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 {}
```