개요

Nest는 전통적인(모놀리식이라고도 하는) 애플리케이션 아키텍처 외에도 마이크로서비스 아키텍처 개발 스타일을 기본적으로 지원합니다. 종속성 주입, 데코레이터, 예외 필터, 파이프, 가드, 인터셉터 등 이 문서의 다른 곳에서 설명하는 대부분의 개념은 마이크로서비스에도 동일하게 적용됩니다. 가능한 경우 Nest는 구현 세부 사항을 추상화하여 동일한 구성 요소가 HTTP 기반 플랫폼, WebSocket 및 마이크로서비스에서 실행될 수 있도록 합니다. 이 섹션에서는 마이크로서비스에 특화된 Nest의 측면을 다룹니다.

Nest에서 마이크로서비스는 기본적으로 HTTP와는 다른 전송 계층을 사용하는 애플리케이션입니다.

Nest는 서로 다른 마이크로서비스 인스턴스 간의 메시지 전송을 담당하는 전송자라고 하는 여러 가지 기본 제공 전송 계층 구현을 지원합니다. 대부분의 전송기는 기본적으로 요청-응답 및 이벤트 기반 메시지 스타일을 모두 지원합니다. Nest는 요청-응답 및 이벤트 기반 메시징 모두에 대한 표준 인터페이스 뒤에 있는 각 전송기의 구현 세부 사항을 추상화합니다. 따라서 애플리케이션 코드에 영향을 주지 않고도 특정 전송 계층의 특정 안정성 또는 성능 기능을 활용하기 위해 한 전송 계층에서 다른 전송 계층으로 쉽게 전환할 수 있습니다.

설치

마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

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

시작하기

마이크로서비스를 인스턴스화하려면 `NestFactory` 클래스의 `createMicroservice()` 메서드를 사용합니다:

```
@@파일명(메인)
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/microservices'에서 { Transport, MicroserviceOptions }를 임포트하고,
'./app.module'에서 { AppModule }을 임포트합니다;

비동기 함수 부트스트랩() {
  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      전송: Transport.TCP,
    },
  );
  await app.listen();
}
부트스트랩(); @@스위
치
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/microservices'에서 { Transport }를 가져옵니다;
```

```
  './app.module'에서 { AppModule }을 가져옵니다;

  비동기 함수 부트스트랩() {
    const app = await NestFactory.createMicroservice(AppModule, {
      transport: Transport.TCP,
    });
    await app.listen();
  }
  부트스트랩();
```

> 정보 힌트 마이크로서비스는 기본적으로 TCP 전송 계층을 사용합니다.

`createMicroservice()` 메서드의 두 번째 인수는 `옵션` 객체입니다. 이 객체는 두 개의 멤버로 구성될 수 있습니다:

전송자	전송자를 지정합니다(예: `Transport.NATS`).
`옵션`	트랜스포터 동작을 결정하는 트랜스포터별 옵션 객체입니다.

`옵션` 객체는 선택한 전송자에 따라 다릅니다. TCP 전송자는 아래에 설명된 속성을 노출합니다. 다른 전송자(예: Redis, MQTT 등)의 경우 사용 가능한 옵션에 대한 설명은 관련 챕터를 참조하세요.

`호스트`	연결 호스트 이름
`포트`	연결 포트
`재시도 시도`	메시지 재시도 횟수(기본값: `0`)
`재시도 지연`	메시지 재시도 시도 사이의 지연 시간(ms)(기본값: `0`)

패턴

마이크로서비스는 패턴으로 메시지와 이벤트를 모두 인식합니다. 패턴은 리터럴 객체나 문자열과 같은 일반 값입니다. 패턴은 자동으로 직렬화되어 메시지의 데이터 부분과 함께 네트워크를 통해 전송됩니다. 이러한 방식으로 메시지 발신자와 수신자는 어떤 요청이 어떤 핸들러에 의해 소비되는지 조정할 수 있습니다.

요청-응답

요청-응답 메시지 스타일은 다양한 외부 서비스 간에 메시지를 교환해야 할 때 유용합니다. 이 패러다임을 사용하면

메시지 확인 프로토콜을 수동으로 구현할 필요 없이 서비스가 실제로 메시지를 수신했는지 확인할 수 있습니다. 하지만 요청-응답 패러다임이 항상 최선의 선택은 아닙니다. 예를 들어, 로그 기반 지속성을 사용하는 스트리밍 전송자(예: 카프카 또는 NATS 스트리밍)는 이벤트 메시징 패러다임에 더 적합한 다른 범위의 문제를 해결하는 데 최적화되어 있습니다(자세한 내용은 아래 이벤트 기반 메시징 참조).

요청-응답 메시지 유형을 활성화하기 위해 Nest는 두 개의 논리 채널을 생성합니다. 한 채널은 데이터 전송을 담당하고 다른 채널은 수신 응답을 대기합니다. NATS와 같은 일부 기본 전송의 경우 이 이중 채널 지원이 기본으로 제공됩니다. 다른 전송의 경우 Nest는 수동으로 보정합니다.

별도의 채널을 만들어야 합니다. 이 경우 오버헤드가 발생할 수 있으므로 요청-응답 메시지 스타일이 필요하지 않은 경우에는 이벤트 기반 방법을 사용하는 것이 좋습니다.

요청-응답 패러다임에 기반한 메시지 핸들러를 만들려면 `@nestjs/microservices` 패키지에서 가져온 `@MessagePattern()` 데코레이터를 사용합니다. 이 데코레이터는 컨트롤러 클래스가 애플리케이션의 진입점이므로 컨트롤러 클래스 내에서만 사용해야 합니다. 프로바이더 내부에서 사용하면 Nest 런타임에서 무시되므로 아무런 효과가 없습니다.

```
@@파일명(math.controller)
'@nestjs/common'에서 { Controller }를 가져옵니다;
'@nestjs/microservices'에서 { MessagePattern }을 가져옵니다;

컨트롤러()
export class MathController {
  @MessagePattern({ cmd: 'sum' })
  accumulate(data: number[]): number {
    반환 (데이터 || []).reduce((a, b) => a + b);
  }
}
@@switch
'@nestjs/common'에서 { Controller }를 가져옵니다;
'@nestjs/microservices'에서 { MessagePattern }을 가져옵니다;

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

위 코드에서 `accumulate()` 메시지 핸들러는 `{{ '{' }} cmd: 'sum' {{ '}' }}` 메시지 패턴을 충족하는 메시지를 수신합니다. 메시지 핸들러는 클라이언트에서 전달된 `데이터라는` 단일 인수를 받습니다. 이 경우 데이터는 누적될 숫자 배열입니다.

비동기 응답

메시지 핸들러는 동기식 또는 비동기식으로 응답할 수 있습니다. 따라서 `비동기` 메서드가 지원됩니다.

```
@@파일명()
메시지 패턴({ cmd: '합계' })
async accumulate(data: number[]): Promise<number> {
  return (data || []).reduce((a, b) => a + b);
}
@@switch
메시지패턴({ cmd: '합계' }) async
accumulate(data) {
```

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

메시지 핸들러는 `Observable을` 반환할 수도 있으며, 이 경우 스트림이 완료될 때까지 결과값이 방출됩니다.

```
@@파일명()
메시지 패턴({ cmd: '합계' }) 누적(데이터: 숫자[]):
Observable<number> {
  return from([1, 2, 3]);
}
@@switch
메시지 패턴({ cmd: '합계' }) 누적(데이터: 숫자[]):
Observable<number> {
  return from([1, 2, 3]);
}
```

위의 예에서 메시지 핸들러는 배열의 각 항목에 대해 3번 응답합니다. 이벤트 기반

요청-응답 방식은 서비스 간에 메시지를 교환하는 데 이상적이지만 다음과 같은 경우에는 적합하지 않습니다.
메시지 스타일이 이벤트 기반인 경우 - 응답을 기다리지 않고 이벤트만 게시하려는 경우. 이 경우 두 채널을 유지하기 위해 요청-응답에 필요한 오버헤드를 원하지 않을 수 있습니다.

시스템의 이 부분에서 특정 조건이 발생했음을 다른 서비스에 간단히 알리고 싶다고 가정해 보겠습니다. 이것이 이벤트 기반 메시지 스타일의 이상적인 사용 사례입니다.

이벤트 핸들러를 생성하기 위해 `@EventPattern()` 데코레이터를 사용합니다.

`nestjs/microservices` 패키지.

```
@@파일명()
@EventPattern('user_created')
async handleUserCreated(data: Record<string, unknown>) {
  // 비즈니스 로직
}
@@switch
@EventPattern('user_created')
async handleUserCreated(data) {
  // 비즈니스 로직
}
```

> 정보 힌트 단일 이벤트 패턴에 대해 여러 이벤트 핸들러를 등록할 수 있으며 모든 이벤트 핸들러가 자동으로 병렬로 트리거됩니다.

`handleUserCreated()` 이벤트 핸들러는 `'user_created'` 이벤트를 수신 대기합니다. 이벤트 핸들러는 클라이언트에서 전달된 `데이터`(이 경우 네트워크를 통해 전송된 이벤트 페이로드)를 단일 인수로 받습니다.

데코레이터

보다 정교한 시나리오에서는 수신 요청에 대한 자세한 정보에 액세스하고 싶을 수도 있습니다. 예를 들어 와일드카드 구독이 있는 NATS의 경우 제작자가 메시지를 보낸 원래 제목을 얻고 싶을 수 있습니다. 마찬가지로 Kafka에서는 메시지 헤더에 액세스하고 싶을 수 있습니다. 이를 위해 다음과 같이 기본 제공 데코레이터를 사용할 수 있습니다:

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

> 정보 힌트 `@Payload()`, `@Ctx()` 및 NatsContext는 `@nestjs/microservices에서` 가져옵니다.

> 정보 힌트 `@Payload()` 데코레이터에 속성 키를 전달하여 들어오는 페이로드 객체에서 특정 속성(예: `@Payload('id')`을 추출할 수도 있습니다.

클라이언트

클라이언트 Nest 애플리케이션은 `ClientProxy` 클래스를 사용하여 메시지를 교환하거나 Nest 마이크로서비스에 이벤트를 게시할 수 있습니다. 이 클래스는 원격 마이크로서비스와 통신할 수 있는 `send()`(요청-응답 메시징용) 및 `emit()`(이벤트 중심 메시징용)와 같은 여러 메서드를 정의합니다. 다음 방법 중 하나를 사용하여 이 클래스의 인스턴스를 가져옵니다.

한 가지 기법은 정적 `register()` 메서드를 노출하는 `ClientsModule을` 임포트하는 것입니다. 이 메서드

는 마이크로서비스 전송자를 나타내는 객체 배열인 인수를 받습니다. 이러한 각 객체에는 `이름` 속성, 선택적 `전송` 속성(기본값은 `Transport.TCP`) 및 선택적 트랜스포터별 `옵션` 속성이 있습니다.

`name` 속성은 필요한 경우 `ClientProxy` 인스턴스를 주입하는 데 사용할 수 있는 주입 토큰 역할을 합니다. `이름` 속성의 값은 인젝션 토큰으로서 여기에 설명된 대로 임의의 문자열 또는 JavaScript 심볼일 수 있습니다.

`옵션` 프로퍼티는 `createMicroservice()`에서 본 것과 동일한 프로퍼티를 가진 객체입니다.

메서드를 사용합니다.

```
모듈({ import: [
    ClientsModule.register([
      { name: 'MATH_SERVICE', transport: Transport.TCP },
    ]),
  ]
  ...
})
```

모듈을 임포트한 후에는 위에 표시된 `'MATH_SERVICE'` 트랜스포터 옵션을 통해 지정된 대로 구성된 `ClientProxy` 인스턴스를 `@Inject()` 데코레이터를 사용하여 주입할 수 있습니다.

```
생성자(
  @Inject('MATH_SERVICE') 비공개 클라이언트: ClientProxy,
) {}
```

> 정보 힌트 `클라이언트` 모듈과 `클라이언트 프록시` 클래스는 `nestjs/microservices` 패키지.

때로는 클라이언트 애플리케이션에서 트랜스포터 구성을 하드코딩하는 대신 다른 서비스(예: `컨피그서비스`)에서 가져와야 할 수도 있습니다. 이를 위해 `ClientProxyFactory` 클래스를 사용하여 사용자 정의 프로바이더를 등록할 수 있습니다. 이 클래스에는 정적 `create()` 메서드가 있는데, 이 메서드는 트랜스포터 옵션 객체를 받아들이고 사용자 정의된 `ClientProxy` 인스턴스를 반환합니다.

```
모듈({ providers: [
    {
      제공: 'math_service',
      사용 팩토리: (config서비스: 구성 서비스) => {
        const mathSvcOptions = configService.getMathSvcOptions();
        return ClientProxyFactory.create(mathSvcOptions);
      },
      주입합니다: [구성 서비스],
    }
  ]
  ...
})
```

> 정보 힌트 ClientProxyFactory는 `@nestjs/microservices` 패키지에서 가져옵니다.

또 다른 옵션은 `@Client()` 속성 데코레이터를 사용하는 것입니다.

```
@Client({ 전송: Transport.TCP }) 클라이언
트: ClientProxy;
```

> 정보 힌트 `@Client()` 데코레이터는 `@nestjs/microservices` 패키지에서 가져옵니다.

클라이언트 인스턴스를 테스트하기 어렵고 공유하기 어렵기 때문에 `@Client()` 데코레이터를 사용하는 것은 선호되지 않습니다.

`클라이언트 프록시는` 게으르다. 즉시 연결을 시작하지 않습니다. 대신 첫 번째 마이크로서비스 호출 전에 설정된 다음 각 후속 호출에서 재사용됩니다. 그러나 연결이 설정될 때까지 애플리케이션 부트스트랩 프로세스를 지연시키려면 `OnApplicationBootstrap` 라이프사이클 후크 내에서 `ClientProxy` 객체의 `connect()` 메서드를 사용하여 수동으로 연결을 시작할 수 있습니다.

```
@@파일명()
async onApplicationBootstrap() {
  await this.client.connect();
}
```

연결을 만들 수 없는 경우 `connect()` 메서드는 해당 오류 객체와 함께 거부됩니다. 메시지 보내기

`ClientProxy는 send()` 메서드를 노출합니다. 이 메서드는 마이크로서비스를 호출하고
옵저버블에 응답을 추가합니다. 따라서 방출된 값을 쉽게 구독할 수 있습니다.

```
@@파일명()
accumulate(): Observable<number> {
  const pattern = { cmd: 'sum' };
  const payload = [1, 2, 3];
  이.클라이언트.보내기<번호>(패턴, 페이로드)를 반환합니다;
}
@@switch
accumulate() {
  const pattern = { cmd: '합계' };
  const payload = [1, 2, 3];
  이.클라이언트.보내기(패턴, 페이로드)를 반환합니다;
}
```

`send()` 메서드는 패턴과 `페이로드라는` 두 개의 인수를 받습니다. `패턴은 @MessagePattern()` 데코레이터에 정의된 `패턴과` 일치해야 합니다. `페이로드는` 원격 마이크로서비스로 전송하려는 메시지입니다. 이 메서드는 콜드 **옵저버블을** 반환하므로 메시지가 전송되기 전에 명시적으로 옵저버블을 구독해야 합니다.

이벤트 게시

이벤트를 전송하려면 `ClientProxy` 객체의 `emit()` 메서드를 사용합니다. 이 메서드는 메시지 브로커에 이벤트를 게시합니다.

```
```

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

emit() 메서드는 패턴과 페이로드라는 두 개의 인수를 받습니다. 패턴은 @EventPattern() 데코레이터에 정의된 패턴과 일치해야 합니다. 페이로드는 원격 마이크로서비스로 전송하려는 이벤트 페이로드입니다. 이 메서드는 send()가 반환하는 콜드 옵저버블과 달리 핫 옵저버블을반환하므로, 명시적으로 옵저버블을 구독하는지 여부와 관계없이 프록시는 즉시 이벤트를 전달하려고 시도합니다.

범위

다른 프로그래밍 언어 배경을 가진 사람들에게는 Nest에서 거의 모든 것이 들어오는 요청에서 공유된다는 사실이 의외로 느껴질 수 있습니다. 데이터베이스에 대한 연결 풀, 전역 상태를 가진 싱글톤 서비스 등이 있습니다. Node.js는 모든 요청이 별도의 스레드에서 처리되는 요청/응답 다중 스레드 상태 비저장 모델을 따르지 않는다는 점을 기억하세요. 따라서 싱글톤 인스턴스를 사용하는 것은 애플리케이션에 완전히 안전합니다.

그러나 GraphQL 애플리케이션의 요청별 캐싱, 요청 추적 또는 멀티테넌시와 같이 핸들러의 요청 기반 수명이 원하는 동작일 수 있는 예외적인 경우가 있습니다. 여기에서 범위를 제어하는 방법을 알아보세요.

요청 범위가 지정된 핸들러와 프로바이더는 @Inject() 데코레이터를 CONTEXT 토큰과 함께 사용하여 RequestContext를 주입할 수 있습니다:

```
'@nestjs/common'에서 { Injectable, Scope, Inject }를 가져오고,
'@nestjs/microservices'에서 { CONTEXT, RequestContext }를 가져옵니다
;

주입 가능({ 범위: Scope.REQUEST }) 내보내기
클래스 CatsService {
  생성자(@Inject(CONTEXT) private ctx: RequestContext) {}
}
```

이렇게 하면 두 가지 프로퍼티가 있는 RequestContext 객체에 액세스할 수 있습니다:

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

`데이터` 속성은 메시지 제작자가 보낸 메시지 페이로드입니다. `패턴` 속성은 수신 메시지를 처리할 적절한 핸들러를 식별하는 데 사용되는 패턴입니다.

시간 초과 처리

분산 시스템에서는 때때로 마이크로서비스가 다운되거나 사용할 수 없는 경우가 있습니다. 무한히 오래 기다리지 않으려면 타임아웃을 사용할 수 있습니다. 타임아웃은 다른 서비스와 통신할 때 매우 유용한 패턴입니다. 마이크로서비스 호출에 타임아웃을 적용하려면 RxJS `타임아웃` 연산자를 사용하면 됩니다. 마이크로서비스가 특정 시간 내에 요청에 응답하지 않으면 예외가 발생하고, 이를 포착하여 적절히 처리할 수 있습니다.

이 문제를 해결하려면 `rxjs` 패키지를 사용해야 합니다. 파이프에서 `타임아웃` 연산자를 사용하기만 하면 됩니다:

```
@@파일명()
this.client
      .send<TResult, TInput>(패턴, 데이터)
      .pipe(timeout(5000));
@@switch
this.client
      .send(패턴, 데이터)
      .pipe(timeout(5000));
```

> 정보 힌트 `타임아웃` 연산자는 `rxjs/operators` 패키지에서 가져옵니다.

5초 후에도 마이크로서비스가 응답하지 않으면 오류가 발생합니다.

Redis

Redis 트랜스포터는 게시/구독 메시징 패러다임을 구현하고 Redis의 게시/구독 기능을 활용합니다. 게시된 메시지는 어떤 구독자(있는 경우)가 최종적으로 메시지를 수신할지 알 수 없이 채널로 분류됩니다. 각 마이크로서비스는 원하는 수의 채널에 구독할 수 있습니다. 또한 한 번에 하나 이상의 채널을 구독할 수 있습니다. 채널을 통해 교환되는 메시지는 다음과 같습니다.

즉, 메시지가 게시된 후 해당 메시지에 관심 있는 구독자가 없는 경우 메시지를 삭제합니다,
를 누르면 메시지가 삭제되어 복구할 수 없습니다. 따라서 다음과 같은 메시지가 삭제되지 않는다는 보장이 없습니다.
또는 이벤트가 하나 이상의 서비스에서 처리됩니다. 하나의 메시지를 여러 가입자가 구독하고 수신할 수 있습니다.

설치

Redis 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

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

개요

Redis 트랜스포터를 사용하려면 `createMicroservice()` 메서드에 다음 옵션 객체를 전달합니다:

```
@@파일명(메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.REDIS, 옵션:
  {
    호스트: 'localhost',
    포트: 6379,
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.REDIS,
  옵션: {
    호스트: 'localhost',
    포트: 6379,
  },
});
```

정보 힌트 `Transport` 열거형은 `@nestjs/microservices` 패키지에서 가져옵니다.

옵션

옵션 속성은 선택한 트랜스포터에 따라 다릅니다. Redis 트랜스포터는 아래에 설명된 속성을 노출합니다.

호스트	연결 URL
포트	연결 포트
재시도 시도	메시지 재시도 횟수(기본값: 0) 재시도 지연 메시지 재시도 시도 사이의 지연 시간(ms)(기본값: 0)
와일드카드	Redis 윌카드 구독을 활성화하여 트랜스포터가 다음을 사용하도록 지시합니다. 구독/메시지 내부를 살펴봅니다. (기본값: false)

공식 아이오레디스 클라이언트에서 지원하는 모든 프로퍼티는 이 트랜스포터에서도 지원됩니다. 클

라이언트

다른 마이크로서비스 전송기와 마찬가지로, Redis ClientProxy 인스턴스를 생성하기 위한 몇 가지 옵션이 있습니다.

인스턴스를 생성하는 한 가지 방법은 ClientsModule을 사용하는 것입니다. ClientsModule을 사용하여 클라이언트 인스턴스를 만들려면 인스턴스를 임포트한 후 register() 메서드를 사용하여 위에서 createMicroservice() 메서드에 표시된 것과 동일한 속성을 가진 옵션 객체와 주입 토큰으로 사용할 이름 속성을 전달합니다. 클라이언트 모듈에 대한 자세한 내용은 여기를 참조하세요.

```
모듈({ import: [
    ClientsModule.register([
      {
        이름: 'MATH_SERVICE',
        transport: Transport.REDIS, 옵션
        : {
          호스트: 'localhost', 포
          트: 6379,
        }
      },
    ]),
  ]
  ...
})
```

클라이언트를 생성하는 다른 옵션(`ClientProxyFactory` 또는 `@Client()`)도 사용할 수 있습니다. 자세한 내용은 여기에서 확인할 수 있습니다.

컨텍스트

보다 정교한 시나리오에서는 들어오는 요청에 대한 자세한 정보에 액세스하고 싶을 수 있습니다. Redis 트랜스포터 사용 시, `RedisContext` 객체에 액세스할 수 있습니다.

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: RedisContext)
{
  console.log(`채널: ${context.getChannel()}`);
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
  console.log(`채널: ${context.getChannel()}`);
}
```

> 정보 힌트 `@Payload()`, `@Ctx()` 및 RedisContext는 `nestjs/microservices` 패키지.

MQTT

MQTT(메시지 큐 텔레메트리 전송)는 짧은 지연 시간에 최적화된 오픈 소스 경량 메시징 프로토콜입니다. 이 프로토콜은 게시/구독 모델을 사용하여 장치를 연결하는 확장 가능하고 비용 효율적인 방법을 제공합니다. MQTT를 기반으로 구축된 통신 시스템은 퍼블리싱 서버, 브로커 및 하나 이상의 클라이언트로 구성됩니다. 제한된 장치와 낮은 대역폭, 높은 지연 시간 또는 불안정한 네트워크를 위해 설계되었습니다.

설치

MQTT 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

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

개요

MQTT 트랜스포터를 사용하려면 `createMicroservice()` 메서드에 다음 옵션 객체를 전달합니다:

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

> 정보 힌트 `Transport` 열거형은 `@nestjs/microservices` 패키지에서 가져옵니다.

옵션

`옵션` 객체는 선택한 트랜스포터에 따라 다릅니다. MQTT 트랜스포터는 여기에 설명된 속성을 노출합니다.

클라이언트

다른 마이크로서비스 전송기와 마찬가지로 MQTT `ClientProxy` 인스턴스를 생성하는 데는 여러 가지 옵션이 있습니다.

인스턴스를 생성하는 한 가지 방법은 `ClientsModule을` 사용하는 것입니다. `ClientsModule을` 사용하여 클라이언트 인스턴스를 만들려면 인스턴스를 임포트한 후 `register()` 메서드를 사용하여 위에서 `createMicroservice()` 메서드에 표시된 것과 동일한 속성을 가진 옵션 객체와 주입 토큰으로 사용할 `이름` 속성을 전달합니다. `클라이언트 모듈에` 대한 자세한 내용은 여기를 참조하세요.

```
모듈({ import: [
    ClientsModule.register([
      {
        이름: 'MATH_SERVICE',
        전송: Transport.MQTT, 옵션: {
          url: 'mqtt://localhost:1883',
        }
      },
    ]),
  ]
  ...
})
```

클라이언트를 생성하는 다른 옵션(`ClientProxyFactory` 또는 `@Client()`)도 사용할 수 있습니다. 자세한 내용은 여기에서 확인할 수 있습니다.

컨텍스트

보다 정교한 시나리오에서는 들어오는 요청에 대한 자세한 정보에 액세스하고 싶을 수 있습니다. MQTT 트랜스포터를 사용하는 경우 `MqttContext` 객체에 액세스할 수 있습니다.

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: MqttContext) {
  console.log(`Topic: ${context.getTopic()}`);
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
  console.log(`Topic: ${context.getTopic()}`);
}
```

```
@@파일명() @메시지패턴('알림')
```

정보 힌트 `@Payload()`, `@Ctx()` 및 MqttContext는 `nestjs/microservices` 패키지.

원본 mqtt 패킷에 액세스하려면 다음과 같이 `MqttContext` 객체의 `getPacket()` 메서드를 사용합니다:

```
  getNotifications(@Payload() data: number[], @Ctx() context: MqttContext) {
    console.log(context.getPacket());
  }
  @@switch
  @Bind(Payload(), Ctx())
  메시지 패턴('알림') getNotifications(데이터,
  컨텍스트) {
    console.log(context.getPacket());
  }
```

와일드카드

구독은 명시적인 주제에 대한 구독이거나 와일드카드를 포함할 수 있습니다. 와일드카드는 +와 # 두 가지를 사용할 수 있습니다.

+ 는 단일 레벨 와일드카드이고 #는 여러 주제 레벨을 포괄하는 다중 레벨 와일드카드입니다.

```
  @@ 파일 이름 () @ 메시지 패턴('센서/+ / 온도/+')
  getTemperature(@Ctx() context: MqttContext) {
    console.log(`Topic: ${context.getTopic()}`);
  }
  @@switch
  @Bind(Ctx())
  @MessagePattern('sensors/+/temp/+') getTemp(context)
  {
    console.log(`Topic: ${context.getTopic()}`);
  }
```

레코드 빌더

메시지 옵션을 구성하려면(QoS 수준 조정, Retain 또는 DUP 플래그 설정, 페이로드에 추가 속성 추가 등) `MqttRecordBuilder` 클래스를 사용하면 됩니다. 예를 들어, QoS를 `2로` 설정하려면 다음과 같이 `setQoS` 메서드를 사용합니다:

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

  @@파일명() @메시지패턴('이모티콘 대체')
```

> 정보 힌트 `MqttRecordBuilder` 클래스는 `@nestjs/microservices` 패키지에서 내보내집니다.

또한 서버 측에서도 `MqttContext에` 액세스하여 이러한 옵션을 읽을 수 있습니다.

```
  replaceEmoji(@Payload() 데이터: 문자열, @Ctx() 컨텍스트: MqttContext): 문자열
  {
    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' ? '□ ' : '🐈 ';
  }
```

여러 요청에 대해 사용자 속성을 구성해야 하는 경우도 있는데, 이러한 옵션을 `ClientProxyFactory에` 전달할 수 있습니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/microservices'에서 { ClientProxyFactory, Transport }를 가져옵니다;

모듈({ providers:
  [
    {
      제공: 'API_v1', useFactory:
      () =>.
        ClientProxyFactory.create({
          transport: Transport.MQTT,
          options: {
            url: 'mqtt://localhost:1833',
            userProperties: { 'x-version': '1.0.0' },
          },
        }),
    },
  ],
})
내보내기 클래스 ApiModule {}
```

NATS

NATS는 클라우드 네이티브 애플리케이션, IoT 메시징 및 마이크로서비스 아키텍처를 위한 간단하고 안전한 고성능 오픈 소스 메시징 시스템입니다. NATS 서버는 Go 프로그래밍 언어로 작성되었지만 서버와 상호 작용하는 클라이언트 라이브러리는 수십 개의 주요 프로그래밍 언어로 사용할 수 있습니다. NATS는 최대 한 번 전송과 최소 한 번 전송을 모두 지원합니다. 대규모 서버와 클라우드 인스턴스, 엣지 게이트웨이, 심지어 사물 인터넷 장치에 이르기까지 어디서나 실행할 수 있습니다.

설치

NATS 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

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

개요

NATS 트랜스포터를 사용하려면 `createMicroservice()` 메서드에 다음 옵션 객체를 전달합니다:

```
@@파일명(메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.NATS, 옵션:
  {
    서버: ['nats://localhost:4222'],
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.NATS,
  옵션: {
    서버: ['nats://localhost:4222'],
  },
});
```

> 정보 힌트 `Transport` 열거형은 `@nestjs/microservices` 패키지에서 가져옵니다.

옵션

`옵션` 개체는 선택한 트랜스포터에 따라 다릅니다. NATS 트랜스포터는 여기에 설명된 속성을 노출합니다. 또한

서버가 구독해야 하는 큐의 이름을 지정할 수 있는 큐 속성이 있습니다(이 설정을 무시하려면 정의되지 않은 상태로 두세요). 아래에서 NATS 큐 그룹에 대해 자세히 알아보세요.

클라이언트

다른 마이크로서비스 전송기와 마찬가지로 NATS `ClientProxy` 인스턴스를 생성하는 데는 몇 가지 옵션이 있습니다.

인스턴스를 생성하는 한 가지 방법은 `ClientsModule`을 사용하는 것입니다. `ClientsModule`을 사용하여 클라이언트 인스턴스를 만들려면 인스턴스를 임포트한 후 `register()` 메서드를 사용하여 위에서 `createMicroservice()` 메서드에 표시된 것과 동일한 속성을 가진 옵션 객체와 주입 토큰으로 사용할 `이름` 속성을 전달합니다. `클라이언트 모듈에` 대한 자세한 내용은 여기를 참조하세요.

```
모듈({ import: [
    ClientsModule.register([
      {
        이름: 'MATH_SERVICE',
        transport: Transport.NATS, 옵션:
        {
          서버: ['nats://localhost:4222'],
        }
      },
    ]),
  ]
  ...
})
```

클라이언트를 생성하는 다른 옵션(`ClientProxyFactory` 또는 `@Client()`)도 사용할 수 있습니다. 자세한 내용은 여기에서 확인할 수 있습니다.

요청-응답

요청-응답 메시지 스타일(자세히 읽기)의 경우, NATS 전송자는 NATS 기본 제공 요청-응답 메커니즘을 사용하지 않습니다. 대신, "요청"은 `게시()` 함수를 사용하여 지정된 주제에 게시됩니다.

메서드에 고유한 회신 제목 이름을 지정하면 응답자는 해당 제목을 수신하고 회신 제목에 응답을 보냅니다. 회신 제목은 양쪽 당사자의 위치에 관계없이 요청자에게 동적으로 다시 전달됩니다.

이벤트 기반

이벤트 기반 메시지 스타일(자세히 읽기)의 경우 NATS 트랜스포터는 NATS에 내장된 게시-구독 메커니즘을 사용합니다. 게시자는 주제에 대한 메시지를 보내고 수신 중인 활성 구독자가 있는 경우

를 입력하면 해당 주체가 메시지를 받습니다. 또한 구독자는 정규식처럼 작동하는 와일드카드 주제에 관심을 등록할 수도 있습니다. 이러한 일대다 패턴을 팬아웃이라고도 합니다.

대기열 그룹

NATS는 분산 큐라는 기본 제공 부하 분산 기능을 제공합니다. 큐 구독을 만들려면 다음과 같이 `큐` 속성을 사용합니다:

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

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

컨텍스트

보다 정교한 시나리오에서는 들어오는 요청에 대한 자세한 정보에 액세스하고 싶을 수 있습니다. NATS 트랜스포터를 사용하는 경우 `NatsContext` 객체에 액세스할 수 있습니다.

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`);
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
  console.log(`Subject: ${context.getSubject()}`);
}
```

> 정보 힌트 `@Payload()`, `@Ctx()` 및 NatsContext는 `nestjs/microservices` 패키지.

와일드카드

구독은 명시적인 주제에 대한 구독일 수도 있고 와일드카드를 포함할 수도 있습니다.

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

레코드 빌더

메시지 옵션을 구성하려면 `NatsRecordBuilder` 클래스를 사용할 수 있습니다(참고: 이벤트 기반 흐름에서도 이 작업이 가능합니다). 예를 들어, `x 버전` 헤더를 추가하려면 다음과 같이 `setHeaders` 메서드를 사용합니다:

```
'nats'에서 *를 nats로 가져옵니다;

// 코드 어딘가에 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(...);
```

> 정보 힌트 `NatsRecordBuilder` 클래스는 `@nestjs/microservices` 패키지에서 내보내집니다.

서버 측에서도 다음과 같이 `NatsContext에` 액세스하여 이러한 헤더를 읽을 수 있습니다:

```
@@파일명() @메시지패턴('이모티콘 대체')
replaceEmoji(@Payload() 데이터: 문자열, @Ctx() context: NatsContext): 문자열
{
  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' ? '□ ' : '🐱 ';
}
```

경우에 따라 여러 요청에 대한 헤더를 구성하고 싶을 수 있으며, 이를 `ClientProxyFactory에` 옵션으로 전달할 수 있습니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/microservices'에서 { ClientProxyFactory, Transport }를 가져옵니다;

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

```
          }),
      },
    ],
})
내보내기 클래스 ApiModule {}
```

RabbitMQ

RabbitMQ는 여러 메시징 프로토콜을 지원하는 오픈소스 경량 메시지 브로커입니다. 분산 및 연합 구성으로 배포하여 대규모, 고가용성 요구 사항을 충족할 수 있습니다. 또한, 전 세계적으로 소규모 스타트업과 대기업에서 가장 널리 배포된 메시지 브로커입니다.

설치

RabbitMQ 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

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

개요

RabbitMQ 트랜스포터를 사용하려면 `createMicroservice()`에 다음 옵션 객체를 전달합니다.

메서드를 사용합니다:

```
@@파일명(메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.RMQ, 옵션: {
    urls: ['amqp://localhost:5672'],
    queue: 'cats_queue',
    queueOptions: {
      내구성: 거짓
    },
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.RMQ,
  옵션: {
    urls: ['amqp://localhost:5672'],
    queue: 'cats_queue',
    queueOptions: {
      내구성: 거짓
    },
  },
});
```

> 정보 힌트 `Transport` 열거형은 `@nestjs/microservices` 패키지에서 가져옵니다.

옵션

옵션 속성은 선택한 트랜스포터에 따라 다릅니다. RabbitMQ 트랜스포터는 아래에 설명된 속성을 노출합니다.

URL	연결 URL
큐	서버가 수신 대기할 대기열 이름 프리페치 카
운트	채널에 대한 프리페치 횟수를 설정합니다.
채널별 프리페치 활성화	
noAck	거짓이면 수동 승인 모드가 활성화됩니다.
queueOptions	추가 대기열 옵션(자세한 내용은 여기를 참조
하세요) 소켓 옵션	추가 소켓 옵션(여기에서 자세히 읽기) 헤더
	모든 메시지와 함께 전송할 헤더

클라이언트

다른 마이크로서비스 전송기와 마찬가지로 RabbitMQ ClientProxy를 생성하기 위한 몇 가지 옵션이 있습니다. 인스턴스입니다.

인스턴스를 생성하는 한 가지 방법은 ClientsModule을 사용하는 것입니다. ClientsModule을 사용하여 클라이언트 인스턴스를 만들려면 인스턴스를 임포트한 후 register() 메서드를 사용하여 위에서 createMicroservice() 메서드에 표시된 것과 동일한 속성을 가진 옵션 객체와 주입 토큰으로 사용할 이름 속성을 전달합니다. 클라이언트 모듈에 대한 자세한 내용은 여기를 참조하세요.

```
모듈({ import: [
    ClientsModule.register([
      {
        이름: 'MATH_SERVICE',
        transport: Transport.RMQ, 옵션:
        {
          urls: ['amqp://localhost:5672'],
          queue: 'cats_queue',
          queueOptions: {
            내구성: 거짓
          },
        },
      },
    ]),
  ]
  ...
})
```

클라이언트를 생성하는 다른 옵션(`ClientProxyFactory` 또는 `@Client()`)도 사용할 수 있습니다. 자세한 내용은 여기에서 확인할 수 있습니다.

컨텍스트

보다 정교한 시나리오에서는 들어오는 요청에 대한 자세한 정보에 액세스하고 싶을 수 있습니다. RabbitMQ 트랜스포터 사용 시, `RmqContext` 객체에 액세스할 수 있습니다.

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  console.log(`패턴: ${context.getPattern()}`);
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
  console.log(`패턴: ${context.getPattern()}`);
}
```

> 정보 힌트 `@Payload()`, `@Ctx()` 및 RmqContext는 `nestjs/microservices` 패키지.

`속성`, `필드` 및 콘텐츠가 포함된 원본 RabbitMQ 메시지에 액세스하려면 `getMessage()` 메서드를 다음과 같이 호출합니다:

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  console.log(context.getMessage());
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
  console.log(context.getMessage());
}
```

RabbitMQ 채널에 대한 참조를 검색하려면 `RmqContext의 getChannelRef` 메서드를 사용합니다. 객체를 다음과 같이 설정합니다:

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  console.log(context.getChannelRef());
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
  console.log(context.getChannelRef());
}
```

메시지 확인

메시지가 손실되지 않도록 하기 위해 RabbitMQ는 메시지 확인을 지원합니다. 확인은 소비자가 다시 전송하여 특정 메시지가 수신, 처리되었으며 RabbitMQ가 자유롭게 삭제할 수 있음을 RabbitMQ에 알립니다. 소비자가 응답을 보내지 않고 죽으면(채널이 닫히거나, 연결이 끊기거나, TCP 연결이 끊어지면) RabbitMQ는 메시지가 완전히 처리되지 않았다는 것을 이해하고 다시 대기열에 넣습니다.

수동 승인 모드를 사용하려면 `noAck` 속성을 `false로` 설정합니다:

```
옵션: {
  urls: ['amqp://localhost:5672'],
  queue: 'cats_queue',
  noAck: false,
  queueOptions: {
    내구성: 거짓
  },
},
```

수동 소비자 확인이 켜져 있는 경우 작업자가 작업을 완료했음을 알리기 위해 적절한 확인을 보내야 합니다.

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

  channel.ack(originalMsg);
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
  const channel = context.getChannelRef();
  const originalMsg = context.getMessage();

  channel.ack(originalMsg);
}
```

레코드 빌더

메시지 옵션을 구성하려면 `RmqRecordBuilder` 클래스를 사용할 수 있습니다(참고: 이벤트 기반 플로우에서도

이 작업을 수행할 수 있음). 예를 들어 `헤더` 및 `우선순위` 속성을 설정하려면 다음과 같이 `setOptions` 메서드를 사용합니다:

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

이.클라이언트.보내기('이모티콘 교체', 레코드).구독(...);
```

> 정보 힌트 `RmqRecordBuilder` 클래스는 `@nestjs/microservices` 패키지에서 내보내집니다.

또한 서버 측에서도 다음과 같이 `RmqContext에` 액세스하여 이러한 값을 읽을 수 있습니다:

```
@@파일명() @메시지패턴('이모티콘 대체')
replaceEmoji(@Payload() 데이터: 문자열, @Ctx() 컨텍스트: RmqContext): 문자열
{
  const { 속성: { 헤더 } } = context.getMessage(); return
  headers['x-version'] === '1.0.0' ? '□ ' : '🐈 ';
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('replace-emoji')
replaceEmoji(data, context) {
  const { 속성: { 헤더 } } = context.getMessage(); return
  headers['x-version'] === '1.0.0' ? '□ ' : '🐈 ';
}
```

카프카

카프카는 오픈 소스 분산 스트리밍 플랫폼으로, 세 가지 주요 기능을 갖추고 있습니다:

- 메시지 큐 또는 엔터프라이즈 메시징 시스템과 유사하게 레코드 스트림을 게시하고 구독합니다.
- 내결함성 내구성 있는 방식으로 기록 스트림을 저장하세요.
- 레코드 스트림이 발생하는 대로 처리합니다.

카프카 프로젝트는 실시간 데이터 피드를 처리하기 위해 처리량이 많고 지연 시간이 짧은 통합 플랫폼을 제공하는 것을 목표로 합니다. 실시간 스트리밍 데이터 분석을 위해 Apache Storm 및 Spark와 매우 잘 통합됩니다.

설치

Kafka 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

```
$ npm i -- 저장 카프카즈
```

개요

다른 Nest 마이크로서비스 전송 계층 구현과 마찬가지로, 아래와 같이 `createMicroservice()` 메서드에 전달된 옵션 객체의 `전송` 속성을 옵션 `옵션` 속성과 함께 사용하여 Kafka 전송기 메커니즘을 선택합니다:

```
@@파일명(메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  운송: Transport.KAFKA, 옵션: {
    클라이언트: {
      브로커: ['localhost:9092'],
    }
  }
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.KAFKA,
  옵션: { client:
    {
      브로커: ['localhost:9092'],
    }
  }
});
```

정보 힌트 `Transport` 열거형은 `@nestjs/microservices` 패키지에서 가져옵니다.

옵션

옵션 속성은 선택한 트랜스포터에 따라 다릅니다. 카프카 트랜스포터는 아래에 설명된 속성을 노출합니다.

클라이언트	클라이언트 구성 옵션(여기에서 자세히 읽기) 소비
자	소비자 구성 옵션(여기에서 자세히 읽기) 실행
	실행 구성 옵션(여기에서 자세히 읽기) 구독하기
	구독 구성 옵션(여기에서 자세히 읽기) 생산자
	프로듀서 구성 옵션(여기에서 자세히 읽기) 보내기
프로듀서 전용 모드	보내기 구성 옵션(여기에서 자세히 읽기) 소비자 그룹 등록을 건너뛰고 생산자 역할만 수행하는 기능 플래그 (부울)
postfixId	clientId 값의 접미사 변경(문자열)

클라이언트

다른 마이크로서비스 전송자와 비교했을 때 Kafka에는 약간의 차이가 있습니다. 대신 ClientProxy 클래스를 사용합니다.

다른 마이크로서비스 전송자와 마찬가지로, ClientKafka 인스턴스를 생성하는 데는 몇 가지 옵션이 있습니다. 인스턴스를 생성하는 한 가지 방법은 ClientsModule을 사용하는 것입니다. 클라이언트 인스턴스를 생성하려면 클라이언트 모듈을 가져온 다음 register() 메서드를 사용하여 위에서 createMicroservice() 메서드에 표시된 것과 동일한 속성을 가진 옵션 객체와 인젝션 토큰으로 사용할 이름 속성을 전달합니다. 클라이언트 모듈에 대한 자세한 내용은 여기를 참조하세요.

```
모듈({ import: [
    ClientsModule.register([
      {
        이름: 'HERO_SERVICE',
        운송: Transport.KAFKA, 옵션: {
          클라이언트: {
            clientId: 'hero',
            브로커: ['localhost:9092'],
          },
          소비자: {
            groupId: 'hero-consumer'
          }
        }
      },
    ]),
  ]
  ...
})
```

클라이언트를 생성하는 다른 옵션(`ClientProxyFactory` 또는 `@Client()`)도 사용할 수 있습니다. 자세한 내용은 여기에서 확인할 수 있습니다.

다음과 같이 `@Client()` 데코레이터를 사용합니다:

```
@Client({
  운송: Transport.KAFKA, 옵션: {
    클라이언트: {
      clientId: 'hero',
      브로커: ['localhost:9092'],
    },
    소비자: {
      groupId: 'hero-consumer'
    }
  }
})
클라이언트: ClientKafka;
```

메시지 패턴

Kafka 마이크로서비스 메시지 패턴은 요청 채널과 응답 채널에 두 개의 토픽을 사용합니다. `ClientKafka#send()` 메서드는 상관관계 ID, 회신 토픽, 회신 파티션을 요청 메시지와 연결하여 반환 주소가 포함된 메시지를 전송합니다. 이를 위해서는 메시지를 보내기 전에 `ClientKafka` 인스턴스가 회신 토픽에 가입되어 있어야 하고 적어도 하나의 파티션에 할당되어야 합니다.

따라서 실행 중인 모든 Nest 애플리케이션에 대해 회신 주제 파티션이 하나 이상 있어야 합니다. 예를 들어 4개의 Nest 애플리케이션을 실행 중인데 답장 주제에 3개의 파티션만 있는 경우에는 메시지를 보내려고 할 때 Nest 애플리케이션 중 1개에서 오류가 발생합니다.

새 `ClientKafka` 인스턴스가 시작되면 소비자 그룹에 가입하고 해당 토픽을 구독합니다. 이 프로세스는 소비자 그룹의 소비자에게 할당된 토픽 파티션의 재밸런싱을 트리거합니다.

일반적으로 주제 파티션은 애플리케이션 실행 시 무작위로 설정된 소비자 이름별로 정렬된 소비자 컬렉션에 주제 파티션을 할당하는 라운드 로빈 파티셔너를 사용하여 할당됩니다. 그러나 새 소비자가 소비자 그룹에 가입하면 새 소비자는 소비자 컬렉션 내의 어느 곳에나 배치될 수 있습니다. 따라서 기존 소비자가 새 소비자 다음에 배치될 때 기존 소비자에게 다른 파티션이 할당될 수 있는 조건이 만들어집니다. 결과적으로 다른 파티션이 할당된 소비자는 리

밸런싱 전에 전송된 요청에 대한 응답 메시지를 잃게 됩니다.

`ClientKafka` 소비자들이 응답 메시지를 잃지 않도록 하기 위해 Nest에 내장된 사용자 정의 파티셔너가 사용됩니다. 이 사용자 정의 파티셔너는 애플리케이션 시작 시 설정된 고해상도 타임스탬프(`process.hrtime()`)를 기준으로 정렬된 소비자 컬렉션에 파티션을 할당합니다.

메시지 응답 구독

> 경고 참고 이 섹션은 요청-응답 메시지 스타일을 사용하는 경우에만 해당됩니다(`메시지패턴` 데코레이터와 `ClientKafka#send` 메서드). 응답 구독하기

> 토픽은 이벤트 기반 통신에 필요하지 않습니다(`@EventPattern` 데코레이터 및 메소드를 전송합니다).

`ClientKafka` 클래스는 `subscribeToResponseOf()` 메서드를 제공합니다. `subscribeToResponseOf()` 메서드는 요청의 토픽 이름을 인수로 받아 파생된 응답 토픽 이름을 응답 토픽 컬렉션에 추가합니다. 이 메서드는 메시지 패턴을 구현할 때 필요합니다.

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

`ClientKafka` 인스턴스가 비동기적으로 생성된 경우, `connect()`) 메서드를 호출하기 전에 `subscribeToResponseOf()` 메서드를 호출해야 합니다.

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

수신

Nest는 들어오는 Kafka 메시지를 `키`, `값`, `헤더` 속성이 있는 객체로 수신하며, 그 값은 `Buffer` 유형입니다. 그런 다음 Nest는 버퍼를 문자열로 변환하여 이러한 값을 구문 분석합니다. 문자열이 "object like"인 경우 Nest는 문자열을 `JSON으로` 구문 분석하려고 시도합니다. 그런 다음 해당 값은 연결된 핸들러로 전달됩니다.

발신

Nest는 이벤트를 게시하거나 메시지를 보낼 때 직렬화 프로세스를 거친 후 발신 Kafka 메시지를 보냅니다. 이는 `ClientKafka emit()`) 및 `send()` 메서드에 전달된 인수 또는 `@MessagePattern` 메서드에서 반환된 값에 대해 발생합니다. 이 직렬화는 `JSON.stringify()` 또는 `toString()` 프로토타입 메서드를 사용하여 문자열이나 버퍼가 아닌 객체를 "문자열화"합니다.

```
@@파일명(heroes.controller) @Controller()
export class HeroesController {
  @MessagePattern('hero.kill.dragon')
  killDragon(@Payload() message: KillDragonMessage): any {
    const dragonId = message.dragonId;
    const items = [
      { id: 1, name: '신화검' },
      { id: 2, name: '던전 열쇠' },
    ];
```

```
      품목 반품;
    }
  }
```

> 정보 힌트 `@Payload()`는 `@nestjs/microservices`에서 가져옵니다.

`키` 및 `값` 속성이 있는 개체를 전달하여 발신 메시지에 키를 지정할 수도 있습니다. 메시지 키 지정은 공동 파티셔닝 요구 사항을 충족하는 데 중요합니다.

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

    const items = [
      { id: 1, name: '신화검' },
      { id: 2, name: '던전 열쇠' },
    ];

    반환 { headers:
      {
        영역
      },
      키: heroId,
      값: items
    }
  }
}
```

또한 이 형식으로 전달되는 메시지에는 `헤더` 해시 속성에 설정된 사용자 지정 헤더도 포함될 수 있습니다. 헤더 해시 속성 값은 `문자열` 또는 `버퍼` 유형 중 하나여야 합니다.

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

    const items = [
      { id: 1, name: '신화검' },
      { id: 2, name: '던전 열쇠' },
    ];
```

```
      반환 { headers:
        {
          kafka_nestRealm: 영역
        },
        키: heroId,
        값: items
      }
    }
  }
```

이벤트 기반

요청-응답 방식은 서비스 간에 메시지를 주고받는 데는 이상적이지만, 메시지 스타일이 이벤트 기반인 경우, 즉 응답을 기다리지 않고 이벤트를 게시하려는 경우에는 적합하지 않습니다(카프카에 이상적). 이 경우 두 개의 토픽을 유지 관리하기 위해 요청-응답에 필요한 오버헤드를 원하지 않을 수 있습니다.

이에 대해 자세히 알아보려면 다음 두 섹션을 확인하세요: 개요: 이벤트 기반 및 개요: 이벤트 게시하기.

컨텍스트

보다 정교한 시나리오에서는 들어오는 요청에 대한 더 많은 정보에 액세스하고 싶을 수 있습니다. Kafka 트랜스포터 사용 시, `KafkaContext` 객체에 액세스할 수 있습니다.

```
@@파일명() @메시지패턴('hero.kill.dragon')
killDragon(@Payload() 메시지: 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()}`);
}
```

> 정보 힌트 `@Payload()`, `@Ctx()`, KafkaContext는 `nestjs/microservices` 패키지.

원본 카프카 인커밍메시지 객체에 접근하기 위해서는 `KafkaContext` 객체를 다음과 같이 생성합니다:

```
@@파일명() @메시지패턴('hero.kill.dragon')
killDragon(@Payload() 메시지: KillDragonMessage, @Ctx() 컨텍스트:
```

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

여기서 `IncomingMessage는` 다음 인터페이스를 충족합니다:

```
인터페이스 IncomingMessage { 주제:
  문자열;
  파티션: 숫자; 타임스탬
  프: 문자열; 크기: 숫자
  ; 속성: 숫자; 오프셋:
  문자열; 키: any;
  값: any;
  헤더를 추가합니다: 레코드<스트링, 임의>;
}
```

핸들러에서 수신된 각 메시지의 처리 시간이 느린 경우 `하트비트` 콜백을 사용하는 것이 좋습니다. `하트비트` 함수를 검색하려면 다음과 같이 `KafkaContext의 getHeartbeat()` 메서드를 사용하세요:

```
@@파일명() @메시지패턴('hero.kill.dragon')
비동기 킬드래곤(@Payload() 메시지: KillDragonMessage, @Ctx() context:
KafkaContext) {
  const heartbeat = context.getHeartbeat();

  // 느린 처리 대기 doWorkPart1()을
  수행합니다;

  // 세션타임아웃을 초과하지 않도록 하트비트를 전송합니다;

  // 느린 처리를 다시 대기하면서
  doWorkPart2()를 기다립니다;
}
```

이름 지정 규칙

Kafka 마이크로서비스 구성 요소는 Nest 마이크로서비스 클라이언트와 서버 구성 요소 간의 충돌을 방지하기 위해 `client.clientId` 및 `consumer.groupId` 옵션에 각자의 역할에 대한 설명을 추가합니다. 기본적으로 `ClientKafka` 구성 요소는 이 두 옵션에 `-client`를 추가하고 `ServerKafka` 구성 요소는 `-server`를 추가합니다. 아래에 제공된 값이 이러한 방식으로 어떻게 변환되는지 참고하세요(주석에 표시된 대로).

```
@@파일명(메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  운송: Transport.KAFKA, 옵션: {
    클라이언트: {
      clientId: 'hero', // 히어로 서버 브
      로커: ['localhost:9092'],
    },
    소비자: {
      groupId: 'hero-consumer' // hero-consumer-server
    },
  }
});
```

그리고 클라이언트를 위해서도요:

```
@@파일명(heroes.controller)
@Client({
  운송: Transport.KAFKA, 옵션: {
    클라이언트: {
      clientId: 'hero', // 히어로-클라이언
      트 브로커: ['localhost:9092'],
    },
    소비자: {
      groupId: 'hero-consumer' // hero-consumer-client
    }
  }
})
클라이언트: ClientKafka;
```

> 정보 힌트 카프카 클라이언트 및 소비자 이름 지정 규칙을 확장하여 사용자 지정할 수 있습니다.
>
> `ClientKafka` 및 KafkaServer를 사용자 지정 공급자에 추가하고 생성자를 재정의하세요.

Kafka 마이크로서비스 메시지 패턴은 요청 채널과 응답 채널에 대해 두 개의 토픽을 사용하므로, 응답 패턴은 요청 토픽에서 파생되어야 합니다. 기본적으로 응답 토픽의 이름은 요청 토픽 이름에 `.reply가` 추가된 합성어입니다.

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

> 정보 힌트 카프카 응답 주제 명명 규칙은 사용자 지정 공급자에서 ClientKafka를 확장하고 `getResponsePatternName` 메서드를 재정의하여 사용자 지정할 수 있습니다.

검색 가능한 예외

다른 전송기와 마찬가지로, 처리되지 않은 모든 예외는 자동으로 `RpcException으로` 래핑되어 "사용자 친화적인" 형식으로 변환됩니다. 그러나 이 메커니즘을 우회하여 예외를 `kafkajs` 드라이버가 대신 사용하도록 하고 싶은 예외적인 경우가 있습니다. 메시지를 처리할 때 예외를 던지면 `kafkajs가` 메시지를 재시도(재전송)하도록 지시하므로 메시지(또는 이벤트) 핸들러가 트리거되었더라도 오프셋이 Kafka에 커밋되지 않습니다.

> 경고 경고 이벤트 핸들러(이벤트 기반 통신)의 경우 처리되지 않은 모든 예외는 기본적으로 검색 가능한 예외로 간주됩니다.

이를 위해 다음과 같이 `KafkaRetriableException이라는` 전용 클래스를 사용할 수 있습니다:

```
새로운 KafkaRetriableException('...')을 던집니다;
```

> 정보 힌트 `KafkaRetriableException` 클래스는 `@nestjs/microservices에서` 내보내집니다. 패키지입니다.

커밋 오프셋

카프카로 작업할 때는 오프셋 커밋이 필수입니다. 기본적으로 메시지는 특정 시간이 지나면 자동으로 커밋됩니다. 자세한 내용은 KafkaJS 문서를 참조하세요. `ClientKafka는` 기본 KafkaJS 구현처럼 작동하는 오프셋을 수동으로 커밋하는 방법을 제공합니다.

```
@@파일명() @EventPattern('user.created')

비동기 처리 사용자 생성(@Payload() 데이터: IncomingMessage, @Ctx() context:
KafkaContext) {
  // 비즈니스 로직

  const { 오프셋 } = 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(데이터, 컨텍스트) {
```

```
    // 비즈니스 로직

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

메시지의 자동 커밋을 비활성화하려면 다음과 같이 `실행` 구성에서 `autoCommit:` false를 설정합니다:

```
@@파일명(메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  운송: Transport.KAFKA, 옵션: {
    클라이언트: {
      브로커: ['localhost:9092'],
    },
    run: {
      자동 커밋: false
    }
  }
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.KAFKA,
  옵션: { client:
    {
      브로커: ['localhost:9092'],
    },
    run: {
      자동 커밋: false
    }
  }
});
```

gRPC

gRPC는 모든 환경에서 실행할 수 있는 최신 오픈 소스 고성능 RPC 프레임워크입니다. 로드 밸런싱, 추적, 상태 확인 및 인증에 대한 플러그형 지원을 통해 데이터센터 안팎의 서비스를 효율적으로 연결할 수 있습니다.

많은 RPC 시스템과 마찬가지로 gRPC는 원격으로 호출할 수 있는 함수(메서드)로 서비스를 정의하는 개념을 기반으로 합니다. 각 메서드에 대해 매개변수와 반환 유형을 정의합니다. 서비스, 매개변수 및 반환 유형은 Google의 오픈 소스 언어 중립 프로토콜 버퍼 메커니즘을 사용하여 `.proto` 파일에 정의됩니다.

Nest는 `.proto` 파일을 사용하여 클라이언트와 서버를 동적으로 바인딩함으로써 원격 프로시저 호출을 쉽게 구현하고 구조화된 데이터를 자동으로 직렬화 및 역직렬화할 수 있도록 합니다.

설치

gRPC 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

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

개요

다른 Nest 마이크로서비스 전송 계층 구현과 마찬가지로, `createMicroservice()` 메서드에 전달된 옵션 객체의 `전송` 속성을 사용하여 gRPC 전송기 메커니즘을 선택합니다. 다음 예제에서는 영웅 서비스를 설정하겠습니다. `옵션` 속성은 해당 서비스에 대한 메타데이터를 제공하며, 그 속성은 아래에 설명되어 있습니다.

```
@@파일명(메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.GRPC, 옵션:
  {
    패키지: '영웅',
    protoPath: join(__dirname, 'hero/hero.proto'),
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.GRPC,
  옵션을 추가합니다: {
    package: 'hero',
    protoPath: join(__dirname, 'hero/hero.proto'),
  },
});
```

> 정보 힌트 `join()` 함수는 `path` 패키지에서 가져오고, `Transport` 열거형은 `@nestjs/microservices` 패키지에서 가져옵니다.

`nest-cli.json` 파일에 TypeScript가 아닌 파일을 배포할 수 있는 `자산` 속성을 추가하고, 모든 TypeScript가 아닌 자산 감시를 켜기 위해 `watchAssets 속성을` 추가합니다. 이 경우 `.proto` 파일이 `dist` 폴더에 자동으로 복사되도록 합니다.

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

옵션

gRPC 트랜스포터 옵션 개체는 아래에 설명된 속성을 노출합니다.

`패키지`	Protobuf 패키지 이름(`.proto` 파일의 `패키지` 설정과 일치). 필수
`protoPath`	`.proto` 파일의 절대(또는 루트 디렉터리에 상대) 경로입니다. 필수
`url`	연결 URL. `IP 주소/dns 이름:포트` 형식의 문자열(예, `'localhost:50051'`) 트랜스포터가 연결을 설정하는 주소/포트를 정의합니다. 선택 사항입니다. 기본값은 `'localhost:5000'`입니다.
`protoLoader`	`.proto` 파일을 로드할 유틸리티의 NPM 패키지 이름입니다. 선택 사항입니다. 기본값은 `'@grpc/proto-loader'`
`로더`	`grpc/proto-loader` 옵션. 이를 통해 다음과 같은 동작을 세부적으로 제어할 수 있습니다. `.proto` 파일. 선택 사항입니다. 자세한 내용은 여기를 참조하세요.
`자격 증명`	서버 자격 증명. 선택 사항입니다. 여기에서 자세히 알아보기

샘플 gRPC 서비스

`HeroesService라는` 샘플 gRPC 서비스를 정의해 보겠습니다. 위의 `옵션` 오브젝트에서 `protoPath` 프로퍼티는 `.proto` 정의 파일 `hero.proto의` 경로를 설정합니다. `hero.proto` 파일은 프로토콜 버퍼를 사용하여

구조화되어 있습니다. 파일은 다음과 같습니다:

```
// hero/hero.proto 구문
= "proto3";

패키지 영웅;

서비스 히어로즈서비스 {
  rpc FindOne (HeroById) 반환 (Hero) {}
}
```

```
메시지 HeroById {
  int32 id = 1;
}

메시지 Hero { int32
  id = 1; 문자열
  name = 2;
}
```

히어로서비스는 FindOne() 메서드를 노출합니다. 이 메서드는 HeroById 유형의 입력 인수를 받고 Hero 메시지를 반환합니다(프로토콜 버퍼는 메시지 요소를 사용하여 매개변수 유형과 반환 유형을 모두 정의합니다).

다음으로 서비스를 구현해야 합니다. 이 정의를 충족하는 핸들러를 정의하기 위해 아래와 같이 컨트롤러에서 @GrpcMethod() 데코레이터를 사용합니다. 이 데코레이터는 메서드를 gRPC 서비스 메서드로 선언하는 데 필요한 메타데이터를 제공합니다.

> 정보 힌트 이전 마이크로서비스 챕터에서 소개한 @MessagePattern() 데코레이터(자세히 읽기)는 gRPC 기반 마이크로서비스에는 사용되지 않습니다. gRPC 기반 마이크로서비스에서는

```
@@파일명(heroes.controller) @Controller()
export class HeroesController {
  @GrpcMethod('HeroesService', 'FindOne')
  findOne(데이터: HeroById, 메타데이터: 메타데이터, 호출: ServerUnaryCall<any,
any>): Hero {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
}
@@스위치
@Controller()
export class HeroesController {
  @GrpcMethod('HeroesService', 'FindOne')
  findOne(data, metadata, call) {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
```

> 정보 힌트 @GrpcMethod() 데코레이터는 @nestjs/microservices 패키지에서, 메타데이터와 ServerUnaryCall은 grpc 패키지에서 가져옵니다.

위에 표시된 데코레이터는 두 개의 인수를 받습니다. 첫 번째 인수는 hero.proto의 히어로즈서비스 서비스 정의에 해당하는 서비스 이름(예: 'HeroesService')입니다. 두 번째 인자(문자열 'FindOne')는 hero.proto 파일의 히어로즈서비스 내에 정의된 FindOne() rpc 메서드에 해당합니다.

findOne() 핸들러 메서드에는 호출자가 전달한 데이터, gRPC 요청 메타데이터를 저장하는 메타데이터, 클라이언트에 메타데이터를 전송하기 위한 sendMetadata와 같은 GrpcCall 객체 속성을 가져오기 위한 호출 등 세 가지 인수가 필요합니다.

두 @GrpcMethod() 데코레이터 인수는 모두 선택 사항입니다. 두 번째 인수(예: 'FindOne') 없이 호출하면 Nest는 처리기 이름을 낙타 대문자로 변환하여 .proto 파일 rpc 메서드를 처리기와 자동으로 연결합니다(예: findOne 처리기는 FindOne rpc 호출 정의와 연결됨). 이는 아래와 같습니다.

```
@@파일명(heroes.controller) @Controller()
export class HeroesController {
  @GrpcMethod('HeroesService')
  findOne(데이터: HeroById, 메타데이터: 메타데이터, 호출: ServerUnaryCall<any,
any>): Hero {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
}
@@스위치
@Controller()
export class HeroesController {
  @GrpcMethod('HeroesService')
  findOne(data, metadata, call) {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
}
```

첫 번째 @GrpcMethod() 인수를 생략할 수도 있습니다. 이 경우 Nest는 처리기가 정의된 클래스 이름을 기준으로 처리기를 프로토 정의 파일의 서비스 정의와 자동으로 연결합니다. 예를 들어, 다음 코드에서 히어로즈서비스 클래스는 '히어로즈서비스'라는 이름의 일치 여부에 따라 핸들러 메서드를 hero.proto 파일의 히어로즈서비스

서비스 정의와 연결합니다.

```
@@파일명(heroes.controller) @Controller()
내보내기 클래스 HeroesService {
```

```
  @GrpcMethod()
  findOne(데이터: HeroById, 메타데이터: 메타데이터, 호출: ServerUnaryCall<any,
any>): Hero {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
}
@@스위치
@Controller()
내보내기 클래스 HeroesService {
  @GrpcMethod()
  findOne(data, metadata, call) {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
}
```

클라이언트

Nest 애플리케이션은 `.proto` 파일에 정의된 서비스를 소비하는 gRPC 클라이언트로 작동할 수 있습니다. `ClientGrpc` 객체를 통해 원격 서비스에 액세스합니다. 여러 가지 방법으로 `ClientGrpc` 객체를 얻을 수 있습니다.

선호하는 기법은 `ClientsModule을` 임포트하는 것입니다. `register()` 메서드를 사용하여 `.proto` 파일에 정의된 서비스 패키지를 인젝션 토큰에 바인딩하고 서비스를 구성할 수 있습니다. `이름` 속성은 인젝션 토큰입니다. gRPC 서비스의 경우 `전송을` 사용합니다: `Transport.GRPC를 사용합니다`. `옵션` 속성은 위에서 설명한 것과 동일한 속성을 가진 개체입니다.

```
임포트합니다: [
  ClientsModule.register([
    {
      이름: 'HERO_PACKAGE',
      transport: Transport.GRPC, 옵션: {
        패키지: '영웅',
        protoPath: join(__dirname, 'hero/hero.proto'),
      },
    },
  ]),
];
```

> 정보 힌트 `register()` 메서드는 객체 배열을 받습니다. 쉼표로 구분된 등록 객체 목록을 제공하여 여러 패키지를 등록할 수 있습니다.

등록이 완료되면 `@Inject()를` 사용하여 구성된 `ClientGrpc` 객체를 삽입할 수 있습니다. 그런 다음 `ClientGrpc` 객체의 `getService()` 메서드를 사용하여 아래와 같이 서비스 인스턴스를 검색할 수 있습니다.

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

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

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

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

> 오류 경고 gRPC 클라이언트는 프로토 로더 구성(마이크로서비스 트랜스포터 구성의 `options.loader.keepcase)`에서 `keepCase` 옵션이 `true로` 설정되어 있지 않으면 이름에 밑줄 `_이` 포함된 필드를 전송하지 않습니다.

다른 마이크로서비스 전송 메서드에 사용된 기술과 비교하면 약간의 차이가 있습니다. `ClientProxy` 클래스 대신 `getService()` 메서드를 제공하는 `ClientGrpc` 클래스를 사용합니다. `getService()` 일반 메서드는 서비스 이름을 인자로 받아 해당 인스턴스(사용 가능한 경우)를 반환합니다.

또는 다음과 같이 `@Client()` 데코레이터를 사용하여 `ClientGrpc` 객체를 인스턴스화할 수 있습니다:

```
@Injectable()
내보내기 클래스 AppService 구현 OnModuleInit {
  @Client({
    transport: Transport.GRPC, 옵션
    : {
      패키지: '영웅',
      protoPath: join(__dirname, 'hero/hero.proto'),
    },
  })
  클라이언트: ClientGrpc;

  private heroesService: HeroesService;

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

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

```
    }
  }
```

마지막으로, 더 복잡한 시나리오의 경우 동적으로 구성된 클라이언트를 삽입할 수 있습니다. `ClientProxyFactory` 클래스입니다.

두 경우 모두 `히어로즈서비스` 프록시 객체에 대한 참조가 생성되며, 이 객체는 `.proto` 파일에 정의된 것과 동일한 메서드 집합을 노출합니다. 이제 이 프록시 객체(즉, `영웅 서비스`)에 액세스하면 gRPC 시스템이 자동으로 요청을 직렬화하여 원격 시스템으로 전달하고 응답을 반환한 후 응답을 역직렬화합니다. gRPC는 이러한 네트워크 통신 세부 정보로부터 우리를 보호하기 때문에 `heroesService는` 로컬 공급자처럼 보이고 작동합니다.

모든 서비스 메서드는 언어의 자연스러운 규칙을 따르기 위해 대소문자를 구분합니다. 예를 들어, `.proto` 파일 `히어로즈서비스` 정의에는 `FindOne()` 함수가 포함되어 있지만, `히어로즈서비스` 인스턴스는 `findOne()` 메서드를 제공합니다.

```
인터페이스 히어로즈 서비스 {
  findOne(data: { id: number }): Observable<any>;
}
```

메시지 핸들러는 `Observable을` 반환할 수도 있으며, 이 경우 스트림이 완료될 때까지 결과값이 방출됩니다.

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

요청과 함께 gRPC 메타데이터를 보내려면 다음과 같이 두 번째 인수를 전달할 수 있습니다:

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

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

> 정보 힌트 `메타데이터` 클래스는 `grpc` 패키지에서 가져옵니다.

이를 위해서는 앞서 몇 단계에 걸쳐 정의한 `히어로즈서비스` 인터페이스를 업데이트해야 한다는 점에 유의하세요.

예

작동 예제는 여기에서 확인할 수 있습

니다. gRPC 스트리밍

gRPC 자체는 일반적으로 `스트림으로` 알려진 장기 라이브 연결을 지원합니다. 스트림은 다음과 같습니다.
채팅, 관찰 또는 청크 데이터 전송과 같은 경우에 유용합니다. 자세한 내용은 여기에서 공식 문서를 참조하세요.

Nest는 두 가지 방법으로 GRPC 스트림 핸들러를 지원합니다:

- RxJS `Subject` + `Observable` 핸들러: 컨트롤러 메서드 내부에서 바로 응답을 작성하거나 `Subject/Observable` 소비자에게 전달할 때 유용합니다.
- 순수 GRPC 호출 스트림 핸들러: 노드 표준 `듀플렉스` 스트림 핸들러의 나머지 디스패치를 처리할 일부 실행자에게 전달하는 데 유용할 수 있습니다.

스트리밍 샘플

`HelloService라는` 새로운 샘플 gRPC 서비스를 정의해 보겠습니다. `hello.proto` 파일은 프로토콜 버퍼를 사용하여 구조화되어 있습니다. 파일은 다음과 같습니다:

```
// hello/hello.proto 구
문 = "proto3";

패키지 안녕하세요;

서비스 HelloService {
  rpc BidiHello(스트림 HelloRequest) 반환 (스트림 HelloResponse); rpc
  LotsOfGreetings(스트림 HelloRequest) 반환 (HelloResponse);
}

메시지 HelloRequest {
  문자열 greeting = 1;
}

메시지 HelloResponse {
  문자열 reply = 1;
}
```

> 정보 힌트: `LotsOfGreetings` 메서드는 `@GrpcMethod`를 사용하여 간단하게 구현할 수 있습니다. 데코레이터를 사용해야 합니다(위의 예제에서처럼). 반환된 스트림은 여러 값을 방출할 수 있기 때문입니다.

이 `.proto` 파일을 기반으로 `HelloService` 인터페이스를 정의해 보겠습니다:

```
인터페이스 HelloService {
  bidiHello(업스트림: Observable<HelloRequest>):
Observable<HelloResponse>;
  lotsOfGreetings(
    업스트림: 관찰 가능<HelloRequest>,
  ): 관찰 가능 <헬로 응답>;
}

인터페이스 HelloRequest { 인사말: 문
  자열;
}

인터페이스 HelloResponse { 회신:
  문자열;
}
```

> 정보 힌트 프로토 인터페이스는 ts-proto 패키지에 의해 자동으로 생성될 수 있으며, 여기에서 자세히 알아보세요.

주제 전략

`GrpcStreamMethod()` 데코레이터는 함수 파라미터를 RxJS `옵저버블로` 제공합니다. 따라서 여러 메시지를 수신하고 처리할 수 있습니다.

```
@GrpcStreamMethod()
비디헬로(메시지: Observable<any>, 메타데이터: 메타데이터, 호출:
ServerDuplexStream<any, any>): Observable<any> {
  const subject = new Subject();

  const onNext = message => {
    console.log(message);
    subject.next({
      답장합니다: '안녕, 세상아!'
    });
  };
  const onComplet = () => subject.complete();
  messages.subscribe({
    다음: on다음, 완료: on완

    료,
  });



  subject.asObservable()을 반환합니다;
}
```

> 경고 경고 `@GrpcStreamMethod()` 데코레이터와 전이중 상호 작용을 지원하려면 컨트롤러 메서드가 RxJS `Observable`을 반환해야 합니다.

> 정보 힌트 메타데이터 및 서버유나리콜 클래스/인터페이스는 grpc에서 가져옵니다.

패키지입니다.

서비스 정의(.proto 파일)에 따르면, BidiHello 메서드는 요청을 서비스로 스트리밍해야 합니다. 클라이언트에서 스트림으로 여러 개의 비동기 메시지를 전송하기 위해 RxJS ReplaySubject 클래스를 활용합니다.

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

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

헬로서비스.비디헬로(헬로리퀘스트$)를 반환합니다;
```

위의 예제에서는 스트림에 두 개의 메시지를 쓰고(next() 호출) 데이터 전송이 완료되었음을 서비스에 알렸습니다(complete() 호출).

통화 스트림 핸들러

메서드 반환값이 스트림으로 정의된 경우 @GrpcStreamCall() 데코레이터는 함수 매개변수를 다음과 같은 표준 메서드를 지원하는 grpc.ServerDuplexStream으로 제공합니다.

.on('data', callback), .write(message) 또는 .cancel(). 사용 가능한 메서드에 대한 전체 문서는 여기에서 확인할 수 있습니다.

또는 메서드 반환값이 스트림이 아닌 경우 @GrpcStreamCall() 데코레이터는 두 개의 함수 매개변수 grpc.ServerReadableStream(자세한 내용은 여기를 참조하세요)과 콜백을 각각 제공합니다.

먼저 전이중 상호 작용을 지원해야 하는 BidiHello를 구현해 보겠습니다.

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

> 정보 힌트 이 데코레이터는 특정 반환 매개변수를 제공할 필요가 없습니다. 스트림은 다른 표준 스트림 유형과 유사하게 처리될 것으로 예상됩니다.

위의 예제에서는 `write()` 메서드를 사용하여 응답 스트림에 객체를 썼습니다. `.on()` 메서드에 두 번째 매개변수로 전달된 콜백은 서비스가 새 데이터 청크를 수신할 때마다 호출됩니다.

`LotsOfGreetings` 메서드를 구현해 보겠습니다.

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

여기서는 `콜백` 함수를 사용하여 `요청 스트림` 처리가 완료되면 응답을 보냈습니다.

gRPC 메타데이터

메타데이터는 키-값 쌍의 목록 형태로 된 특정 RPC 호출에 대한 정보로, 키는 문자열이고 값은 일반적으로 문자열이지만 바이너리 데이터일 수도 있습니다. 메타데이터는 클라이언트가 서버에 호출과 관련된 정보를 제공할 수 있고 그 반대의 경우도 마찬가지입니다. 메타데이터에는 인증 토큰, 모니터링 목적의 요청 식별자 및 태그, 데이터 세트의 레코드 수와 같은 데이터 정보가 포함될 수 있습니다.

`GrpcMethod()` 핸들러에서 메타데이터를 읽으려면 두 번째 인수(메타데이터)를 사용하세요.

`메타데이터`(`grpc` 패키지에서 가져온 것).

핸들러에서 메타데이터를 다시 보내려면 `ServerUnaryCall#sendMetadata()` 메서드(세 번째 핸들러 인수)를 사용하세요.

```
@@파일명(heroes.controller) @Controller()
내보내기 클래스 HeroesService {
  @GrpcMethod()
  findOne(데이터: HeroById, 메타데이터: 메타데이터, 호출: 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);

    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
}
@@스위치
@Controller()
```

```
내보내기 클래스 HeroesService {
  @GrpcMethod()
  findOne(데이터, 메타데이터, 호출) {
    const serverMetadata = new Metadata();
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];

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

    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
}
```

마찬가지로 `@GrpcStreamMethod()` 핸들러(주제 전략)로 주석이 달린 핸들러에서 메타데이터를 읽으려면 두 번째 인수(메타데이터)를 사용하는데, 이 인수는 `@grpc` 패키지에서 가져온) `메타데이터` 유형입니다.

핸들러에서 메타데이터를 다시 보내려면 `ServerDuplexStream#sendMetadata()` 메서드(세 번째 핸들러 인수)를 사용하세요.

호출 스트림 핸들러 내에서 메타데이터를 읽으려면(`@GrpcStreamCall()`로 주석 처리된 핸들러) 다음과 같이 하세요. 데코레이터)를 사용하여 다음과 같이 `요청 스트림` 참조의 `메타데이터` 이벤트를 수신합니다:

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

맞춤형 운송업체

Nest는 개발자가 새로운 맞춤형 전송 전략을 구축할 수 있는 API뿐만 아니라 다양한 전송기를 기본으로 제공합니다. 전송기를 사용하면 플러그 가능한 통신 계층과 매우 간단한 애플리케이션 수준 메시지 프로토콜을 사용하여 네트워크를 통해 구성 요소를 연결할 수 있습니다(전체 기사 읽기).

> 정보 힌트 Nest로 마이크로서비스를 구축한다고 해서 반드시 `@nestjs/microservices` 패키지를 사용해야 하는 것은 아닙니다. 예를 들어 외부 서비스(다른 언어로 작성된 다른 마이크로서비스)와 통신하려는 경우 `@nestjs/microservice` 라이브러리에서 제공하는 모든 기능이 필요하지 않을 수 있습니다. 실제로 구독자를 선언적으로 정의할 수 있는 데코레이터(`@EventPattern` 또는 `@MessagePattern`)가 필요하지 않은 경우에는 독립형 애플리케이션을 실행하고 채널에 대한 연결/구독을 수동으로 유지하는 것으로 대부분의 사용 사례에 충분하며 더 많은 유연성을 제공할 수 있습니다.

사용자 지정 전송기를 사용하면 모든 메시징 시스템/프로토콜(Google Cloud Pub/Sub, Amazon Kinesis 등)을 통합하거나 기존 전송기를 확장하여 추가 기능(예: MQTT용 QoS)을 추가할 수 있습니다.

> 정보 힌트 Nest 마이크로서비스의 작동 방식과 기존 전송기의 기능을 확장하는 방법을 더 잘 이해하려면 NestJS 마이크로서비스 실제 사용 및 고급 NestJS 마이크로서비스 문서 시리즈를 읽어보시기 바랍니다.

전략 만들기

먼저 커스텀 트랜스포터에 해당하는 클래스를 정의해 보겠습니다.

```
'@nestjs/microservices'에서 { CustomTransportStrategy, Server } 임포트; 클

래스 GoogleCloudPubSubServer

  서버 확장

  CustomTransportStrategy를 구현합니다 {
  /**
   * 이 메서드는 "app.listen()"을 실행할 때 트리거됩니다.
   */
  listen(callback: () => void) {
    callback();
  }

  /**
   * 이 메서드는 애플리케이션 종료 시 트리거됩니다.
   */
  close() {}
}
```

> 경고 경고 이 장에서는 모든 기능을 갖춘 Google Cloud Pub/Sub 서버를 구현하지 않으므로 트랜스포터에 대한 구체적인 기술 세부 사항을 살펴봐야 합니다.

위의 예제에서는 `GoogleCloudPubSubServer` 클래스를 선언하고 `CustomTransportStrategy` 인터페이스에 의해 시행되는 `listen()` 및 `close()` 메서드를 제공했습니다. 또한 이 클래스는 `@nestjs/microservices` 패키지에서 가져온 `Server` 클래스를 확장하여 Nest 런타임에서 메시지 핸들러를 등록하는 데 사용되는 메서드와 같은 몇 가지 유용한 메서드를 제공합니다. 또는 기존 전송 전략의 기능을 확장하려는 경우 해당 서버 클래스(예: `ServerRedis`)를 확장할 수 있습니다. 일반적으로 메시지/이벤트 구독(및 필요한 경우 응답)을 담당하기 때문에 클래스에 `"서버"` 접미사를 추가했습니다.

이제 기본 제공 트랜스포터 대신 다음과 같이 사용자 지정 전략을 사용할 수 있습니다:

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>(
  AppModule,
  {
    전략: 새로운 구글클라우드펍서브서버(),
  },
);
```

기본적으로 `전송` 및 `옵션` 속성이 있는 일반 트랜스포터 옵션 객체를 전달하는 대신, 사용자 지정 트랜스포터 클래스의 인스턴스 값인 `전략이라는` 단일 속성을 전달합니다.

`GoogleCloudPubSubServer` 클래스로 돌아가서, 실제 애플리케이션에서는 메시지 브로커/외부 서비스에 대한 연결을 설정하고 `listen()` 메서드에서 구독자를 등록하고 특정 채널을 수신한 다음 `close()` 해체 메서드에서 구독을 제거하고 연결을 닫을 수 있지만, Nest 마이크로서비스가 서로 통신하는 방식을 잘 이해해야 하므로 이 문서 시리즈를 읽어보시기를 권장합니다. 대신 이 장에서는 `Server` 클래스가 제공하는 기능과 이를 활용하여 사용자 지정 전략을 구축하는 방법에 중점을 두겠습니다.

예를 들어 애플리케이션 어딘가에 다음과 같은 메시지 핸들러가 정의되어 있다고 가정해 보겠습니다:

```
메시지 패턴('에코') echo(@Payload() 데
이터: 객체) {
  데이터를 반환합니다;
}
```

이 메시지 핸들러는 Nest 런타임에 의해 자동으로 등록됩니다. `Server` 클래스를 사용하면 어떤 메시지 패턴이 등록되었는지 확인하고 해당 패턴에 할당된 실제 메서드에 액세스하여 실행할 수 있습니다. 이를 테스트하기 위해 `콜백` 함수가 호출되기 전에 `listen()` 메서드 안에 간단한 `console.log를` 추가해 보겠습니다:

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

애플리케이션이 다시 시작되면 터미널에 다음 로그가 표시됩니다:

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

> 정보 힌트 `@EventPattern` 데코레이터를 사용하면 동일한 출력을 볼 수 있지만 `isEventHandler` 속성을 `true`로 설정합니다.

보시다시피, `메시지 핸들러` 속성은 패턴이 키로 사용되는 모든 메시지(및 이벤트) 핸들러의 `맵` 컬렉션입니다. 이제 키(예: `"echo"`)를 사용하여 메시지 핸들러에 대한 참조를 받을 수 있습니다:

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

임의의 문자열(여기서는 `"Hello world!"`)을 인자로 전달하는 `echoHandler를` 실행하면 콘솔에서 해당 문자열을 볼 수 있습니다:

```
안녕하세요!
```

즉, 메서드 핸들러가 제대로 실행되었다는 뜻입니다.

인터셉터와 함께 `커스텀 트랜스포트 전략을` 사용할 때 핸들러는 RxJS 스트림으로 래핑됩니다. 즉, 스트림의 기본 로직을 실행하려면 스트림을 구독해야 합니다(예: 인터셉터가 실행된 후 컨트롤러 로직으로 계속 진행).

이에 대한 예는 아래에서 확인할 수 있습니다:

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

클라이언트 프록시

첫 번째 섹션에서 언급했듯이, 마이크로서비스를 생성할 때 반드시 `@nestjs/microservices` 패키지를 사용할 필요는 없지만, 그렇게 하기로 결정하고 사용자 정의 전략을 통합해야 하는 경우 "클라이언트" 클래스도 제공해야 합니다.

> 정보 힌트 다시 한 번 말씀드리지만, 모든 `@nestjs/마이크로서비스` 기능(예: 스트리밍)과 호환되는 완전한 기능을 갖춘 클라이언트 클래스를 구현하려면 프레임워크에서 사용하는 통신 기술에 대한 충분한 이해가 필요합니다. 자세한 내용은 이 문서를 참조하세요.

외부 서비스와 통신하거나 메시지(또는 이벤트)를 전송 및 게시하려면 다음과 같이 라이브러리별 SDK 패키지를 사용하거나 `ClientProxy`를 확장하는 사용자 지정 클라이언트 클래스를 구현할 수 있습니다:

```
'@nestjs/microservices'에서 { ClientProxy, ReadPacket, WritePacket }을 가져옵니다;

GoogleCloudPubSubClient 클래스 ClientProxy 확장 {
  async connect(): Promise<any> {}
  async close() {}
  async dispatchEvent(packet: ReadPacket<any>): Promise<any> {}
  publish(
    패킷을 읽습니다: ReadPacket<any>,
    콜백: (패킷: WritePacket<any>) => void,
  ): 함수 {}
}
```

> 경고 경고 이 장에서는 모든 기능을 갖춘 Google Cloud Pub/Sub 클라이언트를 구현하지 않으므로 트랜스포터에 대한 구체적인 기술 세부 사항을 살펴봐야 합니다.

보시다시피 `ClientProxy` 클래스는 연결 설정 및 종료, 메시지(`publish`) 및 이벤트(`dispatchEvent`) 게시를 위한 몇 가지 메서드를 제공해야 합니다. 요청-응답 통신 스타일 지원이 필요하지 않은 경우 `publish()` 메서드를 비워두면 됩니다. 마찬가지로 이벤트 기반 통신을 지원할 필요가 없는 경우 `dispatchEvent()` 메서드를 건너뛰면 됩니다.

이러한 메서드가 언제, 무엇을 실행하는지 관찰하기 위해 다음과 같이 여러 개의 `console.log` 호출을 추가해 보겠습니다:

```
GoogleCloudPubSubClient 클래스 ClientProxy 확장 {
  async connect(): Promise<any> {
    콘솔 로그('연결');
  }

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

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

  게시(
    패킷을 읽습니다: ReadPacket<any>,
    콜백: (패킷: WritePacket<any>) => void,
  ): 함수 { console.log('message:',
    packet);

    // 실제 애플리케이션에서 "콜백" 함수는 다음과 같아야 합니다.
```

```
실행됨
    // 응답자로부터 페이로드를 전송합니다. 여기서는 간단히 시뮬레이션(5초 지연)해 보겠습
니다.
    // 원래 전달한 것과 동일한 "데이터"를 전달하여 응답을 보냈습니다.
    setTimeout(() => callback({ response: packet.data }), 5000);

    반환 () => 콘솔.로그('teardown');
  }
}
```

이제 `GoogleCloudPubSubClient` 클래스의 인스턴스를 생성하고 `send()` 함수를 실행해 보겠습니다. 메서드(이전 장에서 보셨을 것입니다)를 사용하여 반환된 관찰 가능한 스트림을 구독합니다.

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

이제 터미널에 다음과 같은 출력이 표시됩니다:

```
연결
메시지: { pattern: 'pattern', data: 'Hello world!' }
Hello world! // <-- 5초 후
```

`게시()` 메서드가 반환하는 "teardown" 메서드가 제대로 실행되는지 테스트하기 위해 스트림에 타임아웃 연산자를 적용하고 2초로 설정하여 `setTimeout이 콜백` 함수를 호출하는 것보다 일찍 종료되도록 해 보겠습니다.

```
const googlePubSubClient = new GoogleCloudPubSubClient();
googlePubSubClient
  .send('pattern', 'Hello world!')
  .pipe(timeout(2000))
  .subscribe(
    (응답) => 콘솔.로그(응답), (오류) => 콘솔.오류(오
    류.메시지),
  );
```

> 정보 힌트 `타임아웃` 연산자는 `rxjs/operators` 패키지에서 가져옵니다.

`타임아웃` 연산자를 적용하면 터미널 출력은 다음과 같이 표시됩니다:

```
연결
메시지: { pattern: 'pattern', data: 'Hello world!' }
```

```
해체 // <-- 해체 시간 초과가 발생
했습니다.
```

메시지를 보내는 대신 이벤트를 디스패치하려면 `emit()` 메서드를 사용합니다:

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

이것이 바로 콘솔에 표시되는 내용입니다:

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

메시지 직렬화

클라이언트 측에서 응답 직렬화에 대한 사용자 정의 로직을 추가해야 하는 경우 `ClientProxy` 클래스 또는 그 하위 클래스 중 하나를 확장하는 사용자 정의 클래스를 사용할 수 있습니다. 성공적인 요청을 수정하려면 `serializeResponse` 메서드를 재정의하고, 이 클라이언트를 통과하는 모든 오류를 수정하려면 `serializeError` 메서드를 재정의할 수 있습니다. 이 커스텀 클래스를 사용하려면 `customClass` 속성을 사용하여 클래스 자체를 `ClientsModule.register()` 메서드에 전달하면 됩니다. 아래는 각 오류를 `RpcException`으로 직렬화하는 커스텀 `ClientProxy`의 예시입니다.

```
@@파일명(오류처리.프록시)
'@nestjs/microservices'에서 { ClientTcp, RpcException }을 가져옵니다;

class ErrorHandlingProxy extends ClientTCP {
  serializeError(err: 오류) {
    새로운 RpcException(err)을 반환합니다;
  }
}
```

를 생성한 다음 `ClientsModule`에서 다음과 같이 사용합니다:

```
@@파일명(app.module)
@Module({
  임포트합니다: [
    ClientsModule.register({
      이름: 'CustomProxy',
      customClass: 오류처리 프록시,
    }),
  ]
})
AppModule 클래스 내보내기
```

> 정보 힌트 이것은 클래스의 인스턴스가 아니라 `customClass에` 전달되는 클래스 자체입니다. Nest는 사용자를 위해 내부적으로 인스턴스를 생성하고 `옵션` 프로퍼티에 지정된 모든 옵션을 새 `ClientProxy에` 전달합니다.

예외 필터

HTTP 예외 필터 계층과 해당 마이크로서비스 계층의 유일한 차이점은 `HttpException을` 던지는 대신 `RpcException을` 사용해야 한다는 점입니다.

```
새로운 RpcException('잘못된 자격 증명.')을 던집니다;
```

> 정보 힌트 `RpcException` 클래스는 `@nestjs/microservices` 패키지에서 가져옵니다.

위의 샘플을 사용하면 Nest가 던져진 예외를 처리하고 다음과 같은 구조의 `오류` 객체를 반환합니다:

```
{
  "상태": "오류",
  "메시지": "잘못된 자격 증명입니다."
}
```

필터

마이크로서비스 예외 필터는 HTTP 예외 필터와 유사하게 작동하지만 한 가지 작은 차이가 있습니다. 마이크로서비스 예외 필터의 `catch()` 메서드는 `옵저버블을` 반환해야 합니다.

```
@@파일명(rpc-exception.filter)
'@nestjs/common'에서 { Catch, RpcExceptionFilter, ArgumentsHost }를 임포트하고,
'rxjs'에서 { Observable, throwError }를 임포트합니다;
'@nestjs/microservices'에서 { RpcException }을 가져옵니다;

@Catch(RpcException)
export class ExceptionFilter 구현 RpcExceptionFilter<RpcException> {
  catch(exception: RpcException, host: ArgumentsHost): Observable<any> {
    반환 throwError(() => exception.getError());
  }
}
@@switch
'@nestjs/common'에서 { Catch }를 임포트하
고, 'rxjs'에서 { throwError }를 임포트합니
다;

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

> 경고 하이브리드 애플리케이션을 사용할 때 글로벌 마이크로서비스 예외 필터는 기본적으로 사용하도록 설정되어 있지 않습니다.

다음 예제는 수동으로 인스턴스화된 메서드 범위 필터를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 컨트롤러 범위 필터를 사용할 수도 있습니다(즉, 컨트롤러 클래스 앞에 `@UseFilters()` 데코레이터를 붙이면 됩니다).

```
@@파일명()
사용필터(새로운 예외필터()) 메시지 패턴({ cmd:
'합계' }) 누적(데이터: 숫자[]): 숫자 {
  반환 (데이터 || []).reduce((a, b) => a + b);
}
@@switch
사용필터(새로운 예외필터()) 메시지패턴({ cmd:
'합계' }) 누적(데이터) {
  반환 (데이터 || []).reduce((a, b) => a + b);
}
```

상속

일반적으로 애플리케이션 요구 사항을 충족하도록 완전히 사용자 정의된 예외 필터를 만듭니다. 그러나 핵심 예외 필터를 단순히 확장하고 특정 요인에 따라 동작을 재정의하려는 사용 사례가 있을 수 있습니다.

예외 처리를 기본 필터에 위임하려면 `BaseExceptionFilter`를 확장해야 합니다.
를 생성하고 상속된 `catch()` 메서드를 호출합니다.

```
@@파일명()
'@nestjs/common'에서 { Catch, ArgumentsHost }를 가져옵니다;
'@nestjs/microservices'에서 { BaseRpcExceptionFilter }를 가져옵니다;

@Catch()
export class AllExceptionsFilter extends BaseRpcExceptionFilter {
  catch(exception: any, host: ArgumentsHost) {
    반환 super.catch(예외, 호스트);
  }
}
@@switch
'@nestjs/common'에서 { Catch }를 가져옵니다;
'@nestjs/microservices'에서 { BaseRpcExceptionFilter }를 가져옵니다;

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

위의 구현은 접근 방식을 보여주는 셸일 뿐입니다. 확장 예외 필터의 구현에는 맞춤형 비즈니스 로직(예: 다양한 조건 처리)이 포함될 수 있습니다.

파이프

일반 파이프와 마이크로서비스 파이프 사이에는 근본적인 차이가 없습니다. 유일한 차이점은 `HttpException`을 던지는 대신 `RpcException`을 사용해야 한다는 것입니다.

> 정보 힌트 `RpcException` 클래스는 `@nestjs/microservices` 패키지에서 노출됩니다.

바인딩 파이프

다음 예제는 수동으로 인스턴스화된 메서드 범위 파이프를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 컨트롤러 범위 파이프를 사용할 수도 있습니다(즉, 컨트롤러 클래스의 접두사에 `@UsePipes()` 데코레이터).

```
@@파일명()
@UsePipes(new ValidationPipe()) 메시지 패턴({ cmd: '합계' }) 누적(데이터: 숫자[]): 숫자 {
  반환 (데이터 || []).reduce((a, b) => a + b);
}
@@switch
@UsePipes(new ValidationPipe())
메시지패턴({ cmd: '합계' }) 누적(데이터) {
  반환 (데이터 || []).reduce((a, b) => a + b);
}
```

경비병

마이크로서비스 가드와 일반 HTTP 애플리케이션 가드 사이에는 근본적인 차이가 없습니다. 유일한 차이점은 `HttpException을` 던지는 대신 `RpcException을` 사용해야 한다는 점입니다.

> 정보 힌트 `RpcException` 클래스는 `@nestjs/microservices` 패키지에서 노출됩니다.

바인딩 가드

다음 예제는 메서드 범위 가드를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 컨트롤러 범위 가드를 사용할 수도 있습니다(즉, 컨트롤러 클래스에 `@UseGuards()` 데코레이터를 앞에 붙이면 됩니다).

```
@@파일명() @사용가드(AuthGuard) @메
시지패턴({ cmd: '합계' })
accumulate(data: number[]): number {
  반환 (데이터 || []).reduce((a, b) => a + b);
}
@@스위치 @사용가드
(AuthGuard)
메시지 패턴({ cmd: '합계' }) 누적(데이터) {
  반환 (데이터 || []).reduce((a, b) => a + b);
}
```

인터셉터

일반 인터셉터와 마이크로서비스 인터셉터 간에는 차이가 없습니다. 다음 예제에서는 수동으로 인스턴스화된 메서드 범위 인터셉터를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 컨트롤러 범위 인터셉터도 사용할 수 있습니다(즉, 컨트롤러 클래스 앞에 `@UseInterceptors()`

데코레이터).

```
@@파일명()
사용 인터셉터(새로운 트랜스폼인터셉터())
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
  반환 (데이터 || []).reduce((a, b) => a + b);
}
@@switch
사용 인터셉터(새로운 트랜스폼인터셉터()) 메시지패턴({
cmd: '합계' }) 누적(데이터) {
  반환 (데이터 || []).reduce((a, b) => a + b);
}
```