gRPC

gRPC is a modern, open source, high performance RPC framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication.

Like many RPC systems, gRPC is based on the concept of defining a service in terms of functions (methods) that can be called remotely. For each method, you define the parameters and return types. Services, parameters, and return types are defined in `.proto` files using Google's open source language-neutral protocol buffers mechanism.

With the gRPC transporter, Nest uses `.proto` files to dynamically bind clients and servers to make it easy to implement remote procedure calls, automatically serializing and deserializing structured data.

Installation

To start building gRPC-based microservices, first install the required packages:

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

Overview

Like other Nest microservices transport layer implementations, you select the gRPC transporter mechanism using the `transport` property of the options object passed to the `createMicroservice()` method. In the following example, we'll set up a hero service. The `options` property provides metadata about that service; its properties are described below.

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.GRPC,
  options: {
    package: 'hero',
    protoPath: join(__dirname, 'hero/hero.proto'),
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.GRPC,
  options: {
    package: 'hero',
    protoPath: join(__dirname, 'hero/hero.proto'),
  },
});
```

> info **Hint** The `join()` function is imported from the `path` package; the `Transport` enum is imported from the `@nestjs/microservices` package.

In the `nest-cli.json` file, we add the `assets` property that allows us to distribute non-TypeScript files, and `watchAssets` - to turn on watching all non-TypeScript assets. In our case, we want `.proto` files to be automatically copied to the `dist` folder.

```
{
  "compilerOptions": {
    "assets": ["**/*.proto"],
    "watchAssets": true
  }
}
```

Options

The **gRPC** transporter options object exposes the properties described below.

`package`	Protobuf package name (matches `package` setting from `.proto` file). Required
`protoPath`	Absolute (or relative to the root dir) path to the `.proto` file. Required
`url`	Connection url. String in the format `ip address/dns name:port` (for example, `'localhost:50051'`) defining the address/port on which the transporter establishes a connection. Optional. Defaults to `'localhost:5000'`
`protoLoader`	NPM package name for the utility to load `.proto` files. Optional. Defaults to `'@grpc/proto-loader'`
`loader`	`@grpc/proto-loader` options. These provide detailed control over the behavior of `.proto` files. Optional. See here for more details
`credentials`	Server credentials. Optional. Read more here

Sample gRPC service

Let's define our sample gRPC service called `HeroesService`. In the above `options` object, the`protoPath` property sets a path to the `.proto` definitions file `hero.proto`. The `hero.proto` file is structured using protocol buffers. Here's what it looks like:

```
// hero/hero.proto
syntax = "proto3";

package hero;

service HeroesService {
  rpc FindOne (HeroById) returns (Hero) {}
}
```

```
message HeroById {
  int32 id = 1;
}

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

Our `HeroesService` exposes a `FindOne()` method. This method expects an input argument of type `HeroById` and returns a `Hero` message (protocol buffers use `message` elements to define both parameter types and return types).

Next, we need to implement the service. To define a handler that fulfills this definition, we use the `@GrpcMethod()` decorator in a controller, as shown below. This decorator provides the metadata needed to declare a method as a gRPC service method.

> info **Hint** The `@MessagePattern()` decorator (read more) introduced in previous microservices chapters is not used with gRPC-based microservices. The `@GrpcMethod()` decorator effectively takes its place for gRPC-based microservices.

```
@@filename(heroes.controller)
@Controller()
export class HeroesController {
  @GrpcMethod('HeroesService', 'FindOne')
  findOne(data: HeroById, metadata: Metadata, call: ServerUnaryCall<any,
any>): Hero {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    return items.find(({ id }) => id === data.id);
  }
}
@@switch
@Controller()
export class HeroesController {
  @GrpcMethod('HeroesService', 'FindOne')
  findOne(data, metadata, call) {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    return items.find(({ id }) => id === data.id);
  }
}
```

> info **Hint** The `@GrpcMethod()` decorator is imported from the `@nestjs/microservices` package, while `Metadata` and `ServerUnaryCall` from the `grpc` package.

The decorator shown above takes two arguments. The first is the service name (e.g., `'HeroesService'`), corresponding to the `HeroesService` service definition in `hero.proto`. The second (the string `'FindOne'`) corresponds to the `FindOne()` rpc method defined within `HeroesService` in the `hero.proto` file.

The `findOne()` handler method takes three arguments, the `data` passed from the caller, `metadata` that stores gRPC request metadata and `call` to obtain the `GrpcCall` object properties such as `sendMetadata` for send metadata to client.

Both `@GrpcMethod()` decorator arguments are optional. If called without the second argument (e.g., `'FindOne'`), Nest will automatically associate the `.proto` file rpc method with the handler based on converting the handler name to upper camel case (e.g., the `findOne` handler is associated with the `FindOne` rpc call definition). This is shown below.

```
@@filename(heroes.controller)
@Controller()
export class HeroesController {
  @GrpcMethod('HeroesService')
  findOne(data: HeroById, metadata: Metadata, call: ServerUnaryCall<any,
any>): Hero {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    return items.find(({ id }) => id === data.id);
  }
}
@@switch
@Controller()
export class HeroesController {
  @GrpcMethod('HeroesService')
  findOne(data, metadata, call) {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    return items.find(({ id }) => id === data.id);
  }
}
```

You can also omit the first `@GrpcMethod()` argument. In this case, Nest automatically associates the handler with the service definition from the proto definitions file based on the **class** name where the handler is defined. For example, in the following code, class `HeroesService` associates its handler methods with the `HeroesService` service definition in the `hero.proto` file based on the matching of the name `'HeroesService'`.

```
@@filename(heroes.controller)
@Controller()
export class HeroesService {
```

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

Client

Nest applications can act as gRPC clients, consuming services defined in `.proto` files. You access remote services through a `ClientGrpc` object. You can obtain a `ClientGrpc` object in several ways.

The preferred technique is to import the `ClientsModule`. Use the `register()` method to bind a package of services defined in a `.proto` file to an injection token, and to configure the service. The `name` property is the injection token. For gRPC services, use `transport: Transport.GRPC`. The `options` property is an object with the same properties described above.

```
imports: [
  ClientsModule.register([
    {
      name: 'HERO_PACKAGE',
      transport: Transport.GRPC,
      options: {
        package: 'hero',
        protoPath: join(__dirname, 'hero/hero.proto'),
      },
    },
  ]),
];
```

> info **Hint** The `register()` method takes an array of objects. Register multiple packages by providing a comma separated list of registration objects.

Once registered, we can inject the configured `ClientGrpc` object with `@Inject()`. Then we use the `ClientGrpc` object's `getService()` method to retrieve the service instance, as shown below.

```
@Injectable()
export class AppService implements OnModuleInit {
  private heroesService: HeroesService;

  constructor(@Inject('HERO_PACKAGE') private client: ClientGrpc) {}

  onModuleInit() {
    this.heroesService = this.client.getService<HeroesService>
('HeroesService');
  }

  getHero(): Observable<string> {
    return this.heroesService.findOne({ id: 1 });
  }
}
```

> error **Warning** gRPC Client will not send fields that contain underscore `_` in their names unless the `keepCase` options is set to `true` in the proto loader configuration (`options.loader.keepcase` in the microservice transporter configuration).

Notice that there is a small difference compared to the technique used in other microservice transport methods. Instead of the `ClientProxy` class, we use the `ClientGrpc` class, which provides the `getService()` method. The `getService()` generic method takes a service name as an argument and returns its instance (if available).

Alternatively, you can use the `@Client()` decorator to instantiate a `ClientGrpc` object, as follows:

```
@Injectable()
export class AppService implements OnModuleInit {
  @Client({
    transport: Transport.GRPC,
    options: {
      package: 'hero',
      protoPath: join(__dirname, 'hero/hero.proto'),
    },
  })
  client: ClientGrpc;

  private heroesService: HeroesService;

  onModuleInit() {
    this.heroesService = this.client.getService<HeroesService>
('HeroesService');
  }

  getHero(): Observable<string> {
    return this.heroesService.findOne({ id: 1 });
```

```
  }
}
```

Finally, for more complex scenarios, we can inject a dynamically configured client using the `ClientProxyFactory` class as described here.

In either case, we end up with a reference to our `HeroesService` proxy object, which exposes the same set of methods that are defined inside the `.proto` file. Now, when we access this proxy object (i.e., `heroesService`), the gRPC system automatically serializes requests, forwards them to the remote system, returns a response, and deserializes the response. Because gRPC shields us from these network communication details, `heroesService` looks and acts like a local provider.

Note, all service methods are **lower camel cased** (in order to follow the natural convention of the language). So, for example, while our `.proto` file `HeroesService` definition contains the `FindOne()` function, the `heroesService` instance will provide the `findOne()` method.

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

A message handler is also able to return an `Observable`, in which case the result values will be emitted until the stream is completed.

```
@@filename(heroes.controller)
@Get()
call(): Observable<any> {
  return this.heroesService.findOne({ id: 1 });
}
@@switch
@Get()
call() {
  return this.heroesService.findOne({ id: 1 });
}
```

To send gRPC metadata (along with the request), you can pass a second argument, as follows:

```
call(): Observable<any> {
  const metadata = new Metadata();
  metadata.add('Set-Cookie', 'yummy_cookie=choco');

  return this.heroesService.findOne({ id: 1 }, metadata);
}
```

> info **Hint** The `Metadata` class is imported from the `grpc` package.

Please note that this would require updating the `HeroesService` interface that we've defined a few steps earlier.

Example

A working example is available here.

gRPC Streaming

gRPC on its own supports long-term live connections, conventionally known as `streams`. Streams are useful for cases such as Chatting, Observations or Chunk-data transfers. 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.

Streaming sample

Let's define a new sample gRPC service called `HelloService`. The `hello.proto` file is structured using protocol buffers. Here's what it looks like:

```
// hello/hello.proto
syntax = "proto3";

package hello;

service HelloService {
  rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);
  rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
}
```

> info **Hint** The `LotsOfGreetings` method can be simply implemented with the `@GrpcMethod` decorator (as in the examples above) since the returned stream can emit multiple values.

Based on this `.proto` file, let's define the `HelloService` interface:

```
interface HelloService {
  bidiHello(upstream: Observable<HelloRequest>):
Observable<HelloResponse>;
  lotsOfGreetings(
    upstream: Observable<HelloRequest>,
  ): Observable<HelloResponse>;
}

interface HelloRequest {
  greeting: string;
}

interface HelloResponse {
  reply: string;
}
```

> info **Hint** The proto interface can be automatically generated by the ts-proto package, learn more here.

Subject strategy

The `@GrpcStreamMethod()` decorator provides the function parameter as an RxJS `Observable`. Thus, we can receive and process multiple messages.

```
@GrpcStreamMethod()
bidiHello(messages: Observable<any>, metadata: Metadata, call:
ServerDuplexStream<any, any>): Observable<any> {
  const subject = new Subject();

  const onNext = message => {
    console.log(message);
    subject.next({
      reply: 'Hello, world!'
    });
  };
  const onComplete = () => subject.complete();
  messages.subscribe({
    next: onNext,
    complete: onComplete,
  });


  return subject.asObservable();
}
```

> warning **Warning** For supporting full-duplex interaction with the `@GrpcStreamMethod()` decorator, the controller method must return an RxJS `Observable`.

> info **Hint** The `Metadata` and `ServerUnaryCall` classes/interfaces are imported from the `grpc` package.

According to the service definition (in the `.proto` file), the `BidiHello` method should stream requests to the service. To send multiple asynchronous messages to the stream from a client, we leverage an RxJS `ReplaySubject` class.

```
const helloService = this.client.getService<HelloService>('HelloService');
const helloRequest$ = new ReplaySubject<HelloRequest>();

helloRequest$.next({ greeting: 'Hello (1)!' });
helloRequest$.next({ greeting: 'Hello (2)!' });
helloRequest$.complete();

return helloService.bidiHello(helloRequest$);
```

In the example above, we wrote two messages to the stream (`next()` calls) and notified the service that we've completed sending the data (`complete()` call).

Call stream handler

When the method return value is defined as `stream`, the `@GrpcStreamCall()` decorator provides the 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.

Alternatively, when the method return value is not a `stream`, the `@GrpcStreamCall()` decorator provides two function parameters, respectively `grpc.ServerReadableStream` (read more here) and `callback`.

Let's start with implementing the `BidiHello` which should support a full-duplex interaction.

```
@GrpcStreamCall()
bidiHello(requestStream: any) {
  requestStream.on('data', message => {
    console.log(message);
    requestStream.write({
      reply: 'Hello, world!'
    });
  });
}
```

> info **Hint** This decorator does not require any specific return parameter to be provided. It is expected that the stream will be handled similar to any other standard stream type.

In the example above, we used the `write()` method to write objects to the response stream. The callback passed into the `.on()` method as a second parameter will be called every time our service receives a new chunk of data.

Let's implement the `LotsOfGreetings` method.

```
@GrpcStreamCall()
lotsOfGreetings(requestStream: any, callback: (err: unknown, value:
HelloResponse) => void) {
  requestStream.on('data', message => {
    console.log(message);
  });
  requestStream.on('end', () => callback(null, { reply: 'Hello, world!'
}));
}
```

Here we used the `callback` function to send the response once processing of the `requestStream` has been completed.

gRPC Metadata

Metadata is information about a particular RPC call in the form of a list of key-value pairs, where the keys are strings and the values are typically strings but can be binary data. Metadata is opaque to gRPC itself - it lets the client provide information associated with the call to the server and vice versa. Metadata may include authentication tokens, request identifiers and tags for monitoring purposes, and data information such as the number of records in a data set.

To read the metadata in `@GrpcMethod()` handler, use the second argument (metadata), which is of type `Metadata` (imported from the `grpc` package).

To send back metadata from the handler, use the `ServerUnaryCall#sendMetadata()` method (third handler argument).

```
@@filename(heroes.controller)
@Controller()
export class HeroesService {
  @GrpcMethod()
  findOne(data: HeroById, metadata: Metadata, call: ServerUnaryCall<any,
any>): Hero {
    const serverMetadata = new Metadata();
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];

    serverMetadata.add('Set-Cookie', 'yummy_cookie=choco');
    call.sendMetadata(serverMetadata);

    return items.find(({ id }) => id === data.id);
  }
}
@@switch
@Controller()
```

```
export class HeroesService {
  @GrpcMethod()
  findOne(data, metadata, call) {
    const serverMetadata = new Metadata();
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];

    serverMetadata.add('Set-Cookie', 'yummy_cookie=choco');
    call.sendMetadata(serverMetadata);

    return items.find(({ id }) => id === data.id);
  }
}
```

Likewise, to read the metadata in handlers annotated with the `@GrpcStreamMethod()` handler (subject strategy), use the second argument (metadata), which is of type `Metadata` (imported from the `grpc` package).

To send back metadata from the handler, use the `ServerDuplexStream#sendMetadata()` method (third handler argument).

To read metadata from within the call stream handlers (handlers annotated with `@GrpcStreamCall()` decorator), listen to the `metadata` event on the `requestStream` reference, as follows:

```
requestStream.on('metadata', (metadata: Metadata) => {
  const meta = metadata.get('X-Meta');
});
```