Execution context

Nest provides several utility classes that help make it easy to write applications that function across multiple application contexts (e.g., Nest HTTP server-based, microservices and WebSockets application contexts). These utilities provide information about the current execution context which can be used to build generic guards, filters, and interceptors that can work across a broad set of controllers, methods, and execution contexts.

We cover two such classes in this chapter: `ArgumentsHost` and `ExecutionContext`.

ArgumentsHost class

The `ArgumentsHost` class provides methods for retrieving the arguments being passed to a handler. It allows choosing the appropriate context (e.g., HTTP, RPC (microservice), or WebSockets) to retrieve the arguments from. The framework provides an instance of `ArgumentsHost`, typically referenced as a `host` parameter, in places where you may want to access it. For example, the `catch()` method of an exception filter is called with an `ArgumentsHost`instance.

`ArgumentsHost` simply acts as an abstraction over a handler's arguments. For example, for HTTP server applications (when `@nestjs/platform-express` is being used), the `host` object encapsulates Express's `[request, response, next]` array, where `request` is the request object, `response` is the response object, and `next` is a function that controls the application's request-response cycle. On the other hand, for GraphQL applications, the `host` object contains the `[root, args, context, info]` array.

Current application context

When building generic guards, filters, and interceptors which are meant to run across multiple application contexts, we need a way to determine the type of application that our method is currently running in. Do this with the `getType()` method of `ArgumentsHost`:

```
if (host.getType() === 'http') {
  // do something that is only important in the context of regular HTTP
requests (REST)
} else if (host.getType() === 'rpc') {
  // do something that is only important in the context of Microservice
requests
} else if (host.getType<GqlContextType>() === 'graphql') {
  // do something that is only important in the context of GraphQL
requests
}
```

> info **Hint** The `GqlContextType` is imported from the `@nestjs/graphql` package.

With the application type available, we can write more generic components, as shown below.

Host handler arguments

To retrieve the array of arguments being passed to the handler, one approach is to use the host object's `getArgs()` method.

```
const [req, res, next] = host.getArgs();
```

You can pluck a particular argument by index using the `getArgByIndex()` method:

```
const request = host.getArgByIndex(0);
const response = host.getArgByIndex(1);
```

In these examples we retrieved the request and response objects by index, which is not typically recommended as it couples the application to a particular execution context. Instead, you can make your code more robust and reusable by using one of the `host` object's utility methods to switch to the appropriate application context for your application. The context switch utility methods are shown below.

```
/**
 * Switch context to RPC.
 */
switchToRpc(): RpcArgumentsHost;
/**
 * Switch context to HTTP.
 */
switchToHttp(): HttpArgumentsHost;
/**
 * Switch context to WebSockets.
 */
switchToWs(): WsArgumentsHost;
```

Let's rewrite the previous example using the `switchToHttp()` method. The `host.switchToHttp()` helper call returns an `HttpArgumentsHost` object that is appropriate for the HTTP application context. The `HttpArgumentsHost` object has two useful methods we can use to extract the desired objects. We also use the Express type assertions in this case to return native Express typed objects:

```
const ctx = host.switchToHttp();
const request = ctx.getRequest<Request>();
const response = ctx.getResponse<Response>();
```

Similarly `WsArgumentsHost` and `RpcArgumentsHost` have methods to return appropriate objects in the microservices and WebSockets contexts. Here are the methods for `WsArgumentsHost`:

```
export interface WsArgumentsHost {
  /**
   * Returns the data object.
   */
  getData<T>(): T;
  /**
   * Returns the client object.
   */
  getClient<T>(): T;
}
```

Following are the methods for `RpcArgumentsHost`:

```
export interface RpcArgumentsHost {
  /**
   * Returns the data object.
   */
  getData<T>(): T;

  /**
   * Returns the context object.
   */
  getContext<T>(): T;
}
```

ExecutionContext class

`ExecutionContext` extends `ArgumentsHost`, providing additional details about the current execution process. Like `ArgumentsHost`, Nest provides an instance of `ExecutionContext` in places you may need it, such as in the `canActivate()` method of a guard and the `intercept()` method of an interceptor. It provides the following methods:

```
export interface ExecutionContext extends ArgumentsHost {
  /**
   * Returns the type of the controller class which the current handler
belongs to.
   */
  getClass<T>(): Type<T>;
  /**
   * Returns a reference to the handler (method) that will be invoked next
in the
   * request pipeline.
   */
  getHandler(): Function;
}
```

The `getHandler()` method returns a reference to the handler about to be invoked. The `getClass()` method returns the type of the `Controller` class which this particular handler belongs to. For example, in an HTTP context, if the currently processed request is a `POST` request, bound to the `create()` method on the `CatsController`, `getHandler()` returns a reference to the `create()` method and `getClass()` returns the `CatsController` **type** (not instance).

```
const methodKey = ctx.getHandler().name; // "create"
const className = ctx.getClass().name; // "CatsController"
```

The ability to access references to both the current class and handler method provides great flexibility. Most importantly, it gives us the opportunity to access the metadata set through either decorators created via `Reflector#createDecorator` or the built-in `@SetMetadata()` decorator from within guards or interceptors. We cover this use case below.

Reflection and metadata

Nest provides the ability to attach **custom metadata** to route handlers through decorators created via `Reflector#createDecorator` method, and the built-in `@SetMetadata()` decorator. In this section, let's compare the two approaches and see how to access the metadata from within a guard or interceptor.

To create strongly-typed decorators using `Reflector#createDecorator`, we need to specify the type argument. For example, let's create a `Roles` decorator that takes an array of strings as an argument.

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

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

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

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

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

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

To access the route's role(s) (custom metadata), we'll use the `Reflector` helper class again. `Reflector` can be injected into a class in the normal way:

```
@@filename(roles.guard)
@Injectable()
export class RolesGuard {
  constructor(private reflector: Reflector) {}
}
@@switch
@Injectable()
@Dependencies(Reflector)
export class CatsService {
  constructor(reflector) {
    this.reflector = reflector;
  }
}
```

> info **Hint** The `Reflector` class is imported from the `@nestjs/core` package.

Now, to read the handler metadata, use the `get()` method:

```
const roles = this.reflector.get(Roles, context.getHandler());
```

The `Reflector#get` method allows us to easily access the metadata by passing in two arguments: a decorator reference and a **context** (decorator target) to retrieve the metadata from. In this example, the specified **decorator** is `Roles` (refer back to the `roles.decorator.ts` file above). The context is provided by the call to `context.getHandler()`, which results in extracting the metadata for the currently processed route handler. Remember, `getHandler()` gives us a **reference** to the route handler function.

Alternatively, we may organize our controller by applying metadata at the controller level, applying to all routes in the controller class.

```
@@filename(cats.controller)
@Roles(['admin'])
@Controller('cats')
export class CatsController {}
@@switch
@Roles(['admin'])
@Controller('cats')
export class CatsController {}
```

In this case, to extract controller metadata, we pass `context.getClass()` as the second argument (to provide the controller class as the context for metadata extraction) instead of `context.getHandler()`:

```
@@filename(roles.guard)
const roles = this.reflector.get(Roles, context.getClass());
```

Given the ability to provide metadata at multiple levels, you may need to extract and merge metadata from several contexts. The `Reflector` class provides two utility methods used to help with this. These methods extract **both** controller and method metadata at once, and combine them in different ways.

Consider the following scenario, where you've supplied `Roles` metadata at both levels.

```
@@filename(cats.controller)
@Roles(['user'])
@Controller('cats')
export class CatsController {
  @Post()
  @Roles(['admin'])
  async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
  }
}
@@switch
@Roles(['user'])
@Controller('cats')
export class CatsController {}
  @Post()
  @Roles(['admin'])
  @Bind(Body())
  async create(createCatDto) {
    this.catsService.create(createCatDto);
  }
}
```

If your intent is to specify `'user'` as the default role, and override it selectively for certain methods, you would probably use the `getAllAndOverride()` method.

```
const roles = this.reflector.getAllAndOverride(Roles,
[context.getHandler(), context.getClass()]);
```

A guard with this code, running in the context of the `create()` method, with the above metadata, would result in `roles` containing `['admin']`.

To get metadata for both and merge it (this method merges both arrays and objects), use the `getAllAndMerge()` method:

```
const roles = this.reflector.getAllAndMerge(Roles, [context.getHandler(),
context.getClass()]);
```

This would result in `roles` containing `['user', 'admin']`.

For both of these merge methods, you pass the metadata key as the first argument, and an array of metadata target contexts (i.e., calls to the `getHandler()` and/or `getClass())` methods) as the second argument.

Low-level approach

As mentioned earlier, instead of using `Reflector#createDecorator`, you can also use the built-in `@SetMetadata()` decorator to attach metadata to a handler.

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

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

With the construction above, we attached the `roles` metadata (`roles` is a metadata key and `['admin']` is the associated value) to the `create()` method. While this works, it's not good practice to use `@SetMetadata()` directly in your routes. Instead, you can create your own decorators, as shown below:

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

export const Roles = (...roles: string[]) => SetMetadata('roles', roles);
@@switch
import { SetMetadata } from '@nestjs/common';

export const Roles = (...roles) => SetMetadata('roles', roles);
```

This approach is much cleaner and more readable, and somewhat resembles the `Reflector#createDecorator` approach. The difference is that with `@SetMetadata` you have more control over the metadata key and value, and also can create decorators that take more than one argument.

Now that we have a custom `@Roles()` decorator, we can use it to decorate the `create()` method.

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

To access the route's role(s) (custom metadata), we'll use the `Reflector` helper class again:

```
@@filename(roles.guard)
@Injectable()
export class RolesGuard {
  constructor(private reflector: Reflector) {}
}
@@switch
@Injectable()
@Dependencies(Reflector)
export class CatsService {
  constructor(reflector) {
    this.reflector = reflector;
  }
}
```

> info **Hint** The `Reflector` class is imported from the `@nestjs/core` package.

Now, to read the handler metadata, use the `get()` method.

```
const roles = this.reflector.get<string[]>('roles', context.getHandler());
```

Here instead of passing a decorator reference, we pass the metadata **key** as the first argument (which in our case is `'roles'`). Everything else remains the same as in the `Reflector#createDecorator` example.