## Overview

In addition to traditional (sometimes called monolithic) application architectures, Nest natively supports the microservice architectural style of development. Most of the concepts discussed elsewhere in this documentation, such as dependency injection, decorators, exception filters, pipes, guards and interceptors, apply equally to microservices. Wherever possible, Nest abstracts implementation details so that the same components can run across HTTP-based platforms, WebSockets, and Microservices. This section covers the aspects of Nest that are specific to microservices.

In Nest, a microservice is fundamentally an application that uses a different **transport** layer than HTTP.

Nest supports several built-in transport layer implementations, called **transporters**, which are responsible for transmitting messages between different microservice instances. Most transporters natively support both **request-response** and **event-based** message styles. Nest abstracts the implementation details of each transporter behind a canonical interface for both request-response and event-based messaging. This makes it easy to switch from one transport layer to another -- for example to leverage the specific reliability or performance features of a particular transport layer -- without impacting your application code.

### Installation

To start building microservices, first install the required package:

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

### Getting started

To instantiate a microservice, use the `createMicroservice()` method of the `NestFactory` class:

```
@@filename(main)
import { NestFactory } from '@nestjs/core';
import { Transport, MicroserviceOptions } from '@nestjs/microservices';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.TCP,
    },
  );
  await app.listen();
}
bootstrap();
@@switch
import { NestFactory } from '@nestjs/core';
import { Transport } from '@nestjs/microservices';
```

```
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.createMicroservice(AppModule, {
    transport: Transport.TCP,
  });
  await app.listen();
}
bootstrap();
```

> info **Hint** Microservices use the **TCP** transport layer by default.

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

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

The `options` object is specific to the chosen transporter. The **TCP** transporter exposes the properties described below. For other transporters (e.g, Redis, MQTT, etc.), see the relevant chapter for a description of the available options.

| | |
|---|---|
| `host` | Connection hostname |
| `port` | Connection port |
| `retryAttempts` | Number of times to retry message (default: `0`) |
| `retryDelay` | Delay between message retry attempts (ms) (default: `0`) |

### Patterns

Microservices recognize both messages and events by **patterns**. A pattern is a plain value, for example, a literal object or a string. Patterns are automatically serialized and sent over the network along with the data portion of a message. In this way, message senders and consumers can coordinate which requests are consumed by which handlers.

### Request-response

The request-response message style is useful when you need to **exchange** messages between various external services. With this paradigm, you can be certain that the service has actually received the message (without the need to manually implement a message ACK protocol). However, the request-response paradigm is not always the best choice. For example, streaming transporters that use log-based persistence, such as Kafka or NATS streaming, are optimized for solving a different range of issues, more aligned with an event messaging paradigm (see event-based messaging below for more details).

To enable the request-response message type, Nest creates two logical channels - one is responsible for transferring the data while the other waits for incoming responses. For some underlying transports, such as NATS, this dual-channel support is provided out-of-the-box. For others, Nest compensates by manually

creating separate channels. There can be overhead for this, so if you do not require a request-response message style, you should consider using the event-based method.

To create a message handler based on the request-response paradigm use the `@MessagePattern()` decorator, which is imported from the `@nestjs/microservices` package. This decorator should be used only within the controller classes since they are the entry points for your application. Using them inside providers won't have any effect as they are simply ignored by Nest runtime.

```
@@filename(math.controller)
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);
  }
}
@@switch
import { Controller } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';

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

In the above code, the `accumulate()` **message handler** listens for messages that fulfill the `{{ '{' }} cmd: 'sum' {{ '}' }}` message pattern. The message handler takes a single argument, the `data` passed from the client. In this case, the data is an array of numbers which are to be accumulated.

#### Asynchronous responses

Message handlers are able to respond either synchronously or **asynchronously**. Hence, `async` methods are supported.

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

```
  return (data || []).reduce((a, b) => a + b);
}
```

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

In the example above, the message handler will respond **3 times** (with each item from the array).

**Event-based**

While the request-response method is ideal for exchanging messages between services, it is less suitable when your message style is event-based - when you just want to publish **events** without waiting for a response. In that case, you do not want the overhead required by request-response for maintaining two channels.

Suppose you would like to simply notify another service that a certain condition has occurred in this part of the system. This is the ideal use case for the event-based message style.

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

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

> info **Hint** You can register multiple event handlers for a **single** event pattern and all of them will be automatically triggered in parallel.

The `handleUserCreated()` **event handler** listens for the `'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).

#### Decorators

In more sophisticated scenarios, you may want to access more information about the incoming request. For example, in the case of NATS with wildcard subscriptions, you may want to get the original subject that the producer has sent the message to. Likewise, in Kafka you may want to access the message headers. In order to accomplish that, you can use built-in decorators as follows:

```
@@filename()
@MessagePattern('time.us.*')
getDate(@Payload() data: number[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('time.us.*')
getDate(data, context) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
```

> info **Hint** `@Payload()`, `@Ctx()` and `NatsContext` are imported from `@nestjs/microservices`.

> info **Hint** You can also pass in a property key to the `@Payload()` decorator to extract a specific property from the incoming payload object, for example, `@Payload('id')`.

#### Client

A client Nest application can exchange messages or publish events to a Nest microservice using the `ClientProxy` class. This class defines several methods, such as `send()` (for request-response messaging) and `emit()` (for event-driven messaging) that let you communicate with a remote microservice. Obtain an instance of this class in one of the following ways.

One technique is to import the `ClientsModule`, which exposes the static `register()` method. This method takes an argument which is an array of objects representing microservice transporters. Each such object has a `name` property, an optional `transport` property (default is `Transport.TCP`), and an optional transporter-specific `options` property.

The `name` property serves as an **injection token** that can be used to inject an instance of a `ClientProxy` where needed. The value of the `name` property, as an injection token, can be an arbitrary string or JavaScript symbol, as described here.

The `options` property is an object with the same properties we saw in the `createMicroservice()` method earlier.

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

Once the module has been imported, we can inject an instance of the `ClientProxy` configured as specified via the `'MATH_SERVICE'` transporter options shown above, using the `@Inject()` decorator.

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

> info **Hint** The `ClientsModule` and `ClientProxy` classes are imported from the `@nestjs/microservices` package.

At times we may need to fetch the transporter configuration from another service (say a `ConfigService`), rather than hard-coding it in our client application. To do this, we can register a custom provider using the `ClientProxyFactory` class. This class has a static `create()` method, which accepts a transporter options object, and returns a customized `ClientProxy` instance.

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

> info **Hint** The `ClientProxyFactory` is imported from the `@nestjs/microservices` package.

Another option is to use the `@Client()` property decorator.

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

> info **Hint** The `@Client()` decorator is imported from the `@nestjs/microservices` package.

Using the `@Client()` decorator is not the preferred technique, as it is harder to test and harder to share a 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 the application bootstrapping process until a connection is established, you can manually initiate a connection using the `ClientProxy` object's `connect()` method inside the `OnApplicationBootstrap` lifecycle hook.

```
@@filename()
async onApplicationBootstrap() {
  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 an `Observable` with its response. Thus, we can subscribe to the emitted values easily.

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

The `send()` method takes two arguments, `pattern` and `payload`. The `pattern` should match one defined in a `@MessagePattern()` decorator. The `payload` is a message that we want to transmit to the remote microservice. This method returns a **cold** `Observable`, which means that you have to explicitly subscribe to it before the message will be sent.

### Publishing events

To send an event, use the `ClientProxy` object's `emit()` method. This method publishes an event to the message broker.

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

The `emit()` method takes two arguments, `pattern` and `payload`. The `pattern`should match one defined in an `@EventPattern()` decorator. The `payload` is an event payload that we want to transmit to the remote microservice. This method returns a **hot** `Observable` (unlike the cold `Observable` returned by `send()`), which means that whether or not you explicitly subscribe to the observable, the proxy will immediately try to deliver the event.

**Scopes**

For people coming from different programming language backgrounds, it might be unexpected to learn that in Nest, almost everything is shared across incoming requests. We have a connection pool to the database, singleton services with global state, etc. Remember that Node.js doesn't follow the request/response Multi-Threaded Stateless Model in which every request is processed by a separate thread. Hence, using singleton instances is fully **safe** for our applications.

However, there are edge-cases when request-based lifetime of the handler may be the desired behavior, for instance per-request caching in GraphQL applications, request tracking or multi-tenancy. Learn how to control scopes here.

Request-scoped handlers and providers can inject `RequestContext` using the `@Inject()` decorator in combination with `CONTEXT` token:

```
import { Injectable, Scope, Inject } from '@nestjs/common';
import { CONTEXT, RequestContext } from '@nestjs/microservices';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {
  constructor(@Inject(CONTEXT) private ctx: RequestContext) {}
}
```

This provides access to the `RequestContext` object, which has two properties:

```
export interface RequestContext<T = any> {
  pattern: string | Record<string, any>;
  data: T;
}
```

The `data` property is the message payload sent by the message producer. The `pattern` property is the pattern used to identify an appropriate handler to handle the incoming message.

**Handling timeouts**

In distributed systems, sometimes microservices might be down or not available. To avoid infinitely long waiting, you can use Timeouts. A timeout is an incredibly useful pattern when communicating with other services. To apply timeouts to your microservice calls, you can use the RxJS `timeout` operator. If the microservice does not respond to the request within a certain time, an exception is thrown, which can be caught and handled appropriately.

To solve this problem you have to use `rxjs` package. Just use the `timeout` operator in the pipe:

```
@@filename()
this.client
      .send<TResult, TInput>(pattern, data)
      .pipe(timeout(5000));
@@switch
this.client
      .send(pattern, data)
      .pipe(timeout(5000));
```

> info **Hint** The `timeout` operator is imported from the `rxjs/operators` package.

After 5 seconds, if the microservice isn't responding, it will throw an error.

## Redis

The Redis transporter implements the publish/subscribe messaging paradigm and leverages the Pub/Sub feature of Redis. Published messages are categorized in channels, without knowing what subscribers (if any) will eventually receive the message. Each microservice can subscribe to any number of channels. In addition, more than one channel can be subscribed to at a time. Messages exchanged through channels are **fire-and-forget**, which means that if a message is published and there are no subscribers interested in it, the message is removed and cannot be recovered. Thus, you don't have a guarantee that either messages or events will be handled by at least one service. A single message can be subscribed to (and received) by multiple subscribers.

### Installation

To start building Redis-based microservices, first install the required package:

```
$ npm i --save ioredis
```

### Overview

To use the Redis transporter, pass the following options object to the `createMicroservice()` method:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.REDIS,
  options: {
    host: 'localhost',
    port: 6379,
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.REDIS,
  options: {
    host: 'localhost',
    port: 6379,
  },
});
```

> info **Hint** The `Transport` enum is imported from the `@nestjs/microservices` package.

### Options

The `options` property is specific to the chosen transporter. The **Redis** transporter exposes the properties described below.

| | |
|---|---|
| `host` | Connection url |
| `port` | Connection port |
| `retryAttempts` | Number of times to retry message (default: `0`) |
| `retryDelay` | Delay between message retry attempts (ms) (default: `0`) |
| `wildcards` | Enables Redis wilcard subscriptions, instructing transporter to use `psubscribe`/`pmessage` under the hood. (default: `false`) |

All the properties supported by the official ioredis client are also supported by this transporter.

#### Client

Like other microservice transporters, you have several options for creating a Redis `ClientProxy` instance.

One method for creating an instance is to use the `ClientsModule`. To create a client instance with the `ClientsModule`, import it and use the `register()` method to pass an options object with the same properties shown above in the `createMicroservice()` method, as well as a `name` property to be used as the injection token. Read more about `ClientsModule` here.

```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'MATH_SERVICE',
        transport: Transport.REDIS,
        options: {
          host: 'localhost',
          port: 6379,
        }
      },
    ]),
  ]
  ...
})
```

Other options to create a client (either `ClientProxyFactory` or `@Client()`) can be used as well. You can read about them here.

#### Context

In more sophisticated scenarios, you may want to access more information about the incoming request. When using the Redis transporter, you can access the `RedisContext` object.

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RedisContext)
{
  console.log(`Channel: ${context.getChannel()}`);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(`Channel: ${context.getChannel()}`);
}
```

> info **Hint** `@Payload()`, `@Ctx()` and `RedisContext` are imported from the `@nestjs/microservices` package.

# MQTT

MQTT (Message Queuing Telemetry Transport) is an open source, lightweight messaging protocol, optimized for low latency. This protocol provides a scalable and cost-efficient way to connect devices using a **publish/subscribe** model. A communication system built on MQTT consists of the publishing server, a broker and one or more clients. It is designed for constrained devices and low-bandwidth, high-latency or unreliable networks.

### Installation

To start building MQTT-based microservices, first install the required package:

```
$ npm i --save mqtt
```

### Overview

To use the MQTT transporter, pass the following options object to the `createMicroservice()` method:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.MQTT,
  options: {
    url: 'mqtt://localhost:1883',
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.MQTT,
  options: {
    url: 'mqtt://localhost:1883',
  },
});
```

> info **Hint** The `Transport` enum is imported from the `@nestjs/microservices` package.

### Options

The `options` object is specific to the chosen transporter. The **MQTT** transporter exposes the properties described here.

### Client

Like other microservice transporters, you have several options for creating a MQTT `ClientProxy` instance.

One method for creating an instance is to use use the `ClientsModule`. To create a client instance with the `ClientsModule`, import it and use the `register()` method to pass an options object with the same properties shown above in the `createMicroservice()` method, as well as a `name` property to be used as the injection token. Read more about `ClientsModule` here.

```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'MATH_SERVICE',
        transport: Transport.MQTT,
        options: {
          url: 'mqtt://localhost:1883',
        }
      },
    ]),
  ]
  ...
})
```

Other options to create a client (either `ClientProxyFactory` or `@Client()`) can be used as well. You can read about them here.

**Context**

In more sophisticated scenarios, you may want to access more information about the incoming request. When using the MQTT transporter, you can access the `MqttContext` object.

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: MqttContext) {
  console.log(`Topic: ${context.getTopic()}`);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(`Topic: ${context.getTopic()}`);
}
```

> info **Hint** `@Payload()`, `@Ctx()` and `MqttContext` are imported from the `@nestjs/microservices` package.

To access the original mqtt packet, use the `getPacket()` method of the `MqttContext` object, as follows:

```
@@filename()
@MessagePattern('notifications')
```

```
  getNotifications(@Payload() data: number[], @Ctx() context: MqttContext) {
    console.log(context.getPacket());
  }
  @@switch
  @Bind(Payload(), Ctx())
  @MessagePattern('notifications')
  getNotifications(data, context) {
    console.log(context.getPacket());
  }
```

### Wildcards

A subscription may be to an explicit topic, or it may include wildcards. Two wildcards are available, `+` and `#`. `+` is a single-level wildcard, while `#` is a multi-level wildcard which covers many topic levels.

```
  @@filename()
  @MessagePattern('sensors/+/temperature/+')
  getTemperature(@Ctx() context: MqttContext) {
    console.log(`Topic: ${context.getTopic()}`);
  }
  @@switch
  @Bind(Ctx())
  @MessagePattern('sensors/+/temperature/+')
  getTemperature(context) {
    console.log(`Topic: ${context.getTopic()}`);
  }
```

### Record builders

To configure message options (adjust the QoS level, set the Retain or DUP flags, or add additional properties to the payload), you can use the `MqttRecordBuilder` class. For example, to set `QoS` to `2` use the `setQoS` method, as follows:

```
  const userProperties = { 'x-version': '1.0.0' };
  const record = new MqttRecordBuilder(':cat:')
    .setProperties({ userProperties })
    .setQoS(1)
    .build();
  client.send('replace-emoji', record).subscribe(...);
```

> info **Hint** `MqttRecordBuilder` class is exported from the `@nestjs/microservices` package.

And you can read these options on the server-side as well, by accessing the `MqttContext`.

```
  @@filename()
  @MessagePattern('replace-emoji')
```

```
  replaceEmoji(@Payload() data: string, @Ctx() context: MqttContext): string
  {
    const { properties: { userProperties } } = context.getPacket();
    return userProperties['x-version'] === '1.0.0' ? '🐱 ' : '🐈 ';
  }
  @@switch
  @Bind(Payload(), Ctx())
  @MessagePattern('replace-emoji')
  replaceEmoji(data, context) {
    const { properties: { userProperties } } = context.getPacket();
    return userProperties['x-version'] === '1.0.0' ? '🐱 ' : '🐈 ';
  }
```

In some cases you might want to configure user properties for multiple requests, you can pass these options to the `ClientProxyFactory`.

```
import { Module } from '@nestjs/common';
import { ClientProxyFactory, Transport } from '@nestjs/microservices';

@Module({
  providers: [
    {
      provide: 'API_v1',
      useFactory: () =>
        ClientProxyFactory.create({
          transport: Transport.MQTT,
          options: {
            url: 'mqtt://localhost:1833',
            userProperties: { 'x-version': '1.0.0' },
          },
        }),
    },
  ],
})
export class ApiModule {}
```

# NATS

NATS is a simple, secure and high performance open source messaging system for cloud native applications, IoT messaging, and microservices architectures. The NATS server is written in the Go programming language, but client libraries to interact with the server are available for dozens of major programming languages. NATS supports both **At Most Once** and **At Least Once** delivery. It can run anywhere, from large servers and cloud instances, through edge gateways and even Internet of Things devices.

### Installation

To start building NATS-based microservices, first install the required package:

```
$ npm i --save nats
```

### Overview

To use the NATS transporter, pass the following options object to the `createMicroservice()` method:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.NATS,
  options: {
    servers: ['nats://localhost:4222'],
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.NATS,
  options: {
    servers: ['nats://localhost:4222'],
  },
});
```

> info **Hint** The `Transport` enum is imported from the `@nestjs/microservices` package.

### Options

The `options` object is specific to the chosen transporter. The **NATS** transporter exposes the properties described here. Additionally, there is a `queue` property which allows you to specify the name of the queue that your server should subscribe to (leave `undefined` to ignore this setting). Read more about NATS queue groups below.

### Client

Like other microservice transporters, you have several options for creating a NATS `ClientProxy` instance.

One method for creating an instance is to use the `ClientsModule`. To create a client instance with the `ClientsModule`, import it and use the `register()` method to pass an options object with the same properties shown above in the `createMicroservice()` method, as well as a `name` property to be used as the injection token. Read more about `ClientsModule` here.

```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'MATH_SERVICE',
        transport: Transport.NATS,
        options: {
          servers: ['nats://localhost:4222'],
        }
      },
    ]),
  ]
  ...
})
```

Other options to create a client (either `ClientProxyFactory` or `@Client()`) can be used as well. You can read about them here.

**Request-response**

For the **request-response** message style (read more), the NATS transporter does not use the NATS built-in Request-Reply mechanism. Instead, a "request" is published on a given subject using the `publish()` method with a unique reply subject name, and responders listen on that subject and send responses to the reply subject. Reply subjects are directed back to the requestor dynamically, regardless of location of either party.

**Event-based**

For the **event-based** message style (read more), the NATS transporter uses NATS built-in Publish-Subscribe mechanism. A publisher sends a message on a subject and any active subscriber listening on that subject receives the message. Subscribers can also register interest in wildcard subjects that work a bit like a regular expression. This one-to-many pattern is sometimes called fan-out.

**Queue groups**

NATS provides a built-in load balancing feature called distributed queues. To create a queue subscription, use the `queue` property as follows:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
```

```
  transport: Transport.NATS,
  options: {
    servers: ['nats://localhost:4222'],
    queue: 'cats_queue',
  },
});
```

### Context

In more sophisticated scenarios, you may want to access more information about the incoming request. When using the NATS transporter, you can access the `NatsContext` object.

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(`Subject: ${context.getSubject()}`);
}
```

> info **Hint** `@Payload()`, `@Ctx()` and `NatsContext` are imported from the `@nestjs/microservices` package.

### Wildcards

A subscription may be to an explicit subject, or it may include wildcards.

```
@@filename()
@MessagePattern('time.us.*')
getDate(@Payload() data: number[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('time.us.*')
getDate(data, context) {
  console.log(`Subject: ${context.getSubject()}`); // e.g. "time.us.east"
  return new Date().toLocaleTimeString(...);
}
```

### Record builders

To configure message options, you can use the `NatsRecordBuilder` class (note: this is doable for event-based flows as well). For example, to add `x-version` header, use the `setHeaders` method, as follows:

```
import * as nats from 'nats';

// somewhere in your code
const headers = nats.headers();
headers.set('x-version', '1.0.0');

const record = new NatsRecordBuilder(':cat:').setHeaders(headers).build();
this.client.send('replace-emoji', record).subscribe(...);
```

> info **Hint** `NatsRecordBuilder` class is exported from the `@nestjs/microservices` package.

And you can read these headers on the server-side as well, by accessing the `NatsContext`, as follows:

```
@@filename()
@MessagePattern('replace-emoji')
replaceEmoji(@Payload() data: string, @Ctx() context: NatsContext): string
{
  const headers = context.getHeaders();
  return headers['x-version'] === '1.0.0' ? '🐱 ' : '🦮 ';
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('replace-emoji')
replaceEmoji(data, context) {
  const headers = context.getHeaders();
  return headers['x-version'] === '1.0.0' ? '🐱 ' : '🦮 ';
}
```

In some cases you might want to configure headers for multiple requests, you can pass these as options to the `ClientProxyFactory`:

```
import { Module } from '@nestjs/common';
import { ClientProxyFactory, Transport } from '@nestjs/microservices';

@Module({
  providers: [
    {
      provide: 'API_v1',
      useFactory: () =>
        ClientProxyFactory.create({
          transport: Transport.NATS,
          options: {
            servers: ['nats://localhost:4222'],
            headers: { 'x-version': '1.0.0' },
          },
```

```
          }),
      },
    ],
  })
  export class ApiModule {}
```

## RabbitMQ

RabbitMQ is an open-source and lightweight message broker which supports multiple messaging protocols. It can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements. In addition, it's the most widely deployed message broker, used worldwide at small startups and large enterprises.

### Installation

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

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

### Overview

To use the RabbitMQ transporter, pass the following options object to the `createMicroservice()` method:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.RMQ,
  options: {
    urls: ['amqp://localhost:5672'],
    queue: 'cats_queue',
    queueOptions: {
      durable: false
    },
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.RMQ,
  options: {
    urls: ['amqp://localhost:5672'],
    queue: 'cats_queue',
    queueOptions: {
      durable: false
    },
  },
});
```

> info **Hint** The `Transport` enum is imported from the `@nestjs/microservices` package.

### Options

The `options` property is specific to the chosen transporter. The **RabbitMQ** transporter exposes the properties described below.

| | |
|---|---|
| `urls` | Connection urls |
| `queue` | Queue name which your server will listen to |
| `prefetchCount` | Sets the prefetch count for the channel |
| `isGlobalPrefetchCount` | Enables per channel prefetching |
| `noAck` | If `false`, manual acknowledgment mode enabled |
| `queueOptions` | Additional queue options (read more here) |
| `socketOptions` | Additional socket options (read more here) |
| `headers` | Headers to be sent along with every message |

**Client**

Like other microservice transporters, you have several options for creating a RabbitMQ `ClientProxy` instance.

One method for creating an instance is to use the `ClientsModule`. To create a client instance with the `ClientsModule`, import it and use the `register()` method to pass an options object with the same properties shown above in the `createMicroservice()` method, as well as a `name` property to be used as the injection token. Read more about `ClientsModule` here.

```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'MATH_SERVICE',
        transport: Transport.RMQ,
        options: {
          urls: ['amqp://localhost:5672'],
          queue: 'cats_queue',
          queueOptions: {
            durable: false
          },
        },
      },
    ]),
  ]
  ...
})
```

Other options to create a client (either `ClientProxyFactory` or `@Client()`) can be used as well. You can read about them here.

**Context**

In more sophisticated scenarios, you may want to access more information about the incoming request. When using the RabbitMQ transporter, you can access the `RmqContext` object.

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  console.log(`Pattern: ${context.getPattern()}`);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(`Pattern: ${context.getPattern()}`);
}
```

> info **Hint** `@Payload()`, `@Ctx()` and `RmqContext` are imported from the `@nestjs/microservices` package.

To access the original RabbitMQ message (with the `properties`, `fields`, and `content`), use the `getMessage()` method of the `RmqContext` object, as follows:

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  console.log(context.getMessage());
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(context.getMessage());
}
```

To retrieve a reference to the RabbitMQ channel, use the `getChannelRef` method of the `RmqContext` object, as follows:

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  console.log(context.getChannelRef());
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  console.log(context.getChannelRef());
}
```

### Message acknowledgement

To make sure a message is never lost, RabbitMQ supports message acknowledgements. An acknowledgement is sent back by the consumer to tell RabbitMQ that a particular message has been received, processed and that RabbitMQ is free to delete it. If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it.

To enable manual acknowledgment mode, set the `noAck` property to `false`:

```
options: {
  urls: ['amqp://localhost:5672'],
  queue: 'cats_queue',
  noAck: false,
  queueOptions: {
    durable: false
  },
},
```

When manual consumer acknowledgements are turned on, we must send a proper acknowledgement from the worker to signal that we are done with a task.

```
@@filename()
@MessagePattern('notifications')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  const channel = context.getChannelRef();
  const originalMsg = context.getMessage();

  channel.ack(originalMsg);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('notifications')
getNotifications(data, context) {
  const channel = context.getChannelRef();
  const originalMsg = context.getMessage();

  channel.ack(originalMsg);
}
```

### Record builders

To configure message options, you can use the `RmqRecordBuilder` class (note: this is doable for event-based flows as well). For example, to set `headers` and `priority` properties, use the `setOptions` method, as follows:

```
const message = ':cat:';
const record = new RmqRecordBuilder(message)
  .setOptions({
    headers: {
      ['x-version']: '1.0.0',
    },
    priority: 3,
  })
  .build();

this.client.send('replace-emoji', record).subscribe(...);
```

> info **Hint** `RmqRecordBuilder` class is exported from the `@nestjs/microservices` package.

And you can read these values on the server-side as well, by accessing the `RmqContext`, as follows:

```
@@filename()
@MessagePattern('replace-emoji')
replaceEmoji(@Payload() data: string, @Ctx() context: RmqContext): string
{
  const { properties: { headers } } = context.getMessage();
  return headers['x-version'] === '1.0.0' ? '🐱 ' : '🦮 ';
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('replace-emoji')
replaceEmoji(data, context) {
  const { properties: { headers } } = context.getMessage();
  return headers['x-version'] === '1.0.0' ? '🐱 ' : '🦮 ';
}
```

## Kafka

Kafka is an open source, distributed streaming platform which has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

The Kafka project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. It integrates very well with Apache Storm and Spark for real-time streaming data analysis.

### Installation

To start building Kafka-based microservices, first install the required package:

```
$ npm i --save kafkajs
```

### Overview

Like other Nest microservice transport layer implementations, you select the Kafka transporter mechanism using the `transport` property of the options object passed to the `createMicroservice()` method, along with an optional `options` property, as shown below:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.KAFKA,
  options: {
    client: {
      brokers: ['localhost:9092'],
    }
  }
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.KAFKA,
  options: {
    client: {
      brokers: ['localhost:9092'],
    }
  }
});
```

> info **Hint** The `Transport` enum is imported from the `@nestjs/microservices` package.

### Options

The `options` property is specific to the chosen transporter. The **Kafka** transporter exposes the properties described below.

| | |
|---|---|
| `client` | Client configuration options (read more here) |
| `consumer` | Consumer configuration options (read more here) |
| `run` | Run configuration options (read more here) |
| `subscribe` | Subscribe configuration options (read more here) |
| `producer` | Producer configuration options (read more here) |
| `send` | Send configuration options (read more here) |
| `producerOnlyMode` | Feature flag to skip consumer group registration and only act as a producer (`boolean`) |
| `postfixId` | Change suffix of clientId value (`string`) |

**Client**

There is a small difference in Kafka compared to other microservice transporters. Instead of the `ClientProxy` class, we use the `ClientKafka` class.

Like other microservice transporters, you have several options for creating a `ClientKafka` instance.

One method for creating an instance is to use the `ClientsModule`. To create a client instance with the `ClientsModule`, import it and use the `register()` method to pass an options object with the same properties shown above in the `createMicroservice()` method, as well as a `name` property to be used as the injection token. Read more about `ClientsModule` here.

```
@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'HERO_SERVICE',
        transport: Transport.KAFKA,
        options: {
          client: {
            clientId: 'hero',
            brokers: ['localhost:9092'],
          },
          consumer: {
            groupId: 'hero-consumer'
          }
        }
      },
    ]),
  ]
  ...
})
```

Other options to create a client (either `ClientProxyFactory` or `@Client()`) can be used as well. You can read about them here.

Use the `@Client()` decorator as follows:

```
@Client({
  transport: Transport.KAFKA,
  options: {
    client: {
      clientId: 'hero',
      brokers: ['localhost:9092'],
    },
    consumer: {
      groupId: 'hero-consumer'
    }
  }
})
client: ClientKafka;
```

**Message pattern**

The Kafka microservice message pattern utilizes two topics for the request and reply channels. The `ClientKafka#send()` method sends messages with a return address by associating a correlation id, reply topic, and reply partition with the request message. This requires the `ClientKafka` instance to be subscribed to the reply topic and assigned to at least one partition before sending a message.

Subsequently, you need to have at least one reply topic partition for every Nest application running. For example, if you are running 4 Nest applications but the reply topic only has 3 partitions, then 1 of the Nest applications will error out when trying to send a message.

When new `ClientKafka` instances are launched they join the consumer group and subscribe to their respective topics. This process triggers a rebalance of topic partitions assigned to consumers of the consumer group.

Normally, topic partitions are assigned using the round robin partitioner, which assigns topic partitions to a collection of consumers sorted by consumer names which are randomly set on application launch. However, when a new consumer joins the consumer group, the new consumer can be positioned anywhere within the collection of consumers. This creates a condition where pre-existing consumers can be assigned different partitions when the pre-existing consumer is positioned after the new consumer. As a result, the consumers that are assigned different partitions will lose response messages of requests sent before the rebalance.

To prevent the `ClientKafka` consumers from losing response messages, a Nest-specific built-in custom partitioner is utilized. This custom partitioner assigns partitions to a collection of consumers sorted by high-resolution timestamps (`process.hrtime()`) that are set on application launch.

**Message response subscription**

> warning **Note** This section is only relevant if you use request-response message style (with the `@MessagePattern` decorator and the `ClientKafka#send` method). Subscribing to the response

> topic is not necessary for the event-based communication (`@EventPattern` decorator and `ClientKafka#emit` method).

The `ClientKafka` class provides the `subscribeToResponseOf()` method. The `subscribeToResponseOf()` method takes a request's topic name as an argument and adds the derived reply topic name to a collection of reply topics. This method is required when implementing the message pattern.

```
@@filename(heroes.controller)
onModuleInit() {
  this.client.subscribeToResponseOf('hero.kill.dragon');
}
```

If the `ClientKafka` instance is created asynchronously, the `subscribeToResponseOf()` method must be called before calling the `connect()` method.

```
@@filename(heroes.controller)
async onModuleInit() {
  this.client.subscribeToResponseOf('hero.kill.dragon');
  await this.client.connect();
}
```

#### Incoming

Nest receives incoming Kafka messages as an object with `key`, `value`, and `headers` properties that have values of type `Buffer`. Nest then parses these values by transforming the buffers into strings. If the string is "object like", Nest attempts to parse the string as `JSON`. The `value` is then passed to its associated handler.

#### Outgoing

Nest sends outgoing Kafka messages after a serialization process when publishing events or sending messages. This occurs on arguments passed to the `ClientKafka` `emit()` and `send()` methods or on values returned from a `@MessagePattern` method. This serialization "stringifies" objects that are not strings or buffers by using `JSON.stringify()` or the `toString()` prototype method.

```
@@filename(heroes.controller)
@Controller()
export class HeroesController {
  @MessagePattern('hero.kill.dragon')
  killDragon(@Payload() message: KillDragonMessage): any {
    const dragonId = message.dragonId;
    const items = [
      { id: 1, name: 'Mythical Sword' },
      { id: 2, name: 'Key to Dungeon' },
    ];
```

```
    return items;
  }
}
```

> info **Hint** `@Payload()` is imported from the `@nestjs/microservices`.

Outgoing messages can also be keyed by passing an object with the `key` and `value` properties. Keying messages is important for meeting the co-partitioning requirement.

```
@@filename(heroes.controller)
@Controller()
export class HeroesController {
  @MessagePattern('hero.kill.dragon')
  killDragon(@Payload() message: KillDragonMessage): any {
    const realm = 'Nest';
    const heroId = message.heroId;
    const dragonId = message.dragonId;

    const items = [
      { id: 1, name: 'Mythical Sword' },
      { id: 2, name: 'Key to Dungeon' },
    ];

    return {
      headers: {
        realm
      },
      key: heroId,
      value: items
    }
  }
}
```

Additionally, messages passed in this format can also contain custom headers set in the `headers` hash property. Header hash property values must be either of type `string` or type `Buffer`.

```
@@filename(heroes.controller)
@Controller()
export class HeroesController {
  @MessagePattern('hero.kill.dragon')
  killDragon(@Payload() message: KillDragonMessage): any {
    const realm = 'Nest';
    const heroId = message.heroId;
    const dragonId = message.dragonId;

    const items = [
      { id: 1, name: 'Mythical Sword' },
      { id: 2, name: 'Key to Dungeon' },
    ];
```

```
    return {
      headers: {
        kafka_nestRealm: realm
      },
      key: heroId,
      value: items
    }
  }
}
```

#### Event-based

While the request-response method is ideal for exchanging messages between services, it is less suitable when your message style is event-based (which in turn is ideal for Kafka) - when you just want to publish events **without waiting for a response**. In that case, you do not want the overhead required by request-response for maintaining two topics.

Check out these two sections to learn more about this: Overview: Event-based and Overview: Publishing events.

#### Context

In more sophisticated scenarios, you may want to access more information about the incoming request. When using the Kafka transporter, you can access the `KafkaContext` object.

```
@@filename()
@MessagePattern('hero.kill.dragon')
killDragon(@Payload() message: KillDragonMessage, @Ctx() context:
KafkaContext) {
  console.log(`Topic: ${context.getTopic()}`);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('hero.kill.dragon')
killDragon(message, context) {
  console.log(`Topic: ${context.getTopic()}`);
}
```

> info **Hint** `@Payload()`, `@Ctx()` and `KafkaContext` are imported from the `@nestjs/microservices` package.

To access the original Kafka `IncomingMessage` object, use the `getMessage()` method of the `KafkaContext` object, as follows:

```
@@filename()
@MessagePattern('hero.kill.dragon')
killDragon(@Payload() message: KillDragonMessage, @Ctx() context:
```

```
KafkaContext) {
  const originalMessage = context.getMessage();
  const partition = context.getPartition();
  const { headers, timestamp } = originalMessage;
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('hero.kill.dragon')
killDragon(message, context) {
  const originalMessage = context.getMessage();
  const partition = context.getPartition();
  const { headers, timestamp } = originalMessage;
}
```

Where the `IncomingMessage` fulfills the following interface:

```
interface IncomingMessage {
  topic: string;
  partition: number;
  timestamp: string;
  size: number;
  attributes: number;
  offset: string;
  key: any;
  value: any;
  headers: Record<string, any>;
}
```

If your handler involves a slow processing time for each received message you should consider using the `heartbeat` callback. To retrieve the `heartbeat` function, use the `getHeartbeat()` method of the `KafkaContext`, as follows:

```
@@filename()
@MessagePattern('hero.kill.dragon')
async killDragon(@Payload() message: KillDragonMessage, @Ctx() context:
KafkaContext) {
  const heartbeat = context.getHeartbeat();

  // Do some slow processing
  await doWorkPart1();

  // Send heartbeat to not exceed the sessionTimeout
  await heartbeat();

  // Do some slow processing again
  await doWorkPart2();
}
```

### Naming conventions

The Kafka microservice components append a description of their respective role onto the `client.clientId` and `consumer.groupId` options to prevent collisions between Nest microservice client and server components. By default the `ClientKafka` components append `-client` and the `ServerKafka` components append `-server` to both of these options. Note how the provided values below are transformed in that way (as shown in the comments).

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.KAFKA,
  options: {
    client: {
      clientId: 'hero', // hero-server
      brokers: ['localhost:9092'],
    },
    consumer: {
      groupId: 'hero-consumer' // hero-consumer-server
    },
  }
});
```

And for the client:

```
@@filename(heroes.controller)
@Client({
  transport: Transport.KAFKA,
  options: {
    client: {
      clientId: 'hero', // hero-client
      brokers: ['localhost:9092'],
    },
    consumer: {
      groupId: 'hero-consumer' // hero-consumer-client
    }
  }
})
client: ClientKafka;
```

> info **Hint** Kafka client and consumer naming conventions can be customized by extending `ClientKafka` and `KafkaServer` in your own custom provider and overriding the constructor.

Since the Kafka microservice message pattern utilizes two topics for the request and reply channels, a reply pattern should be derived from the request topic. By default, the name of the reply topic is the composite of the request topic name with `.reply` appended.

```
@@filename(heroes.controller)
onModuleInit() {
  this.client.subscribeToResponseOf('hero.get'); // hero.get.reply
}
```

> info **Hint** Kafka reply topic naming conventions can be customized by extending `ClientKafka` in your own custom provider and overriding the `getResponsePatternName` method.

**Retriable exceptions**

Similar to other transporters, all unhandled exceptions are automatically wrapped into an `RpcException` and converted to a "user-friendly" format. However, there are edge-cases when you might want to bypass this mechanism and let exceptions be consumed by the `kafkajs` driver instead. Throwing an exception when processing a message instructs `kafkajs` to **retry** it (redeliver it) which means that even though the message (or event) handler was triggered, the offset won't be committed to Kafka.

> warning **Warning** For event handlers (event-based communication), all unhandled exceptions are considered **retriable exceptions** by default.

For this, you can use a dedicated class called `KafkaRetriableException`, as follows:

```
throw new KafkaRetriableException('...');
```

> info **Hint** `KafkaRetriableException` class is exported from the `@nestjs/microservices` package.

**Commit offsets**

Committing offsets is essential when working with Kafka. Per default, messages will be automatically committed after a specific time. For more information visit KafkaJS docs. `ClientKafka` offers a way to manually commit offsets that work like the native KafkaJS implementation.

```
@@filename()
@EventPattern('user.created')
async handleUserCreated(@Payload() data: IncomingMessage, @Ctx() context:
KafkaContext) {
  // business logic

  const { offset } = context.getMessage();
  const partition = context.getPartition();
  const topic = context.getTopic();
  await this.client.commitOffsets([{ topic, partition, offset }])
}
@@switch
@Bind(Payload(), Ctx())
@EventPattern('user.created')
async handleUserCreated(data, context) {
```

```
    // business logic

    const { offset } = context.getMessage();
    const partition = context.getPartition();
    const topic = context.getTopic();
    await this.client.commitOffsets([{ topic, partition, offset }])
  }
```

To disable auto-committing of messages set `autoCommit: false` in the `run` configuration, as follows:

```
@@filename(main)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.KAFKA,
  options: {
    client: {
      brokers: ['localhost:9092'],
    },
    run: {
      autoCommit: false
    }
  }
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.KAFKA,
  options: {
    client: {
      brokers: ['localhost:9092'],
    },
    run: {
      autoCommit: false
    }
  }
});
```

## 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');
});
```

## Custom transporters

Nest provides a variety of **transporters** out-of-the-box, as well as an API allowing developers to build new custom transport strategies. Transporters enable you to connect components over a network using a pluggable communications layer and a very simple application-level message protocol (read full article).

> info **Hint** Building a microservice with Nest does not necessarily mean you must use the `@nestjs/microservices` package. For example, if you want to communicate with external services (let's say other microservices written in different languages), you may not need all the features provided by `@nestjs/microservice` library. In fact, if you don't need decorators (`@EventPattern` or `@MessagePattern`) that let you declaratively define subscribers, running a Standalone Application and manually maintaining connection/subscribing to channels should be enough for most use-cases and will provide you with more flexibility.

With a custom transporter, you can integrate any messaging system/protocol (including Google Cloud Pub/Sub, Amazon Kinesis, and others) or extend the existing one, adding extra features on top (for example, QoS for MQTT).

> info **Hint** To better understand how Nest microservices work and how you can extend the capabilities of existing transporters, we recommend reading the NestJS Microservices in Action and Advanced NestJS Microservices article series.

### Creating a strategy

First, let's define a class representing our custom transporter.

```
import { CustomTransportStrategy, Server } from '@nestjs/microservices';

class GoogleCloudPubSubServer
  extends Server
  implements CustomTransportStrategy {
  /**
   * This method is triggered when you run "app.listen()".
   */
  listen(callback: () => void) {
    callback();
  }

  /**
   * This method is triggered on application shutdown.
   */
  close() {}
}
```

> warning **Warning** Please, note we won't be implementing a fully-featured Google Cloud Pub/Sub server in this chapter as this would require diving into transporter specific technical details.

In our example above, we declared the `GoogleCloudPubSubServer` class and provided `listen()` and `close()` methods enforced by the `CustomTransportStrategy` interface. Also, our class extends the `Server` class imported from the `@nestjs/microservices` package that provides a few useful methods, for example, methods used by Nest runtime to register message handlers. Alternatively, in case you want to extend the capabilities of an existing transport strategy, you could extend the corresponding server class, for example, `ServerRedis`. Conventionally, we added the `"Server"` suffix to our class as it will be responsible for subscribing to messages/events (and responding to them, if necessary).

With this in place, we can now use our custom strategy instead of using a built-in transporter, as follows:

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>(
  AppModule,
  {
    strategy: new GoogleCloudPubSubServer(),
  },
);
```

Basically, instead of passing the normal transporter options object with `transport` and `options` properties, we pass a single property, `strategy`, whose value is an instance of our custom transporter class.

Back to our `GoogleCloudPubSubServer` class, in a real-world application, we would be establishing a connection to our message broker/external service and registering subscribers/listening to specific channels in `listen()` method (and then removing subscriptions & closing the connection in the `close()` teardown method), but since this requires a good understanding of how Nest microservices communicate with each other, we recommend reading this article series. In this chapter instead, we'll focus on the capabilities the `Server` class provides and how you can leverage them to build custom strategies.

For example, let's say that somewhere in our application, the following message handler is defined:

```
@MessagePattern('echo')
echo(@Payload() data: object) {
  return data;
}
```

This message handler will be automatically registered by Nest runtime. With `Server` class, you can see what message patterns have been registered and also, access and execute the actual methods that were assigned to them. To test this out, let's add a simple `console.log` inside `listen()` method before `callback` function is called:

```
listen(callback: () => void) {
  console.log(this.messageHandlers);
  callback();
}
```

After your application restarts, you'll see the following log in your terminal:

```
Map { 'echo' => [AsyncFunction] { isEventHandler: false } }
```

> info **Hint** If we used the `@EventPattern` decorator, you would see the same output, but with the `isEventHandler` property set to `true`.

As you can see, the `messageHandlers` property is a `Map` collection of all message (and event) handlers, in which patterns are being used as keys. Now, you can use a key (for example, `"echo"`) to receive a reference to the message handler:

```
async listen(callback: () => void) {
  const echoHandler = this.messageHandlers.get('echo');
  console.log(await echoHandler('Hello world!'));
  callback();
}
```

Once we execute the `echoHandler` passing an arbitrary string as an argument (`"Hello world!"` here), we should see it in the console:

```
Hello world!
```

Which means that our method handler was properly executed.

When using a `CustomTransportStrategy` with Interceptors the handlers are wrapped into RxJS streams. This means that you need to subscribe to them in order to execute the streams underlying logic (e.g. continue into the controller logic after an interceptor has been executed).

An example of this can be seen below:

```
async listen(callback: () => void) {
  const echoHandler = this.messageHandlers.get('echo');
  const streamOrResult = await echoHandler('Hello World');
  if (isObservable(streamOrResult)) {
    streamOrResult.subscribe();
  }
  callback();
}
```

**Client proxy**

As we mentioned in the first section, you don't necessarily need to use the `@nestjs/microservices` package to create microservices, but if you decide to do so and you need to integrate a custom strategy, you will need to provide a "client" class too.

> info **Hint** Again, implementing a fully-featured client class compatible with all `@nestjs/microservices` features (e.g., streaming) requires a good understanding of communication techniques used by the framework. To learn more, check out this article.

To communicate with an external service/emit & publish messages (or events) you can either use a library-specific SDK package, or implement a custom client class that extends the `ClientProxy`, as follows:

```
import { ClientProxy, ReadPacket, WritePacket } from
'@nestjs/microservices';

class GoogleCloudPubSubClient extends ClientProxy {
  async connect(): Promise<any> {}
  async close() {}
  async dispatchEvent(packet: ReadPacket<any>): Promise<any> {}
  publish(
    packet: ReadPacket<any>,
    callback: (packet: WritePacket<any>) => void,
  ): Function {}
}
```

> warning **Warning** Please, note we won't be implementing a fully-featured Google Cloud Pub/Sub client in this chapter as this would require diving into transporter specific technical details.

As you can see, `ClientProxy` class requires us to provide several methods for establishing & closing the connection and publishing messages (`publish`) and events (`dispatchEvent`). Note, if you don't need a request-response communication style support, you can leave the `publish()` method empty. Likewise, if you don't need to support event-based communication, skip the `dispatchEvent()` method.

To observe what and when those methods are executed, let's add multiple `console.log` calls, as follows:

```
class GoogleCloudPubSubClient extends ClientProxy {
  async connect(): Promise<any> {
    console.log('connect');
  }

  async close() {
    console.log('close');
  }

  async dispatchEvent(packet: ReadPacket<any>): Promise<any> {
    return console.log('event to dispatch: ', packet);
  }

  publish(
    packet: ReadPacket<any>,
    callback: (packet: WritePacket<any>) => void,
  ): Function {
    console.log('message:', packet);

    // In a real-world application, the "callback" function should be
```

```
executed
    // with payload sent back from the responder. Here, we'll simply
simulate (5 seconds delay)
    // that response came through by passing the same "data" as we've
originally passed in.
    setTimeout(() => callback({ response: packet.data }), 5000);

    return () => console.log('teardown');
  }
}
```

With this in place, let's create an instance of `GoogleCloudPubSubClient` class and run the `send()` method (which you might have seen in earlier chapters), subscribing to the returned observable stream.

```
const googlePubSubClient = new GoogleCloudPubSubClient();
googlePubSubClient
  .send('pattern', 'Hello world!')
  .subscribe((response) => console.log(response));
```

Now, you should see the following output in your terminal:

```
connect
message: { pattern: 'pattern', data: 'Hello world!' }
Hello world! // <-- after 5 seconds
```

To test if our "teardown" method (which our `publish()` method returns) is properly executed, let's apply a timeout operator to our stream, setting it to 2 seconds to make sure it throws earlier then our `setTimeout` calls the `callback` function.

```
const googlePubSubClient = new GoogleCloudPubSubClient();
googlePubSubClient
  .send('pattern', 'Hello world!')
  .pipe(timeout(2000))
  .subscribe(
    (response) => console.log(response),
    (error) => console.error(error.message),
  );
```

> info **Hint** The `timeout` operator is imported from the `rxjs/operators` package.

With `timeout` operator applied, your terminal output should look as follows:

```
connect
message: { pattern: 'pattern', data: 'Hello world!' }
```

```
teardown // <-- teardown
Timeout has occurred
```

To dispatch an event (instead of sending a message), use the `emit()` method:

```
googlePubSubClient.emit('event', 'Hello world!');
```

And that's what you should see in the console:

```
connect
event to dispatch:  { pattern: 'event', data: 'Hello world!' }
```

**Message serialization**

If you need to add some custom logic around the serialization of responses on the client side, you can use a custom class that extends the `ClientProxy` class or one of its child classes. For modifying successful requests you can override the `serializeResponse` method, and for modifying any errors that go through this client you can override the `serializeError` method. To make use of this custom class, you can pass the class itself to the `ClientsModule.register()` method using the `customClass` property. Below is an example of a custom `ClientProxy` that serializes each error into an `RpcException`.

```
@@filename(error-handling.proxy)
import { ClientTcp, RpcException } from '@nestjs/microservices';

class ErrorHandlingProxy extends ClientTCP {
  serializeError(err: Error) {
    return new RpcException(err);
  }
}
```

and then use it in the `ClientsModule` like so:

```
@@filename(app.module)
@Module({
  imports: [
    ClientsModule.register({
      name: 'CustomProxy',
      customClass: ErrorHandlingProxy,
    }),
  ]
})
export class AppModule
```

> info **hint** This is the class itself being passed to `customClass`, not an instance of the class. Nest will create the instance under the hood for you, and will pass any options given to the `options` property to the new `ClientProxy`.

## Exception filters

The only difference between the HTTP exception filter layer and the corresponding microservices layer is that instead of throwing `HttpException`, you should use `RpcException`.

```
throw new RpcException('Invalid credentials.');
```

> info **Hint** The `RpcException` class is imported from the `@nestjs/microservices` package.

With the sample above, Nest will handle the thrown exception and return the `error` object with the following structure:

```
{
  "status": "error",
  "message": "Invalid credentials."
}
```

### Filters

Microservice exception filters behave similarly to HTTP exception filters, with one small difference. The `catch()` method must return an `Observable`.

```
@@filename(rpc-exception.filter)
import { Catch, RpcExceptionFilter, ArgumentsHost } from '@nestjs/common';
import { Observable, throwError } from 'rxjs';
import { RpcException } from '@nestjs/microservices';

@Catch(RpcException)
export class ExceptionFilter implements RpcExceptionFilter<RpcException> {
  catch(exception: RpcException, host: ArgumentsHost): Observable<any> {
    return throwError(() => exception.getError());
  }
}
@@switch
import { Catch } from '@nestjs/common';
import { throwError } from 'rxjs';

@Catch(RpcException)
export class ExceptionFilter {
  catch(exception, host) {
    return throwError(() => exception.getError());
  }
}
```

> warning **Warning** Global microservice exception filters aren't enabled by default when using a hybrid application.

The following example uses a manually instantiated method-scoped filter. Just as with HTTP based applications, you can also use controller-scoped filters (i.e., prefix the controller class with a `@UseFilters()` decorator).

```
@@filename()
@UseFilters(new ExceptionFilter())
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
@@switch
@UseFilters(new ExceptionFilter())
@MessagePattern({ cmd: 'sum' })
accumulate(data) {
  return (data || []).reduce((a, b) => a + b);
}
```

### Inheritance

Typically, you'll create fully customized exception filters crafted to fulfill your application requirements. However, there might be use-cases when you would like to simply extend the **core exception filter**, and override the behavior based on certain factors.

In order to delegate exception processing to the base filter, you need to extend `BaseExceptionFilter` and call the inherited `catch()` method.

```
@@filename()
import { Catch, ArgumentsHost } from '@nestjs/common';
import { BaseRpcExceptionFilter } from '@nestjs/microservices';

@Catch()
export class AllExceptionsFilter extends BaseRpcExceptionFilter {
  catch(exception: any, host: ArgumentsHost) {
    return super.catch(exception, host);
  }
}
@@switch
import { Catch } from '@nestjs/common';
import { BaseRpcExceptionFilter } from '@nestjs/microservices';

@Catch()
export class AllExceptionsFilter extends BaseRpcExceptionFilter {
  catch(exception, host) {
    return super.catch(exception, host);
  }
}
```

The above implementation is just a shell demonstrating the approach. Your implementation of the extended exception filter would include your tailored **business logic** (e.g., handling various conditions).

## Pipes

There is no fundamental difference between regular pipes and microservices pipes. The only difference is that instead of throwing `HttpException`, you should use `RpcException`.

> info **Hint** The `RpcException` class is exposed from `@nestjs/microservices` package.

### Binding pipes

The following example uses a manually instantiated method-scoped pipe. Just as with HTTP based applications, you can also use controller-scoped pipes (i.e., prefix the controller class with a `@UsePipes()` decorator).

```
@@filename()
@UsePipes(new ValidationPipe())
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
@@switch
@UsePipes(new ValidationPipe())
@MessagePattern({ cmd: 'sum' })
accumulate(data) {
  return (data || []).reduce((a, b) => a + b);
}
```

## Guards

There is no fundamental difference between microservices guards and regular HTTP application guards. The only difference is that instead of throwing `HttpException`, you should use `RpcException`.

> info **Hint** The `RpcException` class is exposed from `@nestjs/microservices` package.

### Binding guards

The following example uses a method-scoped guard. Just as with HTTP based applications, you can also use controller-scoped guards (i.e., prefix the controller class with a `@UseGuards()` decorator).

```
@@filename()
@UseGuards(AuthGuard)
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
@@switch
@UseGuards(AuthGuard)
@MessagePattern({ cmd: 'sum' })
accumulate(data) {
  return (data || []).reduce((a, b) => a + b);
}
```

## Interceptors

There is no difference between regular interceptors and microservices interceptors. The following example uses a manually instantiated method-scoped interceptor. Just as with HTTP based applications, you can also use controller-scoped interceptors (i.e., prefix the controller class with a `@UseInterceptors()` decorator).

```
@@filename()
@UseInterceptors(new TransformInterceptor())
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
@@switch
@UseInterceptors(new TransformInterceptor())
@MessagePattern({ cmd: 'sum' })
accumulate(data) {
  return (data || []).reduce((a, b) => a + b);
}
```