ServerFactory

August 18, 2026

ServerFactory

Kind: Class

Source: `packages/microservices/server/server-factory.ts`

Part of: Microservices

`ServerFactory` creates the appropriate NestJS microservice server implementation for a configured transport, such as TCP, Redis, NATS, MQTT, gRPC, or Kafka. It centralizes transport-to-server resolution so the rest of the microservices package can work with a common server interface.

Methods

Method	Signature	Returns
`create`	`create(microserviceOptions: MicroserviceOptions)`	`void`

Diagram

```
graph LR
  A[MicroserviceOptions] --> B[ServerFactory.create()]
  B --> C{transport}
  C -->|TCP| D[ServerTCP]
  C -->|Redis| E[ServerRedis]
  C -->|NATS| F[ServerNats]
  C -->|MQTT| G[ServerMqtt]
  C -->|gRPC| H[ServerGrpc]
  C -->|Kafka / RMQ| I[Transport-specific Server]
  D --> J[Microservice Server Interface]
  E --> J
  F --> J
  G --> J
  H --> J
  I --> J
```

Usage

```
import { Transport } from '@nestjs/microservices';
import { ServerFactory } from '@nestjs/microservices/server/server-factory';

const server = ServerFactory.create({
  transport: Transport.TCP,
  options: {
    host: '127.0.0.1',
    port: 3001,
  },
});

server.listen(() => {
  console.log('TCP microservice server is listening');
});
```

AI Coding Instructions

- Pass a valid `MicroserviceOptions` object with the intended `Transport` value and transport-specific options.
- Treat the returned value as a common microservice server abstraction; avoid coupling calling code to a specific `Server*` implementation.
- Add support for new transports by extending the factory mapping and returning a server implementation compatible with the existing server contract.
- Ensure transport options match the selected transport; TCP options, broker credentials, and gRPC package settings are not interchangeable.
- Prefer using Nest application factory APIs in application code when possible; use `ServerFactory` directly for framework-level or custom integration scenarios.

How it works

`ServerFactory` is a class with one static method, `create`, that constructs a transport-specific microservice server from a `MicroserviceOptions` configuration. It reads `transport` and `options` from the input after a TypeScript-only exclusion of the `CustomStrategy` union member. packages/microservices/server/server-factory.ts:20-25

`create` selects the constructed class as follows:

- `Transport.REDIS` → `ServerRedis` packages/microservices/server/server-factory.ts:26-28

- `Transport.NATS` → `ServerNats` packages/microservices/server/server-factory.ts:29-30
- `Transport.MQTT` → `ServerMqtt` packages/microservices/server/server-factory.ts:31-32
- `Transport.GRPC` → `ServerGrpc` packages/microservices/server/server-factory.ts:33-34
- `Transport.KAFKA` → `ServerKafka` packages/microservices/server/server-factory.ts:35-36
- `Transport.RMQ` → `ServerRMQ` packages/microservices/server/server-factory.ts:37-38
- Any other value, including an omitted `transport`, → `ServerTCP` packages/microservices/server/server-factory.ts:39-40. The repository test asserts that `create({})` returns a `ServerTCP`. packages/microservices/test/server/server-factory.spec.ts:13-16

The method passes `options` directly to `ServerGrpc`; for the other constructors, it applies a type assertion to the corresponding transport's `options` type. These assertions do not perform runtime validation. packages/microservices/server/server-factory.ts:21-40 The factory itself has no input checks, fallback error, or `try` / `catch`; errors thrown while constructing a selected server escape from `create`. packages/microservices/server/server-factory.ts:21-42

Construction can have immediate side effects before a server starts listening. For example, `ServerTCP` initializes its underlying server and serializer/deserializer in its constructor, packages/microservices/server/server-tcp.ts:49-60 while the Redis, NATS, MQTT, Kafka, RMQ, and gRPC constructors load their respective external packages. packages/microservices/server/server-redis.ts:42-50 packages/microservices/server/server-nats.ts:50-58 packages/microservices/server/server-mqtt.ts:47-56 packages/microservices/server/server-kafka.ts:78-85 packages/microservices/server/server-rmq.ts:77-95 packages/microservices/server/server-grpc.ts:70-88

`MicroserviceOptions` also permits `CustomStrategy`, whose shape has `strategy` and optional `options` rather than `transport`. packages/microservices/interfaces/microservice-configuration.interface.ts:25-33 packages/microservices/interfaces/microservice-configuration.interface.ts:50-53 In the visible `NestMicroservice` caller, configurations with `strategy` bypass `ServerFactory.create` and assign that strategy as the server instance. packages/microservices/nest-microservice.ts:84-109