Server

August 18, 2026

Server

Kind: Class

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

Part of: Microservices

`Server` is the base class for microservice transport servers. It manages message-handler registration and lookup, transport identification, and processing lifecycle hooks while allowing concrete transport implementations to provide connection, listening, shutdown, and native-client behavior.

Methods

Method	Signature	Returns
`on`	`on(event: EventKey, callback: EventCallback)`	`any`
`unwrap`	`unwrap()`	`T`
`listen`	`listen(callback: (...optionalParams: unknown[]) => any)`	`any`
`close`	`close()`	`any`
`setTransportId`	`setTransportId(transportId: Transport	symbol)`
`setOnProcessingStartHook`	`setOnProcessingStartHook(hook: (transportId: Transport	symbol, context: unknown, done: () => Promise,) => void)`
`setOnProcessingEndHook`	`setOnProcessingEndHook(hook: (transportId: Transport	symbol, context: unknown) => void)`
`addHandler`	`addHandler(pattern: any, callback: MessageHandler, isEventHandler: undefined, extras: Record<string, any>)`	`void`
`getHandlers`	`getHandlers()`	`Map<string, MessageHandler>`
`getHandlerByPattern`	`getHandlerByPattern(pattern: string)`	`MessageHandler

Method	Signature	Returns
`send`	`send(stream$: Observable, respond: (data: WritePacket) => Promise	void)`
`handleEvent`	`handleEvent(pattern: string, packet: ReadPacket, context: BaseRpcContext)`	`Promise<any>`
`transformToObservable`	`transformToObservable(resultOrDeferred: Observable	Promise)`
`transformToObservable`	`transformToObservable(resultOrDeferred: T)`	`never extends Observable<ObservedValueOf<T>> ? Observable<T> : Observable<ObservedValueOf<T>>`
`transformToObservable`	`transformToObservable(resultOrDeferred: any)`	`void`
`getOptionsProp`	`getOptionsProp(obj: Options, prop: Attribute)`	`Options[Attribute]`
`getOptionsProp`	`getOptionsProp(obj: Options, prop: Attribute, defaultValue: DefaultValue)`	`Required<Options>[Attribute]`
`getOptionsProp`	`getOptionsProp(obj: Options, prop: Attribute, defaultValue: DefaultValue)`	`void`
`handleError`	`handleError(error: string)`	`void`
`loadPackage`	`loadPackage(name: string, ctx: string, loader: Function)`	`T`
`initializeSerializer`	`initializeSerializer(options: ClientOptions['options'])`	`void`
`initializeDeserializer`	`initializeDeserializer(options: ClientOptions['options'])`	`void`
`getRouteFromPattern`	`getRouteFromPattern(pattern: string)`	`string`
`normalizePattern`	`normalizePattern(pattern: MsPattern)`	`string`

Properties

Property	Type
`transportId`	`Transport
`messageHandlers`	`any`
`logger`	`LoggerService`
`serializer`	`ConsumerSerializer`

Property	Type
`deserializer`	`ConsumerDeserializer`
`onProcessingStartHook`	`( transportId: Transport`
`onProcessingEndHook`	`( transportId: Transport`
`_status$`	`any`

Where it refuses work

- `Server` stops the work with an early return when `!handler`.
- `Server` stops the work with an early return when `resultOrDeferred instanceof Promise`.
- `Server` stops the work with an early return when `isObservable(resultOrDeferred)`.

When something fails

- `Server` handles failure in 1 place: it logs it and continues in all 1.

Diagram

```
graph LR
  Client[Microservice Client] --> Transport[Concrete Server Transport]
  Transport --> Server[Server Base Class]
  Server --> Handlers[Message Handler Map]
  Server --> StartHook[Processing Start Hook]
  Server --> EndHook[Processing End Hook]
  Handlers --> Handler[Message Handler]
  Handler --> Response[Response or Event Processing]
```

Usage

```
import { Server } from '@nestjs/microservices';

class CustomServer extends Server {
  async listen(callback: () => void) {
    // Connect to the underlying transport here.
    callback();
  }

  async close() {
    // Close transport connections and release resources.
  }

  unwrap<T>() {
    // Return the underlying transport client/server instance.
    return {} as T;
```

```
  }
}

const server = new CustomServer();

server.setTransportId('custom-transport');

server.setOnProcessingStartHook(() => {
  console.log('Started processing message');
});

server.setOnProcessingEndHook(() => {
  console.log('Finished processing message');
});

server.addHandler('users.find', async (payload) => {
  return { id: payload.id, name: 'Ada Lovelace' };
});

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

const handler = server.getHandlerByPattern('users.find');

if (handler) {
  const result = await handler({ id: 'user-1' });
  console.log(result);
}

await server.close();
```

AI Coding Instructions

- Extend `Server` when implementing a new transport; provide transport-specific `listen()`, `close()`, and `unwrap()` implementations.
- Register request and event handlers through `addHandler()` rather than mutating the handler map returned by `getHandlers()`.
- Use the same normalized message pattern when registering handlers and retrieving them with `getHandlerByPattern()`.
- Set a transport ID with `setTransportId()` when the transport must be identified by framework integrations or diagnostics.
- Keep processing hooks lightweight; use `setOnProcessingStartHook()` and `setOnProcessingEndHook()` for instrumentation, tracing, or metrics rather than business logic.