Exception filters

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

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

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

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

Throwing standard exceptions

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

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

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

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

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

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

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

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

By default, the JSON response body contains two properties:

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

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

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

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

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

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

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

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

Custom exceptions

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

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

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

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

Built-in HTTP exceptions

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

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

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

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

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

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

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

Exception filters

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

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

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

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

    response
      .status(status)
      .json({
        statusCode: status,
        timestamp: new Date().toISOString(),
        path: request.url,
      });
  }
}
@@switch
import { Catch, HttpException } from '@nestjs/common';

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

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

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

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

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

Arguments host

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

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

Binding filters

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

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

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

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

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

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

In the example above, the `HttpExceptionFilter` is applied only to the single `create()` route handler, making it method-scoped. Exception filters can be scoped at different levels: method-scoped of the controller/resolver/gateway, controller-scoped, or global-scoped.
For example, to set up a filter as controller-scoped, you would do the following:

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

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

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

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

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

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

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

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

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

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

Catch everything

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

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

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

Inheritance

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

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

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

@Catch()
export class AllExceptionsFilter extends BaseExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    super.catch(exception, host);
  }
}
@@switch
import { Catch } from '@nestjs/common';
import { BaseExceptionFilter } from '@nestjs/core';

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

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

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

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

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

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const { httpAdapter } = app.get(HttpAdapterHost);
  app.useGlobalFilters(new AllExceptionsFilter(httpAdapter));

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

The second method is to use the `APP_FILTER` token as shown here.