Web Sockets).

### Build once, use everywhere

The **Overview** section of the documentation primarily shows coding techniques using HTTP server frameworks (e.g., apps providing a REST API or providing an MVC-style server-side rendered app). However, all those building blocks can be used on top of different transport layers ([microservices](#) or [websockets](#)).

Furthermore, Nest comes with a dedicated [GraphQL](#) module. You can use GraphQL as your API layer interchangeably with providing a REST API.

In addition, the [application context](#) feature helps to create any kind of Node.js application - including things like CRON jobs and CLI apps - on top of Nest.

Nest aspires to be a full-fledged platform for Node.js apps that brings a higher-level of modularity and reusability to your applications. Build once, use everywhere!

## Testing

Automated testing is considered an essential part of any serious software development effort. Automation makes it easy to repeat individual tests or test suites quickly and easily during development. This helps ensure that releases meet quality and performance goals. Automation helps increase coverage and provides a faster feedback loop to developers. Automation both increases the productivity of individual developers and ensures that tests are run at critical development lifecycle junctures, such as source code control check-in, feature integration, and version release.

Such tests often span a variety of types, including unit tests, end-to-end (e2e) tests, integration tests, and so on. While the benefits are unquestionable, it can be tedious to set them up. Nest strives to promote development best practices, including effective testing, so it includes features such as the following to help developers and teams build and automate tests. Nest:

- automatically scaffolds default unit tests for components and e2e tests for applications
- provides default tooling (such as a test runner that builds an isolated module/application loader)
- provides integration with [Jest](#) and [Supertest](#) out-of-the-box, while remaining agnostic to testing tools
- makes the Nest dependency injection system available in the testing environment for easily mocking components

As mentioned, you can use any **testing framework** that you like, as Nest doesn't force any specific tooling. Simply replace the elements needed (such as the test runner), and you will still enjoy the benefits of Nest's ready-made testing facilities.

## Installation

To get started, first install the required package:

```
$ npm i --save-dev @nestjs/testing
```

## Unit testing

In the following example, we test two classes: [CatsController](#) and [CatsService](#). As mentioned, [Jest](#) is provided as the default testing framework. It serves as a test-runner and also provides assert functions and test-double utilities that help with mocking, spying, etc. In the following basic test, we manually instantiate these classes, and ensure that the controller and service fulfill their API contract.

```
@@filename(cats.controller.spec)
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController: CatsController;
  let catsservice: CatsService;

  beforeEach(() => {
```

```
catsService = new CatsService();
catsController = new CatsController(catsService);
});

describe('findAll', () => {
  it('should return an array of cats', async () => {
    const result = ['test'];
    jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

    expect(await catsController.findAll()).toBe(result);
  });
});
});

@@switch
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController;
  let catsService;

  beforeEach(() => {
    catsService = new CatsService();
    catsController = new CatsController(catsService);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

      expect(await catsController.findAll()).toBe(result);
    });
  });
});
```

info **Hint** Keep your test files located near the classes they test. Testing files should have a `.spec` or `.test` suffix.

Because the above sample is trivial, we aren't really testing anything Nest-specific. Indeed, we aren't even using dependency injection (notice that we pass an instance of `CatsService` to our `catsController`). This form of testing - where we manually instantiate the classes being tested - is often called **isolated testing** as it is independent from the framework. Let's introduce some more advanced capabilities that help you test applications that make more extensive use of Nest features.

## Testing utilities

The `@nestjs/testing` package provides a set of utilities that enable a more robust testing process. Let's rewrite the previous example using the built-in `Test` class:

```
@@filename(cats.controller.spec)
import { Test } from '@nestjs/testing';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController: CatsController;
  let catsService: CatsService;

  beforeEach(async () => {
    const moduleRef = await Test.createTestingModule({
      controllers: [CatsController],
      providers: [CatsService],
    }).compile();

    catsService = moduleRef.get<CatsService>(CatsService);
    catsController = moduleRef.get<CatsController>(CatsController);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

      expect(await catsController.findAll()).toBe(result);
    });
  });
});

@@switch
import { Test } from '@nestjs/testing';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController;
  let catsService;

  beforeEach(async () => {
    const moduleRef = await Test.createTestingModule({
      controllers: [CatsController],
      providers: [CatsService],
    }).compile();

    catsService = moduleRef.get(CatsService);
    catsController = moduleRef.get(CatsController);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

      expect(await catsController.findAll()).toBe(result);
    });
  });
});
```

```
    });
  });
});
```

The `Test` class is useful for providing an application execution context that essentially mocks the full Nest runtime, but gives you hooks that make it easy to manage class instances, including mocking and overriding. The `Test` class has a `createTestingModule()` method that takes a module metadata object as its argument (the same object you pass to the `@Module()` decorator). This method returns a `TestingModule` instance which in turn provides a few methods. For unit tests, the important one is the `compile()` method. This method bootstraps a module with its dependencies (similar to the way an application is bootstrapped in the conventional `main.ts` file using `NestFactory.create()`), and returns a module that is ready for testing.

**info Hint** The `compile()` method is **asynchronous** and therefore has to be awaited. Once the module is compiled you can retrieve any **static** instance it declares (controllers and providers) using the `get()` method.

`TestingModule` inherits from the `module reference` class, and therefore its ability to dynamically resolve scoped providers (transient or request-scoped). Do this with the `resolve()` method (the `get()` method can only retrieve static instances).

```
const moduleRef = await Test.createTestingModule({
  controllers: [CatsController],
  providers: [CatsService],
}).compile();

catsService = await moduleRef.resolve(CatsService);
```

**warning Warning** The `resolve()` method returns a unique instance of the provider, from its own **DI container sub-tree**. Each sub-tree has a unique context identifier. Thus, if you call this method more than once and compare instance references, you will see that they are not equal.

**info Hint** Learn more about the module reference features [here](#).

Instead of using the production version of any provider, you can override it with a `custom provider` for testing purposes. For example, you can mock a database service instead of connecting to a live database. We'll cover overrides in the next section, but they're available for unit tests as well.

## Auto mocking

Nest also allows you to define a mock factory to apply to all of your missing dependencies. This is useful for cases where you have a large number of dependencies in a class and mocking all of them will take a long time and a lot of setup. To make use of this feature, the `createTestingModule()` will need to be chained up with the `useMocker()` method, passing a factory for your dependency mocks. This factory can take in an optional token, which is an instance token, any token which is valid for a Nest provider, and returns a mock implementation. The below is an example of creating a generic mocker using `jest-mock` and a specific mock for `CatsService` using `jest.fn()`.

```
// ...
import { ModuleMocker, MockFunctionMetadata } from 'jest-mock';

const moduleMocker = new ModuleMocker(global);

describe('CatsController', () => {
  let controller: CatsController;

  beforeEach(async () => {
    const moduleRef = await Test.createTestingModule({
      controllers: [CatsController],
    })
      .useMocker((token) => {
        const results = ['test1', 'test2'];
        if (token === CatsService) {
          return { findAll: jest.fn().mockResolvedValue(results) };
        }
        if (typeof token === 'function') {
          const mockMetadata = moduleMocker.getMetadata(token) as
MockFunctionMetadata<any, any>;
          const Mock = moduleMocker.generateFromMetadata(mockMetadata);
          return new Mock();
        }
      })
      .compile();

    controller = moduleRef.get(CatsController);
  });
});
```

You can also retrieve these mocks out of the testing container as you normally would custom providers, `moduleRef.get(CatsService)`.

**info Hint** A general mock factory, like `createMock` from `@golevelup/ts-jest` can also be passed directly.

**info Hint** `REQUEST` and `INQUIRER` providers cannot be auto-mocked because they're already pre-defined in the context. However, they can be *overwritten* using the custom provider syntax or by utilizing the `.overrideProvider` method.

## End-to-end testing

Unlike unit testing, which focuses on individual modules and classes, end-to-end (e2e) testing covers the interaction of classes and modules at a more aggregate level -- closer to the kind of interaction that end-users will have with the production system. As an application grows, it becomes hard to manually test the end-to-end behavior of each API endpoint. Automated end-to-end tests help us ensure that the overall behavior of the system is correct and meets project requirements. To perform e2e tests we use a similar configuration to the one we just covered in **unit testing**. In addition, Nest makes it easy to use the [Supertest](#) library to simulate HTTP requests.

```
@@filename(cats.e2e-spec)
import * as request from 'supertest';
import { Test } from '@nestjs/testing';
import { CatsModule } from '../../../../../src/cats/cats.module';
import { CatsService } from '../../../../../src/cats/cats.service';
import { INestApplication } from '@nestjs/common';

describe('Cats', () => {
  let app: INestApplication;
  let catsService = { findAll: () => ['test'] };

  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      imports: [CatsModule],
    })
      .overrideProvider(CatsService)
      .useValue(catsService)
      .compile();

    app = moduleRef.createNestApplication();
    await app.init();
  });

  it(`GET /cats`, () => {
    return request(app.getHttpServer())
      .get('/cats')
      .expect(200)
      .expect({
        data: catsService.findAll(),
      });
  });

  afterAll(async () => {
    await app.close();
  });
});

@@switch
import * as request from 'supertest';
import { Test } from '@nestjs/testing';
import { CatsModule } from '../../../../../src/cats/cats.module';
import { CatsService } from '../../../../../src/cats/cats.service';
import { INestApplication } from '@nestjs/common';

describe('Cats', () => {
  let app: INestApplication;
  let catsService = { findAll: () => ['test'] };

  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      imports: [CatsModule],
    })
      .overrideProvider(CatsService)
      .useValue(catsService)
```

```
.compile();

app = moduleRef.createNestApplication();
await app.init();
});

it(`/GET cats`, () => {
  return request(app.getHttpServer())
    .get('/cats')
    .expect(200)
    .expect({
      data: catsService.findAll(),
    });
});

afterAll(async () => {
  await app.close();
});
});
```

info **Hint** If you're using [Fastify](#) as your HTTP adapter, it requires a slightly different configuration, and has built-in testing capabilities:

```
let app: NestFastifyApplication;

beforeAll(async () => {
  app = moduleRef.createNestApplication<NestFastifyApplication>(new
FastifyAdapter());

  await app.init();
  await app.getHttpAdapter().getInstance().ready();
});

it(`/GET cats`, () => {
  return app
    .inject({
      method: 'GET',
      url: '/cats',
    })
    .then((result) => {
      expect(result.statusCode).toEqual(200);
      expect(result.payload).toEqual(/* expectedPayload */);
    });
});

afterAll(async () => {
  await app.close();
});
```

In this example, we build on some of the concepts described earlier. In addition to the `compile()` method we used earlier, we now use the `createNestApplication()` method to instantiate a full Nest runtime environment. We save a reference to the running app in our `app` variable so we can use it to simulate HTTP requests.

We simulate HTTP tests using the `request()` function from Supertest. We want these HTTP requests to route to our running Nest app, so we pass the `request()` function a reference to the HTTP listener that underlies Nest (which, in turn, may be provided by the Express platform). Hence the construction `request(app.getHttpServer())`. The call to `request()` hands us a wrapped HTTP Server, now connected to the Nest app, which exposes methods to simulate an actual HTTP request. For example, using `request(...).get('/cats')` will initiate a request to the Nest app that is identical to an **actual** HTTP request like `get '/cats'` coming in over the network.

In this example, we also provide an alternate (test-double) implementation of the `CatsService` which simply returns a hard-coded value that we can test for. Use `overrideProvider()` to provide such an alternate implementation. Similarly, Nest provides methods to override modules, guards, interceptors, filters and pipes with the `overrideModule()`, `overrideGuard()`, `overrideInterceptor()`, `overrideFilter()`, and `overridePipe()` methods respectively.

Each of the override methods (except for `overrideModule()`) returns an object with 3 different methods that mirror those described for [custom providers](#):

- `useClass`: you supply a class that will be instantiated to provide the instance to override the object (provider, guard, etc.).
- `useValue`: you supply an instance that will override the object.
- `useFactory`: you supply a function that returns an instance that will override the object.

On the other hand, `overrideModule()` returns an object with the `useModule()` method, which you can use to supply a module that will override the original module, as follows:

```
const moduleRef = await Test.createTestingModule({
  imports: [AppModule],
})
  .overrideModule(CatsModule)
  .useModule(AlternateCatsModule)
  .compile();
```

Each of the override method types, in turn, returns the `TestingModule` instance, and can thus be chained with other methods in the [fluent style](#). You should use `compile()` at the end of such a chain to cause Nest to instantiate and initialize the module.

Also, sometimes you may want to provide a custom logger e.g. when the tests are run (for example, on a CI server). Use the `setLogger()` method and pass an object that fulfills the `LoggerService` interface to instruct the `TestModuleBuilder` how to log during tests (by default, only "error" logs will be logged to the console).

**warning** **Warning** The `@nestjs/core` package exposes unique provider tokens with the `APP_` prefix to help on define global enhancers. Those tokens cannot be overridden since they can represent

multiple providers. Thus you can't use `.overrideProvider(APP_GUARD)` (and so on). If you want to override some global enhancer, follow [this workaround](#).

The compiled module has several useful methods, as described in the following table:

<code>createNestApplication()</code>	Creates and returns a Nest application ( <code>INestApplication</code> instance) based on the given module. Note that you must manually initialize the application using the <code>init()</code> method.
<code>createNestMicroservice()</code>	Creates and returns a Nest microservice ( <code>INestMicroservice</code> instance) based on the given module.
<code>get()</code>	Retrieves a static instance of a controller or provider (including guards, filters, etc.) available in the application context. Inherited from the <a href="#">module reference</a> class.
<code>resolve()</code>	Retrieves a dynamically created scoped instance (request or transient) of a controller or provider (including guards, filters, etc.) available in the application context. Inherited from the <a href="#">module reference</a> class.
<code>select()</code>	Navigates through the module's dependency graph; can be used to retrieve a specific instance from the selected module (used along with strict mode ( <code>strict: true</code> ) in <code>get()</code> method).

**info Hint** Keep your e2e test files inside the `test` directory. The testing files should have a `.e2e-spec` suffix.

## Overriding globally registered enhancers

If you have a globally registered guard (or pipe, interceptor, or filter), you need to take a few more steps to override that enhancer. To recap the original registration looks like this:

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

This is registering the guard as a "multi"-provider through the `APP_*` token. To be able to replace the `JwtAuthGuard` here, the registration needs to use an existing provider in this slot:

```
providers: [
  {
    provide: APP_GUARD,
    useExisting: JwtAuthGuard,
    // ^^^^^^^ notice the use of 'useExisting' instead of 'useClass'
  },
],
```

```
  JwtAuthGuard,  
],
```

info **Hint** Change the `useClass` to `useExisting` to reference a registered provider instead of having Nest instantiate it behind the token.

Now the `JwtAuthGuard` is visible to Nest as a regular provider that can be overridden when creating the `TestingModule`:

```
const moduleRef = await Test.createTestingModule({  
  imports: [AppModule],  
}  
  .overrideProvider(JwtAuthGuard)  
  .useClass(MockAuthGuard)  
  .compile();
```

Now all your tests will use the `MockAuthGuard` on every request.

## Testing request-scoped instances

`Request-scoped` providers are created uniquely for each incoming `request`. The instance is garbage-collected after the request has completed processing. This poses a problem, because we can't access a dependency injection sub-tree generated specifically for a tested request.

We know (based on the sections above) that the `resolve()` method can be used to retrieve a dynamically instantiated class. Also, as described [here](#), we know we can pass a unique context identifier to control the lifecycle of a DI container sub-tree. How do we leverage this in a testing context?

The strategy is to generate a context identifier beforehand and force Nest to use this particular ID to create a sub-tree for all incoming requests. In this way we'll be able to retrieve instances created for a tested request.

To accomplish this, use `jest.spyOn()` on the `ContextIdFactory`:

```
const contextId = ContextIdFactory.create();  
jest.spyOn(ContextIdFactory, 'getByRequest').mockImplementation(() =>  
  contextId);
```

Now we can use the `contextId` to access a single generated DI container sub-tree for any subsequent request.

```
catsService = await moduleRef.resolve(CatsService, contextId);
```

## Configuration

Applications often run in different **environments**. Depending on the environment, different configuration settings should be used. For example, usually the local environment relies on specific database credentials, valid only for the local DB instance. The production environment would use a separate set of DB credentials. Since configuration variables change, best practice is to [store configuration variables](#) in the environment.

Externally defined environment variables are visible inside Node.js through the `process.env` global. We could try to solve the problem of multiple environments by setting the environment variables separately in each environment. This can quickly get unwieldy, especially in the development and testing environments where these values need to be easily mocked and/or changed.

In Node.js applications, it's common to use `.env` files, holding key-value pairs where each key represents a particular value, to represent each environment. Running an app in different environments is then just a matter of swapping in the correct `.env` file.

A good approach for using this technique in Nest is to create a [ConfigModule](#) that exposes a [ConfigService](#) which loads the appropriate `.env` file. While you may choose to write such a module yourself, for convenience Nest provides the [@nestjs/config](#) package out-of-the box. We'll cover this package in the current chapter.

## Installation

To begin using it, we first install the required dependency.

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

**info Hint** The [@nestjs/config](#) package internally uses [dotenv](#).

**warning Note** [@nestjs/config](#) requires TypeScript 4.1 or later.

## Getting started

Once the installation process is complete, we can import the [ConfigModule](#). Typically, we'll import it into the root [AppModule](#) and control its behavior using the `.forRoot()` static method. During this step, environment variable key/value pairs are parsed and resolved. Later, we'll see several options for accessing the [ConfigService](#) class of the [ConfigModule](#) in our other feature modules.

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';

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

The above code will load and parse a `.env` file from the default location (the project root directory), merge key/value pairs from the `.env` file with environment variables assigned to `process.env`, and store the result in a private structure that you can access through the `ConfigService`. The `forRoot()` method registers the `ConfigService` provider, which provides a `get()` method for reading these parsed/merged configuration variables. Since `@nestjs/config` relies on `dotenv`, it uses that package's rules for resolving conflicts in environment variable names. When a key exists both in the runtime environment as an environment variable (e.g., via OS shell exports like `export DATABASE_USER=test`) and in a `.env` file, the runtime environment variable takes precedence.

A sample `.env` file looks something like this:

```
DATABASE_USER=test  
DATABASE_PASSWORD=test
```

## Custom env file path

By default, the package looks for a `.env` file in the root directory of the application. To specify another path for the `.env` file, set the `envFilePath` property of an (optional) options object you pass to `forRoot()`, as follows:

```
ConfigModule.forRoot({  
  envFilePath: '.development.env',  
});
```

You can also specify multiple paths for `.env` files like this:

```
ConfigModule.forRoot({  
  envFilePath: ['.env.development.local', '.env.development'],  
});
```

If a variable is found in multiple files, the first one takes precedence.

## Disable env variables loading

If you don't want to load the `.env` file, but instead would like to simply access environment variables from the runtime environment (as with OS shell exports like `export DATABASE_USER=test`), set the options object's `ignoreEnvFile` property to `true`, as follows:

```
ConfigModule.forRoot({  
  ignoreEnvFile: true,  
});
```

## Use module globally

When you want to use `ConfigModule` in other modules, you'll need to import it (as is standard with any Nest module). Alternatively, declare it as a `global module` by setting the options object's `isGlobal` property to `true`, as shown below. In that case, you will not need to import `ConfigModule` in other modules once it's been loaded in the root module (e.g., `AppModule`).

```
ConfigModule.forRoot({
  isGlobal: true,
});
```

## Custom configuration files

For more complex projects, you may utilize custom configuration files to return nested configuration objects. This allows you to group related configuration settings by function (e.g., database-related settings), and to store related settings in individual files to help manage them independently.

A custom configuration file exports a factory function that returns a configuration object. The configuration object can be any arbitrarily nested plain JavaScript object. The `process.env` object will contain the fully resolved environment variable key/value pairs (with `.env` file and externally defined variables resolved and merged as described [above](#)). Since you control the returned configuration object, you can add any required logic to cast values to an appropriate type, set default values, etc. For example:

```
@filename(config/configuration)
export default () => ({
  port: parseInt(process.env.PORT, 10) || 3000,
  database: {
    host: process.env.DATABASE_HOST,
    port: parseInt(process.env.DATABASE_PORT, 10) || 5432
  }
});
```

We load this file using the `load` property of the options object we pass to the `ConfigModule.forRoot()` method:

```
import configuration from './config/configuration';

@Module({
  imports: [
    ConfigModule.forRoot({
      load: [configuration],
    }),
  ],
})
export class AppModule {}
```

**info Notice** The value assigned to the `load` property is an array, allowing you to load multiple configuration files (e.g. `load: [databaseConfig, authConfig]`)

With custom configuration files, we can also manage custom files such as YAML files. Here is an example of a configuration using YAML format:

```
http:
  host: 'localhost'
  port: 8080

db:
  postgres:
    url: 'localhost'
    port: 5432
    database: 'yaml-db'

  sqlite:
    database: 'sqlite.db'
```

To read and parse YAML files, we can leverage the `js-yaml` package.

```
$ npm i js-yaml
$ npm i -D @types/js-yaml
```

Once the package is installed, we use `yaml#load` function to load YAML file we just created above.

```
@@filename(config/configuration)
import { readFileSync } from 'fs';
import * as yaml from 'js-yaml';
import { join } from 'path';

const YAML_CONFIG_FILENAME = 'config.yaml';

export default () => {
  return yaml.load(
    readFileSync(join(__dirname, YAML_CONFIG_FILENAME), 'utf8'),
  ) as Record<string, any>;
};
```

**warning Note** Nest CLI does not automatically move your "assets" (non-TS files) to the `dist` folder during the build process. To make sure that your YAML files are copied, you have to specify this in the `compilerOptions#assets` object in the `nest-cli.json` file. As an example, if the `config` folder is at the same level as the `src` folder, add `compilerOptions#assets` with the value `"assets": [{ "include": ".../config/*.yaml", "outDir": "./dist/config" }]`. Read more [here](#).

## Using the ConfigService

To access configuration values from our [ConfigService](#), we first need to inject [ConfigService](#). As with any provider, we need to import its containing module - the [ConfigModule](#) - into the module that will use it (unless you set the `isGlobal` property in the options object passed to the [ConfigModule.forRoot\(\)](#) method to `true`). Import it into a feature module as shown below.

```
@@filename(feature.module)
@Module({
  imports: [ConfigModule],
  // ...
})
```

Then we can inject it using standard constructor injection:

```
constructor(private configService: ConfigService) {}
```

**info Hint** The [ConfigService](#) is imported from the [@nestjs/config](#) package.

And use it in our class:

```
// get an environment variable
const dbUser = this.configService.get<string>('DATABASE_USER');

// get a custom configuration value
const dbHost = this.configService.get<string>('database.host');
```

As shown above, use the [configService.get\(\)](#) method to get a simple environment variable by passing the variable name. You can do TypeScript type hinting by passing the type, as shown above (e.g., `get<string>(...)`). The [get\(\)](#) method can also traverse a nested custom configuration object (created via a [Custom configuration file](#)), as shown in the second example above.

You can also get the whole nested custom configuration object using an interface as the type hint:

```
interface DatabaseConfig {
  host: string;
  port: number;
}

const dbConfig = this.configService.get<DatabaseConfig>('database');

// you can now use `dbConfig.port` and `dbConfig.host`
const port = dbConfig.port;
```

The `get()` method also takes an optional second argument defining a default value, which will be returned when the key doesn't exist, as shown below:

```
// use "localhost" when "database.host" is not defined
const dbHost = this.configService.get<string>('database.host',
'localhost');
```

`ConfigService` has two optional generics (type arguments). The first one is to help prevent accessing a config property that does not exist. Use it as shown below:

```
interface EnvironmentVariables {
  PORT: number;
  TIMEOUT: string;
}

// somewhere in the code
constructor(private configService: ConfigService<EnvironmentVariables>) {
  const port = this.configService.get('PORT', { infer: true });

  // TypeScript Error: this is invalid as the URL property is not defined
  // in EnvironmentVariables
  const url = this.configService.get('URL', { infer: true });
}
```

With the `infer` property set to `true`, the `ConfigService#get` method will automatically infer the property type based on the interface, so for example, `typeof port === "number"` (if you're not using `strictNullChecks` flag from TypeScript) since `PORT` has a `number` type in the `EnvironmentVariables` interface.

Also, with the `infer` feature, you can infer the type of a nested custom configuration object's property, even when using dot notation, as follows:

```
constructor(private configService: ConfigService<{ database: { host:
  string } }>) {
  const dbHost = this.configService.get('database.host', { infer: true
})!;
  // typeof dbHost === "string"
  //
  +--> non-null assertion operator
}
```

The second generic relies on the first one, acting as a type assertion to get rid of all `undefined` types that `ConfigService`'s methods can return when `strictNullChecks` is on. For instance:

```
// ...
constructor(private configService: ConfigService<{ PORT: number }, true>)
{
    //
    const port = this.configService.get('PORT', { infer: true });
    //      ^^^ The type of port will be 'number' thus you don't need TS type
    assertions anymore
}
```

## Configuration namespaces

The [ConfigModule](#) allows you to define and load multiple custom configuration files, as shown in [Custom configuration files](#) above. You can manage complex configuration object hierarchies with nested configuration objects as shown in that section. Alternatively, you can return a "namespaced" configuration object with the [registerAs\(\)](#) function as follows:

```
@filename(config/database.config)
export default registerAs('database', () => ({
    host: process.env.DATABASE_HOST,
    port: process.env.DATABASE_PORT || 5432
}));
```

As with custom configuration files, inside your [registerAs\(\)](#) factory function, the [process.env](#) object will contain the fully resolved environment variable key/value pairs (with [.env](#) file and externally defined variables resolved and merged as described [above](#)).

**info Hint** The [registerAs](#) function is exported from the [@nestjs/config](#) package.

Load a namespaced configuration with the [load](#) property of the [forRoot\(\)](#) method's options object, in the same way you load a custom configuration file:

```
import databaseConfig from './config/database.config';

@Module({
  imports: [
    ConfigModule.forRoot({
      load: [databaseConfig],
    }),
  ],
})
export class AppModule {}
```

Now, to get the [host](#) value from the [database](#) namespace, use dot notation. Use '['database'](#)' as the prefix to the property name, corresponding to the name of the namespace (passed as the first argument to the [registerAs\(\)](#) function):

```
const dbHost = this.configService.get<string>('database.host');
```

A reasonable alternative is to inject the `database` namespace directly. This allows us to benefit from strong typing:

```
constructor(
  @Inject(databaseConfig.KEY)
  private dbConfig: ConfigType<typeof databaseConfig>,
) {}
```

**info Hint** The `ConfigType` is exported from the `@nestjs/config` package.

## Cache environment variables

As accessing `process.env` can be slow, you can set the `cache` property of the options object passed to `ConfigModule.forRoot()` to increase the performance of `ConfigService#get` method when it comes to variables stored in `process.env`.

```
ConfigModule.forRoot({
  cache: true,
});
```

## Partial registration

Thus far, we've processed configuration files in our root module (e.g., `AppModule`), with the `forRoot()` method. Perhaps you have a more complex project structure, with feature-specific configuration files located in multiple different directories. Rather than load all these files in the root module, the `@nestjs/config` package provides a feature called **partial registration**, which references only the configuration files associated with each feature module. Use the `forFeature()` static method within a feature module to perform this partial registration, as follows:

```
import databaseConfig from './config/database.config';

@Module({
  imports: [ConfigModule.forFeature(databaseConfig)],
})
export class DatabaseModule {}
```

**info Warning** In some circumstances, you may need to access properties loaded via partial registration using the `onModuleInit()` hook, rather than in a constructor. This is because the `forFeature()` method is run during module initialization, and the order of module initialization is indeterminate. If you access values loaded this way by another module, in a constructor, the module

that the configuration depends upon may not yet have initialized. The `onModuleInit()` method runs only after all modules it depends upon have been initialized, so this technique is safe.

## Schema validation

It is standard practice to throw an exception during application startup if required environment variables haven't been provided or if they don't meet certain validation rules. The `@nestjs/config` package enables two different ways to do this:

- `Joi` built-in validator. With `Joi`, you define an object schema and validate JavaScript objects against it.
- A custom `validate()` function which takes environment variables as an input.

To use `Joi`, we must install `Joi` package:

```
$ npm install --save joi
```

Now we can define a `Joi` validation schema and pass it via the `validationSchema` property of the `forRoot()` method's options object, as shown below:

```
@@filename(app.module)
import * as Joi from 'joi';

@Module({
  imports: [
    ConfigModule.forRoot({
      validationSchema: Joi.object({
        NODE_ENV: Joi.string()
          .valid('development', 'production', 'test', 'provision')
          .default('development'),
        PORT: Joi.number().default(3000),
      }),
    }),
  ],
})
export class AppModule {}
```

By default, all schema keys are considered optional. Here, we set default values for `NODE_ENV` and `PORT` which will be used if we don't provide these variables in the environment (`.env` file or process environment). Alternatively, we can use the `required()` validation method to require that a value must be defined in the environment (`.env` file or process environment). In this case, the validation step will throw an exception if we don't provide the variable in the environment. See [Joi validation methods](#) for more on how to construct validation schemas.

By default, unknown environment variables (environment variables whose keys are not present in the schema) are allowed and do not trigger a validation exception. By default, all validation errors are reported. You can alter these behaviors by passing an options object via the `validationOptions` key of the `forRoot()` options object. This options object can contain any of the standard validation options

properties provided by [Joi validation options](#). For example, to reverse the two settings above, pass options like this:

```
@@filename(app.module)
import * as Joi from 'joi';

@Module({
  imports: [
    ConfigModule.forRoot({
      validationSchema: Joi.object({
        NODE_ENV: Joi.string()
          .valid('development', 'production', 'test', 'provision')
          .default('development'),
        PORT: Joi.number().default(3000),
      }),
      validationOptions: {
        allowUnknown: false,
        abortEarly: true,
      },
    }),
  ],
})
export class AppModule {}
```

The [@nestjs/config](#) package uses default settings of:

- **allowUnknown**: controls whether or not to allow unknown keys in the environment variables. Default is `true`
- **abortEarly**: if `true`, stops validation on the first error; if `false`, returns all errors. Defaults to `false`.

Note that once you decide to pass a `validationOptions` object, any settings you do not explicitly pass will default to `Joi` standard defaults (not the [@nestjs/config](#) defaults). For example, if you leave `allowUnknown` unspecified in your custom `validationOptions` object, it will have the `Joi` default value of `false`. Hence, it is probably safest to specify **both** of these settings in your custom object.

## Custom validate function

Alternatively, you can specify a **synchronous** `validate` function that takes an object containing the environment variables (from env file and process) and returns an object containing validated environment variables so that you can convert/mutate them if needed. If the function throws an error, it will prevent the application from bootstrapping.

In this example, we'll proceed with the `class-transformer` and `class-validator` packages. First, we have to define:

- a class with validation constraints,
- a validate function that makes use of the `plainToInstance` and `validateSync` functions.

```
@@filename(env.validation)
import { plainToInstance } from 'class-transformer';
import { IsEnum, IsNumber, validateSync } from 'class-validator';

enum Environment {
    Development = "development",
    Production = "production",
    Test = "test",
    Provision = "provision",
}

class EnvironmentVariables {
    @IsEnum(Environment)
    NODE_ENV: Environment;

    @IsNumber()
    PORT: number;
}

export function validate(config: Record<string, unknown>) {
    const validatedConfig = plainToInstance(
        EnvironmentVariables,
        config,
        { enableImplicitConversion: true },
    );
    const errors = validateSync(validatedConfig, { skipMissingProperties: false });

    if (errors.length > 0) {
        throw new Error(errors.toString());
    }
    return validatedConfig;
}
```

With this in place, use the `validate` function as a configuration option of the `ConfigModule`, as follows:

```
@@filename(app.module)
import { validate } from './env.validation';

@Module({
    imports: [
        ConfigModule.forRoot({
            validate,
        }),
    ],
})
export class AppModule {}
```

## Custom getter functions

ConfigService defines a generic `get()` method to retrieve a configuration value by key. We may also add `getter` functions to enable a little more natural coding style:

```
@@filename()
@Injectable()
export class ApiConfigService {
    constructor(private configService: ConfigService) {}

    get isAuthenticated(): boolean {
        return this.configService.get('AUTH_ENABLED') === 'true';
    }
}

@@switch
@Dependencies(ConfigService)
@Injectable()
export class ApiConfigService {
    constructor(configService) {
        this.configService = configService;
    }

    get isAuthenticated() {
        return this.configService.get('AUTH_ENABLED') === 'true';
    }
}
```

Now we can use the getter function as follows:

```
@@filename(app.service)
@Injectable()
export class AppService {
    constructor(apiConfigService: ApiConfigService) {
        if (apiConfigService.isAuthenticated) {
            // Authentication is enabled
        }
    }
}

@@switch
@Dependencies(ApiConfigService)
@Injectable()
export class AppService {
    constructor(apiConfigService) {
        if (apiConfigService.isAuthenticated) {
            // Authentication is enabled
        }
    }
}
```

## Environment variables loaded hook

If a module configuration depends on the environment variables, and these variables are loaded from the `.env` file, you can use the `ConfigModule.envVariablesLoaded` hook to ensure that the file was loaded before interacting with the `process.env` object, see the following example:

```
export async function getStorageModule() {
  await ConfigModule.envVariablesLoaded;
  return process.env.STORAGE === 'S3' ? S3StorageModule :
  DefaultStorageModule;
}
```

This construction guarantees that after the `ConfigModule.envVariablesLoaded` Promise resolves, all configuration variables are loaded up.

## Expandable variables

The `@nestjs/config` package supports environment variable expansion. With this technique, you can create nested environment variables, where one variable is referred to within the definition of another. For example:

```
APP_URL=mywebsite.com
SUPPORT_EMAIL=support@${APP_URL}
```

With this construction, the variable `SUPPORT_EMAIL` resolves to '`support@mywebsite.com`'. Note the use of the `${{ '{' }}...{{ '}' }}` syntax to trigger resolving the value of the variable `APP_URL` inside the definition of `SUPPORT_EMAIL`.

**info Hint** For this feature, `@nestjs/config` package internally uses `dotenv-expand`.

Enable environment variable expansion using the `expandVariables` property in the options object passed to the `forRoot()` method of the `ConfigModule`, as shown below:

```
@@filename(app.module)
@Module({
  imports: [
    ConfigModule.forRoot({
      // ...
      expandVariables: true,
    }),
  ],
})
export class AppModule {}
```

## Using in the `main.ts`

While our config is stored in a service, it can still be used in the `main.ts` file. This way, you can use it to store variables such as the application port or the CORS host.

To access it, you must use the `app.get()` method, followed by the service reference:

```
const configService = app.get(ConfigService);
```

You can then use it as usual, by calling the `get` method with the configuration key:

```
const port = configService.get('PORT');
```

## Database

Nest is database agnostic, allowing you to easily integrate with any SQL or NoSQL database. You have a number of options available to you, depending on your preferences. At the most general level, connecting Nest to a database is simply a matter of loading an appropriate Node.js driver for the database, just as you would with [Express](#) or Fastify.

You can also directly use any general purpose Node.js database integration **library** or ORM, such as [MikroORM](#) (see [MikroORM recipe](#)), [Sequelize](#) (see [Sequelize integration](#)), [Knex.js](#) (see [Knex.js tutorial](#)), [TypeORM](#), and [Prisma](#) (see [Prisma recipe](#)), to operate at a higher level of abstraction.

For convenience, Nest provides tight integration with TypeORM and Sequelize out-of-the-box with the [@nestjs/typeorm](#) and [@nestjs/sequelize](#) packages respectively, which we'll cover in the current chapter, and Mongoose with [@nestjs/mongoose](#), which is covered in [this chapter](#). These integrations provide additional NestJS-specific features, such as model/repository injection, testability, and asynchronous configuration to make accessing your chosen database even easier.

### TypeORM Integration

For integrating with SQL and NoSQL databases, Nest provides the [@nestjs/typeorm](#) package. [TypeORM](#) is the most mature Object Relational Mapper (ORM) available for TypeScript. Since it's written in TypeScript, it integrates well with the Nest framework.

To begin using it, we first install the required dependencies. In this chapter, we'll demonstrate using the popular [MySQL](#) Relational DBMS, but TypeORM provides support for many relational databases, such as PostgreSQL, Oracle, Microsoft SQL Server, SQLite, and even NoSQL databases like MongoDB. The procedure we walk through in this chapter will be the same for any database supported by TypeORM. You'll simply need to install the associated client API libraries for your selected database.

```
$ npm install --save @nestjs/typeorm typeorm mysql2
```

Once the installation process is complete, we can import the [TypeOrmModule](#) into the root [AppModule](#).

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      entities: [],
    })
  ],
  controllers: [HelloController],
  providers: [HelloService]
})
export class AppModule {}
```

```

        synchronize: true,
    },
],
})
export class AppModule {}

```

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

The `forRoot()` method supports all the configuration properties exposed by the `DataSource` constructor from the `TypeORM` package. In addition, there are several extra configuration properties described below.

<code>retryAttempts</code>	Number of attempts to connect to the database (default: <code>10</code> )
----------------------------	---

<code>retryDelay</code>	Delay between connection retry attempts (ms) (default: <code>3000</code> )
-------------------------	--

<code>autoLoadEntities</code>	If <code>true</code> , entities will be loaded automatically (default: <code>false</code> )
-------------------------------	---

info **Hint** Learn more about the data source options [here](#).

Once this is done, the `TypeORM` `DataSource` and `EntityManager` objects will be available to inject across the entire project (without needing to import any modules), for example:

```

@@filename(app.module)
import { DataSource } from 'typeorm';

@Module({
  imports: [TypeOrmModule.forRoot(), UsersModule],
})
export class AppModule {
  constructor(private dataSource: DataSource) {}
}

@@switch
import { DataSource } from 'typeorm';

@Dependencies(DataSource)
@Module({
  imports: [TypeOrmModule.forRoot(), UsersModule],
})
export class AppModule {
  constructor(dataSource) {
    this.dataSource = dataSource;
  }
}

```

## Repository pattern

`TypeORM` supports the **repository design pattern**, so each entity has its own repository. These repositories can be obtained from the database data source.

To continue the example, we need at least one entity. Let's define the `User` entity.

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

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

    @Column()
    firstName: string;

    @Column()
    lastName: string;

    @Column({ default: true })
    isActive: boolean;
}
```

**info Hint** Learn more about entities in the [TypeORM documentation](#).

The `User` entity file sits in the `users` directory. This directory contains all files related to the `UsersModule`. You can decide where to keep your model files, however, we recommend creating them near their `domain`, in the corresponding module directory.

To begin using the `User` entity, we need to let TypeORM know about it by inserting it into the `entities` array in the module `forRoot()` method options (unless you use a static glob path):

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './users/user.entity';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      entities: [User],
      synchronize: true,
    }),
  ],
})
export class AppModule {}
```

Next, let's look at the `UsersModule`:

```
@@filename(users.module)
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { User } from './user.entity';

@Module({
  imports: [TypeOrmModule.forFeature([User])],
  providers: [UsersService],
  controllers: [UsersController],
})
export class UsersModule {}
```

This module uses the `forFeature()` method to define which repositories are registered in the current scope. With that in place, we can inject the `UsersRepository` into the `UsersService` using the `@InjectRepository()` decorator:

```
@@filename(users.service)
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './user.entity';

@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(User)
    private usersRepository: Repository<User>,
  ) {}

  findAll(): Promise<User[]> {
    return this.usersRepository.find();
  }

  findOne(id: number): Promise<User | null> {
    return this.usersRepository.findOneBy({ id });
  }

  async remove(id: number): Promise<void> {
    await this.usersRepository.delete(id);
  }
}

@@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { getRepositoryToken } from '@nestjs/typeorm';
import { User } from './user.entity';
```

```

@Injectable()
@Dependencies(getRepositoryToken(User))
export class UsersService {
  constructor(usersRepository) {
    this.usersRepository = usersRepository;
  }

  findAll() {
    return this.usersRepository.find();
  }

  findOne(id) {
    return this.usersRepository.findOneBy({ id });
  }

  async remove(id) {
    await this.usersRepository.delete(id);
  }
}

```

**warning** **Notice** Don't forget to import the `UsersModule` into the root `AppModule`.

If you want to use the repository outside of the module which imports `TypeOrmModule.forFeature`, you'll need to re-export the providers generated by it. You can do this by exporting the whole module, like this:

```

@@filename(users.module)
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './user.entity';

@Module({
  imports: [TypeOrmModule.forFeature([User])],
  exports: [TypeOrmModule]
})
export class UsersModule {}

```

Now if we import `UsersModule` in `UserHttpModule`, we can use `@InjectRepository(User)` in the providers of the latter module.

```

@@filename(users-http.module)
import { Module } from '@nestjs/common';
import { UsersModule } from './users.module';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
  imports: [UsersModule],
  providers: [UsersService],
  controllers: [UsersController]
})
export class UserHttpModule {}

```

```
    controllers: [UsersController]
})
export class UserHttpModule {}
```

## Relations

Relations are associations established between two or more tables. Relations are based on common fields from each table, often involving primary and foreign keys.

There are three types of relations:

**One-to-one** Every row in the primary table has one and only one associated row in the foreign table. Use the `@OneToOne()` decorator to define this type of relation.

---

**One-to-many / Many-to-one** Every row in the primary table has one or more related rows in the foreign table. Use the `@OneToMany()` and `@ManyToOne()` decorators to define this type of relation.

---

**Many-to-many** Every row in the primary table has many related rows in the foreign table, and every record in the foreign table has many related rows in the primary table. Use the `@ManyToMany()` decorator to define this type of relation.

To define relations in entities, use the corresponding **decorators**. For example, to define that each `User` can have multiple photos, use the `@OneToMany()` decorator.

```
@@filename(user.entity)
import { Entity, Column, PrimaryGeneratedColumn, OneToMany } from
'typeorm';
import { Photo } from '../photos/photo.entity';

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

  @Column()
  firstName: string;

  @Column()
  lastName: string;

  @Column({ default: true })
  isActive: boolean;

  @OneToMany(type => Photo, photo => photo.user)
  photos: Photo[];
}
```

**info Hint** To learn more about relations in TypeORM, visit the [TypeORM documentation](#).

## Auto-load entities

Manually adding entities to the `entities` array of the data source 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 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:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [
    TypeOrmModule.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.

**warning Warning** 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.

## Separating entity definition

You can define an entity and its columns right in the model, using decorators. But some people prefer to define entities and their columns inside separate files using the "[entity schemas](#)".

```
import { EntitySchema } from 'typeorm';
import { User } from './user.entity';

export const UserSchema = new EntitySchema<User>({
  name: 'User',
  target: User,
  columns: {
    id: {
      type: Number,
      primary: true,
      generated: true,
    },
    firstName: {
```

```

    type: String,
},
lastName: {
  type: String,
},
isActive: {
  type: Boolean,
  default: true,
},
},
relations: {
  photos: {
    type: 'one-to-many',
    target: 'Photo', // the name of the PhotoSchema
  },
},
});

```

warning error **Warning** If you provide the `target` option, the `name` option value has to be the same as the name of the target class. If you do not provide the `target` you can use any name.

Nest allows you to use an `EntitySchema` instance wherever an `Entity` is expected, for example:

```

import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UserSchema } from './user.schema';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
  imports: [TypeOrmModule.forFeature([UserSchema])],
  providers: [UsersService],
  controllers: [UsersController],
})
export class UsersModule {}

```

## TypeORM Transactions

A database transaction symbolizes a unit of work performed within a database management system against a database, and treated in a coherent and reliable way independent of other transactions. A transaction generally represents any change in a database ([learn more](#)).

There are many different strategies to handle [TypeORM transactions](#). We recommend using the `QueryRunner` class because it gives full control over the transaction.

First, we need to inject the `DataSource` object into a class in the normal way:

```

@Injectable()
export class UsersService {

```

```
constructor(private dataSource: DataSource) {}  
}
```

**info Hint** The `DataSource` class is imported from the `typeorm` package.

Now, we can use this object to create a transaction.

```
async createMany(users: User[]) {  
    const queryRunner = this.dataSource.createQueryRunner();  
  
    await queryRunner.connect();  
    await queryRunner.startTransaction();  
    try {  
        await queryRunner.manager.save(users[0]);  
        await queryRunner.manager.save(users[1]);  
  
        await queryRunner.commitTransaction();  
    } catch (err) {  
        // since we have errors lets rollback the changes we made  
        await queryRunner.rollbackTransaction();  
    } finally {  
        // you need to release a queryRunner which was manually instantiated  
        await queryRunner.release();  
    }  
}
```

**info Hint** Note that the `dataSource` is used only to create the `QueryRunner`. However, to test this class would require mocking the entire `DataSource` object (which exposes several methods). Thus, we recommend using a helper factory class (e.g., `QueryRunnerFactory`) and defining an interface with a limited set of methods required to maintain transactions. This technique makes mocking these methods pretty straightforward.

Alternatively, you can use the callback-style approach with the `transaction` method of the `DataSource` object ([read more](#)).

```
async createMany(users: User[]) {  
    await this.dataSource.transaction(async manager => {  
        await manager.save(users[0]);  
        await manager.save(users[1]);  
    });  
}
```

## Subscribers

With TypeORM [subscribers](#), you can listen to specific entity events.

```
import {  
  DataSource,  
  EntitySubscriberInterface,  
  EventSubscriber,  
  InsertEvent,  
} from 'typeorm';  
import { User } from './user.entity';  
  
@EventSubscriber()  
export class UserSubscriber implements EntitySubscriberInterface<User> {  
  constructor(dataSource: DataSource) {  
    dataSource.subscribers.push(this);  
  }  
  
  listenTo() {  
    return User;  
  }  
  
  beforeInsert(event: InsertEvent<User>) {  
    console.log(`BEFORE USER INSERTED: `, event.entity);  
  }  
}
```

**error Warning** Event subscribers can not be [request-scoped](#).

Now, add the `UserSubscriber` class to the `providers` array:

```
import { Module } from '@nestjs/common';  
import { TypeOrmModule } from '@nestjs/typeorm';  
import { User } from './user.entity';  
import { UsersController } from './users.controller';  
import { UsersService } from './users.service';  
import { UserSubscriber } from './user.subscriber';  
  
@Module({  
  imports: [TypeOrmModule.forFeature([User])],  
  providers: [UsersService, UserSubscriber],  
  controllers: [UsersController],  
})  
export class UsersModule {}
```

**info Hint** Learn more about entity subscribers [here](#).

## Migrations

[Migrations](#) provide a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database. To generate, run, and revert migrations, TypeORM provides a dedicated [CLI](#).

Migration classes are separate from the Nest application source code. Their lifecycle is maintained by the TypeORM CLI. Therefore, you are not able to leverage dependency injection and other Nest specific features with migrations. To learn more about migrations, follow the guide in the [TypeORM documentation](#).

## Multiple databases

Some projects require multiple database connections. This can also be achieved with this module. To work with multiple connections, first create the connections. In this case, data source naming becomes **mandatory**.

Suppose you have an `Album` entity stored in its own database.

```
const defaultOptions = {
  type: 'postgres',
  port: 5432,
  username: 'user',
  password: 'password',
  database: 'db',
  synchronize: true,
};

@Module({
  imports: [
    TypeOrmModule.forRoot({
      ...defaultOptions,
      host: 'user_db_host',
      entities: [User],
    }),
    TypeOrmModule.forRoot({
      ...defaultOptions,
      name: 'albumsConnection',
      host: 'album_db_host',
      entities: [Album],
    }),
  ],
})
export class AppModule {}
```

**warning** **Notice** If you don't set the `name` for a data source, its name is set to `default`. Please note that you shouldn't have multiple connections without a name, or with the same name, otherwise they will get overridden.

**warning** **Notice** If you are using `TypeOrmModule.forRootAsync`, you have to **also** set the data source name outside `useFactory`. For example:

```
TypeOrmModule.forRootAsync({
  name: 'albumsConnection',
  useFactory: ...,
```

```
    inject: ...,
}),
```

See [this issue](#) for more details.

At this point, you have `User` and `Album` entities registered with their own data source. With this setup, you have to tell the `TypeOrmModule.forFeature()` method and the `@InjectRepository()` decorator which data source should be used. If you do not pass any data source name, the `default` data source is used.

```
@Module({
  imports: [
    TypeOrmModule.forFeature([User]),
    TypeOrmModule.forFeature([Album], 'albumsConnection'),
  ],
})
export class AppModule {}
```

You can also inject the `DataSource` or `EntityManager` for a given data source:

```
@Injectable()
export class AlbumsService {
  constructor(
    @InjectDataSource('albumsConnection')
    private dataSource: DataSource,
    @InjectEntityManager('albumsConnection')
    private entityManager: EntityManager,
  ) {}
}
```

It's also possible to inject any `DataSource` to the providers:

```
@Module({
  providers: [
    {
      provide: AlbumsService,
      useFactory: (albumsConnection: DataSource) => {
        return new AlbumsService(albumsConnection);
      },
      inject: [getDataSourceToken('albumsConnection')],
    },
  ],
})
export class AlbumsModule {}
```

## Testing

When it comes to unit testing an application, we usually want to avoid making a database connection, keeping our test suites independent and their execution process as fast as possible. But our classes might depend on repositories that are pulled from the data source (connection) instance. How do we handle that? The solution is to create mock repositories. In order to achieve that, we set up [custom providers](#). Each registered repository is automatically represented by an `<EntityName>Repository` token, where `EntityName` is the name of your entity class.

The `@nestjs/typeorm` package exposes the `getRepositoryToken()` function which returns a prepared token based on a given entity.

```
@Module({
  providers: [
    UsersService,
    {
      provide: getRepositoryToken(User),
      useValue: mockRepository,
    },
  ],
})
export class UsersModule {}
```

Now a substitute `mockRepository` will be used as the `UsersRepository`. Whenever any class asks for `UsersRepository` using an `@InjectRepository()` decorator, Nest will use the registered `mockRepository` object.

## Async configuration

You may want to pass your repository module options asynchronously instead of statically. In this case, use the `forRootAsync()` method, which provides several ways to deal with async configuration.

One approach is to use a factory function:

```
TypeOrmModule.forRootAsync({
  useFactory: () => ({
    type: 'mysql',
    host: 'localhost',
    port: 3306,
    username: 'root',
    password: 'root',
    database: 'test',
    entities: [],
    synchronize: true,
  }),
});
```

Our factory behaves like any other [asynchronous provider](#) (e.g., it can be `async` and it's able to inject dependencies through `inject`).

```
TypeOrmModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: (configService: ConfigService) => ({
    type: 'mysql',
    host: configService.get('HOST'),
    port: +configService.get('PORT'),
    username: configService.get('USERNAME'),
    password: configService.get('PASSWORD'),
    database: configService.get('DATABASE'),
    entities: [],
    synchronize: true,
  }),
  inject: [ConfigService],
});
```

Alternatively, you can use the `useClass` syntax:

```
TypeOrmModule.forRoot({
  useClass: TypeOrmConfigService,
});
```

The construction above will instantiate `TypeOrmConfigService` inside `TypeOrmModule` and use it to provide an options object by calling `createTypeOrmOptions()`. Note that this means that the `TypeOrmConfigService` has to implement the `TypeOrmOptionsFactory` interface, as shown below:

```
@Injectable()
export class TypeOrmConfigService implements TypeOrmOptionsFactory {
  createTypeOrmOptions(): TypeOrmModuleOptions {
    return {
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      entities: [],
      synchronize: true,
    };
  }
}
```

In order to prevent the creation of `TypeOrmConfigService` inside `TypeOrmModule` and use a provider imported from a different module, you can use the `useExisting` syntax.

```
TypeOrmModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

This construction works the same as `useClass` with one critical difference - `TypeOrmModule` will lookup imported modules to reuse an existing `ConfigService` instead of instantiating a new one.

**info Hint** Make sure that the `name` property is defined at the same level as the `useFactory`, `useClass`, or `useValue` property. This will allow Nest to properly register the data source under the appropriate injection token.

## Custom DataSource Factory

In conjunction with async configuration using `useFactory`, `useClass`, or `useExisting`, you can optionally specify a `dataSourceFactory` function which will allow you to provide your own TypeORM data source rather than allowing `TypeOrmModule` to create the data source.

`dataSourceFactory` receives the TypeORM `DataSourceOptions` configured during async configuration using `useFactory`, `useClass`, or `useExisting` and returns a `Promise` that resolves a TypeORM `DataSource`.

```
TypeOrmModule.forRootAsync({
  imports: [ConfigModule],
  inject: [ConfigService],
  // Use useFactory, useClass, or useExisting
  // to configure the DataSourceOptions.
  useFactory: (configService: ConfigService) => ({
    type: 'mysql',
    host: configService.get('HOST'),
    port: +configService.get('PORT'),
    username: configService.get('USERNAME'),
    password: configService.get('PASSWORD'),
    database: configService.get('DATABASE'),
    entities: [],
    synchronize: true,
  }),
  // dataSource receives the configured DataSourceOptions
  // and returns a Promise<DataSource>.
  dataSourceFactory: async (options) => {
    const dataSource = await new DataSource(options).initialize();
    return dataSource;
  },
});
```

**info Hint** The `DataSource` class is imported from the `typeorm` package.

## Example

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

## Sequelize Integration

An alternative to using TypeORM is to use the [Sequelize](#) ORM with the [@nestjs/sequelize](#) package. In addition, we leverage the [sequelize-typescript](#) package which provides a set of additional decorators to declaratively define entities.

To begin using it, we first install the required dependencies. In this chapter, we'll demonstrate using the popular [MySQL](#) Relational DBMS, but Sequelize provides support for many relational databases, such as PostgreSQL, MySQL, Microsoft SQL Server, SQLite, and MariaDB. The procedure we walk through in this chapter will be the same for any database supported by Sequelize. You'll simply need to install the associated client API libraries for your selected database.

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

Once the installation process is complete, we can import the [SequelizeModule](#) into the root [AppModule](#).

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';

@Module({
  imports: [
    SequelizeModule.forRoot({
      dialect: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      models: [],
    }),
  ],
})
export class AppModule {}
```

The [forRoot\(\)](#) method supports all the configuration properties exposed by the Sequelize constructor ([read more](#)). In addition, there are several extra configuration properties described below.

<a href="#">retryAttempts</a>	Number of attempts to connect to the database (default: <a href="#">10</a> )
<a href="#">retryDelay</a>	Delay between connection retry attempts (ms) (default: <a href="#">3000</a> )
<a href="#">autoLoadModels</a>	If <a href="#">true</a> , models will be loaded automatically (default: <a href="#">false</a> )
<a href="#">keepConnectionAlive</a>	If <a href="#">true</a> , connection will not be closed on the application shutdown (default:

```
false)
```

---

synchronize	If <b>true</b> , automatically loaded models will be synchronized (default: <b>true</b> )
-------------	---

Once this is done, the **Sequelize** object will be available to inject across the entire project (without needing to import any modules), for example:

```
@@filename(app.service)
import { Injectable } from '@nestjs/common';
import { Sequelize } from 'sequelize-typescript';

@Injectable()
export class AppService {
  constructor(private sequelize: Sequelize) {}
}

@switch
import { Injectable } from '@nestjs/common';
import { Sequelize } from 'sequelize-typescript';

@Dependencies(Sequelize)
@Injectable()
export class AppService {
  constructor(sequelize) {
    this.sequelize = sequelize;
  }
}
```

## Models

Sequelize implements the Active Record pattern. With this pattern, you use model classes directly to interact with the database. To continue the example, we need at least one model. Let's define the **User** model.

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

@Table
export class User extends Model {
  @Column
  firstName: string;

  @Column
  lastName: string;

  @Column({ defaultValue: true })
  isActive: boolean;
}
```

**info Hint** Learn more about the available decorators [here](#).

The `User` model file sits in the `users` directory. This directory contains all files related to the `UsersModule`. You can decide where to keep your model files, however, we recommend creating them near their `domain`, in the corresponding module directory.

To begin using the `User` model, we need to let Sequelize know about it by inserting it into the `models` array in the module `forRoot()` method options:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';
import { User } from './users/user.model';

@Module({
  imports: [
    SequelizeModule.forRoot({
      dialect: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      models: [User],
    }),
  ],
})
export class AppModule {}
```

Next, let's look at the `UsersModule`:

```
@@filename(users.module)
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';
import { User } from './user.model';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
  imports: [SequelizeModule.forFeature([User])],
  providers: [UsersService],
  controllers: [UsersController],
})
export class UsersModule {}
```

This module uses the `forFeature()` method to define which models are registered in the current scope. With that in place, we can inject the `UserModel` into the `UsersService` using the `@InjectModel()` decorator:

```
@@filename(users.service)
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/sequelize';
import { User } from './user.model';

@Injectable()
export class UsersService {
  constructor(
    @InjectModel(User)
    private userModel: typeof User,
  ) {}

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

  findOne(id: string): Promise<User> {
    return this.userModel.findOne({
      where: {
        id,
      },
    });
  }

  async remove(id: string): Promise<void> {
    const user = await this.findOne(id);
    await user.destroy();
  }
}

@@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { getModelToken } from '@nestjs/sequelize';
import { User } from './user.model';

@Injectable()
@Dependencies(getModelToken(User))
export class UsersService {
  constructor(usersRepository) {
    this.usersRepository = usersRepository;
  }

  async findAll() {
    return this.userModel.findAll();
  }

  findOne(id) {
    return this.userModel.findOne({
      where: {
        id,
      },
    });
  }
}
```

```

async remove(id) {
  const user = await this.findOne(id);
  await user.destroy();
}
}

```

**warning** **Notice** Don't forget to import the `UsersModule` into the root `AppModule`.

If you want to use the repository outside of the module which imports `SequelizeModule.forFeature`, you'll need to re-export the providers generated by it. You can do this by exporting the whole module, like this:

```

@@filename(users.module)
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';
import { User } from './user.entity';

@Module({
  imports: [SequelizeModule.forFeature([User])],
  exports: [SequelizeModule]
})
export class UsersModule {}

```

Now if we import `UsersModule` in `UserHttpModule`, we can use `@InjectModel(User)` in the providers of the latter module.

```

@@filename(users-http.module)
import { Module } from '@nestjs/common';
import { UsersModule } from './users.module';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
  imports: [UsersModule],
  providers: [UsersService],
  controllers: [UsersController]
})
export class UserHttpModule {}

```

## Relations

Relations are associations established between two or more tables. Relations are based on common fields from each table, often involving primary and foreign keys.

There are three types of relations:

**One-to-one**

Every row in the primary table has one and only one associated row in the foreign

## table

---

One-to-many	
/ Many-to-one	Every row in the primary table has one or more related rows in the foreign table
Many-to-many	Every row in the primary table has many related rows in the foreign table, and every record in the foreign table has many related rows in the primary table

---

To define relations in models, use the corresponding **decorators**. For example, to define that each **User** can have multiple photos, use the `@HasMany()` decorator.

```
@@filename(user.model)
import { Column, Model, Table, HasMany } from 'sequelize-typescript';
import { Photo } from '../photos/photo.model';

@Table
export class User extends Model {
  @Column
  firstName: string;

  @Column
  lastName: string;

  @Column({ defaultValue: true })
  isActive: boolean;

  @HasMany(() => Photo)
  photos: Photo[];
}
```

**info Hint** To learn more about associations in Sequelize, read [this chapter](#).

## Auto-load models

Manually adding models to the `models` array of the connection options can be tedious. In addition, referencing models from the root module breaks application domain boundaries and causes leaking implementation details to other parts of the application. To solve this issue, automatically load models by setting both `autoLoadModels` and `synchronize` properties of the configuration object (passed into the `forRoot()` method) to `true`, as shown below:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { SequelizeModule } from '@nestjs/sequelize';

@Module({
  imports: [
    SequelizeModule.forRoot({
      ...
    })
  ]
})
```

```

    autoLoadModels: true,
    synchronize: true,
  },
],
})
export class AppModule {}

```

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

**warning Warning** Note that models that aren't registered through the `forFeature()` method, but are only referenced from the model (via an association), won't be included.

## Sequelize Transactions

A database transaction symbolizes a unit of work performed within a database management system against a database, and treated in a coherent and reliable way independent of other transactions. A transaction generally represents any change in a database ([learn more](#)).

There are many different strategies to handle [Sequelize transactions](#). Below is a sample implementation of a managed transaction (auto-callback).

First, we need to inject the `Sequelize` object into a class in the normal way:

```

@Injectable()
export class UsersService {
  constructor(private sequelize: Sequelize) {}
}

```

**info Hint** The `Sequelize` class is imported from the `sequelize-typescript` package.

Now, we can use this object to create a transaction.

```

async createMany() {
  try {
    await this.sequelize.transaction(async t => {
      const transactionHost = { transaction: t };

      await this.userModel.create(
        { firstName: 'Abraham', lastName: 'Lincoln' },
        transactionHost,
      );
      await this.userModel.create(
        { firstName: 'John', lastName: 'Boothe' },
        transactionHost,
      );
    });
  } catch (err) {
    // Transaction has been rolled back
  }
}

```

```
// err is whatever rejected the promise chain returned to the
transaction callback
}
}
```

**info Hint** Note that the `Sequelize` instance is used only to start the transaction. However, to test this class would require mocking the entire `Sequelize` object (which exposes several methods). Thus, we recommend using a helper factory class (e.g., `TransactionRunner`) and defining an interface with a limited set of methods required to maintain transactions. This technique makes mocking these methods pretty straightforward.

## Migrations

[Migrations](#) provide a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database. To generate, run, and revert migrations, Sequelize provides a dedicated [CLI](#).

Migration classes are separate from the Nest application source code. Their lifecycle is maintained by the Sequelize CLI. Therefore, you are not able to leverage dependency injection and other Nest specific features with migrations. To learn more about migrations, follow the guide in the [Sequelize documentation](#).

## Multiple databases

Some projects require multiple database connections. This can also be achieved with this module. To work with multiple connections, first create the connections. In this case, connection naming becomes **mandatory**.

Suppose you have an `Album` entity stored in its own database.

```
const defaultOptions = {
  dialect: 'postgres',
  port: 5432,
  username: 'user',
  password: 'password',
  database: 'db',
  synchronize: true,
};

@Module({
  imports: [
    SequelizeModule.forRoot({
      ...defaultOptions,
      host: 'user_db_host',
      models: [User],
    }),
    SequelizeModule.forRoot({
      ...defaultOptions,
      name: 'albumsConnection',
      host: 'album_db_host',
      models: [Album],
    })
  ]
})
```

```

        },
    ],
})
export class AppModule {}

```

warning **Notice** If you don't set the `name` for a connection, its name is set to `default`. Please note that you shouldn't have multiple connections without a name, or with the same name, otherwise they will get overridden.

At this point, you have `User` and `Album` models registered with their own connection. With this setup, you have to tell the `SequelizeModule.forFeature()` method and the `@InjectModel()` decorator which connection should be used. If you do not pass any connection name, the `default` connection is used.

```

@Module({
  imports: [
    SequelizeModule.forFeature([User]),
    SequelizeModule.forFeature([Album], 'albumsConnection'),
  ],
})
export class AppModule {}

```

You can also inject the `Sequelize` instance for a given connection:

```

@Injectable()
export class AlbumsService {
  constructor(
    @InjectConnection('albumsConnection')
    private sequelize: Sequelize,
  ) {}
}

```

It's also possible to inject any `Sequelize` instance to the providers:

```

@Module({
  providers: [
    {
      provide: AlbumsService,
      useFactory: (albumsSequelize: Sequelize) => {
        return new AlbumsService(albumsSequelize);
      },
      inject: [getDataSourceToken('albumsConnection')],
    },
  ],
})
export class AlbumsModule {}

```

## Testing

When it comes to unit testing an application, we usually want to avoid making a database connection, keeping our test suites independent and their execution process as fast as possible. But our classes might depend on models that are pulled from the connection instance. How do we handle that? The solution is to create mock models. In order to achieve that, we set up [custom providers](#). Each registered model is automatically represented by a `<ModelName>Model` token, where `ModelName` is the name of your model class.

The `@nestjs/sequelize` package exposes the `getModelToken()` function which returns a prepared token based on a given model.

```
@Module({
  providers: [
    UsersService,
    {
      provide: getModelToken(User),
      useValue: mockModel,
    },
  ],
})
export class UsersModule {}
```

Now a substitute `mockModel` will be used as the `UserModel`. Whenever any class asks for `UserModel` using an `@InjectModel()` decorator, Nest will use the registered `mockModel` object.

## Async configuration

You may want to pass your `SequelizeModule` options asynchronously instead of statically. In this case, use the `forRootAsync()` method, which provides several ways to deal with async configuration.

One approach is to use a factory function:

```
SequelizeModule.forRootAsync({
  useFactory: () => ({
    dialect: 'mysql',
    host: 'localhost',
    port: 3306,
    username: 'root',
    password: 'root',
    database: 'test',
    models: [],
  }),
});
```

Our factory behaves like any other [asynchronous provider](#) (e.g., it can be `async` and it's able to inject dependencies through `inject`).

```
SequelizeModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: (configService: ConfigService) => ({
    dialect: 'mysql',
    host: configService.get('HOST'),
    port: +configService.get('PORT'),
    username: configService.get('USERNAME'),
    password: configService.get('PASSWORD'),
    database: configService.get('DATABASE'),
    models: [],
  }),
  inject: [ConfigService],
});
```

Alternatively, you can use the `useClass` syntax:

```
SequelizeModule.forRootAsync({
  useClass: SequelizeConfigService,
});
```

The construction above will instantiate `SequelizeConfigService` inside `SequelizeModule` and use it to provide an options object by calling `createSequelizeOptions()`. Note that this means that the `SequelizeConfigService` has to implement the `SequelizeOptionsFactory` interface, as shown below:

```
@Injectable()
class SequelizeConfigService implements SequelizeOptionsFactory {
  createSequelizeOptions(): SequelizeModuleOptions {
    return {
      dialect: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      models: [],
    };
  }
}
```

In order to prevent the creation of `SequelizeConfigService` inside `SequelizeModule` and use a provider imported from a different module, you can use the `useExisting` syntax.

```
SequelizeModule.forRootAsync({
  imports: [ConfigModule],
```

```
useExisting: ConfigService,  
});
```

This construction works the same as `useClass` with one critical difference - `SequelizerModule` will lookup imported modules to reuse an existing `ConfigService` instead of instantiating a new one.

## Example

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

## Mongo

Nest supports two methods for integrating with the [MongoDB](#) database. You can either use the built-in [TypeORM](#) module described [here](#), which has a connector for MongoDB, or use [Mongoose](#), the most popular MongoDB object modeling tool. In this chapter we'll describe the latter, using the dedicated [@nestjs/mongoose](#) package.

Start by installing the [required dependencies](#):

```
$ npm i @nestjs/mongoose mongoose
```

Once the installation process is complete, we can import the [MongooseModule](#) into the root [AppModule](#).

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [MongooseModule.forRoot('mongodb://localhost/nest')],
})
export class AppModule {}
```

The [forRoot\(\)](#) method accepts the same configuration object as [mongoose.connect\(\)](#) from the Mongoose package, as described [here](#).

## Model injection

With Mongoose, everything is derived from a [Schema](#). Each schema maps to a MongoDB collection and defines the shape of the documents within that collection. Schemas are used to define [Models](#). Models are responsible for creating and reading documents from the underlying MongoDB database.

Schemas can be created with NestJS decorators, or with Mongoose itself manually. Using decorators to create schemas greatly reduces boilerplate and improves overall code readability.

Let's define the [CatSchema](#):

```
@@filename(schemas/cat.schema)
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { HydratedDocument } from 'mongoose';

export type CatDocument = HydratedDocument<Cat>

@Schema()
export class Cat {
  @Prop()
  name: string;
```

```
@Prop()  
age: number;  
  
@Prop()  
breed: string;  
}  
  
export const CatSchema = SchemaFactory.createForClass(Cat);
```

**info Hint** Note you can also generate a raw schema definition using the [DefinitionsFactory](#) class (from the [nestjs/mongoose](#)). This allows you to manually modify the schema definition generated based on the metadata you provided. This is useful for certain edge-cases where it may be hard to represent everything with decorators.

The [@Schema\(\)](#) decorator marks a class as a schema definition. It maps our [Cat](#) class to a MongoDB collection of the same name, but with an additional "s" at the end - so the final mongo collection name will be [cats](#). This decorator accepts a single optional argument which is a schema options object. Think of it as the object you would normally pass as a second argument of the [mongoose.Schema](#) class' constructor (e.g., `new mongoose.Schema(_, options)`). To learn more about available schema options, see [this chapter](#).

The [@Prop\(\)](#) decorator defines a property in the document. For example, in the schema definition above, we defined three properties: [name](#), [age](#), and [breed](#). The [schema types](#) for these properties are automatically inferred thanks to TypeScript metadata (and reflection) capabilities. However, in more complex scenarios in which types cannot be implicitly reflected (for example, arrays or nested object structures), types must be indicated explicitly, as follows:

```
@Prop([String])  
tags: string[];
```

Alternatively, the [@Prop\(\)](#) decorator accepts an options object argument ([read more](#) about the available options). With this, you can indicate whether a property is required or not, specify a default value, or mark it as immutable. For example:

```
@Prop({ required: true })  
name: string;
```

In case you want to specify relation to another model, later for populating, you can use [@Prop\(\)](#) decorator as well. For example, if [Cat](#) has [Owner](#) which is stored in a different collection called [owners](#), the property should have type and ref. For example:

```
import * as mongoose from 'mongoose';  
import { Owner } from '../owners/schemas/owner.schema';
```

```
// inside the class definition
@Prop({ type: mongoose.Schema.Types.ObjectId, ref: 'Owner' })
owner: Owner;
```

In case there are multiple owners, your property configuration should look as follows:

```
@Prop({ type: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Owner' }] })
owner: Owner[];
```

Finally, the **raw** schema definition can also be passed to the decorator. This is useful when, for example, a property represents a nested object which is not defined as a class. For this, use the **raw()** function from the [@nestjs/mongoose](#) package, as follows:

```
@Prop(raw({
  firstName: { type: String },
  lastName: { type: String }
}))
details: Record<string, any>;
```

Alternatively, if you prefer **not using decorators**, you can define a schema manually. For example:

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

The [cat.schema](#) file resides in a folder in the [cats](#) directory, where we also define the [CatsModule](#). While you can store schema files wherever you prefer, we recommend storing them near their related **domain** objects, in the appropriate module directory.

Let's look at the [CatsModule](#):

```
@@filename(cats.module)
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';
import { Cat, CatSchema } from './schemas/cat.schema';

@Module({
  imports: [MongooseModule.forFeature([{ name: Cat.name, schema: CatSchema }])],
  controllers: [CatsController],
  providers: [CatsService],
```

```
})
export class CatsModule {}
```

The `MongooseModule` provides the `forFeature()` method to configure the module, including defining which models should be registered in the current scope. If you also want to use the models in another module, add `MongooseModule` to the `exports` section of `CatsModule` and import `CatsModule` in the other module.

Once you've registered the schema, you can inject a `Cat` model into the `CatsService` using the `@InjectModel()` decorator:

```
@@filename(cats.service)
import { Model } from 'mongoose';
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Cat } from './schemas/cat.schema';
import { CreateCatDto } from './dto/create-cat.dto';

@Injectable()
export class CatsService {
  constructor(@InjectModel(Cat.name) 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 { Model } from 'mongoose';
import { Injectable, Dependencies } from '@nestjs/common';
import { getModelToken } from '@nestjs/mongoose';
import { Cat } from './schemas/cat.schema';

@Injectable()
@Dependencies(getModelToken(Cat.name))
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();
  }
}
```

```
    }
}
```

## Connection

At times you may need to access the native [Mongoose Connection](#) object. For example, you may want to make native API calls on the connection object. You can inject the Mongoose Connection by using the `@InjectConnection()` decorator as follows:

```
import { Injectable } from '@nestjs/common';
import { InjectConnection } from '@nestjs/mongoose';
import { Connection } from 'mongoose';

@Injectable()
export class CatsService {
  constructor(@InjectConnection() private connection: Connection) {}
}
```

## Multiple databases

Some projects require multiple database connections. This can also be achieved with this module. To work with multiple connections, first create the connections. In this case, connection naming becomes **mandatory**.

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [
    MongooseModule.forRoot('mongodb://localhost/test', {
      connectionName: 'cats',
    }),
    MongooseModule.forRoot('mongodb://localhost/users', {
      connectionName: 'users',
    }),
  ],
})
export class AppModule {}
```

**warning** **Notice** Please note that you shouldn't have multiple connections without a name, or with the same name, otherwise they will get overridden.

With this setup, you have to tell the `MongooseModule.forFeature()` function which connection should be used.

```
@Module({
  imports: [
    MongooseModule.forFeature([{ name: Cat.name, schema: CatSchema }]),
    'cats'),
  ],
})
export class CatsModule {}
```

You can also inject the `Connection` for a given connection:

```
import { Injectable } from '@nestjs/common';
import { InjectConnection } from '@nestjs/mongoose';
import { Connection } from 'mongoose';

@Injectable()
export class CatsService {
  constructor(@InjectConnection('cats') private connection: Connection) {}
}
```

To inject a given `Connection` to a custom provider (for example, factory provider), use the `getConnectionToken()` function passing the name of the connection as an argument.

```
{
  provide: CatsService,
  useFactory: (catsConnection: Connection) => {
    return new CatsService(catsConnection);
  },
  inject: [getConnectionToken('cats')],
}
```

If you are just looking to inject the model from a named database, you can use the connection name as a second parameter to the `@InjectModel()` decorator.

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(@InjectModel(Cat.name, 'cats') private catModel: Model<Cat>)
{}}
@@switch
@Injectable()
@Dependencies(getModelToken(Cat.name, 'cats'))
export class CatsService {
  constructor(catModel) {
    this.catModel = catModel;
```

```
    }
}
```

## Hooks (middleware)

Middleware (also called pre and post hooks) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins ([source](#)). Calling `pre()` or `post()` after compiling a model does not work in Mongoose. To register a hook **before** model registration, use the `forFeatureAsync()` method of the `MongooseModule` along with a factory provider (i.e., `useFactory`). With this technique, you can access a schema object, then use the `pre()` or `post()` method to register a hook on that schema. See example below:

```
@Module({
  imports: [
    MongooseModule.forFeatureAsync([
      {
        name: Cat.name,
        useFactory: () => {
          const schema = CatsSchema;
          schema.pre('save', function () {
            console.log('Hello from pre save');
          });
          return schema;
        },
      ],
    ]),
  ],
})
export class AppModule {}
```

Like other [factory providers](#), our factory function can be `async` and can inject dependencies through `inject`.

```
@Module({
  imports: [
    MongooseModule.forFeatureAsync([
      {
        name: Cat.name,
        imports: [ConfigModule],
        useFactory: (configService: ConfigService) => {
          const schema = CatsSchema;
          schema.pre('save', function() {
            console.log(
              `${configService.get('APP_NAME')}: Hello from pre save`,
            );
          });
          return schema;
        },
      ],
    ]),
  ],
})
export class AppModule {}
```

```

        inject: [ConfigService],
    },
],
],
})
export class AppModule {}

```

## Plugins

To register a [plugin](#) for a given schema, use the `forFeatureAsync()` method.

```

@Module({
  imports: [
    MongooseModule.forFeatureAsync([
      {
        name: Cat.name,
        useFactory: () => {
          const schema = CatsSchema;
          schema.plugin(require('mongoose-autopopulate'));
          return schema;
        },
      ],
    ]),
  ],
})
export class AppModule {}

```

To register a plugin for all schemas at once, call the `.plugin()` method of the `Connection` object. You should access the connection before models are created; to do this, use the `connectionFactory`:

```

@@filename(app.module)
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [
    MongooseModule.forRoot('mongodb://localhost/test', {
      connectionFactory: (connection) => {
        connection.plugin(require('mongoose-autopopulate'));
        return connection;
      }
    }),
  ],
})
export class AppModule {}

```

## Discriminators

**Discriminators** are a schema inheritance mechanism. They enable you to have multiple models with overlapping schemas on top of the same underlying MongoDB collection.

Suppose you wanted to track different types of events in a single collection. Every event will have a timestamp.

```
@@filename(event.schema)
@Schema({ discriminatorKey: 'kind' })
export class Event {
  @Prop({
    type: String,
    required: true,
    enum: [ClickedLinkEvent.name, SignUpEvent.name],
  })
  kind: string;

  @Prop({ type: Date, required: true })
  time: Date;
}

export const EventSchema = SchemaFactory.createForClass(Event);
```

**info Hint** The way mongoose tells the difference between the different discriminator models is by the "discriminator key", which is `__t` by default. Mongoose adds a String path called `__t` to your schemas that it uses to track which discriminator this document is an instance of. You may also use the `discriminatorKey` option to define the path for discrimination.

`SignedUpEvent` and `ClickedLinkEvent` instances will be stored in the same collection as generic events.

Now, let's define the `ClickedLinkEvent` class, as follows:

```
@@filename(click-link-event.schema)
@Schema()
export class ClickedLinkEvent {
  kind: string;
  time: Date;

  @Prop({ type: String, required: true })
  url: string;
}

export const ClickedLinkEventSchema =
  SchemaFactory.createForClass(ClickedLinkEvent);
```

And `SignUpEvent` class:

```

@@filename(sign-up-event.schema)
@Schema()
export class SignUpEvent {
  kind: string;
  time: Date;

  @Prop({ type: String, required: true })
  user: string;
}

export const SignUpEventSchema =
SchemaFactory.createForClass(SignUpEvent);

```

With this in place, use the `discriminators` option to register a discriminator for a given schema. It works on both `MongooseModule.forFeature` and `MongooseModule.forFeatureAsync`:

```

@@filename(event.module)
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [
    MongooseModule.forFeature([
      {
        name: Event.name,
        schema: EventSchema,
        discriminators: [
          { name: ClickedLinkEvent.name, schema: ClickedLinkEventSchema },
          { name: SignUpEvent.name, schema: SignUpEventSchema },
        ],
      },
    ]),
  ],
})
export class EventsModule {}

```

## Testing

When unit testing an application, we usually want to avoid any database connection, making our test suites simpler to set up and faster to execute. But our classes might depend on models that are pulled from the connection instance. How do we resolve these classes? The solution is to create mock models.

To make this easier, the `@nestjs/mongoose` package exposes a `getModelToken()` function that returns a prepared `injection token` based on a token name. Using this token, you can easily provide a mock implementation using any of the standard `custom provider` techniques, including `useClass`, `useValue`, and `useFactory`. For example:

```
@Module({
  providers: [
    CatsService,
    {
      provide: getModelToken(Cat.name),
      useValue: catModel,
    },
  ],
})
export class CatsModule {}
```

In this example, a hardcoded `catModel` (object instance) will be provided whenever any consumer injects a `Model<Cat>` using an `@InjectModel()` decorator.

## Async configuration

When you need to pass module options asynchronously instead of statically, use the `forRootAsync()` method. As with most dynamic modules, Nest provides several techniques to deal with async configuration.

One technique is to use a factory function:

```
MongooseModule.forRootAsync({
  useFactory: () => ({
    uri: 'mongodb://localhost/nest',
  }),
});
```

Like other [factory providers](#), our factory function can be `async` and can inject dependencies through `inject`.

```
MongooseModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    uri: configService.get<string>('MONGODB_URI'),
  }),
  inject: [ConfigService],
});
```

Alternatively, you can configure the `MongooseModule` using a class instead of a factory, as shown below:

```
MongooseModule.forRootAsync({
  useClass: MongooseConfigService,
});
```

The construction above instantiates `MongooseConfigService` inside `MongooseModule`, using it to create the required options object. Note that in this example, the `MongooseConfigService` has to implement the `MongooseOptionsFactory` interface, as shown below. The `MongooseModule` will call the `createMongooseOptions()` method on the instantiated object of the supplied class.

```
@Injectable()
export class MongooseConfigService implements MongooseOptionsFactory {
  createMongooseOptions(): MongooseModuleOptions {
    return {
      uri: 'mongodb://localhost/nest',
    };
  }
}
```

If you want to reuse an existing options provider instead of creating a private copy inside the `MongooseModule`, use the `useExisting` syntax.

```
MongooseModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

## Example

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

## Validation

It is best practice to validate the correctness of any data sent into a web application. To automatically validate incoming requests, Nest provides several pipes available right out-of-the-box:

- `ValidationPipe`
- `ParseIntPipe`
- `ParseBoolPipe`
- `ParseArrayPipe`
- `ParseUUIDPipe`

The `ValidationPipe` makes use of the powerful [class-validator](#) package and its declarative validation decorators. The `ValidationPipe` provides a convenient approach to enforce validation rules for all incoming client payloads, where the specific rules are declared with simple annotations in local class/DTO declarations in each module.

## Overview

In the [Pipes](#) chapter, we went through the process of building simple pipes and binding them to controllers, methods or to the global app to demonstrate how the process works. Be sure to review that chapter to best understand the topics of this chapter. Here, we'll focus on various **real world** use cases of the `ValidationPipe`, and show how to use some of its advanced customization features.

### Using the built-in `ValidationPipe`

To begin using it, we first install the required dependency.

```
$ npm i --save class-validator class-transformer
```

**info Hint** The `ValidationPipe` is exported from the [@nestjs/common](#) package.

Because this pipe uses the `class-validator` and `class-transformer` libraries, there are many options available. You configure these settings via a configuration object passed to the pipe. Following are the built-in options:

```
export interface ValidationPipeOptions extends ValidatorOptions {  
  transform?: boolean;  
  disableErrorMessages?: boolean;  
  exceptionFactory?: (errors: ValidationErrors[]) => any;  
}
```

In addition to these, all `class-validator` options (inherited from the `ValidatorOptions` interface) are available:

Option	Type	Description
--------	------	-------------

<code>enableDebugMessages</code>	<code>boolean</code>	If set to true, validator will print extra warning messages to the console when something is not right.
<code>skipUndefinedProperties</code>	<code>boolean</code>	If set to true then validator will skip validation of all properties that are undefined in the validating object.
<code>skipNullProperties</code>	<code>boolean</code>	If set to true then validator will skip validation of all properties that are null in the validating object.
<code>skipMissingProperties</code>	<code>boolean</code>	If set to true then validator will skip validation of all properties that are null or undefined in the validating object.
<code>whitelist</code>	<code>boolean</code>	If set to true, validator will strip validated (returned) object of any properties that do not use any validation decorators.
<code>forbidNonWhitelisted</code>	<code>boolean</code>	If set to true, instead of stripping non-whitelisted properties validator will throw an exception.
<code>forbidUnknownValues</code>	<code>boolean</code>	If set to true, attempts to validate unknown objects fail immediately.
<code>disableErrorMessages</code>	<code>boolean</code>	If set to true, validation errors will not be returned to the client.
<code>errorHttpStatusCode</code>	<code>number</code>	This setting allows you to specify which exception type will be used in case of an error. By default it throws <code>BadRequestException</code> .
<code>exceptionFactory</code>	<code>Function</code>	Takes an array of the validation errors and returns an exception object to be thrown.
<code>groups</code>	<code>string[]</code>	Groups to be used during validation of the object.
<code>always</code>	<code>boolean</code>	Set default for <code>always</code> option of decorators. Default can be overridden in decorator options

`strictGroups boolean` If `groups` is not given or is empty, ignore decorators with at least one group.  
`dismissDefaultMessages boolean` If set to true, the validation will not use default messages. Error message always will be `undefined` if its not explicitly set. `validationError.target boolean` Indicates if target should be exposed in `ValidationError`. `validationError.value boolean` Indicates if validated value should be exposed in `ValidationError`. `validationError.stopAtFirstError boolean` When set to true, validation of the given property will stop after encountering the first error. Defaults to false.

**info** **Notice** Find more information about the `class-validator` package in its [repository](#).

## Auto-validation

We'll start by binding `ValidationPipe` at the application level, thus ensuring all endpoints are protected from receiving incorrect data.

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}
bootstrap();
```

To test our pipe, let's create a basic endpoint.

```
@Post()
create(@Body() createUserDto: CreateUserDto) {
  return 'This action adds a new user';
}
```

**info Hint** Since TypeScript does not store metadata about **generics or interfaces**, when you use them in your DTOs, **ValidationPipe** may not be able to properly validate incoming data. For this reason, consider using concrete classes in your DTOs.

**info Hint** When importing your DTOs, you can't use a type-only import as that would be erased at runtime, i.e. remember to `import {{ '{' }} CreateUserDto {{ '}' }}` instead of `import type {{ '{' }} CreateUserDto {{ '}' }}`.

Now we can add a few validation rules in our `CreateUserDto`. We do this using decorators provided by the `class-validator` package, described in detail [here](#). In this fashion, any route that uses the `CreateUserDto` will automatically enforce these validation rules.

```
import { IsEmail, IsNotEmpty } from 'class-validator';

export class CreateUserDto {
  @IsEmail()
  email: string;

  @IsNotEmpty()
  password: string;
}
```

With these rules in place, if a request hits our endpoint with an invalid `email` property in the request body, the application will automatically respond with a `400 Bad Request` code, along with the following response body:

```
{
  "statusCode": 400,
  "error": "Bad Request",
  "message": ["email must be an email"]
}
```

In addition to validating request bodies, the `ValidationPipe` can be used with other request object properties as well. Imagine that we would like to accept `:id` in the endpoint path. To ensure that only numbers are accepted for this request parameter, we can use the following construct:

```
@Get(':id')
findOne(@Param() params: FindOneParams) {
  return 'This action returns a user';
}
```

`FindOneParams`, like a DTO, is simply a class that defines validation rules using `class-validator`. It would look like this:

```
import { IsNumberString } from 'class-validator';

export class FindOneParams {
  @IsNumberString()
  id: number;
}
```

## Disable detailed errors

Error messages can be helpful to explain what was incorrect in a request. However, some production environments prefer to disable detailed errors. Do this by passing an options object to the `ValidationPipe`:

```
app.useGlobalPipes(
  new ValidationPipe({
    disableErrorMessages: true,
  }),
);
```

As a result, detailed error messages won't be displayed in the response body.

## Stripping properties

Our `ValidationPipe` can also filter out properties that should not be received by the method handler. In this case, we can **whitelist** the acceptable properties, and any property not included in the whitelist is automatically stripped from the resulting object. For example, if our handler expects `email` and `password` properties, but a request also includes an `age` property, this property can be automatically removed from the resulting DTO. To enable such behavior, set `whitelist` to `true`.

```
app.useGlobalPipes(
  new ValidationPipe({
```

```
        whitelist: true,
    }),
);
```

When set to true, this will automatically remove non-whitelisted properties (those without any decorator in the validation class).

Alternatively, you can stop the request from processing when non-whitelisted properties are present, and return an error response to the user. To enable this, set the `forbidNonWhitelisted` option property to `true`, in combination with setting `whitelist` to `true`.

## Transform payload objects

Payloads coming in over the network are plain JavaScript objects. The `ValidationPipe` can automatically transform payloads to be objects typed according to their DTO classes. To enable auto-transformation, set `transform` to `true`. This can be done at a method level:

```
@@filename(cats.controller)
@Post()
@UsePipes(new ValidationPipe({ transform: true }))
async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
}
```

To enable this behavior globally, set the option on a global pipe:

```
app.useGlobalPipes(
    new ValidationPipe({
        transform: true,
    }),
);
```

With the auto-transformation option enabled, the `ValidationPipe` will also perform conversion of primitive types. In the following example, the `findOne()` method takes one argument which represents an extracted `id` path parameter:

```
@Get(':id')
findOne(@Param('id') id: number) {
    console.log(typeof id === 'number'); // true
    return 'This action returns a user';
}
```

By default, every path parameter and query parameter comes over the network as a `string`. In the above example, we specified the `id` type as a `number` (in the method signature). Therefore, the

**ValidationPipe** will try to automatically convert a string identifier to a number.

## Explicit conversion

In the above section, we showed how the **ValidationPipe** can implicitly transform query and path parameters based on the expected type. However, this feature requires having auto-transformation enabled.

Alternatively (with auto-transformation disabled), you can explicitly cast values using the **ParseIntPipe** or **ParseBoolPipe** (note that **ParseStringPipe** is not needed because, as mentioned earlier, every path parameter and query parameter comes over the network as a **string** by default).

```
@Get(':id')
findOne(
  @Param('id', ParseIntPipe) id: number,
  @Query('sort', ParseBoolPipe) sort: boolean,
) {
  console.log(typeof id === 'number'); // true
  console.log(typeof sort === 'boolean'); // true
  return 'This action returns a user';
}
```

**info Hint** The **ParseIntPipe** and **ParseBoolPipe** are exported from the [@nestjs/common](#) package.

## Mapped types

As you build out features like **CRUD** (Create/Read/Update/Delete) it's often useful to construct variants on a base entity type. Nest provides several utility functions that perform type transformations to make this task more convenient.

**Warning** If your application uses the [@nestjs/swagger](#) package, see [this chapter](#) for more information about Mapped Types. Likewise, if you use the [@nestjs/graphql](#) package see [this chapter](#). Both packages heavily rely on types and so they require a different import to be used. Therefore, if you used [@nestjs/mapped-types](#) (instead of an appropriate one, either [@nestjs/swagger](#) or [@nestjs/graphql](#) depending on the type of your app), you may face various, undocumented side-effects.

When building input validation types (also called DTOs), it's often useful to build **create** and **update** variations on the same type. For example, the **create** variant may require all fields, while the **update** variant may make all fields optional.

Nest provides the **PartialType()** utility function to make this task easier and minimize boilerplate.

The **PartialType()** function returns a type (class) with all the properties of the input type set to optional. For example, suppose we have a **create** type as follows:

```
export class CreateCatDto {  
    name: string;  
    age: number;  
    breed: string;  
}
```

By default, all of these fields are required. To create a type with the same fields, but with each one optional, use `PartialType()` passing the class reference (`CreateCatDto`) as an argument:

```
export class UpdateCatDto extends PartialType(CreateCatDto) {}
```

**info Hint** The `PartialType()` function is imported from the `@nestjs/mapped-types` package.

The `PickType()` function constructs a new type (class) by picking a set of properties from an input type. For example, suppose we start with a type like:

```
export class CreateCatDto {  
    name: string;  
    age: number;  
    breed: string;  
}
```

We can pick a set of properties from this class using the `PickType()` utility function:

```
export class UpdateCatAgeDto extends PickType(CreateCatDto, ['age'] as const) {}
```

**info Hint** The `PickType()` function is imported from the `@nestjs/mapped-types` package.

The `OmitType()` function constructs a type by picking all properties from an input type and then removing a particular set of keys. For example, suppose we start with a type like:

```
export class CreateCatDto {  
    name: string;  
    age: number;  
    breed: string;  
}
```

We can generate a derived type that has every property **except** `name` as shown below. In this construct, the second argument to `OmitType` is an array of property names.

```
export class UpdateCatDto extends OmitType(CreateCatDto, ['name'] as const) {}
```

**info Hint** The `OmitType()` function is imported from the `@nestjs/mapped-types` package.

The `IntersectionType()` function combines two types into one new type (class). For example, suppose we start with two types like:

```
export class CreateCatDto {  
  name: string;  
  breed: string;  
}  
  
export class AdditionalCatInfo {  
  color: string;  
}
```

We can generate a new type that combines all properties in both types.

```
export class UpdateCatDto extends IntersectionType(  
  CreateCatDto,  
  AdditionalCatInfo,  
) {}
```

**info Hint** The `IntersectionType()` function is imported from the `@nestjs/mapped-types` package.

The type mapping utility functions are composable. For example, the following will produce a type (class) that has all of the properties of the `CreateCatDto` type except for `name`, and those properties will be set to optional:

```
export class UpdateCatDto extends PartialType(  
  OmitType(CreateCatDto, ['name'] as const),  
) {}
```

## Parsing and validating arrays

TypeScript does not store metadata about generics or interfaces, so when you use them in your DTOs, `ValidationPipe` may not be able to properly validate incoming data. For instance, in the following code, `createUserDtos` won't be correctly validated:

```
@Post()  
createBulk(@Body() createUserDtos: CreateUserDto[]) {
```

```
    return 'This action adds new users';
}
```

To validate the array, create a dedicated class which contains a property that wraps the array, or use the [ParseArrayPipe](#).

```
@Post()
createBulk(
  @Body(new ParseArrayPipe({ items: CreateUserDto }))  
  createUserDtos: CreateUserDto[],
) {
  return 'This action adds new users';
}
```

In addition, the [ParseArrayPipe](#) may come in handy when parsing query parameters. Let's consider a [findByIds\(\)](#) method that returns users based on identifiers passed as query parameters.

```
@Get()
findByIds(
  @Query('ids', new ParseArrayPipe({ items: Number, separator: ',' }))  
  ids: number[],
) {
  return 'This action returns users by ids';
}
```

This construction validates the incoming query parameters from an HTTP [GET](#) request like the following:

```
GET /?ids=1,2,3
```

## WebSockets and Microservices

While this chapter shows examples using HTTP style applications (e.g., Express or Fastify), the [ValidationPipe](#) works the same for WebSockets and microservices, regardless of the transport method that is used.

### Learn more

Read more about custom validators, error messages, and available decorators as provided by the [class-validator](#) package [here](#).

## Caching

Caching is a great and simple **technique** that helps improve your app's performance. It acts as a temporary data store providing high performance data access.

### Installation

First install required packages:

```
$ npm install @nestjs/cache-manager cache-manager
```

warning **Warning** `cache-manager` version 4 uses seconds for `TTL (Time-To-Live)`. The current version of `cache-manager` (v5) has switched to using milliseconds instead. NestJS doesn't convert the value, and simply forwards the ttl you provide to the library. In other words:

- If using `cache-manager` v4, provide ttl in seconds
- If using `cache-manager` v5, provide ttl in milliseconds
- Documentation is referring to seconds, since NestJS was released targeting version 4 of `cache-manager`.

### In-memory cache

Nest provides a unified API for various cache storage providers. The built-in one is an in-memory data store. However, you can easily switch to a more comprehensive solution, like Redis.

In order to enable caching, import the `CacheModule` and call its `register()` method.

```
import { Module } from '@nestjs/common';
import { CacheModule } from '@nestjs/cache-manager';
import { AppController } from './app.controller';

@Module({
  imports: [CacheModule.register()],
  controllers: [AppController],
})
export class AppModule {}
```

### Interacting with the Cache store

To interact with the cache manager instance, inject it to your class using the `CACHE_MANAGER` token, as follows:

```
constructor(@Inject(CACHE_MANAGER) private cacheManager: Cache) {}
```

**info Hint** The `Cache` class is imported from the `cache-manager`, while `CACHE_MANAGER` token from the `@nestjs/cache-manager` package.

The `get` method on the `Cache` instance (from the `cache-manager` package) is used to retrieve items from the cache. If the item does not exist in the cache, `null` will be returned.

```
const value = await this.cacheManager.get('key');
```

To add an item to the cache, use the `set` method:

```
await this.cacheManager.set('key', 'value');
```

The default expiration time of the cache is 5 seconds.

You can manually specify a TTL (expiration time in seconds) for this specific key, as follows:

```
await this.cacheManager.set('key', 'value', 1000);
```

To disable expiration of the cache, set the `ttl` configuration property to `0`:

```
await this.cacheManager.set('key', 'value', 0);
```

To remove an item from the cache, use the `del` method:

```
await this.cacheManager.del('key');
```

To clear the entire cache, use the `reset` method:

```
await this.cacheManager.reset();
```

## Auto-caching responses

**warning Warning** In `GraphQL` applications, interceptors are executed separately for each field resolver. Thus, `CacheModule` (which uses interceptors to cache responses) will not work properly.

To enable auto-caching responses, just tie the `CacheInterceptor` where you want to cache data.

```
@Controller()  
@UseInterceptors(CacheInterceptor)
```

```
export class AppController {  
  @Get()  
  findAll(): string[] {  
    return [];  
  }  
}
```

**warning** Only `GET` endpoints are cached. Also, HTTP server routes that inject the native response object (`@Res()`) cannot use the Cache Interceptor. See [response mapping](#) for more details.

To reduce the amount of required boilerplate, you can bind `CacheInterceptor` to all endpoints globally:

```
import { Module } from '@nestjs/common';  
import { CacheModule, CacheInterceptor } from '@nestjs/cache-manager';  
import { AppController } from './app.controller';  
import { APP_INTERCEPTOR } from '@nestjs/core';  
  
@Module({  
  imports: [CacheModule.register()],  
  controllers: [AppController],  
  providers: [  
    {  
      provide: APP_INTERCEPTOR,  
      useClass: CacheInterceptor,  
    },  
  ],  
})  
export class AppModule {}
```

## Customize caching

All cached data has its own expiration time ([TTL](#)). To customize default values, pass the options object to the `register()` method.

```
CacheModule.register({  
  ttl: 5, // seconds  
  max: 10, // maximum number of items in cache  
});
```

## Use module globally

When you want to use `CacheModule` in other modules, you'll need to import it (as is standard with any Nest module). Alternatively, declare it as a [global module](#) by setting the options object's `isGlobal` property to `true`, as shown below. In that case, you will not need to import `CacheModule` in other modules once it's been loaded in the root module (e.g., `AppModule`).

```
CacheModule.register({
  isGlobal: true,
});
```

## Global cache overrides

While global cache is enabled, cache entries are stored under a [CacheKey](#) that is auto-generated based on the route path. You may override certain cache settings ([@CacheKey\(\)](#) and [@CacheTTL\(\)](#)) on a per-method basis, allowing customized caching strategies for individual controller methods. This may be most relevant while using [different cache stores](#).

```
@Controller()
export class AppController {
  @CacheKey('custom_key')
  @CacheTTL(20)
  findAll(): string[] {
    return [];
  }
}
```

**info Hint** The [@CacheKey\(\)](#) and [@CacheTTL\(\)](#) decorators are imported from the [@nestjs/cache-manager](#) package.

The [@CacheKey\(\)](#) decorator may be used with or without a corresponding [@CacheTTL\(\)](#) decorator and vice versa. One may choose to override only the [@CacheKey\(\)](#) or only the [@CacheTTL\(\)](#). Settings that are not overridden with a decorator will use the default values as registered globally (see [Customize caching](#)).

## WebSockets and Microservices

You can also apply the [CacheInterceptor](#) to WebSocket subscribers as well as Microservice's patterns (regardless of the transport method that is being used).

```
@@filename()
@CacheKey('events')
@UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client: Client, data: string[]): Observable<string[]> {
  return [];
}
@@switch
@CacheKey('events')
@UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client, data) {
  return [];
}
```

However, the additional `@CacheKey()` decorator is required in order to specify a key used to subsequently store and retrieve cached data. Also, please note that you **shouldn't cache everything**. Actions which perform some business operations rather than simply querying the data should never be cached.

Additionally, you may specify a cache expiration time (TTL) by using the `@CacheTTL()` decorator, which will override the global default TTL value.

```
@@filename()
@CacheTTL(10)
@UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client: Client, data: string[]): Observable<string[]> {
  return [];
}
@switch
@CacheTTL(10)
@UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client, data) {
  return [];
}
```

**info Hint** The `@CacheTTL()` decorator may be used with or without a corresponding `@CacheKey()` decorator.

## Adjust tracking

By default, Nest uses the request URL (in an HTTP app) or cache key (in websockets and microservices apps, set through the `@CacheKey()` decorator) to associate cache records with your endpoints. Nevertheless, sometimes you might want to set up tracking based on different factors, for example, using HTTP headers (e.g. `Authorization` to properly identify `profile` endpoints).

In order to accomplish that, create a subclass of `CacheInterceptor` and override the `trackBy()` method.

```
@Injectable()
class HttpCacheInterceptor extends CacheInterceptor {
  trackBy(context: ExecutionContext): string | undefined {
    return 'key';
}
```

## Different stores

This service takes advantage of `cache-manager` under the hood. The `cache-manager` package supports a wide-range of useful stores, for example, `Redis store`. A full list of supported stores is available [here](#). To set

up the Redis store, simply pass the package together with corresponding options to the `register()` method.

```
import type { RedisClientOptions } from 'redis';
import * as redisStore from 'cache-manager-redis-store';
import { Module } from '@nestjs/common';
import { CacheModule } from '@nestjs/cache-manager';
import { AppController } from './app.controller';

@Module({
  imports: [
    CacheModule.register<RedisClientOptions>({
      store: redisStore,

      // Store-specific configuration:
      host: 'localhost',
      port: 6379,
    }),
  ],
  controllers: [AppController],
})
export class AppModule {}
```

**warning** `Warning` `cache-manager-redis-store` does not support redis v4. In order for the `ClientOpts` interface to exist and work correctly you need to install the latest `redis` 3.x.x major release. See this [issue](#) to track the progress of this upgrade.

## Async configuration

You may want to asynchronously pass in module options instead of passing them statically at compile time. In this case, use the `registerAsync()` method, which provides several ways to deal with async configuration.

One approach is to use a factory function:

```
CacheModule.registerAsync({
  useFactory: () => ({
    ttl: 5,
  }),
});
```

Our factory behaves like all other asynchronous module factories (it can be `async` and is able to inject dependencies through `inject`).

```
CacheModule.registerAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
```

```
    ttl: configService.get('CACHE_TTL'),
}),
inject: [ConfigService],
});
```

Alternatively, you can use the `useClass` method:

```
CacheModule.registerAsync({
  useClass: CacheConfigService,
});
```

The above construction will instantiate `CacheConfigService` inside `CacheModule` and will use it to get the options object. The `CacheConfigService` has to implement the `CacheOptionsFactory` interface in order to provide the configuration options:

```
@Injectable()
class CacheConfigService implements CacheOptionsFactory {
  createCacheOptions(): CacheModuleOptions {
    return {
      ttl: 5,
    };
  }
}
```

If you wish to use an existing configuration provider imported from a different module, use the `useExisting` syntax:

```
CacheModule.registerAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

This works the same as `useClass` with one critical difference - `CacheModule` will lookup imported modules to reuse any already-created `ConfigService`, instead of instantiating its own.

**info Hint** `CacheModule#register` and `CacheModule#registerAsync` and `CacheOptionsFactory` has an optional generic (type argument) to narrow down store-specific configuration options, making it type safe.

## Example

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

## Serialization

Serialization is a process that happens before objects are returned in a network response. This is an appropriate place to provide rules for transforming and sanitizing the data to be returned to the client. For example, sensitive data like passwords should always be excluded from the response. Or, certain properties might require additional transformation, such as sending only a subset of properties of an entity. Performing these transformations manually can be tedious and error prone, and can leave you uncertain that all cases have been covered.

### Overview

Nest provides a built-in capability to help ensure that these operations can be performed in a straightforward way. The [ClassSerializerInterceptor](#) interceptor uses the powerful [class-transformer](#) package to provide a declarative and extensible way of transforming objects. The basic operation it performs is to take the value returned by a method handler and apply the [instanceToPlain\(\)](#) function from [class-transformer](#). In doing so, it can apply rules expressed by [class-transformer](#) decorators on an entity/DTO class, as described below.

**info Hint** The serialization does not apply to [StreamableFile](#) responses.

### Exclude properties

Let's assume that we want to automatically exclude a [password](#) property from a user entity. We annotate the entity as follows:

```
import { Exclude } from 'class-transformer';

export class UserEntity {
  id: number;
  firstName: string;
  lastName: string;

  @Exclude()
  password: string;

  constructor(partial: Partial<UserEntity>) {
    Object.assign(this, partial);
  }
}
```

Now consider a controller with a method handler that returns an instance of this class.

```
@UseInterceptors(ClassSerializerInterceptor)
@Get()
findOne(): UserEntity {
  return new UserEntity({
    id: 1,
```

```
    firstName: 'Kamil',
    lastName: 'Mysliwiec',
    password: 'password',
  );
}
```

**Warning** Note that we must return an instance of the class. If you return a plain JavaScript object, for example, `{} '{' } user: new UserEntity() {} '}' {}`, the object won't be properly serialized.

**info Hint** The `ClassSerializerInterceptor` is imported from `@nestjs/common`.

When this endpoint is requested, the client receives the following response:

```
{
  "id": 1,
  "firstName": "Kamil",
  "lastName": "Mysliwiec"
}
```

Note that the interceptor can be applied application-wide (as covered [here](#)). The combination of the interceptor and the entity class declaration ensures that **any** method that returns a `UserEntity` will be sure to remove the `password` property. This gives you a measure of centralized enforcement of this business rule.

## Expose properties

You can use the `@Expose()` decorator to provide alias names for properties, or to execute a function to calculate a property value (analogous to `getter` functions), as shown below.

```
@Expose()
get fullName(): string {
  return `${this.firstName} ${this.lastName}`;
}
```

## Transform

You can perform additional data transformation using the `@Transform()` decorator. For example, the following construct returns the `name` property of the `RoleEntity` instead of returning the whole object.

```
@Transform(({ value }) => value.name)
role: RoleEntity;
```

## Pass options

You may want to modify the default behavior of the transformation functions. To override default settings, pass them in an `options` object with the `@SerializeOptions()` decorator.

```
@SerializeOptions({
  excludePrefixes: ['_'],
})
@Get()
findOne(): UserEntity {
  return new UserEntity();
}
```

**info Hint** The `@SerializeOptions()` decorator is imported from `@nestjs/common`.

Options passed via `@SerializeOptions()` are passed as the second argument of the underlying `instanceToPlain()` function. In this example, we are automatically excluding all properties that begin with the `_` prefix.

## Example

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

## WebSockets and Microservices

While this chapter shows examples using HTTP style applications (e.g., Express or Fastify), the `ClassSerializerInterceptor` works the same for WebSockets and Microservices, regardless of the transport method that is used.

## Learn more

Read more about available decorators and options as provided by the `class-transformer` package [here](#).

## Versioning

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

Versioning allows you to have **different versions** of your controllers or individual routes running within the same application. Applications change very often and it is not unusual that there are breaking changes that you need to make while still needing to support the previous version of the application.

There are 4 types of versioning that are supported:

<b>URI Versioning</b>	The version will be passed within the URI of the request (default)
<b>Header Versioning</b>	A custom request header will specify the version
<b>Media Type Versioning</b>	The <code>Accept</code> header of the request will specify the version
<b>Custom Versioning</b>	Any aspect of the request may be used to specify the version(s). A custom function is provided to extract said version(s).

### URI Versioning Type

URI Versioning uses the version passed within the URI of the request, such as

`https://example.com/v1/route` and `https://example.com/v2/route`.

**warning Notice** With URI Versioning the version will be automatically added to the URI after the `global path prefix` (if one exists), and before any controller or route paths.

To enable URI Versioning for your application, do the following:

```
@@filename(main)
const app = await NestFactory.create(AppModule);
// or "app.enableVersioning()"
app.enableVersioning({
  type: VersioningType.URI,
});
await app.listen(3000);
```

**warning Notice** The version in the URI will be automatically prefixed with `v` by default, however the prefix value can be configured by setting the `prefix` key to your desired prefix or `false` if you wish to disable it.

**info Hint** The `VersioningType` enum is available to use for the `type` property and is imported from the `@nestjs/common` package.

### Header Versioning Type

Header Versioning uses a custom, user specified, request header to specify the version where the value of the header will be the version to use for the request.

Example HTTP Requests for Header Versioning:

To enable **Header Versioning** for your application, do the following:

```
@@filename(main)
const app = await NestFactory.create(AppModule);
app.enableVersioning({
  type: VersioningType.HEADER,
  header: 'Custom-Header',
});
await app.listen(3000);
```

The **header** property should be the name of the header that will contain the version of the request.

**info Hint** The **VersioningType** enum is available to use for the **type** property and is imported from the **@nestjs/common** package.

## Media Type Versioning Type

Media Type Versioning uses the **Accept** header of the request to specify the version.

Within the **Accept** header, the version will be separated from the media type with a semi-colon, **;**. It should then contain a key-value pair that represents the version to use for the request, such as **Accept: application/json;v=2**. The key is treated more as a prefix when determining the version will to be configured to include the key and separator.

To enable **Media Type Versioning** for your application, do the following:

```
@@filename(main)
const app = await NestFactory.create(AppModule);
app.enableVersioning({
  type: VersioningType.MEDIA_TYPE,
  key: 'v=',
});
await app.listen(3000);
```

The **key** property should be the key and separator of the key-value pair that contains the version. For the example **Accept: application/json;v=2**, the **key** property would be set to **v=**.

**info Hint** The **VersioningType** enum is available to use for the **type** property and is imported from the **@nestjs/common** package.

## Custom Versioning Type

Custom Versioning uses any aspect of the request to specify the version (or versions). The incoming request is analyzed using an **extractor** function that returns a string or array of strings.

If multiple versions are provided by the requester, the extractor function can return an array of strings, sorted in order of greatest/highest version to smallest/lowest version. Versions are matched to routes in order from highest to lowest.

If an empty string or array is returned from the **extractor**, no routes are matched and a 404 is returned.

For example, if an incoming request specifies it supports versions 1, 2, and 3, the **extractor** **MUST** return [3, 2, 1]. This ensures that the highest possible route version is selected first.

If versions [3, 2, 1] are extracted, but routes only exist for version 2 and 1, the route that matches version 2 is selected (version 3 is automatically ignored).

**warning** **Notice** Selecting the highest matching version based on the array returned from **extractor** > **does not reliably work** with the Express adapter due to design limitations. A single version (either a string or array of 1 element) works just fine in Express. Fastify correctly supports both highest matching version selection and single version selection.

To enable **Custom Versioning** for your application, create an **extractor** function and pass it into your application like so:

```
@@filename(main)
// Example extractor that pulls out a list of versions from a custom
header and turns it into a sorted array.
// This example uses Fastify, but Express requests can be processed in a
similar way.
const extractor = (request: FastifyRequest): string | string[] =>
  [request.headers['custom-versioning-field'] ?? '']
    .flatMap(v => v.split(','))
    .filter(v => !v)
    .sort()
    .reverse()

const app = await NestFactory.create(AppModule);
app.enableVersioning({
  type: VersioningType.CUSTOM,
  extractor,
});
await app.listen(3000);
```

## Usage

Versioning allows you to version controllers, individual routes, and also provides a way for certain resources to opt-out of versioning. The usage of versioning is the same regardless of the Versioning Type your application uses.

**warning** **Notice** If versioning is enabled for the application but the controller or route does not specify the version, any requests to that controller/route will be returned a 404 response status.

Similarly, if a request is received containing a version that does not have a corresponding controller or route, it will also be returned a **404** response status.

## Controller versions

A version can be applied to a controller, setting the version for all routes within the controller.

To add a version to a controller do the following:

```
@@filename(cats.controller)
@Controller({
  version: '1',
})
export class CatsControllerV1 {
  @Get('cats')
  findAll(): string {
    return 'This action returns all cats for version 1';
  }
}
@@switch
@Controller({
  version: '1',
})
export class CatsControllerV1 {
  @Get('cats')
  findAll() {
    return 'This action returns all cats for version 1';
  }
}
```

## Route versions

A version can be applied to an individual route. This version will override any other version that would effect the route, such as the Controller Version.

To add a version to an individual route do the following:

```
@@filename(cats.controller)
import { Controller, Get, Version } from '@nestjs/common';

@Controller()
export class CatsController {
  @Version('1')
  @Get('cats')
  findAllV1(): string {
    return 'This action returns all cats for version 1';
  }

  @Version('2')
  @Get('cats')
```

```
findAllV2(): string {
    return 'This action returns all cats for version 2';
}
}
@@switch
import { Controller, Get, Version } from '@nestjs/common';

@Controller()
export class CatsController {
    @Version('1')
    @Get('cats')
    findAllV1() {
        return 'This action returns all cats for version 1';
    }

    @Version('2')
    @Get('cats')
    findAllV2() {
        return 'This action returns all cats for version 2';
    }
}
```

## Multiple versions

Multiple versions can be applied to a controller or route. To use multiple versions, you would set the version to be an Array.

To add multiple versions do the following:

```
@@filename(cats.controller)
@Controller({
    version: ['1', '2'],
})
export class CatsController {
    @Get('cats')
    findAll(): string {
        return 'This action returns all cats for version 1 or 2';
    }
}
@@switch
@Controller({
    version: ['1', '2'],
})
export class CatsController {
    @Get('cats')
    findAll() {
        return 'This action returns all cats for version 1 or 2';
    }
}
```

## Version "Neutral"

Some controllers or routes may not care about the version and would have the same functionality regardless of the version. To accommodate this, the version can be set to `VERSION_NEUTRAL` symbol.

An incoming request will be mapped to a `VERSION_NEUTRAL` controller or route regardless of the version sent in the request in addition to if the request does not contain a version at all.

**warning** **Notice** For URI Versioning, a `VERSION_NEUTRAL` resource would not have the version present in the URI.

To add a version neutral controller or route do the following:

```
@@filename(cats.controller)
import { Controller, Get, VERSION_NEUTRAL } from '@nestjs/common';

@Controller({
  version: VERSION_NEUTRAL,
})
export class CatsController {
  @Get('cats')
  findAll(): string {
    return 'This action returns all cats regardless of version';
  }
}
@switch
import { Controller, Get, VERSION_NEUTRAL } from '@nestjs/common';

@Controller({
  version: VERSION_NEUTRAL,
})
export class CatsController {
  @Get('cats')
  findAll() {
    return 'This action returns all cats regardless of version';
  }
}
```

## Global default version

If you do not want to provide a version for each controller/or individual routes, or if you want to have a specific version set as the default version for every controller/route that don't have the version specified, you could set the `defaultVersion` as follows:

```
@@filename(main)
app.enableVersioning({
  // ...
  defaultVersion: '1'
  // or
```

```
defaultVersion: ['1', '2']
// or
defaultVersion: VERSION_NEUTRAL
});
```

## Middleware versioning

Middlewares can also use versioning metadata to configure the middleware for a specific route's version. To do so, provide the version number as one of the parameters for the `MiddlewareConsumer.forRoutes()` method:

```
@@filename(app.module)
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';
import { CatsController } from './cats/cats.controller';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({ path: 'cats', method: RequestMethod.GET, version: '2' });
  }
}
```

With the code above, the `LoggerMiddleware` will only be applied to the version '2' of `/cats` endpoint.

**info** **Notice** Middlewares work with any versioning type described in this section: [URI](#), [Header](#), [Media Type](#) or [Custom](#).

## Task Scheduling

Task scheduling allows you to schedule arbitrary code (methods/functions) to execute at a fixed date/time, at recurring intervals, or once after a specified interval. In the Linux world, this is often handled by packages like [cron](#) at the OS level. For Node.js apps, there are several packages that emulate cron-like functionality. Nest provides the [@nestjs/schedule](#) package, which integrates with the popular Node.js [cron](#) package. We'll cover this package in the current chapter.

### Installation

To begin using it, we first install the required dependencies.

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

To activate job scheduling, import the [ScheduleModule](#) into the root [AppModule](#) and run the [forRoot\(\)](#) static method as shown below:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { ScheduleModule } from '@nestjs/schedule';

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

The [.forRoot\(\)](#) call initializes the scheduler and registers any declarative [cron jobs](#), [timeouts](#) and [intervals](#) that exist within your app. Registration occurs when the [onApplicationBootstrap](#) lifecycle hook occurs, ensuring that all modules have loaded and declared any scheduled jobs.

### Declarative cron jobs

A cron job schedules an arbitrary function (method call) to run automatically. Cron jobs can run:

- Once, at a specified date/time.
- On a recurring basis; recurring jobs can run at a specified instant within a specified interval (for example, once per hour, once per week, once every 5 minutes)

Declare a cron job with the [@Cron\(\)](#) decorator preceding the method definition containing the code to be executed, as follows:

```
import { Injectable, Logger } from '@nestjs/common';
import { Cron } from '@nestjs/schedule';
```

```
@Injectable()
export class TasksService {
  private readonly logger = new Logger(TasksService.name);

  @Cron('45 * * * *')
  handleCron() {
    this.logger.debug('Called when the current second is 45');
  }
}
```

In this example, the `handleCron()` method will be called each time the current second is `45`. In other words, the method will be run once per minute, at the 45 second mark.

The `@Cron()` decorator supports all standard [cron patterns](#):

- Asterisk (e.g. `*`)
- Ranges (e.g. `1-3,5`)
- Steps (e.g. `*/2`)

In the example above, we passed `45 * * * *` to the decorator. The following key shows how each position in the cron pattern string is interpreted:

<code>* * * * *</code>	
	day of week
	months
	day of month
	hours
	minutes
	seconds (optional)

Some sample cron patterns are:

<code>* * * * *</code>	every second
<code>45 * * * *</code>	every minute, on the 45th second
<code>0 10 * * *</code>	every hour, at the start of the 10th minute
<code>0 */30 9-17 * * *</code>	every 30 minutes between 9am and 5pm
<code>0 30 11 * * 1-5</code>	Monday to Friday at 11:30am

The `@nestjs/schedule` package provides a convenient enum with commonly used cron patterns. You can use this enum as follows:

```

import { Injectable, Logger } from '@nestjs/common';
import { Cron, CronExpression } from '@nestjs/schedule';

@Injectable()
export class TasksService {
  private readonly logger = new Logger(TasksService.name);

  @Cron(CronExpression.EVERY_30_SECONDS)
  handleCron() {
    this.logger.debug('Called every 30 seconds');
  }
}

```

In this example, the `handleCron()` method will be called every `30` seconds.

Alternatively, you can supply a JavaScript `Date` object to the `@Cron()` decorator. Doing so causes the job to execute exactly once, at the specified date.

**info Hint** Use JavaScript date arithmetic to schedule jobs relative to the current date. For example, `@Cron(new Date(Date.now() + 10 * 1000))` to schedule a job to run 10 seconds after the app starts.

Also, you can supply additional options as the second parameter to the `@Cron()` decorator.

<code>name</code>	Useful to access and control a cron job after it's been declared.
<code>timeZone</code>	Specify the timezone for the execution. This will modify the actual time relative to your timezone. If the timezone is invalid, an error is thrown. You can check all timezones available at <a href="#">Moment Timezone</a> website.
<code>utcOffset</code>	This allows you to specify the offset of your timezone rather than using the <code>timeZone</code> param.
<code>disabled</code>	This indicates whether the job will be executed at all.

```

import { Injectable } from '@nestjs/common';
import { Cron, CronExpression } from '@nestjs/schedule';

@Injectable()
export class NotificationService {
  @Cron('* * * * *', {
    name: 'notifications',
    timeZone: 'Europe/Paris',
  })
  triggerNotifications() {}
}

```

You can access and control a cron job after it's been declared, or dynamically create a cron job (where its cron pattern is defined at runtime) with the [Dynamic API](#). To access a declarative cron job via the API, you

must associate the job with a name by passing the `name` property in an optional options object as the second argument of the decorator.

## Declarative intervals

To declare that a method should run at a (recurring) specified interval, prefix the method definition with the `@Interval()` decorator. Pass the interval value, as a number in milliseconds, to the decorator as shown below:

```
@Interval(10000)
handleInterval() {
    this.logger.debug('Called every 10 seconds');
}
```

**info Hint** This mechanism uses the JavaScript `setInterval()` function under the hood. You can also utilize a cron job to schedule recurring jobs.

If you want to control your declarative interval from outside the declaring class via the [Dynamic API](#), associate the interval with a name using the following construction:

```
@Interval('notifications', 2500)
handleInterval() {}
```

The [Dynamic API](#) also enables **creating** dynamic intervals, where the interval's properties are defined at runtime, and **listing and deleting** them.

## Declarative timeouts

To declare that a method should run (once) at a specified timeout, prefix the method definition with the `@Timeout()` decorator. Pass the relative time offset (in milliseconds), from application startup, to the decorator as shown below:

```
@Timeout(5000)
handleTimeout() {
    this.logger.debug('Called once after 5 seconds');
}
```

**info Hint** This mechanism uses the JavaScript `setTimeout()` function under the hood.

If you want to control your declarative timeout from outside the declaring class via the [Dynamic API](#), associate the timeout with a name using the following construction:

```
@Timeout('notifications', 2500)
handleTimeout() {}
```

The [Dynamic API](#) also enables **creating** dynamic timeouts, where the timeout's properties are defined at runtime, and **listing and deleting** them.

## Dynamic schedule module API

The [@nestjs/schedule](#) module provides a dynamic API that enables managing declarative [cron jobs](#), [timeouts](#) and [intervals](#). The API also enables creating and managing **dynamic** cron jobs, timeouts and intervals, where the properties are defined at runtime.

### Dynamic cron jobs

Obtain a reference to a [CronJob](#) instance by name from anywhere in your code using the [SchedulerRegistry](#) API. First, inject [SchedulerRegistry](#) using standard constructor injection:

```
constructor(private schedulerRegistry: SchedulerRegistry) {}
```

**info Hint** Import the [SchedulerRegistry](#) from the [@nestjs/schedule](#) package.

Then use it in a class as follows. Assume a cron job was created with the following declaration:

```
@Cron('* * 8 * * *', {
  name: 'notifications',
})
triggerNotifications() {}
```

Access this job using the following:

```
const job = this.schedulerRegistry.getScheduler('notifications');

job.stop();
console.log(job.lastDate());
```

The [getScheduler\(\)](#) method returns the named cron job. The returned [CronJob](#) object has the following methods:

- [stop\(\)](#) - stops a job that is scheduled to run.
- [start\(\)](#) - restarts a job that has been stopped.
- [setTime\(time: CronTime\)](#) - stops a job, sets a new time for it, and then starts it
- [lastDate\(\)](#) - returns a string representation of the last date a job executed
- [nextDates\(count: number\)](#) - returns an array (size [count](#)) of [moment](#) objects representing upcoming job execution dates.

**info Hint** Use [toDate\(\)](#) on [moment](#) objects to render them in human readable form.

**Create** a new cron job dynamically using the `SchedulerRegistry#addCronJob` method, as follows:

```
addCronJob(name: string, seconds: string) {
  const job = new CronJob(`* ${seconds} * * * *`, () => {
    this.logger.warn(`time (${seconds}) for job ${name} to run!`);
  });

  this.schedulerRegistry.addCronJob(name, job);
  job.start();

  this.logger.warn(
    `job ${name} added for each minute at ${seconds} seconds!`,
  );
}
```

In this code, we use the `CronJob` object from the `cron` package to create the cron job. The `CronJob` constructor takes a cron pattern (just like the `@Cron()` decorator) as its first argument, and a callback to be executed when the cron timer fires as its second argument. The `SchedulerRegistry#addCronJob` method takes two arguments: a name for the `CronJob`, and the `CronJob` object itself.

**warning** **Warning** Remember to inject the `SchedulerRegistry` before accessing it. Import `CronJob` from the `cron` package.

**Delete** a named cron job using the `SchedulerRegistry#deleteCronJob` method, as follows:

```
deleteCron(name: string) {
  this.schedulerRegistry.deleteCronJob(name);
  this.logger.warn(`job ${name} deleted!`);
}
```

**List** all cron jobs using the `SchedulerRegistry#getCronJobs` method as follows:

```
getCrongs() {
  const jobs = this.schedulerRegistry.getCronJobs();
  jobs.forEach((value, key, map) => {
    let next;
    try {
      next = value.nextDates().toDate();
    } catch (e) {
      next = 'error: next fire date is in the past!';
    }
    this.logger.log(`job: ${key} -> next: ${next}`);
  });
}
```

The `getCronJobs()` method returns a `map`. In this code, we iterate over the map and attempt to access the `nextDates()` method of each `CronJob`. In the `CronJob` API, if a job has already fired and has no future firing dates, it throws an exception.

## Dynamic intervals

Obtain a reference to an interval with the `SchedulerRegistry#getInterval` method. As above, inject `SchedulerRegistry` using standard constructor injection:

```
constructor(private schedulerRegistry: SchedulerRegistry) {}
```

And use it as follows:

```
const interval = this.schedulerRegistry.getInterval('notifications');
clearInterval(interval);
```

**Create** a new interval dynamically using the `SchedulerRegistry#addInterval` method, as follows:

```
addInterval(name: string, milliseconds: number) {
  const callback = () => {
    this.logger.warn(`Interval ${name} executing at time
(${milliseconds})!`);
  };

  const interval = setInterval(callback, milliseconds);
  this.schedulerRegistry.addInterval(name, interval);
}
```

In this code, we create a standard JavaScript interval, then pass it to the `SchedulerRegistry#addInterval` method. That method takes two arguments: a name for the interval, and the interval itself.

**Delete** a named interval using the `SchedulerRegistry#deleteInterval` method, as follows:

```
deleteInterval(name: string) {
  this.schedulerRegistry.deleteInterval(name);
  this.logger.warn(`Interval ${name} deleted!`);
}
```

**List** all intervals using the `SchedulerRegistry#getIntervals` method as follows:

```
getIntervals() {
  const intervals = this.schedulerRegistry.getIntervals();
```

```
intervals.forEach(key => this.logger.log(`Interval: ${key}`));
}
```

## Dynamic timeouts

Obtain a reference to a timeout with the `SchedulerRegistry#getTimeout` method. As above, inject `SchedulerRegistry` using standard constructor injection:

```
constructor(private readonly schedulerRegistry: SchedulerRegistry) {}
```

And use it as follows:

```
const timeout = this.schedulerRegistry.getTimeout('notifications');
clearTimeout(timeout);
```

**Create** a new timeout dynamically using the `SchedulerRegistry#addTimeout` method, as follows:

```
addTimeout(name: string, milliseconds: number) {
  const callback = () => {
    this.logger.warn(`Timeout ${name} executing after
(${milliseconds})!`);
  };

  const timeout = setTimeout(callback, milliseconds);
  this.schedulerRegistry.addTimeout(name, timeout);
}
```

In this code, we create a standard JavaScript timeout, then pass it to the `SchedulerRegistry#addTimeout` method. That method takes two arguments: a name for the timeout, and the timeout itself.

**Delete** a named timeout using the `SchedulerRegistry#deleteTimeout` method, as follows:

```
deleteTimeout(name: string) {
  this.schedulerRegistry.deleteTimeout(name);
  this.logger.warn(`Timeout ${name} deleted!`);
}
```

**List** all timeouts using the `SchedulerRegistry#getTimeouts` method as follows:

```
getTimeouts() {
  const timeouts = this.schedulerRegistry.getTimeouts();
```

```
    timeouts.forEach(key => this.logger.log(`Timeout: ${key}`));  
}
```

## Example

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

## Queues

Queues are a powerful design pattern that help you deal with common application scaling and performance challenges. Some examples of problems that Queues can help you solve are:

- Smooth out processing peaks. For example, if users can initiate resource-intensive tasks at arbitrary times, you can add these tasks to a queue instead of performing them synchronously. Then you can have worker processes pull tasks from the queue in a controlled manner. You can easily add new Queue consumers to scale up the back-end task handling as the application scales up.
- Break up monolithic tasks that may otherwise block the Node.js event loop. For example, if a user request requires CPU intensive work like audio transcoding, you can delegate this task to other processes, freeing up user-facing processes to remain responsive.
- Provide a reliable communication channel across various services. For example, you can queue tasks (jobs) in one process or service, and consume them in another. You can be notified (by listening for status events) upon completion, error or other state changes in the job life cycle from any process or service. When Queue producers or consumers fail, their state is preserved and task handling can restart automatically when nodes are restarted.

Nest provides the [@nestjs/bull](#) package as an abstraction/wrapper on top of [Bull](#), a popular, well supported, high performance Node.js based Queue system implementation. The package makes it easy to integrate Bull Queues in a Nest-friendly way to your application.

Bull uses [Redis](#) to persist job data, so you'll need to have Redis installed on your system. Because it is Redis-backed, your Queue architecture can be completely distributed and platform-independent. For example, you can have some Queue [producers](#) and [consumers](#) and [listeners](#) running in Nest on one (or several) nodes, and other producers, consumers and listeners running on other Node.js platforms on other network nodes.

This chapter covers the [@nestjs/bull](#) package. We also recommend reading the [Bull documentation](#) for more background and specific implementation details.

## Installation

To begin using it, we first install the required dependencies.

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

Once the installation process is complete, we can import the [BullModule](#) into the root [AppModule](#).

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { BullModule } from '@nestjs/bull';

@Module({
  imports: [
    BullModule.forRoot({
      redis: {
        host: 'localhost',
        port: 6379,
        password: null,
        db: 0
      }
    })
  ]
})
```

```
        host: 'localhost',
        port: 6379,
    },
},
],
})
export class AppModule {}
```

The `forRoot()` method is used to register a `Bull` package configuration object that will be used by all queues registered in the application (unless specified otherwise). A configuration object consist of the following properties:

- `limiter: RateLimiter` - Options to control the rate at which the queue's jobs are processed. See [RateLimiter](#) for more information. Optional.
- `redis: RedisOpts` - Options to configure the Redis connection. See [RedisOpts](#) for more information. Optional.
- `prefix: string` - Prefix for all queue keys. Optional.
- `defaultJobOptions: JobOpts` - Options to control the default settings for new jobs. See [JobOpts](#) for more information. Optional.
- `settings: AdvancedSettings` - Advanced Queue configuration settings. These should usually not be changed. See [AdvancedSettings](#) for more information. Optional.

All the options are optional, providing detailed control over queue behavior. These are passed directly to the Bull `Queue` constructor. Read more about these options [here](#).

To register a queue, import the `BullModule.registerQueue()` dynamic module, as follows:

```
BullModule.registerQueue({
  name: 'audio',
});
```

**info Hint** Create multiple queues by passing multiple comma-separated configuration objects to the `registerQueue()` method.

The `registerQueue()` method is used to instantiate and/or register queues. Queues are shared across modules and processes that connect to the same underlying Redis database with the same credentials. Each queue is unique by its name property. A queue name is used as both an injection token (for injecting the queue into controllers/providers), and as an argument to decorators to associate consumer classes and listeners with queues.

You can also override some of the pre-configured options for a specific queue, as follows:

```
BullModule.registerQueue({
  name: 'audio',
  redis: {
    port: 6380,
```

```
},
});
```

Since jobs are persisted in Redis, each time a specific named queue is instantiated (e.g., when an app is started/restarted), it attempts to process any old jobs that may exist from a previous unfinished session.

Each queue can have one or many producers, consumers, and listeners. Consumers retrieve jobs from the queue in a specific order: FIFO (the default), LIFO, or according to priorities. Controlling queue processing order is discussed [here](#).

## Named configurations

If your queues connect to multiple different Redis instances, you can use a technique called **named configurations**. This feature allows you to register several configurations under specified keys, which then you can refer to in the queue options.

For example, assuming that you have an additional Redis instance (apart from the default one) used by a few queues registered in your application, you can register its configuration as follows:

```
BullModule.forRoot('alternative-config', {
  redis: {
    port: 6381,
  },
});
```

In the example above, '`'alternative-config'`' is just a configuration key (it can be any arbitrary string).

With this in place, you can now point to this configuration in the `registerQueue()` options object:

```
BullModule.registerQueue({
  configKey: 'alternative-config',
  name: 'video'
});
```

## Producers

Job producers add jobs to queues. Producers are typically application services (Nest [providers](#)). To add jobs to a queue, first inject the queue into the service as follows:

```
import { Injectable } from '@nestjs/common';
import { Queue } from 'bull';
import { InjectQueue } from '@nestjs/bull';

@Injectable()
export class AudioService {
```

```
constructor(@InjectQueue('audio') private audioQueue: Queue) {}  
}
```

**info Hint** The `@InjectQueue()` decorator identifies the queue by its name, as provided in the `registerQueue()` method call (e.g., `'audio'`).

Now, add a job by calling the queue's `add()` method, passing a user-defined job object. Jobs are represented as serializable JavaScript objects (since that is how they are stored in the Redis database). The shape of the job you pass is arbitrary; use it to represent the semantics of your job object.

```
const job = await this.audioQueue.add({  
  foo: 'bar',  
});
```

## Named jobs

Jobs may have unique names. This allows you to create specialized `consumers` that will only process jobs with a given name.

```
const job = await this.audioQueue.add('transcode', {  
  foo: 'bar',  
});
```

**Warning Warning** When using named jobs, you must create processors for each unique name added to a queue, or the queue will complain that you are missing a processor for the given job. See [here](#) for more information on consuming named jobs.

## Job options

Jobs can have additional options associated with them. Pass an options object after the `job` argument in the `Queue.add()` method. Job options properties are:

- `priority: number` - Optional priority value. Ranges from 1 (highest priority) to MAX\_INT (lowest priority). Note that using priorities has a slight impact on performance, so use them with caution.
- `delay: number` - An amount of time (milliseconds) to wait until this job can be processed. Note that for accurate delays, both server and clients should have their clocks synchronized.
- `attempts: number` - The total number of attempts to try the job until it completes.
- `repeat: RepeatOpts` - Repeat job according to a cron specification. See [RepeatOpts](#).
- `backoff: number | BackoffOpts` - Backoff setting for automatic retries if the job fails. See [BackoffOpts](#).
- `lifo: boolean` - If true, adds the job to the right end of the queue instead of the left (default false).
- `timeout: number` - The number of milliseconds after which the job should fail with a timeout error.
- `jobId: number | string` - Override the job ID - by default, the job ID is a unique integer, but you can use this setting to override it. If you use this option, it is up to you to ensure the jobId is unique. If you attempt to add a job with an id that already exists, it will not be added.

- `removeOnComplete: boolean | number` - If true, removes the job when it successfully completes. A number specifies the amount of jobs to keep. Default behavior is to keep the job in the completed set.
- `removeOnFail: boolean | number` - If true, removes the job when it fails after all attempts. A number specifies the amount of jobs to keep. Default behavior is to keep the job in the failed set.
- `stackTraceLimit: number` - Limits the amount of stack trace lines that will be recorded in the stacktrace.

Here are a few examples of customizing jobs with job options.

To delay the start of a job, use the `delay` configuration property.

```
const job = await this.audioQueue.add(  
  {  
    foo: 'bar',  
  },  
  { delay: 3000 }, // 3 seconds delayed  
);
```

To add a job to the right end of the queue (process the job as **LIFO** (Last In First Out)), set the `lifo` property of the configuration object to `true`.

```
const job = await this.audioQueue.add(  
  {  
    foo: 'bar',  
  },  
  { lifo: true },  
);
```

To prioritize a job, use the `priority` property.

```
const job = await this.audioQueue.add(  
  {  
    foo: 'bar',  
  },  
  { priority: 2 },  
);
```

## Consumers

A consumer is a **class** defining methods that either process jobs added into the queue, or listen for events on the queue, or both. Declare a consumer class using the `@Processor()` decorator as follows:

```
import { Processor } from '@nestjs/bull';
```

```
@Processor('audio')
export class AudioConsumer {}
```

info **Hint** Consumers must be registered as `providers` so the `@nestjs/bull` package can pick them up.

Where the decorator's string argument (e.g., `'audio'`) is the name of the queue to be associated with the class methods.

Within a consumer class, declare job handlers by decorating handler methods with the `@Process()` decorator.

```
import { Processor, Process } from '@nestjs/bull';
import { Job } from 'bull';

@Processor('audio')
export class AudioConsumer {
  @Process()
  async transcode(job: Job<unknown>) {
    let progress = 0;
    for (i = 0; i < 100; i++) {
      await doSomething(job.data);
      progress += 1;
      await job.progress(progress);
    }
    return {};
  }
}
```

The decorated method (e.g., `transcode()`) is called whenever the worker is idle and there are jobs to process in the queue. This handler method receives the `job` object as its only argument. The value returned by the handler method is stored in the `job` object and can be accessed later on, for example in a listener for the `completed` event.

`Job` objects have multiple methods that allow you to interact with their state. For example, the above code uses the `progress()` method to update the job's progress. See [here](#) for the complete `Job` object API reference.

You can designate that a job handler method will handle **only** jobs of a certain type (jobs with a specific `name`) by passing that `name` to the `@Process()` decorator as shown below. You can have multiple `@Process()` handlers in a given consumer class, corresponding to each job type (`name`). When you use named jobs, be sure to have a handler corresponding to each name.

```
@Process('transcode')
async transcode(job: Job<unknown>) { ... }
```

warning **Warning** When defining multiple consumers for the same queue, the `concurrency` option in `@Process({{ '{' }} concurrency: 1 {{ '}' }})` won't take effect. The minimum `concurrency` will match the number of consumers defined. This also applies even if `@Process()` handlers use a different `name` to handle named jobs.

## Request-scoped consumers

When a consumer is flagged as request-scoped (learn more about the injection scopes [here](#)), a new instance of the class will be created exclusively for each job. The instance will be garbage-collected after the job has completed.

```
@Processor({
  name: 'audio',
  scope: Scope.REQUEST,
})
```

Since request-scoped consumer classes are instantiated dynamically and scoped to a single job, you can inject a `JOB_REF` through the constructor using a standard approach.

```
constructor(@Inject(JOB_REF) jobRef: Job) {
  console.log(jobRef);
}
```

info **Hint** The `JOB_REF` token is imported from the `@nestjs/bull` package.

## Event listeners

Bull generates a set of useful events when queue and/or job state changes occur. Nest provides a set of decorators that allow subscribing to a core set of standard events. These are exported from the `@nestjs/bull` package.

Event listeners must be declared within a `consumer` class (i.e., within a class decorated with the `@Processor()` decorator). To listen for an event, use one of the decorators in the table below to declare a handler for the event. For example, to listen to the event emitted when a job enters the active state in the `audio` queue, use the following construct:

```
import { Processor, Process, OnQueueActive } from '@nestjs/bull';
import { Job } from 'bull';

@Processor('audio')
export class AudioConsumer {

  @OnQueueActive()
  onActive(job: Job) {
    console.log(
      `Processing job ${job.id} of type ${job.name} with data
${job.data}`
    );
  }
}
```

```

${job.data}...`  

    );  

}  

...

```

Since Bull operates in a distributed (multi-node) environment, it defines the concept of event locality. This concept recognizes that events may be triggered either entirely within a single process, or on shared queues from different processes. A **local** event is one that is produced when an action or state change is triggered on a queue in the local process. In other words, when your event producers and consumers are local to a single process, all events happening on queues are local.

When a queue is shared across multiple processes, we encounter the possibility of **global** events. For a listener in one process to receive an event notification triggered by another process, it must register for a global event.

Event handlers are invoked whenever their corresponding event is emitted. The handler is called with the signature shown in the table below, providing access to information relevant to the event. We discuss one key difference between local and global event handler signatures below.

Local event listeners	Global event listeners	Handler method signature / When fired
<code>@OnQueueError()</code>	<code>@OnGlobalQueueError()</code>	<code>handler(error: Error)</code> - An error occurred. <code>error</code> contains the triggering error.
<code>@OnQueueWaiting()</code>	<code>@OnGlobalQueueWaiting()</code>	<code>handler(jobId: number   string)</code> - A Job is waiting to be processed as soon as a worker is idling. <code>jobId</code> contains the id for the job that has entered this state.
<code>@OnQueueActive()</code>	<code>@OnGlobalQueueActive()</code>	<code>handler(job: Job)</code> - Job <code>job</code> has started.
<code>@OnQueueStalled()</code>	<code>@OnGlobalQueueStalled()</code>	<code>handler(job: Job)</code> - Job <code>job</code> has been marked as stalled. This is useful for debugging job workers that crash or pause the event loop.
<code>@OnQueueProgress()</code>	<code>@OnGlobalQueueProgress()</code>	<code>handler(job: Job, progress: number)</code> - Job <code>job</code> 's progress was updated to value <code>progress</code> .
<code>@OnQueueCompleted()</code>	<code>@OnGlobalQueueCompleted()</code>	<code>handler(job: Job, result: any)</code> Job <code>job</code> successfully completed with a result <code>result</code> .
<code>@OnQueueFailed()</code>	<code>@OnGlobalQueueFailed()</code>	<code>handler(job: Job, err: Error)</code> Job <code>job</code> failed with reason <code>err</code> .

<code>@OnQueuePaused()</code>	<code>@OnGlobalQueuePaused()</code>	<code>handler()</code> The queue has been paused.
<code>@OnQueueResumed()</code>	<code>@OnGlobalQueueResumed()</code>	<code>handler(job: Job)</code> The queue has been resumed.
<code>@OnQueueCleaned()</code>	<code>@OnGlobalQueueCleaned()</code>	<code>handler(jobs: Job[], type: string)</code> Old jobs have been cleaned from the queue. <code>jobs</code> is an array of cleaned jobs, and <code>type</code> is the type of jobs cleaned.
<code>@OnQueueDrained()</code>	<code>@OnGlobalQueueDrained()</code>	<code>handler()</code> Emitted whenever the queue has processed all the waiting jobs (even if there can be some delayed jobs not yet processed).
<code>@OnQueueRemoved()</code>	<code>@OnGlobalQueueRemoved()</code>	<code>handler(job: Job)</code> Job <code>job</code> was successfully removed.

When listening for global events, the method signatures can be slightly different from their local counterpart. Specifically, any method signature that receives `job` objects in the local version, instead receives a `jobId (number)` in the global version. To get a reference to the actual `job` object in such a case, use the `Queue#getJob` method. This call should be awaited, and therefore the handler should be declared `async`. For example:

```
@OnGlobalQueueCompleted()
async onGlobalCompleted(jobId: number, result: any) {
  const job = await this.immediateQueue.getJob(jobId);
  console.log('(Global) on completed: job ', job.id, ' -> result: ',
  result);
}
```

**info Hint** To access the `Queue` object (to make a `getJob()` call), you must of course inject it. Also, the Queue must be registered in the module where you are injecting it.

In addition to the specific event listener decorators, you can also use the generic `@OnQueueEvent()` decorator in combination with either `BullQueueEvents` or `BullQueueGlobalEvents` enums. Read more about events [here](#).

## Queue management

Queue's have an API that allows you to perform management functions like pausing and resuming, retrieving the count of jobs in various states, and several more. You can find the full queue API [here](#). Invoke any of these methods directly on the `Queue` object, as shown below with the pause/resume examples.

Pause a queue with the `pause()` method call. A paused queue will not process new jobs until resumed, but current jobs being processed will continue until they are finalized.

```
await audioQueue.pause();
```

To resume a paused queue, use the `resume()` method, as follows:

```
await audioQueue.resume();
```

## Separate processes

Job handlers can also be run in a separate (forked) process ([source](#)). This has several advantages:

- The process is sandboxed so if it crashes it does not affect the worker.
- You can run blocking code without affecting the queue (jobs will not stall).
- Much better utilization of multi-core CPUs.
- Less connections to redis.

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { BullModule } from '@nestjs/bull';
import { join } from 'path';

@Module({
  imports: [
    BullModule.registerQueue({
      name: 'audio',
      processors: [join(__dirname, 'processor.js')],
    }),
  ],
})
export class AppModule {}
```

Please note that because your function is being executed in a forked process, Dependency Injection (and IoC container) won't be available. That means that your processor function will need to contain (or create) all instances of external dependencies it needs.

```
@@filename(processor)
import { Job, DoneCallback } from 'bull';

export default function (job: Job, cb: DoneCallback) {
  console.log(`[${process.pid}] ${JSON.stringify(job.data)}`);
  cb(null, 'It works');
}
```

## Async configuration

You may want to pass `bull` options asynchronously instead of statically. In this case, use the `forRootAsync()` method which provides several ways to deal with async configuration. Likewise, if you want to pass queue options asynchronously, use the `registerQueueAsync()` method.

One approach is to use a factory function:

```
BullModule.forRootAsync({
  useFactory: () => ({
    redis: {
      host: 'localhost',
      port: 6379,
    },
  }),
});
```

Our factory behaves like any other [asynchronous provider](#) (e.g., it can be `async` and it's able to inject dependencies through `inject`).

```
BullModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    redis: {
      host: configService.get('QUEUE_HOST'),
      port: configService.get('QUEUE_PORT'),
    },
  }),
  inject: [ConfigService],
});
```

Alternatively, you can use the `useClass` syntax:

```
BullModule.forRootAsync({
  useClass: BullConfigService,
});
```

The construction above will instantiate `BullConfigService` inside `BullModule` and use it to provide an options object by calling `createSharedConfiguration()`. Note that this means that the `BullConfigService` has to implement the `SharedBullConfigurationFactory` interface, as shown below:

```
@Injectable()
class BullConfigService implements SharedBullConfigurationFactory {
  createSharedConfiguration(): BullModuleOptions {
    return {
      redis: {
```

```
        host: 'localhost',
        port: 6379,
    },
};

}
```

In order to prevent the creation of `BullConfigService` inside `BullModule` and use a provider imported from a different module, you can use the `useExisting` syntax.

```
BullModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

This construction works the same as `useClass` with one critical difference - `BullModule` will lookup imported modules to reuse an existing `ConfigService` instead of instantiating a new one.

## Example

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

## Logger

Nest comes with a built-in text-based logger which is used during application bootstrapping and several other circumstances such as displaying caught exceptions (i.e., system logging). This functionality is provided via the [Logger](#) class in the [@nestjs/common](#) package. You can fully control the behavior of the logging system, including any of the following:

- disable logging entirely
- specify the log level of detail (e.g., display errors, warnings, debug information, etc.)
- override timestamp in the default logger (e.g., use ISO8601 standard as date format)
- completely override the default logger
- customize the default logger by extending it
- make use of dependency injection to simplify composing and testing your application

You can also make use of the built-in logger, or create your own custom implementation, to log your own application-level events and messages.

For more advanced logging functionality, you can make use of any Node.js logging package, such as [Winston](#), to implement a completely custom, production grade logging system.

### Basic customization

To disable logging, set the `logger` property to `false` in the (optional) Nest application options object passed as the second argument to the [NestFactory.create\(\)](#) method.

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

To enable specific logging levels, set the `logger` property to an array of strings specifying the log levels to display, as follows:

```
const app = await NestFactory.create(AppModule, {
  logger: ['error', 'warn'],
});
await app.listen(3000);
```

Values in the array can be any combination of `'log'`, `'error'`, `'warn'`, `'debug'`, and `'verbose'`.

**info Hint** To disable color in the default logger's messages, set the `NO_COLOR` environment variable to some non-empty string.

### Custom implementation

You can provide a custom logger implementation to be used by Nest for system logging by setting the value of the `logger` property to an object that fulfills the `LoggerService` interface. For example, you can tell Nest to use the built-in global JavaScript `console` object (which implements the `LoggerService` interface), as follows:

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

Implementing your own custom logger is straightforward. Simply implement each of the methods of the `LoggerService` interface as shown below.

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

export class MyLogger implements LoggerService {
  /**
   * Write a 'log' level log.
   */
  log(message: any, ...optionalParams: any[]) {}

  /**
   * Write an 'error' level log.
   */
  error(message: any, ...optionalParams: any[]) {}

  /**
   * Write a 'warn' level log.
   */
  warn(message: any, ...optionalParams: any[]) {}

  /**
   * Write a 'debug' level log.
   */
  debug?(message: any, ...optionalParams: any[]) {}

  /**
   * Write a 'verbose' level log.
   */
  verbose?(message: any, ...optionalParams: any[])
}
```

You can then supply an instance of `MyLogger` via the `logger` property of the Nest application options object.

```
const app = await NestFactory.create(AppModule, {
  logger: new MyLogger(),
```

```
});  
await app.listen(3000);
```

This technique, while simple, doesn't utilize dependency injection for the `MyLogger` class. This can pose some challenges, particularly for testing, and limit the reusability of `MyLogger`. For a better solution, see the [Dependency Injection](#) section below.

## Extend built-in logger

Rather than writing a logger from scratch, you may be able to meet your needs by extending the built-in `ConsoleLogger` class and overriding selected behavior of the default implementation.

```
import { ConsoleLogger } from '@nestjs/common';  
  
export class MyLogger extends ConsoleLogger {  
  error(message: any, stack?: string, context?: string) {  
    // add your tailored logic here  
    super.error(...arguments);  
  }  
}
```

You can use such an extended logger in your feature modules as described in the [Using the logger for application logging](#) section below.

You can tell Nest to use your extended logger for system logging by passing an instance of it via the `logger` property of the application options object (as shown in the [Custom implementation](#) section above), or by using the technique shown in the [Dependency Injection](#) section below. If you do so, you should take care to call `super`, as shown in the sample code above, to delegate the specific log method call to the parent (built-in) class so that Nest can rely on the built-in features it expects.

## Dependency injection

For more advanced logging functionality, you'll want to take advantage of dependency injection. For example, you may want to inject a `ConfigService` into your logger to customize it, and in turn inject your custom logger into other controllers and/or providers. To enable dependency injection for your custom logger, create a class that implements `LoggerService` and register that class as a provider in some module. For example, you can

1. Define a `MyLogger` class that either extends the built-in `ConsoleLogger` or completely overrides it, as shown in previous sections. Be sure to implement the `LoggerService` interface.
2. Create a `LoggerModule` as shown below, and provide `MyLogger` from that module.

```
import { Module } from '@nestjs/common';  
import { MyLogger } from './my-logger.service';  
  
@Module({  
  providers: [MyLogger],
```

```
  exports: [MyLogger],  
}  
export class LoggerModule {}
```

With this construct, you are now providing your custom logger for use by any other module. Because your `MyLogger` class is part of a module, it can use dependency injection (for example, to inject a `ConfigService`). There's one more technique needed to provide this custom logger for use by Nest for system logging (e.g., for bootstrapping and error handling).

Because application instantiation (`NestFactory.create()`) happens outside the context of any module, it doesn't participate in the normal Dependency Injection phase of initialization. So we must ensure that at least one application module imports the `LoggerModule` to trigger Nest to instantiate a singleton instance of our `MyLogger` class.

We can then instruct Nest to use the same singleton instance of `MyLogger` with the following construction:

```
const app = await NestFactory.create(AppModule, {  
  bufferLogs: true,  
});  
app.useLogger(app.get(MyLogger));  
await app.listen(3000);
```

**info Note** In the example above, we set the `bufferLogs` to `true` to make sure all logs will be buffered until a custom logger is attached (`MyLogger` in this case) and the application initialisation process either completes or fails. If the initialisation process fails, Nest will fallback to the original `ConsoleLogger` to print out any reported error messages. Also, you can set the `autoFlushLogs` to `false` (default `true`) to manually flush logs (using the `Logger#flush()` method).

Here we use the `get()` method on the `NestApplication` instance to retrieve the singleton instance of the `MyLogger` object. This technique is essentially a way to "inject" an instance of a logger for use by Nest. The `app.get()` call retrieves the singleton instance of `MyLogger`, and depends on that instance being first injected in another module, as described above.

You can also inject this `MyLogger` provider in your feature classes, thus ensuring consistent logging behavior across both Nest system logging and application logging. See [Using the logger for application logging](#) and [Injecting a custom logger](#) below for more information.

## Using the logger for application logging

We can combine several of the techniques above to provide consistent behavior and formatting across both Nest system logging and our own application event/message logging.

A good practice is to instantiate `Logger` class from `@nestjs/common` in each of our services. We can supply our service name as the `context` argument in the `Logger` constructor, like so:

```
import { Logger, Injectable } from '@nestjs/common';
```

```
@Injectable()
class MyService {
  private readonly logger = new Logger(MyService.name);

  doSomething() {
    this.logger.log('Doing something...');
  }
}
```

In the default logger implementation, `context` is printed in the square brackets, like `NestFactory` in the example below:

```
[Nest] 19096 - 12/08/2019, 7:12:59 AM [NestFactory] Starting Nest application...
```

If we supply a custom logger via `app.useLogger()`, it will actually be used by Nest internally. That means that our code remains implementation agnostic, while we can easily substitute the default logger for our custom one by calling `app.useLogger()`.

That way if we follow the steps from the previous section and call `app.useLogger(app.get(MyLogger))`, the following calls to `this.logger.log()` from `MyService` would result in calls to method `log` from `MyLogger` instance.

This should be suitable for most cases. But if you need more customization (like adding and calling custom methods), move to the next section.

## Injecting a custom logger

To start, extend the built-in logger with code like the following. We supply the `scope` option as configuration metadata for the `ConsoleLogger` class, specifying a `transient` scope, to ensure that we'll have a unique instance of the `MyLogger` in each feature module. In this example, we do not extend the individual `ConsoleLogger` methods (like `log()`, `warn()`, etc.), though you may choose to do so.

```
import { Injectable, Scope, ConsoleLogger } from '@nestjs/common';

@Injectable({ scope: Scope.TRANSIENT })
export class MyLogger extends ConsoleLogger {
  customLog() {
    this.log('Please feed the cat!');
  }
}
```

Next, create a `LoggerModule` with a construction like this:

```
import { Module } from '@nestjs/common';
import { MyLogger } from './my-logger.service';
```

```
@Module({
  providers: [MyLogger],
  exports: [MyLogger],
})
export class LoggerModule {}
```

Next, import the `LoggerModule` into your feature module. Since we extended default `Logger` we have the convenience of using `setContext` method. So we can start using the context-aware custom logger, like this:

```
import { Injectable } from '@nestjs/common';
import { MyLogger } from './my-logger.service';

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [];

  constructor(private myLogger: MyLogger) {
    // Due to transient scope, CatsService has its own unique instance of
    // MyLogger,
    // so setting context here will not affect other instances in other
    // services
    this.myLogger.setContext('CatsService');
  }

  findAll(): Cat[] {
    // You can call all the default methods
    this.myLogger.warn('About to return cats!');
    // And your custom methods
    this.myLogger.customLog();
    return this.cats;
  }
}
```

Finally, instruct Nest to use an instance of the custom logger in your `main.ts` file as shown below. Of course in this example, we haven't actually customized the logger behavior (by extending the `Logger` methods like `log()`, `warn()`, etc.), so this step isn't actually needed. But it **would** be needed if you added custom logic to those methods and wanted Nest to use the same implementation.

```
const app = await NestFactory.create(AppModule, {
  bufferLogs: true,
});
app.useLogger(new MyLogger());
await app.listen(3000);
```

**info Hint** Alternatively, instead of setting `bufferLogs` to `true`, you could temporarily disable the logger with `logger: false` instruction. Be mindful that if you supply `logger: false` to `NestFactory.create`, nothing will be logged until you call `useLogger`, so you may miss some important initialization errors. If you don't mind that some of your initial messages will be logged with the default logger, you can just omit the `logger: false` option.

## Use external logger

Production applications often have specific logging requirements, including advanced filtering, formatting and centralized logging. Nest's built-in logger is used for monitoring Nest system behavior, and can also be useful for basic formatted text logging in your feature modules while in development, but production applications often take advantage of dedicated logging modules like [Winston](#). As with any standard Node.js application, you can take full advantage of such modules in Nest.

## Cookies

An **HTTP cookie** is a small piece of data stored by the user's browser. Cookies were designed to be a reliable mechanism for websites to remember stateful information. When the user visits the website again, the cookie is automatically sent with the request.

### Use with Express (default)

First install the [required package](#) (and its types for TypeScript users):

```
$ npm i cookie-parser
$ npm i -D @types/cookie-parser
```

Once the installation is complete, apply the `cookie-parser` middleware as global middleware (for example, in your `main.ts` file).

```
import * as cookieParser from 'cookie-parser';
// somewhere in your initialization file
app.use(cookieParser());
```

You can pass several options to the `cookieParser` middleware:

- `secret` a string or array used for signing cookies. This is optional and if not specified, will not parse signed cookies. If a string is provided, this is used as the secret. If an array is provided, an attempt will be made to unsign the cookie with each secret in order.
- `options` an object that is passed to `cookie.parse` as the second option. See [cookie](#) for more information.

The middleware will parse the `Cookie` header on the request and expose the cookie data as the property `req.cookies` and, if a secret was provided, as the property `req.signedCookies`. These properties are name value pairs of the cookie name to cookie value.

When secret is provided, this module will unsign and validate any signed cookie values and move those name value pairs from `req.cookies` into `req.signedCookies`. A signed cookie is a cookie that has a value prefixed with `s:`. Signed cookies that fail signature validation will have the value `false` instead of the tampered value.

With this in place, you can now read cookies from within the route handlers, as follows:

```
@Get()
findAll(@Req() request: Request) {
  console.log(request.cookies); // or "request.cookies['cookieKey']"
  // or console.log(request.signedCookies);
}
```

**info Hint** The `@Req()` decorator is imported from the `@nestjs/common`, while `Request` from the `express` package.

To attach a cookie to an outgoing response, use the `Response#cookie()` method:

```
@Get()  
findAll(@Res({ passthrough: true }) response: Response) {  
  response.cookie('key', 'value')  
}
```

**warning Warning** If you want to leave the response handling logic to the framework, remember to set the `passthrough` option to `true`, as shown above. Read more [here](#).

**info Hint** The `@Res()` decorator is imported from the `@nestjs/common`, while `Response` from the `express` package.

## Use with Fastify

First install the required package:

```
$ npm i @fastify/cookie
```

Once the installation is complete, register the `@fastify/cookie` plugin:

```
import fastifyCookie from '@fastify/cookie';  
  
// somewhere in your initialization file  
const app = await NestFactory.create<NestFastifyApplication>( AppModule,  
  new FastifyAdapter(), );  
await app.register(fastifyCookie, { secret: 'my-secret', // for cookies signature });
```

With this in place, you can now read cookies from within the route handlers, as follows:

```
@Get()  
findAll(@Req() request: FastifyRequest) {  
  console.log(request.cookies); // or "request.cookies['cookieKey']"  
}
```

**info Hint** The `@Req()` decorator is imported from the `@nestjs/common`, while `FastifyRequest` from the `fastify` package.

To attach a cookie to an outgoing response, use the `FastifyReply#setCookie()` method:

```
@Get()  
findAll(@Res({ passthrough: true }) response: FastifyReply) {  
  response.setCookie('key', 'value')  
}
```

To read more about `FastifyReply#setCookie()` method, check out this [page](#).

**warning** **Warning** If you want to leave the response handling logic to the framework, remember to set the `passthrough` option to `true`, as shown above. Read more [here](#).

**info** **Hint** The `@Res()` decorator is imported from the `@nestjs/common`, while `FastifyReply` from the `fastify` package.

## Creating a custom decorator (cross-platform)

To provide a convenient, declarative way of accessing incoming cookies, we can create a [custom decorator](#).

```
import { createParamDecorator, ExecutionContext } from '@nestjs/common';  
  
export const Cookies = createParamDecorator(  
  (data: string, ctx: ExecutionContext) => {  
    const request = ctx.switchToHttp().getRequest();  
    return data ? request.cookies?.[data] : request.cookies;  
  },  
);
```

The `@Cookies()` decorator will extract all cookies, or a named cookie from the `req.cookies` object and populate the decorated parameter with that value.

With this in place, we can now use the decorator in a route handler signature, as follows:

```
@Get()  
findAll(@Cookies('name') name: string) {}
```

## Events

[Event Emitter](#) package (`@nestjs/event-emitter`) provides a simple observer implementation, allowing you to subscribe and listen for various events that occur in your application. Events serve as a great way to decouple various aspects of your application, since a single event can have multiple listeners that do not depend on each other.

`EventEmitterModule` internally uses the `eventemitter2` package.

### Getting started

First install the required package:

```
$ npm i --save @nestjs/event-emitter
```

Once the installation is complete, import the `EventEmitterModule` into the root `AppModule` and run the `forRoot()` static method as shown below:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { EventEmitterModule } from '@nestjs/event-emitter';

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

The `.forRoot()` call initializes the event emitter and registers any declarative event listeners that exist within your app. Registration occurs when the `onApplicationBootstrap` lifecycle hook occurs, ensuring that all modules have loaded and declared any scheduled jobs.

To configure the underlying `EventEmitter` instance, pass the configuration object to the `.forRoot()` method, as follows:

```
EventEmitterModule.forRoot({
  // set this to `true` to use wildcards
  wildcard: false,
  // the delimiter used to segment namespaces
  delimiter: '.',
  // set this to `true` if you want to emit the newListener event
  newListener: false,
  // set this to `true` if you want to emit the removeListener event
  removeListener: false,
  // the maximum amount of listeners that can be assigned to an event
```

```
maxListeners: 10,  
  // show event name in memory leak message when more than maximum amount  
  // of listeners is assigned  
  verboseMemoryLeak: false,  
  // disable throwing uncaughtException if an error event is emitted and  
  // it has no listeners  
  ignoreErrors: false,  
);
```

## Dispatching Events

To dispatch (i.e., fire) an event, first inject `EventEmitter2` using standard constructor injection:

```
constructor(private eventEmitter: EventEmitter2) {}
```

**info Hint** Import the `EventEmitter2` from the `@nestjs/event-emitter` package.

Then use it in a class as follows:

```
this.eventEmitter.emit(  
  'order.created',  
  new OrderCreatedEvent({  
    orderId: 1,  
    payload: {},  
  }),  
);
```

## Listening to Events

To declare an event listener, decorate a method with the `@OnEvent()` decorator preceding the method definition containing the code to be executed, as follows:

```
@OnEvent('order.created')  
handleOrderCreatedEvent(payload: OrderCreatedEvent) {  
  // handle and process "OrderCreatedEvent" event  
}
```

**warning Warning** Event subscribers cannot be request-scoped.

The first argument can be a `string` or `symbol` for a simple event emitter and a `string | symbol | Array<string | symbol>` in a case of a wildcard emitter. The second argument (optional) is a listener options object ([read more](#)).

```
@OnEvent('order.created', { async: true })
handleOrderCreatedEvent(payload: OrderCreatedEvent) {
    // handle and process "OrderCreatedEvent" event
}
```

To use namespaces/wildcards, pass the `wildcard` option into the `EventEmitterModule#forRoot()` method. When namespaces/wildcards are enabled, events can either be strings (`foo.bar`) separated by a delimiter or arrays (`['foo', 'bar']`). The delimiter is also configurable as a configuration property (`delimiter`). With namespaces feature enabled, you can subscribe to events using a wildcard:

```
@OnEvent('order.*')
handleOrderEvents(payload: OrderCreatedEvent | OrderRemovedEvent |
OrderUpdatedEvent) {
    // handle and process an event
}
```

Note that such a wildcard only applies to one block. The argument `order.*` will match, for example, the events `order.created` and `order.shipped` but not `order.delayed.out_of_stock`. In order to listen to such events, use the `multilevel wildcard` pattern (i.e, `**`), described in the [EventEmitter2 documentation](#).

With this pattern, you can, for example, create an event listener that catches all events.

```
@OnEvent('**')
handleEverything(payload: any) {
    // handle and process an event
}
```

**info Hint** `EventEmitter2` class provides several useful methods for interacting with events, like `waitFor` and `onAny`. You can read more about them [here](#).

## Example

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

## Compression

Compression can greatly decrease the size of the response body, thereby increasing the speed of a web app.

For **high-traffic** websites in production, it is strongly recommended to offload compression from the application server - typically in a reverse proxy (e.g., Nginx). In that case, you should not use compression middleware.

### Use with Express (default)

Use the [compression](#) middleware package to enable gzip compression.

First install the required package:

```
$ npm i --save compression
```

Once the installation is complete, apply the compression middleware as global middleware.

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

### Use with Fastify

If using the [FastifyAdapter](#), you'll want to use [fastify-compress](#):

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

Once the installation is complete, apply the [@fastify/compress](#) middleware as global middleware.

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

By default, [@fastify/compress](#) will use Brotli compression (on Node >= 11.7.0) when browsers indicate support for the encoding. While Brotli is quite efficient in terms of compression ratio, it's also quite slow. Due to this, you may want to tell fastify-compress to only use deflate and gzip to compress responses; you'll end up with larger responses but they'll be delivered much more quickly.

To specify encodings, provide a second argument to [app.register](#):

```
await app.register(compression, { encodings: ['gzip', 'deflate'] });
```

The above tells `fastify-compress` to only use gzip and deflate encodings, preferring gzip if the client supports both.

## File upload

To handle file uploading, Nest provides a built-in module based on the [multer](#) middleware package for Express. Multer handles data posted in the [multipart/form-data](#) format, which is primarily used for uploading files via an HTTP [POST](#) request. This module is fully configurable and you can adjust its behavior to your application requirements.

**warning** **Warning** Multer cannot process data which is not in the supported multipart format ([multipart/form-data](#)). Also, note that this package is not compatible with the [FastifyAdapter](#).

For better type safety, let's install Multer typings package:

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

With this package installed, we can now use the [Express.Multer.File](#) type (you can import this type as follows: `import {{ '{' }} Express {{ '}' }} from 'express'`).

### Basic example

To upload a single file, simply tie the [FileInterceptor\(\)](#) interceptor to the route handler and extract [file](#) from the [request](#) using the [@UploadedFile\(\)](#) decorator.

```
@@filename()
@Post('upload')
@UseInterceptors(FileInterceptor('file'))
uploadFile(@UploadedFile() file: Express.Multer.File) {
  console.log(file);
}
@switch
@Post('upload')
@UseInterceptors(FileInterceptor('file'))
@Bind(UploadedFile())
uploadFile(file) {
  console.log(file);
}
```

**info Hint** The [FileInterceptor\(\)](#) decorator is exported from the [@nestjs/platform-express](#) package. The [@UploadedFile\(\)](#) decorator is exported from [@nestjs/common](#).

The [FileInterceptor\(\)](#) decorator takes two arguments:

- [fieldName](#): string that supplies the name of the field from the HTML form that holds a file
- [options](#): optional object of type [MulterOptions](#). This is the same object used by the multer constructor (more details [here](#)).

warning **Warning** `FileInterceptor()` may not be compatible with third party cloud providers like Google Firebase or others.

## File validation

Often times it can be useful to validate incoming file metadata, like file size or file mime-type. For this, you can create your own `Pipe` and bind it to the parameter annotated with the `UploadedFile` decorator. The example below demonstrates how a basic file size validator pipe could be implemented:

```
import { PipeTransform, Injectable, ArgumentMetadata } from
'@nestjs/common';

@Injectable()
export class FileSizeValidationPipe implements PipeTransform {
  transform(value: any, metadata: ArgumentMetadata) {
    // "value" is an object containing the file's attributes and metadata
    const oneKb = 1000;
    return value.size < oneKb;
  }
}
```

Nest provides a built-in pipe to handle common use cases and facilitate/standardize the addition of new ones. This pipe is called `ParseFilePipe`, and you can use it as follows:

```
@Post('file')
uploadFileAndPassValidation(
  @Body() body: SampleDto,
  @UploadedFile()
  new ParseFilePipe({
    validators: [
      // ... Set of file validator instances here
    ]
  })
  file: Express.Multer.File,
) {
  return {
    body,
    file: file.buffer.toString(),
  };
}
```

As you can see, it's required to specify an array of file validators that will be executed by the `ParseFilePipe`. We'll discuss the interface of a validator, but it's worth mentioning this pipe also has two additional **optional** options:

**errorHttpStatusCode** The HTTP status code to be thrown in case **any** validator fails. Default is **400 (BAD REQUEST)**

---

exceptionFactory	A factory which receives the error message and returns an error.
------------------	--

Now, back to the `FileValidator` interface. To integrate validators with this pipe, you have to either use built-in implementations or provide your own custom `FileValidator`. See example below:

```
export abstract class FileValidator<TValidationOptions = Record<string, any>> {
  constructor(protected readonly validationOptions: TValidationOptions) {}

  /**
   * Indicates if this file should be considered valid, according to the
   * options passed in the constructor.
   * @param file the file from the request object
   */
  abstract isValid(file?: any): boolean | Promise<boolean>;

  /**
   * Builds an error message in case the validation fails.
   * @param file the file from the request object
   */
  abstract buildErrorMessage(file: any): string;
}
```

**info Hint** The `FileValidator` interfaces supports async validation via its `isValid` function. To leverage type security, you can also type the `file` parameter as `Express.Multer.File` in case you are using express (default) as a driver.

`FileValidator` is a regular class that has access to the file object and validates it according to the options provided by the client. Nest has two built-in `FileValidator` implementations you can use in your project:

- `MaxFileSizeValidator` - Checks if a given file's size is less than the provided value (measured in bytes)
- `FileTypeValidator` - Checks if a given file's mime-type matches the given value.

**warning Warning** To verify file type, `FileTypeValidator` class uses the type as detected by multer. By default, multer derives file type from file extension on user's device. However, it does not check actual file contents. As files can be renamed to arbitrary extensions, consider using a custom implementation (like checking the file's [magic number](#)) if your app requires a safer solution.

To understand how these can be used in conjunction with the aforementioned `FileParsePipe`, we'll use an altered snippet of the last presented example:

```
@UploadedFile(
  new ParseFilePipe({
    validators: [
      new MaxFileSizeValidator({ maxSize: 1000 }),
      new FileTypeValidator({ fileType: 'image/jpeg' }),
    ]
  })
)
```

```

    ],
  }),
)
file: Express.Multer.File,

```

**info Hint** If the number of validators increase largely or their options are cluttering the file, you can define this array in a separate file and import it here as a named constant like `fileValidators`.

Finally, you can use the special `ParseFilePipeBuilder` class that lets you compose & construct your validators. By using it as shown below you can avoid manual instantiation of each validator and just pass their options directly:

```

@UploadedFile(
  new ParseFilePipeBuilder()
    .addFileTypeValidator({
      fileType: 'jpeg',
    })
    .addMaxSizeValidator({
      maxSize: 1000
    })
    .build({
      errorHttpStatusCode: HttpStatus.UNPROCESSABLE_ENTITY
    }),
)
file: Express.Multer.File,

```

## Array of files

To upload an array of files (identified with a single field name), use the `FilesInterceptor()` decorator (note the plural **Files** in the decorator name). This decorator takes three arguments:

- `fieldName`: as described above
- `maxCount`: optional number defining the maximum number of files to accept
- `options`: optional `MulterOptions` object, as described above

When using `FilesInterceptor()`, extract files from the `request` with the `@UploadedFiles()` decorator.

```

@@filename()
@Post('upload')
@UseInterceptors(FilesInterceptor('files'))
uploadFile(@UploadedFiles() files: Array<Express.Multer.File>) {
  console.log(files);
}
@@switch
@Post('upload')
@UseInterceptors(FilesInterceptor('files'))
@Bind(UploadedFiles())

```

```
uploadFile(files) {
  console.log(files);
}
```

**info Hint** The `FilesInterceptor()` decorator is exported from the `@nestjs/platform-express` package. The `@UploadedFiles()` decorator is exported from `@nestjs/common`.

## Multiple files

To upload multiple files (all with different field name keys), use the `FileFieldsInterceptor()` decorator. This decorator takes two arguments:

- `uploadedFields`: an array of objects, where each object specifies a required `name` property with a string value specifying a field name, as described above, and an optional `maxCount` property, as described above
- `options`: optional `MulterOptions` object, as described above

When using `FileFieldsInterceptor()`, extract files from the `request` with the `@UploadedFiles()` decorator.

```
@@filename()
@Post('upload')
@UseInterceptors(FileFieldsInterceptor([
  { name: 'avatar', maxCount: 1 },
  { name: 'background', maxCount: 1 },
]))
uploadFile(@UploadedFiles() files: { avatar?: Express.Multer.File[], background?: Express.Multer.File[] }) {
  console.log(files);
}
@@switch
@Post('upload')
@Bind(UploadedFiles())
@UseInterceptors(FileFieldsInterceptor([
  { name: 'avatar', maxCount: 1 },
  { name: 'background', maxCount: 1 },
]))
uploadFile(files) {
  console.log(files);
}
```

## Any files

To upload all fields with arbitrary field name keys, use the `AnyFilesInterceptor()` decorator. This decorator can accept an optional `options` object as described above.

When using `AnyFilesInterceptor()`, extract files from the `request` with the `@UploadedFiles()` decorator.

```
@@filename()
@Post('upload')
@UseInterceptors(AnyFilesInterceptor())
uploadFile(@UploadedFiles() files: Array<Express.Multer.File>) {
  console.log(files);
}

@@switch
@Post('upload')
@Bind(UploadedFiles())
@UseInterceptors(AnyFilesInterceptor())
uploadFile(files) {
  console.log(files);
}
```

## No files

To accept `multipart/form-data` but not allow any files to be uploaded, use the `NoFilesInterceptor`. This sets multipart data as attributes on the request body. Any files sent with the request will throw a `BadRequestException`.

```
@Post('upload')
@UseInterceptors(NoFilesInterceptor())
handleMultiPartData(@Body() body) {
  console.log(body)
}
```

## Default options

You can specify multer options in the file interceptors as described above. To set default options, you can call the static `register()` method when you import the `MulterModule`, passing in supported options. You can use all options listed [here](#).

```
MulterModule.register({
  dest: './upload',
});
```

**info Hint** The `MulterModule` class is exported from the `@nestjs/platform-express` package.

## Async configuration

When you need to set `MulterModule` options asynchronously instead of statically, use the `registerAsync()` method. As with most dynamic modules, Nest provides several techniques to deal with async configuration.

One technique is to use a factory function:

```
MulterModule.registerAsync({
  useFactory: () => ({
    dest: './upload',
  }),
});
```

Like other [factory providers](#), our factory function can be `async` and can inject dependencies through `inject`.

```
MulterModule.registerAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    dest: configService.get<string>('MULTER_DEST'),
  }),
  inject: [ConfigService],
});
```

Alternatively, you can configure the `MulterModule` using a class instead of a factory, as shown below:

```
MulterModule.registerAsync({
  useClass: MulterConfigService,
});
```

The construction above instantiates `MulterConfigService` inside `MulterModule`, using it to create the required options object. Note that in this example, the `MulterConfigService` has to implement the `MulterOptionsFactory` interface, as shown below. The `MulterModule` will call the `createMulterOptions()` method on the instantiated object of the supplied class.

```
@Injectable()
class MulterConfigService implements MulterOptionsFactory {
  createMulterOptions(): MulterModuleOptions {
    return {
      dest: './upload',
    };
  }
}
```

If you want to reuse an existing options provider instead of creating a private copy inside the `MulterModule`, use the `useExisting` syntax.

```
MulterModule.registerAsync({
  imports: [ConfigModule],
```

```
useExisting: ConfigService,  
});
```

## Example

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

## Streaming files

**info Note** This chapter shows how you can stream files from your **HTTP application**. The examples presented below do not apply to GraphQL or Microservice applications.

There may be times where you would like to send back a file from your REST API to the client. To do this with Nest, normally you'd do the following:

```
@Controller('file')
export class FileController {
  @Get()
  getFile(@Res() res: Response) {
    const file = createReadStream(join(process.cwd(), 'package.json'));
    file.pipe(res);
  }
}
```

But in doing so you end up losing access to your post-controller interceptor logic. To handle this, you can return a **StreamableFile** instance and under the hood, the framework will take care of piping the response.

### Streamable File class

A **StreamableFile** is a class that holds onto the stream that is to be returned. To create a new **StreamableFile**, you can pass either a **Buffer** or a **Stream** to the **StreamableFile** constructor.

**info hint** The **StreamableFile** class can be imported from [@nestjs/common](#).

### Cross-platform support

Fastify, by default, can support sending files without needing to call **stream.pipe(res)**, so you don't need to use the **StreamableFile** class at all. However, Nest supports the use of **StreamableFile** in both platform types, so if you end up switching between Express and Fastify there's no need to worry about compatibility between the two engines.

### Example

You can find a simple example of returning the **package.json** as a file instead of a JSON below, but the idea extends out naturally to images, documents, and any other file type.

```
import { Controller, Get, StreamableFile } from '@nestjs/common';
import { createReadStream } from 'fs';
import { join } from 'path';

@Controller('file')
export class FileController {
  @Get()
```

```
getFile(): StreamableFile {
  const file = createReadStream(join(process.cwd(), 'package.json'));
  return new StreamableFile(file);
}

}
```

The default content type is `application/octet-stream`, if you need to customize the response you can use the `res.set` method or the `@Header()` decorator, like this:

```
import { Controller, Get, StreamableFile, Res } from '@nestjs/common';
import { createReadStream } from 'fs';
import { join } from 'path';
import type { Response } from 'express';

@Controller('file')
export class FileController {
  @Get()
  getFile(@Res({ passthrough: true }) res: Response): StreamableFile {
    const file = createReadStream(join(process.cwd(), 'package.json'));
    res.set({
      'Content-Type': 'application/json',
      'Content-Disposition': 'attachment; filename="package.json"',
    });
    return new StreamableFile(file);
  }

  // Or even:
  @Get()
  @Header('Content-Type', 'application/json')
  @Header('Content-Disposition', 'attachment; filename="package.json"')
  getStaticFile(): StreamableFile {
    const file = createReadStream(join(process.cwd(), 'package.json'));
    return new StreamableFile(file);
  }
}
```

## HTTP module

Axios is richly featured HTTP client package that is widely used. Nest wraps Axios and exposes it via the built-in `HttpModule`. The `HttpModule` exports the `HttpService` class, which exposes Axios-based methods to perform HTTP requests. The library also transforms the resulting HTTP responses into `Observables`.

**info Hint** You can also use any general purpose Node.js HTTP client library directly, including `got` or `undici`.

### Installation

To begin using it, we first install required dependencies.

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

### Getting started

Once the installation process is complete, to use the `HttpService`, first import `HttpModule`.

```
@Module({
  imports: [HttpModule],
  providers: [CatsService],
})
export class CatsModule {}
```

Next, inject `HttpService` using normal constructor injection.

**info Hint** `HttpModule` and `HttpService` are imported from `@nestjs/axios` package.

```
@@filename()
@Injectable()
export class CatsService {
  constructor(private readonly httpService: HttpService) {}

  findAll(): Observable<AxiosResponse<Cat[]>> {
    return this.httpService.get('http://localhost:3000/cats');
  }
}
@@switch
@.Injectable()
@Dependencies(HttpService)
export class CatsService {
  constructor(httpService) {
    this.httpService = httpService;
  }
}
```

```
findAll() {
  return this.httpService.get('http://localhost:3000/cats');
}
}
```

info Hint `AxiosResponse` is an interface exported from the `axios` package (`$ npm i axios`).

All `HttpService` methods return an `AxiosResponse` wrapped in an `Observable` object.

## Configuration

`Axios` can be configured with a variety of options to customize the behavior of the `HttpService`. Read more about them [here](#). To configure the underlying Axios instance, pass an optional options object to the `register()` method of `HttpModule` when importing it. This options object will be passed directly to the underlying Axios constructor.

```
@Module({
  imports: [
    HttpModule.register({
      timeout: 5000,
      maxRedirects: 5,
    }),
  ],
  providers: [CatsService],
})
export class CatsModule {}
```

## Async configuration

When you need to pass module options asynchronously instead of statically, use the `registerAsync()` method. As with most dynamic modules, Nest provides several techniques to deal with async configuration.

One technique is to use a factory function:

```
HttpModule.registerAsync({
  useFactory: () => ({
    timeout: 5000,
    maxRedirects: 5,
  }),
});
```

Like other factory providers, our factory function can be `async` and can inject dependencies through `inject`.

```
HttpModule.registerAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    timeout: configService.get('HTTP_TIMEOUT'),
    maxRedirects: configService.get('HTTP_MAX_REDIRECTS'),
  }),
  inject: [ConfigService],
});
```

Alternatively, you can configure the `HttpModule` using a class instead of a factory, as shown below.

```
HttpModule.registerAsync({
  useClass: HttpConfigService,
});
```

The construction above instantiates `HttpConfigService` inside `HttpModule`, using it to create an options object. Note that in this example, the `HttpConfigService` has to implement `HttpModuleOptionsFactory` interface as shown below. The `HttpModule` will call the `createHttpOptions()` method on the instantiated object of the supplied class.

```
@Injectable()
class HttpConfigService implements HttpModuleOptionsFactory {
  createHttpOptions(): HttpModuleOptions {
    return {
      timeout: 5000,
      maxRedirects: 5,
    };
  }
}
```

If you want to reuse an existing options provider instead of creating a private copy inside the `HttpModule`, use the `useExisting` syntax.

```
HttpModule.registerAsync({
  imports: [ConfigModule],
  useExisting: HttpConfigService,
});
```

## Using Axios directly

If you think that `HttpModule.register`'s options are not enough for you, or if you just want to access the underlying Axios instance created by `@nestjs/axios`, you can access it via `HttpService#axiosRef` as follows:

```
@Injectable()
export class CatsService {
  constructor(private readonly httpService: HttpService) {}

  findAll(): Promise<AxiosResponse<Cat[]>> {
    return this.httpService.axiosRef.get('http://localhost:3000/cats');
    // ^ AxiosInstance interface
  }
}
```

## Full example

Since the return value of the `HttpService` methods is an Observable, we can use `rxjs - firstValueFrom` or `lastValueFrom` to retrieve the data of the request in the form of a promise.

```
import { catchError, firstValueFrom } from 'rxjs';

@Injectable()
export class CatsService {
  private readonly logger = new Logger(CatsService.name);
  constructor(private readonly httpService: HttpService) {}

  async findAll(): Promise<Cat[]> {
    const { data } = await firstValueFrom(
      this.httpService.get<Cat[]>('http://localhost:3000/cats').pipe(
        catchError((error: AxiosError) => {
          this.logger.error(error.response.data);
          throw 'An error happened!';
        })
    );
    return data;
  }
}
```

info **Hint** Visit RxJS's documentation on `firstValueFrom` and `lastValueFrom` for differences between them.

## Session

**HTTP sessions** provide a way to store information about the user across multiple requests, which is particularly useful for [MVC](#) applications.

### Use with Express (default)

First install the [required package](#) (and its types for TypeScript users):

```
$ npm i express-session
$ npm i -D @types/express-session
```

Once the installation is complete, apply the [express-session](#) middleware as global middleware (for example, in your [main.ts](#) file).

```
import * as session from 'express-session';
// somewhere in your initialization file
app.use(
  session({
    secret: 'my-secret',
    resave: false,
    saveUninitialized: false,
  }),
);
```

**warning** **Notice** The default server-side session storage is purposely not designed for a production environment. It will leak memory under most conditions, does not scale past a single process, and is meant for debugging and developing. Read more in the [official repository](#).

The [secret](#) is used to sign the session ID cookie. This can be either a string for a single secret, or an array of multiple secrets. If an array of secrets is provided, only the first element will be used to sign the session ID cookie, while all the elements will be considered when verifying the signature in requests. The secret itself should not be easily parsed by a human and would best be a random set of characters.

Enabling the [resave](#) option forces the session to be saved back to the session store, even if the session was never modified during the request. The default value is [true](#), but using the default has been deprecated, as the default will change in the future.

Likewise, enabling the [saveUninitialized](#) option Forces a session that is "uninitialized" to be saved to the store. A session is uninitialized when it is new but not modified. Choosing [false](#) is useful for implementing login sessions, reducing server storage usage, or complying with laws that require permission before setting a cookie. Choosing [false](#) will also help with race conditions where a client makes multiple parallel requests without a session ([source](#)).

You can pass several other options to the [session](#) middleware, read more about them in the [API documentation](#).

**info Hint** Please note that `secure: true` is a recommended option. However, it requires an https-enabled website, i.e., HTTPS is necessary for secure cookies. If `secure` is set, and you access your site over HTTP, the cookie will not be set. If you have your node.js behind a proxy and are using `secure: true`, you need to set "`trust proxy`" in express.

With this in place, you can now set and read session values from within the route handlers, as follows:

```
@Get()
findAll(@Req() request: Request) {
  request.session.visits = request.session.visits ? request.session.visits
+ 1 : 1;
}
```

**info Hint** The `@Req()` decorator is imported from the `@nestjs/common`, while `Request` from the `express` package.

Alternatively, you can use the `@Session()` decorator to extract a session object from the request, as follows:

```
@Get()
findAll(@Session() session: Record<string, any>) {
  session.visits = session.visits ? session.visits + 1 : 1;
}
```

**info Hint** The `@Session()` decorator is imported from the `@nestjs/common` package.

## Use with Fastify

First install the required package:

```
$ npm i @fastify/secure-session
```

Once the installation is complete, register the `fastify-secure-session` plugin:

```
import secureSession from '@fastify/secure-session';

// somewhere in your initialization file
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  new FastifyAdapter(),
);
await app.register(secureSession, {
  secret: 'averylogphrasebiggerthanthirtytwochars',
  salt: 'mq9hDxBVDbspDR6n',
});
```

**info Hint** You can also pregenerate a key ([see instructions](#)) or use [keys rotation](#).

Read more about the available options in the [official repository](#).

With this in place, you can now set and read session values from within the route handlers, as follows:

```
@Get()  
findAll(@Req() request: FastifyRequest) {  
  const visits = request.session.get('visits');  
  request.session.set('visits', visits ? visits + 1 : 1);  
}
```

Alternatively, you can use the `@Session()` decorator to extract a session object from the request, as follows:

```
@Get()  
findAll(@Session() session: secureSession.Session) {  
  const visits = session.get('visits');  
  session.set('visits', visits ? visits + 1 : 1);  
}
```

**info Hint** The `@Session()` decorator is imported from the `@nestjs/common`, while `secureSession.Session` from the `@fastify/secure-session` package (import statement: `import * as secureSession from '@fastify/secure-session'`).

## Model-View-Controller

Nest, by default, makes use of the [Express](#) library under the hood. Hence, every technique for using the MVC (Model-View-Controller) pattern in Express applies to Nest as well.

First, let's scaffold a simple Nest application using the [CLI](#) tool:

```
$ npm i -g @nestjs/cli
$ nest new project
```

In order to create an MVC app, we also need a [template engine](#) to render our HTML views:

```
$ npm install --save hbs
```

We've used the [hbs](#) ([Handlebars](#)) engine, though you can use whatever fits your requirements. Once the installation process is complete, we need to configure the express instance using the following code:

```
@@filename(main)
import { NestFactory } from '@nestjs/core';
import { NestExpressApplication } from '@nestjs/platform-express';
import { join } from 'path';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create<NestExpressApplication>(
    AppModule,
  );

  app.useStaticAssets(join(__dirname, '..', 'public'));
  app.setBaseViewsDir(join(__dirname, '..', 'views'));
  app.setViewEngine('hbs');

  await app.listen(3000);
}
bootstrap();
@@switch
import { NestFactory } from '@nestjs/core';
import { join } from 'path';
import { AppModule } from './app.module';

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

  app.useStaticAssets(join(__dirname, '..', 'public'));
  app.setBaseViewsDir(join(__dirname, '..', 'views'));
```

```
app.setViewEngine('hbs');

await app.listen(3000);
}

bootstrap();
```

We told [Express](#) that the `public` directory will be used for storing static assets, `views` will contain templates, and the `hbs` template engine should be used to render HTML output.

## Template rendering

Now, let's create a `views` directory and `index.hbs` template inside it. In the template, we'll print a `message` passed from the controller:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>App</title>
  </head>
  <body>
    {{ "{{ message }}\n" }}
  </body>
</html>
```

Next, open the `app.controller` file and replace the `root()` method with the following code:

```
@@filename(app.controller)
import { Get, Controller, Render } from '@nestjs/common';

@Controller()
export class AppController {
  @Get()
  @Render('index')
  root() {
    return { message: 'Hello world!' };
  }
}
```

In this code, we are specifying the template to use in the `@Render()` decorator, and the return value of the route handler method is passed to the template for rendering. Notice that the return value is an object with a property `message`, matching the `message` placeholder we created in the template.

While the application is running, open your browser and navigate to <http://localhost:3000>. You should see the `Hello world!` message.

## Dynamic template rendering

If the application logic must dynamically decide which template to render, then we should use the `@Res()` decorator, and supply the view name in our route handler, rather than in the `@Render()` decorator:

**info Hint** When Nest detects the `@Res()` decorator, it injects the library-specific `response` object. We can use this object to dynamically render the template. Learn more about the `response` object API [here](#).

```
@@filename(app.controller)
import { Get, Controller, Res, Render } from '@nestjs/common';
import { Response } from 'express';
import { AppService } from './app.service';

@Controller()
export class AppController {
    constructor(private appService: AppService) {}

    @Get()
    root(@Res() res: Response) {
        return res.render(
            this.appService.getViewName(),
            { message: 'Hello world!' },
        );
    }
}
```

## Example

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

## Fastify

As mentioned in this [chapter](#), we are able to use any compatible HTTP provider together with Nest. One such library is [Fastify](#). In order to create an MVC application with Fastify, we have to install the following packages:

```
$ npm i --save @fastify/static @fastify/view handlebars
```

The next steps cover almost the same process used with Express, with minor differences specific to the platform. Once the installation process is complete, open the `main.ts` file and update its contents:

```
@@filename(main)
import { NestFactory } from '@nestjs/core';
import { NestFastifyApplication, FastifyAdapter } from '@nestjs/platform-fastify';
import { AppModule } from './app.module';
import { join } from 'path';
```

```

async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    new FastifyAdapter(),
  );
  app.useStaticAssets({
    root: join(__dirname, '..', 'public'),
    prefix: '/public/',
  });
  app.setViewEngine({
    engine: {
      handlebars: require('handlebars'),
    },
    templates: join(__dirname, '..', 'views'),
  });
  await app.listen(3000);
}
bootstrap();
@@switch
import { NestFactory } from '@nestjs/core';
import { FastifyAdapter } from '@nestjs/platform-fastify';
import { AppModule } from './app.module';
import { join } from 'path';

async function bootstrap() {
  const app = await NestFactory.create(AppModule, new FastifyAdapter());
  app.useStaticAssets({
    root: join(__dirname, '..', 'public'),
    prefix: '/public/',
  });
  app.setViewEngine({
    engine: {
      handlebars: require('handlebars'),
    },
    templates: join(__dirname, '..', 'views'),
  });
  await app.listen(3000);
}
bootstrap();

```

The Fastify API is slightly different but the end result of those methods calls remains the same. One difference to notice with Fastify is that the template name passed into the `@Render()` decorator must include a file extension.

```

@@filename(app.controller)
import { Get, Controller, Render } from '@nestjs/common';

@Controller()
export class AppController {
  @Get()
  @Render('index.hbs')

```

```
root() {  
    return { message: 'Hello world!' };  
}  
}
```

While the application is running, open your browser and navigate to <http://localhost:3000>. You should see the **Hello world!** message.

## Example

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

## Performance (Fastify)

By default, Nest makes use of the [Express](#) framework. As mentioned earlier, Nest also provides compatibility with other libraries such as, for example, [Fastify](#). Nest achieves this framework independence by implementing a framework adapter whose primary function is to proxy middleware and handlers to appropriate library-specific implementations.

**info Hint** Note that in order for a framework adapter to be implemented, the target library has to provide similar request/response pipeline processing as found in Express.

[Fastify](#) provides a good alternative framework for Nest because it solves design issues in a similar manner to Express. However, fastify is much **faster** than Express, achieving almost two times better benchmarks results. A fair question is why does Nest use Express as the default HTTP provider? The reason is that Express is widely-used, well-known, and has an enormous set of compatible middleware, which is available to Nest users out-of-the-box.

But since Nest provides framework-independence, you can easily migrate between them. Fastify can be a better choice when you place high value on very fast performance. To utilize Fastify, simply choose the built-in [FastifyAdapter](#) as shown in this chapter.

### Installation

First, we need to install the required package:

```
$ npm i --save @nestjs/platform-fastify
```

### Adapter

Once the Fastify platform is installed, we can use the [FastifyAdapter](#).

```
@@filename(main)
import { NestFactory } from '@nestjs/core';
import {
  FastifyAdapter,
  NestFastifyApplication,
} from '@nestjs/platform-fastify';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    new FastifyAdapter()
  );
  await app.listen(3000);
}
bootstrap();
```

By default, Fastify listens only on the `localhost 127.0.0.1` interface ([read more](#)). If you want to accept connections on other hosts, you should specify '`0.0.0.0`' in the `listen()` call:

```
async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    new FastifyAdapter(),
  );
  await app.listen(3000, '0.0.0.0');
}
```

## Platform specific packages

Keep in mind that when you use the `FastifyAdapter`, Nest uses Fastify as the **HTTP provider**. This means that each recipe that relies on Express may no longer work. You should, instead, use Fastify equivalent packages.

## Redirect response

Fastify handles redirect responses slightly differently than Express. To do a proper redirect with Fastify, return both the status code and the URL, as follows:

```
@Get()
index(@Res() res) {
  res.status(302).redirect('/login');
}
```

## Fastify options

You can pass options into the Fastify constructor through the `FastifyAdapter` constructor. For example:

```
new FastifyAdapter({ logger: true });
```

## Middleware

Middleware functions retrieve the raw `req` and `res` objects instead of Fastify's wrappers. This is how the `middle` package works (that's used under the hood) and `fastify` - check out this [page](#) for more information,

```
@@filename(logger.middleware)
import { Injectable, NestMiddleware } from '@nestjs/common';
import { FastifyRequest, FastifyReply } from 'fastify';

@Injectable()
```

```
export class LoggerMiddleware implements NestMiddleware {
  use(req: FastifyRequest['raw'], res: FastifyReply['raw'], next: () =>
void) {
  console.log('Request...');
  next();
}
}

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

@Injectable()
export class LoggerMiddleware {
  use(req, res, next) {
    console.log('Request...');
    next();
  }
}
```

## Route Config

You can use the [route config](#) feature of Fastify with the `@RouteConfig()` decorator.

```
@RouteConfig({ output: 'hello world' })
@Get()
index(@Req() req) {
  return req.routeConfig.output;
}
```

**info Hint** `@RouteConfig()` is imported from [@nestjs/platform-fastify](#).

## Example

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

## Server-Sent Events

Server-Sent Events (SSE) is a server push technology enabling a client to receive automatic updates from a server via HTTP connection. Each notification is sent as a block of text terminated by a pair of newlines (learn more [here](#)).

### Usage

To enable Server-Sent events on a route (route registered within a **controller class**), annotate the method handler with the `@Sse()` decorator.

```
@Sse('sse')
sse(): Observable<MessageEvent> {
  return interval(1000).pipe(map(_ => ({ data: { hello: 'world' } })));
}
```

**info Hint** The `@Sse()` decorator and `MessageEvent` interface are imported from the `@nestjs/common`, while `Observable`, `interval`, and `map` are imported from the `rxjs` package.

**warning Warning** Server-Sent Events routes must return an `Observable` stream.

In the example above, we defined a route named `sse` that will allow us to propagate real-time updates. These events can be listened to using the [EventSource API](#).

The `sse` method returns an `Observable` that emits multiple `MessageEvent` (in this example, it emits a new `MessageEvent` every second). The `MessageEvent` object should respect the following interface to match the specification:

```
export interface MessageEvent {
  data: string | object;
  id?: string;
  type?: string;
  retry?: number;
}
```

With this in place, we can now create an instance of the `EventSource` class in our client-side application, passing the `/sse` route (which matches the endpoint we have passed into the `@Sse()` decorator above) as a constructor argument.

`EventSource` instance opens a persistent connection to an HTTP server, which sends events in `text/event-stream` format. The connection remains open until closed by calling `EventSource.close()`.

Once the connection is opened, incoming messages from the server are delivered to your code in the form of events. If there is an event field in the incoming message, the triggered event is the same as the event field value. If no event field is present, then a generic `message` event is fired ([source](#)).

```
const eventSource = new EventSource('/sse');
eventSource.onmessage = ({ data }) => {
  console.log('New message', JSON.parse(data));
};
```

## Example

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

## Authentication

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

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

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

### Creating an authentication module

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

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

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

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

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

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

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

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

```

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

async findOne(username: string): Promise<User | undefined> {
  return this.users.find(user => user.username === username);
}
}

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

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

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

```

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

```

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

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

```

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

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

## Implementing the "Sign in" endpoint

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

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

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

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

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

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

```

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

```

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

Now, we update our [AuthModule](#) to import the [UsersModule](#).

```

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

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

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

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

```

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

```

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

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

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

```

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

## JWT token

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

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

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

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

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

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

```

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

@Injectable()
export class AuthService {

```

```

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

@Module({
  imports: [
```

```

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

```

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

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

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

```

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

```

## Implementing the authentication guard

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

```

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

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

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

```

```

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

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

```

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

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

```

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

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

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

```

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

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

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

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

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

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

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

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

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

## Enable authentication globally

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

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

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

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

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

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

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

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

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

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

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

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

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

```
}

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

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

## Passport integration

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

To learn how you can integrate Passport with NestJS, check out this [chapter](#).

## Example

You can find a complete version of the code in this chapter [here](#).

## Authorization

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

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

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

### Basic RBAC implementation

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

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

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

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

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

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

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

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

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

```

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

```

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

```

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

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

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

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

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

```

```
}

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

**info Hint** Refer to the [Reflection and metadata](#) section of the Execution context chapter for more details on utilizing [Reflector](#) in a context-sensitive way.

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

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

To make sure this example works, your [User](#) class must look as follows:

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

Lastly, make sure to register the [RolesGuard](#), for example, at the controller level, or globally:

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

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

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

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

## Claims-based authorization

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

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

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

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

## Integrating CASL

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Then use it in a class as follows.

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

**info Hint** Learn more about the `Ability` class in the official [CASL documentation](#).

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

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

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

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

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

```

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

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

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

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

```

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

### Advanced: Implementing a PoliciesGuard

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

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

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

```

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

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

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

export type PolicyHandler = IPolicyHandler | PolicyHandlerCallback;

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

And use it as follows:

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

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

## Encryption and Hashing

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

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

### Encryption

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

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

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

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

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

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

Now to decrypt [encryptedText](#) value:

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

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

## Hashing

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

As an example, let's use [bcrypt](#) to hash a random password.

First install required packages:

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

Once the installation is complete, you can use the [hash](#) function, as follows:

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

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

To generate a salt, use the [genSalt](#) function:

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

To compare/check a password, use the [compare](#) function:

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

You can read more about available functions [here](#).

## Helmet

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

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

### Use with Express (default)

Start by installing the required package.

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

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

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

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

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

### Use with Fastify

If you are using the [FastifyAdapter](#), install the [@fastify/helmet](#) package:

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

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

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

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

```
await app.register(fastifyHelmet, {  
  contentSecurityPolicy: {  
    directives: {  
      defaultSrc: [`'self'`, 'unpkg.com'],  
      styleSrc: [  
        `'self'`,  
        `'unsafe-inline'`,  
        'cdn.jsdelivr.net',  
        'fonts.googleapis.com',  
        'unpkg.com',  
      ],  
      fontSrc: [`'self'`, 'fonts.gstatic.com', 'data:'],  
      imgSrc: [`'self'`, 'data:', 'cdn.jsdelivr.net'],  
      scriptSrc: [  
        `'self'`,  
        `https: 'unsafe-inline'`,  
        'cdn.jsdelivr.net',  
        `'unsafe-eval'`,  
      ],  
    },  
  },  
});  
  
// If you are not going to use CSP at all, you can use this:  
await app.register(fastifyHelmet, {  
  contentSecurityPolicy: false,  
});
```

## CORS

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

### Getting started

To enable CORS, call the [enableCors\(\)](#) method on the Nest application object.

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

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

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

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

## CSRF Protection

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

### Use with Express (default)

Start by installing the required package:

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

warning **Warning** This package is deprecated, refer to [csurf docs](#) for more information.

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

Once the installation is complete, apply the [csurf](#) middleware as global middleware.

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

### Use with Fastify

Start by installing the required package:

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

Once the installation is complete, register the [@fastify/csrf-protection](#) plugin, as follows:

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

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

## Rate Limiting

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

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

Once the installation is complete, the [ThrottlerModule](#) can be configured as any other Nest package with [forRoot](#) or [forRootAsync](#) methods.

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

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

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

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

## Customization

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

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

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

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

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

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

## Proxies

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

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

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

```
}

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

@UseGuards(ThrottlerBehindProxyGuard)
```

info **Hint** You can find the API of the `req` Request object for express [here](#) and for fastify [here](#).

## Websockets

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

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

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

        return true;
    }
}
```

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

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

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

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

## GraphQL

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

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

## Configuration

The following options are valid for the **ThrottlerModule**:

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

## Async Configuration

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

One approach would be to use a factory function:

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

You can also use the **useClass** syntax:

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

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

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

## Storage

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

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

**info Note** `ThrottlerStorage` can be imported from [@nestjs/throttler](#).

## Gateways

Most of the concepts discussed elsewhere in this documentation, such as dependency injection, decorators, exception filters, pipes, guards and interceptors, apply equally to gateways. Wherever possible, Nest abstracts implementation details so that the same components can run across HTTP-based platforms, WebSockets, and Microservices. This section covers the aspects of Nest that are specific to WebSockets.

In Nest, a gateway is simply a class annotated with `@WebSocketGateway()` decorator. Technically, gateways are platform-agnostic which makes them compatible with any WebSockets library once an adapter is created. There are two WS platforms supported out-of-the-box: `socket.io` and `ws`. You can choose the one that best suits your needs. Also, you can build your own adapter by following this [guide](#).



**info Hint** Gateways can be treated as [providers](#); this means they can inject dependencies through the class constructor. Also, gateways can be injected by other classes (providers and controllers) as well.

### Installation

To start building WebSockets-based applications, first install the required package:

```
@@filename()  
$ npm i --save @nestjs/websockets @nestjs/platform-socket.io  
@@switch  
$ npm i --save @nestjs/websockets @nestjs/platform-socket.io
```

### Overview

In general, each gateway is listening on the same port as the **HTTP server**, unless your app is not a web application, or you have changed the port manually. This default behavior can be modified by passing an argument to the `@WebSocketGateway(80)` decorator where `80` is a chosen port number. You can also set a [namespace](#) used by the gateway using the following construction:

```
@WebSocketGateway(80, { namespace: 'events' })
```

**warning Warning** Gateways are not instantiated until they are referenced in the providers array of an existing module.

You can pass any supported [option](#) to the socket constructor with the second argument to the `@WebSocketGateway()` decorator, as shown below:

```
@WebSocketGateway(81, { transports: ['websocket'] })
```

The gateway is now listening, but we have not yet subscribed to any incoming messages. Let's create a handler that will subscribe to the `events` messages and respond to the user with the exact same data.

```
@@filename(events.gateway)
@SubscribeMessage('events')
handleEvent(@MessageBody() data: string): string {
  return data;
}
@@switch
@Bind(MessageBody())
@SubscribeMessage('events')
handleEvent(data) {
  return data;
}
```

**info Hint** `@SubscribeMessage()` and `@MessageBody()` decorators are imported from `@nestjs/websockets` package.

Once the gateway is created, we can register it in our module.

```
import { Module } from '@nestjs/common';
import { EventsGateway } from './events.gateway';

@@filename(events.module)
@Module({
  providers: [EventsGateway]
})
export class EventsModule {}
```

You can also pass in a property key to the decorator to extract it from the incoming message body:

```
@@filename(events.gateway)
@SubscribeMessage('events')
handleEvent(@MessageBody('id') id: number): number {
  // id === messageBody.id
  return id;
}
@@switch
@Bind(MessageBody('id'))
@SubscribeMessage('events')
handleEvent(id) {
  // id === messageBody.id
  return id;
}
```

If you would prefer not to use decorators, the following code is functionally equivalent:

```
@@filename(events.gateway)
@SubscribeMessage('events')
handleEvent(client: Socket, data: string): string {
  return data;
}
@@switch
@SubscribeMessage('events')
handleEvent(client, data) {
  return data;
}
```

In the example above, the `handleEvent()` function takes two arguments. The first one is a platform-specific `socket instance`, while the second one is the data received from the client. This approach is not recommended though, because it requires mocking the `socket` instance in each unit test.

Once the `events` message is received, the handler sends an acknowledgment with the same data that was sent over the network. In addition, it's possible to emit messages using a library-specific approach, for example, by making use of `client.emit()` method. In order to access a connected socket instance, use `@ConnectedSocket()` decorator.

```
@@filename(events.gateway)
@SubscribeMessage('events')
handleEvent(
  @MessageBody() data: string,
  @ConnectedSocket() client: Socket,
): string {
  return data;
}
@@switch
@Bind(MessageBody(), ConnectedSocket())
@SubscribeMessage('events')
handleEvent(data, client) {
  return data;
}
```

**info Hint** `@ConnectedSocket()` decorator is imported from `@nestjs/websockets` package.

However, in this case, you won't be able to leverage interceptors. If you don't want to respond to the user, you can simply skip the `return` statement (or explicitly return a "falsy" value, e.g. `undefined`).

Now when a client emits the message as follows:

```
socket.emit('events', { name: 'Nest' });
```

The `handleEvent()` method will be executed. In order to listen for messages emitted from within the above handler, the client has to attach a corresponding acknowledgment listener:

```
socket.emit('events', { name: 'Nest' }, (data) => console.log(data));
```

## Multiple responses

The acknowledgment is dispatched only once. Furthermore, it is not supported by native WebSockets implementation. To solve this limitation, you may return an object which consists of two properties. The **event** which is a name of the emitted event and the **data** that has to be forwarded to the client.

```
@@filename(events.gateway)
@SubscribeMessage('events')
handleEvent(@MessageBody() data: unknown): WsResponse<unknown> {
  const event = 'events';
  return { event, data };
}

@switch
@Bind(MessageBody())
@SubscribeMessage('events')
handleEvent(data) {
  const event = 'events';
  return { event, data };
}
```

**info Hint** The **WsResponse** interface is imported from [@nestjs/websockets](#) package.

**warning Warning** You should return a class instance that implements **WsResponse** if your **data** field relies on **ClassSerializerInterceptor**, as it ignores plain JavaScript object responses.

In order to listen for the incoming response(s), the client has to apply another event listener.

```
socket.on('events', (data) => console.log(data));
```

## Asynchronous responses

Message handlers are able to respond either synchronously or **asynchronously**. Hence, **async** methods are supported. A message handler is also able to return an **Observable**, in which case the result values will be emitted until the stream is completed.

```
@@filename(events.gateway)
@SubscribeMessage('events')
onEvent(@MessageBody() data: unknown): Observable<WsResponse<number>> {
  const event = 'events';
  const response = [1, 2, 3];

  return from(response).pipe(
    map(data => ({ event, data })),
  );
}
```

```

    );
}

@@switch
@Bind(MessageBody())
@SubscribeMessage('events')
onEvent(data) {
  const event = 'events';
  const response = [1, 2, 3];

  return from(response).pipe(
    map(data => ({ event, data })),
  );
}

```

In the example above, the message handler will respond **3 times** (with each item from the array).

## Lifecycle hooks

There are 3 useful lifecycle hooks available. All of them have corresponding interfaces and are described in the following table:

<code>OnGatewayInit</code>	Forces to implement the <code>afterInit()</code> method. Takes library-specific server instance as an argument (and spreads the rest if required).
<code>OnGatewayConnection</code>	Forces to implement the <code>handleConnection()</code> method. Takes library-specific client socket instance as an argument.
<code>OnGatewayDisconnect</code>	Forces to implement the <code>handleDisconnect()</code> method. Takes library-specific client socket instance as an argument.

**info Hint** Each lifecycle interface is exposed from `@nestjs/websockets` package.

## Server

Occasionally, you may want to have a direct access to the native, **platform-specific** server instance. The reference to this object is passed as an argument to the `afterInit()` method (`OnGatewayInit` interface). Another option is to use the `@WebSocketServer()` decorator.

```

@WebSocketServer()
server: Server;

```

**warning Notice** The `@WebSocketServer()` decorator is imported from the `@nestjs/websockets` package.

Nest will automatically assign the server instance to this property once it is ready to use.

## Example

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



## Exception filters

The only difference between the HTTP **exception filter** layer and the corresponding web sockets layer is that instead of throwing `HttpException`, you should use `WsException`.

```
throw new WsException('Invalid credentials.');
```

**info Hint** The `WsException` class is imported from the `@nestjs/websockets` package.

With the sample above, Nest will handle the thrown exception and emit the `exception` message with the following structure:

```
{
  status: 'error',
  message: 'Invalid credentials.'
}
```

## Filters

Web sockets exception filters behave equivalently to HTTP exception filters. The following example uses a manually instantiated method-scoped filter. Just as with HTTP based applications, you can also use gateway-scoped filters (i.e., prefix the gateway class with a `@UseFilters()` decorator).

```
@UseFilters(new WsExceptionFilter())
@SubscribeMessage('events')
onEvent(client, data: any): WsResponse<any> {
  const event = 'events';
  return { event, data };
}
```

## Inheritance

Typically, you'll create fully customized exception filters crafted to fulfill your application requirements. However, there might be use-cases when you would like to simply extend the **core exception filter**, and override the behavior based on certain factors.

In order to delegate exception processing to the base filter, you need to extend `BaseWsExceptionFilter` and call the inherited `catch()` method.

```
@@filename()
import { Catch, ArgumentsHost } from '@nestjs/common';
import { BaseWsExceptionFilter } from '@nestjs/websockets';
```

```
@Catch()
export class AllExceptionsFilter extends BaseWsExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    super.catch(exception, host);
  }
}

@@switch
import { Catch } from '@nestjs/common';
import { BaseWsExceptionFilter } from '@nestjs/websockets';

@Catch()
export class AllExceptionsFilter extends BaseWsExceptionFilter {
  catch(exception, host) {
    super.catch(exception, host);
  }
}
```

The above implementation is just a shell demonstrating the approach. Your implementation of the extended exception filter would include your tailored **business logic** (e.g., handling various conditions).

## Pipes

There is no fundamental difference between [regular pipes](#) and web sockets pipes. The only difference is that instead of throwing [HttpException](#), you should use [WsException](#). In addition, all pipes will be only applied to the [data](#) parameter (because validating or transforming [client](#) instance is useless).

**info Hint** The [WsException](#) class is exposed from [@nestjs/websockets](#) package.

### Binding pipes

The following example uses a manually instantiated method-scoped pipe. Just as with HTTP based applications, you can also use gateway-scoped pipes (i.e., prefix the gateway class with a [@UsePipes\(\)](#) decorator).

```
@@filename()
@UsePipes(new ValidationPipe())
@SubscribeMessage('events')
handleEvent(client: Client, data: unknown): WsResponse<unknown> {
  const event = 'events';
  return { event, data };
}
@@switch
@UsePipes(new ValidationPipe())
@SubscribeMessage('events')
handleEvent(client, data) {
  const event = 'events';
  return { event, data };
}
```

## Guards

There is no fundamental difference between web sockets guards and regular HTTP application guards. The only difference is that instead of throwing `HttpException`, you should use `WsException`.

**info Hint** The `WsException` class is exposed from `@nestjs/websockets` package.

### Binding guards

The following example uses a method-scoped guard. Just as with HTTP based applications, you can also use gateway-scoped guards (i.e., prefix the gateway class with a `@UseGuards()` decorator).

```
@@filename()
@UseGuards(AuthGuard)
@SubscribeMessage('events')
handleEvent(client: Client, data: unknown): WsResponse<unknown> {
  const event = 'events';
  return { event, data };
}
@@switch
@UseGuards(AuthGuard)
@SubscribeMessage('events')
handleEvent(client, data) {
  const event = 'events';
  return { event, data };
}
```

## Interceptors

There is no difference between [regular interceptors](#) and web sockets interceptors. The following example uses a manually instantiated method-scoped interceptor. Just as with HTTP based applications, you can also use gateway-scoped interceptors (i.e., prefix the gateway class with a `@UseInterceptors()` decorator).

```
@@filename()
@UseInterceptors(new TransformInterceptor())
@SubscribeMessage('events')
handleEvent(client: Client, data: unknown): WsResponse<unknown> {
    const event = 'events';
    return { event, data };
}
@@switch
@UseInterceptors(new TransformInterceptor())
@SubscribeMessage('events')
handleEvent(client, data) {
    const event = 'events';
    return { event, data };
}
```

## Adapters

The WebSockets module is platform-agnostic, hence, you can bring your own library (or even a native implementation) by making use of `WebSocketAdapter` interface. This interface forces to implement few methods described in the following table:

<code>create</code>	Creates a socket instance based on passed arguments
<code>bindClientConnect</code>	Binds the client connection event
<code>bindClientDisconnect</code>	Binds the client disconnection event (optional*)
<code>bindMessageHandlers</code>	Binds the incoming message to the corresponding message handler
<code>close</code>	Terminates a server instance

## Extend socket.io

The `socket.io` package is wrapped in an `IoAdapter` class. What if you would like to enhance the basic functionality of the adapter? For instance, your technical requirements require a capability to broadcast events across multiple load-balanced instances of your web service. For this, you can extend `IoAdapter` and override a single method which responsibility is to instantiate new socket.io servers. But first of all, let's install the required package.

**warning** **Warning** To use socket.io with multiple load-balanced instances you either have to disable polling by setting `transports: ['websocket']` in your clients socket.io configuration or you have to enable cookie based routing in your load balancer. Redis alone is not enough. See [here](#) for more information.

```
$ npm i --save redis socket.io @socket.io/redis-adapter
```

Once the package is installed, we can create a `RedisIoAdapter` class.

```
import { IoAdapter } from '@nestjs/platform-socket.io';
import { ServerOptions } from 'socket.io';
import { createAdapter } from '@socket.io/redis-adapter';
import { createClient } from 'redis';

export class RedisIoAdapter extends IoAdapter {
    private adapterConstructor: ReturnType<typeof createAdapter>;

    async connectToRedis(): Promise<void> {
        const pubClient = createClient({ url: `redis://localhost:6379` });
        const subClient = pubClient.duplicate();

        await Promise.all([pubClient.connect(), subClient.connect()]);

        this.adapterConstructor = createAdapter(pubClient, subClient);
    }
}
```

```
createI0Server(port: number, options?: ServerOptions): any {
  const server = super.createI0Server(port, options);
  server.adapter(this.adapterConstructor);
  return server;
}
}
```

Afterward, simply switch to your newly created Redis adapter.

```
const app = await NestFactory.create(AppModule);
const redisIoAdapter = new RedisIoAdapter(app);
await redisIoAdapter.connectToRedis();

app.useWebSocketAdapter(redisIoAdapter);
```

## Ws library

Another available adapter is a [WsAdapter](#) which in turn acts like a proxy between the framework and integrate blazing fast and thoroughly tested [ws](#) library. This adapter is fully compatible with native browser WebSockets and is far faster than socket.io package. Unluckily, it has significantly fewer functionalities available out-of-the-box. In some cases, you may just don't necessarily need them though.

**info Hint** [ws](#) library does not support namespaces (communication channels popularised by [socket.io](#)). However, to somehow mimic this feature, you can mount multiple [ws](#) servers on different paths (example: `@WebSocketGateway({{ '}' }} path: '/users' {{ '}' }})`).

In order to use [ws](#), we firstly have to install the required package:

```
$ npm i --save @nestjs/platform-ws
```

Once the package is installed, we can switch an adapter:

```
const app = await NestFactory.create(AppModule);
app.useWebSocketAdapter(new WsAdapter(app));
```

**info Hint** The [WsAdapter](#) is imported from [@nestjs/platform-ws](#).

## Advanced (custom adapter)

For demonstration purposes, we are going to integrate the [ws](#) library manually. As mentioned, the adapter for this library is already created and is exposed from the [@nestjs/platform-ws](#) package as a [WsAdapter](#) class. Here is how the simplified implementation could potentially look like:

```
@@filename(ws-adapter)
import * as WebSocket from 'ws';
import { WebSocketAdapter, INestApplicationContext } from
'@nestjs/common';
import { MessageMappingProperties } from '@nestjs/websockets';
import { Observable, fromEvent, EMPTY } from 'rxjs';
import { mergeMap, filter } from 'rxjs/operators';

export class WsAdapter implements WebSocketAdapter {
  constructor(private app: INestApplicationContext) {}

  create(port: number, options: any = {}): any {
    return new WebSocket.Server({ port, ...options });
  }

  bindClientConnect(server, callback: Function) {
    server.on('connection', callback);
  }

  bindMessageHandlers(
    client: WebSocket,
    handlers: MessageMappingProperties[],
    process: (data: any) => Observable<any>,
  ) {
    fromEvent(client, 'message')
      .pipe(
        mergeMap(data => this.bindMessageHandler(data, handlers,
process)),
        filter(result => result),
      )
      .subscribe(response => client.send(JSON.stringify(response)));
  }

  bindMessageHandler(
    buffer,
    handlers: MessageMappingProperties[],
    process: (data: any) => Observable<any>,
  ): Observable<any> {
    const message = JSON.parse(buffer.data);
    const messageHandler = handlers.find(
      handler => handler.message === message.event,
    );
    if (!messageHandler) {
      return EMPTY;
    }
    return process(messageHandler.callback(message.data));
  }

  close(server) {
    server.close();
  }
}
```

**info Hint** When you want to take advantage of `ws` library, use built-in `WsAdapter` instead of creating your own one.

Then, we can set up a custom adapter using `useWebSocketAdapter()` method:

```
@@filename(main)
const app = await NestFactory.create(AppModule);
app.useWebSocketAdapter(new WsAdapter(app));
```

## Example

A working example that uses `WsAdapter` is available [here](#).

## Overview

In addition to traditional (sometimes called monolithic) application architectures, Nest natively supports the microservice architectural style of development. Most of the concepts discussed elsewhere in this documentation, such as dependency injection, decorators, exception filters, pipes, guards and interceptors, apply equally to microservices. Wherever possible, Nest abstracts implementation details so that the same components can run across HTTP-based platforms, WebSockets, and Microservices. This section covers the aspects of Nest that are specific to microservices.

In Nest, a microservice is fundamentally an application that uses a different **transport** layer than HTTP.



Nest supports several built-in transport layer implementations, called **transporters**, which are responsible for transmitting messages between different microservice instances. Most transporters natively support both **request-response** and **event-based** message styles. Nest abstracts the implementation details of each transporter behind a canonical interface for both request-response and event-based messaging. This makes it easy to switch from one transport layer to another -- for example to leverage the specific reliability or performance features of a particular transport layer -- without impacting your application code.

## Installation

To start building microservices, first install the required package:

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

## Getting started

To instantiate a microservice, use the `createMicroservice()` method of the `NestFactory` class:

```
@@filename(main)
import { NestFactory } from '@nestjs/core';
import { Transport, MicroserviceOptions } from '@nestjs/microservices';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.TCP,
    },
  );
  await app.listen();
}
bootstrap();
@@switch
import { NestFactory } from '@nestjs/core';
import { Transport } from '@nestjs/microservices';
```

```
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.createMicroservice(AppModule, {
    transport: Transport.TCP,
  });
  await app.listen();
}

bootstrap();
```

**info Hint** Microservices use the **TCP** transport layer by default.

The second argument of the `createMicroservice()` method is an `options` object. This object may consist of two members:

`transport` Specifies the transporter (for example, `Transport.NATS`)

`options` A transporter-specific options object that determines transporter behavior

The `options` object is specific to the chosen transporter. The **TCP** transporter exposes the properties described below. For other transporters (e.g, Redis, MQTT, etc.), see the relevant chapter for a description of the available options.

<code>host</code>	Connection hostname
<code>port</code>	Connection port
<code>retryAttempts</code>	Number of times to retry message (default: <code>0</code> )
<code>retryDelay</code>	Delay between message retry attempts (ms) (default: <code>0</code> )

## Patterns

Microservices recognize both messages and events by **patterns**. A pattern is a plain value, for example, a literal object or a string. Patterns are automatically serialized and sent over the network along with the data portion of a message. In this way, message senders and consumers can coordinate which requests are consumed by which handlers.

## Request-response

The request-response message style is useful when you need to **exchange** messages between various external services. With this paradigm, you can be certain that the service has actually received the message (without the need to manually implement a message ACK protocol). However, the request-response paradigm is not always the best choice. For example, streaming transporters that use log-based persistence, such as [Kafka](#) or [NATS streaming](#), are optimized for solving a different range of issues, more aligned with an event messaging paradigm (see [event-based messaging](#) below for more details).

To enable the request-response message type, Nest creates two logical channels - one is responsible for transferring the data while the other waits for incoming responses. For some underlying transports, such as [NATS](#), this dual-channel support is provided out-of-the-box. For others, Nest compensates by manually

creating separate channels. There can be overhead for this, so if you do not require a request-response message style, you should consider using the event-based method.

To create a message handler based on the request-response paradigm use the `@MessagePattern()` decorator, which is imported from the `@nestjs/microservices` package. This decorator should be used only within the `controller` classes since they are the entry points for your application. Using them inside providers won't have any effect as they are simply ignored by Nest runtime.

```
@@filename(math.controller)
import { Controller } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';

@Controller()
export class MathController {
  @MessagePattern({ cmd: 'sum' })
  accumulate(data: number[]): number {
    return (data || []).reduce((a, b) => a + b);
  }
}
@@switch
import { Controller } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';

@Controller()
export class MathController {
  @MessagePattern({ cmd: 'sum' })
  accumulate(data) {
    return (data || []).reduce((a, b) => a + b);
  }
}
```

In the above code, the `accumulate()` **message handler** listens for messages that fulfill the `{} {{ }} cmd: 'sum' {{ }} message pattern`. The message handler takes a single argument, the `data` passed from the client. In this case, the data is an array of numbers which are to be accumulated.

## Asynchronous responses

Message handlers are able to respond either synchronously or **asynchronously**. Hence, `async` methods are supported.

```
@@filename()
@MessagePattern({ cmd: 'sum' })
async accumulate(data: number[]): Promise<number> {
  return (data || []).reduce((a, b) => a + b);
}
@@switch
@MessagePattern({ cmd: 'sum' })
async accumulate(data) {
```

```
    return (data || []).reduce((a, b) => a + b);
}
```

A message handler is also able to return an `Observable`, in which case the result values will be emitted until the stream is completed.

```
@@filename()
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): Observable<number> {
  return from([1, 2, 3]);
}
@switch
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): Observable<number> {
  return from([1, 2, 3]);
}
```

In the example above, the message handler will respond **3 times** (with each item from the array).

## Event-based

While the request-response method is ideal for exchanging messages between services, it is less suitable when your message style is event-based - when you just want to publish **events** without waiting for a response. In that case, you do not want the overhead required by request-response for maintaining two channels.

Suppose you would like to simply notify another service that a certain condition has occurred in this part of the system. This is the ideal use case for the event-based message style.

To create an event handler, we use the `@EventPattern()` decorator, which is imported from the `@nestjs/microservices` package.

```
@@filename()
@EventPattern('user_created')
async handleUserCreated(data: Record<string, unknown>) {
  // business logic
}
@switch
@EventPattern('user_created')
async handleUserCreated(data) {
  // business logic
}
```

**info Hint** You can register multiple event handlers for a **single** event pattern and all of them will be automatically triggered in parallel.

The `handleUserCreated()` event handler listens for the '`user_created`' event. The event handler takes a single argument, the `data` passed from the client (in this case, an event payload which has been sent over the network).

## Decorators

In more sophisticated scenarios, you may want to access more information about the incoming request. For example, in the case of NATS with wildcard subscriptions, you may want to get the original subject that the producer has sent the message to. Likewise, in Kafka you may want to access the message headers. In order to accomplish that, you can use built-in decorators as follows:

```
@@filename()
@MessagePattern('time.us.*')
getDate(@Payload() data: number[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
@switch
@Bind(Payload(), Ctx())
@MessagePattern('time.us.*')
getDate(data, context) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
```

**info Hint** `@Payload()`, `@Ctx()` and `NatsContext` are imported from `@nestjs/microservices`.

**info Hint** You can also pass in a property key to the `@Payload()` decorator to extract a specific property from the incoming payload object, for example, `@Payload('id')`.

## Client

A client Nest application can exchange messages or publish events to a Nest microservice using the `ClientProxy` class. This class defines several methods, such as `send()` (for request-response messaging) and `emit()` (for event-driven messaging) that let you communicate with a remote microservice. Obtain an instance of this class in one of the following ways.

One technique is to import the `ClientsModule`, which exposes the static `register()` method. This method takes an argument which is an array of objects representing microservice transporters. Each such object has a `name` property, an optional `transport` property (default is `Transport.TCP`), and an optional transporter-specific `options` property.

The `name` property serves as an **injection token** that can be used to inject an instance of a `ClientProxy` where needed. The value of the `name` property, as an injection token, can be an arbitrary string or JavaScript symbol, as described [here](#).

The `options` property is an object with the same properties we saw in the `createMicroservice()` method earlier.

```
@Module({
  imports: [
    ClientsModule.register([
      { name: 'MATH_SERVICE', transport: Transport.TCP },
    ]),
  ],
  ...
})
```

Once the module has been imported, we can inject an instance of the `ClientProxy` configured as specified via the `'MATH_SERVICE'` transporter options shown above, using the `@Inject()` decorator.

```
constructor(
  @Inject('MATH_SERVICE') private client: ClientProxy,
) {}
```

**info Hint** The `ClientsModule` and `ClientProxy` classes are imported from the `@nestjs/microservices` package.

At times we may need to fetch the transporter configuration from another service (say a `ConfigService`), rather than hard-coding it in our client application. To do this, we can register a `custom provider` using the `ClientProxyFactory` class. This class has a static `create()` method, which accepts a transporter options object, and returns a customized `ClientProxy` instance.

```
@Module({
  providers: [
    {
      provide: 'MATH_SERVICE',
      useFactory: (configService: ConfigService) => {
        const mathSvcOptions = configService.getMathSvcOptions();
        return ClientProxyFactory.create(mathSvcOptions);
      },
      inject: [ConfigService],
    }
  ],
  ...
})
```

**info Hint** The `ClientProxyFactory` is imported from the `@nestjs/microservices` package.

Another option is to use the `@Client()` property decorator.

```
@Client({ transport: Transport.TCP })
client: ClientProxy;
```

**info Hint** The `@Client()` decorator is imported from the `@nestjs/microservices` package.

Using the `@Client()` decorator is not the preferred technique, as it is harder to test and harder to share a client instance.

The `ClientProxy` is **lazy**. It doesn't initiate a connection immediately. Instead, it will be established before the first microservice call, and then reused across each subsequent call. However, if you want to delay the application bootstrapping process until a connection is established, you can manually initiate a connection using the `ClientProxy` object's `connect()` method inside the `OnApplicationBootstrap` lifecycle hook.

```
@@filename()
async onApplicationBootstrap() {
  await this.client.connect();
}
```

If the connection cannot be created, the `connect()` method will reject with the corresponding error object.

## Sending messages

The `ClientProxy` exposes a `send()` method. This method is intended to call the microservice and returns an `Observable` with its response. Thus, we can subscribe to the emitted values easily.

```
@@filename()
accumulate(): Observable<number> {
  const pattern = { cmd: 'sum' };
  const payload = [1, 2, 3];
  return this.client.send<number>(pattern, payload);
}
@@switch
accumulate() {
  const pattern = { cmd: 'sum' };
  const payload = [1, 2, 3];
  return this.client.send(pattern, payload);
}
```

The `send()` method takes two arguments, `pattern` and `payload`. The `pattern` should match one defined in a `@MessagePattern()` decorator. The `payload` is a message that we want to transmit to the remote microservice. This method returns a **cold Observable**, which means that you have to explicitly subscribe to it before the message will be sent.

## Publishing events

To send an event, use the `ClientProxy` object's `emit()` method. This method publishes an event to the message broker.

```

@@filename()
async publish() {
  this.client.emit<number>('user_created', new UserCreatedEvent());
}

@@switch
async publish() {
  this.client.emit('user_created', new UserCreatedEvent());
}

```

The `emit()` method takes two arguments, `pattern` and `payload`. The `pattern` should match one defined in an `@EventPattern()` decorator. The `payload` is an event payload that we want to transmit to the remote microservice. This method returns a **hot Observable** (unlike the **cold Observable** returned by `send()`), which means that whether or not you explicitly subscribe to the observable, the proxy will immediately try to deliver the event.

## Scopes

For people coming from different programming language backgrounds, it might be unexpected to learn that in Nest, almost everything is shared across incoming requests. We have a connection pool to the database, singleton services with global state, etc. Remember that Node.js doesn't follow the request/response Multi-Threaded Stateless Model in which every request is processed by a separate thread. Hence, using singleton instances is fully **safe** for our applications.

However, there are edge-cases when request-based lifetime of the handler may be the desired behavior, for instance per-request caching in GraphQL applications, request tracking or multi-tenancy. Learn how to control scopes [here](#).

Request-scoped handlers and providers can inject `RequestContext` using the `@Inject()` decorator in combination with `CONTEXT` token:

```

import { Injectable, Scope, Inject } from '@nestjs/common';
import { CONTEXT, RequestContext } from '@nestjs/microservices';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {
  constructor(@Inject(CONTEXT) private ctx: RequestContext) {}
}

```

This provides access to the `RequestContext` object, which has two properties:

```

export interface RequestContext<T = any> {
  pattern: string | Record<string, any>;
  data: T;
}

```

The `data` property is the message payload sent by the message producer. The `pattern` property is the pattern used to identify an appropriate handler to handle the incoming message.

## Handling timeouts

In distributed systems, sometimes microservices might be down or not available. To avoid infinitely long waiting, you can use Timeouts. A timeout is an incredibly useful pattern when communicating with other services. To apply timeouts to your microservice calls, you can use the RxJS `timeout` operator. If the microservice does not respond to the request within a certain time, an exception is thrown, which can be caught and handled appropriately.

To solve this problem you have to use `rxjs` package. Just use the `timeout` operator in the pipe:

```
@@filename()
this.client
  .send<TResult, TInput>(pattern, data)
  .pipe(timeout(5000));

@@switch
this.client
  .send(pattern, data)
  .pipe(timeout(5000));
```

**info Hint** The `timeout` operator is imported from the `rxjs/operators` package.

After 5 seconds, if the microservice isn't responding, it will throw an error.

## Redis

The [Redis](#) transporter implements the publish/subscribe messaging paradigm and leverages the [Pub/Sub](#) feature of Redis. Published messages are categorized in channels, without knowing what subscribers (if any) will eventually receive the message. Each microservice can subscribe to any number of channels. In addition, more than one channel can be subscribed to at a time. Messages exchanged through channels are **fire-and-forget**, which means that if a message is published and there are no subscribers interested in it, the message is removed and cannot be recovered. Thus, you don't have a guarantee that either messages or events will be handled by at least one service. A single message can be subscribed to (and received) by multiple subscribers.



## Installation

To start building Redis-based microservices, first install the required package:

```
$ npm i --save ioredis
```

## Overview

To use the Redis transporter, pass the following options object to the [createMicroservice\(\)](#) method:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.REDIS,
  options: {
    host: 'localhost',
    port: 6379,
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.REDIS,
  options: {
    host: 'localhost',
    port: 6379,
  },
});
```

**info Hint** The [Transport](#) enum is imported from the [@nestjs/microservices](#) package.

## Options

The `options` property is specific to the chosen transporter. The **Redis** transporter exposes the properties described below.

<code>host</code>	Connection url
<code>port</code>	Connection port
<code>retryAttempts</code>	Number of times to retry message (default: <code>0</code> )
<code>retryDelay</code>	Delay between message retry attempts (ms) (default: <code>0</code> )
<code>wildcards</code>	Enables Redis wildcard subscriptions, instructing transporter to use <code>psubscribe/pmessage</code> under the hood. (default: <code>false</code> )

All the properties supported by the official [ioredis](#) client are also supported by this transporter.

## Client

Like other microservice transporters, you have [several options](#) for creating a Redis `ClientProxy` instance.

One method for creating an instance is to use the `ClientsModule`. To create a client instance with the `ClientsModule`, import it and use the `register()` method to pass an options object with the same properties shown above in the `createMicroservice()` method, as well as a `name` property to be used as the injection token. Read more about `ClientsModule` [here](#).

```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'MATH_SERVICE',
        transport: Transport.REDIS,
        options: {
          host: 'localhost',
          port: 6379,
        }
      },
    ]),
  ],
  ...
})
```

Other options to create a client (either `ClientProxyFactory` or `@Client()`) can be used as well. You can read about them [here](#).

## Context

In more sophisticated scenarios, you may want to access more information about the incoming request. When using the Redis transporter, you can access the `RedisContext` object.

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RedisContext)
{
  console.log(`Channel: ${context.getChannel()}`);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(`Channel: ${context.getChannel()}`);
}
```

info Hint `@Payload()`, `@Ctx()` and `RedisContext` are imported from the `@nestjs/microservices` package.

## MQTT

[MQTT](#) (Message Queuing Telemetry Transport) is an open source, lightweight messaging protocol, optimized for low latency. This protocol provides a scalable and cost-efficient way to connect devices using a **publish/subscribe** model. A communication system built on MQTT consists of the publishing server, a broker and one or more clients. It is designed for constrained devices and low-bandwidth, high-latency or unreliable networks.

### Installation

To start building MQTT-based microservices, first install the required package:

```
$ npm i --save mqtt
```

### Overview

To use the MQTT transporter, pass the following options object to the `createMicroservice()` method:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.MQTT,
  options: {
    url: 'mqtt://localhost:1883',
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.MQTT,
  options: {
    url: 'mqtt://localhost:1883',
  },
});
```

**info Hint** The `Transport` enum is imported from the [@nestjs/microservices](#) package.

### Options

The `options` object is specific to the chosen transporter. The **MQTT** transporter exposes the properties described [here](#).

### Client

Like other microservice transporters, you have [several options](#) for creating a MQTT `ClientProxy` instance.

One method for creating an instance is to use the [ClientsModule](#). To create a client instance with the [ClientsModule](#), import it and use the `register()` method to pass an options object with the same properties shown above in the `createMicroservice()` method, as well as a `name` property to be used as the injection token. Read more about [ClientsModule](#) [here](#).

```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'MATH_SERVICE',
        transport: Transport.MQTT,
        options: {
          url: 'mqtt://localhost:1883',
        }
      },
    ]),
  ],
  ...
})
```

Other options to create a client (either [ClientProxyFactory](#) or [@Client\(\)](#)) can be used as well. You can read about them [here](#).

## Context

In more sophisticated scenarios, you may want to access more information about the incoming request. When using the MQTT transporter, you can access the [MqttContext](#) object.

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: MqttContext) {
  console.log(`Topic: ${context.getTopic()}`);
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(`Topic: ${context.getTopic()}`);
}
```

**info Hint** `@Payload()`, `@Ctx()` and `MqttContext` are imported from the [@nestjs/microservices](#) package.

To access the original mqtt `packet`, use the `getPacket()` method of the [MqttContext](#) object, as follows:

```
@@filename()
@MessagePattern('notifications')
```

```
getNotifications(@Payload() data: number[], @Ctx() context: MqttContext) {
  console.log(context.getPacket());
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(context.getPacket());
}
```

## Wildcards

A subscription may be to an explicit topic, or it may include wildcards. Two wildcards are available, `+` and `#`. `+` is a single-level wildcard, while `#` is a multi-level wildcard which covers many topic levels.

```
@@filename()
@MessagePattern('sensors/+/temperature/+')
getTemperature(@Ctx() context: MqttContext) {
  console.log(`Topic: ${context.getTopic()}`);
}
@@switch
@Bind(Ctx())
@MessagePattern('sensors/+/temperature/+')
getTemperature(context) {
  console.log(`Topic: ${context.getTopic()}`);
}
```

## Record builders

To configure message options (adjust the QoS level, set the Retain or DUP flags, or add additional properties to the payload), you can use the `MqttRecordBuilder` class. For example, to set `QoS` to `2` use the `setQoS` method, as follows:

```
const userProperties = { 'x-version': '1.0.0' };
const record = new MqttRecordBuilder(':cat:')
  .setProperties({ userProperties })
  .setQoS(1)
  .build();
client.send('replace-emoji', record).subscribe(...);
```

**info Hint** `MqttRecordBuilder` class is exported from the `@nestjs/microservices` package.

And you can read these options on the server-side as well, by accessing the `MqttContext`.

```
@@filename()
@MessagePattern('replace-emoji')
```

```
replaceEmoji(@Payload() data: string, @Ctx() context: MqttContext): string
{
  const { properties: { userProperties } } = context.getPacket();
  return userProperties['x-version'] === '1.0.0' ? '😺' : '😸';
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('replace-emoji')
replaceEmoji(data, context) {
  const { properties: { userProperties } } = context.getPacket();
  return userProperties['x-version'] === '1.0.0' ? '😺' : '😸';
}
```

In some cases you might want to configure user properties for multiple requests, you can pass these options to the [ClientProxyFactory](#).

```
import { Module } from '@nestjs/common';
import { ClientProxyFactory, Transport } from '@nestjs/microservices';

@Module({
  providers: [
    {
      provide: 'API_v1',
      useFactory: () =>
        ClientProxyFactory.create({
          transport: Transport.MQTT,
          options: {
            url: 'mqtt://localhost:1833',
            userProperties: { 'x-version': '1.0.0' },
          },
        }),
    },
  ],
})
export class ApiModule {}
```

## NATS

NATS is a simple, secure and high performance open source messaging system for cloud native applications, IoT messaging, and microservices architectures. The NATS server is written in the Go programming language, but client libraries to interact with the server are available for dozens of major programming languages. NATS supports both **At Most Once** and **At Least Once** delivery. It can run anywhere, from large servers and cloud instances, through edge gateways and even Internet of Things devices.

### Installation

To start building NATS-based microservices, first install the required package:

```
$ npm i --save nats
```

### Overview

To use the NATS transporter, pass the following options object to the `createMicroservice()` method:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.NATS,
  options: {
    servers: ['nats://localhost:4222'],
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.NATS,
  options: {
    servers: ['nats://localhost:4222'],
  },
});
```

**info Hint** The `Transport` enum is imported from the `@nestjs/microservices` package.

### Options

The `options` object is specific to the chosen transporter. The **NATS** transporter exposes the properties described [here](#). Additionally, there is a `queue` property which allows you to specify the name of the queue that your server should subscribe to (leave `undefined` to ignore this setting). Read more about NATS queue groups [below](#).

### Client

Like other microservice transporters, you have [several options](#) for creating a NATS [ClientProxy](#) instance.

One method for creating an instance is to use the [ClientsModule](#). To create a client instance with the [ClientsModule](#), import it and use the [register\(\)](#) method to pass an options object with the same properties shown above in the [createMicroservice\(\)](#) method, as well as a [name](#) property to be used as the injection token. Read more about [ClientsModule](#) [here](#).

```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'MATH_SERVICE',
        transport: Transport.NATS,
        options: {
          servers: ['nats://localhost:4222'],
        }
      },
    ]),
  ]
  ...
})
```

Other options to create a client (either [ClientProxyFactory](#) or [@Client\(\)](#)) can be used as well. You can read about them [here](#).

## Request-response

For the **request-response** message style ([read more](#)), the NATS transporter does not use the NATS built-in [Request-Reply](#) mechanism. Instead, a "request" is published on a given subject using the [publish\(\)](#) method with a unique reply subject name, and responders listen on that subject and send responses to the reply subject. Reply subjects are directed back to the requestor dynamically, regardless of location of either party.

## Event-based

For the **event-based** message style ([read more](#)), the NATS transporter uses NATS built-in [Publish-Subscribe](#) mechanism. A publisher sends a message on a subject and any active subscriber listening on that subject receives the message. Subscribers can also register interest in wildcard subjects that work a bit like a regular expression. This one-to-many pattern is sometimes called fan-out.

## Queue groups

NATS provides a built-in load balancing feature called [distributed queues](#). To create a queue subscription, use the [queue](#) property as follows:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
  (AppModule, {
```

```
transport: Transport.NATS,
options: {
  servers: ['nats://localhost:4222'],
  queue: 'cats_queue',
},
});
```

## Context

In more sophisticated scenarios, you may want to access more information about the incoming request. When using the NATS transporter, you can access the `NatsContext` object.

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`);
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(`Subject: ${context.getSubject()}`);
}
```

info Hint `@Payload()`, `@Ctx()` and `NatsContext` are imported from the `@nestjs/microservices` package.

## Wildcards

A subscription may be to an explicit subject, or it may include wildcards.

```
@@filename()
@MessagePattern('time.us.*')
getDate(@Payload() data: number[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('time.us.*')
getDate(data, context) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
```

## Record builders

To configure message options, you can use the `NatsRecordBuilder` class (note: this is doable for event-based flows as well). For example, to add `x-version` header, use the `setHeaders` method, as follows:

```
import * as nats from 'nats';

// somewhere in your code
const headers = nats.headers();
headers.set('x-version', '1.0.0');

const record = new NatsRecordBuilder(':cat:').setHeaders(headers).build();
this.client.send('replace-emoji', record).subscribe(...);
```

**info Hint** `NatsRecordBuilder` class is exported from the `@nestjs/microservices` package.

And you can read these headers on the server-side as well, by accessing the `NatsContext`, as follows:

```
@@filename()
@MessagePattern('replace-emoji')
replaceEmoji(@Payload() data: string, @Ctx() context: NatsContext): string
{
  const headers = context.getHeaders();
  return headers['x-version'] === '1.0.0' ? '😺' : '😸';
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('replace-emoji')
replaceEmoji(data, context) {
  const headers = context.getHeaders();
  return headers['x-version'] === '1.0.0' ? '😺' : '😸';
}
```

In some cases you might want to configure headers for multiple requests, you can pass these as options to the `ClientProxyFactory`:

```
import { Module } from '@nestjs/common';
import { ClientProxyFactory, Transport } from '@nestjs/microservices';

@Module({
  providers: [
    {
      provide: 'API_v1',
      useFactory: () =>
        ClientProxyFactory.create({
          transport: Transport.NATS,
          options: {
            servers: ['nats://localhost:4222'],
            headers: { 'x-version': '1.0.0' },
          },
        }),
    },
  ],
})
```

```
    },
  ],
})
export class ApiModule {}
```

## RabbitMQ

RabbitMQ is an open-source and lightweight message broker which supports multiple messaging protocols. It can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements. In addition, it's the most widely deployed message broker, used worldwide at small startups and large enterprises.

### Installation

To start building RabbitMQ-based microservices, first install the required packages:

```
$ npm i --save amqplib amqp-connection-manager
```

### Overview

To use the RabbitMQ transporter, pass the following options object to the `createMicroservice()` method:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.RMQ,
  options: {
    urls: ['amqp://localhost:5672'],
    queue: 'cats_queue',
    queueOptions: {
      durable: false
    },
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.RMQ,
  options: {
    urls: ['amqp://localhost:5672'],
    queue: 'cats_queue',
    queueOptions: {
      durable: false
    },
  },
});
```

**info Hint** The `Transport` enum is imported from the `@nestjs/microservices` package.

### Options

The **options** property is specific to the chosen transporter. The **RabbitMQ** transporter exposes the properties described below.

<b>urls</b>	Connection urls
<b>queue</b>	Queue name which your server will listen to
<b>prefetchCount</b>	Sets the prefetch count for the channel
<b>isGlobalPrefetchCount</b>	Enables per channel prefetching
<b>noAck</b>	If <b>false</b> , manual acknowledgment mode enabled
<b>queueOptions</b>	Additional queue options (read more <a href="#">here</a> )
<b>socketOptions</b>	Additional socket options (read more <a href="#">here</a> )
<b>headers</b>	Headers to be sent along with every message

## Client

Like other microservice transporters, you have [several options](#) for creating a RabbitMQ **ClientProxy** instance.

One method for creating an instance is to use the **ClientsModule**. To create a client instance with the **ClientsModule**, import it and use the **register()** method to pass an options object with the same properties shown above in the **createMicroservice()** method, as well as a **name** property to be used as the injection token. Read more about **ClientsModule** [here](#).

```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'MATH_SERVICE',
        transport: Transport.RMQ,
        options: {
          urls: ['amqp://localhost:5672'],
          queue: 'cats_queue',
          queueOptions: {
            durable: false
          },
        },
      },
    ]),
  ]
})
...
```

Other options to create a client (either **ClientProxyFactory** or **@Client()**) can be used as well. You can read about them [here](#).

## Context

In more sophisticated scenarios, you may want to access more information about the incoming request. When using the RabbitMQ transporter, you can access the `RmqContext` object.

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  console.log(`Pattern: ${context.getPattern()}`);
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(`Pattern: ${context.getPattern()}`);
}
```

info Hint `@Payload()`, `@Ctx()` and `RmqContext` are imported from the `@nestjs/microservices` package.

To access the original RabbitMQ message (with the `properties`, `fields`, and `content`), use the `getMessage()` method of the `RmqContext` object, as follows:

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  console.log(context.getMessage());
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(context.getMessage());
}
```

To retrieve a reference to the RabbitMQ [channel](#), use the `getChannelRef` method of the `RmqContext` object, as follows:

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  console.log(context.getChannelRef());
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(context.getChannelRef());
}
```

## Message acknowledgement

To make sure a message is never lost, RabbitMQ supports [message acknowledgements](#). An acknowledgement is sent back by the consumer to tell RabbitMQ that a particular message has been received, processed and that RabbitMQ is free to delete it. If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it.

To enable manual acknowledgment mode, set the `noAck` property to `false`:

```
options: {
  urls: ['amqp://localhost:5672'],
  queue: 'cats_queue',
  noAck: false,
  queueOptions: {
    durable: false
  },
},
```

When manual consumer acknowledgements are turned on, we must send a proper acknowledgement from the worker to signal that we are done with a task.

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  const channel = context.getChannelRef();
  const originalMsg = context.getMessage();

  channel.ack(originalMsg);
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  const channel = context.getChannelRef();
  const originalMsg = context.getMessage();

  channel.ack(originalMsg);
}
```

## Record builders

To configure message options, you can use the `RmqRecordBuilder` class (note: this is doable for event-based flows as well). For example, to set `headers` and `priority` properties, use the `setOptions` method, as follows:

```
const message = ':cat:';
const record = new RmqRecordBuilder(message)
  .setOptions({
    headers: {
      ['x-version']: '1.0.0',
    },
    priority: 3,
  })
  .build();

this.client.send('replace-emoji', record).subscribe(...);
```

info Hint `RmqRecordBuilder` class is exported from the `@nestjs/microservices` package.

And you can read these values on the server-side as well, by accessing the `RmqContext`, as follows:

```
@@filename()
@MessagePattern('replace-emoji')
replaceEmoji(@Payload() data: string, @Ctx() context: RmqContext): string
{
  const { properties: { headers } } = context.getMessage();
  return headers['x-version'] === '1.0.0' ? '😺' : '😸';
}

@switch
@Bind(Payload(), Ctx())
@MessagePattern('replace-emoji')
replaceEmoji(data, context) {
  const { properties: { headers } } = context.getMessage();
  return headers['x-version'] === '1.0.0' ? '😺' : '😸';
}
```

## Kafka

Kafka is an open source, distributed streaming platform which has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

The Kafka project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. It integrates very well with Apache Storm and Spark for real-time streaming data analysis.

## Installation

To start building Kafka-based microservices, first install the required package:

```
$ npm i --save kafkajs
```

## Overview

Like other Nest microservice transport layer implementations, you select the Kafka transporter mechanism using the `transport` property of the options object passed to the `createMicroservice()` method, along with an optional `options` property, as shown below:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.KAFKA,
  options: {
    client: {
      brokers: ['localhost:9092'],
    }
  }
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.KAFKA,
  options: {
    client: {
      brokers: ['localhost:9092'],
    }
  }
});
```

**info Hint** The `Transport` enum is imported from the `@nestjs/microservices` package.

## Options

The **options** property is specific to the chosen transporter. The **Kafka** transporter exposes the properties described below.

<b>client</b>	Client configuration options (read more <a href="#">here</a> )
<b>consumer</b>	Consumer configuration options (read more <a href="#">here</a> )
<b>run</b>	Run configuration options (read more <a href="#">here</a> )
<b>subscribe</b>	Subscribe configuration options (read more <a href="#">here</a> )
<b>producer</b>	Producer configuration options (read more <a href="#">here</a> )
<b>send</b>	Send configuration options (read more <a href="#">here</a> )
<b>producerOnlyMode</b>	Feature flag to skip consumer group registration and only act as a producer ( <b>boolean</b> )
<b>postfixId</b>	Change suffix of clientId value ( <b>string</b> )

## Client

There is a small difference in Kafka compared to other microservice transporters. Instead of the **ClientProxy** class, we use the **ClientKafka** class.

Like other microservice transporters, you have **several options** for creating a **ClientKafka** instance.

One method for creating an instance is to use the **ClientsModule**. To create a client instance with the **ClientsModule**, import it and use the **register()** method to pass an options object with the same properties shown above in the **createMicroservice()** method, as well as a **name** property to be used as the injection token. Read more about **ClientsModule** [here](#).

```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'HERO_SERVICE',
        transport: Transport.KAFKA,
        options: {
          client: {
            clientId: 'hero',
            brokers: ['localhost:9092'],
          },
          consumer: {
            groupId: 'hero-consumer'
          }
        }
      },
    ]),
  ...
})
```

Other options to create a client (either `ClientProxyFactory` or `@Client()`) can be used as well. You can read about them [here](#).

Use the `@Client()` decorator as follows:

```
@Client({
  transport: Transport.KAFKA,
  options: {
    client: {
      clientId: 'hero',
      brokers: ['localhost:9092'],
    },
    consumer: {
      groupId: 'hero-consumer'
    }
  }
})
client: ClientKafka;
```

## Message pattern

The Kafka microservice message pattern utilizes two topics for the request and reply channels. The `ClientKafka#send()` method sends messages with a [return address](#) by associating a [correlation id](#), reply topic, and reply partition with the request message. This requires the `ClientKafka` instance to be subscribed to the reply topic and assigned to at least one partition before sending a message.

Subsequently, you need to have at least one reply topic partition for every Nest application running. For example, if you are running 4 Nest applications but the reply topic only has 3 partitions, then 1 of the Nest applications will error out when trying to send a message.

When new `ClientKafka` instances are launched they join the consumer group and subscribe to their respective topics. This process triggers a rebalance of topic partitions assigned to consumers of the consumer group.

Normally, topic partitions are assigned using the round robin partitioner, which assigns topic partitions to a collection of consumers sorted by consumer names which are randomly set on application launch. However, when a new consumer joins the consumer group, the new consumer can be positioned anywhere within the collection of consumers. This creates a condition where pre-existing consumers can be assigned different partitions when the pre-existing consumer is positioned after the new consumer. As a result, the consumers that are assigned different partitions will lose response messages of requests sent before the rebalance.

To prevent the `ClientKafka` consumers from losing response messages, a Nest-specific built-in custom partitioner is utilized. This custom partitioner assigns partitions to a collection of consumers sorted by high-resolution timestamps (`process.hrtime()`) that are set on application launch.

## Message response subscription

**warning Note** This section is only relevant if you use [request-response](#) message style (with the `@MessagePattern` decorator and the `ClientKafka#send` method). Subscribing to the response

topic is not necessary for the `event-based` communication (`@EventPattern` decorator and `ClientKafka#emit` method).

The `ClientKafka` class provides the `subscribeToResponseOf()` method. The `subscribeToResponseOf()` method takes a request's topic name as an argument and adds the derived reply topic name to a collection of reply topics. This method is required when implementing the message pattern.

```
@@filename(heroes.controller)
onModuleInit() {
    this.client.subscribeToResponseOf('hero.kill.dragon');
}
```

If the `ClientKafka` instance is created asynchronously, the `subscribeToResponseOf()` method must be called before calling the `connect()` method.

```
@@filename(heroes.controller)
async onModuleInit() {
    this.client.subscribeToResponseOf('hero.kill.dragon');
    await this.client.connect();
}
```

## Incoming

Nest receives incoming Kafka messages as an object with `key`, `value`, and `headers` properties that have values of type `Buffer`. Nest then parses these values by transforming the buffers into strings. If the string is "object like", Nest attempts to parse the string as `JSON`. The `value` is then passed to its associated handler.

## Outgoing

Nest sends outgoing Kafka messages after a serialization process when publishing events or sending messages. This occurs on arguments passed to the `ClientKafka emit()` and `send()` methods or on values returned from a `@MessagePattern` method. This serialization "stringifies" objects that are not strings or buffers by using `JSON.stringify()` or the `toString()` prototype method.

```
@@filename(heroes.controller)
@Controller()
export class HeroesController {
    @MessagePattern('hero.kill.dragon')
    killDragon(@Payload() message: KillDragonMessage): any {
        const dragonId = message.dragonId;
        const items = [
            { id: 1, name: 'Mythical Sword' },
            { id: 2, name: 'Key to Dungeon' },
        ];
    }
}
```

```
        return items;
    }
}
```

info Hint `@Payload()` is imported from the [@nestjs/microservices](#).

Outgoing messages can also be keyed by passing an object with the `key` and `value` properties. Keying messages is important for meeting the [co-partitioning requirement](#).

```
@@filename(heroes.controller)
@Controller()
export class HeroesController {
  @MessagePattern('hero.kill.dragon')
  killDragon(@Payload() message: KillDragonMessage): any {
    const realm = 'Nest';
    const heroId = message.heroId;
    const dragonId = message.dragonId;

    const items = [
      { id: 1, name: 'Mythical Sword' },
      { id: 2, name: 'Key to Dungeon' },
    ];

    return {
      headers: {
        realm
      },
      key: heroId,
      value: items
    }
  }
}
```

Additionally, messages passed in this format can also contain custom headers set in the `headers` hash property. Header hash property values must be either of type `string` or type `Buffer`.

```
@@filename(heroes.controller)
@Controller()
export class HeroesController {
  @MessagePattern('hero.kill.dragon')
  killDragon(@Payload() message: KillDragonMessage): any {
    const realm = 'Nest';
    const heroId = message.heroId;
    const dragonId = message.dragonId;

    const items = [
      { id: 1, name: 'Mythical Sword' },
      { id: 2, name: 'Key to Dungeon' },
    ];
  }
}
```

```
    return {
      headers: {
        kafka_nestRealm: realm
      },
      key: heroId,
      value: items
    }
  }
}
```

## Event-based

While the request-response method is ideal for exchanging messages between services, it is less suitable when your message style is event-based (which in turn is ideal for Kafka) - when you just want to publish events **without waiting for a response**. In that case, you do not want the overhead required by request-response for maintaining two topics.

Check out these two sections to learn more about this: [Overview: Event-based](#) and [Overview: Publishing events](#).

## Context

In more sophisticated scenarios, you may want to access more information about the incoming request. When using the Kafka transporter, you can access the [KafkaContext](#) object.

```
@@filename()
@MessagePattern('hero.kill.dragon')
killDragon(@Payload() message: KillDragonMessage, @Ctx() context:
KafkaContext) {
  console.log(`Topic: ${context.getTopic()}`);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('hero.kill.dragon')
killDragon(message, context) {
  console.log(`Topic: ${context.getTopic()}`);
}
```

info Hint [@Payload\(\)](#), [@Ctx\(\)](#) and [KafkaContext](#) are imported from the [@nestjs/microservices](#) package.

To access the original Kafka [IncomingMessage](#) object, use the [getMessage\(\)](#) method of the [KafkaContext](#) object, as follows:

```
@@filename()
@MessagePattern('hero.kill.dragon')
killDragon(@Payload() message: KillDragonMessage, @Ctx() context:
KafkaContext) {
  const incomingMessage = context.getMessage();
  // ...
}
```

```

KafkaContext) {
  const originalMessage = context.getMessage();
  const partition = context.getPartition();
  const { headers, timestamp } = originalMessage;
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('hero.kill.dragon')
killDragon(message, context) {
  const originalMessage = context.getMessage();
  const partition = context.getPartition();
  const { headers, timestamp } = originalMessage;
}

```

Where the `IncomingMessage` fulfills the following interface:

```

interface IncomingMessage {
  topic: string;
  partition: number;
  timestamp: string;
  size: number;
  attributes: number;
  offset: string;
  key: any;
  value: any;
  headers: Record<string, any>;
}

```

If your handler involves a slow processing time for each received message you should consider using the `heartbeat` callback. To retrieve the `heartbeat` function, use the `getHeartbeat()` method of the `KafkaContext`, as follows:

```

@@filename()
@MessagePattern('hero.kill.dragon')
async killDragon(@Payload() message: KillDragonMessage, @Ctx() context: KafkaContext) {
  const heartbeat = context.getHeartbeat();

  // Do some slow processing
  await doWorkPart1();

  // Send heartbeat to not exceed the sessionTimeout
  await heartbeat();

  // Do some slow processing again
  await doWorkPart2();
}

```

## Naming conventions

The Kafka microservice components append a description of their respective role onto the `client.clientId` and `consumer.groupId` options to prevent collisions between Nest microservice client and server components. By default the `ClientKafka` components append `-client` and the `ServerKafka` components append `-server` to both of these options. Note how the provided values below are transformed in that way (as shown in the comments).

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.KAFKA,
  options: {
    client: {
      clientId: 'hero', // hero-server
      brokers: ['localhost:9092'],
    },
    consumer: {
      groupId: 'hero-consumer' // hero-consumer-server
    },
  }
});
```

And for the client:

```
@@filename(heroes.controller)
@Client({
  transport: Transport.KAFKA,
  options: {
    client: {
      clientId: 'hero', // hero-client
      brokers: ['localhost:9092'],
    },
    consumer: {
      groupId: 'hero-consumer' // hero-consumer-client
    }
  }
})
client: ClientKafka;
```

info **Hint** Kafka client and consumer naming conventions can be customized by extending `ClientKafka` and `KafkaServer` in your own custom provider and overriding the constructor.

Since the Kafka microservice message pattern utilizes two topics for the request and reply channels, a reply pattern should be derived from the request topic. By default, the name of the reply topic is the composite of the request topic name with `.reply` appended.

```
@@filename(heroes.controller)
onModuleInit() {
  this.client.subscribeToResponseOf('hero.get'); // hero.get.reply
}
```

**info Hint** Kafka reply topic naming conventions can be customized by extending [ClientKafka](#) in your own custom provider and overriding the [getResponsePatternName](#) method.

## Retriable exceptions

Similar to other transporters, all unhandled exceptions are automatically wrapped into an [RpcException](#) and converted to a "user-friendly" format. However, there are edge-cases when you might want to bypass this mechanism and let exceptions be consumed by the [kafkajs](#) driver instead. Throwing an exception when processing a message instructs [kafkajs](#) to [retry](#) it (redeliver it) which means that even though the message (or event) handler was triggered, the offset won't be committed to Kafka.

**warning Warning** For event handlers (event-based communication), all unhandled exceptions are considered **retriable exceptions** by default.

For this, you can use a dedicated class called [KafkaRetriableException](#), as follows:

```
throw new KafkaRetriableException('...');
```

**info Hint** [KafkaRetriableException](#) class is exported from the [@nestjs/microservices](#) package.

## Commit offsets

Committing offsets is essential when working with Kafka. Per default, messages will be automatically committed after a specific time. For more information visit [KafkaJS docs](#). [ClientKafka](#) offers a way to manually commit offsets that work like the [native KafkaJS implementation](#).

```
@@filename()
@EventPattern('user.created')
async handleUserCreated(@Payload() data: IncomingMessage, @Ctx() context: KafkaContext) {
  // business logic

  const { offset } = context.getMessage();
  const partition = context.getPartition();
  const topic = context.getTopic();
  await this.client.commitOffsets([{ topic, partition, offset }])
}

@switch
@Bind(Payload(), Ctx())
@EventPattern('user.created')
async handleUserCreated(data, context) {
```

```
// business logic

const { offset } = context.getMessage();
const partition = context.getPartition();
const topic = context.getTopic();
await this.client.commitOffsets([{ topic, partition, offset }])
}
```

To disable auto-committing of messages set `autoCommit: false` in the `run` configuration, as follows:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.KAFKA,
  options: {
    client: {
      brokers: ['localhost:9092'],
    },
    run: {
      autoCommit: false
    }
  }
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.KAFKA,
  options: {
    client: {
      brokers: ['localhost:9092'],
    },
    run: {
      autoCommit: false
    }
  }
});
```

## gRPC

gRPC is a modern, open source, high performance RPC framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication.

Like many RPC systems, gRPC is based on the concept of defining a service in terms of functions (methods) that can be called remotely. For each method, you define the parameters and return types. Services, parameters, and return types are defined in `.proto` files using Google's open source language-neutral [protocol buffers](#) mechanism.

With the gRPC transporter, Nest uses `.proto` files to dynamically bind clients and servers to make it easy to implement remote procedure calls, automatically serializing and deserializing structured data.

### Installation

To start building gRPC-based microservices, first install the required packages:

```
$ npm i --save @grpc/grpc-js @grpc/proto-loader
```

### Overview

Like other Nest microservices transport layer implementations, you select the gRPC transporter mechanism using the `transport` property of the options object passed to the `createMicroservice()` method. In the following example, we'll set up a hero service. The `options` property provides metadata about that service; its properties are described [below](#).

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.GRPC,
  options: {
    package: 'hero',
    protoPath: join(__dirname, 'hero/hero.proto'),
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.GRPC,
  options: {
    package: 'hero',
    protoPath: join(__dirname, 'hero/hero.proto'),
  },
});
```

**info Hint** The `join()` function is imported from the `path` package; the `Transport` enum is imported from the `@nestjs/microservices` package.

In the `nest-cli.json` file, we add the `assets` property that allows us to distribute non-TypeScript files, and `watchAssets` - to turn on watching all non-TypeScript assets. In our case, we want `.proto` files to be automatically copied to the `dist` folder.

```
{
  "compilerOptions": {
    "assets": ["**/*.proto"],
    "watchAssets": true
  }
}
```

## Options

The **gRPC** transporter options object exposes the properties described below.

<code>package</code>	Protobuf package name (matches <code>package</code> setting from <code>.proto</code> file). Required
<code>protoPath</code>	Absolute (or relative to the root dir) path to the <code>.proto</code> file. Required
<code>url</code>	Connection url. String in the format <code>ip address/dns name:port</code> (for example, <code>'localhost:50051'</code> ) defining the address/port on which the transporter establishes a connection. Optional. Defaults to <code>'localhost:5000'</code>
<code>protoLoader</code>	NPM package name for the utility to load <code>.proto</code> files. Optional. Defaults to <code>'@grpc/proto-loader'</code>
<code>loader</code>	<code>@grpc/proto-loader</code> options. These provide detailed control over the behavior of <code>.proto</code> files. Optional. See <a href="#">here</a> for more details
<code>credentials</code>	Server credentials. Optional. <a href="#">Read more here</a>

## Sample gRPC service

Let's define our sample gRPC service called `HeroesService`. In the above `options` object, the `protoPath` property sets a path to the `.proto` definitions file `hero.proto`. The `hero.proto` file is structured using [protocol buffers](#). Here's what it looks like:

```
// hero/hero.proto
syntax = "proto3";

package hero;

service HeroesService {
  rpc FindOne (HeroById) returns (Hero) {}
}
```

```
message HeroById {
    int32 id = 1;
}

message Hero {
    int32 id = 1;
    string name = 2;
}
```

Our `HeroesService` exposes a `FindOne()` method. This method expects an input argument of type `HeroById` and returns a `Hero` message (protocol buffers use `message` elements to define both parameter types and return types).

Next, we need to implement the service. To define a handler that fulfills this definition, we use the `@GrpcMethod()` decorator in a controller, as shown below. This decorator provides the metadata needed to declare a method as a gRPC service method.

**info Hint** The `@MessagePattern()` decorator ([read more](#)) introduced in previous microservices chapters is not used with gRPC-based microservices. The `@GrpcMethod()` decorator effectively takes its place for gRPC-based microservices.

```
@@filename(heroes.controller)
@Controller()
export class HeroesController {
    @GrpcMethod('HeroesService', 'FindOne')
    findOne(data: HeroById, metadata: Metadata, call: ServerUnaryCall<any, any>): Hero {
        const items = [
            { id: 1, name: 'John' },
            { id: 2, name: 'Doe' },
        ];
        return items.find(({ id }) => id === data.id);
    }
}

@@switch
@Controller()
export class HeroesController {
    @GrpcMethod('HeroesService', 'FindOne')
    findOne(data, metadata, call) {
        const items = [
            { id: 1, name: 'John' },
            { id: 2, name: 'Doe' },
        ];
        return items.find(({ id }) => id === data.id);
    }
}
```

**info Hint** The `@GrpcMethod()` decorator is imported from the `@nestjs/microservices` package, while `Metadata` and `ServerUnaryCall` from the `grpc` package.

The decorator shown above takes two arguments. The first is the service name (e.g., '`'HeroesService'`), corresponding to the `HeroesService` service definition in `hero.proto`. The second (the string '`'FindOne'`') corresponds to the `FindOne()` rpc method defined within `HeroesService` in the `hero.proto` file.

The `findOne()` handler method takes three arguments, the `data` passed from the caller, `metadata` that stores gRPC request metadata and `call` to obtain the `GrpcCall` object properties such as `sendMetadata` for send metadata to client.

Both `@GrpcMethod()` decorator arguments are optional. If called without the second argument (e.g., '`'FindOne'`'), Nest will automatically associate the `.proto` file rpc method with the handler based on converting the handler name to upper camel case (e.g., the `findOne` handler is associated with the `FindOne` rpc call definition). This is shown below.

```
@@filename(heroes.controller)
@Controller()
export class HeroesController {
  @GrpcMethod('HeroesService')
  findOne(data: HeroById, metadata: Metadata, call: ServerUnaryCall<any,
any>): Hero {
  const items = [
    { id: 1, name: 'John' },
    { id: 2, name: 'Doe' },
  ];
  return items.find(({ id }) => id === data.id);
}
}

@switch
@Controller()
export class HeroesController {
  @GrpcMethod('HeroesService')
  findOne(data, metadata, call) {
  const items = [
    { id: 1, name: 'John' },
    { id: 2, name: 'Doe' },
  ];
  return items.find(({ id }) => id === data.id);
}
}
```

You can also omit the first `@GrpcMethod()` argument. In this case, Nest automatically associates the handler with the service definition from the proto definitions file based on the `class` name where the handler is defined. For example, in the following code, class `HeroesService` associates its handler methods with the `HeroesService` service definition in the `hero.proto` file based on the matching of the name '`'HeroesService'`'.

```
@@filename(heroes.controller)
@Controller()
export class HeroesService {
```

```

@GrpcMethod()
findOne(data: HeroById, metadata: Metadata, call: ServerUnaryCall<any,
any>): Hero {
  const items = [
    { id: 1, name: 'John' },
    { id: 2, name: 'Doe' },
  ];
  return items.find(({ id }) => id === data.id);
}
}

@@switch
@Controller()
export class HeroesService {
  @GrpcMethod()
  findOne(data, metadata, call) {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    return items.find(({ id }) => id === data.id);
  }
}

```

## Client

Nest applications can act as gRPC clients, consuming services defined in `.proto` files. You access remote services through a `ClientGrpc` object. You can obtain a `ClientGrpc` object in several ways.

The preferred technique is to import the `ClientsModule`. Use the `register()` method to bind a package of services defined in a `.proto` file to an injection token, and to configure the service. The `name` property is the injection token. For gRPC services, use `transport: Transport.GRPC`. The `options` property is an object with the same properties described [above](#).

```

imports: [
  ClientsModule.register([
    {
      name: 'HERO_PACKAGE',
      transport: Transport.GRPC,
      options: {
        package: 'hero',
        protoPath: join(__dirname, 'hero/hero.proto'),
      },
    },
  ]),
];

```

**info Hint** The `register()` method takes an array of objects. Register multiple packages by providing a comma separated list of registration objects.

Once registered, we can inject the configured `ClientGrpc` object with `@Inject()`. Then we use the `ClientGrpc` object's `getService()` method to retrieve the service instance, as shown below.

```
@Injectable()
export class AppService implements OnModuleInit {
    private heroesService: HeroesService;

    constructor(@Inject('HERO_PACKAGE') private client: ClientGrpc) {}

    onModuleInit() {
        this.heroesService = this.client.getService<HeroesService>(
            'HeroesService');
    }

    getHero(): Observable<string> {
        return this.heroesService.findOne({ id: 1 });
    }
}
```

error **Warning** gRPC Client will not send fields that contain underscore `_` in their names unless the `keepCase` options is set to `true` in the proto loader configuration (`options.loader.keepcase` in the microservice transporter configuration).

Notice that there is a small difference compared to the technique used in other microservice transport methods. Instead of the `ClientProxy` class, we use the `ClientGrpc` class, which provides the `getService()` method. The `getService()` generic method takes a service name as an argument and returns its instance (if available).

Alternatively, you can use the `@Client()` decorator to instantiate a `ClientGrpc` object, as follows:

```
@Injectable()
export class AppService implements OnModuleInit {
    @Client({
        transport: Transport.GRPC,
        options: {
            package: 'hero',
            protoPath: join(__dirname, 'hero/hero.proto'),
        },
    })
    client: ClientGrpc;

    private heroesService: HeroesService;

    onModuleInit() {
        this.heroesService = this.client.getService<HeroesService>(
            'HeroesService');
    }

    getHero(): Observable<string> {
        return this.heroesService.findOne({ id: 1 });
    }
}
```

```

    }
}
```

Finally, for more complex scenarios, we can inject a dynamically configured client using the [ClientProxyFactory](#) class as described [here](#).

In either case, we end up with a reference to our `HeroesService` proxy object, which exposes the same set of methods that are defined inside the `.proto` file. Now, when we access this proxy object (i.e., `heroesService`), the gRPC system automatically serializes requests, forwards them to the remote system, returns a response, and deserializes the response. Because gRPC shields us from these network communication details, `heroesService` looks and acts like a local provider.

Note, all service methods are **lower camel cased** (in order to follow the natural convention of the language). So, for example, while our `.proto` file `HeroesService` definition contains the `FindOne()` function, the `heroesService` instance will provide the `findOne()` method.

```

interface HeroesService {
  findOne(data: { id: number }): Observable<any>;
}
```

A message handler is also able to return an `Observable`, in which case the result values will be emitted until the stream is completed.

```

@@filename(heroes.controller)
@Get()
call(): Observable<any> {
  return this.heroesService.findOne({ id: 1 });
}
@@switch
@Get()
call() {
  return this.heroesService.findOne({ id: 1 });
}
```

To send gRPC metadata (along with the request), you can pass a second argument, as follows:

```

call(): Observable<any> {
  const metadata = new Metadata();
  metadata.add('Set-Cookie', 'yummy_cookie=choco');

  return this.heroesService.findOne({ id: 1 }, metadata);
}
```

**info Hint** The `Metadata` class is imported from the `grpc` package.

Please note that this would require updating the `HeroesService` interface that we've defined a few steps earlier.

## Example

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

## gRPC Streaming

gRPC on its own supports long-term live connections, conventionally known as `streams`. Streams are useful for cases such as Chatting, Observations or Chunk-data transfers. Find more details in the official documentation [here](#).

Nest supports GRPC stream handlers in two possible ways:

- RxJS `Subject` + `Observable` handler: can be useful to write responses right inside of a Controller method or to be passed down to `Subject/Observable` consumer
- Pure GRPC call stream handler: can be useful to be passed to some executor which will handle the rest of dispatch for the Node standard `Duplex` stream handler.

## Streaming sample

Let's define a new sample gRPC service called `HelloService`. The `hello.proto` file is structured using [protocol buffers](#). Here's what it looks like:

```
// hello/hello.proto
syntax = "proto3";

package hello;

service HelloService {
    rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);
    rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);
}

message HelloRequest {
    string greeting = 1;
}

message HelloResponse {
    string reply = 1;
}
```

**info Hint** The `LotsOfGreetings` method can be simply implemented with the `@GrpcMethod` decorator (as in the examples above) since the returned stream can emit multiple values.

Based on this `.proto` file, let's define the `HelloService` interface:

```
interface HelloService {
  bidiHello(upstream: Observable<HelloRequest>):
  Observable<HelloResponse>;
  lotsOfGreetings(
    upstream: Observable<HelloRequest>,
  ): Observable<HelloResponse>;
}

interface HelloRequest {
  greeting: string;
}

interface HelloResponse {
  reply: string;
}
```

info **Hint** The proto interface can be automatically generated by the [ts-proto package](#), learn more [here](#).

## Subject strategy

The `@GrpcStreamMethod()` decorator provides the function parameter as an RxJS `Observable`. Thus, we can receive and process multiple messages.

```
@GrpcStreamMethod()
bidiHello(messages: Observable<any>, metadata: Metadata, call:
ServerDuplexStream<any, any>): Observable<any> {
  const subject = new Subject();

  const onNext = message => {
    console.log(message);
    subject.next({
      reply: 'Hello, world!'
    });
  };
  const onComplete = () => subject.complete();
  messages.subscribe({
    next: onNext,
    complete: onComplete,
  });

  return subject.asObservable();
}
```

warning **Warning** For supporting full-duplex interaction with the `@GrpcStreamMethod()` decorator, the controller method must return an RxJS `Observable`.

**info Hint** The `Metadata` and `ServerUnaryCall` classes/interfaces are imported from the `grpc` package.

According to the service definition (in the `.proto` file), the `BidiHello` method should stream requests to the service. To send multiple asynchronous messages to the stream from a client, we leverage an RxJS `ReplaySubject` class.

```
const helloService = this.client.getService<HelloService>('HelloService');
const helloRequest$ = new ReplaySubject<HelloRequest>();

helloRequest$.next({ greeting: 'Hello (1)!' });
helloRequest$.next({ greeting: 'Hello (2)!' });
helloRequest$.complete();

return helloService.bidiHello(helloRequest$);
```

In the example above, we wrote two messages to the stream (`next()` calls) and notified the service that we've completed sending the data (`complete()` call).

## Call stream handler

When the method return value is defined as `stream`, the `@GrpcStreamCall()` decorator provides the function parameter as `grpc.ServerDuplexStream`, which supports standard methods like `.on('data', callback)`, `.write(message)` or `.cancel()`. Full documentation on available methods can be found [here](#).

Alternatively, when the method return value is not a `stream`, the `@GrpcStreamCall()` decorator provides two function parameters, respectively `grpc.ServerReadableStream` (read more [here](#)) and `callback`.

Let's start with implementing the `BidiHello` which should support a full-duplex interaction.

```
@GrpcStreamCall()
bidiHello(requestStream: any) {
  requestStream.on('data', message => {
    console.log(message);
    requestStream.write({
      reply: 'Hello, world!'
    });
  });
}
```

**info Hint** This decorator does not require any specific return parameter to be provided. It is expected that the stream will be handled similar to any other standard stream type.

In the example above, we used the `write()` method to write objects to the response stream. The callback passed into the `.on()` method as a second parameter will be called every time our service receives a new chunk of data.

Let's implement the `LotsOfGreetings` method.

```
@GrpcStreamCall()
lotsOfGreetings(requestStream: any, callback: (err: unknown, value: HelloResponse) => void) {
  requestStream.on('data', message => {
    console.log(message);
  });
  requestStream.on('end', () => callback(null, { reply: 'Hello, world!' }));
}
```

Here we used the `callback` function to send the response once processing of the `requestStream` has been completed.

## gRPC Metadata

Metadata is information about a particular RPC call in the form of a list of key-value pairs, where the keys are strings and the values are typically strings but can be binary data. Metadata is opaque to gRPC itself - it lets the client provide information associated with the call to the server and vice versa. Metadata may include authentication tokens, request identifiers and tags for monitoring purposes, and data information such as the number of records in a data set.

To read the metadata in `@GrpcMethod()` handler, use the second argument (`metadata`), which is of type `Metadata` (imported from the `grpc` package).

To send back metadata from the handler, use the `ServerUnaryCall#sendMetadata()` method (third handler argument).

```
@@filename(heroes.controller)
@Controller()
export class HeroesService {
  @GrpcMethod()
  findOne(data: HeroById, metadata: Metadata, call: ServerUnaryCall<any, any>): Hero {
    const serverMetadata = new Metadata();
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    serverMetadata.add('Set-Cookie', 'yummy_cookie=choco');
    call.sendMetadata(serverMetadata);

    return items.find(({ id }) => id === data.id);
  }
}
@@switch
@Controller()
```

```
export class HeroesService {
  @GrpcMethod()
  findOne(data, metadata, call) {
    const serverMetadata = new Metadata();
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];

    serverMetadata.add('Set-Cookie', 'yummy_cookie=choco');
    call.sendMetadata(serverMetadata);

    return items.find(({ id }) => id === data.id);
  }
}
```

Likewise, to read the metadata in handlers annotated with the `@GrpcStreamMethod()` handler (subject strategy), use the second argument (`metadata`), which is of type `Metadata` (imported from the `grpc` package).

To send back metadata from the handler, use the `ServerDuplexStream#sendMetadata()` method (third handler argument).

To read metadata from within the `call stream handlers` (handlers annotated with `@GrpcStreamCall()` decorator), listen to the `metadata` event on the `requestStream` reference, as follows:

```
requestStream.on('metadata', (metadata: Metadata) => {
  const meta = metadata.get('X-Meta');
});
```

## Custom transporters

Nest provides a variety of **transporters** out-of-the-box, as well as an API allowing developers to build new custom transport strategies. Transporters enable you to connect components over a network using a pluggable communications layer and a very simple application-level message protocol (read full [article](#)).

**info Hint** Building a microservice with Nest does not necessarily mean you must use the `@nestjs/microservices` package. For example, if you want to communicate with external services (let's say other microservices written in different languages), you may not need all the features provided by `@nestjs/microservice` library. In fact, if you don't need decorators (`@EventPattern` or `@MessagePattern`) that let you declaratively define subscribers, running a [Standalone Application](#) and manually maintaining connection/subscribing to channels should be enough for most use-cases and will provide you with more flexibility.

With a custom transporter, you can integrate any messaging system/protocol (including Google Cloud Pub/Sub, Amazon Kinesis, and others) or extend the existing one, adding extra features on top (for example, [QoS](#) for MQTT).

**info Hint** To better understand how Nest microservices work and how you can extend the capabilities of existing transporters, we recommend reading the [NestJS Microservices in Action](#) and [Advanced NestJS Microservices](#) article series.

### Creating a strategy

First, let's define a class representing our custom transporter.

```
import { CustomTransportStrategy, Server } from '@nestjs/microservices';

class GoogleCloudPubSubServer
  extends Server
  implements CustomTransportStrategy {
  /**
   * This method is triggered when you run "app.listen()".
   */
  listen(callback: () => void) {
    callback();
  }

  /**
   * This method is triggered on application shutdown.
   */
  close() {}
}
```

**warning Warning** Please, note we won't be implementing a fully-featured Google Cloud Pub/Sub server in this chapter as this would require diving into transporter specific technical details.

In our example above, we declared the `GoogleCloudPubSubServer` class and provided `listen()` and `close()` methods enforced by the `CustomTransportStrategy` interface. Also, our class extends the `Server` class imported from the `@nestjs/microservices` package that provides a few useful methods, for example, methods used by Nest runtime to register message handlers. Alternatively, in case you want to extend the capabilities of an existing transport strategy, you could extend the corresponding server class, for example, `ServerRedis`. Conventionally, we added the "`Server`" suffix to our class as it will be responsible for subscribing to messages/events (and responding to them, if necessary).

With this in place, we can now use our custom strategy instead of using a built-in transporter, as follows:

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>(
  AppModule,
  {
    strategy: new GoogleCloudPubSubServer(),
  },
);
```

Basically, instead of passing the normal transporter options object with `transport` and `options` properties, we pass a single property, `strategy`, whose value is an instance of our custom transporter class.

Back to our `GoogleCloudPubSubServer` class, in a real-world application, we would be establishing a connection to our message broker/external service and registering subscribers/listening to specific channels in `listen()` method (and then removing subscriptions & closing the connection in the `close()` teardown method), but since this requires a good understanding of how Nest microservices communicate with each other, we recommend reading this [article series](#). In this chapter instead, we'll focus on the capabilities the `Server` class provides and how you can leverage them to build custom strategies.

For example, let's say that somewhere in our application, the following message handler is defined:

```
@MessagePattern('echo')
echo(@Payload() data: object) {
  return data;
}
```

This message handler will be automatically registered by Nest runtime. With `Server` class, you can see what message patterns have been registered and also, access and execute the actual methods that were assigned to them. To test this out, let's add a simple `console.log` inside `listen()` method before `callback` function is called:

```
listen(callback: () => void) {
  console.log(this.messageHandlers);
  callback();
}
```

After your application restarts, you'll see the following log in your terminal:

```
Map { 'echo' => [AsyncFunction] { isEventHandler: false } }
```

**info Hint** If we used the `@EventPattern` decorator, you would see the same output, but with the `isEventHandler` property set to `true`.

As you can see, the `messageHandlers` property is a `Map` collection of all message (and event) handlers, in which patterns are being used as keys. Now, you can use a key (for example, `"echo"`) to receive a reference to the message handler:

```
async listen(callback: () => void) {
  const echoHandler = this.messageHandlers.get('echo');
  console.log(await echoHandler('Hello world!'));
  callback();
}
```

Once we execute the `echoHandler` passing an arbitrary string as an argument (`"Hello world!"` here), we should see it in the console:

```
Hello world!
```

Which means that our method handler was properly executed.

When using a `CustomTransportStrategy` with `Interceptors` the handlers are wrapped into RxJS streams. This means that you need to subscribe to them in order to execute the streams underlying logic (e.g. continue into the controller logic after an interceptor has been executed).

An example of this can be seen below:

```
async listen(callback: () => void) {
  const echoHandler = this.messageHandlers.get('echo');
  const streamOrResult = await echoHandler('Hello World');
  if (isObservable(streamOrResult)) {
    streamOrResult.subscribe();
  }
  callback();
}
```

## Client proxy

As we mentioned in the first section, you don't necessarily need to use the `@nestjs/microservices` package to create microservices, but if you decide to do so and you need to integrate a custom strategy, you will need to provide a "client" class too.

**info Hint** Again, implementing a fully-featured client class compatible with all `@nestjs/microservices` features (e.g., streaming) requires a good understanding of communication techniques used by the framework. To learn more, check out this [article](#).

To communicate with an external service/emit & publish messages (or events) you can either use a library-specific SDK package, or implement a custom client class that extends the `ClientProxy`, as follows:

```
import { ClientProxy, ReadPacket, WritePacket } from
'@nestjs/microservices';

class GoogleCloudPubSubClient extends ClientProxy {
  async connect(): Promise<any> {}
  async close() {}
  async dispatchEvent(packet: ReadPacket<any>): Promise<any> {}
  publish(
    packet: ReadPacket<any>,
    callback: (packet: WritePacket<any>) => void,
  ): Function {}
}
```

**warning Warning** Please, note we won't be implementing a fully-featured Google Cloud Pub/Sub client in this chapter as this would require diving into transporter specific technical details.

As you can see, `ClientProxy` class requires us to provide several methods for establishing & closing the connection and publishing messages (`publish`) and events (`dispatchEvent`). Note, if you don't need a request-response communication style support, you can leave the `publish()` method empty. Likewise, if you don't need to support event-based communication, skip the `dispatchEvent()` method.

To observe what and when those methods are executed, let's add multiple `console.log` calls, as follows:

```
class GoogleCloudPubSubClient extends ClientProxy {
  async connect(): Promise<any> {
    console.log('connect');
  }

  async close() {
    console.log('close');
  }

  async dispatchEvent(packet: ReadPacket<any>): Promise<any> {
    return console.log('event to dispatch: ', packet);
  }

  publish(
    packet: ReadPacket<any>,
    callback: (packet: WritePacket<any>) => void,
  ): Function {
    console.log('message:', packet);

    // In a real-world application, the "callback" function should be
  }
}
```

```
executed
  // with payload sent back from the responder. Here, we'll simply
  // simulate (5 seconds delay)
  // that response came through by passing the same "data" as we've
  // originally passed in.
  setTimeout(() => callback({ response: packet.data }), 5000);

  return () => console.log('teardown');
}
}
```

With this in place, let's create an instance of `GoogleCloudPubSubClient` class and run the `send()` method (which you might have seen in earlier chapters), subscribing to the returned observable stream.

```
const googlePubSubClient = new GoogleCloudPubSubClient();
googlePubSubClient
  .send('pattern', 'Hello world!')
  .subscribe((response) => console.log(response));
```

Now, you should see the following output in your terminal:

```
connect
message: { pattern: 'pattern', data: 'Hello world!' }
Hello world! // <-- after 5 seconds
```

To test if our "teardown" method (which our `publish()` method returns) is properly executed, let's apply a timeout operator to our stream, setting it to 2 seconds to make sure it throws earlier than our `setTimeout` calls the `callback` function.

```
const googlePubSubClient = new GoogleCloudPubSubClient();
googlePubSubClient
  .send('pattern', 'Hello world!')
  .pipe(timeout(2000))
  .subscribe(
    (response) => console.log(response),
    (error) => console.error(error.message),
  );
```

**info Hint** The `timeout` operator is imported from the `rxjs/operators` package.

With `timeout` operator applied, your terminal output should look as follows:

```
connect
message: { pattern: 'pattern', data: 'Hello world!' }
```

```
teardown // <-- teardown  
Timeout has occurred
```

To dispatch an event (instead of sending a message), use the `emit()` method:

```
googlePubSubClient.emit('event', 'Hello world!');
```

And that's what you should see in the console:

```
connect  
event to dispatch: { pattern: 'event', data: 'Hello world!' }
```

## Message serialization

If you need to add some custom logic around the serialization of responses on the client side, you can use a custom class that extends the `ClientProxy` class or one of its child classes. For modifying successful requests you can override the `serializeResponse` method, and for modifying any errors that go through this client you can override the `serializeError` method. To make use of this custom class, you can pass the class itself to the `ClientsModule.register()` method using the `customClass` property. Below is an example of a custom `ClientProxy` that serializes each error into an `RpcException`.

```
@@filename(error-handling.proxy)  
import { ClientTcp, RpcException } from '@nestjs/microservices';  
  
class ErrorHandlingProxy extends ClientTCP {  
  serializeError(err: Error) {  
    return new RpcException(err);  
  }  
}
```

and then use it in the `ClientsModule` like so:

```
@@filename(app.module)  
@Module({  
  imports: [  
    ClientsModule.register({  
      name: 'CustomProxy',  
      customClass: ErrorHandlingProxy,  
    }),  
  ]  
})  
export class AppModule
```

**info hint** This is the class itself being passed to `customClass`, not an instance of the class. Nest will create the instance under the hood for you, and will pass any options given to the `options` property to the new `ClientProxy`.

## Exception filters

The only difference between the HTTP [exception filter](#) layer and the corresponding microservices layer is that instead of throwing [HttpException](#), you should use [RpcException](#).

```
throw new RpcException('Invalid credentials.');
```

**info Hint** The [RpcException](#) class is imported from the [@nestjs/microservices](#) package.

With the sample above, Nest will handle the thrown exception and return the [error](#) object with the following structure:

```
{
  "status": "error",
  "message": "Invalid credentials."
}
```

## Filters

Microservice exception filters behave similarly to HTTP exception filters, with one small difference. The [catch\(\)](#) method must return an [Observable](#).

```
@@filename(rpc-exception.filter)
import { Catch, RpcExceptionFilter, ArgumentsHost } from '@nestjs/common';
import { Observable, throwError } from 'rxjs';
import { RpcException } from '@nestjs/microservices';

@Catch(RpcException)
export class ExceptionFilter implements RpcExceptionFilter<RpcException> {
  catch(exception: RpcException, host: ArgumentsHost): Observable<any> {
    return throwError(() => exception.getError());
  }
}
@@switch
import { Catch } from '@nestjs/common';
import { throwError } from 'rxjs';

@Catch(RpcException)
export class ExceptionFilter {
  catch(exception, host) {
    return throwError(() => exception.getError());
  }
}
```

warning **Warning** Global microservice exception filters aren't enabled by default when using a [hybrid application](#).

The following example uses a manually instantiated method-scoped filter. Just as with HTTP based applications, you can also use controller-scoped filters (i.e., prefix the controller class with a `@UseFilters()` decorator).

```
@@filename()
@UseFilters(new ExceptionFilter())
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}

@@switch
@UseFilters(new ExceptionFilter())
@MessagePattern({ cmd: 'sum' })
accumulate(data) {
  return (data || []).reduce((a, b) => a + b);
}
```

## Inheritance

Typically, you'll create fully customized exception filters crafted to fulfill your application requirements. However, there might be use-cases when you would like to simply extend the **core exception filter**, and override the behavior based on certain factors.

In order to delegate exception processing to the base filter, you need to extend `BaseExceptionFilter` and call the inherited `catch()` method.

```
@@filename()
import { Catch, ArgumentsHost } from '@nestjs/common';
import { BaseRpcExceptionFilter } from '@nestjs/microservices';

@Catch()
export class AllExceptionsFilter extends BaseRpcExceptionFilter {
  catch(exception: any, host: ArgumentsHost) {
    return super.catch(exception, host);
  }
}

@@switch
import { Catch } from '@nestjs/common';
import { BaseRpcExceptionFilter } from '@nestjs/microservices';

@Catch()
export class AllExceptionsFilter extends BaseRpcExceptionFilter {
  catch(exception, host) {
    return super.catch(exception, host);
  }
}
```

The above implementation is just a shell demonstrating the approach. Your implementation of the extended exception filter would include your tailored **business logic** (e.g., handling various conditions).

## Pipes

There is no fundamental difference between [regular pipes](#) and microservices pipes. The only difference is that instead of throwing [HttpException](#), you should use [RpcException](#).

**info Hint** The [RpcException](#) class is exposed from [@nestjs/microservices](#) package.

### Binding pipes

The following example uses a manually instantiated method-scoped pipe. Just as with HTTP based applications, you can also use controller-scoped pipes (i.e., prefix the controller class with a [@UsePipes\(\)](#) decorator).

```
@@filename()
@UsePipes(new ValidationPipe())
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
@@switch
@UsePipes(new ValidationPipe())
@MessagePattern({ cmd: 'sum' })
accumulate(data) {
  return (data || []).reduce((a, b) => a + b);
}
```

## Guards

There is no fundamental difference between microservices guards and regular HTTP application guards. The only difference is that instead of throwing `HttpException`, you should use `RpcException`.

**info Hint** The `RpcException` class is exposed from `@nestjs/microservices` package.

### Binding guards

The following example uses a method-scoped guard. Just as with HTTP based applications, you can also use controller-scoped guards (i.e., prefix the controller class with a `@UseGuards()` decorator).

```
@@filename()
@UseGuards(AuthGuard)
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
@@switch
@UseGuards(AuthGuard)
@MessagePattern({ cmd: 'sum' })
accumulate(data) {
  return (data || []).reduce((a, b) => a + b);
}
```

## Interceptors

There is no difference between [regular interceptors](#) and microservices interceptors. The following example uses a manually instantiated method-scoped interceptor. Just as with HTTP based applications, you can also use controller-scoped interceptors (i.e., prefix the controller class with a `@UseInterceptors()` decorator).

```
@@filename()
@UseInterceptors(new TransformInterceptor())
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
@@switch
@UseInterceptors(new TransformInterceptor())
@MessagePattern({ cmd: 'sum' })
accumulate(data) {
  return (data || []).reduce((a, b) => a + b);
}
```

## Overview

The [Nest CLI](#) is a command-line interface tool that helps you to initialize, develop, and maintain your Nest applications. It assists in multiple ways, including scaffolding the project, serving it in development mode, and building and bundling the application for production distribution. It embodies best-practice architectural patterns to encourage well-structured apps.

## Installation

**Note:** In this guide we describe using [npm](#) to install packages, including the Nest CLI. Other package managers may be used at your discretion. With npm, you have several options available for managing how your OS command line resolves the location of the [nest](#) CLI binary file. Here, we describe installing the [nest](#) binary globally using the [-g](#) option. This provides a measure of convenience, and is the approach we assume throughout the documentation. Note that installing **any** [npm](#) package globally leaves the responsibility of ensuring they're running the correct version up to the user. It also means that if you have different projects, each will run the **same** version of the CLI. A reasonable alternative is to use the [npx](#) program, built into the [npm](#) cli (or similar features with other package managers) to ensure that you run a **managed version** of the Nest CLI. We recommend you consult the [npx documentation](#) and/or your DevOps support staff for more information.

Install the CLI globally using the [npm install -g](#) command (see the **Note** above for details about global installs).

```
$ npm install -g @nestjs/cli
```

**info Hint** Alternatively, you can use this command [npx @nestjs/cli@latest](#) without installing the cli globally.

## Basic workflow

Once installed, you can invoke CLI commands directly from your OS command line through the [nest](#) executable. See the available [nest](#) commands by entering the following:

```
$ nest --help
```

Get help on an individual command using the following construct. Substitute any command, like [new](#), [add](#), etc., where you see [generate](#) in the example below to get detailed help on that command:

```
$ nest generate --help
```

To create, build and run a new basic Nest project in development mode, go to the folder that should be the parent of your new project, and run the following commands:

```
$ nest new my-nest-project
$ cd my-nest-project
$ npm run start:dev
```

In your browser, open <http://localhost:3000> to see the new application running. The app will automatically recompile and reload when you change any of the source files.

**info Hint** We recommend using the [SWC builder](#) for faster builds (10x more performant than the default TypeScript compiler).

## Project structure

When you run `nest new`, Nest generates a boilerplate application structure by creating a new folder and populating an initial set of files. You can continue working in this default structure, adding new components, as described throughout this documentation. We refer to the project structure generated by `nest new` as **standard mode**. Nest also supports an alternate structure for managing multiple projects and libraries called **monorepo mode**.

Aside from a few specific considerations around how the **build** process works (essentially, monorepo mode simplifies build complexities that can sometimes arise from monorepo-style project structures), and built-in [library](#) support, the rest of the Nest features, and this documentation, apply equally to both standard and monorepo mode project structures. In fact, you can easily switch from standard mode to monorepo mode at any time in the future, so you can safely defer this decision while you're still learning about Nest.

You can use either mode to manage multiple projects. Here's a quick summary of the differences:

Feature	Standard Mode	Monorepo Mode
Multiple projects	Separate file system structure	Single file system structure
<code>node_modules</code> & <code>package.json</code>	Separate instances	Shared across monorepo
Default compiler	<code>tsc</code>	<code>webpack</code>
Compiler settings	Specified separately	Monorepo defaults that can be overridden per project
Config files like <code>.eslintrc.js</code> , <code>.prettierrc</code> , etc.	Specified separately	Shared across monorepo
<code>nest build</code> and <code>nest start</code> commands	Target defaults automatically to the (only) project in the context	Target defaults to the <b>default project</b> in the monorepo
Libraries	Managed manually, usually via npm packaging	Built-in support, including path management and bundling

Read the sections on [Workspaces](#) and [Libraries](#) for more detailed information to help you decide which mode is most suitable for you.

## CLI command syntax

All `nest` commands follow the same format:

```
nest commandOrAlias requiredArg [optionalArg] [options]
```

For example:

```
$ nest new my-nest-project --dry-run
```

Here, `new` is the `commandOrAlias`. The `new` command has an alias of `n`. `my-nest-project` is the `requiredArg`. If a `requiredArg` is not supplied on the command line, `nest` will prompt for it. Also, `--dry-run` has an equivalent short-hand form `-d`. With this in mind, the following command is the equivalent of the above:

```
$ nest n my-nest-project -d
```

Most commands, and some options, have aliases. Try running `nest new --help` to see these options and aliases, and to confirm your understanding of the above constructs.

## Command overview

Run `nest <command> --help` for any of the following commands to see command-specific options.

See [usage](#) for detailed descriptions for each command.

Command	Alias	Description
<code>new</code>	<code>n</code>	Scaffolds a new <i>standard mode</i> application with all boilerplate files needed to run.
<code>generate</code>	<code>g</code>	Generates and/or modifies files based on a schematic.
<code>build</code>		Compiles an application or workspace into an output folder.
<code>start</code>		Compiles and runs an application (or default project in a workspace).
<code>add</code>		Imports a library that has been packaged as a <code>nest library</code> , running its install schematic.
<code>info</code>	<code>i</code>	Displays information about installed nest packages and other helpful system info.

## Requirements

Nest CLI requires a Node.js binary built with [internationalization support](#) (ICU), such as the official binaries from the [Node.js project page](#). If you encounter errors related to ICU, check that your binary meets this requirement.

```
node -p process.versions.icu
```

If the command prints `undefined`, your Node.js binary has no internationalization support.

## Workspaces

Nest has two modes for organizing code:

- **standard mode**: useful for building individual project-focused applications that have their own dependencies and settings, and don't need to optimize for sharing modules, or optimizing complex builds. This is the default mode.
- **monorepo mode**: this mode treats code artifacts as part of a lightweight **monorepo**, and may be more appropriate for teams of developers and/or multi-project environments. It automates parts of the build process to make it easy to create and compose modular components, promotes code reuse, makes integration testing easier, makes it easy to share project-wide artifacts like `eslint` rules and other configuration policies, and is easier to use than alternatives like github submodules.  
Monorepo mode employs the concept of a **workspace**, represented in the `nest-cli.json` file, to coordinate the relationship between the components of the monorepo.

It's important to note that virtually all of Nest's features are independent of your code organization mode. The **only** effect of this choice is how your projects are composed and how build artifacts are generated. All other functionality, from the CLI to core modules to add-on modules work the same in either mode.

Also, you can easily switch from **standard mode** to **monorepo mode** at any time, so you can delay this decision until the benefits of one or the other approach become more clear.

### Standard mode

When you run `nest new`, a new **project** is created for you using a built-in schematic. Nest does the following:

1. Create a new folder, corresponding to the `name` argument you provide to `nest new`
2. Populate that folder with default files corresponding to a minimal base-level Nest application. You can examine these files at the [typescript-starter](#) repository.
3. Provide additional files such as `nest-cli.json`, `package.json` and `tsconfig.json` that configure and enable various tools for compiling, testing and serving your application.

From there, you can modify the starter files, add new components, add dependencies (e.g., `npm install`), and otherwise develop your application as covered in the rest of this documentation.

### Monorepo mode

To enable monorepo mode, you start with a *standard mode* structure, and add **projects**. A project can be a full **application** (which you add to the workspace with the command `nest generate app`) or a **library** (which you add to the workspace with the command `nest generate library`). We'll discuss the details of these specific types of project components below. The key point to note now is that it is the **act of adding a project** to an existing standard mode structure that **converts it** to monorepo mode. Let's look at an example.

If we run:

```
$ nest new my-project
```

We've constructed a *standard mode* structure, with a folder structure that looks like this:

```
node_modules
src
app.controller.ts
app.module.ts
app.service.ts
main.ts
nest-cli.json
package.json
tsconfig.json
.eslintrc.js
```

We can convert this to a monorepo mode structure as follows:

```
$ cd my-project
$ nest generate app my-app
```

At this point, `nest` converts the existing structure to a **monorepo mode** structure. This results in a few important changes. The folder structure now looks like this:

```
apps
my-app
src
app.controller.ts
app.module.ts
app.service.ts
main.ts
tsconfig.app.json
my-project
src
app.controller.ts
app.module.ts
app.service.ts
main.ts
tsconfig.app.json
nest-cli.json
package.json
tsconfig.json
.eslintrc.js
```

The `generate app` schematic has reorganized the code - moving each **application** project under the `apps` folder, and adding a project-specific `tsconfig.app.json` file in each project's root folder. Our original `my-project` app has become the **default project** for the monorepo, and is now a peer with the just-added `my-app`, located under the `apps` folder. We'll cover default projects below.

**error Warning** The conversion of a standard mode structure to monorepo only works for projects that have followed the canonical Nest project structure. Specifically, during conversion, the schematic attempts to relocate the `src` and `test` folders in a project folder beneath the `apps` folder in the root. If a project does not use this structure, the conversion will fail or produce unreliable results.

## Workspace projects

A monorepo uses the concept of a workspace to manage its member entities. Workspaces are composed of **projects**. A project may be either:

- an **application**: a full Nest application including a `main.ts` file to bootstrap the application. Aside from compile and build considerations, an application-type project within a workspace is functionally identical to an application within a *standard mode* structure.
- a **library**: a library is a way of packaging a general purpose set of features (modules, providers, controllers, etc.) that can be used within other projects. A library cannot run on its own, and has no `main.ts` file. Read more about libraries [here](#).

All workspaces have a **default project** (which should be an application-type project). This is defined by the top-level "`root`" property in the `nest-cli.json` file, which points at the root of the default project (see [CLI properties](#) below for more details). Usually, this is the **standard mode** application you started with, and later converted to a monorepo using `nest generate app`. When you follow these steps, this property is populated automatically.

Default projects are used by `nest` commands like `nest build` and `nest start` when a project name is not supplied.

For example, in the above monorepo structure, running

```
$ nest start
```

will start up the `my-project` app. To start `my-app`, we'd use:

```
$ nest start my-app
```

## Applications

Application-type projects, or what we might informally refer to as just "applications", are complete Nest applications that you can run and deploy. You generate an application-type project with `nest generate app`.

This command automatically generates a project skeleton, including the standard `src` and `test` folders from the [typescript starter](#). Unlike standard mode, an application project in a monorepo does not have any of the package dependency (`package.json`) or other project configuration artifacts like `.prettierrc` and `.eslintrc.js`. Instead, the monorepo-wide dependencies and config files are used.

However, the schematic does generate a project-specific `tsconfig.app.json` file in the root folder of the project. This config file automatically sets appropriate build options, including setting the compilation output folder properly. The file extends the top-level (monorepo) `tsconfig.json` file, so you can manage global settings monorepo-wide, but override them if needed at the project level.

## Libraries

As mentioned, library-type projects, or simply "libraries", are packages of Nest components that need to be composed into applications in order to run. You generate a library-type project with `nest generate library`. Deciding what belongs in a library is an architectural design decision. We discuss libraries in depth in the [libraries](#) chapter.

## CLI properties

Nest keeps the metadata needed to organize, build and deploy both standard and monorepo structured projects in the `nest-cli.json` file. Nest automatically adds to and updates this file as you add projects, so you usually do not have to think about it or edit its contents. However, there are some settings you may want to change manually, so it's helpful to have an overview understanding of the file.

After running the steps above to create a monorepo, our `nest-cli.json` file looks like this:

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "apps/my-project/src",
  "monorepo": true,
  "root": "apps/my-project",
  "compilerOptions": {
    "webpack": true,
    "tsConfigPath": "apps/my-project/tsconfig.app.json"
  },
  "projects": {
    "my-project": {
      "type": "application",
      "root": "apps/my-project",
      "entryFile": "main",
      "sourceRoot": "apps/my-project/src",
      "compilerOptions": {
        "tsConfigPath": "apps/my-project/tsconfig.app.json"
      }
    },
    "my-app": {
      "type": "application",
      "root": "apps/my-app",
      "entryFile": "main",
      "sourceRoot": "apps/my-app/src",
      "compilerOptions": {
        "tsConfigPath": "apps/my-app/tsconfig.app.json"
      }
    }
  }
}
```

The file is divided into sections:

- a global section with top-level properties controlling standard and monorepo-wide settings
- a top level property ("projects") with metadata about each project. This section is present only for monorepo-mode structures.

The top-level properties are as follows:

- "collection": points at the collection of schematics used to generate components; you generally should not change this value
- "sourceRoot": points at the root of the source code for the single project in standard mode structures, or the *default project* in monorepo mode structures
- "compilerOptions": a map with keys specifying compiler options and values specifying the option setting; see details below
- "generateOptions": a map with keys specifying global generate options and values specifying the option setting; see details below
- "monorepo": (monorepo only) for a monorepo mode structure, this value is always `true`
- "root": (monorepo only) points at the project root of the *default project*

## Global compiler options

These properties specify the compiler to use as well as various options that affect **any** compilation step, whether as part of `nest build` or `nest start`, and regardless of the compiler, whether `tsc` or webpack.

Property Name	Property Value Type	Description
<code>webpack</code>	boolean	If <code>true</code> , use <a href="#">webpack compiler</a> . If <code>false</code> or not present, use <code>tsc</code> . In monorepo mode, the default is <code>true</code> (use webpack), in standard mode, the default is <code>false</code> (use <code>tsc</code> ). See below for details. (deprecated: use <code>builder</code> instead)
<code>tsConfigPath</code>	string	(monorepo only) Points at the file containing the <code>tsconfig.json</code> settings that will be used when <code>nest build</code> or <code>nest start</code> is called without a <code>project</code> option (e.g., when the default project is built or started).
<code>webpackConfigPath</code>	string	Points at a webpack options file. If not specified, Nest looks for the file <code>webpack.config.js</code> . See below for more details.
<code>deleteOutDir</code>	boolean	If <code>true</code> , whenever the compiler is invoked, it will first remove the compilation output directory (as configured in <code>tsconfig.json</code> , where the default is <code>./dist</code> ).
<code>assets</code>	array	Enables automatically distributing non-TypeScript assets whenever a compilation step begins (asset distribution does <b>not</b> happen on incremental compiles in <code>--watch</code> mode). See below for details.

Property Name	Property Value Type	Description
<code>watchAssets</code>	boolean	If <code>true</code> , run in watch-mode, watching <b>all</b> non-TypeScript assets. (For more fine-grained control of the assets to watch, see <a href="#">Assets</a> section below).
<code>manualRestart</code>	boolean	If <code>true</code> , enables the shortcut <code>rs</code> to manually restart the server. Default value is <code>false</code> .
<code>builder</code>	string/object	Instructs CLI on what <code>builder</code> to use to compile the project ( <code>tsc</code> , <code>swc</code> , or <code>webpack</code> ). To customize builder's behavior, you can pass an object containing two attributes: <code>type</code> ( <code>tsc</code> , <code>swc</code> , or <code>webpack</code> ) and <code>options</code> .
<code>typeCheck</code>	boolean	If <code>true</code> , enables type checking for SWC-driven projects (when <code>builder</code> is <code>swc</code> ). Default value is <code>false</code> .

## Global generate options

These properties specify the default generate options to be used by the `nest generate` command.

Property Name	Property Value Type	Description
<code>spec</code>	boolean or object	If the value is boolean, a value of <code>true</code> enables <code>spec</code> generation by default and a value of <code>false</code> disables it. A flag passed on the CLI command line overrides this setting, as does a project-specific <code>generateOptions</code> setting (more below). If the value is an object, each key represents a schematic name, and the boolean value determines whether the default spec generation is enabled / disabled for that specific schematic.
<code>flat</code>	boolean	If true, all generate commands will generate a flat structure

The following example uses a boolean value to specify that spec file generation should be disabled by default for all projects:

```
{
  "generateOptions": {
    "spec": false
  },
  ...
}
```

The following example uses a boolean value to specify flat file generation should be the default for all projects:

```
{  
  "generateOptions": {  
    "flat": true  
  },  
  ...  
}
```

In the following example, `spec` file generation is disabled only for `service` schematics (e.g., `nest generate service...`):

```
{  
  "generateOptions": {  
    "spec": {  
      "service": false  
    }  
  },  
  ...  
}
```

**warning** **Warning** When specifying the `spec` as an object, the key for the generation schematic does not currently support automatic alias handling. This means that specifying a key as for example `service: false` and trying to generate a service via the alias `s`, the spec would still be generated. To make sure both the normal schematic name and the alias work as intended, specify both the normal command name as well as the alias, as seen below.

```
{  
  "generateOptions": {  
    "spec": {  
      "service": false,  
      "s": false  
    }  
  },  
  ...  
}
```

## Project-specific generate options

In addition to providing global generate options, you may also specify project-specific generate options. The project specific generate options follow the exact same format as the global generate options, but are specified directly on each project.

Project-specific generate options override global generate options.

```
{  
  "projects": {
```

```
"cats-project": {  
  "generateOptions": {  
    "spec": {  
      "service": false  
    }  
  },  
  ...  
},  
...  
}
```

warning **Warning** The order of precedence for generate options is as follows. Options specified on the CLI command line take precedence over project-specific options. Project-specific options override global options.

## Specified compiler

The reason for the different default compilers is that for larger projects (e.g., more typical in a monorepo) webpack can have significant advantages in build times and in producing a single file bundling all project components together. If you wish to generate individual files, set "webpack" to `false`, which will cause the build process to use `tsc` (or `swc`).

## Webpack options

The webpack options file can contain standard [webpack configuration options](#). For example, to tell webpack to bundle `node_modules` (which are excluded by default), add the following to `webpack.config.js`:

```
module.exports = {  
  externals: [],  
};
```

Since the webpack config file is a JavaScript file, you can even expose a function that takes default options and returns a modified object:

```
module.exports = function (options) {  
  return {  
    ...options,  
    externals: [],  
  };  
};
```

## Assets

TypeScript compilation automatically distributes compiler output (`.js` and `.d.ts` files) to the specified output directory. It can also be convenient to distribute non-TypeScript files, such as `.graphql` files, `images`, `.html` files and other assets. This allows you to treat `nest build` (and any initial compilation step) as a lightweight **development build** step, where you may be editing non-TypeScript files and iteratively compiling and testing. The assets should be located in the `src` folder otherwise they will not be copied.

The value of the `assets` key should be an array of elements specifying the files to be distributed. The elements can be simple strings with `glob`-like file specs, for example:

```
"assets": ["**/*.graphql"],  
"watchAssets": true,
```

For finer control, the elements can be objects with the following keys:

- `"include"`: `glob`-like file specifications for the assets to be distributed
- `"exclude"`: `glob`-like file specifications for assets to be **excluded** from the `include` list
- `"outDir"`: a string specifying the path (relative to the root folder) where the assets should be distributed. Defaults to the same output directory configured for compiler output.
- `"watchAssets"`: boolean; if `true`, run in watch mode watching specified assets

For example:

```
"assets": [  
  { "include": "**/*.graphql", "exclude": "**/omitted.graphql",  
  "watchAssets": true },  
]
```

warning **Warning** Setting `watchAssets` in a top-level `compilerOptions` property overrides any `watchAssets` settings within the `assets` property.

## Project properties

This element exists only for monorepo-mode structures. You generally should not edit these properties, as they are used by Nest to locate projects and their configuration options within the monorepo.

## Libraries

Many applications need to solve the same general problems, or re-use a modular component in several different contexts. Nest has a few ways of addressing this, but each works at a different level to solve the problem in a way that helps meet different architectural and organizational objectives.

Nest [modules](#) are useful for providing an execution context that enables sharing components within a single application. Modules can also be packaged with [npm](#) to create a reusable library that can be installed in different projects. This can be an effective way to distribute configurable, re-usable libraries that can be used by different, loosely connected or unaffiliated organizations (e.g., by distributing/installing 3rd party libraries).

For sharing code within closely organized groups (e.g., within company/project boundaries), it can be useful to have a more lightweight approach to sharing components. Monorepos have arisen as a construct to enable that, and within a monorepo, a [library](#) provides a way to share code in an easy, lightweight fashion. In a Nest monorepo, using libraries enables easy assembly of applications that share components. In fact, this encourages decomposition of monolithic applications and development processes to focus on building and composing modular components.

### Nest libraries

A Nest library is a Nest project that differs from an application in that it cannot run on its own. A library must be imported into a containing application in order for its code to execute. The built-in support for libraries described in this section is only available for [monorepos](#) (standard mode projects can achieve similar functionality using npm packages).

For example, an organization may develop an [AuthModule](#) that manages authentication by implementing company policies that govern all internal applications. Rather than build that module separately for each application, or physically packaging the code with npm and requiring each project to install it, a monorepo can define this module as a library. When organized this way, all consumers of the library module can see an up-to-date version of the [AuthModule](#) as it is committed. This can have significant benefits for coordinating component development and assembly, and simplifying end-to-end testing.

### Creating libraries

Any functionality that is suitable for re-use is a candidate for being managed as a library. Deciding what should be a library, and what should be part of an application, is an architectural design decision. Creating libraries involves more than simply copying code from an existing application to a new library. When packaged as a library, the library code must be decoupled from the application. This may require [more](#) time up front and force some design decisions that you may not face with more tightly coupled code. But this additional effort can pay off when the library can be used to enable more rapid application assembly across multiple applications.

To get started with creating a library, run the following command:

```
$ nest g library my-library
```

When you run the command, the `library` schematic prompts you for a prefix (AKA alias) for the library:

```
What prefix would you like to use for the library (default: @app)?
```

This creates a new project in your workspace called `my-library`. A library-type project, like an application-type project, is generated into a named folder using a schematic. Libraries are managed under the `libs` folder of the monorepo root. Nest creates the `libs` folder the first time a library is created.

The files generated for a library are slightly different from those generated for an application. Here is the contents of the `libs` folder after executing the command above:

```
libs
my-library
src
index.ts
my-library.module.ts
my-library.service.ts
tsconfig.lib.json
```

The `nest-cli.json` file will have a new entry for the library under the `"projects"` key:

```
...
{
  "my-library": {
    "type": "library",
    "root": "libs/my-library",
    "entryFile": "index",
    "sourceRoot": "libs/my-library/src",
    "compilerOptions": {
      "tsConfigPath": "libs/my-library/tsconfig.lib.json"
    }
}
...
```

There are two differences in `nest-cli.json` metadata between libraries and applications:

- the `"type"` property is set to `"library"` instead of `"application"`
- the `"entryFile"` property is set to `"index"` instead of `"main"`

These differences key the build process to handle libraries appropriately. For example, a library exports its functions through the `index.js` file.

As with application-type projects, libraries each have their own `tsconfig.lib.json` file that extends the root (monorepo-wide) `tsconfig.json` file. You can modify this file, if necessary, to provide library-specific compiler options.

You can build the library with the CLI command:

```
$ nest build my-library
```

## Using libraries

With the automatically generated configuration files in place, using libraries is straightforward. How would we import `MyLibraryService` from the `my-library` library into the `my-project` application?

First, note that using library modules is the same as using any other Nest module. What the monorepo does is manage paths in a way that importing libraries and generating builds is now transparent. To use `MyLibraryService`, we need to import its declaring module. We can modify `my-project/src/app.module.ts` as follows to import `MyLibraryModule`.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { MyLibraryModule } from '@app/my-library';

@Module({
  imports: [MyLibraryModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Notice above that we've used a path alias of `@app` in the ES module `import` line, which was the `prefix` we supplied with the `nest g library` command above. Under the covers, Nest handles this through `tsconfig` path mapping. When adding a library, Nest updates the global (monorepo) `tsconfig.json` file's "paths" key like this:

```
"paths": {
  "@app/my-library": [
    "libs/my-library/src"
  ],
  "@app/my-library/*": [
    "libs/my-library/src/*"
  ]
}
```

So, in a nutshell, the combination of the monorepo and library features has made it easy and intuitive to include library modules into applications.

This same mechanism enables building and deploying applications that compose libraries. Once you've imported the `MyLibraryModule`, running `nest build` handles all the module resolution automatically and bundles the app along with any library dependencies, for deployment. The default compiler for a

monorepo is **webpack**, so the resulting distribution file is a single file that bundles all of the transpiled JavaScript files into a single file. You can also switch to **tsc** as described [here](#).

## CLI command reference

### nest new

Creates a new (standard mode) Nest project.

```
$ nest new <name> [options]
$ nest n <name> [options]
```

#### Description

Creates and initializes a new Nest project. Prompts for package manager.

- Creates a folder with the given `<name>`
- Populates the folder with configuration files
- Creates sub-folders for source code (`/src`) and end-to-end tests (`/test`)
- Populates the sub-folders with default files for app components and tests

#### Arguments

Argument	Description
----------	-------------

<code>&lt;name&gt;</code>	The name of the new project
---------------------------	-----------------------------

#### Options

Option	Description
--------	-------------

<code>--dry-run</code>	Reports changes that would be made, but does not change the filesystem. Alias: <code>-d</code>
<code>--skip-git</code>	Skip git repository initialization. Alias: <code>-g</code>
<code>--skip-install</code>	Skip package installation. Alias: <code>-s</code>
<code>--package-manager</code> [ <code>package-manager</code> ]	Specify package manager. Use <code>npm</code> , <code>yarn</code> , or <code>pnpm</code> . Package manager must be installed globally. Alias: <code>-p</code>
<code>--language</code> [ <code>language</code> ]	Specify programming language ( <code>TS</code> or <code>JS</code> ). Alias: <code>-l</code>
<code>--collection</code> [ <code>collectionName</code> ]	Specify schematics collection. Use package name of installed npm package containing schematic. Alias: <code>-c</code>

Option	Description	
--strict	Start the project with the following TypeScript compiler flags enabled: <code>strictNullChecks, noImplicitAny, strictBindCallApply, forceConsistentCasingInFileNames, noFallthroughCasesInSwitch</code>	
<b>nest generate</b>		
Generates and/or modifies files based on a schematic		
<pre>\$ nest generate &lt;schematic&gt; &lt;name&gt; [options] \$ nest g &lt;schematic&gt; &lt;name&gt; [options]</pre>		
<b>Arguments</b>		
Argument	Description	
<schematic>	The <code>schematic</code> or <code>collection:schematic</code> to generate. See the table below for the available schematics.	
<name>	The name of the generated component.	
<b>Schematics</b>		
Name	Alias	Description
app		Generate a new application within a monorepo (converting to monorepo if it's a standard structure).
library	lib	Generate a new library within a monorepo (converting to monorepo if it's a standard structure).
class	cl	Generate a new class.
controller	co	Generate a controller declaration.
decorator	d	Generate a custom decorator.
filter	f	Generate a filter declaration.
gateway	ga	Generate a gateway declaration.
guard	gu	Generate a guard declaration.
interface	itf	Generate an interface.
interceptor	itc	Generate an interceptor declaration.
middleware	mi	Generate a middleware declaration.
module	mo	Generate a module declaration.

Name	Alias	Description
pipe	pi	Generate a pipe declaration.
provider	pr	Generate a provider declaration.
resolver	r	Generate a resolver declaration.
resource	res	Generate a new CRUD resource. See the <a href="#">CRUD (resource) generator</a> for more details.
service	s	Generate a service declaration.

## Options

Option	Description
--dry-run	Reports changes that would be made, but does not change the filesystem. Alias: <code>-d</code>
--project [project]	Project that element should be added to. Alias: <code>-p</code>
--flat	Do not generate a folder for the element.
--collection [collectionName]	Specify schematics collection. Use package name of installed npm package containing schematic. Alias: <code>-c</code>
--spec	Enforce spec files generation (default)
--no-spec	Disable spec files generation

## nest build

Compiles an application or workspace into an output folder.

Also, the `build` command is responsible for:

- mapping paths (if using path aliases) via `tsconfig-paths`
- annotating DTOs with OpenAPI decorators (if `@nestjs/swagger` CLI plugin is enabled)
- annotating DTOs with GraphQL decorators (if `@nestjs/graphql` CLI plugin is enabled)

```
$ nest build <name> [options]
```

## Arguments

Argument	Description
<name>	The name of the project to build.

## Options

Option	Description
--path [path]	Path to <code>tsconfig</code> file. Alias <code>-p</code>
--config [path]	Path to <code>nest-cli</code> configuration file. Alias <code>-c</code>
--watch	Run in watch mode (live-reload). If you're using <code>tsc</code> for compilation, you can type <code>rs</code> to restart the application (when <code>manualRestart</code> option is set to <code>true</code> ). Alias <code>-w</code>
--builder [name]	Specify the builder to use for compilation ( <code>tsc</code> , <code>swc</code> , or <code>webpack</code> ). Alias <code>-b</code>
--webpack	Use webpack for compilation (deprecated: use <code>--builder webpack</code> instead).
--webpackPath	Path to webpack configuration.
--tsc	Force use <code>tsc</code> for compilation.

## nest start

Compiles and runs an application (or default project in a workspace).

```
$ nest start <name> [options]
```

## Arguments

Argument	Description
<name>	The name of the project to run.

## Options

Option	Description
--path [path]	Path to <code>tsconfig</code> file. Alias <code>-p</code>
--config [path]	Path to <code>nest-cli</code> configuration file. Alias <code>-c</code>
--watch	Run in watch mode (live-reload) Alias <code>-w</code>

Option	Description
--builder [name]	Specify the builder to use for compilation ( <code>tsc</code> , <code>swc</code> , or <code>webpack</code> ). Alias <code>-b</code>
--preserveWatchOutput	Keep outdated console output in watch mode instead of clearing the screen. ( <code>tsc</code> watch mode only)
--watchAssets	Run in watch mode (live-reload), watching non-TS files (assets). See <a href="#">Assets</a> for more details.
--debug [hostport]	Run in debug mode (with --inspect flag) Alias <code>-d</code>
--webpack	Use webpack for compilation. (deprecated: use <code>--builder webpack</code> instead)
--webpackPath	Path to webpack configuration.
--tsc	Force use <code>tsc</code> for compilation.
--exec [binary]	Binary to run (default: <code>node</code> ). Alias <code>-e</code>

## nest add

Imports a library that has been packaged as a **nest library**, running its install schematic.

```
$ nest add <name> [options]
```

### Arguments

Argument	Description
<name>	The name of the library to import.

## nest info

Displays information about installed nest packages and other helpful system info. For example:

```
$ nest info
```



```
\_\| \_\_/\ \_\_||\_\_/\ \_\_|\ \_\_/\ \_\_/\ \_\_/\ \_\_/\ \_\_/\ \_\_/\
```

[System Information]

OS Version : macOS High Sierra

NodeJS Version : v16.18.0

[Nest Information]

microservices version : 10.0.0

websockets version : 10.0.0

testing version : 10.0.0

common version : 10.0.0

core version : 10.0.0

## Nest CLI and scripts

This section provides additional background on how the `nest` command interacts with compilers and scripts to help DevOps personnel manage the development environment.

A Nest application is a **standard** TypeScript application that needs to be compiled to JavaScript before it can be executed. There are various ways to accomplish the compilation step, and developers/teams are free to choose a way that works best for them. With that in mind, Nest provides a set of tools out-of-the-box that seek to do the following:

- Provide a standard build/execute process, available at the command line, that "just works" with reasonable defaults.
- Ensure that the build/execute process is **open**, so developers can directly access the underlying tools to customize them using native features and options.
- Remain a completely standard TypeScript/Node.js framework, so that the entire compile/deploy/execute pipeline can be managed by any external tools that the development team chooses to use.

This goal is accomplished through a combination of the `nest` command, a locally installed TypeScript compiler, and `package.json` scripts. We describe how these technologies work together below. This should help you understand what's happening at each step of the build/execute process, and how to customize that behavior if necessary.

### The `nest` binary

The `nest` command is an OS level binary (i.e., runs from the OS command line). This command actually encompasses 3 distinct areas, described below. We recommend that you run the build (`nest build`) and execution (`nest start`) sub-commands via the `package.json` scripts provided automatically when a project is scaffolded (see [typescript starter](#) if you wish to start by cloning a repo, instead of running `nest new`).

#### Build

`nest build` is a wrapper on top of the standard `tsc` compiler or `swc` compiler (for [standard projects](#)) or the webpack bundler using the `ts-loader` (for [monorepos](#)). It does not add any other compilation features or steps except for handling `tsconfig-paths` out of the box. The reason it exists is that most developers, especially when starting out with Nest, do not need to adjust compiler options (e.g., `tsconfig.json` file) which can sometimes be tricky.

See the [nest build](#) documentation for more details.

#### Execution

`nest start` simply ensures the project has been built (same as `nest build`), then invokes the `node` command in a portable, easy way to execute the compiled application. As with builds, you are free to customize this process as needed, either using the `nest start` command and its options, or completely replacing it. The entire process is a standard TypeScript application build and execute pipeline, and you are free to manage the process as such.

See the [nest start](#) documentation for more details.

## Generation

The `nest generate` commands, as the name implies, generate new Nest projects, or components within them.

## Package scripts

Running the `nest` commands at the OS command level requires that the `nest` binary be installed globally. This is a standard feature of npm, and outside of Nest's direct control. One consequence of this is that the globally installed `nest` binary is **not** managed as a project dependency in `package.json`. For example, two different developers can be running two different versions of the `nest` binary. The standard solution for this is to use package scripts so that you can treat the tools used in the build and execute steps as development dependencies.

When you run `nest new`, or clone the [typescript starter](#), Nest populates the new project's `package.json` scripts with commands like `build` and `start`. It also installs the underlying compiler tools (such as `typescript`) as **dev dependencies**.

You run the build and execute scripts with commands like:

```
$ npm run build
```

and

```
$ npm run start
```

These commands use npm's script running capabilities to execute `nest build` or `nest start` using the **locally installed** `nest` binary. By using these built-in package scripts, you have full dependency management over the Nest CLI commands\*. This means that, by following this **recommended** usage, all members of your organization can be assured of running the same version of the commands.

\*This applies to the `build` and `start` commands. The `nest new` and `nest generate` commands aren't part of the build/execute pipeline, so they operate in a different context, and do not come with built-in `package.json` scripts.

For most developers/teams, it is recommended to utilize the package scripts for building and executing their Nest projects. You can fully customize the behavior of these scripts via their options (`--path`, `--webpack`, `--webpackPath`) and/or customize the `tsc` or webpack compiler options files (e.g., `tsconfig.json`) as needed. You are also free to run a completely custom build process to compile the TypeScript (or even to execute TypeScript directly with `ts-node`).

## Backward compatibility

Because Nest applications are pure TypeScript applications, previous versions of the Nest build/execute scripts will continue to operate. You are not required to upgrade them. You can choose to take advantage of the new `nest build` and `nest start` commands when you are ready, or continue running previous or customized scripts.

## Migration

While you are not required to make any changes, you may want to migrate to using the new CLI commands instead of using tools such as `tsc-watch` or `ts-node`. In this case, simply install the latest version of the `@nestjs/cli`, both globally and locally:

```
$ npm install -g @nestjs/cli
$ cd /some/project/root/folder
$ npm install -D @nestjs/cli
```

You can then replace the `scripts` defined in `package.json` with the following ones:

```
"build": "nest build",
"start": "nest start",
"start:dev": "nest start --watch",
"start:debug": "nest start --debug --watch",
```

## Introduction

The [OpenAPI](#) specification is a language-agnostic definition format used to describe RESTful APIs. Nest provides a dedicated [module](#) which allows generating such a specification by leveraging decorators.

## Installation

To begin using it, we first install the required dependency.

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

## Bootstrap

Once the installation process is complete, open the [main.ts](#) file and initialize Swagger using the [SwaggerModule](#) class:

```
@@filename(main)
import { NestFactory } from '@nestjs/core';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { AppModule } from './app.module';

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

  const config = new DocumentBuilder()
    .setTitle('Cats example')
    .setDescription('The cats API description')
    .setVersion('1.0')
    .addTag('cats')
    .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api', app, document);

  await app.listen(3000);
}

bootstrap();
```

**info Hint** [document](#) (returned by the [SwaggerModule#createDocument\(\)](#) method) is a serializable object conforming to [OpenAPI Document](#). Instead of hosting it via HTTP, you could also save it as a JSON/YAML file, and consume it in different ways.

The [DocumentBuilder](#) helps to structure a base document that conforms to the OpenAPI Specification. It provides several methods that allow setting such properties as title, description, version, etc. In order to create a full document (with all HTTP routes defined) we use the [createDocument\(\)](#) method of the [SwaggerModule](#) class. This method takes two arguments, an application instance and a Swagger options

object. Alternatively, we can provide a third argument, which should be of type [SwaggerDocumentOptions](#). More on this in the [Document options section](#).

Once we create a document, we can call the `setup()` method. It accepts:

1. The path to mount the Swagger UI
2. An application instance
3. The document object instantiated above
4. Optional configuration parameter (read more [here](#))

Now you can run the following command to start the HTTP server:

```
$ npm run start
```

While the application is running, open your browser and navigate to <http://localhost:3000/api>. You should see the Swagger UI.



As you can see, the [SwaggerModule](#) automatically reflects all of your endpoints.

**info Hint** To generate and download a Swagger JSON file, navigate to <http://localhost:3000/api-json> (assuming that your Swagger documentation is available under <http://localhost:3000/api>).

**warning Warning** When using [fastify](#) and [helmet](#), there may be a problem with [CSP](#), to solve this collision, configure the CSP as shown below:

```
app.register(helmet, {
  contentSecurityPolicy: {
    directives: {
      defaultSrc: [`'self'`],
      styleSrc: [`'self'`, `'unsafe-inline'`],
      imgSrc: [`'self'`, 'data:', 'validator.swagger.io'],
      scriptSrc: [`'self'`, `https: 'unsafe-inline'`],
    },
  },
});

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

## Document options

When creating a document, it is possible to provide some extra options to fine tune the library's behavior. These options should be of type [SwaggerDocumentOptions](#), which can be the following:

```
export interface SwaggerDocumentOptions {  
    /**  
     * List of modules to include in the specification  
     */  
    include?: Function[];  
  
    /**  
     * Additional, extra models that should be inspected and included in the  
     * specification  
     */  
    extraModels?: Function[];  
  
    /**  
     * If `true`, swagger will ignore the global prefix set through  
     * `setGlobalPrefix()` method  
     */  
    ignoreGlobalPrefix?: boolean;  
  
    /**  
     * If `true`, swagger will also load routes from the modules imported by  
     * `include` modules  
     */  
    deepScanRoutes?: boolean;  
  
    /**  
     * Custom operationIdFactory that will be used to generate the  
     * `operationId`  
     * based on the `controllerKey` and `methodKey`  
     * @default () => controllerKey_methodKey  
     */  
    operationIdFactory?: (controllerKey: string, methodKey: string) =>  
        string;  
}
```

For example, if you want to make sure that the library generates operation names like `createUser` instead of `UserController_createUser`, you can set the following:

```
const options: SwaggerDocumentOptions = {  
    operationIdFactory: (  
        controllerKey: string,  
        methodKey: string  
    ) => methodKey  
};  
const document = SwaggerModule.createDocument(app, config, options);
```

## Setup options

You can configure Swagger UI by passing the options object which fulfills the `ExpressSwaggerCustomOptions` (if you use express) interface as a fourth argument of the `SwaggerModule#setup` method.

```
export interface ExpressSwaggerCustomOptions {  
    explorer?: boolean;  
    swaggerOptions?: Record<string, any>;  
    customCss?: string;  
    customCssUrl?: string;  
    customJs?: string;  
    customfavIcon?: string;  
    swaggerUrl?: string;  
    customSiteTitle?: string;  
    validatorUrl?: string;  
    url?: string;  
    urls?: Record<'url' | 'name', string>[];  
    patchDocumentOnRequest?: <TRequest = any, TResponse = any> (req:  
        TRequest, res: TResponse, document: OpenAPIObject) => OpenAPIObject;  
}
```

## Example

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

## Types and parameters

The `SwaggerModule` searches for all `@Body()`, `@Query()`, and `@Param()` decorators in route handlers to generate the API document. It also creates corresponding model definitions by taking advantage of reflection. Consider the following code:

```
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

**info Hint** To explicitly set the body definition use the `@ApiBody()` decorator (imported from the `@nestjs/swagger` package).

Based on the `CreateCatDto`, the following model definition Swagger UI will be created:



As you can see, the definition is empty although the class has a few declared properties. In order to make the class properties visible to the `SwaggerModule`, we have to either annotate them with the `@ApiProperty()` decorator or use the CLI plugin (read more in the **Plugin** section) which will do it automatically:

```
import { ApiProperty } from '@nestjs/swagger';

export class CreateCatDto {
  @ApiProperty()
  name: string;

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}
```

**info Hint** Instead of manually annotating each property, consider using the Swagger plugin (see **Plugin** section) which will automatically provide this for you.

Let's open the browser and verify the generated `CreateCatDto` model:



In addition, the `@ApiProperty()` decorator allows setting various **Schema Object** properties:

```
@ApiProperty({
  description: 'The age of a cat',
```

```
minimum: 1,  
default: 1,  
}  
age: number;
```

**info Hint** Instead of explicitly typing the `{ "@ApiProperty({ required: false })" }` you can use the `@ApiPropertyOptional()` short-hand decorator.

In order to explicitly set the type of the property, use the `type` key:

```
@ApiProperty({  
  type: Number,  
})  
age: number;
```

## Arrays

When the property is an array, we must manually indicate the array type as shown below:

```
@ApiProperty({ type: [String] })  
names: string[];
```

**info Hint** Consider using the Swagger plugin (see [Plugin](#) section) which will automatically detect arrays.

Either include the type as the first element of an array (as shown above) or set the `isArray` property to `true`.

## Circular dependencies

When you have circular dependencies between classes, use a lazy function to provide the `SwaggerModule` with type information:

```
@ApiProperty({ type: () => Node })  
node: Node;
```

**info Hint** Consider using the Swagger plugin (see [Plugin](#) section) which will automatically detect circular dependencies.

## Generics and interfaces

Since TypeScript does not store metadata about generics or interfaces, when you use them in your DTOs, `SwaggerModule` may not be able to properly generate model definitions at runtime. For instance, the following code won't be correctly inspected by the Swagger module:

```
createBulk(@Body() usersDto: CreateUserDto[])
```

In order to overcome this limitation, you can set the type explicitly:

```
@ApiBody({ type: [CreateUserDto] })
createBulk(@Body() usersDto: CreateUserDto[])
```

## Enums

To identify an `enum`, we must manually set the `enum` property on the `@ApiProperty` with an array of values.

```
@ApiProperty({ enum: ['Admin', 'Moderator', 'User'] })
role: UserRole;
```

Alternatively, define an actual TypeScript enum as follows:

```
export enum UserRole {
  Admin = 'Admin',
  Moderator = 'Moderator',
  User = 'User',
}
```

You can then use the enum directly with the `@Query()` parameter decorator in combination with the `@ApiQuery()` decorator.

```
@ApiQuery({ name: 'role', enum: UserRole })
async filterByRole(@Query('role') role: UserRole = UserRole.User) {}
```



With `isArray` set to `true`, the `enum` can be selected as a **multi-select**:



## Enums schema

By default, the `enum` property will add a raw definition of `Enum` on the `parameter`.

- breed:
  - type: 'string'

```
enum:
  - Persian
  - Tabby
  - Siamese
```

The above specification works fine for most cases. However, if you are utilizing a tool that takes the specification as **input** and generates **client-side** code, you might run into a problem with the generated code containing duplicated **enums**. Consider the following code snippet:

```
// generated client-side code
export class CatDetail {
  breed: CatDetailEnum;
}

export class CatInformation {
  breed: CatInformationEnum;
}

export enum CatDetailEnum {
  Persian = 'Persian',
  Tabby = 'Tabby',
  Siamese = 'Siamese',
}

export enum CatInformationEnum {
  Persian = 'Persian',
  Tabby = 'Tabby',
  Siamese = 'Siamese',
}
```

**info Hint** The above snippet is generated using a tool called [NSwag](#).

You can see that now you have two **enums** that are exactly the same. To address this issue, you can pass an **enumName** along with the **enum** property in your decorator.

```
export class CatDetail {
  @ApiProperty({ enum: CatBreed, enumName: 'CatBreed' })
  breed: CatBreed;
}
```

The **enumName** property enables [@nestjs/swagger](#) to turn **CatBreed** into its own **schema** which in turns makes **CatBreed** enum reusable. The specification will look like the following:

```
CatDetail:
  type: 'object'
  properties:
    ...
```

```

    - breed:
      schema:
        $ref: '#/components/schemas/CatBreed'
CatBreed:
  type: string
  enum:
    - Persian
    - Tabby
    - Siamese

```

**info Hint** Any **decorator** that takes `enum` as a property will also take `enumName`.

## Raw definitions

In some specific scenarios (e.g., deeply nested arrays, matrices), you may want to describe your type by hand.

```

@ApiProperty({
  type: 'array',
  items: {
    type: 'array',
    items: {
      type: 'number',
    },
  },
})
coords: number[][];
```

Likewise, in order to define your input/output content manually in controller classes, use the `schema` property:

```

@ApiBody({
  schema: {
    type: 'array',
    items: {
      type: 'array',
      items: {
        type: 'number',
      },
    },
  },
})
async create(@Body() coords: number[][]) {}
```

## Extra models

To define additional models that are not directly referenced in your controllers but should be inspected by the Swagger module, use the `@ApiExtraModels()` decorator:

```
@ApiExtraModels(ExtraModel)
export class CreateCatDto {}
```

**info Hint** You only need to use `@ApiExtraModels()` once for a specific model class.

Alternatively, you can pass an options object with the `extraModels` property specified to the `SwaggerModule#createDocument()` method, as follows:

```
const document = SwaggerModule.createDocument(app, options, {
  extraModels: [ExtraModel],
});
```

To get a reference (`$ref`) to your model, use the `getSchemaPath(ExtraModel)` function:

```
'application/vnd.api+json': {
  schema: { $ref: getSchemaPath(ExtraModel) },
},
```

## oneOf, anyOf, allOf

To combine schemas, you can use the `oneOf`, `anyOf` or `allOf` keywords ([read more](#)).

```
@ApiProperty({
  oneOf: [
    { $ref: getSchemaPath(Cat) },
    { $ref: getSchemaPath(Dog) },
  ],
})
pet: Cat | Dog;
```

If you want to define a polymorphic array (i.e., an array whose members span multiple schemas), you should use a raw definition (see above) to define your type by hand.

```
type Pet = Cat | Dog;

@ApiProperty({
  type: 'array',
  items: {
    oneOf: [
      { $ref: getSchemaPath(Cat) },

```

```
{ $ref: getSchemaPath(Dog) },  
],  
,  
}  
}  
pets: Pet[];
```

**info Hint** The `getSchemaPath()` function is imported from `@nestjs/swagger`.

Both `Cat` and `Dog` must be defined as extra models using the `@ApiExtraModels()` decorator (at the class-level).

## Operations

In OpenAPI terms, paths are endpoints (resources), such as `/users` or `/reports/summary`, that your API exposes, and operations are the HTTP methods used to manipulate these paths, such as `GET`, `POST` or `DELETE`.

## Tags

To attach a controller to a specific tag, use the `@ApiTags(...tags)` decorator.

```
@ApiTags('cats')
@Controller('cats')
export class CatsController {}
```

## Headers

To define custom headers that are expected as part of the request, use `@ApiHeader()`.

```
@ApiHeader({
  name: 'X-MyHeader',
  description: 'Custom header',
})
@Controller('cats')
export class CatsController {}
```

## Responses

To define a custom HTTP response, use the `@ApiResponse()` decorator.

```
@Post()
@ApiResponse({ status: 201, description: 'The record has been successfully created.'})
@ApiResponse({ status: 403, description: 'Forbidden.'})
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

Nest provides a set of short-hand **API response** decorators that inherit from the `@ApiResponse` decorator:

- `@ApiOkResponse()`
- `@ApiCreatedResponse()`
- `@ApiAcceptedResponse()`
- `@ApiNoContentResponse()`

- `@ApiMovedPermanentlyResponse()`
- `@ApiFoundResponse()`
- `@ApiBadRequestResponse()`
- `@ApiUnauthorizedResponse()`
- `@ApiNotFoundResponse()`
- `@ApiForbiddenResponse()`
- `@ApiMethodNotAllowedResponse()`
- `@ApiNotAcceptableResponse()`
- `@ApiRequestTimeoutResponse()`
- `@ApiConflictResponse()`
- `@ApiPreconditionFailedResponse()`
- `@ApiTooManyRequestsResponse()`
- `@ApiGoneResponse()`
- `@ApiPayloadTooLargeResponse()`
- `@ApiUnsupportedMediaTypeResponse()`
- `@ApiUnprocessableEntityResponse()`
- `@ApiInternalServerErrorResponse()`
- `@ApiNotImplementedResponse()`
- `@ApiBadGatewayResponse()`
- `@ApiServiceUnavailableResponse()`
- `@ApiGatewayTimeoutResponse()`
- `@ApiDefaultResponse()`

```
@Post()
@ApiCreatedResponse({ description: 'The record has been successfully
created.'})
@ApiForbiddenResponse({ description: 'Forbidden.'})
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

To specify a return model for a request, we must create a class and annotate all properties with the `@ApiProperty()` decorator.

```
export class Cat {
  @ApiProperty()
  id: number;

  @ApiProperty()
  name: string;

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}
```

Then the **Cat** model can be used in combination with the **type** property of the response decorator.

```
@ApiTags('cats')
@Controller('cats')
export class CatsController {
  @Post()
  @ApiCreatedResponse({
    description: 'The record has been successfully created.',
    type: Cat,
  })
  async create(@Body() createCatDto: CreateCatDto): Promise<Cat> {
    return this.catsService.create(createCatDto);
  }
}
```

Let's open the browser and verify the generated **Cat** model:



## File upload

You can enable file upload for a specific method with the **@ApiBody** decorator together with **@ApiConsumes()**. Here's a full example using the [File Upload](#) technique:

```
@UseInterceptors(FileInterceptor('file'))
@ApiConsumes('multipart/form-data')
@ApiBody({
  description: 'List of cats',
  type: FileUploadDto,
})
uploadFile(@UploadedFile() file) {}
```

Where **FileUploadDto** is defined as follows:

```
class FileUploadDto {
  @ApiProperty({ type: 'string', format: 'binary' })
  file: any;
}
```

To handle multiple files uploading, you can define **FilesUploadDto** as follows:

```
class FilesUploadDto {
  @ApiProperty({ type: 'array', items: { type: 'string', format: 'binary' } })
}
```

```
    files: any[];  
}
```

## Extensions

To add an Extension to a request use the `@ApiExtension()` decorator. The extension name must be prefixed with `x-`.

```
@ApiExtension('x-foo', { hello: 'world' })
```

## Advanced: Generic ApiResponse

With the ability to provide [Raw Definitions](#), we can define Generic schema for Swagger UI. Assume we have the following DTO:

```
export class PaginatedDto<TData> {  
  @ApiProperty()  
  total: number;  
  
  @ApiProperty()  
  limit: number;  
  
  @ApiProperty()  
  offset: number;  
  
  results: TData[];  
}
```

We skip decorating `results` as we will be providing a raw definition for it later. Now, let's define another DTO and name it, for example, `CatDto`, as follows:

```
export class CatDto {  
  @ApiProperty()  
  name: string;  
  
  @ApiProperty()  
  age: number;  
  
  @ApiProperty()  
  breed: string;  
}
```

With this in place, we can define a `PaginatedDto<CatDto>` response, as follows:

```

@ApiResponse({
  schema: {
    allOf: [
      { $ref: getSchemaPath(PaginatedDto) },
      {
        properties: {
          results: {
            type: 'array',
            items: { $ref: getSchemaPath(CatDto) },
          },
        },
      },
    ],
  },
})
async findAll(): Promise<PaginatedDto<CatDto>> {}

```

In this example, we specify that the response will have allOf `PaginatedDto` and the `results` property will be of type `Array<CatDto>`.

- `getSchemaPath()` function that returns the OpenAPI Schema path from within the OpenAPI Spec File for a given model.
- `allOf` is a concept that OAS 3 provides to cover various Inheritance related use-cases.

Lastly, since `PaginatedDto` is not directly referenced by any controller, the `SwaggerModule` will not be able to generate a corresponding model definition just yet. In this case, we must add it as an [Extra Model](#). For example, we can use the `@ApiExtraModels()` decorator on the controller level, as follows:

```

@Controller('cats')
@ApiExtraModels(PaginatedDto)
export class CatsController {}

```

If you run Swagger now, the generated `swagger.json` for this specific endpoint should have the following response defined:

```

"responses": {
  "200": {
    "description": "",
    "content": {
      "application/json": {
        "schema": {
          "allOf": [
            {
              "$ref": "#/components/schemas/PaginatedDto"
            },
            {
              "properties": {
                "results": {

```

```
        "$ref": "#/components/schemas/CatDto"
    }
}
]
}
}
}
}
```

To make it reusable, we can create a custom decorator for `PaginatedDto`, as follows:

```
export const ApiPaginatedResponse = <TModel extends Type<any>>(
  model: TModel,
) => {
  return applyDecorators(
    ApiExtraModels(model),
    ApiOkResponse({
      schema: {
        allOf: [
          { $ref: getSchemaPath(PaginatedDto) },
          {
            properties: {
              results: {
                type: 'array',
                items: { $ref: getSchemaPath(model) },
              },
            },
          },
          ],
        },
      );
    });
};
```

info Hint `Type<any>` interface and `applyDecorators` function are imported from the `@nestjs/common` package.

To ensure that `SwaggerModule` will generate a definition for our model, we must add it as an extra model, like we did earlier with the `PaginatedDto` in the controller.

With this in place, we can use the custom `@ApiPaginatedResponse()` decorator on our endpoint:

```
@ApiPaginatedResponse(CatDto)
async findAll(): Promise<PaginatedDto<CatDto>> {}
```

For client generation tools, this approach poses an ambiguity in how the `PaginatedResponse<TModel>` is being generated for the client. The following snippet is an example of a client generator result for the above `GET /` endpoint.

```
// Angular
findAll(): Observable<{ total: number, limit: number, offset: number,
results: CatDto[] }>
```

As you can see, the **Return Type** here is ambiguous. To workaround this issue, you can add a `title` property to the `schema` for `ApiPaginatedResponse`:

```
export const ApiPaginatedResponse = <TModel extends Type<any>>(model:
TModel) => {
  return applyDecorators(
    ApiOkResponse({
      schema: {
        title: `PaginatedResponseOf${model.name}`
        allOf: [
          // ...
        ],
        },
      }),
    );
};
```

Now the result of the client generator tool will become:

```
// Angular
findAll(): Observable<PaginatedResponseOfCatDto>
```

## Security

To define which security mechanisms should be used for a specific operation, use the `@ApiSecurity()` decorator.

```
@ApiSecurity('basic')
@Controller('cats')
export class CatsController {}
```

Before you run your application, remember to add the security definition to your base document using `DocumentBuilder`:

```
const options = new DocumentBuilder().addSecurity('basic', {
  type: 'http',
  scheme: 'basic',
});
```

Some of the most popular authentication techniques are built-in (e.g., `basic` and `bearer`) and therefore you don't have to define security mechanisms manually as shown above.

### Basic authentication

To enable basic authentication, use `@ApiBasicAuth()`.

```
@ApiBasicAuth()
@Controller('cats')
export class CatsController {}
```

Before you run your application, remember to add the security definition to your base document using `DocumentBuilder`:

```
const options = new DocumentBuilder().addBasicAuth();
```

### Bearer authentication

To enable bearer authentication, use `@ApiBearerAuth()`.

```
@ApiBearerAuth()
@Controller('cats')
export class CatsController {}
```

Before you run your application, remember to add the security definition to your base document using [DocumentBuilder](#):

```
const options = new DocumentBuilder().addBearerAuth();
```

## OAuth2 authentication

To enable OAuth2, use [@ApiOAuth2\(\)](#).

```
@ApiOAuth2(['pets:write'])
@Controller('cats')
export class CatsController {}
```

Before you run your application, remember to add the security definition to your base document using [DocumentBuilder](#):

```
const options = new DocumentBuilder().addOAuth2();
```

## Cookie authentication

To enable cookie authentication, use [@ApiCookieAuth\(\)](#).

```
@ApiCookieAuth()
@Controller('cats')
export class CatsController {}
```

Before you run your application, remember to add the security definition to your base document using [DocumentBuilder](#):

```
const options = new DocumentBuilder().addCookieAuth('optional-session-
id');
```

## Mapped types

As you build out features like **CRUD** (Create/Read/Update/Delete) it's often useful to construct variants on a base entity type. Nest provides several utility functions that perform type transformations to make this task more convenient.

### Partial

When building input validation types (also called DTOs), it's often useful to build **create** and **update** variations on the same type. For example, the **create** variant may require all fields, while the **update** variant may make all fields optional.

Nest provides the **PartialType()** utility function to make this task easier and minimize boilerplate.

The **PartialType()** function returns a type (class) with all the properties of the input type set to optional. For example, suppose we have a **create** type as follows:

```
import { ApiProperty } from '@nestjs/swagger';

export class CreateCatDto {
  @ApiProperty()
  name: string;

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}
```

By default, all of these fields are required. To create a type with the same fields, but with each one optional, use **PartialType()** passing the class reference (**CreateCatDto**) as an argument:

```
export class UpdateCatDto extends PartialType(CreateCatDto) {}
```

**info Hint** The **PartialType()** function is imported from the **@nestjs/swagger** package.

### Pick

The **PickType()** function constructs a new type (class) by picking a set of properties from an input type. For example, suppose we start with a type like:

```
import { ApiProperty } from '@nestjs/swagger';

export class CreateCatDto {
  @ApiProperty()
  name: string;
```

```

name: string;

@ApiProperty()
age: number;

@apiProperty()
breed: string;
}

```

We can pick a set of properties from this class using the `PickType()` utility function:

```
export class UpdateCatAgeDto extends PickType(CreateCatDto, ['age'] as const) {}
```

**info Hint** The `PickType()` function is imported from the `@nestjs/swagger` package.

## Omit

The `OmitType()` function constructs a type by picking all properties from an input type and then removing a particular set of keys. For example, suppose we start with a type like:

```

import { ApiProperty } from '@nestjs/swagger';

export class CreateCatDto {
  @ApiProperty()
  name: string;

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}

```

We can generate a derived type that has every property **except** `name` as shown below. In this construct, the second argument to `OmitType` is an array of property names.

```
export class UpdateCatDto extends OmitType(CreateCatDto, ['name'] as const) {}
```

**info Hint** The `OmitType()` function is imported from the `@nestjs/swagger` package.

## Intersection

The `IntersectionType()` function combines two types into one new type (class). For example, suppose we start with two types like:

```
import { ApiProperty } from '@nestjs/swagger';

export class CreateCatDto {
  @ApiProperty()
  name: string;

  @ApiProperty()
  breed: string;
}

export class AdditionalCatInfo {
  @ApiProperty()
  color: string;
}
```

We can generate a new type that combines all properties in both types.

```
export class UpdateCatDto extends IntersectionType(
  CreateCatDto,
  AdditionalCatInfo,
) {}
```

**info Hint** The `IntersectionType()` function is imported from the `@nestjs/swagger` package.

## Composition

The type mapping utility functions are composable. For example, the following will produce a type (class) that has all of the properties of the `CreateCatDto` type except for `name`, and those properties will be set to optional:

```
export class UpdateCatDto extends PartialType(
  OmitType(CreateCatDto, ['name'] as const),
) {}
```

## Decorators

All of the available OpenAPI decorators have an **Api** prefix to distinguish them from the core decorators. Below is a full list of the exported decorators along with a designation of the level at which the decorator may be applied.

<code>@ApiBasicAuth()</code>	Method / Controller
<code>@ApiBearerAuth()</code>	Method / Controller
<code>@ApiBody()</code>	Method
<code>@ApiConsumes()</code>	Method / Controller
<code>@ApiCookieAuth()</code>	Method / Controller
<code>@ApiExcludeController()</code>	Controller
<code>@ApiExcludeEndpoint()</code>	Method
<code>@ApiExtension()</code>	Method
<code>@ApiExtraModels()</code>	Method / Controller
<code>@ApiHeader()</code>	Method / Controller
<code>@ApiHideProperty()</code>	Model
<code>@ApiOAuth2()</code>	Method / Controller
<code>@ApiOperation()</code>	Method
<code>@ApiParam()</code>	Method
<code>@ApiProduces()</code>	Method / Controller
<code>@ApiProperty()</code>	Model
<code>@ApiPropertyOptional()</code>	Model
<code>@ApiQuery()</code>	Method
<code>@ApiResponse()</code>	Method / Controller
<code>@ApiSecurity()</code>	Method / Controller
<code>@ApiTags()</code>	Method / Controller

## CLI Plugin

TypeScript's metadata reflection system has several limitations which make it impossible to, for instance, determine what properties a class consists of or recognize whether a given property is optional or required. However, some of these constraints can be addressed at compilation time. Nest provides a plugin that enhances the TypeScript compilation process to reduce the amount of boilerplate code required.

**info Hint** This plugin is **opt-in**. If you prefer, you can declare all decorators manually, or only specific decorators where you need them.

### Overview

The Swagger plugin will automatically:

- annotate all DTO properties with `@ApiProperty` unless `@ApiHideProperty` is used
- set the `required` property depending on the question mark (e.g. `name?: string` will set `required: false`)
- set the `type` or `enum` property depending on the type (supports arrays as well)
- set the `default` property based on the assigned default value
- set several validation rules based on `class-validator` decorators (if `classValidatorShim` set to `true`)
- add a response decorator to every endpoint with a proper status and `type` (response model)
- generate descriptions for properties and endpoints based on comments (if `introspectComments` set to `true`)
- generate example values for properties based on comments (if `introspectComments` set to `true`)

Please, note that your filenames **must have** one of the following suffixes: `'.dto.ts'`, `'.entity.ts'` (e.g., `create-user.dto.ts`) in order to be analysed by the plugin.

If you are using a different suffix, you can adjust the plugin's behavior by specifying the `dtoFileNameSuffix` option (see below).

Previously, if you wanted to provide an interactive experience with the Swagger UI, you had to duplicate a lot of code to let the package know how your models/components should be declared in the specification. For example, you could define a simple `CreateUserDto` class as follows:

```
export class CreateUserDto {  
  @ApiProperty()  
  email: string;  
  
  @ApiProperty()  
  password: string;  
  
  @ApiProperty({ enum: RoleEnum, default: [], isArray: true })  
  roles: RoleEnum[] = [];  
  
  @ApiProperty({ required: false, default: true })  
  isEnabled?: boolean = true;  
}
```

While not a significant issue with medium-sized projects, it becomes verbose & hard to maintain once you have a large set of classes.

By [enabling the Swagger plugin](#), the above class definition can be declared simply:

```
export class CreateUserDto {  
  email: string;  
  password: string;  
  roles: RoleEnum[] = [];  
  isEnabled?: boolean = true;  
}
```

**info Note** The Swagger plugin will derive the `@ApiProperty()` annotations from the TypeScript types and class-validator decorators. This helps in clearly describing your API for the generated Swagger UI documentation. However, the validation at runtime would still be handled by class-validator decorators. So, it is required to continue using validators like `IsEmail()`, `IsNumber()`, etc.

Hence, if you intend to rely on automatic annotations for generating documentations and still wish for runtime validations, then the class-validator decorators are still necessary.

**info Hint** When using `mapped types utilities` (like `PartialType`) in DTOs import them from `@nestjs/swagger` instead of `@nestjs/mapped-types` for the plugin to pick up the schema.

The plugin adds appropriate decorators on the fly based on the **Abstract Syntax Tree**. Thus you won't have to struggle with `@ApiProperty` decorators scattered throughout the code.

**info Hint** The plugin will automatically generate any missing swagger properties, but if you need to override them, you simply set them explicitly via `@ApiProperty()`.

## Comments introspection

With the comments introspection feature enabled, CLI plugin will generate descriptions and example values for properties based on comments.

For example, given an example `roles` property:

```
/**  
 * A list of user's roles  
 * @example ['admin']  
 */  
@ApiModelProperty({  
  description: `A list of user's roles`,  
  example: ['admin'],  
})  
roles: RoleEnum[] = [];
```

You must duplicate both description and example values. With `introspectComments` enabled, the CLI plugin can extract these comments and automatically provide descriptions (and examples, if defined) for properties. Now, the above property can be declared simply as follows:

```
/**  
 * A list of user's roles  
 * @example ['admin']  
 */  
roles: RoleEnum[] = [];
```

There are `dtoKeyOfComment` and `controllerKeyOfComment` plugin options that you can use to customize how the plugin will set the value for `ApiProperty` and `ApiOperation` decorators respectively. Take a look at the following example:

```
export class SomeController {  
  /**  
   * Create some resource  
   */  
  @Post()  
  create() {}  
}
```

By default, these options are set to `"description"`. This means the plugin will assign `"Create some resource"` to `description` key on the `ApiOperation` operator. Like so:

```
@ApiOperation({ description: "Create some resource" })
```

**info Hint** For models, the same logic applies but to `ApiProperty` decorator instead.

## Using the CLI plugin

To enable the plugin, open `nest-cli.json` (if you use [Nest CLI](#)) and add the following `plugins` configuration:

```
{  
  "collection": "@nestjs/schematics",  
  "sourceRoot": "src",  
  "compilerOptions": {  
    "plugins": ["@nestjs/swagger"]  
  }  
}
```

You can use the `options` property to customize the behavior of the plugin.

```
"plugins": [
  {
    "name": "@nestjs/swagger",
    "options": {
      "classValidatorShim": false,
      "introspectComments": true
    }
  }
]
```

The `options` property has to fulfill the following interface:

```
export interface PluginOptions {
  dtoFileNameSuffix?: string[];
  controllerFileNameSuffix?: string[];
  classValidatorShim?: boolean;
  dtoKeyOfComment?: string;
  controllerKeyOfComment?: string;
  introspectComments?: boolean;
}
```

Option	Default	Description
<code>dtoFileNameSuffix</code>	<code>['.dto.ts', '.entity.ts']</code>	DTO (Data Transfer Object) files suffix
<code>controllerFileNameSuffix</code>	<code>.controller.ts</code>	Controller files suffix
<code>classValidatorShim</code>	<code>true</code>	If set to true, the module will reuse <code>class-validator</code> validation decorators (e.g. <code>@Max(10)</code> will add <code>max: 10</code> to schema definition)
<code>dtoKeyOfComment</code>	<code>'description'</code>	The property key to set the comment text to on <code>ApiProperty</code> .
<code>controllerKeyOfComment</code>	<code>'description'</code>	The property key to set the comment text to on <code>ApiOperation</code> .
<code>introspectComments</code>	<code>false</code>	If set to true, plugin will generate descriptions and example values for properties based on comments

Make sure to delete the `/dist` folder and rebuild your application whenever plugin options are updated. If you don't use the CLI but instead have a custom `webpack` configuration, you can use this plugin in combination with `ts-loader`:

```
getCustomTransformers: (program: any) => ({
  before: [require('@nestjs/swagger/plugin').before({}, program)]
}),
```

## SWC builder

For standard setups (non-monorepo), to use CLI Plugins with the SWC builder, you need to enable type checking, as described [here](#).

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

For monorepo setups, follow the instructions [here](#).

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

Now, the serialized metadata file must be loaded by the `SwaggerModule#loadPluginMetadata` method, as shown below:

```
import metadata from './metadata'; // <-- file auto-generated by the
"PluginMetadataGenerator"

await SwaggerModule.loadPluginMetadata(metadata); // <-- here
const document = SwaggerModule.createDocument(app, config);
```

## Integration with `ts-jest` (e2e tests)

To run e2e tests, `ts-jest` compiles your source code files on the fly, in memory. This means, it doesn't use Nest CLI compiler and does not apply any plugins or perform AST transformations.

To enable the plugin, create the following file in your e2e tests directory:

```
const transformer = require('@nestjs/swagger/plugin');

module.exports.name = 'nestjs-swagger-transformer';
// you should change the version number anytime you change the
configuration below – otherwise, jest will not detect changes
module.exports.version = 1;

module.exports.factory = (cs) => {
  return transformer.before(
    {
      // @nestjs/swagger/plugin options (can be empty)
```

```

    },
    cs.program, // "cs.tsCompiler.program" for older versions of Jest (<=
v27)
);
}
;
```

With this in place, import AST transformer within your `jest` configuration file. By default (in the starter application), e2e tests configuration file is located under the `test` folder and is named `jest-e2e.json`.

```
{
  ... // other configuration
  "globals": {
    "ts-jest": {
      "astTransformers": {
        "before": ["<path to the file created above>"]
      }
    }
  }
}
```

If you use `jest@^29`, then use the snippet below, as the previous approach got deprecated.

```
{
  ... // other configuration
  "transform": {
    "^.+\\.(t|j)s$": [
      "ts-jest",
      {
        "astTransformers": {
          "before": ["<path to the file created above>"]
        }
      }
    ]
  }
}
```

## Troubleshooting `jest` (e2e tests)

In case `jest` does not seem to pick up your configuration changes, it's possible that Jest has already **cached** the build result. To apply the new configuration, you need to clear Jest's cache directory.

To clear the cache directory, run the following command in your NestJS project folder:

```
$ npx jest --clearCache
```

In case the automatic cache clearance fails, you can still manually remove the cache folder with the following commands:

```
# Find jest cache directory (usually /tmp/jest_rs)
# by running the following command in your NestJS project root
$ npx jest --showConfig | grep cache
# ex result:
#   "cache": true,
#   "cacheDirectory": "/tmp/jest_rs"

# Remove or empty the Jest cache directory
$ rm -rf <cacheDirectory value>
# ex:
# rm -rf /tmp/jest_rs
```

## Other features

This page lists all the other available features that you may find useful.

### Global prefix

To ignore a global prefix for routes set through `setGlobalPrefix()`, use `ignoreGlobalPrefix`:

```
const document = SwaggerModule.createDocument(app, options, {  
    ignoreGlobalPrefix: true,  
});
```

### Global parameters

You can add parameter definitions to all routes using `DocumentBuilder`:

```
const options = new DocumentBuilder().addGlobalParameters({  
    name: 'tenantId',  
    in: 'header',  
});
```

### Multiple specifications

The `SwaggerModule` provides a way to support multiple specifications. In other words, you can serve different documentation, with different UIs, on different endpoints.

To support multiple specifications, your application must be written with a modular approach. The `createDocument()` method takes a 3rd argument, `extraOptions`, which is an object with a property named `include`. The `include` property takes a value which is an array of modules.

You can setup multiple specifications support as shown below:

```
import { NestFactory } from '@nestjs/core';  
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';  
import { AppModule } from './app.module';  
import { CatsModule } from './cats/cats.module';  
import { DogsModule } from './dogs/dogs.module';  
  
async function bootstrap() {  
    const app = await NestFactory.create(AppModule);  
  
    /**  
     * createDocument(application, configurationOptions, extraOptions);  
     *  
     * createDocument method takes an optional 3rd argument "extraOptions"  
     * which is an object with "include" property where you can pass an
```

```
Array
  * of Modules that you want to include in that Swagger Specification
  * E.g: CatsModule and DogsModule will have two separate Swagger
Specifications which
  * will be exposed on two different SwaggerUI with two different
endpoints.
 */

const options = new DocumentBuilder()
  .setTitle('Cats example')
  .setDescription('The cats API description')
  .setVersion('1.0')
  .addTag('cats')
  .build();

const catDocument = SwaggerModule.createDocument(app, options, {
  include: [CatsModule],
});
SwaggerModule.setup('api/cats', app, catDocument);

const secondOptions = new DocumentBuilder()
  .setTitle('Dogs example')
  .setDescription('The dogs API description')
  .setVersion('1.0')
  .addTag('dogs')
  .build();

const dogDocument = SwaggerModule.createDocument(app, secondOptions, {
  include: [DogsModule],
});
SwaggerModule.setup('api/dogs', app, dogDocument);

await app.listen(3000);
}
bootstrap();
```

Now you can start your server with the following command:

```
$ npm run start
```

Navigate to <http://localhost:3000/api/cats> to see the Swagger UI for cats:



In turn, <http://localhost:3000/api/dogs> will expose the Swagger UI for dogs:



## Migration guide

If you're currently using `@nestjs/swagger@3.*`, note the following breaking/API changes in version 4.0.

### Breaking changes

The following decorators have been changed/rename:

- `@ApiModelProperty` is now `@ApiProperty`
- `@ApiModelPropertyOptional` is now `@ApiPropertyOptional`
- `@ApiResponseModelProperty` is now `@ApiResponseProperty`
- `@ApiImplicitQuery` is now `@ApiQuery`
- `@ApiImplicitParam` is now `@ApiParam`
- `@ApiImplicitBody` is now `@ApiBody`
- `@ApiImplicitHeader` is now `@ApiHeader`
- `@ApiOperation({{ '{' }} title: 'test' {{ '}' }})` is now `@ApiOperation({{ '{' }} summary: 'test' {{ '}' }})`
- `@ApiUseTags` is now `@ApiTags`

`DocumentBuilder` breaking changes (updated method signatures):

- `addTag`
- `addBearerAuth`
- `addOAuth2`
- `setContactEmail` is now `setContact`
- `setHost` has been removed
- `setSchemes` has been removed (use the `addServer` instead, e.g., `addServer('http://')`)

### New methods

The following methods have been added:

- `addServer`
- `addApiKey`
- `addBasicAuth`
- `addSecurity`
- `addSecurityRequirements`

## 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 \  string) =&gt; void</code>
<code>get</code> or <code>\$</code>	Retrieves an instance of either injectable or controller, otherwise, throws exception.	<code>get(token: InjectionToken) =&gt; any</code>
<code>methods</code>	Display all public methods available on a given provider or controller.	<code>methods(token: ClassRef \  string) =&gt; void</code>
<code>resolve</code>	Resolves transient or request-scoped instance of either injectable or controller, otherwise, throws exception.	<code>resolve(token: InjectionToken, contextId: any) =&gt; Promise&lt;any&gt;</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 \  ClassRef) =&gt; INestApplicationContext</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() {
  const replServer = await repl(AppModule);
  replServer.setupHistory(".nestjs_repl_history", (err) => {
    if (err) {
      console.error(err);
    }
  });
}
```

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.usersService.create(createUserDto);
    }

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

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

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

    @Delete(':id')
    remove(@Param('id') id: string) {
        return this.usersService.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 `UsersService`, so let's generate that module and service now:

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

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

```
@@filename(users/users.service)  
import { Injectable } from '@nestjs/common';  
  
// This should be a real class/interface representing a user entity  
export type User = any;
```

```

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

async findOne(username: string): Promise<User | undefined> {
  return this.users.find(user => user.username === username);
}

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

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

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

```

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

```

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

```

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

@switch
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';

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

Our `AuthService` has the job of retrieving a user and verifying the password. We create a `validateUser()` method for this purpose. In the code below, we use a convenient ES6 spread operator to strip the `password` property from the `user` object before returning it. We'll be calling into the `validateUser()` method from our Passport local strategy in a moment.

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

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

  async validateUser(username: string, pass: string): Promise<any> {
    const user = await this.usersService.findOne(username);
    if (user && user.password === pass) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }
}

@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { UsersService } from '../users/users.service';

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

  async validateUser(username, pass) {
    const user = await this.usersService.findOne(username);
```

```

    if (user && user.password === pass) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }
}

```

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

Now, we update our [AuthModule](#) to import the [UsersModule](#).

```

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

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

@@switch
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UsersModule } from '../users/users.module';

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

```

## Implementing Passport local

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

```

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

```

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

  async validate(username: string, password: string): Promise<any> {
    const user = await this.authService.validateUser(username, password);
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
}

@switch
import { Strategy } from 'passport-local';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException, Dependencies } from
  '@nestjs/common';
import { AuthService } from './auth.service';

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

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

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

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

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

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

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

We need to configure our `AuthModule` to use the Passport features we just defined. Update `auth.module.ts` to look like this:

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

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

@@switch
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UsersModule } from '../users/users.module';
import { PassportModule } from '@nestjs/passport';
import { LocalStrategy } from './local.strategy';

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

## Built-in Passport Guards

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

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

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

- Restrict the routes an unauthenticated user can access (i.e., deny access to restricted routes). We'll use Guards in their familiar capacity to handle this function, by placing a Guard on the protected routes. As you may anticipate, we'll be checking for the presence of a valid JWT in this Guard, so we'll work on this Guard later, once we are successfully issuing JWTs.
- Initiate the **authentication step** itself when a previously unauthenticated user attempts to login. This is the step where we'll **issue** a JWT to a valid user. Thinking about this for a moment, we know we'll need to **POST** username/password credentials to initiate authentication, so we'll set up a **POST /auth/login** route to handle that. This raises the question: how exactly do we invoke the passport-local strategy in that route?

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

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

## Login route

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

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

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

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

@Controller()
export class AppController {
  @UseGuards(AuthGuard('local'))
```

```

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

```

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

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

Since these are API routes, we'll test them using the commonly available `cURL` library. You can test with any of the `user` objects hard-coded in the `UsersService`.

```

$ # POST to /auth/login
$ curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
$ # result -> {"userId":1,"username":"john"}

```

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

```

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

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

```

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

```

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

```

## JWT functionality

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

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

We'll need to install a couple more packages to support our JWT requirements:

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

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

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

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

With this in mind, we can now finally generate a real JWT, and return it in this route. To keep our services cleanly modularized, we'll handle generating the JWT in the `authService`. Open the `auth.service.ts` file in the `auth` folder, and add the `login()` method, and import the `JwtService` as shown:

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

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

  async validateUser(username: string, pass: string): Promise<any> {
    const user = await this.usersService.findOne(username);
    if (user && user.password === pass) {
      const { password, ...result } = user;
      return result;
    }
    return null;
  }

  async login(user: any): Promise<{ accessToken: string }> {
    const payload = { ...user, id: user._id };
    return {
      accessToken: this.jwtService.sign(payload),
    };
  }
}
```

```

}

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** **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 `UserService`.

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

## Implementing Passport JWT

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

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

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

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

@@switch
import { ExtractJwt, Strategy } from 'passport-jwt';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable } from '@nestjs/common';
import { jwtConstants } from './constants';

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

```

    });
}

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

```

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

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

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

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

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

Add the new [JwtStrategy](#) as a provider in the [AuthModule](#):

```

@@filename(auth/auth.module)
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { LocalStrategy } from './local.strategy';

```

```

import { JwtStrategy } from './jwt.strategy';
import { UsersModule } from '../users/users.module';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';
import { jwtConstants } from './constants';

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

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

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

```

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

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

```

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

```

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

## Implement protected route and JWT strategy guards

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

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

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

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

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

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

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

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

  @UseGuards(LocalAuthGuard)
  @Post('auth/login')
  @Bind(Request())
  async login(req) {
```

```

        return this.authService.login(req.user);
    }

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

```

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

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

```

$ # GET /profile
$ curl http://localhost:3000/profile
$ # result -> {"statusCode":401,"message":"Unauthorized"}

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

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

```

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

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

## Extending guards

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

the built-in class and override methods within a sub-class.

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

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

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

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

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

## Enable authentication globally

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

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

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

```
},  
],
```

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

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

```
import { SetMetadata } from '@nestjs/common';  
  
export const IS_PUBLIC_KEY = 'isPublic';  
export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

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

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

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

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

```
@Injectable()  
export class JwtAuthGuard extends AuthGuard('jwt') {  
  constructor(private reflector: Reflector) {  
    super();  
  }  
  
  canActivate(context: ExecutionContext) {  
    const isPublic = this.reflector.getAllAndOverride<boolean>(IS_PUBLIC_KEY, [  
      context.getHandler(),  
      context.getClass(),  
    ]);  
    if (isPublic) {  
      return true;  
    }  
    return super.canActivate(context);  
  }  
}
```

## Request-scoped strategies

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

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

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

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

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

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

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

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

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

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

## Customize Passport

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

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

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

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

Take a look at the official [Passport Website](#) for property names.

## Named strategies

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

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

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

## GraphQL

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

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

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

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

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

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

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

## 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.IgnorePlugin({  
        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
```

**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.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** **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** `@Use RequestContext()` 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 `@Use RequestContext()` 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) {}

    @Use RequestContext()
    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
    })
  ],
  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** **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'),
  },
];
@@switch
import * as mongoose from 'mongoose';

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

**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.

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

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

**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 [@nestjs/terminus](#) we need to install the required dependency.

```
$ npm install --save @nestjs/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, [@nestjs/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 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.

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

#### 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 [@nestjs/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 '@nestjs/common';
import { HealthIndicator, HealthIndicatorResult, HealthCheckError } from
  '@nestjs/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** **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
<code>json</code> (default)	Prints a summary of the health check result in case of an error as JSON object	
<code>pretty</code>	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: string, dragonId: string) {
    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)
@EventsHandler(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
@Injectable()
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
        xceptionInfo) => {
        // 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** **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** **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(CLSE_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 **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** **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.

## Serverless

Serverless computing is a cloud computing execution model in which the cloud provider allocates machine resources on-demand, taking care of the servers on behalf of their customers. When an app is not in use, there are no computing resources allocated to the app. Pricing is based on the actual amount of resources consumed by an application ([source](#)).

With a **serverless architecture**, you focus purely on the individual functions in your application code. Services such as AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions take care of all the physical hardware, virtual machine operating system, and web server software management.

**info Hint** This chapter does not cover the pros and cons of serverless functions nor dives into the specifics of any cloud providers.

### Cold start

A cold start is the first time your code has been executed in a while. Depending on a cloud provider you use, it may span several different operations, from downloading the code and bootstrapping the runtime to eventually running your code. This process adds **significant latency** depending on several factors, the language, the number of packages your application require, etc.

The cold start is important and although there are things which are beyond our control, there's still a lot of things we can do on our side to make it as short as possible.

While you can think of Nest as a fully-fledged framework designed to be used in complex, enterprise applications, it is also **suitable for much "simpler" applications** (or scripts). For example, with the use of [Standalone applications](#) feature, you can take advantage of Nest's DI system in simple workers, CRON jobs, CLIs, or serverless functions.

### Benchmarks

To better understand what's the cost of using Nest or other, well-known libraries (like [express](#)) in the context of serverless functions, let's compare how much time Node runtime needs to run the following scripts:

```
// #1 Express
import * as express from 'express';

async function bootstrap() {
  const app = express();
  app.get('/', (req, res) => res.send('Hello world!'));
  await new Promise<void>((resolve) => app.listen(3000, resolve));
}
bootstrap();

// #2 Nest (with @nestjs/platform-express)
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
```

```

async function bootstrap() {
  const app = await NestFactory.create(AppModule, { logger: ['error'] });
  await app.listen(3000);
}

bootstrap();

// #3 Nest as a Standalone application (no HTTP server)
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { AppService } from './app.service';

async function bootstrap() {
  const app = await NestFactory.createApplicationContext(AppModule, {
    logger: ['error'],
  });
  console.log(app.get(AppService).getHello());
}
bootstrap();

// #4 Raw Node.js script
async function bootstrap() {
  console.log('Hello world!');
}
bootstrap();

```

For all these scripts, we used the `tsc` (TypeScript) compiler and so the code remains unbundled (`webpack` isn't used).

Express	0.0079s (7.9ms)
Nest with <code>@nestjs/platform-express</code>	0.1974s (197.4ms)
Nest (standalone application)	0.1117s (111.7ms)
Raw Node.js script	0.0071s (7.1ms)

**info Note** Machine: MacBook Pro Mid 2014, 2.5 GHz Quad-Core Intel Core i7, 16 GB 1600 MHz DDR3, SSD.

Now, let's repeat all benchmarks but this time, using `webpack` (if you have `Nest CLI` installed, you can run `nest build --webpack`) to bundle our application into a single executable JavaScript file. However, instead of using the default `webpack` configuration that Nest CLI ships with, we'll make sure to bundle all dependencies (`node_modules`) together, as follows:

```

module.exports = (options, webpack) => {
  const lazyImports = [
    '@nestjs/microservices/microservices-module',
    '@nestjs/websockets/socket-module',
  ];

  return {

```

```

    ...options,
    externals: [],
    plugins: [
      ...options.plugins,
      new webpack.IgnorePlugin({
        checkResource(resource) {
          if (lazyImports.includes(resource)) {
            try {
              require.resolve(resource);
            } catch (err) {
              return true;
            }
          }
          return false;
        },
      }),
    ],
  };
}

```

**info Hint** To instruct Nest CLI to use this configuration, create a new `webpack.config.js` file in the root directory of your project.

With this configuration, we received the following results:

Express	0.0068s (6.8ms)
Nest with <code>@nestjs/platform-express</code>	0.0815s (81.5ms)
Nest (standalone application)	0.0319s (31.9ms)
Raw Node.js script	0.0066s (6.6ms)

**info Note** Machine: MacBook Pro Mid 2014, 2.5 GHz Quad-Core Intel Core i7, 16 GB 1600 MHz DDR3, SSD.

**info Hint** You could optimize it even further by applying additional code minification & optimization techniques (using `webpack` plugins, etc.).

As you can see, the way you compile (and whether you bundle your code) is crucial and has a significant impact on the overall startup time. With `webpack`, you can get the bootstrap time of a standalone Nest application (starter project with one module, controller, and service) down to ~32ms on average, and down to ~81.5ms for a regular HTTP, express-based NestJS app.

For more complicated Nest applications, for example, with 10 resources (generated through `$ nest g resource` schematic = 10 modules, 10 controllers, 10 services, 20 DTO classes, 50 HTTP endpoints + `AppModule`), the overall startup on MacBook Pro Mid 2014, 2.5 GHz Quad-Core Intel Core i7, 16 GB 1600 MHz DDR3, SSD is approximately 0.1298s (129.8ms). Running a monolithic application as a serverless function typically doesn't make too much sense anyway, so think of this benchmark more as an example of how the bootstrap time may potentially increase as your application grows.

## Runtime optimizations

Thus far we covered compile-time optimizations. These are unrelated to the way you define providers and load Nest modules in your application, and that plays an essential role as your application gets bigger.

For example, imagine having a database connection defined as an [asynchronous provider](#). Async providers are designed to delay the application start until one or more asynchronous tasks are completed. That means, if your serverless function on average requires 2s to connect to the database (on bootstrap), your endpoint will need at least two extra seconds (because it must wait till the connection is established) to send a response back (when it's a cold start and your application wasn't running already).

As you can see, the way you structure your providers is somewhat different in a [serverless environment](#) where bootstrap time is important. Another good example is if you use Redis for caching, but only in certain scenarios. Perhaps, in this case, you should not define a Redis connection as an async provider, as it would slow down the bootstrap time, even if it's not required for this specific function invocation.

Also, sometimes you could lazy-load entire modules, using the [LazyModuleLoader](#) class, as described in [this chapter](#). Caching is a great example here too. Imagine that your application has, let's say, [CacheModule](#) which internally connects to Redis and also, exports the [CacheService](#) to interact with the Redis storage. If you don't need it for all potential function invocations, you can just load it on-demand, lazily. This way you'll get a faster startup time (when a cold start occurs) for all invocations that don't require caching.

```
if (request.method === RequestMethod[RequestMethod.GET]) {
  const { CacheModule } = await import('./cache.module');
  const moduleRef = await this.lazyModuleLoader.load(() => CacheModule);

  const { CacheService } = await import('./cache.service');
  const cacheService = moduleRef.get(CacheService);

  return cacheService.get(ENDPOINT_KEY);
}
```

Another great example is a webhook or worker, which depending on some specific conditions (e.g., input arguments), may perform different operations. In such a case, you could specify a condition inside your route handler that lazily loads an appropriate module for the specific function invocation, and just load every other module lazily.

```
if (workerType === WorkerType.A) {
  const { WorkerAModule } = await import('./worker-a.module');
  const moduleRef = await this.lazyModuleLoader.load(() => WorkerAModule);
  // ...
} else if (workerType === WorkerType.B) {
  const { WorkerBModule } = await import('./worker-b.module');
  const moduleRef = await this.lazyModuleLoader.load(() => WorkerBModule);
  // ...
}
```

## Example integration

The way your application's entry file (typically `main.ts` file) is supposed to look like **depends on several factors** and so **there's no single template** that just works for every scenario. For example, the initialization file required to spin up your serverless function varies by cloud providers (AWS, Azure, GCP, etc.). Also, depending on whether you want to run a typical HTTP application with multiple routes/endpoints or just provide a single route (or execute a specific portion of code), your application's code will look different (for example, for the endpoint-per-function approach you could use the `NestFactory.createApplicationContext` instead of booting the HTTP server, setting up middleware, etc.).

Just for illustration purposes, we'll integrate Nest (using `@nestjs/platform-express` and so spinning up the whole, fully functional HTTP router) with the `Serverless` framework (in this case, targeting AWS Lambda). As we've mentioned earlier, your code will differ depending on the cloud provider you choose, and many other factors.

First, let's install the required packages:

```
$ npm i @vendia/serverless-express aws-lambda
$ npm i -D @types/aws-lambda serverless-offline
```

**info Hint** To speed up development cycles, we install the `serverless-offline` plugin which emulates AWS λ and API Gateway.

Once the installation process is complete, let's create the `serverless.yml` file to configure the Serverless framework:

```
service: serverless-example

plugins:
  - serverless-offline

provider:
  name: aws
  runtime: nodejs14.x

functions:
  main:
    handler: dist/main.handler
    events:
      - http:
          method: ANY
          path: /
      - http:
          method: ANY
          path: '{proxy+}'
```

**info Hint** To learn more about the Serverless framework, visit the [official documentation](#).

With this place, we can now navigate to the `main.ts` file and update our bootstrap code with the required boilerplate:

```
import { NestFactory } from '@nestjs/core';
import serverlessExpress from '@vendia/serverless-express';
import { Callback, Context, Handler } from 'aws-lambda';
import { AppModule } from './app.module';

let server: Handler;

async function bootstrap(): Promise<Handler> {
  const app = await NestFactory.create(AppModule);
  await app.init();

  const expressApp = app.getHttpAdapter().getInstance();
  return serverlessExpress({ app: expressApp });
}

export const handler: Handler = async (
  event: any,
  context: Context,
  callback: Callback,
) => {
  server = server ?? (await bootstrap());
  return server(event, context, callback);
};
```

**info Hint** For creating multiple serverless functions and sharing common modules between them, we recommend using the [CLI Monorepo mode](#).

**warning Warning** If you use `@nestjs/swagger` package, there are a few additional steps required to make it work properly in the context of serverless function. Check out this [thread](#) for more information.

Next, open up the `tsconfig.json` file and make sure to enable the `esModuleInterop` option to make the `@vendia/serverless-express` package load properly.

```
{
  "compilerOptions": {
    ...
    "esModuleInterop": true
  }
}
```

Now we can build our application (with `nest build` or `tsc`) and use the `serverless` CLI to start our lambda function locally:

```
$ npm run build
$ npx serverless offline
```

Once the application is running, open your browser and navigate to [http://localhost:3000/dev/\[ANY\\_ROUTE\]](http://localhost:3000/dev/[ANY_ROUTE]) (where [ANY\_ROUTE] is any endpoint registered in your application).

In the sections above, we've shown that using `webpack` and bundling your app can have significant impact on the overall bootstrap time. However, to make it work with our example, there are a few additional configurations you must add in your `webpack.config.js` file. Generally, to make sure our `handler` function will be picked up, we must change the `output.libraryTarget` property to `commonjs2`.

```
return {
  ...options,
  externals: [],
  output: {
    ...options.output,
    libraryTarget: 'commonjs2',
  },
  // ... the rest of the configuration
};
```

With this in place, you can now use `$ nest build --webpack` to compile your function's code (and then `$ npx serverless offline` to test it).

It's also recommended (but **not required** as it will slow down your build process) to install the `terser-webpack-plugin` package and override its configuration to keep classnames intact when minifying your production build. Not doing so can result in incorrect behavior when using `class-validator` within your application.

```
const TerserPlugin = require('terser-webpack-plugin');

return {
  ...options,
  externals: [],
  optimization: {
    minimizer: [
      new TerserPlugin({
        terserOptions: {
          keep_classnames: true,
        },
      }),
    ],
  },
  output: {
    ...options.output,
    libraryTarget: 'commonjs2',
  }
};
```

```
},
// ... the rest of the configuration
};
```

## Using standalone application feature

Alternatively, if you want to keep your function very lightweight and you don't need any HTTP-related features (routing, but also guards, interceptors, pipes, etc.), you can just use

`NestFactory.createApplicationContext` (as mentioned earlier) instead of running the entire HTTP server (and `express` under the hood), as follows:

```
@@filename(main)
import { HttpStatus } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { Callback, Context, Handler } from 'aws-lambda';
import { AppModule } from './app.module';
import { AppService } from './app.service';

export const handler: Handler = async (
  event: any,
  context: Context,
  callback: Callback,
) => {
  const appContext = await
NestFactory.createApplicationContext(AppModule);
  const appService = appContext.get(AppService);

  return {
    body: appService.getHello(),
    statusCode: HttpStatus.OK,
  };
};
```

**info Hint** Be aware that `NestFactory.createApplicationContext` does not wrap controller methods with enhancers (guard, interceptors, etc.). For this, you must use the `NestFactory.create` method.

You could also pass the `event` object down to, let's say, `EventsService` provider that could process it and return a corresponding value (depending on the input value and your business logic).

```
export const handler: Handler = async (
  event: any,
  context: Context,
  callback: Callback,
) => {
  const appContext = await
NestFactory.createApplicationContext(AppModule);
  const eventsService = appContext.get(EventsService);
```

```
return eventsService.process(event);  
};
```

## HTTP adapter

Occasionally, you may want to access the underlying HTTP server, either within the Nest application context or from the outside.

Every native (platform-specific) HTTP server/library (e.g., Express and Fastify) instance is wrapped in an **adapter**. The adapter is registered as a globally available provider that can be retrieved from the application context, as well as injected into other providers.

### Outside application context strategy

To get a reference to the **HttpAdapter** from outside of the application context, call the **getHttpAdapter()** method.

```
@@filename()
const app = await NestFactory.create(AppModule);
const httpAdapter = app.getHttpAdapter();
```

### In-context strategy

To get a reference to the **HttpAdapterHost** from within the application context, inject it using the same technique as any other existing provider (e.g., using constructor injection).

```
@@filename()
export class CatsService {
  constructor(private adapterHost: HttpAdapterHost) {}
}
@@switch
@Dependencies(HttpAdapterHost)
export class CatsService {
  constructor(adapterHost) {
    this.adapterHost = adapterHost;
  }
}
```

**info Hint** The **HttpAdapterHost** is imported from the **@nestjs/core** package.

The **HttpAdapterHost** is **not** an actual **HttpAdapter**. To get the actual **HttpAdapter** instance, simply access the **httpAdapter** property.

```
const adapterHost = app.get(HttpAdapterHost);
const httpAdapter = adapterHost.httpAdapter;
```

The **httpAdapter** is the actual instance of the HTTP adapter used by the underlying framework. It is an instance of either **ExpressAdapter** or **FastifyAdapter** (both classes extend **AbstractHttpAdapter**).

The adapter object exposes several useful methods to interact with the HTTP server. However, if you want to access the library instance (e.g., the Express instance) directly, call the `getInstance()` method.

```
const instance = httpAdapter.getInstance();
```

## Global prefix

To set a prefix for **every route** registered in an HTTP application, use the `setGlobalPrefix()` method of the `INestApplication` instance.

```
const app = await NestFactory.create(AppModule);
app.setGlobalPrefix('v1');
```

You can exclude routes from the global prefix using the following construction:

```
app.setGlobalPrefix('v1', {
  exclude: [{ path: 'health', method: RequestMethod.GET }],
});
```

Alternatively, you can specify route as a string (it will apply to every request method):

```
app.setGlobalPrefix('v1', { exclude: ['cats'] });
```

**info Hint** The `path` property supports wildcard parameters using the [path-to-regexp package](#). Note: this does not accept wildcard asterisks `*`. Instead, you must use parameters (e.g., `(.*)`, `:splat*`).

## Raw body

One of the most common use-case for having access to the raw request body is performing webhook signature verifications. Usually to perform webhook signature validations the unserialized request body is required to calculate an HMAC hash.

**warning** **Warning** This feature can be used only if the built-in global body parser middleware is enabled, ie., you must not pass `bodyParser: false` when creating the app.

### Use with Express

First enable the option when creating your Nest Express application:

```
import { NestFactory } from '@nestjs/core';
import type { NestExpressApplication } from '@nestjs/platform-express';
import { AppModule } from './app.module';

// in the "bootstrap" function
const app = await NestFactory.create<NestExpressApplication>(AppModule, {
  rawBody: true,
});
await app.listen(3000);
```

To access the raw request body in a controller, a convenience interface `RawBodyRequest` is provided to expose a `rawBody` field on the request: use the interface `RawBodyRequest` type:

```
import { Controller, Post, RawBodyRequest, Req } from '@nestjs/common';
import { Request } from 'express';

@Controller('cats')
class CatsController {
  @Post()
  create(@Req() req: RawBodyRequest<Request>) {
    const raw = req.rawBody; // returns a `Buffer`.
  }
}
```

### Registering a different parser

By default, only `json` and `urlencoded` parsers are registered. If you want to register a different parser on the fly, you will need to do so explicitly.

For example, to register a `text` parser, you can use the following code:

```
app.useBodyParser('text');
```

warning **Warning** Ensure that you are providing the correct application type to the `NestFactory.create` call. For Express applications, the correct type is `NestExpressApplication`. Otherwise the `.useBodyParser` method will not be found.

## Body parser size limit

If your application needs to parse a body larger than the default `100kb` of Express, use the following:

```
app.useBodyParser('json', { limit: '10mb' });
```

The `.useBodyParser` method will respect the `rawBody` option that is passed in the application options.

## Use with Fastify

First enable the option when creating your Nest Fastify application:

```
import { NestFactory } from '@nestjs/core';
import {
  FastifyAdapter,
  NestFastifyApplication,
} from '@nestjs/platform-fastify';
import { AppModule } from './app.module';

// in the "bootstrap" function
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  new FastifyAdapter(),
  {
    rawBody: true,
  },
);
await app.listen(3000);
```

To access the raw request body in a controller, a convenience interface `RawBodyRequest` is provided to expose a `rawBody` field on the request: use the interface `RawBodyRequest` type:

```
import { Controller, Post, RawBodyRequest, Req } from '@nestjs/common';
import { FastifyRequest } from 'fastify';

@Controller('cats')
class CatsController {
  @Post()
  create(@Req() req: RawBodyRequest<FastifyRequest>) {
    const raw = req.rawBody; // returns a `Buffer`.
  }
}
```

## Registering a different parser

By default, only `application/json` and `application/x-www-form-urlencoded` parsers are registered. If you want to register a different parser on the fly, you will need to do so explicitly.

For example, to register a `text/plain` parser, you can use the following code:

```
app.useBodyParser('text/plain');
```

**warning** **Warning** Ensure that you are providing the correct application type to the `NestFactory.create` call. For Fastify applications, the correct type is `NestFastifyApplication`. Otherwise the `.useBodyParser` method will not be found.

## Body parser size limit

If your application needs to parse a body larger than the default 1MiB of Fastify, use the following:

```
const bodyLimit = 10_485_760; // 10MiB
app.useBodyParser('application/json', { bodyLimit });
```

The `.useBodyParser` method will respect the `rawBody` option that is passed in the application options.

## Hybrid application

A hybrid application is one that both listens for HTTP requests, as well as makes use of connected microservices. The `INestApplication` instance can be connected with `INestMicroservice` instances through the `connectMicroservice()` method.

```
const app = await NestFactory.create(AppModule);
const microservice = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.TCP,
});

await app.startAllMicroservices();
await app.listen(3001);
```

To connect multiple microservice instances, issue the call to `connectMicroservice()` for each microservice:

```
const app = await NestFactory.create(AppModule);
// microservice #1
const microserviceTcp = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.TCP,
  options: {
    port: 3001,
  },
});
// microservice #2
const microserviceRedis = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.REDIS,
  options: {
    host: 'localhost',
    port: 6379,
  },
});

await app.startAllMicroservices();
await app.listen(3001);
```

To bind `@MessagePattern()` to only one transport strategy (for example, MQTT) in a hybrid application with multiple microservices, we can pass the second argument of type `Transport` which is an enum with all the built-in transport strategies defined.

```
@@filename()
@MessagePattern('time.us.*', Transport.NATS)
getDate(@Payload() data: number[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
```

```
@MessagePattern({ cmd: 'time.us' }, Transport.TCP)
getTCPDate(@Payload() data: number[]) {
  return new Date().toLocaleTimeString(...);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('time.us.*', Transport.NATS)
getDate(data, context) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
@Bind(Payload(), Ctx())
@MessagePattern({ cmd: 'time.us' }, Transport.TCP)
getTCPDate(data, context) {
  return new Date().toLocaleTimeString(...);
}
```

info Hint `@Payload()`, `@Ctx()`, `Transport` and `NatsContext` are imported from `@nestjs/microservices`.

## Sharing configuration

By default a hybrid application will not inherit global pipes, interceptors, guards and filters configured for the main (HTTP-based) application. To inherit these configuration properties from the main application, set the `inheritAppConfig` property in the second argument (an optional options object) of the `connectMicroservice()` call, as follow:

```
const microservice = app.connectMicroservice<MicroserviceOptions>(
{
  transport: Transport.TCP,
},
{ inheritAppConfig: true },
);
```

## HTTPS

To create an application that uses the HTTPS protocol, set the `httpsOptions` property in the options object passed to the `create()` method of the `NestFactory` class:

```
const httpsOptions = {
  key: fs.readFileSync('./secrets/private-key.pem'),
  cert: fs.readFileSync('./secrets/public-certificate.pem'),
};

const app = await NestFactory.create(AppModule, {
  httpsOptions,
});

await app.listen(3000);
```

If you use the `FastifyAdapter`, create the application as follows:

```
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  new FastifyAdapter({ https: httpsOptions }),
);
```

## Multiple simultaneous servers

The following recipe shows how to instantiate a Nest application that listens on multiple ports (for example, on a non-HTTPS port and an HTTPS port) simultaneously.

```
const httpsOptions = {
  key: fs.readFileSync('./secrets/private-key.pem'),
  cert: fs.readFileSync('./secrets/public-certificate.pem'),
};

const server = express();
const app = await NestFactory.create(
  AppModule,
  new ExpressAdapter(server),
);
await app.init();

const httpServer = http.createServer(server).listen(3000);
const httpsServer = https.createServer(httpsOptions, server).listen(443);
```

Because we called `http.createServer`/`https.createServer` ourselves, NestJS doesn't close them when calling `app.close` on termination signal. We need to do this ourselves:

```
@Injectable()
export class ShutdownObserver implements OnApplicationShutdown {
  private httpServers: http.Server[] = [];

  public addHttpServer(server: http.Server): void {
    this.httpServers.push(server);
  }

  public async onApplicationShutdown(): Promise<void> {
    await Promise.all(
      this.httpServers.map((server) =>
        new Promise((resolve, reject) => {
          server.close((error) => {
            if (error) {
              reject(error);
            } else {
              resolve(null);
            }
          });
        })
      ),
    );
  }
}

const shutdownObserver = app.get(ShutdownObserver);
shutdownObserver.addHttpServer(httpServer);
shutdownObserver.addHttpServer(httpsServer);
```

**info Hint** The `ExpressAdapter` is imported from the `@nestjs/platform-express` package. The `http` and `https` packages are native Node.js packages.

**Warning** This recipe does not work with [GraphQL Subscriptions](#).

## Request lifecycle

Nest applications handle requests and produce responses in a sequence we refer to as the **request lifecycle**. With the use of middleware, pipes, guards, and interceptors, it can be challenging to track down where a particular piece of code executes during the request lifecycle, especially as global, controller level, and route level components come into play. In general, a request flows through middleware to guards, then to interceptors, then to pipes and finally back to interceptors on the return path (as the response is generated).

## Middleware

Middleware is executed in a particular sequence. First, Nest runs globally bound middleware (such as middleware bound with `app.use`) and then it runs **module bound middleware**, which are determined on paths. Middleware are run sequentially in the order they are bound, similar to the way middleware in Express works. In the case of middleware bound across different modules, the middleware bound to the root module will run first, and then middleware will run in the order that the modules are added to the imports array.

## Guards

Guard execution starts with global guards, then proceeds to controller guards, and finally to route guards. As with middleware, guards run in the order in which they are bound. For example:

```
@UseGuards(Guard1, Guard2)
@Controller('cats')
export class CatsController {
  constructor(private catsService: CatsService) {}

  @UseGuards(Guard3)
  @Get()
  getCats(): Cats[] {
    return this.catsService.getCats();
  }
}
```

Guard1 will execute before Guard2 and both will execute before Guard3.

**info Hint** When speaking about globally bound vs controller or locally bound, the difference is where the guard (or other component is bound). If you are using `app.useGlobalGuard()` or providing the component via a module, it is globally bound. Otherwise, it is bound to a controller if the decorator precedes a controller class, or to a route if the decorator proceeds a route declaration.

## Interceptors

Interceptors, for the most part, follow the same pattern as guards, with one catch: as interceptors return RxJS Observables, the observables will be resolved in a first in last out manner. So inbound requests will go through the standard global, controller, route level resolution, but the response side of the request (i.e., after returning from the controller method handler) will be resolved from route to controller to global. Also,

any errors thrown by pipes, controllers, or services can be read in the `catchError` operator of an interceptor.

## Pipes

Pipes follow the standard global to controller to route bound sequence, with the same first in first out in regards to the `@UsePipes()` parameters. However, at a route parameter level, if you have multiple pipes running, they will run in the order of the last parameter with a pipe to the first. This also applies to the route level and controller level pipes. For example, if we have the following controller:

```
@UsePipes(GeneralValidationPipe)
@Controller('cats')
export class CatsController {
    constructor(private catsService: CatsService) {}

    @UsePipes(RouteSpecificPipe)
    @Patch(':id')
    updateCat(
        @Body() body: UpdateCatDTO,
        @Param() params: UpdateCatParams,
        @Query() query: UpdateCatQuery,
    ) {
        return this.catsService.updateCat(body, params, query);
    }
}
```

then the `GeneralValidationPipe` will run for the `query`, then the `params`, and then the `body` objects before moving on to the `RouteSpecificPipe`, which follows the same order. If any parameter-specific pipes were in place, they would run (again, from the last to first parameter) after the controller and route level pipes.

## Filters

Filters are the only component that do not resolve global first. Instead, filters resolve from the lowest level possible, meaning execution starts with any route bound filters and proceeding next to controller level, and finally to global filters. Note that exceptions cannot be passed from filter to filter; if a route level filter catches the exception, a controller or global level filter cannot catch the same exception. The only way to achieve an effect like this is to use inheritance between the filters.

**info Hint** Filters are only executed if any uncaught exception occurs during the request process.

Caught exceptions, such as those caught with a `try/catch` will not trigger Exception Filters to fire.

As soon as an uncaught exception is encountered, the rest of the lifecycle is ignored and the request skips straight to the filter.

## Summary

In general, the request lifecycle looks like the following:

1. Incoming request

## 2. Middleware

- 2.1. Globally bound middleware
- 2.2. Module bound middleware

## 3. Guards

- 3.1 Global guards
- 3.2 Controller guards
- 3.3 Route guards

## 4. Interceptors (pre-controller)

- 4.1 Global interceptors
- 4.2 Controller interceptors
- 4.3 Route interceptors

## 5. Pipes

- 5.1 Global pipes
- 5.2 Controller pipes
- 5.3 Route pipes
- 5.4 Route parameter pipes

## 6. Controller (method handler)

## 7. Service (if exists)

## 8. Interceptors (post-request)

- 8.1 Route interceptor
- 8.2 Controller interceptor
- 8.3 Global interceptor

## 9. Exception filters

- 9.1 route
- 9.2 controller
- 9.3 global

## 10. Server response

## Common errors

During your development with NestJS, you may encounter various errors as you learn the framework.

### "Cannot resolve dependency" error

**info Hint** Check out the [NestJS Devtools](#) which can help you resolve the "Cannot resolve dependency" error effortlessly.

Probably the most common error message is about Nest not being able to resolve dependencies of a provider. The error message usually looks something like this:

```
Nest can't resolve dependencies of the <provider> (?). Please make sure  
that the argument <unknown_token> at index [<index>] is available in the  
<module> context.
```

Potential solutions:

- Is <module> a valid NestJS module?
- If <unknown\_token> is a provider, is it part of the current <module>?
- If <unknown\_token> is exported from a separate @Module, is that module imported within <module>?

```
@Module({  
  imports: [ /* the Module containing <unknown_token> */ ]  
})
```

The most common culprit of the error, is not having the `<provider>` in the module's `providers` array. Please make sure that the provider is indeed in the `providers` array and following [standard NestJS provider practices](#).

There are a few gotchas, that are common. One is putting a provider in an `imports` array. If this is the case, the error will have the provider's name where `<module>` should be.

If you run across this error while developing, take a look at the module mentioned in the error message and look at its `providers`. For each provider in the `providers` array, make sure the module has access to all of the dependencies. Often times, `providers` are duplicated in a "Feature Module" and a "Root Module" which means Nest will try to instantiate the provider twice. More than likely, the module containing the `<provider>` being duplicated should be added in the "Root Module"'s `imports` array instead.

If the `<unknown_token>` above is the string `dependency`, you might have a circular file import. This is different from the `circular dependency` below because instead of having providers depend on each other in their constructors, it just means that two files end up importing each other. A common case would be a module file declaring a token and importing a provider, and the provider import the token constant from the module file. If you are using barrel files, ensure that your barrel imports do not end up creating these circular imports as well.

If the `<unknown_token>` above is the string `Object`, it means that you're injecting using an type/interface without a proper provider's token. To fix that, make sure you're importing the class reference or use a custom token with `@Inject()` decorator. Read the [custom providers page](#).

Also, make sure you didn't end up injecting the provider on itself because self-injections are not allowed in NestJS. When this happens, `<unknown_token>` will likely be equal to `<provider>`.

If you are in a **monorepo setup**, you may face the same error as above but for core provider called `ModuleRef` as a `<unknown_token>`:

```
Nest can't resolve dependencies of the <provider> (?).
Please make sure that the argument ModuleRef at index [<index>] is
available in the <module> context.

...
```

This likely happens when your project ends up loading two Node modules of the package `@nestjs/core`, like this:

```
.
├── package.json
└── apps
    └── api
        └── node_modules
            └── @nestjs/bull
                └── node_modules
                    └── @nestjs/core
    └── node_modules
        ├── (other packages)
        └── @nestjs/core
```

Solutions:

- For **Yarn Workspaces**, use the [nohoist feature](#) to prevent hoisting the package `@nestjs/core`.
- For **pnpm Workspaces**, set `@nestjs/core` as a `peerDependencies` in your other module and `"dependenciesMeta": {{ '{' }}"other-module-name": {{ '{' }}"injected": true{{ '}' }}' }}` in the app package.json where the module is imported. see: [dependenciesmetainjected](#)

### **"Circular dependency" error**

Occasionally you'll find it difficult to avoid [circular dependencies](#) in your application. You'll need to take some steps to help Nest resolve these. Errors that arise from circular dependencies look like this:

```
Nest cannot create the <module> instance.
The module at index [<index>] of the <module> "imports" array is
undefined.
```

Potential causes:

- A circular dependency between modules. Use `forwardRef()` to avoid it. Read more: <https://docs.nestjs.com/fundamentals/circular-dependency>
- The module at index [<index>] is of type "undefined". Check your import

```
statements and the type of the module.
```

```
Scope [<module_import_chain>]  
# example chain AppModule -> FooModule
```

Circular dependencies can arise from both providers depending on each other, or typescript files depending on each other for constants, such as exporting constants from a module file and importing them in a service file. In the latter case, it is advised to create a separate file for your constants. In the former case, please follow the guide on circular dependencies and make sure that both the modules **and** the providers are marked with `forwardRef`.

## Debugging dependency errors

Along with just manually verifying your dependencies are correct, as of Nest 8.1.0 you can set the `NEST_DEBUG` environment variable to a string that resolves as truthy, and get extra logging information while Nest is resolving all of the dependencies for the application.



In the above image, the string in yellow is the host class of the dependency being injected, the string in blue is the name of the injected dependency, or its injection token, and the string in purple is the module in which the dependency is being searched for. Using this, you can usually trace back the dependency resolution for what's happening and why you're getting dependency injection problems.

## "File change detected" loops endlessly

Windows users who are using TypeScript version 4.9 and up may encounter this problem. This happens when you're trying to run your application in watch mode, e.g `npm run start:dev` and see an endless loop of the log messages:

```
XX:XX:XX AM - File change detected. Starting incremental compilation...  
XX:XX:XX AM - Found 0 errors. Watching for file changes.
```

When you're using the NestJS CLI to start your application in watch mode it is done by calling `tsc --watch`, and as of version 4.9 of TypeScript, a [new strategy](#) for detecting file changes is used which is likely to be the cause of this problem. In order to fix this problem, you need to add a setting to your `tsconfig.json` file after the `"compilerOptions"` option as follows:

```
"watchOptions": {  
  "watchFile": "fixedPollingInterval"  
}
```

This tells TypeScript to use the polling method for checking for file changes instead of file system events (the new default method), which can cause issues on some machines. You can read more about the `"watchFile"` option in [TypeScript documentation](#).

## Overview

**info Hint** This chapter covers the Nest Devtools integration with the Nest framework. If you are looking for the Devtools application, please visit the [Devtools](#) website.

To start debugging your local application, open up the `main.ts` file and make sure to set the `snapshot` attribute to `true` in the application options object, as follows:

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule, {
    snapshot: true,
  });
  await app.listen(3000);
}
```

This will instruct the framework to collect necessary metadata that will let Nest Devtools visualize your application's graph.

Next up, let's install the required dependency:

```
$ npm i @nestjs/devtools-integration
```

**warning Warning** If you're using `@nestjs/graphql` package in your application, make sure to install the latest version (`npm i @nestjs/graphql@11`).

With this dependency in place, let's open up the `app.module.ts` file and import the `DevtoolsModule` that we just installed:

```
@Module({
  imports: [
    DevtoolsModule.register({
      http: process.env.NODE_ENV !== 'production',
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

**warning Warning** The reason we are checking the `NODE_ENV` environment variable here is that you should never use this module in production!

Once the `DevtoolsModule` is imported and your application is up and running (`npm run start:dev`), you should be able to navigate to [Devtools](#) URL and see the introspected graph.



**info Hint** As you can see on the screenshot above, every module connects to the `InternalCoreModule`. `InternalCoreModule` is a global module that is always imported into the root module. Since it's registered as a global node, Nest automatically creates edges between all of the modules and the `InternalCoreModule` node. Now, if you want to hide global modules from the graph, you can use the "`Hide global modules`" checkbox (in the sidebar).

So as we can see, `DevtoolsModule` makes your application expose an additional HTTP server (on port 8000) that the Devtools application will use to introspect your app.

Just to double-check that everything works as expected, change the graph view to "Classes". You should see the following screen:



To focus on a specific node, click on the rectangle and the graph will show a popup window with the "`Focus`" button. You can also use the search bar (located in the sidebar) to find a specific node.

**info Hint** If you click on the `Inspect` button, application will take you to the `/debug` page with that specific node selected.



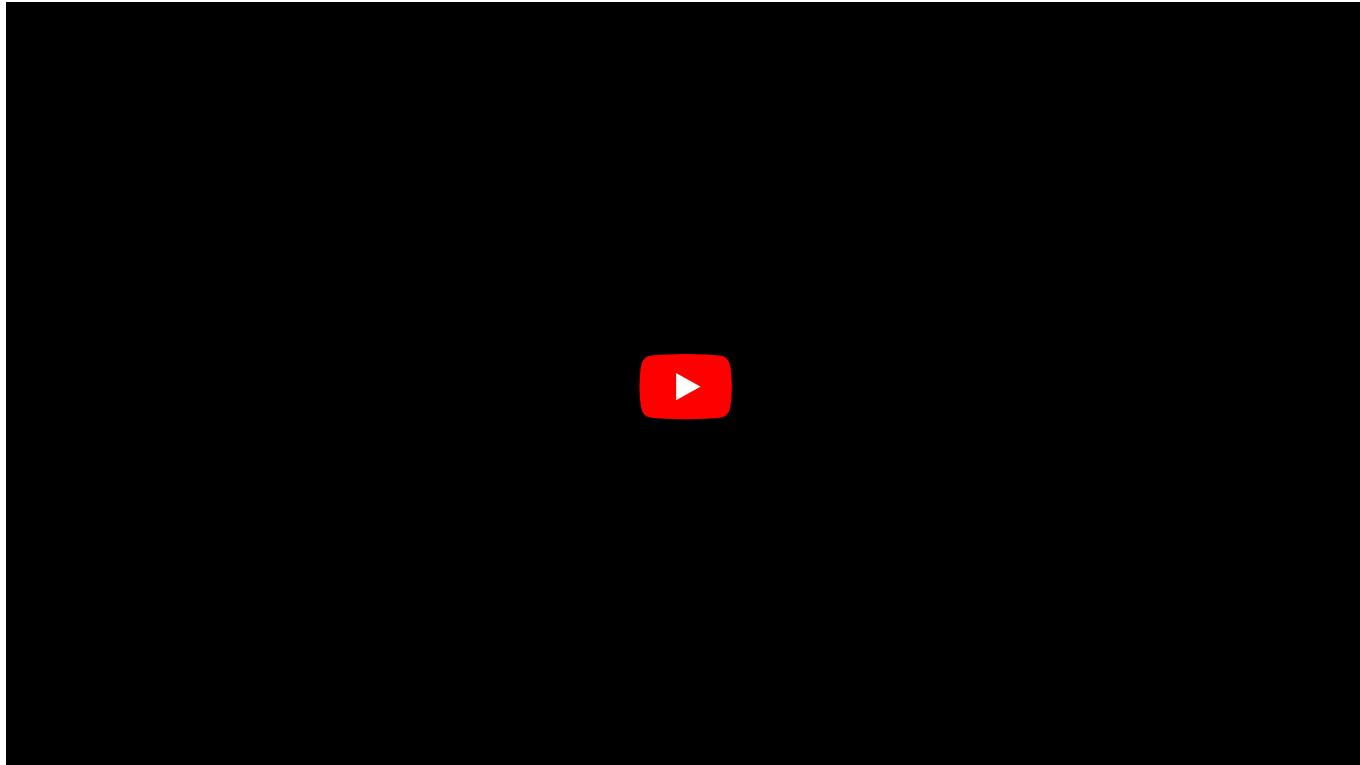
**info Hint** To export a graph as an image, click on the `Export as PNG` button in the right corner of the graph.

Using the form controls located in the sidebar (on the left), you can control edges proximity to, for example, visualize a specific application sub-tree:



This can be particularly useful when you have **new developers** on your team and you want to show them how your application is structured. You can also use this feature to visualize a specific module (e.g. `TasksModule`) and all of its dependencies, which can come in handy when you're breaking down a large application into smaller modules (for example, individual micro-services).

You can watch this video to see the **Graph Explorer** feature in action:



## Investigating the "Cannot resolve dependency" error

info Note This feature is supported for `@nestjs/core >= v9.3.10`.

Probably the most common error message you might have seen is about Nest not being able to resolve dependencies of a provider. Using Nest Devtools, you can effortlessly identify the issue and learn how to resolve it.

First, open up the `main.ts` file and update the `bootstrap()` call, as follows:

```
bootstrap().catch((err) => {
  fs.writeFileSync('graph.json', PartialGraphHost.toString() ?? '');
  process.exit(1);
});
```

Also, make sure to set the `abortOnError` to `false`:

```
const app = await NestFactory.create(AppModule, {
  snapshot: true,
  abortOnError: false, // <---- THIS
});
```

Now every time your application fails to bootstrap due to the "Cannot resolve dependency" error, you'll find the `graph.json` (that represents a partial graph) file in the root directory. You can then drag & drop this file into Devtools (make sure to switch the current mode from "Interactive" to "Preview"):



Upon successful upload, you should see the following graph & dialog window:



As you can see, the highlighted `TasksModule` is the one we should look into. Also, in the dialog window you can already see some instructions on how to use fix this issue.

If we switch to the "Classes" view instead, that's what we'll see:



This graph illustrates that the `DiagnosticsService` which we want to inject into the `TasksService` was not found in the context of the `TasksModule` module, and we should likely just import the `DiagnosticsModule` into the `TasksModule` module to fix this up!

## Routes explorer

When you navigate to the `Routes explorer` page, you should see all of the registered entrypoints:



**info Hint** This page shows not only HTTP routes, but also all of the other entrypoints (e.g. WebSockets, gRPC, GraphQL resolvers etc.).

Entrypoints are grouped by their host controllers. You can also use the search bar to find a specific endpoint.

If you click on a specific endpoint, a **flow graph** will be displayed. This graph shows the execution flow of the endpoint (e.g. guards, interceptors, pipes, etc. bound to this route). This is particularly useful when you want to understand how the request/response cycle looks for a specific route, or when troubleshooting why a specific guard/interceptor/pipe is not being executed.

## Sandbox

To execute JavaScript code on the fly & interact with your application in real-time, navigate to the **Sandbox** page:



The playground can be used to test and debug API endpoints in **real-time**, allowing developers to quickly identify and fix issues without using, for example, an HTTP client. We can also bypass the authentication layer, and so we no longer need that extra step of logging in, or even a special user account for testing purposes. For event-driven applications, we can also trigger events directly from the playground, and see how the application reacts to them.

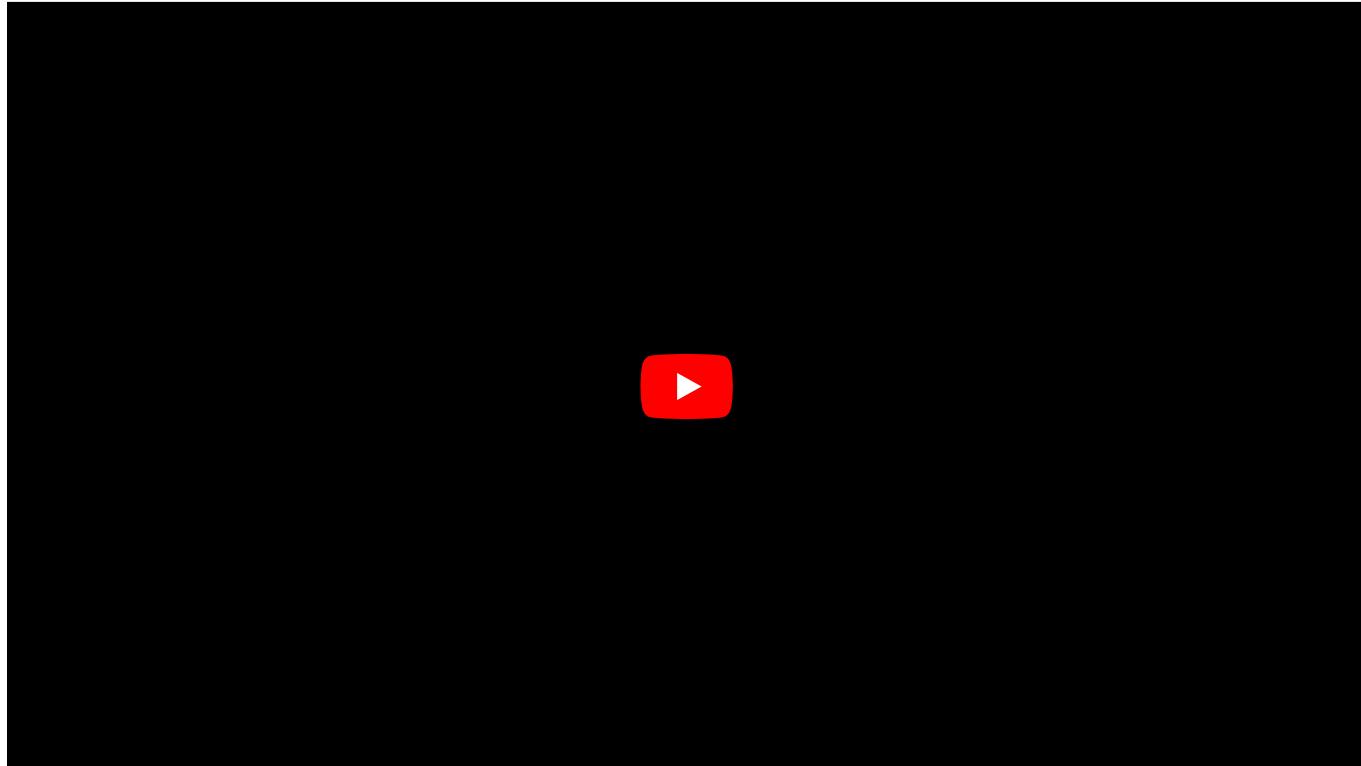
Anything that gets logged down is streamlined to the playground's console, so we can easily see what's going on.

Just execute the code **on the fly** and see the results instantly, without having to rebuild the application and restart the server.



**info Hint** To pretty display an array of objects, use the `console.table()` (or just `table()`) function.

You can watch this video to see the **Interactive Playground** feature in action:



## Bootstrap performance analyzer

To see a list of all class nodes (controllers, providers, enhancers, etc.) and their corresponding instantiation times, navigate to the **Bootstrap performance** page:



This page is particularly useful when you want to identify the slowest parts of your application's bootstrap process (e.g. when you want to optimize the application's startup time which is crucial for, for example, serverless environments).

## Audit

To see the auto-generated audit - errors/warnings/hints that the application came up with while analyzing your serialized graph, navigate to the **Audit** page:



**info Hint** The screenshot above doesn't show all of the available audit rules.

This page comes in handy when you want to identify potential issues in your application.

## Preview static files

To save a serialized graph to a file, use the following code:

```
await app.listen(3000); // OR await app.init()  
fs.writeFileSync('./graph.json', app.get(SerializedGraph).toString());
```

 info Hint `SerializedGraph` is exported from the `@nestjs/core` package.

Then you can drag and drop/upload this file:



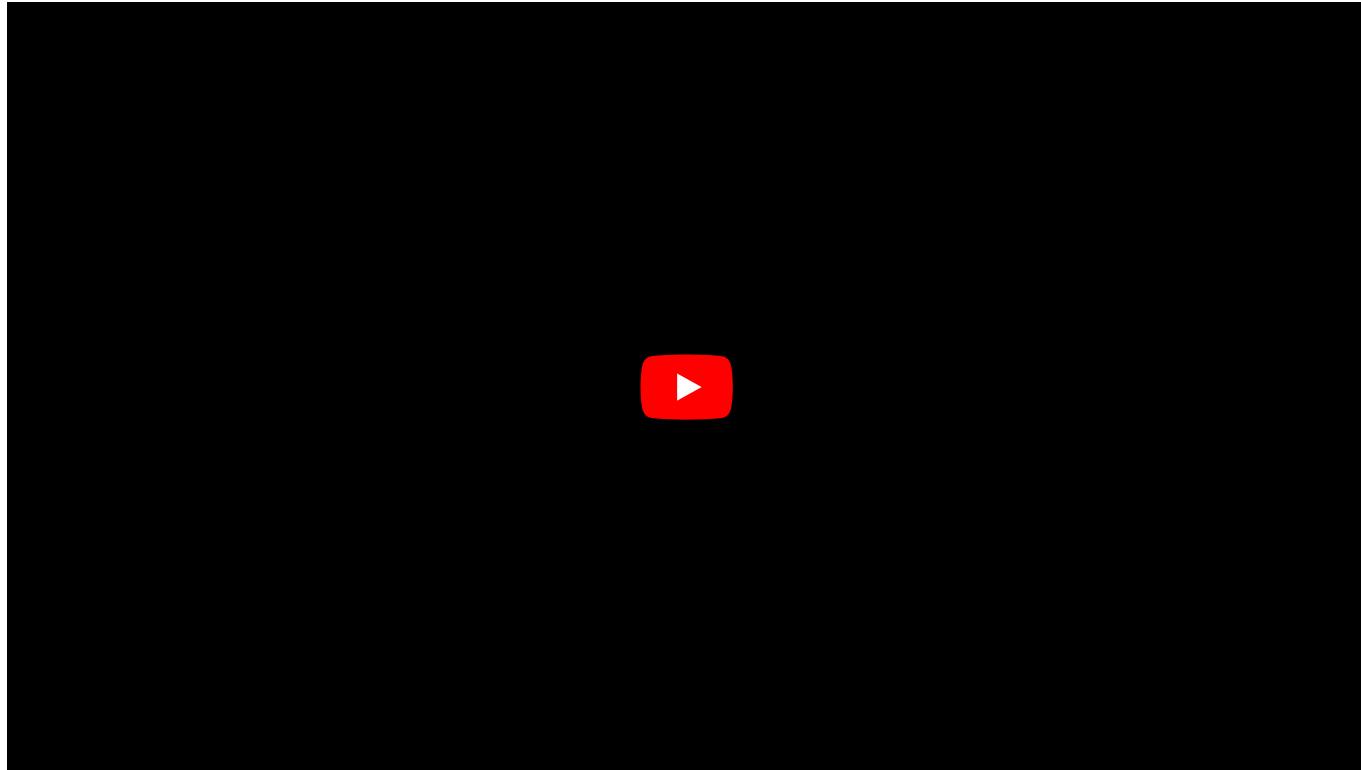
This is helpful when you want to share your graph with someone else (e.g., co-worker), or when you want to analyze it offline.

## CI/CD integration

**info Hint** This chapter covers the Nest Devtools integration with the Nest framework. If you are looking for the Devtools application, please visit the [Devtools](#) website.

CI/CD integration is available for users with the [Enterprise](#) plan.

You can watch this video to learn why & how CI/CD integration can help you:



### Publishing graphs

Let's first configure the application bootstrap file ([main.ts](#)) to use the [GraphPublisher](#) class (exported from the [@nestjs/devtools-integration](#) - see previous chapter for more details), as follows:

```
async function bootstrap() {
  const shouldPublishGraph = process.env.PUBLISH_GRAPH === "true";

  const app = await NestFactory.create(AppModule, {
    snapshot: true,
    preview: shouldPublishGraph,
  });

  if (shouldPublishGraph) {
    await app.init();

    const publishOptions = { ... } // NOTE: this options object will vary depending on the CI/CD provider you're
    using
    const graphPublisher = new GraphPublisher(app);
    await graphPublisher.publish(publishOptions);

    await app.close();
  } else {
    await app.listen(3000);
  }
}
```

As we can see, we're using the [GraphPublisher](#) here to publish our serialized graph to the centralized registry. The [PUBLISH\\_GRAPH](#) is a custom environment variable that will let us control whether the graph should be published (CI/CD workflow), or not (regular application bootstrap). Also, we set the [preview](#) attribute here to [true](#). With this flag enabled, our application will bootstrap in the preview mode - which basically means that constructors (and lifecycle hooks) of all controllers, enhancers, and providers in our application will not be executed. Note - this isn't [required](#), but makes things simpler for us since in this case we won't really have to connect to the database etc. when running our application in the CI/CD pipeline.

The [publishOptions](#) object will vary depending on the CI/CD provider you're using. We will provide you with instructions for the most popular CI/CD providers below, in later sections.

Once the graph is successfully published, you'll see the following output in your workflow view:



Every time our graph is published, we should see a new entry in the project's corresponding page:



## Reports

Devtools generate a report for every build **IF** there's a corresponding snapshot already stored in the centralized registry. So for example, if you create a PR against the `master` branch for which the graph was already published - then the application will be able to detect differences and generate a report. Otherwise, the report will not be generated.

To see reports, navigate to the project's corresponding page (see organizations).



This is particularly helpful in identifying changes that may have gone unnoticed during code reviews. For instance, let's say someone has changed the scope of a **deeply nested provider**. This change might not be immediately obvious to the reviewer, but with Devtools, we can easily spot such changes and make sure that they're intentional. Or if we remove a guard from a specific endpoint, it will show up as affected in the report. Now if we didn't have integration or e2e tests for that route, we might not notice that it's no longer protected, and by the time we do, it could be too late.

Similarly, if we're working on a **large codebase** and we modify a module to be global, we'll see how many edges were added to the graph, and this - in most cases - is a sign that we're doing something wrong.

## Build preview

For every published graph we can go back in time and preview how it looked before by clicking at the **Preview** button. Furthermore, if the report was generated, we should see the differences highlighted on our graph:

- green nodes represent added elements
- light white nodes represent updated elements
- red nodes represent deleted elements

See screenshot below:



The ability to go back in time lets you investigate and troubleshoot the issue by comparing the current graph with the previous one. Depending on how you set things up, every pull request (or even every commit) will have a corresponding snapshot in the registry, so you can easily go back in time and see what changed. Think of Devtools as a Git but with an understanding of how Nest constructs your application graph, and with the ability to **visualize** it.

## Integrations: Github Actions

First let's start from creating a new Github workflow in the `.github/workflows` directory in our project and call it, for example, `publish-graph.yml`. Inside this file, let's use the following definition:

```
name: Devtools

on:
  push:
    branches:
      - master
  pull_request:
    branches:
      - '*'

jobs:
  publish:
    if: github.actor!= 'dependabot[bot]'
    name: Publish graph
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '16'
          cache: 'npm'
      - name: Install dependencies
        run: npm ci
      - name: Setup Environment (PR)
        if: {{ '${{' }} github.event_name == 'pull_request' {{ '}}' }}
        shell: bash
        run:
          echo "COMMIT_SHA={{ '${{' }} github.event.pull_request.head.sha {{ '}}' }}" >> ${GITHUB_ENV}
      - name: Setup Environment (Push)
```

```

if: {{ '${{' }} github.event_name == 'push' {{ '}}' }}
shell: bash
run: |
  echo "COMMIT_SHA=\${GITHUB_SHA}" >> \${GITHUB_ENV}
- name: Publish
  run: PUBLISH_GRAPH=true npm run start
env:
  DEVTOOLS_API_KEY: CHANGE_THIS_TO_YOUR_API_KEY
  REPOSITORY_NAME: {{ '${{' }} github.event.repository.name {{ '}}' }}
  BRANCH_NAME: {{ '${{' }} github.head_ref || github.ref_name {{ '}}' }}
  TARGET_SHA: {{ '${{' }} github.event.pull_request.base.sha {{ '}}' }}

```

Ideally, `DEVTOOLS_API_KEY` environment variable should be retrieved from Github Secrets, read more [here](#).

This workflow will run per each pull request that's targeting the `master` branch OR in case there's a direct commit to the `master` branch. Feel free to align this configuration to whatever your project needs. What's essential here is that we provide necessary environment variables for our `GraphPublisher` class (to run).

However, there's one variable that needs to be updated before we can start using this workflow - `DEVTOOLS_API_KEY`. We can generate an API key dedicated for our project on this [page](#).

Lastly, let's navigate to the `main.ts` file again and update the `publishOptions` object we previously left empty.

```

const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY,
  repository: process.env.REPOSITORY_NAME,
  owner: process.env.GITHUB_REPOSITORY_OWNER,
  sha: process.env.COMMIT_SHA,
  target: process.env.TARGET_SHA,
  trigger: process.env.GITHUB_BASE_REF ? 'pull' : 'push',
  branch: process.env.BRANCH_NAME,
};

```

For the best developer experience, make sure to integrate the **Github application** for your project by clicking on the "Integrate Github app" button (see screenshot below). Note - this isn't required.



With this integration, you'll be able to see the status of the preview/report generation process right in your pull request:



### Integrations: Gitlab Pipelines

First let's start from creating a new Gitlab CI configuration file in the root directory of our project and call it, for example, `.gitlab-ci.yml`. Inside this file, let's use the following definition:

```

const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY,
  repository: process.env.REPOSITORY_NAME,
  owner: process.env.GITHUB_REPOSITORY_OWNER,
  sha: process.env.COMMIT_SHA,
  target: process.env.TARGET_SHA,
  trigger: process.env.GITHUB_BASE_REF ? 'pull' : 'push',
  branch: process.env.BRANCH_NAME,
};

```

**info Hint** Ideally, `DEVTOOLS_API_KEY` environment variable should be retrieved from secrets.

This workflow will run per each pull request that's targeting the `master` branch OR in case there's a direct commit to the `master` branch. Feel free to align this configuration to whatever your project needs. What's essential here is that we provide necessary environment variables for our `GraphPublisher` class (to run).

However, there's one variable (in this workflow definition) that needs to be updated before we can start using this workflow - `DEVTOOLS_API_KEY`. We can generate an API key dedicated for our project on this [page](#).

Lastly, let's navigate to the `main.ts` file again and update the `publishOptions` object we previously left empty.

```

image: node:16
stages:
  - build
cache:

```

```

key:
  files:
    - package-lock.json
paths:
  - node_modules/

workflow:
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      when: always
    - if: $CI_COMMIT_BRANCH == "master" && $CI_PIPELINE_SOURCE == "push"
      when: always
    - when: never

install_dependencies:
  stage: build
  script:
    - npm ci

publish_graph:
  stage: build
  needs:
    - install_dependencies
  script: npm run start
  variables:
    PUBLISH_GRAPH: 'true'
    DEVTOOLS_API_KEY: 'CHANGE_THIS_TO_YOUR_API_KEY'

```

## Other CI/CD tools

Nest Devtools CI/CD integration can be used with any CI/CD tool of your choice (e.g., [Bitbucket Pipelines](#), [CircleCI](#), etc) so don't feel limited to providers we described here.

Look at the following `publishOptions` object configuration to understand what information is required to publish the graph for a given commit/build/PR.

```

const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY,
  repository: process.env.CI_PROJECT_NAME,
  owner: process.env.CI_PROJECT_ROOT_NAMESPACE,
  sha: process.env.CI_COMMIT_SHA,
  target: process.env.CI_MERGE_REQUEST_DIFF_BASE_SHA,
  trigger: process.env.CI_MERGE_REQUEST_DIFF_BASE_SHA ? 'pull' : 'push',
  branch:
    process.env.CI_COMMIT_BRANCH ??
    process.env.CI_MERGE_REQUEST_SOURCE_BRANCH_NAME,
};

```

Most of this information is provided through CI/CD built-in environment variables (see [CircleCI built-in environment list](#) and [Bitbucket variables](#)).

When it comes to the pipeline configuration for publishing graphs, we recommend using the following triggers:

- `push` event - only if the current branch represents a deployment environment, for example `master`, `main`, `staging`, `production`, etc.
- `pull request` event - always, or when the **target branch** represents a deployment environment (see above)

# Harnessing the power of TypeScript & GraphQL

GraphQL is a powerful query language for APIs and a runtime for fulfilling those queries with your existing data. It's an elegant approach that solves many problems typically found with REST APIs. For background, we suggest reading this [comparison](#) between GraphQL and REST. GraphQL combined with [TypeScript](#) helps you develop better type safety with your GraphQL queries, giving you end-to-end typing.

In this chapter, we assume a basic understanding of GraphQL, and focus on how to work with the built-in `@nestjs/graphql` module. The `GraphQLModule` can be configured to use [Apollo](#) server (with the `@nestjs/apollo` driver) and [Mercurius](#) (with the `@nestjs/mercurius`). We provide official integrations for these proven GraphQL packages to provide a simple way to use GraphQL with Nest (see more integrations [here](#)).

You can also build your own dedicated driver (read more on that [here](#)).

## Installation

Start by installing the required packages:

```
# For Express and Apollo (default)
$ npm i @nestjs/graphql @nestjs/apollo @apollo/server graphql

# For Fastify and Apollo
# npm i @nestjs/graphql @nestjs/apollo @apollo/server @as-
integrations/fastify graphql

# For Fastify and Mercurius
# npm i @nestjs/graphql @nestjs/mercurius graphql mercurius
```

**warning** **Warning** `@nestjs/graphql@>=9` and `@nestjs/apollo^10` packages are compatible with **Apollo v3** (check out Apollo Server 3 [migration guide](#) for more details), while `@nestjs/graphql@^8` only supports **Apollo v2** (e.g., `apollo-server-express@2.x.x` package).

## Overview

Nest offers two ways of building GraphQL applications, the **code first** and the **schema first** methods. You should choose the one that works best for you. Most of the chapters in this GraphQL section are divided into two main parts: one you should follow if you adopt **code first**, and the other to be used if you adopt **schema first**.

In the **code first** approach, you use decorators and TypeScript classes to generate the corresponding GraphQL schema. This approach is useful if you prefer to work exclusively with TypeScript and avoid context switching between language syntaxes.

In the **schema first** approach, the source of truth is GraphQL SDL (Schema Definition Language) files. SDL is a language-agnostic way to share schema files between different platforms. Nest automatically generates

your TypeScript definitions (using either classes or interfaces) based on the GraphQL schemas to reduce the need to write redundant boilerplate code.

## Getting started with GraphQL & TypeScript

**info Hint** In the following chapters, we'll be integrating the `@nestjs/apollo` package. If you want to use `mercurius` package instead, navigate to [this section](#).

Once the packages are installed, we can import the `GraphQLModule` and configure it with the `forRoot()` static method.

```
@@filename()
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
    }),
  ],
})
export class AppModule {}
```

**info Hint** For `mercurius` integration, you should be using the `MercuriusDriver` and `MercuriusDriverConfig` instead. Both are exported from the `@nestjs/mercurius` package.

The `forRoot()` method takes an options object as an argument. These options are passed through to the underlying driver instance (read more about available settings here: [Apollo](#) and [Mercurius](#)). For example, if you want to disable the `playground` and turn off `debug` mode (for Apollo), pass the following options:

```
@@filename()
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      playground: false,
    }),
  ],
})
export class AppModule {}
```

In this case, these options will be forwarded to the `ApolloServer` constructor.

## GraphQL playground

The playground is a graphical, interactive, in-browser GraphQL IDE, available by default on the same URL as the GraphQL server itself. To access the playground, you need a basic GraphQL server configured and running. To see it now, you can install and build the [working example here](#). Alternatively, if you're following along with these code samples, once you've completed the steps in the [Resolvers chapter](#), you can access the playground.

With that in place, and with your application running in the background, you can then open your web browser and navigate to `http://localhost:3000/graphql` (host and port may vary depending on your configuration). You will then see the GraphQL playground, as shown below.

**warning** **Note** `@nestjs/mercurius` integration does not ship with the built-in GraphQL Playground integration. Instead, you can use [GraphiQL](#) (set `graphiql: true`).

## Multiple endpoints

Another useful feature of the `@nestjs/graphql` module is the ability to serve multiple endpoints at once. This lets you decide which modules should be included in which endpoint. By default, [GraphQL](#) searches for resolvers throughout the whole app. To limit this scan to only a subset of modules, use the `include` property.

```
GraphQLModule.forRoot({
  include: [CatsModule],
}),
```

**warning** **Warning** If you use the `@apollo/server` with `@as-integrations/fastify` package with multiple GraphQL endpoints in a single application, make sure to enable the `disableHealthCheck` setting in the [GraphQLModule](#) configuration.

## Code first

In the **code first** approach, you use decorators and TypeScript classes to generate the corresponding GraphQL schema.

To use the code first approach, start by adding the `autoSchemaFile` property to the options object:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  autoSchemaFile: join(process.cwd(), 'src/schema.gql'),
}),
```

The `autoSchemaFile` property value is the path where your automatically generated schema will be created. Alternatively, the schema can be generated on-the-fly in memory. To enable this, set the `autoSchemaFile` property to `true`:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  autoSchemaFile: true,
}),
```

By default, the types in the generated schema will be in the order they are defined in the included modules. To sort the schema lexicographically, set the `sortSchema` property to `true`:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  autoSchemaFile: join(process.cwd(), 'src/schema.gql'),
  sortSchema: true,
}),
```

## Example

A fully working code first sample is available [here](#).

## Schema first

To use the schema first approach, start by adding a `typePaths` property to the options object. The `typePaths` property indicates where the `GraphQLModule` should look for GraphQL SDL schema definition files you'll be writing. These files will be combined in memory; this allows you to split your schemas into several files and locate them near their resolvers.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  typePaths: ['./**/*.graphql'],
}),
```

You will typically also need to have TypeScript definitions (classes and interfaces) that correspond to the GraphQL SDL types. Creating the corresponding TypeScript definitions by hand is redundant and tedious. It leaves us without a single source of truth -- each change made within SDL forces us to adjust TypeScript definitions as well. To address this, the `@nestjs/graphql` package can **automatically generate** TypeScript definitions from the abstract syntax tree (`AST`). To enable this feature, add the `definitions` options property when configuring the `GraphQLModule`.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  typePaths: ['./**/*.graphql'],
  definitions: {
    path: join(process.cwd(), 'src/graphql.ts'),
  },
}),
```

The path property of the `definitions` object indicates where to save generated TypeScript output. By default, all generated TypeScript types are created as interfaces. To generate classes instead, specify the `outputAs` property with a value of '`class`'.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  typePaths: ['./**/*.graphql'],
  definitions: {
    path: join(process.cwd(), 'src/graphql.ts'),
    outputAs: 'class',
  },
}),
```

The above approach dynamically generates TypeScript definitions each time the application starts. Alternatively, it may be preferable to build a simple script to generate these on demand. For example, assume we create the following script as `generate-typings.ts`:

```
import { GraphQLDefinitionsFactory } from '@nestjs/graphql';
import { join } from 'path';

const definitionsFactory = new GraphQLDefinitionsFactory();
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  path: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
});
```

Now you can run this script on demand:

```
$ ts-node generate-typings
```

**info Hint** You can compile the script beforehand (e.g., with `tsc`) and use `node` to execute it.

To enable watch mode for the script (to automatically generate typings whenever any `.graphql` file changes), pass the `watch` option to the `generate()` method.

```
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  path: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
  watch: true,
});
```

To automatically generate the additional `__typename` field for every object type, enable the `emitTypenameField` option.

```
definitionsFactory.generate({
  // ...,
  emitTypenameField: true,
});
```

To generate resolvers (queries, mutations, subscriptions) as plain fields without arguments, enable the `skipResolverArgs` option.

```
definitionsFactory.generate({
  // ...,
  skipResolverArgs: true,
});
```

## Apollo Sandbox

To use [Apollo Sandbox](#) instead of the `graphql-playground` as a GraphQL IDE for local development, use the following configuration:

```
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { ApolloServerPluginLandingPageLocalDefault } from
  '@apollo/server/plugin/landingPage/default';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      playground: false,
      plugins: [ApolloServerPluginLandingPageLocalDefault()],
    }),
  ],
})
export class AppModule {}
```

## Example

A fully working schema first sample is available [here](#).

## Accessing generated schema

In some circumstances (for example end-to-end tests), you may want to get a reference to the generated schema object. In end-to-end tests, you can then run queries using the `graphql` object without using any HTTP listeners.

You can access the generated schema (in either the code first or schema first approach), using the `GraphQLSchemaHost` class:

```
const { schema } = app.get(GraphQLSchemaHost);
```

**info Hint** You must call the `GraphQLSchemaHost#schema` getter after the application has been initialized (after the `onModuleInit` hook has been triggered by either the `app.listen()` or `app.init()` method).

## Async configuration

When you need to pass module options asynchronously instead of statically, use the `forRootAsync()` method. As with most dynamic modules, Nest provides several techniques to deal with async configuration.

One technique is to use a factory function:

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
  driver: ApolloDriver,
  useFactory: () => ({
    typePaths: ['./**/*.graphql'],
  }),
}) ,
```

Like other factory providers, our factory function can be `async` and can inject dependencies through `inject`.

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
  driver: ApolloDriver,
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    typePaths: configService.get<string>('GRAPHQL_TYPE_PATHS'),
  }),
  inject: [ConfigService],
}) ,
```

Alternatively, you can configure the `GraphQLModule` using a class instead of a factory, as shown below:

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
  driver: ApolloDriver,
  useClass: GqlConfigService,
}) ,
```

The construction above instantiates `GqlConfigService` inside `GraphQLModule`, using it to create options object. Note that in this example, the `GqlConfigService` has to implement the `GqlOptionsFactory` interface, as shown below. The `GraphQLModule` will call the `createGqlOptions()` method on the instantiated object of the supplied class.

```
@Injectable()
class GqlConfigService implements GqlOptionsFactory {
    createGqlOptions(): ApolloDriverConfig {
        return {
            typePaths: ['./**/*.graphql'],
        };
    }
}
```

If you want to reuse an existing options provider instead of creating a private copy inside the `GraphQLModule`, use the `useExisting` syntax.

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
    imports: [ConfigModule],
    useExisting: ConfigService,
}),
```

## Mercurius integration

Instead of using Apollo, Fastify users (read more [here](#)) can alternatively use the `@nestjs/mercurius` driver.

```
@@filename()
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { MercuriusDriver, MercuriusDriverConfig } from
    '@nestjs/mercurius';

@Module({
    imports: [
        GraphQLModule.forRoot<MercuriusDriverConfig>({
            driver: MercuriusDriver,
            graphql: true,
        }),
    ],
})
export class AppModule {}
```

**info Hint** Once the application is running, open your browser and navigate to <http://localhost:3000/graphiql>. You should see the [GraphQL IDE](#).

The `forRoot()` method takes an options object as an argument. These options are passed through to the underlying driver instance. Read more about available settings [here](#).

## Third-party integrations

- [GraphQL Yoga](#)

## Example

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

## Resolvers

Resolvers provide the instructions for turning a [GraphQL](#) operation (a query, mutation, or subscription) into data. They return the same shape of data we specify in our schema -- either synchronously or as a promise that resolves to a result of that shape. Typically, you create a **resolver map** manually. The [@nestjs/graphql](#) package, on the other hand, generates a resolver map automatically using the metadata provided by decorators you use to annotate classes. To demonstrate the process of using the package features to create a GraphQL API, we'll create a simple authors API.

### Code first

In the code first approach, we don't follow the typical process of creating our GraphQL schema by writing GraphQL SDL by hand. Instead, we use TypeScript decorators to generate the SDL from TypeScript class definitions. The [@nestjs/graphql](#) package reads the metadata defined through the decorators and automatically generates the schema for you.

### Object types

Most of the definitions in a GraphQL schema are **object types**. Each object type you define should represent a domain object that an application client might need to interact with. For example, our sample API needs to be able to fetch a list of authors and their posts, so we should define the [Author](#) type and [Post](#) type to support this functionality.

If we were using the schema first approach, we'd define such a schema with SDL like this:

```
type Author {  
    id: Int!  
    firstName: String  
    lastName: String  
    posts: [Post!]!  
}
```

In this case, using the code first approach, we define schemas using TypeScript classes and using TypeScript decorators to annotate the fields of those classes. The equivalent of the above SDL in the code first approach is:

```
@@filename(authors/models/author.model)  
import { Field, Int, ObjectType } from '@nestjs/graphql';  
import { Post } from './post';  
  
@ObjectType()  
export class Author {  
    @Field(type => Int)  
    id: number;  
  
    @Field({ nullable: true })  
    firstName?: string;
```

```

@Field({ nullable: true })
lastName?: string;

@Field(type => [Post])
posts: Post[];
}

```

**info Hint** TypeScript's metadata reflection system has several limitations which make it impossible, for instance, to determine what properties a class consists of or recognize whether a given property is optional or required. Because of these limitations, we must either explicitly use the `@Field()` decorator in our schema definition classes to provide metadata about each field's GraphQL type and optionality, or use a [CLI plugin](#) to generate these for us.

The `Author` object type, like any class, is made of a collection of fields, with each field declaring a type. A field's type corresponds to a [GraphQL type](#). A field's GraphQL type can be either another object type or a scalar type. A GraphQL scalar type is a primitive (like `ID`, `String`, `Boolean`, or `Int`) that resolves to a single value.

**info Hint** In addition to GraphQL's built-in scalar types, you can define custom scalar types ([read more](#)).

The above `Author` object type definition will cause Nest to **generate** the SDL we showed above:

```

type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post!]!
}

```

The `@Field()` decorator accepts an optional type function (e.g., `type => Int`), and optionally an options object.

The type function is required when there's the potential for ambiguity between the TypeScript type system and the GraphQL type system. Specifically: it is **not** required for `string` and `boolean` types; it **is** required for `number` (which must be mapped to either a GraphQL `Int` or `Float`). The type function should simply return the desired GraphQL type (as shown in various examples in these chapters).

The options object can have any of the following key/value pairs:

- `nullable`: for specifying whether a field is nullable (in SDL, each field is non-nullable by default); `boolean`
- `description`: for setting a field description; `string`
- `deprecationReason`: for marking a field as deprecated; `string`

For example:

```
@Field({ description: `Book title`, deprecationReason: 'Not useful in v2 schema' })
title: string;
```

**info Hint** You can also add a description to, or deprecate, the whole object type: `@ObjectType({{ '{' }} description: 'Author model' {{ '}' }})`.

When the field is an array, we must manually indicate the array type in the `Field()` decorator's type function, as shown below:

```
@Field(type => [Post])
posts: Post[];
```

**info Hint** Using array bracket notation (`[ ]`), we can indicate the depth of the array. For example, using `[[Int]]` would represent an integer matrix.

To declare that an array's items (not the array itself) are nullable, set the `nullable` property to '`'items'`' as shown below:

```
@Field(type => [Post], { nullable: 'items' })
posts: Post[];
```

**info Hint** If both the array and its items are nullable, set `nullable` to '`'itemsAndList'`' instead.

Now that the `Author` object type is created, let's define the `Post` object type.

```
@@filename(posts/models/post.model)
import { Field, Int, ObjectType } from '@nestjs/graphql';

@ObjectType()
export class Post {
  @Field(type => Int)
  id: number;

  @Field()
  title: string;

  @Field(type => Int, { nullable: true })
  votes?: number;
}
```

The `Post` object type will result in generating the following part of the GraphQL schema in SDL:

```
type Post {  
  id: Int!  
  title: String!  
  votes: Int  
}
```

## Code first resolver

At this point, we've defined the objects (type definitions) that can exist in our data graph, but clients don't yet have a way to interact with those objects. To address that, we need to create a resolver class. In the code first method, a resolver class both defines resolver functions **and** generates the **Query type**. This will be clear as we work through the example below:

```
@@filename(authors/authors.resolver)  
@Resolver(of => Author)  
export class AuthorsResolver {  
  constructor(  
    private authorsService: AuthorsService,  
    private postsService: PostsService,  
  ) {}  
  
  @Query(returns => Author)  
  async author(@Args('id', { type: () => Int }) id: number) {  
    return this.authorsService.findOneById(id);  
  }  
  
  @ResolveField()  
  async posts(@Parent() author: Author) {  
    const { id } = author;  
    return this.postsService.findAll({ authorId: id });  
  }  
}
```

**info Hint** All decorators (e.g., `@Resolver`, `@ResolveField`, `@Args`, etc.) are exported from the `@nestjs/graphql` package.

You can define multiple resolver classes. Nest will combine these at run time. See the [module](#) section below for more on code organization.

**warning Note** The logic inside the `AuthorsService` and `PostsService` classes can be as simple or sophisticated as needed. The main point of this example is to show how to construct resolvers and how they can interact with other providers.

In the example above, we created the `AuthorsResolver` which defines one query resolver function and one field resolver function. To create a resolver, we create a class with resolver functions as methods, and annotate the class with the `@Resolver()` decorator.

In this example, we defined a query handler to get the author object based on the `id` sent in the request. To specify that the method is a query handler, use the `@Query()` decorator.

The argument passed to the `@Resolver()` decorator is optional, but comes into play when our graph becomes non-trivial. It's used to supply a parent object used by field resolver functions as they traverse down through an object graph.

In our example, since the class includes a **field resolver** function (for the `posts` property of the `Author` object type), we **must** supply the `@Resolver()` decorator with a value to indicate which class is the parent type (i.e., the corresponding `ObjectType` class name) for all field resolvers defined within this class. As should be clear from the example, when writing a field resolver function, it's necessary to access the parent object (the object the field being resolved is a member of). In this example, we populate an author's posts array with a field resolver that calls a service which takes the author's `id` as an argument. Hence the need to identify the parent object in the `@Resolver()` decorator. Note the corresponding use of the `@Parent()` method parameter decorator to then extract a reference to that parent object in the field resolver.

We can define multiple `@Query()` resolver functions (both within this class, and in any other resolver class), and they will be aggregated into a single **Query type** definition in the generated SDL along with the appropriate entries in the resolver map. This allows you to define queries close to the models and services that they use, and to keep them well organized in modules.

**info Hint** 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. Read more about this feature [here](#).

## Query type names

In the above examples, the `@Query()` decorator generates a GraphQL schema query type name based on the method name. For example, consider the following construction from the example above:

```
@Query(returns => Author)
async author(@Args('id', { type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

This generates the following entry for the author query in our schema (the query type uses the same name as the method name):

```
type Query {
  author(id: Int!): Author
}
```

**info Hint** Learn more about GraphQL queries [here](#).

Conventionally, we prefer to decouple these names; for example, we prefer to use a name like `getAuthor()` for our query handler method, but still use `author` for our query type name. The same

applies to our field resolvers. We can easily do this by passing the mapping names as arguments of the `@Query()` and `@ResolveField()` decorators, as shown below:

```
@@filename(authors/authors.resolver)
@Resolver(of => Author)
export class AuthorsResolver {
  constructor(
    private authorsService: AuthorsService,
    private postsService: PostsService,
  ) {}

  @Query(returns => Author, { name: 'author' })
  async getAuthor(@Args('id', { type: () => Int }) id: number) {
    return this.authorsService.findOneById(id);
  }

  @ResolveField('posts', returns => [Post])
  async getPosts(@Parent() author: Author) {
    const { id } = author;
    return this.postsService.findAll({ authorId: id });
  }
}
```

The `getAuthor` handler method above will result in generating the following part of the GraphQL schema in SDL:

```
type Query {
  author(id: Int!): Author
}
```

## Query decorator options

The `@Query()` decorator's options object (where we pass `{ name: 'author' }` above) accepts a number of key/value pairs:

- `name`: name of the query; a `string`
- `description`: a description that will be used to generate GraphQL schema documentation (e.g., in GraphQL playground); a `string`
- `deprecationReason`: sets query metadata to show the query as deprecated (e.g., in GraphQL playground); a `string`
- `nullable`: whether the query can return a null data response; `boolean` or `'items'` or `'itemsAndList'` (see above for details of `'items'` and `'itemsAndList'`)

## Args decorator options

Use the `@Args()` decorator to extract arguments from a request for use in the method handler. This works in a very similar fashion to [REST route parameter argument extraction](#).

Usually your `@Args()` decorator will be simple, and not require an object argument as seen with the `getAuthor()` method above. For example, if the type of an identifier is string, the following construction is sufficient, and simply plucks the named field from the inbound GraphQL request for use as a method argument.

```
@Args('id') id: string
```

In the `getAuthor()` case, the `number` type is used, which presents a challenge. The `number` TypeScript type doesn't give us enough information about the expected GraphQL representation (e.g., `Int` vs. `Float`). Thus we have to **explicitly** pass the type reference. We do that by passing a second argument to the `Args()` decorator, containing argument options, as shown below:

```
@Query(returns => Author, { name: 'author' })
async getAuthor(@Args('id', { type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

The options object allows us to specify the following optional key value pairs:

- `type`: a function returning the GraphQL type
- `defaultValue`: a default value; `any`
- `description`: description metadata; `string`
- `deprecationReason`: to deprecate a field and provide meta data describing why; `string`
- `nullable`: whether the field is nullable

Query handler methods can take multiple arguments. Let's imagine that we want to fetch an author based on its `firstName` and `lastName`. In this case, we can call `@Args` twice:

```
getAuthor(
  @Args('firstName', { nullable: true }) firstName?: string,
  @Args('lastName', { defaultValue: '' }) lastName?: string,
) {}
```

## Dedicated arguments class

With inline `@Args()` calls, code like the example above becomes bloated. Instead, you can create a dedicated `GetAuthorArgs` arguments class and access it in the handler method as follows:

```
@Args() args: GetAuthorArgs
```

Create the `GetAuthorArgs` class using `@ArgsType()` as shown below:

```

@@filename(authors/dto/get-author.args)
import { MinLength } from 'class-validator';
import { Field, ArgsType } from '@nestjs/graphql';

@ArgsType()
class GetAuthorArgs {
  @Field({ nullable: true })
  firstName?: string;

  @Field({ defaultValue: '' })
  @MinLength(3)
  lastName: string;
}

```

**info Hint** Again, due to TypeScript's metadata reflection system limitations, it's required to either use the `@Field` decorator to manually indicate type and optionality, or use a [CLI plugin](#).

This will result in generating the following part of the GraphQL schema in SDL:

```

type Query {
  author(firstName: String, lastName: String = ''): Author
}

```

**info Hint** Note that arguments classes like `GetAuthorArgs` play very well with the `ValidationPipe` ([read more](#)).

## Class inheritance

You can use standard TypeScript class inheritance to create base classes with generic utility type features (fields and field properties, validations, etc.) that can be extended. For example, you may have a set of pagination related arguments that always include the standard `offset` and `limit` fields, but also other index fields that are type-specific. You can set up a class hierarchy as shown below.

Base `@ArgsType()` class:

```

@ArgsType()
class PaginationArgs {
  @Field((type) => Int)
  offset: number = 0;

  @Field((type) => Int)
  limit: number = 10;
}

```

Type specific sub-class of the base `@ArgsType()` class:

```
@ArgsType()
class GetAuthorArgs extends PaginationArgs {
  @Field({ nullable: true })
  firstName?: string;

  @Field({ defaultValue: '' })
  @MinLength(3)
  lastName: string;
}
```

The same approach can be taken with `@ObjectType()` objects. Define generic properties on the base class:

```
@ObjectType()
class Character {
  @Field((type) => Int)
  id: number;

  @Field()
  name: string;
}
```

Add type-specific properties on sub-classes:

```
@ObjectType()
class Warrior extends Character {
  @Field()
  level: number;
}
```

You can use inheritance with a resolver as well. You can ensure type safety by combining inheritance and TypeScript generics. For example, to create a base class with a generic `findAll` query, use a construction like this:

```
function BaseResolver<T extends Type<unknown>>(<T>classRef: T): any {
  @Resolver({ isAbstract: true })
  abstract class BaseResolverHost {
    @Query((type) => [classRef], { name: `findAll${classRef.name}` })
    async findAll(): Promise<T[]> {
      return [];
    }
  }
  return BaseResolverHost;
}
```

Note the following:

- an explicit return type (`any` above) is required: otherwise TypeScript complains about the usage of a private class definition. Recommended: define an interface instead of using `any`.
- `Type` is imported from the `@nestjs/common` package
- The `isAbstract: true` property indicates that SDL (Schema Definition Language statements) shouldn't be generated for this class. Note, you can set this property for other types as well to suppress SDL generation.

Here's how you could generate a concrete sub-class of the `BaseResolver`:

```
@Resolver((of) => Recipe)
export class RecipesResolver extends BaseResolver(Recipe) {
  constructor(private recipesService: RecipesService) {
    super();
  }
}
```

This construct would generate the following SDL:

```
type Query {
  findAllRecipe: [Recipe!]!
}
```

## Generics

We saw one use of generics above. This powerful TypeScript feature can be used to create useful abstractions. For example, here's a sample cursor-based pagination implementation based on [this documentation](#):

```
import { Field, ObjectType, Int } from '@nestjs/graphql';
import { Type } from '@nestjs/common';

interface IEdgeType<T> {
  cursor: string;
  node: T;
}

export interface IPaginatedType<T> {
  edges: IEdgeType<T>[];
  nodes: T[];
  totalCount: number;
  hasNextPage: boolean;
}

export function Paginated<T>(classRef: Type<T>): Type<IPaginatedType<T>> {
  @ObjectType(` ${classRef.name}Edge`)
```

```

abstract class EdgeType {
  @Field((type) => String)
  cursor: string;

  @Field((type) => classRef)
  node: T;
}

@ObjectType({ isAbstract: true })
abstract class PaginatedType implements IPaginatedType<T> {
  @Field((type) => [EdgeType], { nullable: true })
  edges: EdgeType[];

  @Field((type) => [classRef], { nullable: true })
  nodes: T[];

  @Field((type) => Int)
  totalCount: number;

  @Field()
  hasNextPage: boolean;
}

return PaginatedType as Type<IPaginatedType<T>>;
}

```

With the above base class defined, we can now easily create specialized types that inherit this behavior. For example:

```

@ObjectType()
class PaginatedAuthor extends Paginated(Author) {}

```

## Schema first

As mentioned in the [previous](#) chapter, in the schema first approach we start by manually defining schema types in SDL (read [more](#)). Consider the following SDL type definitions.

**info Hint** For convenience in this chapter, we've aggregated all of the SDL in one location (e.g., one `.graphql` file, as shown below). In practice, you may find it appropriate to organize your code in a modular fashion. For example, it can be helpful to create individual SDL files with type definitions representing each domain entity, along with related services, resolver code, and the Nest module definition class, in a dedicated directory for that entity. Nest will aggregate all the individual schema type definitions at run time.

```

type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]

```

```
}

type Post {
  id: Int!
  title: String!
  votes: Int
}

type Query {
  author(id: Int!): Author
}
```

## Schema first resolver

The schema above exposes a single query - `author(id: Int!): Author`.

**info Hint** Learn more about GraphQL queries [here](#).

Let's now create an `AuthorsResolver` class that resolves author queries:

```
@@filename(authors/authors.resolver)
@Resolver('Author')
export class AuthorsResolver {
  constructor(
    private authorsService: AuthorsService,
    private postsService: PostsService,
  ) {}

  @Query()
  async author(@Args('id') id: number) {
    return this.authorsService.findOneById(id);
  }

  @ResolveField()
  async posts(@Parent() author) {
    const { id } = author;
    return this.postsService.findAll({ authorId: id });
  }
}
```

**info Hint** All decorators (e.g., `@Resolver`, `@ResolveField`, `@Args`, etc.) are exported from the `@nestjs/graphql` package.

**warning Note** The logic inside the `AuthorsService` and `PostsService` classes can be as simple or sophisticated as needed. The main point of this example is to show how to construct resolvers and how they can interact with other providers.

The `@Resolver()` decorator is required. It takes an optional string argument with the name of a class. This class name is required whenever the class includes `@ResolveField()` decorators to inform Nest that the

decorated method is associated with a parent type (the `Author` type in our current example). Alternatively, instead of setting `@Resolver()` at the top of the class, this can be done for each method:

```
@Resolver('Author')
@ResolveField()
async posts(@Parent() author) {
  const { id } = author;
  return this.postsService.findAll({ authorId: id });
}
```

In this case (`@Resolver()` decorator at the method level), if you have multiple `@ResolveField()` decorators inside a class, you must add `@Resolver()` to all of them. This is not considered the best practice (as it creates extra overhead).

**info Hint** Any class name argument passed to `@Resolver()` **does not** affect queries (`@Query()` decorator) or mutations (`@Mutation()` decorator).

**warning Warning** Using the `@Resolver` decorator at the method level is not supported with the **code first** approach.

In the above examples, the `@Query()` and `@ResolveField()` decorators are associated with GraphQL schema types based on the method name. For example, consider the following construction from the example above:

```
@Query()
async author(@Args('id') id: number) {
  return this.authorsService.findOneById(id);
}
```

This generates the following entry for the `author` query in our schema (the query type uses the same name as the method name):

```
type Query {
  author(id: Int!): Author
}
```

Conventionally, we would prefer to decouple these, using names like `getAuthor()` or `getPosts()` for our resolver methods. We can easily do this by passing the mapping name as an argument to the decorator, as shown below:

```
@@filename(authors/authors.resolver)
@Resolver('Author')
export class AuthorsResolver {
  constructor(
    private authorsService: AuthorsService,
```

```

    private postsService: PostsService,
) {}

@Query('author')
async getAuthor(@Args('id') id: number) {
  return this.authorsService.findOneById(id);
}

@ResolveField('posts')
async getPosts(@Parent() author) {
  const { id } = author;
  return this.postsService.findAll({ authorId: id });
}
}

```

**info Hint** 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. Read more about this feature [here](#).

## Generating types

Assuming that we use the schema first approach and have enabled the typings generation feature (with `outputAs: 'class'` as shown in the [previous](#) chapter), once you run the application it will generate the following file (in the location you specified in the `GraphQLModule.forRoot()` method). For example, in `src/graphql.ts`:

```

@@filename(graphql)
export class Author {
  id: number;
  firstName?: string;
  lastName?: string;
  posts?: Post[];
}

export class Post {
  id: number;
  title: string;
  votes?: number;
}

export abstract class IQuery {
  abstract author(id: number): Author | Promise<Author>;
}

```

By generating classes (instead of the default technique of generating interfaces), you can use declarative validation **decorators** in combination with the schema first approach, which is an extremely useful technique (read [more](#)). For example, you could add `class-validator` decorators to the generated `CreatePostInput` class as shown below to enforce minimum and maximum string lengths on the `title` field:

```
import { MinLength, MaxLength } from 'class-validator';

export class CreatePostInput {
  @MinLength(3)
  @MaxLength(50)
  title: string;
}
```

warning **Notice** To enable auto-validation of your inputs (and parameters), use [ValidationPipe](#).  
Read more about validation [here](#) and more specifically about pipes [here](#).

However, if you add decorators directly to the automatically generated file, they will be **overwritten** each time the file is generated. Instead, create a separate file and simply extend the generated class.

```
import { MinLength, MaxLength } from 'class-validator';
import { Post } from '../../graphql.ts';

export class CreatePostInput extends Post {
  @MinLength(3)
  @MaxLength(50)
  title: string;
}
```

## GraphQL argument decorators

We can access the standard GraphQL resolver arguments using dedicated decorators. Below is a comparison of the Nest decorators and the plain Apollo parameters they represent.

<code>@Root()</code> and <code>@Parent()</code>	<code>root/parent</code>
<code>@Context(param?: string)</code>	<code>context/context[param]</code>
<code>@Info(param?: string)</code>	<code>info/info[param]</code>
<code>@Args(param?: string)</code>	<code>args/args[param]</code>

These arguments have the following meanings:

- `root`: an object that contains the result returned from the resolver on the parent field, or, in the case of a top-level `Query` field, the `rootValue` passed from the server configuration.
- `context`: an object shared by all resolvers in a particular query; typically used to contain per-request state.
- `info`: an object that contains information about the execution state of the query.
- `args`: an object with the arguments passed into the field in the query.

## Module

Once we're done with the above steps, we have declaratively specified all the information needed by the [GraphQLModule](#) to generate a resolver map. The [GraphQLModule](#) uses reflection to introspect the meta data provided via the decorators, and transforms classes into the correct resolver map automatically.

The only other thing you need to take care of is to **provide** (i.e., list as a [provider](#) in some module) the resolver class(es) ([AuthorsResolver](#)), and importing the module ([AuthorsModule](#)) somewhere, so Nest will be able to utilize it.

For example, we can do this in an [AuthorsModule](#), which can also provide other services needed in this context. Be sure to import [AuthorsModule](#) somewhere (e.g., in the root module, or some other module imported by the root module).

```
@@filename(authors/authors.module)
@Module({
  imports: [PostsModule],
  providers: [AuthorsService, AuthorsResolver],
})
export class AuthorsModule {}
```

**info Hint** It is helpful to organize your code by your so-called **domain model** (similar to the way you would organize entry points in a REST API). In this approach, keep your models ([ObjectType](#) classes), resolvers and services together within a Nest module representing the domain model. Keep all of these components in a single folder per module. When you do this, and use the [Nest CLI](#) to generate each element, Nest will wire all of these parts together (locating files in appropriate folders, generating entries in [provider](#) and [imports](#) arrays, etc.) automatically for you.

## Mutations

Most discussions of GraphQL focus on data fetching, but any complete data platform needs a way to modify server-side data as well. In REST, any request could end up causing side-effects on the server, but best practice suggests we should not modify data in GET requests. GraphQL is similar – technically any query could be implemented to cause a data write. However, like REST, it's recommended to observe the convention that any operations that cause writes should be sent explicitly via a mutation (read more [here](#)).

The official [Apollo](#) documentation uses an `upvotePost()` mutation example. This mutation implements a method to increase a post's `votes` property value. To create an equivalent mutation in Nest, we'll make use of the `@Mutation()` decorator.

### Code first

Let's add another method to the `AuthorResolver` used in the previous section (see [resolvers](#)).

```
@Mutation(returns => Post)
async upvotePost(@Args({ name: 'postId', type: () => Int }) postId:
number) {
  return this.postsService.upvoteById({ id: postId });
}
```

**info Hint** All decorators (e.g., `@Resolver`, `@ResolveField`, `@Args`, etc.) are exported from the `@nestjs/graphql` package.

This will result in generating the following part of the GraphQL schema in SDL:

```
type Mutation {
  upvotePost(postId: Int!): Post
}
```

The `upvotePost()` method takes `postId` (`Int`) as an argument and returns an updated `Post` entity. For the reasons explained in the [resolvers](#) section, we have to explicitly set the expected type.

If the mutation needs to take an object as an argument, we can create an **input type**. The input type is a special kind of object type that can be passed in as an argument (read more [here](#)). To declare an input type, use the `@InputType()` decorator.

```
import { InputType, Field } from '@nestjs/graphql';

@InputType()
export class UpvotePostInput {
  @Field()
  postId: number;
}
```

**info Hint** The `@InputType()` decorator takes an options object as an argument, so you can, for example, specify the input type's description. Note that, due to TypeScript's metadata reflection system limitations, you must either use the `@Field` decorator to manually indicate a type, or use a [CLI plugin](#).

We can then use this type in the resolver class:

```
@Mutation(returns => Post)
async upvotePost(
  @Args('upvotePostData') upvotePostData: UpvotePostInput,
) {}
```

## Schema first

Let's extend our `AuthorResolver` used in the previous section (see [resolvers](#)).

```
@Mutation()
async upvotePost(@Args('postId') postId: number) {
  return this.postsService.upvoteById({ id: postId });
}
```

Note that we assumed above that the business logic has been moved to the `PostsService` (querying the post and incrementing its `votes` property). The logic inside the `PostsService` class can be as simple or sophisticated as needed. The main point of this example is to show how resolvers can interact with other providers.

The last step is to add our mutation to the existing types definition.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Mutation {
```

```
    upvotePost(postId: Int!): Post  
}
```

The `upvotePost(postId: Int!): Post` mutation is now available to be called as part of our application's GraphQL API.

## Subscriptions

In addition to fetching data using queries and modifying data using mutations, the GraphQL spec supports a third operation type, called [subscription](#). GraphQL subscriptions are a way to push data from the server to the clients that choose to listen to real time messages from the server. Subscriptions are similar to queries in that they specify a set of fields to be delivered to the client, but instead of immediately returning a single answer, a channel is opened and a result is sent to the client every time a particular event happens on the server.

A common use case for subscriptions is notifying the client side about particular events, for example the creation of a new object, updated fields and so on ([read more here](#)).

### Enable subscriptions with Apollo driver

To enable subscriptions, set the [installSubscriptionHandlers](#) property to `true`.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  installSubscriptionHandlers: true,
}),
```

**warning** **Warning** The [installSubscriptionHandlers](#) configuration option has been removed from the latest version of Apollo server and will be soon deprecated in this package as well. By default, [installSubscriptionHandlers](#) will fallback to use the [subscriptions-transport-ws](#) ([read more](#)) but we strongly recommend using the [graphql-ws](#) ([read more](#)) library instead.

To switch to use the [graphql-ws](#) package instead, use the following configuration:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'graphql-ws': true
  },
}),
```

**info** **Hint** You can also use both packages ([subscriptions-transport-ws](#) and [graphql-ws](#)) at the same time, for example, for backward compatibility.

### Code first

To create a subscription using the code first approach, we use the [@Subscription\(\)](#) decorator (exported from the [@nestjs/graphql](#) package) and the [PubSub](#) class from the [graphql-subscriptions](#) package, which provides a simple [publish/subscribe API](#).

The following subscription handler takes care of **subscribing** to an event by calling [PubSub#asyncIterator](#). This method takes a single argument, the [triggerName](#), which corresponds to

an event topic name.

```
const pubSub = new PubSub();

@Resolver((of) => Author)
export class AuthorResolver {
  // ...
  @Subscription((returns) => Comment)
  commentAdded() {
    return pubSub.asyncIterator('commentAdded');
  }
}
```

**info Hint** All decorators are exported from the `@nestjs/graphql` package, while the `PubSub` class is exported from the `graphql-subscriptions` package.

**warning Note** `PubSub` is a class that exposes a simple `publish` and `subscribe` API. Read more about it [here](#). Note that the Apollo docs warn that the default implementation is not suitable for production (read more [here](#)). Production apps should use a `PubSub` implementation backed by an external store (read more [here](#)).

This will result in generating the following part of the GraphQL schema in SDL:

```
type Subscription {
  commentAdded(): Comment!
}
```

Note that subscriptions, by definition, return an object with a single top level property whose key is the name of the subscription. This name is either inherited from the name of the subscription handler method (i.e., `commentAdded` above), or is provided explicitly by passing an option with the key `name` as the second argument to the `@Subscription()` decorator, as shown below.

```
@Subscription(returns => Comment, {
  name: 'commentAdded',
})
subscribeToCommentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

This construct produces the same SDL as the previous code sample, but allows us to decouple the method name from the subscription.

## Publishing

Now, to publish the event, we use the `PubSub#publish` method. This is often used within a mutation to trigger a client-side update when a part of the object graph has changed. For example:

```

@@filename(posts/posts.resolver)
@Mutation(returns => Post)
async addComment(
  @Args('postId', { type: () => Int }) postId: number,
  @Args('comment', { type: () => Comment }) comment: CommentInput,
) {
  const newComment = this.commentsService.addComment({ id: postId, comment });
  pubSub.publish('commentAdded', { commentAdded: newComment });
  return newComment;
}

```

The `PubSub#publish` method takes a `triggerName` (again, think of this as an event topic name) as the first parameter, and an event payload as the second parameter. As mentioned, the subscription, by definition, returns a value and that value has a shape. Look again at the generated SDL for our `commentAdded` subscription:

```

type Subscription {
  commentAdded(): Comment!
}

```

This tells us that the subscription must return an object with a top-level property name of `commentAdded` that has a value which is a `Comment` object. The important point to note is that the shape of the event payload emitted by the `PubSub#publish` method must correspond to the shape of the value expected to return from the subscription. So, in our example above, the `pubSub.publish('commentAdded', {{ '}} commentAdded: newComment {{ '}} )` statement publishes a `commentAdded` event with the appropriately shaped payload. If these shapes don't match, your subscription will fail during the GraphQL validation phase.

## Filtering subscriptions

To filter out specific events, set the `filter` property to a filter function. This function acts similar to the function passed to an array `filter`. It takes two arguments: `payload` containing the event payload (as sent by the event publisher), and `variables` taking any arguments passed in during the subscription request. It returns a boolean determining whether this event should be published to client listeners.

```

@Subscription(returns => Comment, {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded(@Args('title') title: string) {
  return pubSub.asyncIterator('commentAdded');
}

```

## Mutating subscription payloads

To mutate the published event payload, set the `resolve` property to a function. The function receives the event payload (as sent by the event publisher) and returns the appropriate value.

```
@Subscription(returns => Comment, {
  resolve: value => value,
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

**warning Note** If you use the `resolve` option, you should return the unwrapped payload (e.g., with our example, return a `newComment` object directly, not a `{} {{ }} commentAdded: newComment {{ }} object`).

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction.

```
@Subscription(returns => Comment, {
  resolve(this: AuthorResolver, value) {
    // "this" refers to an instance of "AuthorResolver"
    return value;
  }
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

The same construction works with filters:

```
@Subscription(returns => Comment, {
  filter(this: AuthorResolver, payload, variables) {
    // "this" refers to an instance of "AuthorResolver"
    return payload.commentAdded.title === variables.title;
  }
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

## Schema first

To create an equivalent subscription in Nest, we'll make use of the `@Subscription()` decorator.

```
const pubSub = new PubSub();
```

```
@Resolver('Author')
export class AuthorResolver {
    // ...
    @Subscription()
    commentAdded() {
        return pubSub.asyncIterator('commentAdded');
    }
}
```

To filter out specific events based on context and arguments, set the `filter` property.

```
@Subscription('commentAdded', {
    filter: (payload, variables) =>
        payload.commentAdded.title === variables.title,
})
commentAdded() {
    return pubSub.asyncIterator('commentAdded');
}
```

To mutate the published payload, we can use a `resolve` function.

```
@Subscription('commentAdded', {
    resolve: value => value,
})
commentAdded() {
    return pubSub.asyncIterator('commentAdded');
}
```

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction:

```
@Subscription('commentAdded', {
    resolve(this: AuthorResolver, value) {
        // "this" refers to an instance of "AuthorResolver"
        return value;
    }
})
commentAdded() {
    return pubSub.asyncIterator('commentAdded');
}
```

The same construction works with filters:

```
@Subscription('commentAdded', {
    filter(this: AuthorResolver, payload, variables) {
```

```
// "this" refers to an instance of "AuthorResolver"
  return payload.commentAdded.title === variables.title;
}
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

The last step is to update the type definitions file.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Comment {
  id: String
  content: String
}

type Subscription {
  commentAdded(title: String!): Comment
}
```

With this, we've created a single `commentAdded(title: String!): Comment` subscription. You can find a full sample implementation [here](#).

## PubSub

We instantiated a local `PubSub` instance above. The preferred approach is to define `PubSub` as a provider and inject it through the constructor (using the `@Inject()` decorator). This allows us to re-use the instance across the whole application. For example, define a provider as follows, then inject '`'PUB_SUB'` where needed.

```
{
  provide: 'PUB_SUB',
```

```
    useValue: new PubSub(),
}
```

## Customize subscriptions server

To customize the subscriptions server (e.g., change the path), use the `subscriptions` options property.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'subscriptions-transport-ws': {
      path: '/graphql'
    },
  }
}),
```

If you're using the `graphql-ws` package for subscriptions, replace the `subscriptions-transport-ws` key with `graphql-ws`, as follows:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'graphql-ws': {
      path: '/graphql'
    },
  }
}),
```

## Authentication over WebSockets

Checking whether the user is authenticated can be done inside the `onConnect` callback function that you can specify in the `subscriptions` options.

The `onConnect` will receive as a first argument the `connectionParams` passed to the `SubscriptionClient` (read [more](#)).

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'subscriptions-transport-ws': {
      onConnect: (connectionParams) => {
        const authToken = connectionParams.authToken;
        if (!isValid(authToken)) {
          throw new Error('Token is not valid');
        }
        // extract user information from token
    }
  }
}),
```

```

        const user = parseToken(authToken);
        // return user info to add them to the context later
        return { user };
    },
}
},
context: ({ connection }) => {
    // connection.context will be equal to what was returned by the
    "onConnect" callback
},
}),

```

The `authToken` in this example is only sent once by the client, when the connection is first established. All subscriptions made with this connection will have the same `authToken`, and thus the same user info.

**warning Note** There is a bug in `subscriptions-transport-ws` that allows connections to skip the `onConnect` phase (read [more](#)). You should not assume that `onConnect` was called when the user starts a subscription, and always check that the `context` is populated.

If you're using the `graphql-ws` package, the signature of the `onConnect` callback will be slightly different:

```

GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'graphql-ws': {
      onConnect: (context: Context<any>) => {
        const { connectionParams, extra } = context;
        // user validation will remain the same as in the example above
        // when using with graphql-ws, additional context value should be
        stored in the extra field
        extra.user = { user: {} };
      },
    },
    context: ({ extra }) => {
      // you can now access your additional context value through the extra
      field
    },
  });
});

```

## Enable subscriptions with Mercurius driver

To enable subscriptions, set the `subscription` property to `true`.

```

GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  subscription: true,
}),

```

**info Hint** You can also pass the options object to set up a custom emitter, validate incoming connections, etc. Read more [here](#) (see [subscription](#)).

## Code first

To create a subscription using the code first approach, we use the [@Subscription\(\)](#) decorator (exported from the [@nestjs/graphql](#) package) and the [PubSub](#) class from the [mercurius](#) package, which provides a simple **publish/subscribe API**.

The following subscription handler takes care of **subscribing** to an event by calling [PubSub#asyncIterator](#). This method takes a single argument, the [triggerName](#), which corresponds to an event topic name.

```
@Resolver((of) => Author)
export class AuthorResolver {
  // ...
  @Subscription((returns) => Comment)
  commentAdded(@Context('pubsub') pubSub: PubSub) {
    return pubSub.subscribe('commentAdded');
  }
}
```

**info Hint** All decorators used in the example above are exported from the [@nestjs/graphql](#) package, while the [PubSub](#) class is exported from the [mercurius](#) package.

**warning Note** [PubSub](#) is a class that exposes a simple [publish](#) and [subscribe](#) API. Check out [this section](#) on how to register a custom [PubSub](#) class.

This will result in generating the following part of the GraphQL schema in SDL:

```
type Subscription {
  commentAdded(): Comment!
}
```

Note that subscriptions, by definition, return an object with a single top level property whose key is the name of the subscription. This name is either inherited from the name of the subscription handler method (i.e., [commentAdded](#) above), or is provided explicitly by passing an option with the key [name](#) as the second argument to the [@Subscription\(\)](#) decorator, as shown below.

```
@Subscription(returns => Comment, {
  name: 'commentAdded',
})
subscribeToCommentAdded(@Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

This construct produces the same SDL as the previous code sample, but allows us to decouple the method name from the subscription.

## Publishing

Now, to publish the event, we use the `PubSub#publish` method. This is often used within a mutation to trigger a client-side update when a part of the object graph has changed. For example:

```
@@filename(posts/posts.resolver)
@Mutation(returns => Post)
async addComment(
  @Args('postId', { type: () => Int }) postId: number,
  @Args('comment', { type: () => Comment }) comment: CommentInput,
  @Context('pubsub') pubSub: PubSub,
) {
  const newComment = this.commentsService.addComment({ id: postId, comment });
  await pubSub.publish({
    topic: 'commentAdded',
    payload: {
      commentAdded: newComment
    }
  });
  return newComment;
}
```

As mentioned, the subscription, by definition, returns a value and that value has a shape. Look again at the generated SDL for our `commentAdded` subscription:

```
type Subscription {
  commentAdded(): Comment!
}
```

This tells us that the subscription must return an object with a top-level property name of `commentAdded` that has a value which is a `Comment` object. The important point to note is that the shape of the event payload emitted by the `PubSub#publish` method must correspond to the shape of the value expected to return from the subscription. So, in our example above, the `pubSub.publish({{ '{' }} topic: 'commentAdded', payload: {{ '{' }} commentAdded: newComment {{ '}' }} {{ '}' }})` statement publishes a `commentAdded` event with the appropriately shaped payload. If these shapes don't match, your subscription will fail during the GraphQL validation phase.

## Filtering subscriptions

To filter out specific events, set the `filter` property to a filter function. This function acts similar to the function passed to an array `filter`. It takes two arguments: `payload` containing the event payload (as sent by the event publisher), and `variables` taking any arguments passed in during the subscription request. It returns a boolean determining whether this event should be published to client listeners.

```
@Subscription(returns => Comment, {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded(@Args('title') title: string, @Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction.

```
@Subscription(returns => Comment, {
  filter(this: AuthorResolver, payload, variables) {
    // "this" refers to an instance of "AuthorResolver"
    return payload.commentAdded.title === variables.title;
  }
})
commentAdded(@Args('title') title: string, @Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

## Schema first

To create an equivalent subscription in Nest, we'll make use of the `@Subscription()` decorator.

```
const pubSub = new PubSub();

@Resolver('Author')
export class AuthorResolver {
  // ...
  @Subscription()
  commentAdded(@Context('pubsub') pubSub: PubSub) {
    return pubSub.subscribe('commentAdded');
  }
}
```

To filter out specific events based on context and arguments, set the `filter` property.

```
@Subscription('commentAdded', {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded(@Context('pubsub') pubSub: PubSub) {
```

```
    return pubSub.subscribe('commentAdded');
}
```

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction:

```
@Subscription('commentAdded', {
  filter(this: AuthorResolver, payload, variables) {
    // "this" refers to an instance of "AuthorResolver"
    return payload.commentAdded.title === variables.title;
  }
})
commentAdded(@Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

The last step is to update the type definitions file.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Comment {
  id: String
  content: String
}

type Subscription {
  commentAdded(title: String!): Comment
}
```

With this, we've created a single `commentAdded(title: String!): Comment` subscription.

## PubSub

In the examples above, we used the default **PubSub** emitter (`mqemitter`) The preferred approach (for production) is to use `mqemitter-redis`. Alternatively, a custom **PubSub** implementation can be provided (read more [here](#))

```
GraphQLModule.forRoot<MercuriusDriverConfig>({  
  driver: MercuriusDriver,  
  subscription: {  
    emitter: require('mqemitter-redis')({  
      port: 6579,  
      host: '127.0.0.1',  
    }),  
  },  
});
```

## Authentication over WebSockets

Checking whether the user is authenticated can be done inside the `verifyClient` callback function that you can specify in the `subscription` options.

The `verifyClient` will receive the `info` object as a first argument which you can use to retrieve the request's headers.

```
GraphQLModule.forRoot<MercuriusDriverConfig>({  
  driver: MercuriusDriver,  
  subscription: {  
    verifyClient: (info, next) => {  
      const authorization = info.req.headers?.authorization as string;  
      if (!authorization?.startsWith('Bearer ')) {  
        return next(false);  
      }  
      next(true);  
    },  
  },  
});
```

## Scalars

A GraphQL object type has a name and fields, but at some point those fields have to resolve to some concrete data. That's where the scalar types come in: they represent the leaves of the query (read more [here](#)). GraphQL includes the following default types: `Int`, `Float`, `String`, `Boolean` and `ID`. In addition to these built-in types, you may need to support custom atomic data types (e.g., `Date`).

### Code first

The code-first approach ships with five scalars in which three of them are simple aliases for the existing GraphQL types.

- `ID` (alias for `GraphQLID`) - represents a unique identifier, often used to refetch an object or as the key for a cache
- `Int` (alias for `GraphQLInt`) - a signed 32-bit integer
- `Float` (alias for `GraphQLFloat`) - a signed double-precision floating-point value
- `GraphQLISODateTime` - a date-time string at UTC (used by default to represent `Date` type)
- `GraphQLTimestamp` - a signed integer which represents date and time as number of milliseconds from start of UNIX epoch

The `GraphQLISODateTime` (e.g. `2019-12-03T09:54:33Z`) is used by default to represent the `Date` type. To use the `GraphQLTimestamp` instead, set the `dateScalarMode` of the `buildSchemaOptions` object to `'timestamp'` as follows:

```
GraphQLModule.forRoot({
  buildSchemaOptions: {
    dateScalarMode: 'timestamp',
  }
}),
```

Likewise, the `GraphQLFloat` is used by default to represent the `number` type. To use the `GraphQLInt` instead, set the `numberScalarMode` of the `buildSchemaOptions` object to `'integer'` as follows:

```
GraphQLModule.forRoot({
  buildSchemaOptions: {
    numberScalarMode: 'integer',
  }
}),
```

In addition, you can create custom scalars.

### Override a default scalar

To create a custom implementation for the `Date` scalar, simply create a new class.

```

import { Scalar, CustomScalar } from '@nestjs/graphql';
import { Kind, ValueNode } from 'graphql';

@Scalar('Date', (type) => Date)
export class DateScalar implements CustomScalar<number, Date> {
  description = 'Date custom scalar type';

  parseValue(value: number): Date {
    return new Date(value); // value from the client
  }

  serialize(value: Date): number {
    return value.getTime(); // value sent to the client
  }

  parseLiteral(ast: ValueNode): Date {
    if (ast.kind === Kind.INT) {
      return new Date(ast.value);
    }
    return null;
  }
}

```

With this in place, register `DateScalar` as a provider.

```

@Module({
  providers: [DateScalar],
})
export class CommonModule {}

```

Now we can use the `Date` type in our classes.

```

@Field()
creationDate: Date;

```

## Import a custom scalar

To use a custom scalar, import and register it as a resolver. We'll use the `graphql-type-json` package for demonstration purposes. This npm package defines a `JSON` GraphQL scalar type.

Start by installing the package:

```
$ npm i --save graphql-type-json
```

Once the package is installed, we pass a custom resolver to the `forRoot()` method:

```
import GraphQLJSON from 'graphql-type-json';

@Module({
  imports: [
    GraphQLModule.forRoot({
      resolvers: { JSON: GraphQLJSON },
    }),
  ],
})
export class AppModule {}
```

Now we can use the `JSON` type in our classes.

```
@Field((type) => GraphQLJSON)
info: JSON;
```

For a suite of useful scalars, take a look at the [graphql-scalars](#) package.

## Create a custom scalar

To define a custom scalar, create a new `GraphQLScalarType` instance. We'll create a custom `UUID` scalar.

```
const regex = /^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}$/i;

function validate(uuid: unknown): string | never {
  if (typeof uuid !== "string" || !regex.test(uuid)) {
    throw new Error("invalid uuid");
  }
  return uuid;
}

export const CustomUuidScalar = new GraphQLScalarType({
  name: 'UUID',
  description: 'A simple UUID parser',
  serialize: (value) => validate(value),
  parseValue: (value) => validate(value),
  parseLiteral: (ast) => validate(ast.value)
})
```

We pass a custom resolver to the `forRoot()` method:

```
@Module({
  imports: [
    GraphQLModule.forRoot({
      resolvers: { UUID: CustomUuidScalar },
    })
  ],
})
```

```

    },
],
})
export class AppModule {}

```

Now we can use the **UUID** type in our classes.

```

@Field((type) => CustomUuidScalar)
uuid: string;

```

## Schema first

To define a custom scalar (read more about scalars [here](#)), create a type definition and a dedicated resolver. Here (as in the official documentation), we'll use the **graphql-type-json** package for demonstration purposes. This npm package defines a **JSON** GraphQL scalar type.

Start by installing the package:

```
$ npm i --save graphql-type-json
```

Once the package is installed, we pass a custom resolver to the **forRoot()** method:

```

import GraphQLJSON from 'graphql-type-json';

@Module({
  imports: [
    GraphQLModule.forRoot({
      typePaths: ['./**/*.graphql'],
      resolvers: { JSON: GraphQLJSON },
    }),
  ],
})
export class AppModule {}

```

Now we can use the **JSON** scalar in our type definitions:

```

scalar JSON

type Foo {
  field: JSON
}

```

Another method to define a scalar type is to create a simple class. Assume we want to enhance our schema with the `Date` type.

```
import { Scalar, CustomScalar } from '@nestjs/graphql';
import { Kind, ValueNode } from 'graphql';

@Scalar('Date')
export class DateScalar implements CustomScalar<number, Date> {
  description = 'Date custom scalar type';

  parseValue(value: number): Date {
    return new Date(value); // value from the client
  }

  serialize(value: Date): number {
    return value.getTime(); // value sent to the client
  }

  parseLiteral(ast: ValueNode): Date {
    if (ast.kind === Kind.INT) {
      return new Date(ast.value);
    }
    return null;
  }
}
```

With this in place, register `DateScalar` as a provider.

```
@Module({
  providers: [DateScalar],
})
export class CommonModule {}
```

Now we can use the `Date` scalar in type definitions.

```
scalar Date
```

By default, the generated TypeScript definition for all scalars is `any` - which isn't particularly typesafe. But, you can configure how Nest generates typings for your custom scalars when you specify how to generate types:

```
import { GraphQLDefinitionsFactory } from '@nestjs/graphql';
import { join } from 'path';

const definitionsFactory = new GraphQLDefinitionsFactory();
```

```
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  path: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
  defaultScalarType: 'unknown',
  customScalarTypeMapping: {
    DateTime: 'Date',
    BigNumber: '_BigNumber',
  },
  additionalHeader: "import _BigNumber from 'bignumber.js''",
});
```

**info Hint** Alternatively, you can use a type reference instead, for example: `DateTime: Date`. In this case, `GraphQLDefinitionsFactory` will extract the name property of the specified type (`Date.name`) to generate TS definitions. Note: adding an import statement for non-built-in types (custom types) is required.

Now, given the following GraphQL custom scalar types:

```
scalar DateTime
scalar BigNumber
scalar Payload
```

We will now see the following generated TypeScript definitions in `src/graphql.ts`:

```
import _BigNumber from 'bignumber.js';

export type DateTime = Date;
export type BigNumber = _BigNumber;
export type Payload = unknown;
```

Here, we've used the `customScalarTypeMapping` property to supply a map of the types we wish to declare for our custom scalars. We've also provided an `additionalHeader` property so that we can add any imports required for these type definitions. Lastly, we've added a `defaultScalarType` of `'unknown'`, so that any custom scalars not specified in `customScalarTypeMapping` will be aliased to `unknown` instead of `any` (which [TypeScript recommends](#) using since 3.0 for added type safety).

**info Hint** Note that we've imported `_BigNumber` from `bignumber.js`; this is to avoid [circular type references](#).

## Directives

A directive can be attached to a field or fragment inclusion, and can affect execution of the query in any way the server desires (read more [here](#)). The GraphQL specification provides several default directives:

- `@include(if: Boolean)` - only include this field in the result if the argument is true
- `@skip(if: Boolean)` - skip this field if the argument is true
- `@deprecated(reason: String)` - marks field as deprecated with message

A directive is an identifier preceded by a `@` character, optionally followed by a list of named arguments, which can appear after almost any element in the GraphQL query and schema languages.

### Custom directives

To instruct what should happen when Apollo/Mercurius encounters your directive, you can create a transformer function. This function uses the `mapSchema` function to iterate through locations in your schema (field definitions, type definitions, etc.) and perform corresponding transformations.

```
import { getDirective, MapperKind, mapSchema } from '@graphql-tools/utils';
import { defaultFieldResolver, GraphQLSchema } from 'graphql';

export function upperDirectiveTransformer(
  schema: GraphQLSchema,
  directiveName: string,
) {
  return mapSchema(schema, {
    [MapperKind.OBJECT_FIELD]: (fieldConfig) => {
      const upperDirective = getDirective(
        schema,
        fieldConfig,
        directiveName,
      )?.[0];

      if (upperDirective) {
        const { resolve = defaultFieldResolver } = fieldConfig;

        // Replace the original resolver with a function that *first* calls
        // the original resolver, then converts its result to upper case
        fieldConfig.resolve = async function (source, args, context, info) {
          const result = await resolve(source, args, context, info);
          if (typeof result === 'string') {
            return result.toUpperCase();
          }
          return result;
        };
      }
    },
  });
}
```

```
    });
}
```

Now, apply the `upperDirectiveTransformer` transformation function in the `GraphQLModule#forRoot` method using the `transformSchema` function:

```
GraphQLModule.forRoot({
  // ...
  transformSchema: (schema) => upperDirectiveTransformer(schema, 'upper'),
});
```

Once registered, the `@upper` directive can be used in our schema. However, the way you apply the directive will vary depending on the approach you use (code first or schema first).

## Code first

In the code first approach, use the `@Directive()` decorator to apply the directive.

```
@Directive('@upper')
@Field()
title: string;
```

**info Hint** The `@Directive()` decorator is exported from the `@nestjs/graphql` package.

Directives can be applied on fields, field resolvers, input and object types, as well as queries, mutations, and subscriptions. Here's an example of the directive applied on the query handler level:

```
@Directive('@deprecated(reason: "This query will be removed in the next
version")')
@Query(returns => Author, { name: 'author' })
async getAuthor(@Args({ name: 'id', type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

**warn Warning** Directives applied through the `@Directive()` decorator will not be reflected in the generated schema definition file.

Lastly, make sure to declare directives in the `GraphQLModule`, as follows:

```
GraphQLModule.forRoot({
  // ...
  transformSchema: schema => upperDirectiveTransformer(schema, 'upper'),
  buildSchemaOptions: {
    directives: [
      ...
```

```
new GraphQLDirective({  
  name: 'upper',  
  locations: [DirectiveLocation.FIELD_DEFINITION],  
},  
],  
},  
),
```

info **Hint** Both `GraphQLDirective` and `DirectiveLocation` are exported from the `graphql` package.

## Schema first

In the schema first approach, apply directives directly in SDL.

```
directive @upper on FIELD_DEFINITION  
  
type Post {  
  id: Int!  
  title: String! @upper  
  votes: Int  
}
```

## Interfaces

Like many type systems, GraphQL supports interfaces. An **Interface** is an abstract type that includes a certain set of fields that a type must include to implement the interface (read more [here](#)).

### Code first

When using the code first approach, you define a GraphQL interface by creating an abstract class annotated with the `@InterfaceType()` decorator exported from the `@nestjs/graphql`.

```
import { Field, ID, InterfaceType } from '@nestjs/graphql';

@InterfaceType()
export abstract class Character {
  @Field(() => ID)
  id: string;

  @Field()
  name: string;
}
```

**warning** **Warning** TypeScript interfaces cannot be used to define GraphQL interfaces.

This will result in generating the following part of the GraphQL schema in SDL:

```
interface Character {
  id: ID!
  name: String!
}
```

Now, to implement the `Character` interface, use the `implements` key:

```
@ObjectType({
  implements: () => [Character],
})
export class Human implements Character {
  id: string;
  name: string;
}
```

**info** **Hint** The `@ObjectType()` decorator is exported from the `@nestjs/graphql` package.

The default `resolveType()` function generated by the library extracts the type based on the value returned from the resolver method. This means that you must return class instances (you cannot return literal JavaScript objects).

To provide a customized `resolveType()` function, pass the `resolveType` property to the options object passed into the `@InterfaceType()` decorator, as follows:

```
@InterfaceType({
  resolveType(book) {
    if (book.colors) {
      return ColoringBook;
    }
    return TextBook;
  },
})
export abstract class Book {
  @Field((type) => ID)
  id: string;

  @Field()
  title: string;
}
```

## Interface resolvers

So far, using interfaces, you could only share field definitions with your objects. If you also want to share the actual field resolvers implementation, you can create a dedicated interface resolver, as follows:

```
import { Resolver, ResolveField, Parent, Info } from '@nestjs/graphql';

@Resolver(type => Character) // Reminder: Character is an interface
export class CharacterInterfaceResolver {
  @ResolveField(() => [Character])
  friends(
    @Parent() character, // Resolved object that implements Character
    @Info() { parentType }, // Type of the object that implements
    Character
    @Args('search', { type: () => String }) searchTerm: string,
  ) {
    // Get character's friends
    return [];
  }
}
```

Now the `friends` field resolver is auto-registered for all object types that implement the `Character` interface.

## Schema first

To define an interface in the schema first approach, simply create a GraphQL interface with SDL.

```
interface Character {  
  id: ID!  
  name: String!  
}
```

Then, you can use the typings generation feature (as shown in the [quick start](#) chapter) to generate corresponding TypeScript definitions:

```
export interface Character {  
  id: string;  
  name: string;  
}
```

Interfaces require an extra `__resolveType` field in the resolver map to determine which type the interface should resolve to. Let's create a `CharactersResolver` class and define the `__resolveType` method:

```
@Resolver('Character')  
export class CharactersResolver {  
  @ResolveField()  
  __resolveType(value) {  
    if ('age' in value) {  
      return Person;  
    }  
    return null;  
  }  
}
```

**info Hint** All decorators are exported from the `@nestjs/graphql` package.

## Unions

Union types are very similar to interfaces, but they don't get to specify any common fields between the types (read more [here](#)). Unions are useful for returning disjoint data types from a single field.

### Code first

To define a GraphQL union type, we must define classes that this union will be composed of. Following the example from the Apollo documentation, we'll create two classes. First, **Book**:

```
import { Field, ObjectType } from '@nestjs/graphql';

@ObjectType()
export class Book {
  @Field()
  title: string;
}
```

And then **Author**:

```
import { Field, ObjectType } from '@nestjs/graphql';

@ObjectType()
export class Author {
  @Field()
  name: string;
}
```

With this in place, register the **ResultUnion** union using the **createUnionType** function exported from the **@nestjs/graphql** package:

```
export const ResultUnion = createUnionType({
  name: 'ResultUnion',
  types: () => [Author, Book] as const,
});
```

**warning Warning** The array returned by the **types** property of the **createUnionType** function should be given a const assertion. If the const assertion is not given, a wrong declaration file will be generated at compile time, and an error will occur when using it from another project.

Now, we can reference the **ResultUnion** in our query:

```
@Query(returns => [ResultUnion])
search(): Array<typeof ResultUnion> {
```

```
    return [new Author(), new Book()];
}
```

This will result in generating the following part of the GraphQL schema in SDL:

```
type Author {
  name: String!
}

type Book {
  title: String!
}

union ResultUnion = Author | Book

type Query {
  search: [ResultUnion!]!
}
```

The default `resolveType()` function generated by the library will extract the type based on the value returned from the resolver method. That means returning class instances instead of literal JavaScript object is obligatory.

To provide a customized `resolveType()` function, pass the `resolveType` property to the options object passed into the `createUnionType()` function, as follows:

```
export const ResultUnion = createUnionType({
  name: 'ResultUnion',
  types: () => [Author, Book] as const,
  resolveType(value) {
    if (value.name) {
      return Author;
    }
    if (value.title) {
      return Book;
    }
    return null;
  },
});
```

## Schema first

To define a union in the schema first approach, simply create a GraphQL union with SDL.

```
type Author {
  name: String!
```

```

}

type Book {
  title: String!
}

union ResultUnion = Author | Book

```

Then, you can use the typings generation feature (as shown in the [quick start](#) chapter) to generate corresponding TypeScript definitions:

```

export class Author {
  name: string;
}

export class Book {
  title: string;
}

export type ResultUnion = Author | Book;

```

Unions require an extra `__resolveType` field in the resolver map to determine which type the union should resolve to. Also, note that the `ResultUnionResolver` class has to be registered as a provider in any module. Let's create a `ResultUnionResolver` class and define the `__resolveType` method.

```

@Resolver('ResultUnion')
export class ResultUnionResolver {
  @ResolveField()
  __resolveType(value) {
    if (value.name) {
      return 'Author';
    }
    if (value.title) {
      return 'Book';
    }
    return null;
  }
}

```

**info Hint** All decorators are exported from the `@nestjs/graphql` package.

## Enums

Enumeration types are a special kind of scalar that is restricted to a particular set of allowed values (read more [here](#)). This allows you to:

- validate that any arguments of this type are one of the allowed values

- communicate through the type system that a field will always be one of a finite set of values

## Code first

When using the code first approach, you define a GraphQL enum type by simply creating a TypeScript enum.

```
export enum AllowedColor {  
  RED,  
  GREEN,  
  BLUE,  
}
```

With this in place, register the `AllowedColor` enum using the `registerEnumType` function exported from the `@nestjs/graphql` package:

```
registerEnumType(AllowedColor, {  
  name: 'AllowedColor',  
});
```

Now you can reference the `AllowedColor` in our types:

```
@Field(type => AllowedColor)  
favoriteColor: AllowedColor;
```

This will result in generating the following part of the GraphQL schema in SDL:

```
enum AllowedColor {  
  RED  
  GREEN  
  BLUE  
}
```

To provide a description for the enum, pass the `description` property into the `registerEnumType()` function.

```
registerEnumType(AllowedColor, {  
  name: 'AllowedColor',  
  description: 'The supported colors.',  
});
```

To provide a description for the enum values, or to mark a value as deprecated, pass the `valuesMap` property, as follows:

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
  description: 'The supported colors.',
  valuesMap: {
    RED: {
      description: 'The default color.',
    },
    BLUE: {
      deprecationReason: 'Too blue.',
    },
  },
});
```

This will generate the following GraphQL schema in SDL:

```
#####
The supported colors.
#####
enum AllowedColor {
  #####
  The default color.
  #####
  RED
  GREEN
  BLUE @deprecated(reason: "Too blue.")
}
```

## Schema first

To define an enumerator in the schema first approach, simply create a GraphQL enum with SDL.

```
enum AllowedColor {
  RED
  GREEN
  BLUE
}
```

Then you can use the typings generation feature (as shown in the [quick start](#) chapter) to generate corresponding TypeScript definitions:

```
export enum AllowedColor {
  RED
```

```
GREEN  
BLUE  
}
```

Sometimes a backend forces a different value for an enum internally than in the public API. In this example the API contains **RED**, however in resolvers we may use **#f00** instead (read more [here](#)). To accomplish this, declare a resolver object for the **AllowedColor** enum:

```
export const allowedColorResolver: Record<keyof typeof AllowedColor, any>  
= {  
  RED: '#f00',  
};
```

**info Hint** All decorators are exported from the [@nestjs/graphql](#) package.

Then use this resolver object together with the **resolvers** property of the **GraphQLModule#forRoot()** method, as follows:

```
GraphQLModule.forRoot({  
  resolvers: {  
    AllowedColor: allowedColorResolver,  
  },  
});
```

## Field middleware

**warning** **Warning** This chapter applies only to the code first approach.

Field Middleware lets you run arbitrary code **before or after** a field is resolved. A field middleware can be used to convert the result of a field, validate the arguments of a field, or even check field-level roles (for example, required to access a target field for which a middleware function is executed).

You can connect multiple middleware functions to a field. In this case, they will be called sequentially along the chain where the previous middleware decides to call the next one. The order of the middleware functions in the `middleware` array is important. The first resolver is the "most-outer" layer, so it gets executed first and last (similarly to the `graphql-middleware` package). The second resolver is the "second-outer" layer, so it gets executed second and second to last.

### Getting started

Let's start off by creating a simple middleware that will log a field value before it's sent back to the client:

```
import { FieldMiddleware, MiddlewareContext, NextFn } from
'@nestjs/graphql';

const loggerMiddleware: FieldMiddleware = async (
  ctx: MiddlewareContext,
  next: NextFn,
) => {
  const value = await next();
  console.log(value);
  return value;
};
```

**info** **Hint** The `MiddlewareContext` is an object that consist of the same arguments that are normally received by the GraphQL resolver function (`{} {} source, args, context, info {} {}`), while `NextFn` is a function that let you execute the next middleware in the stack (bound to this field) or the actual field resolver.

**warning** **Warning** Field middleware functions cannot inject dependencies nor access Nest's DI container as they are designed to be very lightweight and shouldn't perform any potentially time-consuming operations (like retrieving data from the database). If you need to call external services/query data from the data source, you should do it in a guard/interceptor bounded to a root query/mutation handler and assign it to `context` object which you can access from within the field middleware (specifically, from the `MiddlewareContext` object).

Note that field middleware must match the `FieldMiddleware` interface. In the example above, we first run the `next()` function (which executes the actual field resolver and returns a field value) and then, we log this value to our terminal. Also, the value returned from the middleware function completely overrides the previous value and since we don't want to perform any changes, we simply return the original value.

With this in place, we can register our middleware directly in the `@Field()` decorator, as follows:

```
@ObjectType()
export class Recipe {
  @Field({ middleware: [loggerMiddleware] })
  title: string;
}
```

Now whenever we request the `title` field of `Recipe` object type, the original field's value will be logged to the console.

**info Hint** To learn how you can implement a field-level permissions system with the use of [extensions](#) feature, check out this [section](#).

**warning Warning** Field middleware can be applied only to `ObjectType` classes. For more details, check out this [issue](#).

Also, as mentioned above, we can control the field's value from within the middleware function. For demonstration purposes, let's capitalise a recipe's title (if present):

```
const value = await next();
return value?.toUpperCase();
```

In this case, every title will be automatically uppercased, when requested.

Likewise, you can bind a field middleware to a custom field resolver (a method annotated with the `@ResolveField()` decorator), as follows:

```
@ResolveField(() => String, { middleware: [loggerMiddleware] })
title() {
  return 'Placeholder';
}
```

**warning Warning** In case enhancers are enabled at the field resolver level ([read more](#)), field middleware functions will run before any interceptors, guards, etc., **bounded to the method** (but after the root-level enhancers registered for query or mutation handlers).

## Global field middleware

In addition to binding a middleware directly to a specific field, you can also register one or multiple middleware functions globally. In this case, they will be automatically connected to all fields of your object types.

```
GraphQLModule.forRoot({
  autoSchemaFile: 'schema.gql',
  buildSchemaOptions: {
    fieldMiddleware: [loggerMiddleware],
  }
})
```

```
},  
}) ,
```

info **Hint** Globally registered field middleware functions will be executed **before** locally registered ones (those bound directly to specific fields).

## Mapped types

**warning Warning** This chapter applies only to the code first approach.

As you build out features like CRUD (Create/Read/Update/Delete) it's often useful to construct variants on a base entity type. Nest provides several utility functions that perform type transformations to make this task more convenient.

### Partial

When building input validation types (also called Data Transfer Objects or DTOs), it's often useful to build **create** and **update** variations on the same type. For example, the **create** variant may require all fields, while the **update** variant may make all fields optional.

Nest provides the **PartialType()** utility function to make this task easier and minimize boilerplate.

The **PartialType()** function returns a type (class) with all the properties of the input type set to optional. For example, suppose we have a **create** type as follows:

```
@InputType()
class CreateUserInput {
  @Field()
  email: string;

  @Field()
  password: string;

  @Field()
  firstName: string;
}
```

By default, all of these fields are required. To create a type with the same fields, but with each one optional, use **PartialType()** passing the class reference (**CreateUserInput**) as an argument:

```
@InputType()
export class UpdateUserInput extends PartialType(CreateUserInput) {}
```

**info Hint** The **PartialType()** function is imported from the [@nestjs/graphql](#) package.

The **PartialType()** function takes an optional second argument that is a reference to a decorator factory. This argument can be used to change the decorator function applied to the resulting (child) class. If not specified, the child class effectively uses the same decorator as the **parent** class (the class referenced in the first argument). In the example above, we are extending **CreateUserInput** which is annotated with the **@InputType()** decorator. Since we want **UpdateUserInput** to also be treated as if it were decorated with **@InputType()**, we didn't need to pass **InputType** as the second argument. If the parent and child

types are different, (e.g., the parent is decorated with `@ObjectType`), we would pass `InputType` as the second argument. For example:

```
@InputType()
export class UpdateUserInput extends PartialType(User, InputType) {}
```

## Pick

The `PickType()` function constructs a new type (class) by picking a set of properties from an input type. For example, suppose we start with a type like:

```
@InputType()
class CreateUserInput {
  @Field()
  email: string;

  @Field()
  password: string;

  @Field()
  firstName: string;
}
```

We can pick a set of properties from this class using the `PickType()` utility function:

```
@InputType()
export class UpdateEmailInput extends PickType(CreateUserInput, [
  'email',
] as const) {}
```

**info Hint** The `PickType()` function is imported from the `@nestjs/graphql` package.

## Omit

The `OmitType()` function constructs a type by picking all properties from an input type and then removing a particular set of keys. For example, suppose we start with a type like:

```
@InputType()
class CreateUserInput {
  @Field()
  email: string;

  @Field()
  password: string;
```

```
@Field()
firstName: string;
}
```

We can generate a derived type that has every property **except** `email` as shown below. In this construct, the second argument to `OmitType` is an array of property names.

```
@InputType()
export class UpdateUserInput extends OmitType(CreateUserInput, [
  'email',
] as const) {}
```

**info Hint** The `OmitType()` function is imported from the `@nestjs/graphql` package.

## Intersection

The `IntersectionType()` function combines two types into one new type (class). For example, suppose we start with two types like:

```
@InputType()
class CreateUserInput {
  @Field()
  email: string;

  @Field()
  password: string;
}

@ObjectType()
export class AdditionalUserInfo {
  @Field()
  firstName: string;

  @Field()
  lastName: string;
}
```

We can generate a new type that combines all properties in both types.

```
@InputType()
export class UpdateUserInput extends IntersectionType(
  CreateUserInput,
  AdditionalUserInfo,
) {}
```

**info Hint** The `IntersectionType()` function is imported from the `@nestjs/graphql` package.

## Composition

The type mapping utility functions are composable. For example, the following will produce a type (class) that has all of the properties of the `CreateUserInput` type except for `email`, and those properties will be set to optional:

```
@InputType()
export class UpdateUserInput extends PartialType(
  OmitType(CreateUserInput, ['email'] as const),
) {}
```

## Plugins with Apollo

Plugins enable you to extend Apollo Server's core functionality by performing custom operations in response to certain events. Currently, these events correspond to individual phases of the GraphQL request lifecycle, and to the startup of Apollo Server itself (read more [here](#)). For example, a basic logging plugin might log the GraphQL query string associated with each request that's sent to Apollo Server.

### Custom plugins

To create a plugin, declare a class annotated with the `@Plugin` decorator exported from the `@nestjs/apollo` package. Also, for better code autocompletion, implement the `ApolloServerPlugin` interface from the `@apollo/server` package.

```
import { ApolloServerPlugin, GraphQLRequestListener } from
'@apollo/server';
import { Plugin } from '@nestjs/apollo';

@Plugin()
export class LoggingPlugin implements ApolloServerPlugin {
  async requestDidStart(): Promise<GraphQLRequestListener<any>> {
    console.log('Request started');
    return {
      async willSendResponse() {
        console.log('Will send response');
      },
    };
  }
}
```

With this in place, we can register the `LoggingPlugin` as a provider.

```
@Module({
  providers: [LoggingPlugin],
})
export class CommonModule {}
```

Nest will automatically instantiate a plugin and apply it to the Apollo Server.

### Using external plugins

There are several plugins provided out-of-the-box. To use an existing plugin, simply import it and add it to the `plugins` array:

```
GraphQLModule.forRoot({
  // ...
})
```

```
plugins: [ApolloServerOperationRegistry({ /* options */})]  
},
```

**info Hint** The `ApolloServerOperationRegistry` plugin is exported from the `@apollo/server-plugin-operation-registry` package.

## Plugins with Mercurius

Some of the existing mercurius-specific Fastify plugins must be loaded after the mercurius plugin (read more [here](#)) on the plugin tree.

**warning Warning** `mercurius-upload` is an exception and should be registered in the main file.

For this, `MercuriusDriver` exposes an optional `plugins` configuration option. It represents an array of objects that consist of two attributes: `plugin` and its `options`. Therefore, registering the `cache` plugin would look like this:

```
GraphQLModule.forRoot({  
  driver: MercuriusDriver,  
  // ...  
  plugins: [  
    {  
      plugin: cache,  
      options: {  
        ttl: 10,  
        policy: {  
          Query: {  
            add: true  
          }  
        }  
      },  
    },  
  ]  
}),
```

## Complexity

**warning** **Warning** This chapter applies only to the code first approach.

Query complexity allows you to define how complex certain fields are, and to restrict queries with a **maximum complexity**. The idea is to define how complex each field is by using a simple number. A common default is to give each field a complexity of 1. In addition, the complexity calculation of a GraphQL query can be customized with so-called complexity estimators. A complexity estimator is a simple function that calculates the complexity for a field. You can add any number of complexity estimators to the rule, which are then executed one after another. The first estimator that returns a numeric complexity value determines the complexity for that field.

The `@nestjs/graphql` package integrates very well with tools like `graphql-query-complexity` that provides a cost analysis-based solution. With this library, you can reject queries to your GraphQL server that are deemed too costly to execute.

### Installation

To begin using it, we first install the required dependency.

```
$ npm install --save graphql-query-complexity
```

### Getting started

Once the installation process is complete, we can define the `ComplexityPlugin` class:

```
import { GraphQLSchemaHost } from "@nestjs/graphql";
import { Plugin } from "@nestjs/apollo";
import {
  ApolloServerPlugin,
  GraphQLRequestListener,
} from 'apollo-server-plugin-base';
import { GraphQLError } from 'graphql';
import {
  fieldExtensionsEstimator,
  getComplexity,
  simpleEstimator,
} from 'graphql-query-complexity';

@Plugin()
export class ComplexityPlugin implements ApolloServerPlugin {
  constructor(private gqlSchemaHost: GraphQLSchemaHost) {}

  async requestDidStart(): Promise<GraphQLRequestListener> {
    const maxComplexity = 20;
    const { schema } = this.gqlSchemaHost;

    return {
      
```

```

async didResolveOperation({ request, document }) {
  const complexity = getComplexity({
    schema,
    operationName: request.operationName,
    query: document,
    variables: request.variables,
    estimators: [
      fieldExtensionsEstimator(),
      simpleEstimator({ defaultComplexity: 1 }),
    ],
  });
  if (complexity > maxComplexity) {
    throw new GraphQLError(
      `Query is too complex: ${complexity}. Maximum allowed
complexity: ${maxComplexity}`,
    );
  }
  console.log('Query Complexity:', complexity);
},
};

}
}

```

For demonstration purposes, we specified the maximum allowed complexity as `20`. In the example above, we used 2 estimators, the `simpleEstimator` and the `fieldExtensionsEstimator`.

- `simpleEstimator`: the simple estimator returns a fixed complexity for each field
- `fieldExtensionsEstimator`: the field extensions estimator extracts the complexity value for each field of your schema

**info Hint** Remember to add this class to the providers array in any module.

## Field-level complexity

With this plugin in place, we can now define the complexity for any field by specifying the `complexity` property in the options object passed into the `@Field()` decorator, as follows:

```

@Field({ complexity: 3 })
title: string;

```

Alternatively, you can define the estimator function:

```

@Field({ complexity: (options: ComplexityEstimatorArgs) => ... })
title: string;

```

## Query/Mutation-level complexity

In addition, `@Query()` and `@Mutation()` decorators may have a `complexity` property specified like so:

```
@Query({ complexity: (options: ComplexityEstimatorArgs) =>
  options.args.count * options.childComplexity })
  items(@Args('count') count: number) {
    return this.itemsService.getItems({ count });
}
```

## Extensions

**warning** **Warning** This chapter applies only to the code first approach.

Extensions is an **advanced, low-level feature** that lets you define arbitrary data in the types configuration. Attaching custom metadata to certain fields allows you to create more sophisticated, generic solutions. For example, with extensions, you can define field-level roles required to access particular fields. Such roles can be reflected at runtime to determine whether the caller has sufficient permissions to retrieve a specific field.

### Adding custom metadata

To attach custom metadata for a field, use the `@Extensions()` decorator exported from the `@nestjs/graphql` package.

```
@Field()
@Extensions({ role: Role.ADMIN })
password: string;
```

In the example above, we assigned the `role` metadata property the value of `Role.ADMIN`. `Role` is a simple TypeScript enum that groups all the user roles available in our system.

Note, in addition to setting metadata on fields, you can use the `@Extensions()` decorator at the class level and method level (e.g., on the query handler).

### Using custom metadata

Logic that leverages the custom metadata can be as complex as needed. For example, you can create a simple interceptor that stores/logs events per method invocation, or a `field middleware` that matches roles required to retrieve a field with the caller permissions (field-level permissions system).

For illustration purposes, let's define a `checkRoleMiddleware` that compares a user's role (hardcoded here) with a role required to access a target field:

```
export const checkRoleMiddleware: FieldMiddleware = async (
  ctx: MiddlewareContext,
  next: NextFn,
) => {
  const { info } = ctx;
  const { extensions } = info.parentType.getFields()[info.fieldName];

  /**
   * In a real-world application, the "userRole" variable
   * should represent the caller's (user) role (for example,
   * "ctx.user.role").
   */
  const userRole = Role.USER;
  if (userRole === extensions.role) {
```

```
// or just "return null" to ignore
throw new ForbiddenException(
  `User does not have sufficient permissions to access
"${info.fieldName}" field.`,
);
}
return next();
};
```

With this in place, we can register a middleware for the `password` field, as follows:

```
@Field({ middleware: [checkRoleMiddleware] })
@Extensions({ role: Role.ADMIN })
password: string;
```

## CLI Plugin

**warning** **Warning** This chapter applies only to the code first approach.

TypeScript's metadata reflection system has several limitations which make it impossible to, for instance, determine what properties a class consists of or recognize whether a given property is optional or required. However, some of these constraints can be addressed at compilation time. Nest provides a plugin that enhances the TypeScript compilation process to reduce the amount of boilerplate code required.

**info** **Hint** This plugin is **opt-in**. If you prefer, you can declare all decorators manually, or only specific decorators where you need them.

### Overview

The GraphQL plugin will automatically:

- annotate all input object, object type and args classes properties with `@Field` unless `@HideField` is used
- set the `nullable` property depending on the question mark (e.g. `name?: string` will set `nullable: true`)
- set the `type` property depending on the type (supports arrays as well)
- generate descriptions for properties based on comments (if `introspectComments` set to `true`)

Please, note that your filenames **must have** one of the following suffixes in order to be analyzed by the plugin: `['.input.ts', '.args.ts', '.entity.ts', '.model.ts']` (e.g., `author.entity.ts`). If you are using a different suffix, you can adjust the plugin's behavior by specifying the `typeFileNameSuffix` option (see below).

With what we've learned so far, you have to duplicate a lot of code to let the package know how your type should be declared in GraphQL. For example, you could define a simple `Author` class as follows:

```
@@filename(authors/models/author.model)
@ObjectType()
export class Author {
  @Field(type => ID)
  id: number;

  @Field({ nullable: true })
  firstName?: string;

  @Field({ nullable: true })
  lastName?: string;

  @Field(type => [Post])
  posts: Post[];
}
```

While not a significant issue with medium-sized projects, it becomes verbose & hard to maintain once you have a large set of classes.

By enabling the GraphQL plugin, the above class definition can be declared simply:

```
@@filename(authors/models/author.model)
@ObjectType()
export class Author {
  @Field(type => ID)
  id: number;
  firstName?: string;
  lastName?: string;
  posts: Post[];
}
```

The plugin adds appropriate decorators on-the-fly based on the **Abstract Syntax Tree**. Thus, you won't have to struggle with `@Field` decorators scattered throughout the code.

**info Hint** The plugin will automatically generate any missing GraphQL properties, but if you need to override them, simply set them explicitly via `@Field()`.

## Comments introspection

With the comments introspection feature enabled, CLI plugin will generate descriptions for fields based on comments.

For example, given an example `roles` property:

```
/**
 * A list of user's roles
 */
@Field(() => [String], {
  description: `A list of user's roles`
})
roles: string[];
```

You must duplicate description values. With `introspectComments` enabled, the CLI plugin can extract these comments and automatically provide descriptions for properties. Now, the above field can be declared simply as follows:

```
/**
 * A list of user's roles
 */
roles: string[];
```

## Using the CLI plugin

To enable the plugin, open `nest-cli.json` (if you use [Nest CLI](#)) and add the following `plugins` configuration:

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "src",
  "compilerOptions": {
    "plugins": ["@nestjs/graphql"]
  }
}
```

You can use the `options` property to customize the behavior of the plugin.

```
"plugins": [
  {
    "name": "@nestjs/graphql",
    "options": {
      "typeFileNameSuffix": [".input.ts", ".args.ts"],
      "introspectComments": true
    }
  }
]
```

The `options` property has to fulfill the following interface:

```
export interface PluginOptions {
  typeFileNameSuffix?: string[];
  introspectComments?: boolean;
}
```

Option	Default	Description
<code>typeFileNameSuffix</code>	<code>['.input.ts', '.args.ts', '.entity.ts', '.model.ts']</code>	GraphQL types files suffix
<code>introspectComments</code>	<code>false</code>	If set to true, plugin will generate descriptions for properties based on comments

If you don't use the CLI but instead have a custom `webpack` configuration, you can use this plugin in combination with `ts-loader`:

```
getCustomTransformers: (program: any) => ({
  before: [require('@nestjs/graphql/plugin').before({}, program)]
}),
```

## SWC builder

For standard setups (non-monorepo), to use CLI Plugins with the SWC builder, you need to enable type checking, as described [here](#).

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

For monorepo setups, follow the instructions [here](#).

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

Now, the serialized metadata file must be loaded by the `GraphQLModule` method, as shown below:

```
import metadata from './metadata'; // <-- file auto-generated by the
"PluginMetadataGenerator"

GraphQLModule.forRoot<...>({
  ...,
  // other options
  metadata,
}),
```

## Integration with `ts-jest` (e2e tests)

When running e2e tests with this plugin enabled, you may run into issues with compiling schema. For example, one of the most common errors is:

```
Object type <name> must define one or more fields.
```

This happens because `jest` configuration does not import `@nestjs/graphql/plugin` plugin anywhere.

To fix this, create the following file in your e2e tests directory:

```
const transformer = require('@nestjs/graphql/plugin');

module.exports.name = 'nestjs-graphql-transformer';
// you should change the version number anytime you change the
configuration below - otherwise, jest will not detect changes
module.exports.version = 1;

module.exports.factory = (cs) => {
  return transformer.before(
    {
```

```
// @nestjs/graphql/plugin options (can be empty)
},
cs.program, // "cs.tsCompiler.program" for older versions of Jest (<=
v27)
);
};
```

With this in place, import AST transformer within your `jest` configuration file. By default (in the starter application), e2e tests configuration file is located under the `test` folder and is named `jest-e2e.json`.

```
{
  ... // other configuration
  "globals": {
    "ts-jest": {
      "astTransformers": {
        "before": ["<path to the file created above>"]
      }
    }
  }
}
```

If you use `jest@^29`, then use the snippet below, as the previous approach got deprecated.

```
{
  ... // other configuration
  "transform": {
    "^.+\\.(t|j)s$": [
      "ts-jest",
      {
        "astTransformers": {
          "before": ["<path to the file created above>"]
        }
      }
    ]
  }
}
```

## Sharing models

**warning Warning** This chapter applies only to the code first approach.

One of the biggest advantages of using Typescript for the backend of your project is the ability to reuse the same models in a Typescript-based frontend application, by using a common Typescript package.

But there's a problem: the models created using the code first approach are heavily decorated with GraphQL related decorators. Those decorators are irrelevant in the frontend, negatively impacting performance.

### Using the model shim

To solve this issue, NestJS provides a "shim" which allows you to replace the original decorators with inert code by using a [webpack](#) (or similar) configuration. To use this shim, configure an alias between the [@nestjs/graphql](#) package and the shim.

For example, for webpack this is resolved this way:

```
resolve: { // see: https://webpack.js.org/configuration/resolve/
  alias: {
    "@nestjs/graphql": path.resolve(__dirname,
    "../node_modules/@nestjs/graphql/dist/extras/graphql-model-shim")
  }
}
```

**info Hint** The [TypeORM](#) package has a similar shim that can be found [here](#).

## Other features

In the GraphQL world, there is a lot of debate about handling issues like **authentication**, or **side-effects** of operations. Should we handle things inside the business logic? Should we use a higher-order function to enhance queries and mutations with authorization logic? Or should we use **schema directives**? There is no single one-size-fits-all answer to these questions.

Nest helps address these issues with its cross-platform features like **guards** and **interceptors**. The philosophy is to reduce redundancy and provide tooling that helps create well-structured, readable, and consistent applications.

### Overview

You can use standard **guards**, **interceptors**, **filters** and **pipes** in the same fashion with GraphQL as with any RESTful application. Additionally, you can easily create your own decorators by leveraging the **custom decorators** feature. Let's take a look at a sample GraphQL query handler.

```
@Query('author')
@UseGuards(AuthGuard)
async getAuthor(@Args('id', ParseIntPipe) id: number) {
  return this.authorsService.findOneById(id);
}
```

As you can see, GraphQL works with both guards and pipes in the same way as HTTP REST handlers. Because of this, you can move your authentication logic to a guard; you can even reuse the same guard class across both a REST and GraphQL API interface. Similarly, interceptors work across both types of applications in the same way:

```
@Mutation()
@UseInterceptors(EventsInterceptor)
async upvotePost(@Args('postId') postId: number) {
  return this.postsService.upvoteById({ id: postId });
}
```

### Execution context

Since GraphQL receives a different type of data in the incoming request, the **execution context** received by both guards and interceptors is somewhat different with GraphQL vs. REST. GraphQL resolvers have a distinct set of arguments: **root**, **args**, **context**, and **info**. Thus guards and interceptors must transform the generic **ExecutionContext** to a **GqlExecutionContext**. This is straightforward:

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

```
@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const ctx = GqlExecutionContext.create(context);
    return true;
  }
}
```

The GraphQL context object returned by `GqlExecutionContext.create()` exposes a `get` method for each GraphQL resolver argument (e.g., `getArgs()`, `getContext()`, etc). Once transformed, we can easily pick out any GraphQL argument for the current request.

## Exception filters

Nest standard `exception filters` are compatible with GraphQL applications as well. As with `ExecutionContext`, GraphQL apps should transform the `ArgumentsHost` object to a `GqlArgumentsHost` object.

```
@Catch(HttpException)
export class HttpExceptionFilter implements GqlExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const gqlHost = GqlArgumentsHost.create(host);
    return exception;
  }
}
```

**info Hint** Both `GqlExceptionFilter` and `GqlArgumentsHost` are imported from the `@nestjs/graphql` package.

Note that unlike the REST case, you don't use the native `response` object to generate a response.

## Custom decorators

As mentioned, the `custom decorators` feature works as expected with GraphQL resolvers.

```
export const User = createParamDecorator(
  (data: unknown, ctx: ExecutionContext) =>
    GqlExecutionContext.create(ctx).getContext().user,
);
```

Use the `@User()` custom decorator as follows:

```
@Mutation()
async upvotePost(
  @User() user: UserEntity,
```

```
@Args('postId') postId: number,
) {}
```

**info Hint** In the above example, we have assumed that the `user` object is assigned to the context of your GraphQL application.

## Execute enhancers at the field resolver level

In the GraphQL context, Nest does not run **enhancers** (the generic name for interceptors, guards and filters) at the field level [see this issue](#): they only run for the top level `@Query()`/`@Mutation()` method. You can tell Nest to execute interceptors, guards or filters for methods annotated with `@ResolveField()` by setting the `fieldResolverEnhancers` option in `GqlModuleOptions`. Pass it a list of '`interceptors`', '`guards`', and/or '`filters`' as appropriate:

```
GraphQLModule.forRoot({
  fieldResolverEnhancers: ['interceptors']
}),
```

**Warning** Enabling enhancers for field resolvers can cause performance issues when you are returning lots of records and your field resolver is executed thousands of times. For this reason, when you enable `fieldResolverEnhancers`, we advise you to skip execution of enhancers that are not strictly necessary for your field resolvers. You can do this using the following helper function:

```
export function isResolvingGraphQLField(context: ExecutionContext): boolean {
  if (context.getType<GqlContextType>() === 'graphql') {
    const gqlContext = GqlExecutionContext.create(context);
    const info = gqlContext.getInfo();
    const parentType = info.parentType.name;
    return parentType !== 'Query' && parentType !== 'Mutation';
  }
  return false;
}
```

## Creating a custom driver

Nest provides two official drivers out-of-the-box: `@nestjs/apollo` and `@nestjs/mercurius`, as well as an API allowing developers to build new **custom drivers**. With a custom driver, you can integrate any GraphQL library or extend the existing integration, adding extra features on top.

For example, to integrate the `express-graphql` package, you could create the following driver class:

```
import { AbstractGraphQLDriver, GqlModuleOptions } from '@nestjs/graphql';
import { graphqlHTTP } from 'express-graphql';
```

```
class ExpressGraphQLDriver extends AbstractGraphQLDriver {  
    async start(options: GqlModuleOptions<any>): Promise<void> {  
        options = await this.graphQLFactory.mergeWithSchema(options);  
  
        const { httpAdapter } = this.httpAdapterHost;  
        httpAdapter.use(  
            '/graphql',  
            graphqlHTTP({  
                schema: options.schema,  
                graphiql: true,  
            }),  
        );  
    }  
  
    async stop() {}  
}
```

And then use it as follows:

```
GraphQLModule.forRoot({  
    driver: ExpressGraphQLDriver,  
});
```

## Federation

Federation offers a means of splitting your monolithic GraphQL server into independent microservices. It consists of two components: a gateway and one or more federated microservices. Each microservice holds part of the schema and the gateway merges the schemas into a single schema that can be consumed by the client.

To quote the [Apollo docs](#), Federation is designed with these core principles:

- Building a graph should be **declarative**. With federation, you compose a graph declaratively from within your schema instead of writing imperative schema stitching code.
- Code should be separated by **concern**, not by types. Often no single team controls every aspect of an important type like a User or Product, so the definition of these types should be distributed across teams and codebases, rather than centralized.
- The graph should be simple for clients to consume. Together, federated services can form a complete, product-focused graph that accurately reflects how it's being consumed on the client.
- It's just **GraphQL**, using only spec-compliant features of the language. Any language, not just JavaScript, can implement federation.

 **Warning** Federation currently does not support subscriptions.

In the following sections, we'll set up a demo application that consists of a gateway and two federated endpoints: Users service and Posts service.

### Federation with Apollo

Start by installing the required dependencies:

```
$ npm install --save @apollo/federation @apollo/subgraph
```

### Schema first

The "User service" provides a simple schema. Note the `@key` directive: it instructs the Apollo query planner that a particular instance of `User` can be fetched if you specify its `id`. Also, note that we `extend` the `Query` type.

```
type User @key(fields: "id") {
  id: ID!
  name: String!
}

extend type Query {
  getUser(id: ID!): User
}
```

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Apollo Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { UsersService } from './users.service';

@Resolver('User')
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query()
  getUser(@Args('id') id: string) {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: string }) {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `ApolloFederationDriver` driver in the configuration object:

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { UsersResolver } from './users.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      typePaths: ['**/*.graphql'],
    }),
  ],
  providers: [UsersResolver],
})
export class AppModule {}
```

## Code first

Start by adding some extra decorators to the `User` entity.

```
import { Directive, Field, ID, ObjectType } from '@nestjs/graphql';

@ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  id: number;

  @Field()
  name: string;
}
```

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Apollo Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { User } from './user.entity';
import { UsersService } from './users.service';

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

  @Query((returns) => User)
  getUser(@Args('id') id: number): User {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: number }): User {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `ApolloFederationDriver` driver in the configuration object:

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { UsersResolver } from './users.resolver';
import { UsersService } from './users.service'; // Not included in this example
```

```
@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: true,
    }),
  ],
  providers: [UsersResolver, UsersService],
})
export class AppModule {}
```

A working example is available [here](#) in code first mode and [here](#) in schema first mode.

## Federated example: Posts

Post service is supposed to serve aggregated posts through the `getPosts` query, but also extend our `User` type with the `user.posts` field.

### Schema first

"Posts service" references the `User` type in its schema by marking it with the `extend` keyword. It also declares one additional property on the `User` type (`posts`). Note the `@key` directive used for matching instances of User, and the `@external` directive indicating that the `id` field is managed elsewhere.

```
type Post @key(fields: "id") {
  id: ID!
  title: String!
  body: String!
  user: User
}

extend type User @key(fields: "id") {
  id: ID! @external
  posts: [Post]
}

extend type Query {
  getPosts: [Post]
}
```

In the following example, the `PostsResolver` provides the `getUser()` method that returns a reference containing `__typename` and some additional properties your application may need to resolve the reference, in this case `id`. `__typename` is used by the GraphQL Gateway to pinpoint the microservice responsible for the User type and retrieve the corresponding instance. The "Users service" described above will be requested upon execution of the `resolveReference()` method.

```

import { Query, Resolver, Parent, ResolveField } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './posts.interfaces';

@Resolver('Post')
export class PostsResolver {
    constructor(private postsService: PostsService) {}

    @Query('getPosts')
    getPosts() {
        return this.postsService.findAll();
    }

    @ResolveField('user')
    getUser(@Parent() post: Post) {
        return { __typename: 'User', id: post.userId };
    }
}

```

Lastly, we must register the `GraphQLModule`, similarly to what we did in the "Users service" section.

```

import {
    ApolloFederationDriver,
    ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { PostsResolver } from './posts.resolver';

@Module({
    imports: [
        GraphQLModule.forRoot<ApolloFederationDriverConfig>({
            driver: ApolloFederationDriver,
            typePaths: ['**/*.graphql'],
        }),
    ],
    providers: [PostsResolvers],
})
export class AppModule {}

```

## Code first

First, we will have to declare a class representing the `User` entity. Although the entity itself lives in another service, we will be using it (extending its definition) here. Note the `@extends` and `@external` directives.

```

import { Directive, ObjectType, Field, ID } from '@nestjs/graphql';
import { Post } from './post.entity';

```

```

@ObjectType()
@Directive('@extends')
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  @Directive('@external')
  id: number;

  @Field((type) => [Post])
  posts?: Post[];
}

```

Now let's create the corresponding resolver for our extension on the `User` entity, as follows:

```

import { Parent, ResolveField, Resolver } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => User)
export class UsersResolver {
  constructor(private readonly postsService: PostsService) {}

  @ResolveField((of) => [Post])
  public posts(@Parent() user: User): Post[] {
    return this.postsService.forAuthor(user.id);
  }
}

```

We also have to define the `Post` entity class:

```

import { Directive, Field, ID, Int, ObjectType } from '@nestjs/graphql';
import { User } from './user.entity';

@ObjectType()
@Directive('@key(fields: "id")')
export class Post {
  @Field((type) => ID)
  id: number;

  @Field()
  title: string;

  @Field((type) => Int)
  authorId: number;

  @Field((type) => User)
  user?: User;
}

```

And its resolver:

```
import { Query, Args, ResolveField, Resolver, Parent } from
'@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => Post)
export class PostsResolver {
    constructor(private readonly postsService: PostsService) {}

    @Query((returns) => Post)
    findPost(@Args('id') id: number): Post {
        return this.postsService.findOne(id);
    }

    @Query((returns) => [Post])
    getPosts(): Post[] {
        return this.postsService.all();
    }

    @ResolveField((of) => User)
    user(@Parent() post: Post): any {
        return { __typename: 'User', id: post.authorId };
    }
}
```

And finally, tie it together in a module. Note the schema build options, where we specify that `User` is an orphaned (external) type.

```
import {
    ApolloFederationDriver,
    ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { User } from './user.entity';
import { PostsResolvers } from './posts.resolvers';
import { UsersResolvers } from './users.resolvers';
import { PostsService } from './posts.service'; // Not included in example

@Module({
    imports: [
        GraphQLModule.forRoot<ApolloFederationDriverConfig>({
            driver: ApolloFederationDriver,
            autoSchemaFile: true,
            buildSchemaOptions: {
                orphanedTypes: [User],
            },
        }),
    ],
    providers: [PostsService],
})
```

```

    },
],
providers: [PostsResolver, UsersResolver, PostsService],
})
export class AppModule {}

```

A working example is available [here](#) for the code first mode and [here](#) for the schema first mode.

## Federated example: Gateway

Start by installing the required dependency:

```
$ npm install --save @apollo/gateway
```

The gateway requires a list of endpoints to be specified and it will auto-discover the corresponding schemas. Therefore the implementation of the gateway service will remain the same for both code and schema first approaches.

```

import { IntrospectAndCompose } from '@apollo/gateway';
import { ApolloGatewayDriver, ApolloGatewayDriverConfig } from
  '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloGatewayDriverConfig>({
      driver: ApolloGatewayDriver,
      server: {
        // ... Apollo server options
        cors: true,
      },
      gateway: {
        supergraphSdl: new IntrospectAndCompose({
          subgraphs: [
            { name: 'users', url: 'http://user-service/graphql' },
            { name: 'posts', url: 'http://post-service/graphql' },
          ],
        }),
      },
    ]),
  ],
})
export class AppModule {}

```

A working example is available [here](#) for the code first mode and [here](#) for the schema first mode.

## Federation with Mercurius

Start by installing the required dependencies:

```
$ npm install --save @apollo/subgraph @nestjs/mercurius
```

**info Note** The `@apollo/subgraph` package is required to build a subgraph schema (`buildSubgraphSchema`, `printSubgraphSchema` functions).

### Schema first

The "User service" provides a simple schema. Note the `@key` directive: it instructs the Mercurius query planner that a particular instance of `User` can be fetched if you specify its `id`. Also, note that we `extend` the `Query` type.

```
type User @key(fields: "id") {
  id: ID!
  name: String!
}

extend type Query {
  getUser(id: ID!): User
}
```

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Mercurius Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { UsersService } from './users.service';

@Resolver('User')
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query()
  getUser(@Args('id') id: string) {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: string }) {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `MercuriusFederationDriver` driver in the configuration object:

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { UsersResolver } from './users.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      typePaths: ['**/*.graphql'],
      federationMetadata: true,
    }),
  ],
  providers: [UsersResolver],
})
export class AppModule {}
```

## Code first

Start by adding some extra decorators to the `User` entity.

```
import { Directive, Field, ID, ObjectType } from '@nestjs/graphql';

@ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field(() => ID)
  id: number;

  @Field()
  name: string;
}
```

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Mercurius Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { User } from './user.entity';
import { UsersService } from './users.service';
```

```

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

  @Query((returns) => User)
  getUser(@Args('id') id: number): User {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: number }): User {
    return this.usersService.findById(reference.id);
  }
}

```

Finally, we hook everything up by registering the [GraphQLModule](#) passing the [MercuriusFederationDriver](#) driver in the configuration object:

```

import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { UsersResolver } from './users.resolver';
import { UsersService } from './users.service'; // Not included in this example

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      autoSchemaFile: true,
      federationMetadata: true,
    }),
  ],
  providers: [UsersResolver, UsersService],
})
export class AppModule {}

```

## Federated example: Posts

Post service is supposed to serve aggregated posts through the [getPosts](#) query, but also extend our [User](#) type with the [user.posts](#) field.

### Schema first

"Posts service" references the [User](#) type in its schema by marking it with the [extend](#) keyword. It also declares one additional property on the [User](#) type ([posts](#)). Note the [@key](#) directive used for matching

instances of User, and the `@external` directive indicating that the `id` field is managed elsewhere.

```
type Post @key(fields: "id") {
  id: ID!
  title: String!
  body: String!
  user: User
}

extend type User @key(fields: "id") {
  id: ID! @external
  posts: [Post]
}

extend type Query {
  getPosts: [Post]
}
```

In the following example, the `PostsResolver` provides the `getUser()` method that returns a reference containing `__typename` and some additional properties your application may need to resolve the reference, in this case `id`. `__typename` is used by the GraphQL Gateway to pinpoint the microservice responsible for the User type and retrieve the corresponding instance. The "Users service" described above will be requested upon execution of the `resolveReference()` method.

```
import { Query, Resolver, Parent, ResolveField } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './posts.interfaces';

@Resolver('Post')
export class PostsResolver {
  constructor(private postsService: PostsService) {}

  @Query('getPosts')
  getPosts() {
    return this.postsService.findAll();
  }

  @ResolveField('user')
  getUser(@Parent() post: Post) {
    return { __typename: 'User', id: post.userId };
  }
}
```

Lastly, we must register the `GraphQLModule`, similarly to what we did in the "Users service" section.

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
```

```

} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { PostsResolver } from './posts.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      federationMetadata: true,
      typePaths: ['**/*.graphql'],
    }),
  ],
  providers: [PostsResolvers],
})
export class AppModule {}

```

## Code first

First, we will have to declare a class representing the `User` entity. Although the entity itself lives in another service, we will be using it (extending its definition) here. Note the `@extends` and `@external` directives.

```

import { Directive, ObjectType, Field, ID } from '@nestjs/graphql';
import { Post } from './post.entity';

@ObjectType()
@Directive('@extends')
@Directive('@key(fields: "id")')
export class User {
  @Field(() => ID)
  @Directive('@external')
  id: number;

  @Field(() => [Post])
  posts?: Post[];
}

```

Now let's create the corresponding resolver for our extension on the `User` entity, as follows:

```

import { Parent, ResolveField, Resolver } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => User)
export class UsersResolver {
  constructor(private readonly postsService: PostsService) {}

  @ResolveField((of) => [Post])

```

```

public posts(@Parent() user: User): Post[] {
    return this.postsService.forAuthor(user.id);
}
}

```

We also have to define the **Post** entity class:

```

import { Directive, Field, ID, Int, ObjectType } from '@nestjs/graphql';
import { User } from './user.entity';

@ObjectType()
@Directive('@key(fields: "id")')
export class Post {
    @Field((type) => ID)
    id: number;

    @Field()
    title: string;

    @Field((type) => Int)
    authorId: number;

    @Field((type) => User)
    user?: User;
}

```

And its resolver:

```

import { Query, Args, ResolveField, Resolver, Parent } from
'@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => Post)
export class PostsResolver {
    constructor(private readonly postsService: PostsService) {}

    @Query((returns) => Post)
    findPost(@Args('id') id: number): Post {
        return this.postsService.findOne(id);
    }

    @Query((returns) => [Post])
    getPosts(): Post[] {
        return this.postsService.all();
    }

    @ResolveField((of) => User)

```

```
user(@Parent() post: Post): any {
    return { __typename: 'User', id: post.authorId };
}
}
```

And finally, tie it together in a module. Note the schema build options, where we specify that `User` is an orphaned (external) type.

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { User } from './user.entity';
import { PostsResolvers } from './posts.resolvers';
import { UsersResolvers } from './users.resolvers';
import { PostsService } from './posts.service'; // Not included in example

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      autoSchemaFile: true,
      federationMetadata: true,
      buildSchemaOptions: {
        orphanedTypes: [User],
      },
    }),
  ],
  providers: [PostsResolver, UsersResolver, PostsService],
})
export class AppModule {}
```

## Federated example: Gateway

The gateway requires a list of endpoints to be specified and it will auto-discover the corresponding schemas. Therefore the implementation of the gateway service will remain the same for both code and schema first approaches.

```
import {
  MercuriusGatewayDriver,
  MercuriusGatewayDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [
```

```

GraphQLModule.forRoot<MercuriusGatewayDriverConfig>({
  driver: MercuriusGatewayDriver,
  gateway: {
    services: [
      { name: 'users', url: 'http://user-service/graphql' },
      { name: 'posts', url: 'http://post-service/graphql' },
    ],
  },
}),
],
})
export class AppModule {}

```

## Federation 2

To quote the [Apollo docs](#), Federation 2 improves developer experience from the original Apollo Federation (called Federation 1 in this doc), which is backward compatible with most original supergraphs.

**warning** **Warning** Mercurius doesn't fully support Federation 2. You can see the list of libraries that support Federation 2 [here](#).

In the following sections, we'll upgrade the previous example to Federation 2.

### Federated example: Users

One change in Federation 2 is that entities have no originating subgraph, so we don't need to extend [Query](#) anymore. For more detail please refer to [the entities topic](#) in Apollo Federation 2 docs.

#### Schema first

We can simply remove [extend](#) keyword from the schema.

```

type User @key(fields: "id") {
  id: ID!
  name: String!
}

type Query {
  getUser(id: ID!): User
}

```

#### Code first

To use Federation 2, we need to specify the federation version in [autoSchemaFile](#) option.

```

import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
}

```

```

} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { UsersResolver } from './users.resolver';
import { UserService } from './users.service'; // Not included in this
example

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: {
        federation: 2,
      },
    }),
  ],
  providers: [UsersResolver, UserService],
})
export class AppModule {}

```

## Federated example: Posts

With the same reason as above, we don't need to extend `User` and `Query` anymore.

### Schema first

We can simply remove `extend` and `external` directives from the schema

```

type Post @key(fields: "id") {
  id: ID!
  title: String!
  body: String!
  user: User
}

type User @key(fields: "id") {
  id: ID!
  posts: [Post]
}

type Query {
  getPosts: [Post]
}

```

### Code first

Since we don't extend `User` entity anymore, we can simply remove `extends` and `external` directives from `User`.

```
import { Directive, ObjectType, Field, ID } from '@nestjs/graphql';
import { Post } from './post.entity';

@ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  id: number;

  @Field((type) => [Post])
  posts?: Post[];
}
```

Also, similarly to the User service, we need to specify in the [GraphQLModule](#) to use Federation 2.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { User } from './user.entity';
import { PostsResolvers } from './posts.resolvers';
import { UsersResolvers } from './users.resolvers';
import { PostsService } from './posts.service'; // Not included in example

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: {
        federation: 2,
      },
      buildSchemaOptions: {
        orphanedTypes: [User],
      },
    }),
  ],
  providers: [PostsResolver, UsersResolver, PostsService],
})
export class AppModule {}
```

## Migrating to v11 from v10

This chapter provides a set of guidelines for migrating from [@nestjs/graphql](#) version 10 to version 11. As part of this major release, we updated the Apollo driver to be compatible with Apollo Server v4 (instead of v3). Note: there are several breaking changes in Apollo Server v4 (especially around plugins and ecosystem packages), so you'll have to update your codebase accordingly. For more information, see the [Apollo Server v4 migration guide](#).

### Apollo packages

Instead of installing the [apollo-server-express](#) package, you'll have to install [@apollo/server](#):

```
$ npm uninstall apollo-server-express
$ npm install @apollo/server
```

If you use the Fastify adapter, you'll have to install the [@as-integrations/fastify](#) package instead:

```
$ npm uninstall apollo-server-fastify
$ npm install @apollo/server @as-integrations/fastify
```

### Mercurius packages

Mercurius gateway is no longer a part of the [mercurius](#) package. Instead, you'll have to install the [@mercuriusjs/gateway](#) package separately:

```
$ npm install @mercuriusjs/gateway
```

Similarly, for creating federated schemas, you'll have to install the [@mercuriusjs/federation](#) package:

```
$ npm install @mercuriusjs/federation
```

## Migrating to v10 from v9

This chapter provides a set of guidelines for migrating from [@nestjs/graphql](#) version 9 to version 10. The focus of this major-version release is to provide a lighter, platform-agnostic core library.

### Introducing "driver" packages

In the latest version, we made a decision to break the [@nestjs/graphql](#) package up into a few separate libraries, letting you choose whether to use Apollo ([@nestjs/apollo](#)), Mercurius ([@nestjs/mercurius](#)), or another GraphQL library in your project.

This implies that now you have to explicitly specify what driver your application will use.

```
// Before
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [
    GraphQLModule.forRoot({
      autoSchemaFile: 'schema.gql',
    }),
  ],
})
export class AppModule {}

// After
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      autoSchemaFile: 'schema.gql',
    }),
  ],
})
export class AppModule {}
```

## Plugins

Apollo Server plugins let you perform custom operations in response to certain events. Since this is an exclusive Apollo feature, we moved it from the `@nestjs/graphql` to the newly created `@nestjs/apollo` package so you'll have to update imports in your application.

```
// Before
import { Plugin } from '@nestjs/graphql';

// After
import { Plugin } from '@nestjs/apollo';
```

## Directives

`schemaDirectives` feature has been replaced with the new [Schema directives API](#) in v8 of `@graphql-tools/schema` package.

```
// Before
import { SchemaDirectiveVisitor } from '@graphql-tools/utils';
import { defaultFieldResolver, GraphQLField } from 'graphql';

export classUpperCaseDirective extends SchemaDirectiveVisitor {
  visitFieldDefinition(field: GraphQLField<any, any>) {
    const { resolve = defaultFieldResolver } = field;
    field.resolve = async function (...args) {
      const result = await resolve.apply(this, args);
      if (typeof result === 'string') {
        return result.toUpperCase();
      }
      return result;
    };
  }
}

// After
import { getDirective, MapperKind, mapSchema } from '@graphql-tools/utils';
import { defaultFieldResolver, GraphQLSchema } from 'graphql';

export function upperDirectiveTransformer(
  schema: GraphQLSchema,
  directiveName: string,
) {
  return mapSchema(schema, {
    [MapperKind.OBJECT_FIELD]: (fieldConfig) => {
      const upperDirective = getDirective(
        schema,
        fieldConfig,
        directiveName,
      )?.[0];

      if (upperDirective) {
        const { resolve = defaultFieldResolver } = fieldConfig;

        // Replace the original resolver with a function that *first*
        // calls the original resolver, then converts its result to upper case
        fieldConfig.resolve = async function (source, args, context, info) {
          const result = await resolve(source, args, context, info);
          if (typeof result === 'string') {
            return result.toUpperCase();
          }
          return result;
        };
        return fieldConfig;
      }
    },
  });
}
```

To apply this directive implementation to a schema that contains `@upper` directives, use the `transformSchema` function:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  ...
  transformSchema: schema => upperDirectiveTransformer(schema, 'upper'),
})
```

## Federation

`GraphQLFederationModule` has been removed and replaced with the corresponding driver class:

```
// Before
GraphQLFederationModule.forRoot({
  autoSchemaFile: true,
});

// After
GraphQLModule.forRoot<ApolloFederationDriverConfig>({
  driver: ApolloFederationDriver,
  autoSchemaFile: true,
});
```

**info Hint** Both `ApolloFederationDriver` class and `ApolloFederationDriverConfig` are exported from the `@nestjs/apollo` package.

Likewise, instead of using a dedicated `GraphQLGatewayModule`, simply pass the appropriate `driver` class to your `GraphQLModule` settings:

```
// Before
GraphQLGatewayModule.forRoot({
  gateway: {
    supergraphSdl: new IntrospectAndCompose({
      subgraphs: [
        { name: 'users', url: 'http://localhost:3000/graphql' },
        { name: 'posts', url: 'http://localhost:3001/graphql' },
      ],
    }),
  },
});

// After
GraphQLModule.forRoot<ApolloGatewayDriverConfig>({
  driver: ApolloGatewayDriver,
  gateway: {
    supergraphSdl: new IntrospectAndCompose({
```

```
subgraphs: [
  { name: 'users', url: 'http://localhost:3000/graphql' },
  { name: 'posts', url: 'http://localhost:3001/graphql' },
],
}),
},
});
```

info **Hint** Both `ApolloGatewayDriver` class and `ApolloGatewayDriverConfig` are exported from the `@nestjs/apollo` package.