Controllers

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

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

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

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

Routing

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

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

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

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

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

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

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

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

Standard (recommended)	Using this built-in method, when a request handler returns a JavaScript object or array, it will **automatically** be serialized to JSON. When it returns a JavaScript primitive type (e.g., `string`, `number`, `boolean`), however, Nest will send just the value without attempting to serialize it. This makes response handling simple: just return the value, and Nest takes care of the rest. Furthermore, the response's **status code** is always 200 by default, except for POST requests which use 201. We can easily change this behavior by adding the `@HttpCode(...)` decorator at a handler-level (see Status codes).
Library-specific	We can use the library-specific (e.g., Express) response object, which can be injected using the `@Res()` decorator in the method handler signature (e.g., `findAll(@Res() response)`). With this approach, you have the ability to use the native response handling methods exposed by that object. For example, with Express, you can construct responses using code like `response.status(200).send()`.

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

Request object

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

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

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

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

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

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

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

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

> info **Hint** To learn how to create your own custom decorators, visit this chapter.

Resources

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

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

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

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

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

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

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

that handles all of them.

Route wildcards

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

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

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

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

Status code

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

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

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

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

Headers

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

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

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

Redirection

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

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

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

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

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

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

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

Route parameters

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

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

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

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

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

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

Sub-Domain Routing

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

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

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

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

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

Scopes

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

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

Asynchronicity

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

> info **Hint** Learn more about `async / await` feature here

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

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

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

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

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

Request payloads

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

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

Let's create the `CreateCatDto` class:

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

It has only three basic properties. Thereafter we can use the newly created DTO inside the `CatsController`:

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

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

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

Handling errors

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

Full resource sample

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

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

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

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

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

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

```
  @Delete(':id')
  remove(@Param('id') id: string) {
    return `This action removes a #${id} cat`;
  }
}
@@switch
import { Controller, Get, Query, Post, Body, Put, Param, Delete, Bind }
from '@nestjs/common';

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

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

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

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

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

> info **Hint** Nest CLI provides a generator (schematic) that automatically generates **all the boilerplate code** to help us avoid doing all of this, and make the developer experience much simpler. Read more about this feature here.

Getting up and running

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

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

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

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

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

Library-specific approach

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

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

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

  @Get()
  findAll(@Res() res: Response) {
     res.status(HttpStatus.OK).json([]);
  }
}
@@switch
import { Controller, Get, Post, Bind, Res, Body, HttpStatus } from
'@nestjs/common';

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

```
  }

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

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

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

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

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