NestJS in a NxMonoRepo

        - As Object with `provide` and `useExisting` properties, whereby the value of the `useExisting` property is the class of the service
- Modules
    - Feature (Child) Modules
    - Global Modules
    - Dynamic Modules
    - Shared Modules
    - Core Module (Singleton)
- Middleware
    - Applying Middleware
        - `forRoutes()`
    - Multiple Middleware
    - Excluding Routes
    - Functional Middleware
    - Global Middleware
    - Further Details
- Exception Filters
    - Throwing Standard Exceptions
    - Custom Exceptions
    - Built-in HTTP exceptions
    - Full Control with Exception Filters
    - Arguments Host
    - Binding Filters
        - `APP_FILTER` Provider (Module-scoped Filter)
        - `@UseFilters()` Decorator (Controller-scoped Filter)
        - `@UseFilters()` Decorator (Method-scoped Filter)
        - Application-scoped Filter
    - Catch Everything
    - Inheritance
    - Exception Filters vs Pipes
- Pipes
    - Built-in Pipes
        - ValidationPipe
        - ParseIntPipe
        - ParseBoolPipe
        - ParseArrayPipe
        - ParseFloatPipe
        - ParseUUIDPipe

- Decorator Factories
- Decorator Composition
- Passing Data
- Decorator Internals
- Decorator as middleware
- Decorator inheritance
- Decorator ordering
- Decorator interface
- Working with Pipes
- Working with Guards
- Working with Interceptors
- Working with Filters

Nx Setup

Create a new NestJS project within a NxMonoRepo via Nx Console:

```
npm i -g @nestjs/cli
npm i -D @nx/nest
npx nx generate @nx/nest:application
  --name=middleware-basis-app
  --frontendProject=basis-app
  --directory=apps/basis/middleware-basis-app
  --projectNameAndRootFormat=as-provided
  --tags=middleware-basis-app
  --no-interactive
  # --dry-run
```

Nx Workspace: Run the NestJS Application

```
nx run middleware-basis-app:serve:development
```

Or run via `nest` (If created with NestJS CLI)

> **Hint** If created with **nestjs/cli (nest)** instead of **Nx**
> To speed up the development process (x20 times faster builds), you can use the SWC builder by passing the `-b swc` flag to the `start` script, as follows `nest start -b swc` .

Nx & SWC

For Nx & swc -> see https://nx.dev/nx-api/js/executors/swc

Theoretical Background

Controllers

Controllers are responsible for handling **incoming requests** and **returning responses** to the client and for transforming the incoming request to the format required by the service.

They should be kept as lean as possible and should not contain any business logic except for routing logic. They are bound to a specific **path** (or paths) and **HTTP method** combination.

Example

CatsController

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
} from '@nestjs/common';
import { CreateCatDto } from './dto/create-cat.dto';
import { UpdateCatDto } from './dto/update-cat.dto';
import { CatsService } from './cats.service';
import { Cat } from './entities/cat.entity';

/**
 * The `@Controller()` decorator takes an optional `path`
 * argument that will be prefixed to each path
 * defined within the controller. `/cats` in this case.
 */

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  /**
   * The `@Post()` decorator takes an optional `path` argument
   * that will be appended to the path defined at the controller level.
   * In this case, the path will be `/cats` + `/create`.
   *
   * The `@Body()` decorator takes an optional `param` argument
   * that will be used to retrieve the value of the parameter
   * from the request object.
   *
   * In this case, the value of the `createCatDto` parameter
   * will be retrieved from the `body` property of the request object.
   *
   * The `create()` method returns a `Promise` of type `Cat`.
   */
```

```
@Post()
create(@Body() createCatDto: CreateCatDto): Promise<Cat> {
  return this.catsService.create(createCatDto);
}

/**
 * The `@Get()` decorator takes an optional `path` argument
 * that will be appended to the path defined at the controller level.
 * In this case, the path will be `/cats` + `/findAll`.
 * The `findAll()` method returns a `Promise` of type `Cat[]`.
 */

@Get()
findAll(): Promise<Cat[]> {
  return this.catsService.findAll();
}

/**
 * The `@Get()` decorator takes an optional `path` argument
 * that will be appended to the path defined at the controller level.
 * In this case, the path will be `/cats` + `/:id`.
 *
 * The `@Param()` decorator takes an optional `param` argument
 * that will be used to retrieve the value of the parameter
 * from the request object.
 *
 * In this case, the value of the `id` parameter
 * will be retrieved from the `params` property of the request object.
 * The `findOne()` method returns a `Promise` of type `Cat`.
 */

@Get(':id')
findOne(@Param('id') id: string): Promise<Cat> {
  return this.catsService.findOne(+id);
}

/**
 * The `@Patch()` decorator takes an optional `path` argument
 * that will be appended to the path defined at the controller level.
 * In this case, the path will be `/cats` + `/:id`.
 *
 * The `@Param()` decorator takes an optional `param` argument
 * that will be used to retrieve the value of the parameter
```

```
 * from the request object.
 *
 * In this case, the value of the `id` parameter
 * will be retrieved from the `params` property of the request object.
 *
 * The `@Body()` decorator takes an optional `param` argument
 * that will be used to retrieve the value of the parameter
 * from the request object.
 *
 * In this case, the value of the `updateCatDto` parameter
 * will be retrieved from the `body` property of the request object.
 *
 * The `update()` method returns a `Promise` of type `Cat`.
 */

@Patch(':id')
update(
  @Param('id') id: string,
  @Body() updateCatDto: UpdateCatDto,
): Promise<Cat> {
  return this.catsService.update(+id, updateCatDto);
}

/**
 * The `@Delete()` decorator takes an optional `path` argument
 * that will be appended to the path defined at the controller level.
 *
 * In this case, the path will be `/cats` + `/:id`.
 *
 * The `@Param()` decorator takes an optional `param` argument
 * that will be used to retrieve the value of the parameter
 * from the request object.
 *
 * In this case, the value of the `id` parameter
 * will be retrieved from the `params` property of the request object.
 *
 * The `remove()` method returns a `Promise` of type `void`.
 */

@Delete(':id')
remove(@Param('id') id: string): Promise<void> {
  return this.catsService.remove(+id);
```

```
  }
}
```

CreateCatDto

```
export class CreateCatDto {
  name: string;
  age: number;
  breed: string;
}
```

UpdateCatDto

```
export class UpdateCatDto {
  name: string;
  age: number;
  breed: string;
}
```

Cat Class

```
export class Cat {
  id: number;
  name: string;
  age: number;
  breed: string;
}
```

CatsService

```
import { Injectable } from '@nestjs/common';
import { CreateCatDto } from './dto/create-cat.dto';
import { UpdateCatDto } from './dto/update-cat.dto';
import { Cat } from './entities/cat.entity';

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [];

  create(createCatDto: CreateCatDto): Promise<Cat> {
    const cat = new Cat();
    cat.name = createCatDto.name;
    cat.age = createCatDto.age;
    cat.breed = createCatDto.breed;

    this.cats.push(cat);
    return Promise.resolve(cat);
  }

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

  findOne(id: number): Promise<Cat> {
    return Promise.resolve(this.cats[id]);
  }

  update(id: number, updateCatDto: UpdateCatDto): Promise<Cat> {
    const cat = this.cats[id];
    cat.name = updateCatDto.name;
    cat.age = updateCatDto.age;
    cat.breed = updateCatDto.breed;

    return Promise.resolve(cat);
  }

  remove(id: number): Promise<void> {
    this.cats.splice(id, 1);
    return Promise.resolve();
  }
}
```

Providers

Everything, that can be annotated with the `@Injectable()` decorator, can be used as a provider. The `@Injectable()` decorator is optional, but it is good practice to use it to keep the code clean and explicit.

Providers & Scope

By default, Nest makes each provider **singleton**, which means that **the same instance of each provider will be shared across the entire module**. If you want to **limit the scope of a provider**, you can use the `@Scope()` decorator.

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

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

The `@Scope()` decorator takes a single argument, which is a `Scope` enum member. The following table describes the available options:

> **Hint** The `REQUEST` scope is **not** available in **microservices**. If you want to **share** the provider instance across consumers, you should **not** use the `REQUEST` scope.

Scope	Description
`DEFAULT`	The provider is instantiated **only once** in the scope of the entire application. The same instance is shared across **all** consumers.
`TRANSIENT`	The provider is instantiated **every time** it is requested. This behavior is **independent** of the consumer's lifetime, which means that **every** consumer gets a **dedicated** instance of the provider.
`REQUEST`	The provider is instantiated **once** per incoming request. That means that **each** provider instance lives **only** within the **scope** of a single request and **is not** shared across consumers. If you want to **share** the provider instance across consumers, you should **not** use the `REQUEST` scope.

Providers, Value Providers, Factory Providers

Value providers are a special type of provider that **returns a value** rather than an instance when requested. Value providers are **not** instantiated by the Nest IoC container.

Factory providers are a special type of provider that returns a value created by a **factory function**. Factory providers are **not** instantiated by the Nest IoC container.

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

@Injectable()
export class ConfigService {
  private readonly envConfig: Record<string, string>;

  constructor() {
    this.envConfig = dotenv.parse(fs.readFileSync('.env'));
  }

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

Optional Providers

If you want to **inject** a provider **conditionally**, you can use the `@Optional()` decorator. If the provider is not found, the `@Optional()` decorator will cause the **dependency injector** to **inject** `undefined` instead of throwing an exception.

```
import { Injectable, Optional } from '@nestjs/common';
import { CatsService } from './cats.service';

@Injectable()
export class DogsService {
  constructor(@Optional() private catsService: CatsService) {}
}
```

Property-based Injection

Details online https://docs.nestjs.com/providers#property-based-injection

The technique we've used so far is called constructor-based injection, as providers are injected via the constructor method. In some very specific cases, **property-based injection** might be useful. For instance, if your top-level class depends on either one or multiple providers, passing them all the way up by calling `super()` in sub-classes from the constructor can be very tedious. In order to avoid this, you can use the `@Inject()` decorator at the property level.

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

@Injectable()
export class HttpService<T> {
  @Inject('HTTP_OPTIONS')
  private readonly httpClient: T;
}
```

Provider registration

Providers are registered in modules.

Adding the service to the providers array of the @Module() decorator

CatsController has CatsService as a dependency. The CatsService is a provider. The CatsService is registered in the providers array of the @Module() decorator.

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

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

As Object with `provide` and `useClass` properties

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

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

As Object with `provide` and `useValue` properties, whereby the value is an instance of the service

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

@Module({
  controllers: [CatsController],
  providers: [
    {
      provide: 'CatsService',
      useValue: new CatsService(),
    },
  ],
})
export class CatsModule {}
```

As Object with `provide` and `useFactory` properties, whereby the factory function returns an instance of the service

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

@Module({
  controllers: [CatsController],
  providers: [
    {
      provide: 'CatsService',
      useFactory: () => {
        const catsService = new CatsService();
        // ... do something with the catsService
        // e.g. bind to events, etc.
        return catsService;
      },
    },
  ],
})
export class CatsModule {}
```

As Object with `provide`, `useFactory` and `inject` properties, whereby the factory function returns an instance of the service and awaits the result of the injected service before returning the instance of the service

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

@Module({
  controllers: [CatsController],
  providers: [
    {
      provide: 'CatsService',
      useFactory: async () => {
        const catsService = new CatsService();
        await catsService.init();
        return catsService;
      },
    },
  ],
})
export class CatsModule {}
```

As Object with `provide`, `useFactory` and `inject` properties, whereby the factory function accepts parameters (`connection: Connection`) and dependend on them returns one or the other service instance

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

@Module({
  controllers: [CatsController],
  providers: [
    {
      provide: 'CatsService',
      useFactory: (connection: Connection) => {
        if (connection.isConnected) {
          return new CatsService(connection);
        }
        return new MockCatsService();
      },
      inject: [Connection],
    },
  ],
})
export class CatsModule {}
```

As Object with `provide` , `useFactory` and `inject` properties, whereby the factory function accepts a provider (`optionsProvider: OptionsProvider`) as parameter, awaits its result and creates the service instance with the result.

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

@Module({
  controllers: [CatsController],
  providers: [
    {
      provide: 'CatsService',
      useFactory: async (optionsProvider: OptionsProvider) => {
        const options = await optionsProvider.get();
        return new CatsService(options);
      },
      inject: [OptionsProvider],
    },
  ],
})
export class CatsModule {}
```

As Object with `provide` and `useExisting` properties, whereby the value of the `useExisting` property is the class of the service

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';
import { Connection } from './connection.provider';

@Module({
  controllers: [CatsController],
  providers: [
    CatsService,
    Connection,
    {
      provide: 'CONNECTION',
      useExisting: Connection,
    },
  ],
})
export class CatsModule {}
```

Modules

Source: https://docs.nestjs.com/modules

Each application has at least one module, a **root module**. The root module is the starting point Nest uses to build the **application graph** - the internal data structure Nest uses to resolve module and provider relationships and dependencies.

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

A module can have **one or more** of the following properties:

- `controllers` - the set of controllers defined in the module which have to be instantiated
- `exports` - the subset of providers that are provided by this module and should be available in other modules which import this module
- `imports` - the set of modules required by this module
- `providers` - the set of providers defined in the module which have to be instantiated by the Nest injector

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

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

Feature (Child) Modules

Feature modules are modules that **organize the code** related to a specific feature, keeping code that belongs together **closely**. They can be **imported** by other modules.

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

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

Global Modules

Global modules are modules that **should be available everywhere** in the application context. They should be **registered only once** and **not** be **re-exported** from any other module.

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

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

Dynamic Modules

Dynamic modules are modules that are **not** known at compile time and are **created** dynamically based on runtime information. They are declared using a **provider-based** syntax.

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

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

Shared Modules

Shared modules are modules that **export** providers that are widely used and **re-exported** by other modules.

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

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

Core Module (Singleton)

The core module is a special module that is **available everywhere** in the application context. It should be **registered only once** and **not** be **re-exported** from any other module.

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';

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

Middleware

Source: https://docs.nestjs.com/middleware

Middleware are functions that have access to the **request** and **response** object. Middleware functions can **execute** any code, **make changes** to the request and the response objects, **end** the request-response cycle, and **call** the next middleware function in the stack.

The difference to interceptors is, that middleware functions have **full access** to the **request** and **response** objects, whereas interceptors are **not** allowed to **modify** the **request** object.

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

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

Applying Middleware

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

@Module({
  imports: [CatsModule],
})

// The `configure()` method takes a `consumer` argument
// which defines the set of routes which are handled by the middleware.
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer.apply(LoggerMiddleware).forRoutes('cats');
  }
}
```

`forRoutes()`

The `forRoutes()` method can take:

- a **string** representing a **path** (or paths)
- a **class** representing a **controller**
- an **object** representing a **specific route configuration**

Multiple Middleware

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

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

Excluding Routes

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

@Module({
  imports: [CatsModule],
})

// The `configure()` method takes a `consumer` argument
// which defines the set of routes which are handled by the middleware.
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .exclude(
        { path: 'cats', method: RequestMethod.GET },
        { path: 'cats', method: RequestMethod.POST },
        'cats/(.*)',
      )
      .forRoutes(CatsController);
  }
}
```

Functional Middleware

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  resolve(...args: any[]): MiddlewareFunction {
    return (req, res, next) => {
      console.log('Request...');
      next();
    };
  }
}
```

Global Middleware

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

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

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

Further Details

```
import { Module } from '@nestjs/common';
import { APP_GUARD } from '@nestjs/core';
import { RolesGuard } from './common/guards/roles.guard';
import { APP_FILTER } from '@nestjs/core';
import { HttpExceptionFilter } from './common/exception-filters/http-exception.filter';
import { APP_INTERCEPTOR } from '@nestjs/core';
import { LoggingInterceptor } from './common/interceptors/logging.interceptor';
import { APP_PIPE } from '@nestjs/core';
import { ValidationPipe } from './common/pipes/validation.pipe';
```

The `APP_GUARD` token is used to provide a guard that is applied globally to all controllers.

RolesGuard is a provider with `provide` and `useClass` properties. The value of the `useClass` property is the class of the guard. The `provide` property is optional and defaults to the class of the guard. The `provide` property is used to inject the guard into other providers.

The `APP_FILTER` token is used to provide a filter that is applied globally to all controllers.

HttpExceptionFilter is a provider with `provide` and `useClass` properties. The value of the `useClass` property is the class of the filter. The `provide` property is optional and defaults to the class of the filter. The `provide` property is used to inject the filter into other providers.

The `APP_INTERCEPTOR` token is used to provide an interceptor that is applied globally to all controllers.

LoggingInterceptor is a provider with `provide` and `useClass` properties. The value of the `useClass` property is the class of the interceptor. The `provide` property is optional and defaults to the class of the interceptor. The `provide` property is used to inject the interceptor into other providers.

The `APP_PIPE` token is used to provide a pipe that is applied globally to all controllers.

ValidationPipe is a provider with `provide` and `useClass` properties. The value of the `useClass` property is the class of the pipe. The `provide` property is optional and defaults to the class of the pipe. The `provide` property is used to inject the pipe into other providers.

Exception Filters

Source: https://docs.nestjs.com/exception-filters

Exception filters are used to **catch** exceptions **globally** and **transform** them into **proper responses**.

You can compare them with Angular's **interceptors**. The difference is that **interceptors** are called **before** a handler is executed, and **exception filters** are called **after** an exception is thrown from the handler. Exception filters should implement the `ExceptionFilter` interface.

```
import {
  ExceptionFilter,
  Catch,
  ArgumentsHost,
  HttpException,
} from '@nestjs/common';
import { Request, Response } from 'express';

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

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

```
import { Module } from '@nestjs/common';
import { APP_FILTER } from '@nestjs/core';
import { HttpExceptionFilter } from './common/exception-filters/http-exception.filter';

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

Out of the box, this action is performed by a built-in **global exception filter**, which handles exceptions of type `HttpException` (and subclasses of it). When an exception is **unrecognized** (is

neither `HttpException` nor a class that inherits from `HttpException`), the built-in exception filter generates the following default JSON response:

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

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

Throwing Standard Exceptions

Nest provides a set of standard exceptions that are mapped to HTTP response status codes. For example, the `NotFoundException` exception is mapped to the `404` status code. When this exception is thrown, Nest automatically responds with the following JSON:

```
{
  "statusCode": 404,
  "message": "Not Found"
}
```

The following table describes the built-in exceptions and their corresponding HTTP status codes:

Exception Name	Status Code
`BadRequestException`	`400`
`UnauthorizedException`	`401`
`NotFoundException`	`404`
`ForbiddenException`	`403`
`NotAcceptableException`	`406`
`RequestTimeoutException`	`408`
`ConflictException`	`409`
`GoneException`	`410`

Exception Name	Status Code
`PayloadTooLargeException`	`413`
`UnsupportedMediaTypeException`	`415`
`UnprocessableEntityException`	`422`
`InternalServerErrorException`	`500`
`NotImplementedException`	`501`
`BadGatewayException`	`502`
`ServiceUnavailableException`	`503`
`GatewayTimeoutException`	`504`

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

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

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

The `HttpException` class takes two arguments: a `response` and a `status` code. The `response` argument can be either a string or an object. If it's an object, it will be stringified and returned as a JSON response. The `status` argument is an HTTP response status code.

The `HttpException` class is a subclass of the built-in `Error` class. This means that you can pass an error stack to the `response` argument. For example:

```
@Get()
async findAll() {
  throw new HttpException({
    status: HttpStatus.FORBIDDEN,
    error: 'This is a custom message',
  }, HttpStatus.FORBIDDEN);
}
```

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

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

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

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

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

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

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

```
{
  "statusCode": 403,
  "error": "This is a custom message"
}
```

Custom Exceptions

You can create your own custom exceptions by extending the built-in `HttpException` class. For example, let's create a `ForbiddenException` class:

```
import { HttpException, HttpStatus } from '@nestjs/common';

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

Now, we can throw this exception from the `CatsController`:

```
@Get()
async findAll() {
  throw new ForbiddenException();
}
```

Built-in HTTP exceptions

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

- `BadRequestException`
- `UnauthorizedException`
- `NotFoundException`
- `ForbiddenException`
- `NotAcceptableException`
- `RequestTimeoutException`
- `ConflictException`
- `GoneException`
- `HttpVersionNotSupportedException`

- `PayloadTooLargeException`
- `UnsupportedMediaTypeException`
- `UnprocessableEntityException`
- `InternalServerErrorException`
- `NotImplementedException`
- `ImATeapotException`
- `MethodNotAllowedException`
- `BadGatewayException`
- `ServiceUnavailableException`
- `GatewayTimeoutException`
- `PreconditionFailedException`

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

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

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

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

Full Control with Exception Filters

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

Let's create an exception filter that is responsible for catching exceptions which are an instance of the `HttpException` class, and implementing custom response logic for them. To do this, we'll need to access the underlying platform `Request` and `Response` objects. We'll access the `Request`

object so we can pull out the original `url` and include that in the logging information. We'll use the `Response` object to take direct control of the response that is sent, using the `response.json()` method.

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

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

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

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

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

The `catch()` method is the heart of the exception filter. It receives two arguments:

- `exception` - the exception instance being handled
- `host` - the arguments host object

The `host` object provides access to the underlying platform `Request` and `Response` objects. The `switchToHttp()` method returns an object with two methods: `getRequest()` and `getResponse()`. These methods return the underlying `Request` and `Response` objects respectively.

The `exception` object is an instance of the `HttpException` class. This class has a `getStatus()` method that returns the HTTP status code of the exception. We use this to set the status code of the response.

Finally, we use the `response.json()` method to send a JSON response back to the client. We use the `request.url` property to include the original URL in the response.

Arguments Host

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

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

Binding Filters

Now that we've created our exception filter, we need to bind it to the application. We can do this in several ways. The first way is to bind it globally, using the `APP_FILTER` provider:

`APP_FILTER` Provider (Module-scoped Filter)

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

```
import { Module } from '@nestjs/common';
import { APP_FILTER } from '@nestjs/core';
import { HttpExceptionFilter } from './common/exception-filters/http-exception.filter';

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

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

The `APP_FILTER` token is used to provide a filter that is applied globally to all controllers.

HttpExceptionFilter is a provider with `provide` and `useClass` properties. The value of the `useClass` property is the class of the filter. The `provide` property is optional and defaults to the class of the filter. The `provide` property is used to inject the filter into other providers.

The second way is to bind it to a particular controller or set of controllers using the `@UseFilters()` decorator:

`@UseFilters()` Decorator (Controller-scoped Filter)

```
import { Controller, Get, UseFilters } from '@nestjs/common';
import { HttpExceptionFilter } from './common/exception-filters/http-exception.filter';

@Controller('cats')
@UseFilters(new HttpExceptionFilter())
export class CatsController {}
```

The `@UseFilters()` decorator takes a single argument, which is the exception filter instance. This is the same instance that would be provided by the `APP_FILTER` provider.

The third way is to bind it to a particular method or set of methods using the `@UseFilters()` decorator:

`@UseFilters()` Decorator (Method-scoped Filter)

```
import { Controller, Get, UseFilters } from '@nestjs/common';
import { HttpExceptionFilter } from './common/exception-filters/http-exception.filter';

@Controller('cats')
export class CatsController {
  @Get()
  @UseFilters(new HttpExceptionFilter())
  async findAll() {
    throw new ForbiddenException();
  }
}
```

Application-scoped Filter

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

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

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

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

Catch Everything

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

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

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

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

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

    const ctx = host.switchToHttp();

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

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

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

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

> correctly handle the bound type.

Inheritance

Exception filters can be inherited. This means that you can create a base exception filter that implements some common logic, and then extend it to create more specialized exception filters. For example, let's create a `HttpExceptionFilter` that implements the logic for handling `HttpException` exceptions, and then extend it to create a `ForbiddenExceptionFilter` that handles `ForbiddenException` exceptions:

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

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    // ...
  }
}
```

```
import { ForbiddenException } from './forbidden.exception';
import { HttpExceptionFilter } from './http-exception.filter';

@Catch(ForbiddenException)
export class ForbiddenExceptionFilter extends HttpExceptionFilter {
  catch(exception: ForbiddenException, host: ArgumentsHost) {
    super.catch(exception, host);
  }
}
```

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

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

```
import { Catch, ArgumentsHost } from '@nestjs/common';
import { BaseExceptionFilter } from '@nestjs/core';

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

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

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

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

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

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

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

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

The second method is to use the APP_FILTER token as shown further above.

Exception Filters vs Pipes

Exception filters and pipes are **different** things. Pipes are about **data transformation** and should be used to **transform** the **input** data to the **desired output**. Exception filters are about **handling exceptions** and should be used to **create** a **proper response**.

Pipes

Source: https://docs.nestjs.com/pipes

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

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

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

Pipes have two typical use cases:

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

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

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

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

Built-in Pipes

Nest provides a set of **built-in pipes** that cover most of the use-cases for the applications. These pipes are available from the `@nestjs/common` package. Let's take a look at each of them.

ValidationPipe

The `ValidationPipe` uses the class-validator.

```
import { Controller, Get, Post, Body } from '@nestjs/common';
import { CreateCatDto } from './create-cat.dto';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';
import { ValidationPipe } from './common/pipes/validation.pipe';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

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

  @Get()
  async findAll(): Promise<Cat[]> {
    return this.catsService.findAll();
  }
}
```

ParseIntPipe

The `ParseIntPipe` converts a string to an integer value. If the string is not a number, an exception will be thrown (HTTP response status code `400 Bad Request`).

```
import {
  Controller,
  Get,
  Param,
  ParseIntPipe,
} from '@nestjs/common';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

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

ParseBoolPipe

The `ParseBoolPipe` converts a string to a boolean value. If the string is not a boolean, an exception will be thrown (HTTP response status code `400 Bad Request`).

```
import {
  Controller,
  Get,
  Param,
  ParseBoolPipe,
} from '@nestjs/common';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Get(':id')
  async findOne(
    @Param('id', ParseBoolPipe) id: boolean,
  ): Promise<Cat> {
    return this.catsService.findOne(id);
  }
}
```

ParseArrayPipe

The `ParseArrayPipe` converts a string to an array. If the string is not an array, an exception will be thrown (HTTP response status code `400 Bad Request`).

```
import {
  Controller,
  Get,
  Param,
  ParseArrayPipe,
} from '@nestjs/common';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Get(':id')
  async findOne(
    @Param('id', ParseArrayPipe) id: number[],
  ): Promise<Cat> {
    return this.catsService.findOne(id);
  }
}
```

ParseFloatPipe

The `ParseFloatPipe` converts a string to a float value. If the string is not a number, an exception will be thrown (HTTP response status code `400 Bad Request`).

```
import {
  Controller,
  Get,
  Param,
  ParseFloatPipe,
} from '@nestjs/common';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Get(':id')
  async findOne(
    @Param('id', ParseFloatPipe) id: number,
  ): Promise<Cat> {
    return this.catsService.findOne(id);
  }
}
```

ParseUUIDPipe

The `ParseUUIDPipe` converts a string to a UUID value. If the string is not a UUID, an exception will be thrown (HTTP response status code `400 Bad Request`).

```
import {
  Controller,
  Get,
  Param,
  ParseUUIDPipe,
} from '@nestjs/common';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Get(':id')
  async findOne(
    @Param('id', ParseUUIDPipe) id: number,
  ): Promise<Cat> {
    return this.catsService.findOne(id);
  }
}
```

ParseEnumPipe

The `ParseEnumPipe` converts a string to a enum value. If the string is not a enum, an exception will be thrown (HTTP response status code `400 Bad Request`).

```
import {
  Controller,
  Get,
  Param,
  ParseEnumPipe,
} from '@nestjs/common';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Get(':id')
  async findOne(
    @Param('id', ParseEnumPipe) id: number,
  ): Promise<Cat> {
    return this.catsService.findOne(id);
  }
}
```

DefaultValuePipe

The `DefaultValuePipe` sets a default value for the parameter. If the parameter is not present in the request, the default value will be used. If the parameter is present, the value from the request will be used.

```
import {
  Controller,
  Get,
  Param,
  DefaultValuePipe,
} from '@nestjs/common';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Get(':id')
  async findOne(
    @Param('id', new DefaultValuePipe(0)) id: number,
  ): Promise<Cat> {
    return this.catsService.findOne(id);
  }
}
```

ParseFilePipe

The `ParseFilePipe` converts a file to a `multer` file object. If the file is not a file, an exception will be thrown (HTTP response status code `400 Bad Request`).

```
import {
  Controller,
  Post,
  UploadedFile,
  UseInterceptors,
} from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';

@Controller('cats')
export class CatsController {
  @Post('upload')
  @UseInterceptors(FileInterceptor('file', { dest: './uploads' }))
  uploadFile(@UploadedFile() file) {
    console.log(file);
  }
}
```

Custom Pipes

You can create your own custom pipes by implementing the `PipeTransform` interface. The `PipeTransform` interface defines a single `transform()` method that you must implement. This method takes two arguments:

- `value` - the value passed into the method
- `metadata` - additional information about the value

The `transform()` method should return the transformed value. If the value cannot be transformed, you should throw an exception. The exception will be handled by Nest and an appropriate HTTP response will be sent back to the client.

Let's create a custom pipe that transforms a string to an integer value. If the string is not a number, we'll throw an exception.

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

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

The `ParseIntPipe` class implements the `PipeTransform` interface. The `PipeTransform` interface is a generic interface that takes two type arguments:

- `T` - the type of the value being transformed
- `R` - the type of the transformed value

In this case, we're transforming a string to a number, so `T` is `string` and `R` is `number`.

The `transform()` method takes two arguments:

- `value` - the value passed into the method
- `metadata` - additional information about the value

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

- `type` - the type of the value being transformed
- `metatype` - the type of the object that contains the value being transformed
- `data` - the value being transformed

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

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

The `transform()` method should return the transformed value. If the value cannot be transformed, you should throw an exception. The exception will be handled by Nest and an appropriate HTTP response will be sent back to the client.

The `transform()` method is where you implement your custom logic. In this case, we're using the built-in `parseInt()` function to convert the string to a number. If the string is not a number, `parseInt()` will return `NaN`. We check for this and throw an exception if it occurs.

> **Hint** The `metadata` argument is an instance of the `ArgumentMetadata` interface. This interface defines two properties: `type` and `metatype`. The `type` property is the type of the value being transformed. The `metatype` property is the type of the object that contains the value being transformed. For example, if the value is a property of a class, the `metatype` property will be the type of that class.

Schema-based Validation

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

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

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

```
// create-cat.dto.ts

export class CreateCatDto {
  name: string;
  age: number;
  breed: string;
}
```

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

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

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

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

Binding Pipes (Overview)

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

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

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

For example, assume the route is called like:

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

Nest will throw an exception like this:

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

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

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

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

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

For example with a query string parameter:

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

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

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

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

Above we've seen examples of binding the various `Parse*` family of built-in pipes.

Binding Pipes (Other Techniques)

Now that we've created our custom pipe, we need to bind it to the application. We can do this in several ways. The first way is to bind it globally, using the `APP_PIPE` provider:

`APP_PIPE` Provider (Module-scoped Pipe)

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

```
import { Module } from '@nestjs/common';
import { APP_PIPE } from '@nestjs/core';
import { ValidationPipe } from './common/pipes/validation.pipe';

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

`@UsePipes()` Decorator (Controller-scoped Pipe)

```
import {
  Controller,
  Get,
  Post,
  Body,
  UsePipes,
} from '@nestjs/common';
import { CreateCatDto } from './create-cat.dto';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';
import { ValidationPipe } from './common/pipes/validation.pipe';

@Controller('cats')
@UsePipes(new ValidationPipe())
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

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

  @Get()
  async findAll(): Promise<Cat[]> {
    return this.catsService.findAll();
  }
}
```

`@UsePipes()` Decorator (Method-scoped Pipe)

```
import {
  Controller,
  Get,
  Post,
  Body,
  UsePipes,
} from '@nestjs/common';
import { CreateCatDto } from './create-cat.dto';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';
import { ValidationPipe } from './common/pipes/validation.pipe';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Post()
  @UsePipes(new ValidationPipe())
  async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
  }

  @Get()
  async findAll(): Promise<Cat[]> {
    return this.catsService.findAll();
  }
}
```

Application-scoped Pipe

To create a application-scoped pipe, you would do the following:

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

> **Warning** The `useGlobalPipes()` method does not set up pipes for gateways or hybrid applications.

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

Binding Multiple Pipes

You can bind multiple pipes to a single route handler. In this case, the pipes will be executed in the order they are bound. For example, let's bind the `ValidationPipe` and the `ParseIntPipe` to the `findOne()` method of the `CatsController`:

```
import {
  Controller,
  Get,
  Param,
  ParseIntPipe,
  UsePipes,
  ValidationPipe,
} from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get(':id')
  @UsePipes(ValidationPipe, ParseIntPipe)
  async findOne(@Param('id') id: number) {
    return this.catsService.findOne(id);
  }
}
```

Binding Multiple Pipes (with options)

You can bind multiple pipes to a single route handler. In this case, the pipes will be executed in the order they are bound. For example, let's bind the `ValidationPipe` and the `ParseIntPipe` to the `findOne()` method of the `CatsController`:

```
import {
  Controller,
  Get,
  Param,
  ParseIntPipe,
  UsePipes,
  ValidationPipe,
} from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get(':id')
  @UsePipes(
    new ValidationPipe({ transform: true }),
    new ParseIntPipe({
      errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE,
    }),
  )
  async findOne(@Param('id') id: number) {
    return this.catsService.findOne(id);
  }
}
```

Guards

Source: https://docs.nestjs.com/guards

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

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

> Guards are executed **after** each middleware, but **before** any pipe and interceptor.

Guards have a single method `canActivate()` which returns a boolean or a `Promise<boolean>`. If any guard returns `false`, the execution flow will be stopped and the request will be rejected. If `true`, Nest will proceed with executing the route handler.

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

Authorization Guards

Authorization guards are used to control access to resources based on the roles of the user. For example, you might want to restrict access to certain routes to only users with the role of **admin**. Authorization guards are defined as classes annotated with the `@Injectable()` decorator that implement the `CanActivate` interface.

```
import {
  Injectable,
  CanActivate,
  ExecutionContext,
} from '@nestjs/common';
import { Observable } from 'rxjs';

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

```
import {
  Injectable,
  CanActivate,
  ExecutionContext,
} from '@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
  }
}
```

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

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

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

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

Guards have a **single responsibility**. They determine whether a given request will be handled by the route handler or not, depending on certain conditions (like permissions, roles, ACLs, etc.) present at run-time. This is often referred to as **authorization**. Authorization (and its cousin, **authentication**, with which it usually collaborates) has typically been handled by middleware in traditional Express applications. Middleware is a fine choice for authentication, since things like token validation and attaching properties to the `request` object are not strongly connected with a particular route context (and its metadata).

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

Authorization guard

Source https://docs.nestjs.com/guards#authorization-guard

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

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

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
  }
}
```

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

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

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

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

Execution context

Source https://docs.nestjs.com/guards#execution-context

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

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

Role based authentication

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

```
// roles.guard.ts

import {
  Injectable,
  CanActivate,
  ExecutionContext,
} from '@nestjs/common';
import { Observable } from 'rxjs';

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

Binding guards

Source: https://docs.nestjs.com/guards#binding-guards

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

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

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

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

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

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

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

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

> **Notice** In the case of hybrid apps the `useGlobalGuards()` method doesn't set up guards for gateways and micro services by default (see Hybrid application for information on how to change this behavior). For "standard" (non-hybrid) microservice apps, `useGlobalGuards()` does mount the guards globally.

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

```
// app.module.ts

import { Module } from '@nestjs/common';
import { APP_GUARD } from '@nestjs/core';

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

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

Setting roles per handler

Source https://docs.nestjs.com/guards#setting-roles-per-handler

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

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

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

```
// roles.decorator.ts
import { Reflector } from '@nestjs/core';

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

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

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

```
// cats.controller.ts

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

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

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

Putting it all together

Source: https://docs.nestjs.com/guards#putting-it-all-together

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

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

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

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

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

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

> cycle.

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

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

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

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

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

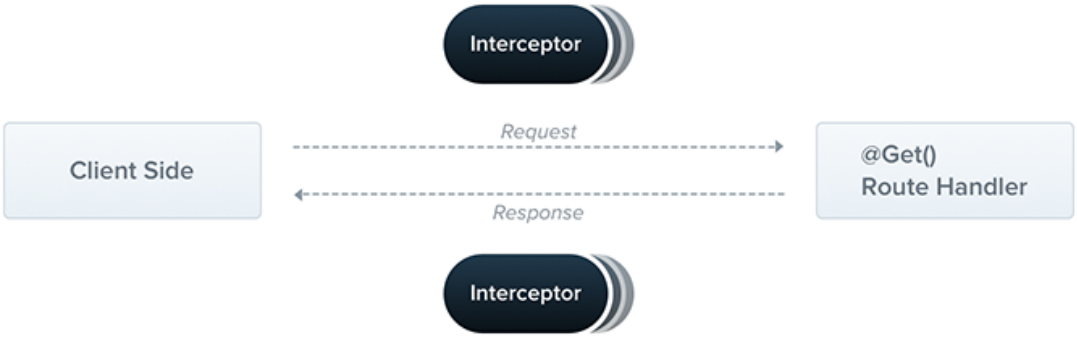
Any exception thrown by a guard will be handled by the exceptions layer (global exceptions filter and any exceptions filters that are applied to the current context).

> **Hint** If you are looking for a real-world example on how to implement authorization, check this chapter.

Interceptors

Source: https://docs.nestjs.com/interceptors

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



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

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

Basics

Source: https://docs.nestjs.com/interceptors#basics

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

Execution context

Source https://docs.nestjs.com/interceptors#execution-context

By extending `ArgumentsHost`, `ExecutionContext` also adds several new helper methods that provide additional details about the current execution process. These details can be helpful in

building more generic interceptors that can work across a broad set of controllers, methods, and execution contexts. Learn more about `ExecutionContext` here.

Call handler

Source: https://docs.nestjs.com/interceptors#call-handler

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

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

Consider, for example, an incoming `POST /cats` request. This request is destined for the `create()` handler defined inside the `CatsController`. If an interceptor which does not call the `handle()` method is called anywhere along the way, the `create()` method won't be executed. Once `handle()` is called (and its `Observable` has been returned), the `create()` handler will be triggered. And once the response stream is received via the `Observable`, additional operations can be performed on the stream, and a final result returned to the caller.

Aspect Interception

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

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

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

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

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

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

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

Binding interceptors

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

Below, we set up a controller-scoped interceptor using the @UseInterceptors() decorator. This decorator may take a single argument, or a comma-separated list of arguments. This lets you easily apply the appropriate set of interceptors with one declaration.

```
@Controller('cats')
@UseInterceptors(LoggingInterceptor)
export class CatsController {}
```

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

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

```
@Controller('cats')
@UseInterceptors(new LoggingInterceptor())
export class CatsController {}
```

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

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

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

> **Notice** In the case of hybrid apps the `useGlobalInterceptors()` method doesn't set up interceptors for gateways and micro services by default (see Hybrid application for information on how to change this behavior). For "standard" (non-hybrid) microservice apps, `useGlobalInterceptors()` does mount the interceptors globally.

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

```
// app.module.ts

import { Module } from '@nestjs/common';
import { APP_INTERCEPTOR } from '@nestjs/core';

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

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

Interceptor composition

Source: https://docs.nestjs.com/interceptors#interceptor-composition

Interceptors can be composed. This means that you can apply multiple interceptors to a single handler. The interceptors will then be executed in the order they are set up (from left to right). For example, let's bind the `LoggingInterceptor` and the `TransformInterceptor` to the `findOne()` method of the `CatsController` :

```
@Controller('cats')
@UseInterceptors(LoggingInterceptor, TransformInterceptor)
export class CatsController {
  @Get(':id')
  async findOne(@Param('id') id: number) {
    return this.catsService.findOne(id);
  }
}
```

Interceptor context

Source: https://docs.nestjs.com/interceptors#interceptor-context

Interceptors have access to a `ExecutionContext` instance, and thus know exactly what's going to be executed next. They're designed, much like exception filters, pipes, and guards, to let you interpose processing logic at exactly the right point in the request/response cycle, and to do so declaratively. This helps keep your code DRY and declarative.

Interceptor exclusion

Source: https://docs.nestjs.com/interceptors#interceptor-exclusion

You can exclude interceptors from being applied to specific routes. To do so, use the `@ExcludeInterceptors()` decorator. This decorator may take a single argument, or a comma-separated list of arguments. This lets you easily exclude the appropriate set of interceptors with one declaration.

```
@Controller('cats')
@UseInterceptors(LoggingInterceptor)
export class CatsController {
  @Get(':id')
  @ExcludeInterceptors(TransformInterceptor)
  async findOne(@Param('id') id: number) {
    return this.catsService.findOne(id);
  }
}
```

Interceptor inheritance

Source: https://docs.nestjs.com/interceptors#interceptor-inheritance

Interceptors can be inherited. This means that you can apply an interceptor to a controller, and it will be automatically applied to all the handlers within that controller. For example, let's bind the `LoggingInterceptor` to the `CatsController`:

```
@UseInterceptors(LoggingInterceptor)
@Controller('cats')
export class CatsController {
  @Get(':id')
  async findOne(@Param('id') id: number) {
    return this.catsService.findOne(id);
  }
}
```

In the example above, the `LoggingInterceptor` will be applied to the `findOne()` method. This is because the `findOne()` method is defined within the `CatsController` class. If you want to exclude the interceptor from a specific handler, you can use the `@ExcludeInterceptors()` decorator.

Interceptor ordering

Source: https://docs.nestjs.com/interceptors#interceptor-ordering

Interceptors are executed in the order they are defined. The order of global interceptors is not guaranteed. The order of controller-scoped interceptors is also not guaranteed. The order of method-scoped interceptors is guaranteed.

Interceptor interface

Source: https://docs.nestjs.com/interceptors#interceptor-interface

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

```
export interface NestInterceptor<T, R> {
  intercept(
    context: ExecutionContext,
    next: CallHandler<T>,
  ): Observable<R> | Promise<Observable<R>>;
}
```

Interceptor as middleware

Source: https://docs.nestjs.com/interceptors#interceptor-as-middleware

Interceptors can be used as middleware. This means that you can apply an interceptor to a controller, and it will be automatically applied to all the handlers within that controller. For example, let's bind the `LoggingInterceptor` to the `CatsController`:

```
@UseInterceptors(LoggingInterceptor)
@Controller('cats')
export class CatsController {
  @Get(':id')
  async findOne(@Param('id') id: number) {
    return this.catsService.findOne(id);
  }
}
```

Response mapping

Source: https://docs.nestjs.com/interceptors#response-mapping

Interceptors can be used to transform the result returned from a function. For example, let's create a `TransformInterceptor` that will transform the result returned from the `CatsController`:

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

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

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

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

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

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

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

```
// transform.interceptor.ts
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  CallHandler,
} from '@nestjs/common';

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


// main.ts
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalInterceptors(new TransformInterceptor());
  await app.listen(3000);
}
bootstrap();


// cats.controller.ts
@Get()
async findAll(): Promise<any[]> {
  return [null];
}


[""]
```

Stream Overriding

Source: https://docs.nestjs.com/interceptors#stream-overriding

Interceptors can be used to completely override a function depending on specific conditions (e.g., for caching purposes). For example, let's create a `CacheInterceptor` that will cache the response for 5 seconds:

```
// cache.interceptor.ts
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  CallHandler,
} from '@nestjs/common';

@Injectable()
export class CacheInterceptor implements NestInterceptor {
  intercept(
    context: ExecutionContext,
    next: CallHandler,
  ): Observable<any> {
    const isCached = true;
    if (isCached) {
      return of([]);
    }
    return next.handle();
  }
}
```

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

More Operators

Source: https://docs.nestjs.com/interceptors#more-operators

Interceptors can be used to extend the basic function behavior. For example, let's create a `TimeoutInterceptor` that will throw an exception if the request takes longer than 5000 milliseconds:

```
// timeout.interceptor.ts
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  CallHandler,
} from '@nestjs/common';
import { Observable, TimeoutError } from 'rxjs';
import { timeout } from 'rxjs/operators';

@Injectable()
export class TimeoutInterceptor implements NestInterceptor {
  intercept(
    context: ExecutionContext,
    next: CallHandler,
  ): Observable<any> {
    return next.handle().pipe(timeout(5000));
  }
}
```

Custom Decorators

Source: https://docs.nestjs.com/custom-decorators

Decorators are a language feature that allow you to annotate or modify classes and class members (properties, methods, and so on) at compile time. Decorators use the form `@expression`, where `expression` must evaluate to a function that will be called at runtime with information about the decorated declaration.

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

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

Decorator Factories

Source: https://docs.nestjs.com/custom-decorators#decorator-factories

A decorator factory is simply a function that returns the expression that will be called by the decorator at runtime. For example, let's create a `@Roles()` decorator factory:

```
// roles.decorator.ts
import { SetMetadata } from '@nestjs/common';

export const Roles = (...roles: string[]) =>
  SetMetadata('roles', roles);
```

Decorator Composition

Source: https://docs.nestjs.com/custom-decorators#decorator-composition

Decorators can be composed. This means that you can apply multiple decorators to a single class, method, or property. The decorators will then be executed in the order they are set up (from left to right). For example, let's bind the `@Roles()` and the `@UseGuards()` decorators to the `CatsController`:

```
@Roles('admin')
@UseGuards(AuthGuard, RolesGuard)
export class CatsController {}
```

Passing Data

Source: https://docs.nestjs.com/custom-decorators#passing-data

Decorators can take arguments. For example, let's create a `@Roles()` decorator that takes a variable number of string arguments:

```
// roles.decorator.ts
import { SetMetadata } from '@nestjs/common';

export const Roles = (...roles: string[]) =>
  SetMetadata('roles', roles);
```

Decorator Internals

Source: https://docs.nestjs.com/custom-decorators#decorator-internals

Decorators are just functions, and like any function, they can be synchronous or asynchronous. Nest supports both types of decorators. Synchronous decorators are the most common, and are used to modify or replace the target declaration. Asynchronous decorators are used to perform additional processing after the target declaration has been evaluated.

Decorator as middleware

Source: https://docs.nestjs.com/custom-decorators#decorator-as-middleware

Decorators can be used as middleware. This means that you can apply a decorator to a controller, and it will be automatically applied to all the handlers within that controller. For example, let's bind the `@Roles()` decorator to the `CatsController` :

```
@Roles('admin')
@Controller('cats')
export class CatsController {}
```

In the example above, the `@Roles()` decorator will be applied to all the handlers defined within the `CatsController` class. If you want to exclude the decorator from a specific handler, you can use the `@ExcludeRoles()` decorator.

Decorator inheritance

Source: https://docs.nestjs.com/custom-decorators#decorator-inheritance

Decorators can be inherited. This means that you can apply a decorator to a controller, and it will be automatically applied to all the handlers within that controller. For example, let's bind the `@Roles()` decorator to the `CatsController` :

```
@Roles('admin')
@Controller('cats')
export class CatsController {}
```

In the example above, the `@Roles()` decorator will be applied to all the handlers defined within the `CatsController` class. If you want to exclude the decorator from a specific handler, you can use the `@ExcludeRoles()` decorator.

Decorator ordering

Source: https://docs.nestjs.com/custom-decorators#decorator-ordering

Decorators are executed in the order they are defined. The order of global decorators is not guaranteed. The order of controller-scoped decorators is also not guaranteed. The order of method-scoped decorators is guaranteed.

Decorator interface

Source: https://docs.nestjs.com/custom-decorators#decorator-interface

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

```
export interface NestInterceptor<T, R> {
  intercept(
    context: ExecutionContext,
    next: CallHandler<T>,
  ): Observable<R> | Promise<Observable<R>>;
}
```

Working with Pipes

Source: https://docs.nestjs.com/custom-decorators#working-with-pipes

Decorators can be used to apply pipes. For example, let's create a `@UsePipes()` decorator that will apply the `ValidationPipe` to the `CatsController` :

```
// use-pipes.decorator.ts
import { applyDecorators, UsePipes } from '@nestjs/common';
import { ValidationPipe } from '@nestjs/common';

export function UseValidationPipe() {
  return applyDecorators(UsePipes(new ValidationPipe()));
}
```

```
// cats.controller.ts
@UseValidationPipe()
@Controller('cats')
export class CatsController {}
```

Working with Guards

Source: https://docs.nestjs.com/custom-decorators#working-with-guards

Decorators can be used to apply guards. For example, let's create a `@UseGuards()` decorator that will apply the `AuthGuard` to the `CatsController`:

```
// use-guards.decorator.ts
import { applyDecorators, UseGuards } from '@nestjs/common';
import { AuthGuard } from '@nestjs/common';

export function UseAuthGuard() {
  return applyDecorators(UseGuards(AuthGuard));
}
```

```
// cats.controller.ts
@UseAuthGuard()
@Controller('cats')
export class CatsController {}
```

Working with Interceptors

Source: https://docs.nestjs.com/custom-decorators#working-with-interceptors

Decorators can be used to apply interceptors. For example, let's create a `@UseInterceptors()` decorator that will apply the `LoggingInterceptor` to the `CatsController`:

```
// use-interceptors.decorator.ts
import { applyDecorators, UseInterceptors } from '@nestjs/common';
import { LoggingInterceptor } from '@nestjs/common';

export function UseLoggingInterceptor() {
  return applyDecorators(UseInterceptors(LoggingInterceptor));
}
```

```
// cats.controller.ts
@UseLoggingInterceptor()
@Controller('cats')
export class CatsController {}
```

Working with Filters

Source: https://docs.nestjs.com/custom-decorators#working-with-filters

Decorators can be used to apply filters. For example, let's create a `@UseFilters()` decorator that will apply the `HttpExceptionFilter` to the `CatsController` :

```
// use-filters.decorator.ts
import { applyDecorators, UseFilters } from '@nestjs/common';
import { HttpExceptionFilter } from '@nestjs/common';

export function UseHttpExceptionFilter() {
  return applyDecorators(UseFilters(HttpExceptionFilter));
}
```

```
// cats.controller.ts
@UseHttpExceptionFilter()
@Controller('cats')
export class CatsController {}
```