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' ? '🐱 ' : '🦮 ';
}
```