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
    }
  }
});
```