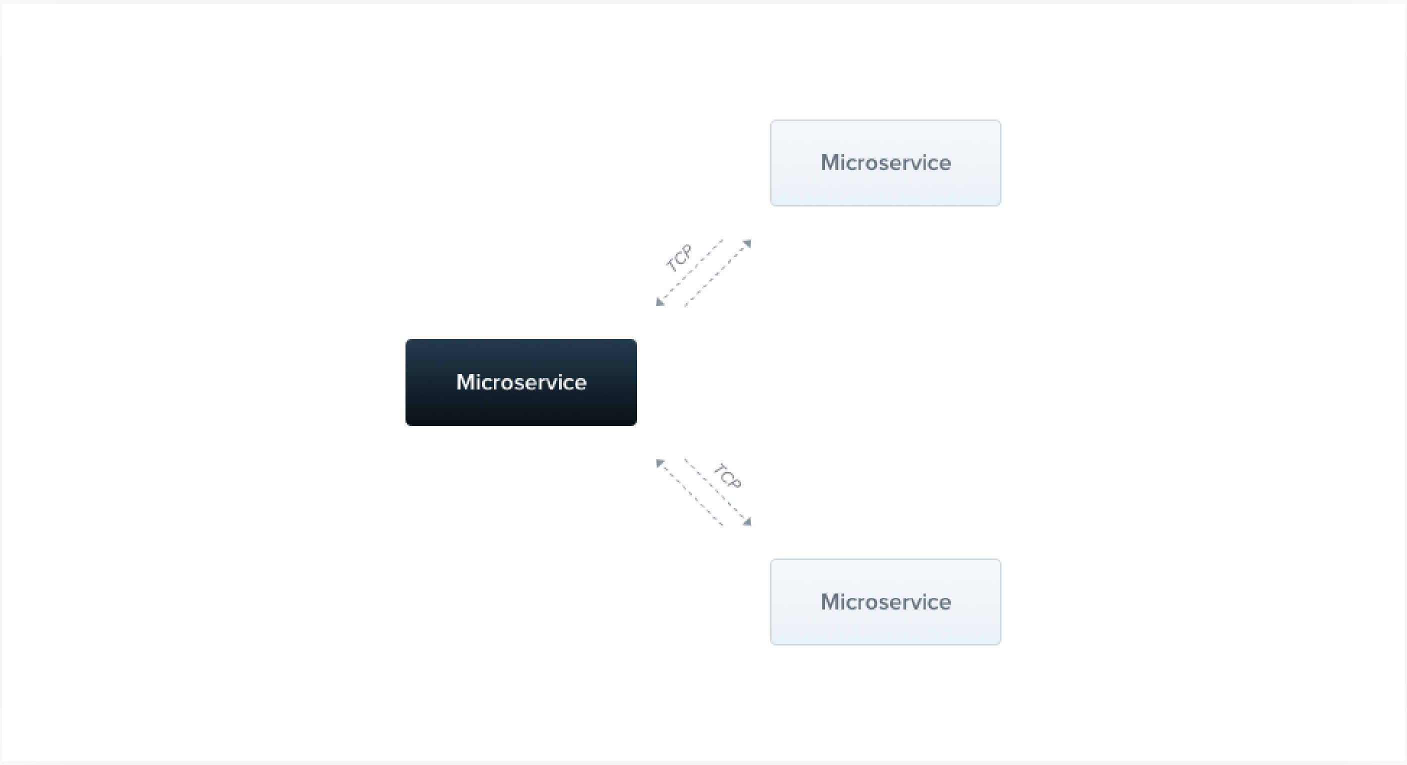
Basics

Nest microservice is a type of application that uses a different **transport** layer than HTTP.



Installation

Firstly, we need to install the required package:

```
$ npm i --save @nestjs/microservices
```

Overview

In general, Nest supports several built-in transporters. They are based on the **request-response** and **event-based** paradigms, and a whole communication logic is hidden behind an abstraction layer. This makes it easy to switch between transporters without changing the line of code. However, the request-response paradigm doesn't make too much sense with streaming platforms supplied with log based persistence, such as Kafka or NATS streaming as they are designed to solve a different range of issues. Nonetheless, they can still be used with either **event-based** (unidirectional) communication or application context feature.

Getting started

In order to create a microservice, we use `createMicroservice()` method of the `NestFactory` class.

main.ts JS

```js
import { NestFactory } from '@nestjs/core';
import { Transport } from '@nestjs/microservices';
import { ApplicationModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.createMicroservice(ApplicationModule, {
    transport: Transport.TCP,
  });
  app.listen(() => console.log('Microservice is listening'));
}
bootstrap();
```

HINT
A microservice is listening to messages through **TCP** protocol by default.

A second argument of the `createMicroservice()` method is an options object. This object may have two members:

`transport`	Specifies the transporter (for example, `Transport.NATS`)
`options`	A transporter-specific options object that determines transporter behaviour

The `options` object is different depending on chosen transporter. A **TCP** transporter exposes few properties described below.

`host`	Connection hostname
`port`	Connection port
`retryAttempts`	A total amount of connection attempts
`retryDelay`	A connection retrying delay (ms)

Patterns

Microservices recognize both messages and events through patterns. A pattern is a plain value, for example, a literal object or a string. Eventually, every pattern is being serialized, so it can be sent over the network along with the data. Hence, the receiver can easily associate the incoming message with the corresponding handler.

Request-response

The request-response communication mechanism is useful when you have to **exchange** messages among various, external services. Additionally, with this paradigm, you can be sure that the service has actually received the message.

In order to enable services to exchange data over the network, Nest creates two channels in which one is responsible for transferring the data while the other listens to the incoming response. However, it's not always the case. For instance, platforms as NATS provide such a feature out-of-the-box so we don't have to do it on our own.

Basically, to create a message handler (based on the request-response paradigm), we use the `@MessagePattern()` decorator which is imported from the `@nestjs/microservices` package.

math.controller.ts JS

```
import { Controller } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';

@Controller()
export class MathController {
  @MessagePattern({ cmd: 'sum' })
  accumulate(data: number[]): number {
    return (data || []).reduce((a, b) => a + b);
  }
}
```

The `accumulate()` handler is listening to messages that fulfil the `cmd: 'sum'` pattern. The pattern handler takes a single argument, the `data` passed from the client. In this case, the data is an array of numbers which has to be accumulated.

Asynchronous responses

Each message handler is able to respond either synchronously or **asynchronously**. Hence, `async` methods are supported.

JS

```
@MessagePattern({ cmd: 'sum' })
async accumulate(data: number[]): Promise<number> {
  return (data || []).reduce((a, b) => a + b);
}
```

Additionally, we are able to return an `Observable`, and thus the values will be emitted until the stream is completed.

```js
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): Observable<number> {
  return from([1, 2, 3]);
}
```

Above message handler will respond **3 times** (with each item from the array).

Event-based

While the request-response method is great when you have to constantly exchange messages between services, it brings too much unnecessary overhead that is completely useless when you just want to publish **events** (without waiting for a response). For instance, you would like to simply notify another service that a certain situation has happened in this part of the system. Thus, we provide a support for event-based communication as well.

In order to create an event handler, we use the `@EventPattern()` decorator which is imported from the `@nestjs/microservices` package.

```js
@EventPattern('user_created')
async handleUserCreated(data: Record<string, unknown>) {
  // business logic
}
```

The `handleUserCreated()` method is listening to `user_created` event. The event handler takes a single argument, the `data` passed from the client (in this case, an event payload which has been sent over the network).

Client

In order to either exchange messages or publish events to the Nest microservice, we use the `ClientProxy` class which instance can be created in a few ways. Firstly, we may import the `ClientsModule` which exposes static `register()` method. This method takes an array as a parameter in which every element has a `name` (which is a sort of the microservice identifier) as well as microservice-specific options (it's the same object as this one passed in to the `createMicroservice()` method).

```
ClientsModule.register([
  { name: 'MATH_SERVICE', transport: Transport.TCP },
]),
```

HINT
The `ClientsModule` is imported from the `@nestjs/microservices` package.

Once the module has been imported, we can inject `MATH_SERVICE` using the `@Inject()` decorator.

```
constructor(
  constructor(
    @Inject('MATH_SERVICE') private readonly client: ClientProxy,
  ) {}
)
```

HINT
The `ClientProxy` class is imported from the `@nestjs/microservices` package.

Nonetheless, this approach doesn't allow us to asynchronously fetch the microservice configuration. In this case, we can directly use `ClientProxyFactory` to register a **custom provider** (which is a client instance):

```
{
  provide: 'MATH_SERVICE',
  useFactory: (configService: ConfigService) => {
    const mathSvcOptions = configService.getMathSvcOptions();
    return ClientProxyFactory.create(mathSvcOptions);
  },
  inject: [ConfigService],
}
```

HINT
The `ClientProxyFactory` is imported from the `@nestjs/microservices` package.

The last feasible solution is to use the `@Client()` property decorator.

```
@Client({ transport: Transport.TCP })
client: ClientProxy;
```

HINT
The `@Client()` is imported from the `@nestjs/microservices` package.

However, using decorator is not a recommended way (hard to test, tough to share client instance).

The `ClientProxy` is **lazy**. It doesn't initiate a connection immediately. Instead, it will be established before the first microservice call, and then reused across each subsequent call. However, if you want to delay an application bootstrapping process and manually initialize a connection, you can use a `connect()` method inside the `OnModuleInit` lifecycle hook.

JS

```
async onModuleInit() {
  await this.client.connect();
}
```

If the connection cannot be created, the `connect()` method will reject with the corresponding error object.

Sending messages

The `ClientProxy` exposes a `send()` method. This method is intended to call the microservice and returns the `Observable` with its response. Consequently, we can subscribe to the emitted values easily.

JS

```
accumulate(): Observable<number> {
  const pattern = { cmd: 'sum' };
  const payload = [1, 2, 3];
  return this.client.send<number>(pattern, payload);
}
```

The `send()` method takes two arguments, `pattern` and `payload`. The `pattern` has to be equal to this one defined in the `@MessagePattern()` decorator while `payload` is a message that we want to transmit to another microservice.

Publishing events

Another available method is `emit()`. This method responsibility is to publish an event to the message broker.

```js
async publish() {
  this.client.emit<number>('user_created', new UserCreatedEvent());
}
```

The `emit()` method takes two arguments, `pattern` and `payload`. The `pattern` has to be equal to this one defined in the `@EventPattern()` decorator while `payload` is an event payload that we want to transmit to another microservice.

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more here.

Principal Sponsor

Sponsors / Partners

Become a sponsor