# 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 |
| `protoLoader` | NPM package name (if you want to use another proto-loader) |

| | |
|---|---|
| `protoLoader` | NPM package name (if you want to use another proto-loader) |
| `protoPath` | Absolute (or relative to the root dir) path to the `.proto` file |
| `loader` | `@grpc/proto-loader` options. They are well-described **here**. |
| `package` | Protobuf package name |
| `credentials` | Server credentials (**read more**) |

## 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.controller.ts JS

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

A full working example is available **here**.

## gRPC Streaming

GRPC on it's own supports long-term live connections more known as `streams`. Streams can be a very useful instrument for such service cases as Chatting, Observations or Chunk-data transfers. You can find more details in the official documentation (**here**).

Nest supports GRPC stream handlers in two possible ways:

- RxJS `Subject` + `Observable` handler: can be useful to write responses right inside of a Controller method or to be passed down to `Subject` / `Observable` consumer
- Pure GRPC call stream handler: can be useful to be passed to some executor which will handle the rest of dispatch for the Node standard `Duplex` stream handler.

## Subject strategy

`@GrpcStreamMethod()` decorator will provide the function parameter as RxJS `Observable`.

```
// Set decorator with selecting a Service definition from protobuf package
// the string is matching to: package proto_example.orders.OrdersService
@GrpcStreamMethod('orders.OrderService')
handleStream(messages: Observable<any>): Observable<any> {
  const subject = new Subject();
  messages.subscribe(message => {
    console.log(message);
    subject.next({
      shipmentType: {
        carrier: 'test-carrier',
      },
    });
  });
  return subject.asObservable();
}
```

For support full-duplex interaction with `@GrpcStreamMethod()` decorator, it is required to return an RxJS `Observable` from the controller method.

### Pure GRPC call stream handler

`@GrpcStreamCall()` decorator will provide function parameter as `grpc.ServerDuplexStream`, which supports standard methods like `.on('data', callback)`, `.write(message)` or `.cancel()`, full documentation on available methods can be found **here**.

```
// Set decorator with selecting a Service definition from protobuf package
// the string is matching to: package proto_example.orders.OrdersService
@GrpcStreamCall('orders.OrderService')
handleStream(stream: any) {
  stream.on('data', (msg: any) => {
    console.log(msg);
    // Answer here or anywhere else using stream reference
    stream.write({
      shipmentType: {
        carrier: 'test-carrier',
      },
    });
  });
}
```

This decorator do not require any specific return parameter to be provided. It is expected that stream will be handled in the way like any other standard stream type.

---

## 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