

## gRPC

The **gRPC** is a high-performance, open-source universal RPC framework.

### Installation

Before we start, we have to install required package:

```
$ npm i --save grpc @grpc/proto-loader
```

### Transporter

In order to switch to **gRPC** transporter, we need to modify an options object passed to the `createMicroservice()` method.

main.ts

JS

```
const app = await NestFactory.createMicroservice(ApplicationModule, {
  transport: Transport.GRPC,
  options: {
    package: 'hero',
    protoPath: join(__dirname, 'hero/hero.proto'),
  },
});
```

#### HINT

The `join()` function is imported from `path` package, while `Transport` enum is coming from `@nestjs/microservices`.

### Options

There are a bunch of available options that determine a transporter behavior.

`url`

Connection url

`proto-loader`

NPM package name (if you want to use another proto loader)

<code>protoLoader</code>	NPM package name (if you want to use another proto-loader)
<code>protoPath</code>	Absolute (or relative to the root dir) path to the <code>.proto</code> file
<code>loader</code>	<code>@grpc/proto-loader</code> options. They are well-described <a href="#">here</a> .
<code>package</code>	Protobuf package name
<code>credentials</code>	Server credentials ( <a href="#">read more</a> )

## Overview

In general, a `package` property sets a **protobuf** package name, while `protoPath` is a path to the `.proto` definitions file. The `hero.proto` file is structured using protocol buffer language.

```
syntax = "proto3";

package hero;

service HeroService {
  rpc FindOne (HeroById) returns (Hero) {}
}

message HeroById {
  int32 id = 1;
}

message Hero {
  int32 id = 1;
  string name = 2;
}
```

In the above example, we defined a `HeroService` that exposes a `FindOne()` gRPC handler which expects `HeroById` as an input and returns a `Hero` message. In order to define a handler that fulfills this protobuf definition, we have to use a `@GrpcMethod()` decorator. The previously known `@MessagePattern()` is no longer useful.

hero.controller.ts

JS

```
@GrpcMethod('HeroService', 'FindOne')
findOne(data: HeroById, metadata: any): Hero {
```

```
const items = [
  { id: 1, name: 'John' },
  { id: 2, name: 'Doe' },
];
return items.find(({ id }) => id === data.id);
}
```

## HINT

The `@GrpcMethod()` decorator is imported from `@nestjs/microservices` package.

The `HeroService` is a service name, while `FindOne` points to a `FindOne()` gRPC handler. The corresponding `findOne()` method takes two arguments, the `data` passed from the caller and `metadata` that stores gRPC request's metadata.

Furthermore, the `FindOne` is actually redundant here. If you don't pass a second argument to the `@GrpcMethod()`, Nest will automatically use the method name with the capitalized first letter, for example, `findOne` -> `FindOne`.

hero.controller.ts

JS

```
@Controller()
export class HeroService {
  @GrpcMethod()
  findOne(data: HeroById, metadata: any): Hero {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    return items.find(({ id }) => id === data.id);
  }
}
```

Likewise, you might not pass any argument. In this case, Nest would use a class name.

hero.controller.ts

JS

```
@Controller()
export class HeroService {
  @GrpcMethod()
  findOne(data: HeroById, metadata: any): Hero {
    const items = [
      { id: 1, name: 'John' },
    ];
  }
}
```

```

    { id: 2, name: 'Doe' },
  ];
  return items.find(({ id }) => id === data.id);
}
}

```

## Client

In order to create a client instance, we need to use `@Client()` decorator.

```

@Client({
  transport: Transport.GRPC,
  options: {
    package: 'hero',
    protoPath: join(__dirname, 'hero/hero.proto'),
  },
})
client: ClientGrpc;

```

There is a small difference compared to the previous examples. Instead of the `ClientProxy` class, we use the `ClientGrpc` that provides a `getService()` method. The `getService()` generic method takes service name as an argument and returns its instance if available.

hero.controller.ts

JS

```

onModuleInit() {
  this.heroService = this.client.getService<HeroService>('HeroService');
}

```

The `heroService` object exposes the same set of methods that have been defined inside `.proto` file. Note, all of them are **lowercased** (in order to follow the natural convention). Basically, our gRPC `HeroService` definition contains `FindOne()` function. It means that `heroService` instance will provide the `findOne()` method.

```

interface HeroService {
  findOne(data: { id: number }): Observable<any>;
}

```

All service methods return `Observable`. Since Nest supports **RxJS** streams and works pretty well with them, we can return them within HTTP handler as well

return them within HTTP handler as well.

hero.controllers.ts

JS

```
@Get()  
call(): Observable<any> {  
  return this.heroService.findOne({ id: 1 });  
}
```

A full working example is available [here](#).

## 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

