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 `allowUnknowns` 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 isAuthEnabled(): boolean {
    return this.configService.get('AUTH_ENABLED') === 'true';
  }
}
@@switch
@Dependencies(ConfigService)
@Injectable()
export class ApiConfigService {
  constructor(configService) {
    this.configService = configService;
  }

  get isAuthEnabled() {
    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.isAuthEnabled) {
      // Authentication is enabled
    }
  }
}
@@switch
@Dependencies(ApiConfigService)
@Injectable()
export class AppService {
  constructor(apiConfigService) {
    if (apiConfigService.isAuthEnabled) {
      // 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 a 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: [],
```

```
      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.

`retryAttempts`	Number of attempts to connect to the database (default: `10`)
`retryDelay`	Delay between connection retry attempts (ms) (default: `3000`)
`autoLoadEntities`	If `true`, entities will be loaded automatically (default: `false`)

> 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 { UsersService } from './users.service';
import { UsersController } from './users.controller';

@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. 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.forRootAsync({
  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.

`retryAttempts`	Number of attempts to connect to the database (default: `10`)
`retryDelay`	Delay between connection retry attempts (ms) (default: `3000`)
`autoLoadModels`	If `true`, models will be loaded automatically (default: `false`)
`keepConnectionAlive`	If `true`, connection will not be closed on the application shutdown (default:

	`false`)
`synchronize`	If `true`, automatically loaded models will be synchronized (default: `true`)

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 { UsersService } from './users.service';
import { UsersController } from './users.controller';

@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 - `SequelizeModule` 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;
        },
```

```
        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: ValidationError[]) => any;
}
```

In addition to these, all `class-validator` options (inherited from the `ValidatorOptions` interface) are available:

Option	Type	Description

`enableDebugMessages`	`boolean`	If set to true, validator will print extra warning messages to the console when something is not right.
`skipUndefinedProperties`	`boolean`	If set to true then validator will skip validation of all properties that are undefined in the validating object.
`skipNullProperties`	`boolean`	If set to true then validator will skip validation of all properties that are null in the validating object.
`skipMissingProperties`	`boolean`	If set to true then validator will skip validation of all properties that are null or undefined in the validating object.
`whitelist`	`boolean`	If set to true, validator will strip validated (returned) object of any properties that do not use any validation decorators.
`forbidNonWhitelisted`	`boolean`	If set to true, instead of stripping non-whitelisted properties validator will throw an exception.
`forbidUnknownValues`	`boolean`	If set to true, attempts to validate unknown objects fail immediately.
`disableErrorMessages`	`boolean`	If set to true, validation errors will not be returned to the client.
`errorHttpStatusCode`	`number`	This setting allows you to specify which exception type will be used in case of an error. By default it throws `BadRequestException`.
`exceptionFactory`	`Function`	Takes an array of the validation errors and returns an exception object to be thrown.
`groups`	`string[]`	Groups to be used during validation of the object.
`always`	`boolean`	Set default for `always` 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`. `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**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:

`URI Versioning`	The version will be passed within the URI of the request (default)
`Header Versioning`	A custom request header will specify the version
`Media Type Versioning`	The `Accept` header of the request will specify the version
`Custom Versioning`	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`. They 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 the 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:

```
* * * * * *
| | | | | |
| | | | | day of week
| | | | months
| | | day of month
| | hours
| minutes
seconds (optional)
```

Some sample cron patterns are:

`* * * * * *`	every second
`45 * * * * *`	every minute, on the 45th second
`0 10 * * * *`	every hour, at the start of the 10th minute
`0 */30 9-17 * * *`	every 30 minutes between 9am and 5pm
`0 30 11 * * 1-5`	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.

`name`	Useful to access and control a cron job after it's been declared.
`timeZone`	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 Moment Timezone website.
`utcOffset`	This allows you to specify the offset of your timezone rather than using the `timeZone` param.
`disabled`	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('* * 0 * * *', {
    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.getCronJob('notifications');

job.stop();
console.log(job.lastDate());
```

The `getCronJob()` 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:

```
getCrons() {
  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,
      },
    }),
  ],
})
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}...`,
      );
    }
    ...
```

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
`@OnQueueError()`	`@OnGlobalQueueError()`	`handler(error: Error)` - An error occurred. `error` contains the triggering error.
`@OnQueueWaiting()`	`@OnGlobalQueueWaiting()`	`handler(jobId: number | string)` - A Job is waiting to be processed as soon as a worker is idling. `jobId` contains the id for the job that has entered this state.
`@OnQueueActive()`	`@OnGlobalQueueActive()`	`handler(job: Job)` - Job `job`has started.
`@OnQueueStalled()`	`@OnGlobalQueueStalled()`	`handler(job: Job)` - Job `job` has been marked as stalled. This is useful for debugging job workers that crash or pause the event loop.
`@OnQueueProgress()`	`@OnGlobalQueueProgress()`	`handler(job: Job, progress: number)` - Job `job`'s progress was updated to value `progress`.
`@OnQueueCompleted()`	`@OnGlobalQueueCompleted()`	`handler(job: Job, result: any)` Job `job` successfully completed with a result `result`.
`@OnQueueFailed()`	`@OnGlobalQueueFailed()`	`handler(job: Job, err: Error)` Job `job` failed with reason `err`.

`@OnQueuePaused()`	`@OnGlobalQueuePaused()`	`handler()` The queue has been paused.
`@OnQueueResumed()`	`@OnGlobalQueueResumed()`	`handler(job: Job)` The queue has been resumed.
`@OnQueueCleaned()`	`@OnGlobalQueueCleaned()`	`handler(jobs: Job[], type: string)` Old jobs have been cleaned from the queue. `jobs` is an array of cleaned jobs, and `type` is the type of jobs cleaned.
`@OnQueueDrained()`	`@OnGlobalQueueDrained()`	`handler()` Emitted whenever the queue has processed all the waiting jobs (even if there can be some delayed jobs not yet processed).
`@OnQueueRemoved()`	`@OnGlobalQueueRemoved()`	`handler(job: Job)` Job `job` 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 be not 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 }\}" }}
  </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 `middie` 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.