Exception filters

The only difference between the HTTP exception filter layer and the corresponding microservices layer is that instead of throwing `HttpException`, you should use `RpcException`.

```
throw new RpcException('Invalid credentials.');
```

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

With the sample above, Nest will handle the thrown exception and return the `error` object with the following structure:

```
{
  "status": "error",
  "message": "Invalid credentials."
}
```

Filters

Microservice exception filters behave similarly to HTTP exception filters, with one small difference. The `catch()` method must return an `Observable`.

```
@@filename(rpc-exception.filter)
import { Catch, RpcExceptionFilter, ArgumentsHost } from '@nestjs/common';
import { Observable, throwError } from 'rxjs';
import { RpcException } from '@nestjs/microservices';

@Catch(RpcException)
export class ExceptionFilter implements RpcExceptionFilter<RpcException> {
  catch(exception: RpcException, host: ArgumentsHost): Observable<any> {
    return throwError(() => exception.getError());
  }
}
@@switch
import { Catch } from '@nestjs/common';
import { throwError } from 'rxjs';

@Catch(RpcException)
export class ExceptionFilter {
  catch(exception, host) {
    return throwError(() => exception.getError());
  }
}
```

> warning **Warning** Global microservice exception filters aren't enabled by default when using a hybrid application.

The following example uses a manually instantiated method-scoped filter. Just as with HTTP based applications, you can also use controller-scoped filters (i.e., prefix the controller class with a `@UseFilters()` decorator).

```
@@filename()
@UseFilters(new ExceptionFilter())
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
@@switch
@UseFilters(new ExceptionFilter())
@MessagePattern({ cmd: 'sum' })
accumulate(data) {
  return (data || []).reduce((a, b) => a + b);
}
```

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 **core 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()
import { Catch, ArgumentsHost } from '@nestjs/common';
import { BaseRpcExceptionFilter } from '@nestjs/microservices';

@Catch()
export class AllExceptionsFilter extends BaseRpcExceptionFilter {
  catch(exception: any, host: ArgumentsHost) {
    return super.catch(exception, host);
  }
}
@@switch
import { Catch } from '@nestjs/common';
import { BaseRpcExceptionFilter } from '@nestjs/microservices';

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

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).