RouterProxy

August 17, 2026

RouterProxy

Kind: Class

Source: `packages/core/router/router-proxy.ts`

Part of: Core

`RouterProxy` wraps HTTP route and error-layer callbacks with centralized exception handling. It creates async proxy functions that preserve the Express-style middleware signature while forwarding thrown errors to Nest's `ExceptionsHandler` with an HTTP execution context.

Methods

Method	Signature	Returns
`createProxy`	`createProxy(targetCallback: RouterProxyCallback, exceptionsHandler: ExceptionsHandler)`	`void`
`createExceptionLayerProxy`	`createExceptionLayerProxy(targetCallback: <TError, TRequest, TResponse>( err: TError, req: TRequest, res: TResponse, next: () => void, ) => void`	Promise, exceptionsHandler: ExceptionsHandler)`

When something fails

- `RouterProxy` handles failure in 2 places: it turns it into a return value in all 2.

Diagram

```
graph LR
  A[Incoming HTTP request] --> B[RouterProxy.createProxy]
  B --> C[Route callback]
  C -->|Success| D[Response / next()]
```

```
  C -->|Throws error| E[ExecutionContextHost]
  E --> F[ExceptionsHandler.next]
  F --> G[Exception filters / HTTP response]

  H[Express error middleware] --> I[createExceptionLayerProxy]
  I --> J[Error callback]
  J -->|Throws error| E
```

Usage

```
import { RouterProxy } from '@nestjs/core/router/router-proxy';

// In Nest internals, ExceptionsHandler is configured with exception filters.
const routerProxy = new RouterProxy();

const routeHandler = async (req: any, res: any) => {
  const user = await usersService.findById(req.params.id);

  if (!user) {
    throw new Error('User not found');
  }

  res.json(user);
};

// `exceptionsHandler` is typically created by Nest's router explorer.
const proxiedHandler = routerProxy.createProxy(
  routeHandler,
  exceptionsHandler,
);

// Register the wrapped handler with the underlying HTTP adapter/router.
expressRouter.get('/users/:id', proxiedHandler);
```

AI Coding Instructions

- Wrap route callbacks with `createProxy()` so synchronous and asynchronous errors reach Nest exception filters consistently.
- Use `createExceptionLayerProxy()` only for error middleware that follows the `(err, req, res, next)` signature.
- Preserve the original callback arguments and avoid handling exceptions directly inside the proxy; delegate them through `ExceptionsHandler`.
- Ensure the supplied exception handler is configured for the current route context before registering the proxied callback.

- Treat this class as framework routing infrastructure; application code should normally use controllers, guards, and filters instead of calling it directly.

How it works

`RouterProxy` is a class that creates asynchronous HTTP-style callback wrappers around route, middleware, and error-layer callbacks. Its normal callback type accepts `(req, res, next)` and may return `void` or `Promise<void>`. [packages/core/router/router-proxy.ts:4-10]

- `createProxy(targetCallback, exceptionsHandler)` returns an async `(req, res, next)` function. It invokes and awaits `targetCallback(req, res, next)`. On normal completion, it does not explicitly return a value. [packages/core/router/router-proxy.ts:11-21]
- If that invocation throws synchronously or its returned promise rejects, the wrapper constructs an `ExecutionContextHost` containing `[req, res, next]`, passes the caught value and host to `exceptionsHandler.next`, then returns `res`. [packages/core/router/router-proxy.ts:20-26]
- `createExceptionLayerProxy(targetCallback, exceptionsHandler)` has the same behavior for callbacks with the error-layer signature `(err, req, res, next)`: it awaits the target callback, catches failures, creates a context from `[req, res, next]` —not including `err` —dispatches the newly caught failure to `exceptionsHandler.next`, and returns `res` from the catch path. [packages/core/router/router-proxy.ts:30-52]
- Neither factory performs runtime validation of either argument before returning its wrapper. [packages/core/router/router-proxy.ts:11-28] [packages/core/router/router-proxy.ts:30-53]

The context created after a failure defaults to the `http` context type. Its HTTP accessors expose the captured values as request at index `0`, response at index `1`, and next callback at index `2`. [packages/core/helpers/execution-context-host.ts:10-17] [packages/core/helpers/execution-context-host.ts:50-55]

`ExceptionsHandler.next` first attempts to invoke a selected custom exception filter; when no matching custom filter handles the exception, it delegates to `BaseExceptionFilter.catch`. [packages/core/exceptions/exceptions-handler.ts:12-17] Custom filters are selected from configured metadata and called with the exception and context. [packages/core/exceptions/exceptions-handler.ts:26-37] The base filter writes HTTP exception responses through its adapter when headers have not been sent, otherwise ends the response; unknown errors are converted to an HTTP-error-shaped response or a 500 response and may be logged.

[packages/core/exceptions/base-exception-filter.ts:26-47]
[packages/core/exceptions/base-exception-filter.ts:50-74]

Current router integration includes:

- Controller execution contexts are wrapped with `createProxy` together with an exception filter created for the controller callback. [packages/core/router/router-explorer.ts:350-375]
- Bound middleware `use` methods are wrapped with `createProxy` and a middleware exception handler. [packages/core/middleware/middleware-module.ts:303-315]
- The not-found callback throws a `NotFoundException`; it is wrapped with `createProxy` before registration as the adapter's not-found handler. [packages/core/router/routes-resolver.ts:146-159]
- The external error handler maps certain external errors, throws the mapped error, and is wrapped with `createExceptionLayerProxy` before registration through `setErrorHandler`. [packages/core/router/routes-resolver.ts:162-182]