Implementing RabbitMQ Microservice Communication in NestJS

Diogo Miguel Chaves dos Santos Antunes Pereira

February 5, 2024

1 Workspace Setup

For the purposes of this tutorial, we will require both the Node and Docker technologies to have been previously installed.

Firstly, we need to setup our projects. We will require Node to be installed in our machine in order to use NestJS. After installing Node, if we run the command:

```
$ npm i -g @nestjs/cli
```

we will have installed the Nest Cli, and we will be ready to create our NestJS workspace. With Nest installed, running the following commands:

```
$ nest new host
$ cd host
$ nest g app app1
```

will setup a workspace with the following structure:

```
apps
├── app1
│   ├── src
│   │   ├── app.controller.ts
│   │   ├── app.module.ts
│   │   ├── app.service.ts
│   │   └── main.ts
│   └── tsconfig.app.json
└── host
    ├── src
    │   ├── app.controller.ts
    │   ├── app.module.ts
    │   ├── app.service.ts
    │   └── main.ts
    └── tsconfig.app.json
nest-cli.json
package.json
tsconfig.json
.eslintrc.js
```

For now, both of these projects will run as apps. However, we will want app1 to become a microservice. Therefore, we will need NestJS' microservice package. To acquire it, we can run the following line in the terminal:

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

Finally, we need to have RabbitMQ available on our workspace. For that, we need to install it by going to the root folder and running the command:

```
$ npm i --save amqplib amqp-connection-manager
```

2 Docker

We will use docker as part of our solution. In the root folder, create a docker-compose.yml file with the following lines:

```
version: "3.8"

services:
  rabbitmq:
    image: rabbitmq:3-management-alpine
    container_name: 'rabbitmq-tutorial'
    ports:
      - "5672:5672"
      - "15672:15672"
    environment:
      RABBITMQ_HOST: rabbitmq
```

To run this file, we need to run the command `docker compose up -d` in the terminal. This will then create the necessary containers. While port 5672 will be used for communication, we can use port 15672 to access a communication hub with credentials 'guest' for both username and password.

3 Microservice Setup

In our microservice, which in this tutorial would be app1, we need to change the main.ts file from a normal app to a microservice consuming the RabbitMQ message broker.

After replacing the contents of the `bootstrap` method in the main.ts file, this should be its content:

```
import { NestFactory } from '@nestjs/core';
import { App1Module } from './app1.module';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';

async function bootstrap() {
    const app =
    await NestFactory.createMicroservice<MicroserviceOptions>(App1Module,
    {
        transport: Transport.RMQ,
        options: {
          urls: ['amqp://localhost:5672'],
          queue: 'tutorial_queue',
          queueOptions: {
            durable: false
          },
        },
    });
    app.listen();
}
bootstrap();
```

4 Host Application Setup

In our host app, we will need to define the microservice with which we want to communicate. For this goal, we will need to create a RabbitMQ `ClientProxy` instance. To create the `ClientProxy`, we will use the `ClientsModule` from NestJS' microservices package, adding the client to our host app's app.module file as such:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ClientsModule, Transport } from '@nestjs/microservices';

@Module({
    imports: [
        ClientsModule.register([
            {
                name: 'TUTORIAL_SERVICE',
                transport: Transport.RMQ,
                options: {
                    urls: ['amqp://localhost:5672'],
                    queue: 'tutorial_queue',
                    queueOptions: {
                        durable: false
                    },
                },
            },
        ])
    ],
    controllers: [AppController],
    providers: [AppService],
})
export class AppModule {}
```

Make sure that the transport and the options are the same on the app1's microservice and on the host app's `ClientsModule` definition.

5 Communication

In the controller file's constructor of our host app service, we will have to inject the Client we created in our app module with the respective client name, and after we have the client variable, we will use it to send our message through the RabbitMQ broker.

```
import { Inject, Injectable } from '@nestjs/common';
import { ClientProxy } from '@nestjs/microservices';

@Injectable()
export class AppService {
    constructor(
        @Inject('TUTORIAL_SERVICE') private readonly client: ClientProxy
    ) {}

    async getHello() {
        return await this.client.send('tutorial_hello', 'Hello␣World!');
    }
}
```

In this case, we send the string 'Hello World' with the message pattern 'tutorial_hello'.

Finally, in our microservice's controller, we need to identify what to do when we receive this pattern. For that, we use the decorator `@MessagePattern()`. We also need the `@Payload()` to have access to the message sent.

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

@Controller()
export class App1Controller {
    constructor() {}

    @MessagePattern('tutorial_hello')
    getHello(@Payload() data: any): string {
        console.log(data);
        return 'Message␣Received␣from␣Host';
    }
}
```

In our case, we are simply logging the data received and sending and acknowledgement message back.

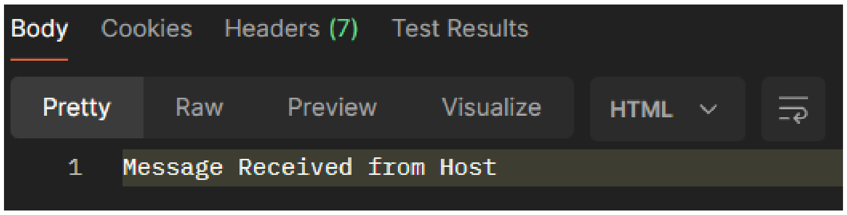
6 Testing

we can run each application by running the following command in the workspace's root folder:

```
$ npm run start:dev [app_name]
```

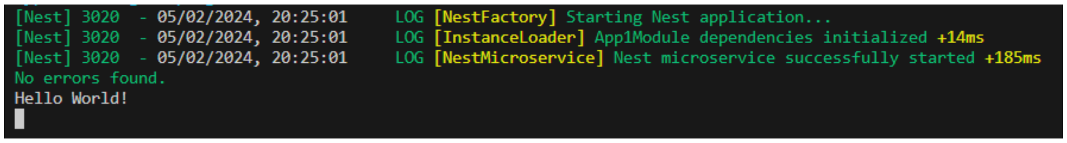
where `app_name` is the name of each application, in our case app1 and host.

After booting both applications, if we try to access the port in which the host app is running (by default, port 3000), we will get the response:



```
Message Received from Host
```

While if we check the logs of our microservice we will see:



```
[Nest] 3020  - 05/02/2024, 20:25:01     LOG [NestFactory] Starting Nest application...
[Nest] 3020  - 05/02/2024, 20:25:01     LOG [InstanceLoader] App1Module dependencies initialized +14ms
[Nest] 3020  - 05/02/2024, 20:25:01     LOG [NestMicroservice] Nest microservice successfully started +185ms
No errors found.
Hello World!
```

Therefore we can conclude the message was sent from the host application to the microservice, which processed it by logging the data, and sent a reply to the host app.

While usually requests are not directly processed in an app's controller, we were anyways able to implement communication between microservices in NestJS using the RabbitMQ message broker.