NestMicroservice

August 18, 2026

NestMicroservice

Kind: Class

Source: `packages/microservices/nest-microservice.ts`

Part of: Microservices

`NestMicroservice` is the runtime application class used to bootstrap and manage a NestJS microservice transport server. It creates the server, registers modules and message listeners, applies global enhancers, and coordinates the `init()` and `listen()` lifecycle stages.

Extends: `NestApplicationContext`

Implements: `INestMicroservice`

Methods

Method	Signature	Returns
`createServer`	`createServer(config: CompleteMicroserviceOptions)`	`void`
`registerModules`	`registerModules()`	`Promise<any>`
`registerListeners`	`registerListeners()`	`void`
`useWebSocketAdapter`	`useWebSocketAdapter(adapter: WebSocketAdapter)`	`this`
`useGlobalFilters`	`useGlobalFilters(filters: ExceptionFilter[])`	`this`
`useGlobalPipes`	`useGlobalPipes(pipes: PipeTransform<any>[])`	`this`
`useGlobalInterceptors`	`useGlobalInterceptors(interceptors: NestInterceptor[])`	`this`
`useGlobalGuards`	`useGlobalGuards(guards: CanActivate[])`	`this`
`init`	`init()`	`Promise<this>`

Method	Signature	Returns
`listen`	`listen()`	`Promise<any>`
`close`	`close()`	`Promise<any>`
`setIsInitialized`	`setIsInitialized(isInitialized: boolean)`	`void`
`setIsTerminated`	`setIsTerminated(isTerminated: boolean)`	`void`
`setIsInitHookCalled`	`setIsInitHookCalled(isInitHookCalled: boolean)`	`void`
`on`	`on(event: string	number
`unwrap`	`unwrap()`	`T`
`closeApplication`	`closeApplication()`	`Promise<any>`
`dispose`	`dispose()`	`Promise<void>`
`resolveAsyncOptions`	`resolveAsyncOptions(config: AsyncMicroserviceOptions)`	`void`

Properties

Property	Type
`logger`	`any`

Where it refuses work

- `NestMicroservice` stops the work with an early return when `this.isTerminated`, in 2 places.
- `NestMicroservice` stops the work with an early return when `this.isInitialized`.
- `NestMicroservice` stops the work with an early return when `err`.
- `NestMicroservice` stops the work with an early return when `'on' in this.serverInstance`.
- `NestMicroservice` stops the work with an early return when `'unwrap' in this.serverInstance`.
- `NestMicroservice` stops the work with an early return when `this.appOptions.instrument?.instanceDecorator`.

When something fails

- `NestMicroservice` handles failure in 1 place: it lets it reach the caller in all 1.

Diagram

```
graph LR
  A[NestFactory.createMicroservice] --> B[NestMicroservice]
  B --> C[createServer]
  B --> D[registerModules]
  D --> E[registerListeners]
  B --> F[Global Pipes / Guards / Filters / Interceptors]
  B --> G[init]
  G --> H[listen]
  H --> I[Transport Server]
  I --> J[Message Handlers]
```

Usage

```
import { ValidationPipe } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { Transport } from '@nestjs/microservices';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.createMicroservice(AppModule, {
    transport: Transport.TCP,
    options: {
      host: '127.0.0.1',
      port: 3001,
    },
  });

  app.useGlobalPipes(
    new ValidationPipe({
      transform: true,
      whitelist: true,
    }),
  );

  await app.listen();
}

bootstrap();
```

AI Coding Instructions

- Prefer creating microservices through `NestFactory.createMicroservice()` rather than instantiating `NestMicroservice` directly.
- Configure transport options before calling `listen()`, since server creation and listener registration depend on the selected transport.
- Register global pipes, guards, interceptors, and filters during application setup, before the microservice begins listening.
- Preserve the lifecycle order: create server, register modules and handlers, initialize the application, then start listening.
- Ensure controllers use microservice message-pattern decorators such as `@MessagePattern()` so `registerListeners()` can bind handlers to the transport.

Relationships

- IMPORTS → `CanActivate`
- IMPORTS → `ExceptionFilter`
- IMPORTS → `INestMicroservice`
- IMPORTS → `NestInterceptor`
- IMPORTS → `PipeTransform`
- IMPORTS → `WebSocketAdapter`
- IMPORTS → `NestMicroserviceOptions`
- IMPORTS → `Logger`
- IMPORTS → `ApplicationConfig`
- IMPORTS → `MESSAGES`
- IMPORTS → `optionalRequire`
- IMPORTS → `NestContainer`
- IMPORTS → `Injector`
- IMPORTS → `GraphInspector`
- IMPORTS → `NestApplicationContext`
- IMPORTS → `SocketModule`