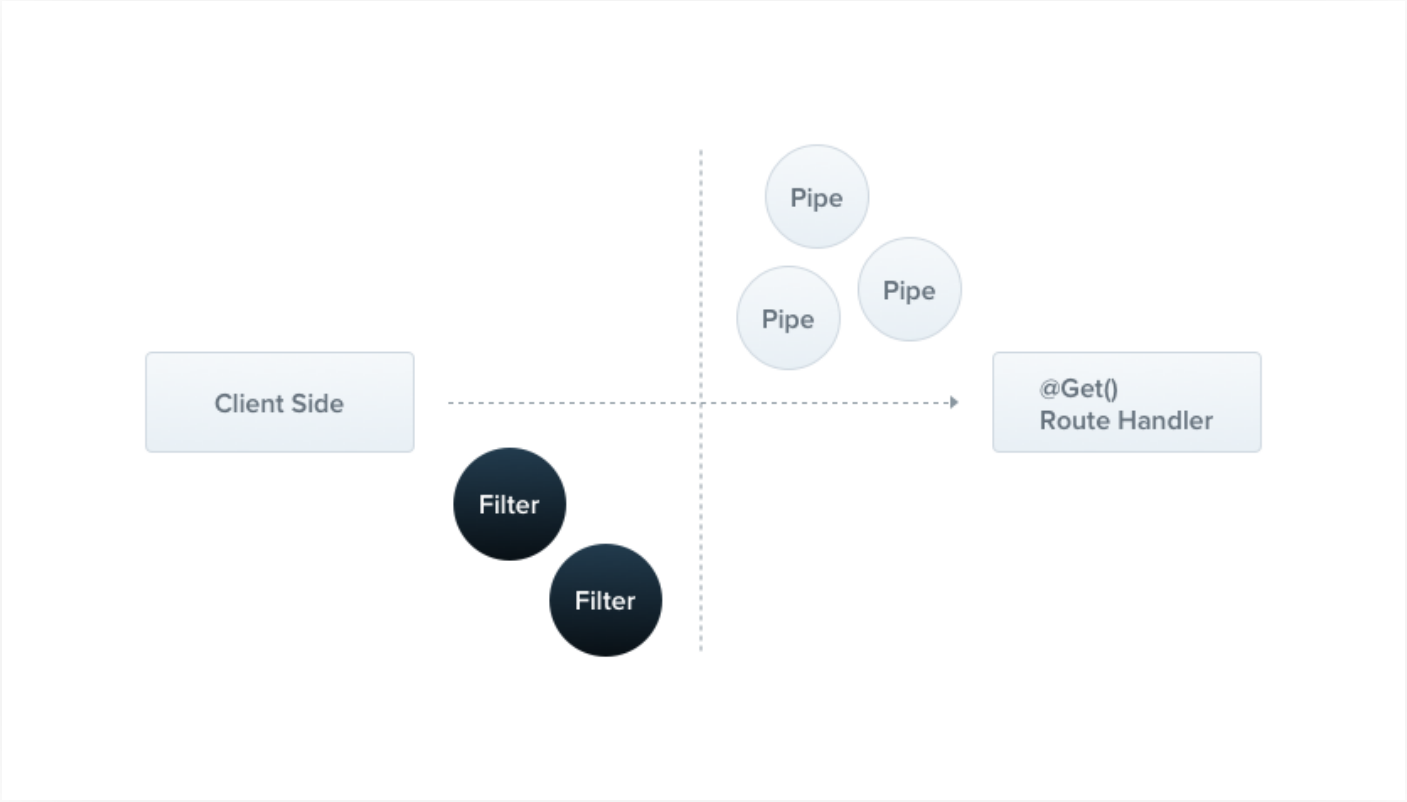
Exception filters

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 client receives the following default JSON response:

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

Base exceptions

The built-in `HttpException` class is exposed from the `@nestjs/common` package.

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:

cats.controller.ts JS

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

> **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 arguments which determine the JSON response body and the **HTTP response status code** respectively. The first argument is of type `string | object`. Pass a string to customize the error message (as shown in the `GET` handler of the `CatsController` above). Pass a plain literal `object` with properties `status` (the status code to appear in the JSON response body) and `error` (the message string) in the first parameter, instead of a `string`, to completely override the response body. The second constructor argument should be the actual HTTP response status code. Here's an example overriding the entire response body:

cats.controller.ts JS

```
@Get()
async findAll() {
  throw new HttpException({
    status: HttpStatus.FORBIDDEN,
    error: 'This is a custom message',
  }, 403);
}
```

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

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

Exceptions hierarchy

It is good practice to create your own **exceptions hierarchy**. This means that your custom HTTP exceptions should inherit from the base `HttpException` class. As a result, Nest will recognize your exceptions, and automatically take care of the error responses. Let's implement such a custom exception:

forbidden.exception.ts JS

```
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.

cats.controller.ts JS

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

HTTP exceptions

In order to reduce the need to write boilerplate code, Nest provides a set of usable exceptions that inherit from the core `HttpException`. All of them are exposed from the `@nestjs/common` package:

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

- `GoneException`
- `PayloadTooLargeException`
- `UnsupportedMediaTypeException`
- `UnprocessableEntityException`
- `InternalServerErrorException`
- `NotImplementedException`
- `BadGatewayException`
- `ServiceUnavailableException`
- `GatewayTimeoutException`

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.

Let's create an exception filter which is responsible for catching exceptions that are an instance of the `HttpException` class, and implementing custom response logic for them.

http-exception.filter.ts JS

```
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,
      });
  }
}
```

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.

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 wrapper around the arguments that have been passed to the **original** request handler (where the exception originated). It contains a specific arguments array based on the type of the application (and platform which is being used). Here's what an `ArgumentsHost` looks like:

```
export interface ArgumentsHost {
  getArgs<T extends Array<any> = any[]>(): T;
  getArgByIndex<T = any>(index: number): T;
  switchToRpc(): RpcArgumentsHost;
  switchToHttp(): HttpArgumentsHost;
  switchToWs(): WsArgumentsHost;
}
```

The `ArgumentsHost` supplies us with a set of convenience methods that help to pick the correct arguments from the underlying array, across different application contexts. In other words, `ArgumentsHost` is nothing more than an **array of arguments**. For example, when the filter is used within the HTTP application context, `ArgumentsHost` will contain a `[request, response]` array. However, when the current context is a web sockets application, it will contain a `[client, data]` array, as appropriate to that context. This approach enables you to access any argument that would eventually be passed to the original handler in your custom `catch()` method.

Binding filters

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

cats.controller.ts JS

```
@Post()
@UseFilters(new HttpExceptionFilter())
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
```

> **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**.

cats.controller.ts JS

```
@Post()
@UseFilters(HttpExceptionFilter)
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
```

> **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, controller-scoped, or global-scoped. For example, to set up a filter as controller-scoped, you would do the following:

cats.controller.ts JS

```
@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:

main.ts JS

```
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 following construction:

app.module.ts JS

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

@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**.

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()` .

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

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

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

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

In the example above the filter will catch each exception thrown, regardless of its type (class).

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.

all-exceptions.filter.ts JS

```
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 `HttpServer` 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**.

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more **here**.

Principal Sponsor

Sponsors / Partners

Become a sponsor