TypeScript 및 GraphQL의 강력한 기능 활용하기

GraphQL은 API를 위한 강력한 쿼리 언어이자 기존 데이터로 이러한 쿼리를 수행하기 위한 런타임입니다. 이는 REST API에서 일반적으로 발견되는 많은 문제를 해결하는 우아한 접근 방식입니다. 배경 지식이 필요하다면 GraphQL과 REST를 비교한 이 글을 읽어보시기 바랍니다. GraphQL과 TypeScript를 결합하면 GraphQL 쿼리의 유형 안전성을 개선하여 엔드투엔드 타이핑을 개발할 수 있습니다.

이 장에서는 GraphQL에 대한 기본적인 이해를 가정하고, 기본 제공 `@nestjs/graphql` 모듈로 작업하는 방법을 중점적으로 설명합니다. `GraphQL모듈은` Apollo 서버(`@nestjs/apollo` 드라이버 사용)와 Mercurius(`@nestjs/mercurius 사용)를` 사용하도록 구성할 수 있습니다. 이러한 검증된 GraphQL 패키지에 대한 공식 통합을 제공하여 Nest에서 GraphQL을 사용하는 간단한 방법을 제공합니다(더 많은 통합은 여기에서 참조하세요).

전용 드라이버를 직접 빌드할 수도 있습니다(자세한 내용은 여기를 참조하세요). 설

치

필요한 패키지를 설치하는 것으로 시작하세요:

```
# Express 및 Apollo의 경우(기본값)
$ npm i @nestjs/graphql @nestjs/apollo @apollo/server graphql

# Fastify 및 Apollo의 경우
# npm i @nestjs/graphql @nestjs/apollo @apollo/server @as-
integrations/fastify graphql

# Fastify 및 Mercurius용
# npm i @nestjs/graphql @nestjs/mercurius graphql mercurius
```

> 경고 경고 `@nestjs/graphql@>=9` 및 `@nestjs/apollo^10` 패키지는 Apollo v3와 호환되지만(자세한 내용은 Apollo Server 3 마이그레이션 가이드 참조), `@nestjs/graphql@^8은` Apollo v2(예: `apollo-server-express@2.x.x` 패키지)만 지원합니다.

개요

Nest는 코드 우선 방식과 스키마 우선 방식의 두 가지 GraphQL 애플리케이션 구축 방법을 제공합니다. 자신에게 가장 적합한 방법을 선택해야 합니다. 이 GraphQL 섹션의 대부분의 장은 코드 우선 방식을 채택하는 경우 따라야 할 부분과 스키마 우선 방식을 채택하는 경우 사용할 부분으로 나뉩니다.

코드 우선 접근 방식에서는 데코레이터와 TypeScript 클래스를 사용하여 해당 GraphQL 스키마를 생성합니다. 이 접근 방식은 TypeScript로만 작업하고 언어 구문 간의 컨텍스트 전환을 피하려는 경우에 유용합니다.

스키마 우선 접근 방식에서 진실의 소스는 GraphQL SDL(스키마 정의 언어) 파일입니다. SDL은 서로 다른 플랫폼 간에 스키마 파일을 공유할 수 있는 언어에 구애받지 않는 방법입니다. Nest는 다음을 자동으로 생성합니다.

클래스 또는 인터페이스를 사용하는 타입스크립트 정의는 GraphQL 스키마를 기반으로 하여 중복 상용구 코드를 작성할 필요성을 줄여줍니다.

GraphQL 및 TypeScript 시작하기

> 정보 힌트 다음 챕터에서는 `@nestjs/apollo` 패키지를 통합할 것입니다. 대신 `머큐리우스` 패키지를 사용하려면 이 섹션으로 이동하세요.

패키지가 설치되면 `GraphQLModule을` 임포트하고 `forRoot()`를 사용하여 구성할 수 있습니다.
정적 메서드입니다.

```
@@파일명()
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;
'@nestjs/apollo'에서 { ApolloDriver, ApolloDriverConfig }를 임포트합니다;

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
    }),
  ],
})
내보내기 클래스 AppModule {}
```

> 정보 힌트 머큐리우스 통합을 위해서는 `머큐리우스` 드라이버와 `MercuriusDriverConfig를` 대신 사용하세요. 둘 다 `@nestjs/mercurius` 패키지에서 내보냅니다.

`forRoot()` 메서드는 옵션 객체를 인자로 받습니다. 이러한 옵션은 기본 드라이버 인스턴스로 전달됩니다(사용 가능한 설정에 대한 자세한 내용은 Apollo 및 Mercurius를 참조하세요). 예를 들어 `플레이그라운드를` 비활성화하고 `디버그` 모드(Apollo의 경우)를 끄려면 다음 옵션을 전달합니다:

```
@@파일명()
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;
'@nestjs/apollo'에서 { ApolloDriver, ApolloDriverConfig }를 임포트합니다;

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      놀이터: 거짓,
    }),
  ],
})
내보내기 클래스 AppModule {}
```

이 경우 이러한 옵션은 `ApolloServer` 생성자에게 전달됩니다.

GraphQL 놀이터

플레이그라운드는 그래픽 인터랙티브 브라우저 내 GraphQL IDE로, 기본적으로 GraphQL 서버 자체와 동일한 URL에서 사용할 수 있습니다. 플레이그라운드에 액세스하려면 기본 GraphQL 서버가 구성 및 실행 중이어야 합니다. 지금 바로 확인하려면 여기에서 작동하는 예제를 설치 및 빌드할 수 있습니다. 또는 이 코드 샘플을 따라 하는 경우 리졸버 챕터의 단계를 완료한 후 플레이그라운드에 액세스할 수 있습니다.

애플리케이션이 백그라운드에서 실행 중인 상태에서 웹 브라우저를 열고 http://localhost:3000/graphql(호스트와 포트는 구성에 따라 다를 수 있음)로 이동하면 됩니다. 그러면 아래와 같이 GraphQL 플레이그라운드가 표시됩니다.

> 경고 참고 `@nestjs/mercurius` 통합은 기본 제공 GraphQL Playground 통합과 함께 제공되지 않습니다. 대신 GraphiQL을 사용할 수 있습니다(`graphiql: true` 설정).

여러 엔드포인트

`nestjs/graphql` 모듈의 또 다른 유용한 기능은 한 번에 여러 엔드포인트를 제공할 수 있다는 점입니다. 이를 통해 어떤 모듈을 어떤 엔드포인트에 포함할지 결정할 수 있습니다. 기본적으로 `GraphQL은` 전체 앱에서 리졸버를 검색합니다. 이 검색을 모듈의 하위 집합으로만 제한하려면 `include` 속성을 사용하세요.

```
GraphQLModule.forRoot({ 포
  함: [CatsModule],
}),
```

> 경고 경고 단일 애플리케이션에서 여러 GraphQL 엔드포인트와 함께 @apollo/server와 `@as-integrations/fastify` 패키지를 사용하는 경우, `GraphQLModule` 구성에서 `disableHealthCheck` 설정을 활성화해야 합니다.

코드 우선

코드 우선 접근 방식에서는 데코레이터와 TypeScript 클래스를 사용하여 해당 GraphQL 스키마를 생성합니다.

코드 우선 접근 방식을 사용하려면 먼저 옵션 개체에 `autoSchemaFile` 속성을 추가합니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  autoSchemaFile: join(process.cwd(), 'src/schema.gql'),
}),
```

`자동 스키마 파일` 속성 값은 자동으로 생성된 스키마가 생성될 경로입니다. 또는 스키마를 메모리에서 즉석에서 생성할 수도 있습니다. 이 기능을 사용하려면 `autoSchemaFile` 속성을 `true로` 설정합니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  autoSchemaFile: true,
}),
```

기본적으로 생성된 스키마의 유형은 포함된 모듈에 정의된 순서대로 정렬됩니다. 스키마를 사전순으로 정렬하려면 `sortSchema` 속성을 `true로` 설정하세요:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  autoSchemaFile: join(process.cwd(), 'src/schema.gql'),
  sortSchema: true,
}),
```

예

완전히 작동하는 코드 우선 샘플은 여기에서 확인할

수 있습니다. 스키마 우선

스키마 우선 접근 방식을 사용하려면 먼저 옵션 개체에 `typePaths` 속성을 추가합니다. 이 속성은 `typePaths` 속성은 GraphQLModule이 작성할 GraphQL SDL 스키마 정의 파일을 어디에서 찾아야 하는지를 나타냅니다. 이러한 파일은 메모리에 결합되므로 스키마를 여러 파일로 분할하여 해당 리졸버 근처에서 찾을 수 있습니다.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  typePaths: ['./**/*.graphql'],
}),
```

일반적으로 GraphQL SDL 유형에 해당하는 TypeScript 정의(클래스 및 인터페이스)도 있어야 합니다. 해당 TypeScript 정의를 수작업으로 생성하는 것은 중복되고 지루한 작업입니다. SDL 내에서 변경할 때마다 TypeScript 정의도 조정해야 하므로 신뢰할 수 있는 단일 소스가 없습니다. 이 문제를 해결하기 위해 `@nestjs/graphql` 패키지는 추상 구문 트리(AST)에서 TypeScript 정의를 자동으로 생성할 수 있습니다. 이 기능을 사용하려면 `GraphQLModule을` 구성할 때 `정의` 옵션 속성을 추가하세요.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  typePaths: ['./**/*.graphql'], 정의:
  {
    경로: join(process.cwd(), 'src/graphql.ts'),
  },
}),
```

정의 객체의 경로 속성은 생성된 TypeScript 출력을 저장할 위치를 나타냅니다. 기본적으로 생성된 모든 TypeScript 유형은 인터페이스로 생성됩니다. 대신 클래스를 생성하려면 출력As 속성에 'class' 값을 지정합니다.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  typePaths: ['./**/*.graphql'], 정의:
  {
    경로: join(process.cwd(), 'src/graphql.ts'),
    outputAs: 'class',
  },
}),
```

위의 접근 방식은 애플리케이션이 시작될 때마다 TypeScript 정의를 동적으로 생성합니다. 또는 필요에 따라 생성하는 간단한 스크립트를 작성하는 것이 더 바람직할 수 있습니다. 예를 들어 다음 스크립트를 generate-typings.ts로 생성한다고 가정해 보겠습니다:

```
'@nestjs/graphql'에서 { GraphQLDefinitionsFactory }를 가져오고,
'path'에서 { join }를 가져옵니다;

const definitionsFactory = new GraphQLDefinitionsFactory();
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  경로: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
});
```

이제 이 스크립트를 필요에 따라 실행할 수 있습니다:

```
ts-node 생성 타이핑
```

> 정보 힌트 스크립트를 미리 컴파일한 후(예: tsc를 사용하여) node를 사용하여 실행할 수 있습니다.

스크립트에 대해 감시 모드를 사용하려면(.graphql 파일이 변경될 때마다 자동으로 타이핑을 생성하려면) generate() 메서드에 watch 옵션을 전달합니다.

```
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  경로: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
  watch: true,
});
```

모든 객체 유형에 대해 추가 `__typename` 필드를 자동으로 생성하려면, 모든 객체 유형에 대해 `emitTypenameField` 옵션을 사용합니다.

```
definitionsFactory.generate({
  // ...,
  emitTypenameField: true,
});
```

인수가 없는 일반 필드로 리졸버(쿼리, 돌연변이, 구독)를 생성하려면 `skipResolverArgs` 옵션.

```
definitionsFactory.generate({
  // ...,
  skipResolverArgs: true,
});
```

아폴로 샌드박스

로컬 개발용 GraphQL IDE로 `graphql-playground` 대신 Apollo 샌드박스를 사용하려면 다음 구성을 사용하세요:

```
'@nestjs/apollo'에서 { ApolloDriver, ApolloDriverConfig }를 임포트하고
, '@nestjs/common'에서 { Module }을 임포트합니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;
'@apollo/server/plugin/landingPage/default'에서 {
ApolloServerPluginLandingPageLocalDefault }를 임포트합니다;

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      놀이터: 거짓,
      플러그인: [ApolloServerPluginLandingPageLocalDefault()],
    }),
  ],
})
내보내기 클래스 AppModule {}
```

예

완전히 작동하는 스키마 첫 번째 샘플은 여기에서 확인할 수 있습니

다. 생성된 스키마에 액세스하기

일부 상황(예: 엔드투엔드 테스트)에서는 생성된 스키마 객체에 대한 참조를 얻고 싶을 수 있습니다. 그러면 엔드투엔드 테스트에서 HTTP 리스너를 사용하지 않고 `graphql` 객체를 사용하여 쿼리를 실행할 수 있습니다.

코드 우선 또는 스키마 우선 접근 방식 중 하나를 사용하여 생성된 스키마에 액세스할 수 있습니다.

`GraphQLSchemaHost` 클래스:

```
const { schema } = app.get(GraphQLSchemaHost);
```

> 정보 힌트 애플리케이션이 초기화된 후(`app.listen()` 또는 `app.init()` 메서드에 의해 `onModuleInit` 훅이 트리거된 후) `GraphQLSchemaHost#schema` 게터를 호출해야 합니다.

비동기 구성

모듈 옵션을 정적이 아닌 비동기적으로 전달해야 하는 경우, `forRootAsync()`

메서드를 사용합니다. 대부분의 동적 모듈과 마찬가지로 Nest는 비동기 구성을 처리하는 몇 가지 기술을 제공합니

다. 한 가지 기법은 팩토리 함수를 사용하는 것입니다:

```
 GraphQLModule.forRootAsync<ApolloDriverConfig>({
  driver: ApolloDriver,
  useFactory: () => ({
    typePaths: ['./**/*.graphql'],
  }),
}),
```

다른 팩토리 공급자와 마찬가지로 팩토리 함수는 비동기화될 수 있으며 다음을 통해 종속성을 주입할 수 있습니다.

주입합니다.

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
  driver: ApolloDriver,
  임포트: [구성 모듈],
  useFactory: async (configService: ConfigService) => ({
    typePaths: configService.get<string>('GRAPHQL_TYPE_PATHS'),
  }),
  주입합니다: [구성 서비스],
}),
```

또는 아래와 같이 팩토리 대신 클래스를 사용하여 `GraphQLModule을` 구성할 수도 있습니다:

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
  driver: ApolloDriver,
  사용 클래스: GqlConfigService,
}),
```

위의 구성은 GraphQLModule 내부에 GqlConfigService를 인스턴스화하여 이를 사용하여 옵션 객체를 생성합니다. 이 예제에서 GqlConfigService는 아래와 같이 GqlOptionsFactory 인터페이스를 구현해야 한다는 점에 유의하세요. 그래프QL모듈은 제공된 클래스의 인스턴스화된 객체에서 createGqlOptions() 메서드를 호출합니다.

```
@Injectable()
GqlConfigService 클래스는 GqlOptionsFactory { createGqlOptions() 를
  구현합니다: ApolloDriverConfig {
    반환 {
      typePaths: ['./**/*.graphql'],
    };
  }
}
```

내부에 비공개 복사본을 만드는 대신 기존 옵션 공급자를 재사용하려는 경우 GraphQLModule에 사용Existing 구문을 사용합니다.

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({ 임포트:
  [ConfigModule],
  사용Existing: ConfigService,
}),
```

머큐리우스 통합

아폴로를 사용하는 대신 Fastify 사용자(자세한 내용은 여기를 참조)는 @nestjs/mercurius를 사용할 수 있습니다. 드라이버.

```
@@파일명()
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트하고,
'@nestjs/mercurius'에서 { MercuriusDriver,
MercuriusDriverConfig }를 임포트합니다;

모듈({ import:
  [
    GraphQLModule.forRoot<MercuriusDriverConfig>({
      driver: MercuriusDriver,
      graphiql: true,
    }),
  ],
})
내보내기 클래스 AppModule {}
```

> 정보 힌트 애플리케이션이 실행 중이면 브라우저를 열고 다음 위치로 이동합니다.
>
> http://localhost:3000/graphiql. GraphQL IDE가 표시되어야 합니다.

`forRoot()` 메서드는 옵션 객체를 인자로 받습니다. 이러한 옵션은 기본 드라이버 인스턴스로 전달됩니다. 사용 가능한 설정에 대한 자세한 내용은 여기를 참조하세요.

타사 통합

- GraphQL 요가

예제

작동하는 예제는 여기에서 확인할 수 있습니다.

리졸버

리졸버는 GraphQL 연산(쿼리, 변이 또는 구독)을 데이터로 변환하기 위한 지침을 제공합니다. 스키마에서 지정한 것과 동일한 형태의 데이터를 동기식으로 또는 해당 형태의 결과로 해결되는 프로미스로 반환합니다. 일반적으로 리졸버 맵은 수동으로 생성합니다. 반면에 `@nestjs/graphql` 패키지는 클래스에 주석을 다는 데 사용하는 데코레이터가 제공하는 메타데이터를 사용하여 리졸버 맵을 자동으로 생성합니다. 패키지 기능을 사용하여 GraphQL API를 생성하는 과정을 보여드리기 위해 간단한 작성자 API를 만들어 보겠습니다.

코드 우선

코드 우선 접근 방식에서는 GraphQL SDL을 직접 작성하여 GraphQL 스키마를 생성하는 일반적인 프로세스를 따르지 않습니다. 대신 TypeScript 데코레이터를 사용하여 TypeScript 클래스 정의에서 SDL을 생성합니다. `nestjs/graphql` 패키지는 데코레이터를 통해 정의된 메타데이터를 읽고 스키마를 자동으로 생성합니다.

개체 유형

GraphQL 스키마에 있는 대부분의 정의는 객체 유형입니다. 정의하는 각 객체 유형은 애플리케이션 클라이언트가 상호 작용해야 할 수 있는 도메인 객체를 나타내야 합니다. 예를 들어 샘플 API는 작성자와 해당 글의 목록을 가져올 수 있어야 하므로 이 기능을 지원하기 위해 `작성자` 유형과 `글` 유형을 정의해야 합니다.

스키마 우선 접근 방식을 사용한다면 다음과 같이 SDL로 스키마를 정의할 것입니다:

```
type Author {
  id: Int!
  이름: 성: 문자열입니다:
  문자열 게시물:
  [Post!]!
}
```

이 경우 코드 우선 접근 방식을 사용하면 TypeScript 클래스를 사용하여 스키마를 정의하고 TypeScript 데코레이터를 사용하여 해당 클래스의 필드에 주석을 달 수 있습니다. 코드 우선 접근 방식에서 위의 SDL에 해당하는 것은 다음과 같습니다:

```
@@파일명(저자/모델/저자.모델)
'@nestjs/graphql'에서 { Field, Int, ObjectType } 가져오기;
'./post'에서 { Post } 가져오기;

객체 유형()
export class Author {
  @Field(type => Int)
  id: number;

  @Field({ nullable: true })
  firstName?: 문자열;
```

```
  @Field({ nullable: true })
  lastName?: 문자열;

  필드(유형 => [게시물]) 게시물
  : Post[];
}
```

> 정보 힌트 TypeScript의 메타데이터 반영 시스템에는 몇 가지 제한 사항이 있어 클래스가 어떤 속성으로 구성되어 있는지 확인하거나 주어진 속성이 선택 사항인지 필수 사항인지 인식하는 것이 불가능합니다. 이러한 제한 때문에 스키마 정의 클래스에서 `@Field()` 데코레이터를 명시적으로 사용하여 각 필드의 GraphQL 유형 및 선택 가능성에 대한 메타데이터를 제공하거나 CLI 플러그인을 사용하여 이러한 메타데이터를 생성해야 합니다.

다른 클래스와 마찬가지로 `Author` 객체 유형은 필드 모음으로 구성되며 각 필드는 유형을 선언합니다. 필드의 유형은 GraphQL 유형에 해당합니다. 필드의 GraphQL 유형은 다른 객체 유형 또는 스칼라 유형일 수 있습니다. GraphQL 스칼라 유형은 단일 값으로 해석되는 기본 유형(예: `ID`, `문자열`, `부울` 또는 `Int`)입니다.

> 정보 힌트 GraphQL의 기본 제공 스칼라 유형 외에도 사용자 정의 스칼라 유형을 정의할 수 있습니다(자세히 읽기).

위의 `Author` 객체 유형 정의는 Nest가 위에 표시된 SDL을 생성하도록 합니다:

```
type Author {
  id: Int!
  이름: 성: 문자열입니다:
  문자열 게시물:
  [Post!]!
}
```

`필드()` 데코레이터는 선택적 타입 함수(예: `type => Int`)와 옵션 객체를 허용합니다.

타입 함수는 타입스크립트 타입 시스템과 GraphQL 타입 시스템 간에 모호한 부분이 있을 때 필요합니다. 구체적으로, `문자열` 및 `부울` 유형에는 필요하지 않지만 `숫자`(GraphQL `Int` 또는 `Float`에 매핑되어야 함)에는 필요합니다. 타입 함수는 단순히 원하는 GraphQL 타입을 반환해야 합니다(이 장의 다양한 예제에서 볼 수 있듯이).

옵션 객체는 다음 키/값 쌍 중 하나를 가질 수 있습니다:

- `nullable`: 필드의 널 가능 여부를 지정합니다(SDL에서 각 필드는 기본적으로 널 가능하지 않습니다); 부울
- `설명`: 필드 설명 설정용; 문자열
- 사용 중단 `이유`: 필드를 사용 중단으로 표시하는 경우; 문자열

예를 들어

```
@Field({ description: '책 제목', deprecationReason: 'v2 스키마에서 유용하지 않음' })
제목: 문자열;
```

> 정보 힌트 전체 객체 유형에 설명을 추가하거나 더 이상 사용하지 않을 수도 있습니다: 객체 `유형({{ '{' }} 설명: '작성자 모델' {{ '}' }})`.

필드가 배열인 경우 아래와 같이 `Field()` 데코레이터의 유형 함수에서 배열 유형을 수동으로 표시해야 합니다:

```
필드(유형 => [게시물]) 게시물:
Post[];
```

> 정보 힌트 배열 괄호 표기법(`[ ]`)을 사용하면 배열의 깊이를 나타낼 수 있습니다. 예를 들어 `[[Int]]`를 사용하면 정수 행렬을 나타낼 수 있습니다.

배열의 항목(배열 자체가 아닌)이 널 가능함을 선언하려면 아래와 같이 `널 가능` 속성을 `'items'`로 설정합니다:

```
@Field(type => [Post], { nullable: 'items' })
posts: Post[];
```

> 정보 힌트 배열과 해당 항목이 모두 `널러블인` 경우, 대신 `널러블을 'itemsAndList'`로 설정하세요.

이제 `작성자` 객체 유형이 생성되었으므로 `게시물` 객체 유형을 정의해 보겠습니다.

```
@@파일명(posts/models/post.model)
'@nestjs/graphql'에서 { Field, Int, ObjectType }을 가져옵니다;

객체 유형() 내보내기 클
래스 Post {
  @Field(유형 => Int)
  id: 숫자;

  @Field()
  title: 문자열;

  @Field(유형 => Int, { nullable: true })
  votes?: 숫자;
}
```

`Post` 개체 유형은 SDL에서 GraphQL 스키마의 다음 부분을 생성하게 됩니다:

```
유형 Post {
  id: Int!
  title: 문자열! 투표:
  Int
}
```

코드 우선 해결자

이 시점에서 데이터 그래프에 존재할 수 있는 객체(유형 정의)를 정의했지만 클라이언트는 아직 해당 객체와 상호 작용할 수 있는 방법이 없습니다. 이 문제를 해결하려면 리졸버 클래스를 만들어야 합니다. 코드 퍼스트 방법에서 리졸버 클래스는 리졸버 함수를 정의하고 쿼리 유형을 생성합니다. 이는 아래 예제를 진행하면서 명확해질 것입니다:

```
@@파일명(저자/저자.해결자) @Resolver(of =>
저자)
내보내기 클래스 AuthorsResolver { 생성자(
    비공개 작성자 서비스: AuthorsService, 비공개
    postsService: PostsService,
  ) {}

  쿼리(반환값 => 작성자)
  async author(@Args('id', { type: () => Int }) id: number) {
    return this.authorsService.findOneById(id);
  }

  @ResolveField()
  async posts(@Parent() author: Author) {
    const { id } = author;
    return this.postsService.findAll({ authorId: id });
  }
}
```

> 정보 힌트 모든 데코레이터(예: `@Resolver`, `@ResolveField`, `@Args` 등)는 `nestjs/graphql` 패키지.

여러 리졸버 클래스를 정의할 수 있습니다. Nest는 런타임에 이를 결합합니다. 코드 구성에 대한 자세한 내용은 아래 모듈 섹션을 참조하세요.

> 경고 `AuthorsService` 및 `PostsService` 클래스 내부의 로직은 필요에 따라 단순하거나 정교하게 만들 수 있습니다. 이 예제의 요점은 리졸버를 구성하는 방법과 리졸버가 다른 공급자와 상호 작용하는 방법을 보여주기 위한 것입니다.

위의 예제에서는 하나의 쿼리 리졸버 함수와 하나의 필드 리졸버 함수를 정의하는 `AuthorsResolver`를 만들었습니다. 리졸버를 생성하려면 리졸버 함수를 메서드로 사용하는 클래스를 생성하고 `@Resolver()` 데코레이터를 사용하여 클래스에 주석을 달면 됩니다.

이 예제에서는 요청에 전송된 `ID를` 기반으로 작성자 개체를 가져오는 쿼리 처리기를 정의했습니다. 메서드가 쿼리 핸들러임을 지정하려면 `@Query()` 데코레이터를 사용합니다.

`Resolver()` 데코레이터에 전달되는 인수는 선택 사항이지만 그래프가 사소하지 않게 될 때 사용됩니다. 필드 리졸버 함수가 객체 그래프를 내려갈 때 사용하는 부모 객체를 제공하는 데 사용됩니다.

이 예제에서는 클래스에 필드 리졸버 함수(`Author` 객체 유형의 `게시물` 속성에 대한)가 포함되어 있으므로 이 클래스 내에 정의된 모든 필드 리졸버의 부모 유형(즉, 해당 `ObjectType` 클래스 이름)을 나타내는 값을 `@Resolver()` 데코레이터에 제공해야 합니다. 예제에서 알 수 있듯이 필드 해석기 함수를 작성할 때는 부모 객체(해석되는 필드가 멤버로 있는 객체)에 액세스해야 합니다. 이 예제에서는 작성자의 `ID를` 인수로 사용하는 서비스를 호출하는 필드 해석기로 작성자의 게시물 배열을 채웁니다. 따라서 `@Resolver()` 데코레이터에서 부모 객체를 식별해야 합니다. 필드 리졸버에서 해당 부모 객체에 대한 참조를 추출하려면 `@Parent()` 메서드 매개변수 데코레이터의 해당 사용에 유의하세요.

이 클래스와 다른 리졸버 클래스 모두에서 여러 개의 `@Query()` 리졸버 함수를 정의할 수 있으며, 이러한 함수는 리졸버 맵의 적절한 항목과 함께 생성된 SDL에서 단일 쿼리 유형 정의로 집계됩니다. 이를 통해 쿼리를 사용하는 모델 및 서비스에 가깝게 정의하고 모듈에서 잘 정리된 상태로 유지할 수 있습니다.

> 정보 힌트 네스트 CLI는 모든 상용구 코드를 자동으로 생성하는 제너레이터(회로도)를 제공하여 이 모든 작업을 피하고 개발자 환경을 훨씬 더 간단하게 만들 수 있도록 도와줍니다. 이 기능에 대한 자세한 내용은 여기에서 확인하세요.

쿼리 유형 이름

위의 예제에서 `@Query()` 데코레이터는 메서드 이름을 기반으로 GraphQL 스키마 쿼리 유형 이름을 생성합니다. 예를 들어 위의 예제에서 다음 구성을 고려해 보겠습니다:

```
쿼리(반환값 => 작성자)
async author(@Args('id', { type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

이렇게 하면 스키마에서 작성자 쿼리에 대한 다음 항목이 생성됩니다(쿼리 유형은 메서드 이름과 동일한 이름을

사용함):

```
유형 쿼리 {
  author(id: Int!): 저자
}
```

> 정보 힌트 GraphQL 쿼리에 대한 자세한 내용은 여기를 참조하세요.

일반적으로 이러한 이름을 분리하는 것을 선호합니다. 예를 들어 다음과 같은 이름을 사용하는 것을 선호합니다.

`getAuthor()` 메서드 대신 쿼리 핸들러 메서드를 사용하되 쿼리 유형 이름에는 여전히 `author를 사용합니다.` 동일한

는 필드 리졸버에 적용됩니다. 매핑 이름을 함수 호출의 인수로 전달하면 쉽게 이 작업을 수행할 수 있습니다.

쿼리() 및 @ResolveField() 데코레이터를 사용할 수 있습니다:

```
@@파일명(저자/저자.해결자) @Resolver(of =>
저자)
내보내기 클래스 AuthorsResolver { 생성자(
    비공개 작성자 서비스: AuthorsService, 비공개
    postsService: PostsService,
  ) {}

  쿼리(반환값 => 작성자, { 이름: '작성자' })
  async getAuthor(@Args('id', { type: () => Int }) id: number) {
    return this.authorsService.findOneById(id)를 반환합니다;
  }

  ResolveField('posts', returns => [Post])
  async getPosts(@Parent() author: Author) {
    const { id } = 작성자;
    return this.postsService.findAll({ authorId: id });
  }
}
```

위의 getAuthor 핸들러 메서드는 SDL에서 GraphQL 스키마의 다음 부분을 생성하게 됩니다:

```
유형 쿼리 {
  author(id: Int!): 저자
}
```

쿼리 데코레이터 옵션

쿼리() 데코레이터의 옵션 객체(여기서 {{ '{' }}이름: 'author'{{ '}' }}

위)는 여러 키/값 쌍을 허용합니다:

- 이름: 쿼리 이름; 문자열
- 설명: GraphQL 스키마 문서를 생성하는 데 사용되는 설명(예: GraphQL 플레이그라운드); 문자열
- deprecationReason: 쿼리를 더 이상 사용되지 않는 것으로 표시하도록 쿼리 메타데이터를 설정합니다(예: GraphQL 플레이그라운드에서); 문자열
- nullable: 쿼리가 null 데이터 응답을 반환할 수 있는지 여부; 부울 또는 'items' 또는

'itemsAndList'('items' 및 'itemsAndList'에 대한 자세한 내용은 위 참조)

Args 데코레이터 옵션

메서드 핸들러에서 사용할 인수를 요청에서 추출하려면 @Args() 데코레이터를 사용합니다. 다음과 같이 작동합니다.

를 REST 경로 매개변수 인자 추출과 매우 유사한 방식으로 사용합니다.

일반적으로 `@Args()` 데코레이터는 간단하며, 위의 `getAuthor()` 메서드에서 볼 수 있듯이 객체 인수가 필요하지 않습니다. 예를 들어 식별자 유형이 문자열인 경우 다음과 같은 구조로 충분하며, 인바운드 GraphQL 요청에서 명명된 필드를 가져와 메서드 인수로 사용하기만 하면 됩니다.

```
@Args('id') id: 문자열
```

`getAuthor()`의 경우 `숫자` 유형이 사용되므로 문제가 발생합니다. `숫자` 타입스크립트 유형은 예상되는 GraphQL 표현에 대한 충분한 정보를 제공하지 않습니다(예: `Int` 대 `Float`). 따라서 명시적으로 타입 참조를 전달해야 합니다. 이를 위해 아래 그림과 같이 인수 옵션을 포함하는 두 번째 인수를 `Args()` 데코레이터에 전달합니다:

```
쿼리(반환값 => 작성자, { 이름: '작성자' })
async getAuthor(@Args('id', { type: () => Int }) id: number) {
  return this.authorsService.findOneById(id)를 반환합니다;
}
```

옵션 객체를 사용하면 다음과 같은 선택적 키 값 쌍을 지정할 수 있습니다:

- `type`: GraphQL 유형을 반환하는 함수
- `defaultValue`: 기본값; `any`
- `설명`: 설명 메타데이터, `문자열`
- `deprecationReason`: 필드를 사용 중단하고 그 이유를 설명하는 메타 데이터를 제공합니다.

`문자열`• `nullable`: 필드가 null 가능한지 여부입니다.

쿼리 핸들러 메서드는 여러 인수를 받을 수 있습니다. 이름과 `성을` 기준으로 작성자를 가져오고 싶다고 가정해 보겠습니다. 이 경우 `@Args를` 두 번 호출하면 됩니다:

```
getAuthor(
  @Args('firstName', { nullable: true }) firstName?: 문자열,
  @Args('lastName', { defaultValue: '' }) lastName?: 문자열입니다,
) {}
```

전용 인수 클래스

인라인 `@Args()` 호출을 사용하면 위 예제와 같은 코드가 부풀어 오릅니다. 대신 다음과 같이 전용 `GetAuthorArgs` 인수 클래스를 생성하고 핸들러 메서드에서 액세스하면 됩니다:

```
@Args() args: GetAuthorArgs
```

아래와 같이 `@ArgsType()`을 사용하여 `GetAuthorArgs` 클래스를 생성합니다:

```
@@파일명(authors/dto/get-author.args) 'class-
validator'에서 { MinLength }를 가져옵니다;
'@nestjs/graphql'에서 { Field, ArgsType }을 가져옵니다;

@ArgsType()
GetAuthorArgs 클래스 {
  @Field({ nullable: true })
  firstName?: 문자열;

  @Field({ defaultValue: '' })
  @MinLength(3)
  성: 문자열입니다;
}
```

> 정보 힌트 다시 한 번 말씀드리지만, TypeScript의 메타데이터 반영 시스템 제한으로 인해 `@Field` 데코레이터를 사용하여 유형 및 선택 사항을 수동으로 표시하거나 CLI 플러그인을 사용해야 합니다.

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
유형 쿼리 {
  저자(이름: 문자열, 성: 문자열 = ''): Author
}
```

> 정보 힌트 GetAuthorArgs와 같은 인자 클래스는 `유효성 검사 파이프` (자세히 읽기).

클래스 상속

표준 TypeScript 클래스 상속을 사용하여 확장할 수 있는 일반 유틸리티 유형 기능(필드 및 필드 속성, 유효성 검사 등)이 있는 기본 클래스를 만들 수 있습니다. 예를 들어, 항상 표준 `오프셋` 및 `제한` 필드뿐만 아니라 유형별로 다른 인덱스 필드를 포함하는 페이지 매김 관련 인수 집합을 가질 수 있습니다. 아래와 같이 클래스 계층 구조를 설정할 수 있습니다.

베이스 `@ArgsType()` 클래스:

```
@ArgsType()
PaginationArgs 클래스 {
  @Field((type) => Int)
  offset: number = 0;

  @Field((type) => Int)
  limit: number = 10;
}
```

기본 `@ArgsType()` 클래스의 특정 하위 클래스를 입력합니다:

```
@ArgsType()
GetAuthorArgs 클래스 PaginationArgs 확장 { @Field({
  nullable: true })
  firstName?: 문자열;

  @Field({ defaultValue: '' })
  @MinLength(3)
  성: 문자열입니다;
}
```

`ObjectType()` 객체에서도 동일한 접근 방식을 사용할 수 있습니다. 베이스 클래스에서 제네릭 프로퍼티를 정의합니다:

```
객체 유형() 클래스
Character {
  @Field((유형) => Int)
  id: 숫자;

  @Field() 이름:
  문자열;
}
```

하위 클래스에 유형별 속성을 추가합니다:

```
객체 유형()
Warrior 클래스 확장자 Character {
  @Field()
  수준: 숫자;
}
```

리졸버와 함께 상속을 사용할 수도 있습니다. 상속과 TypeScript 제네릭을 결합하여 유형 안전을 보장할 수 있습니다. 예를 들어, 일반 `findAll` 쿼리가 있는 기본 클래스를 만들려면 다음과 같은 구성을 사용합니다:

```
함수 BaseResolver<T extends Type<unknown>>(classRef: T): any {
  @Resolver({ isAbstract: true })
  추상 클래스 BaseResolverHost {
    @Query((type) => [classRef], { name: `findAll${classRef.name}` })
    async findAll(): Promise<T[]> {
      반환 [];
    }
  }
  BaseResolverHost를 반환합니다;
}
```

다음 사항에 유의하세요:

- 명시적 반환 유형(위 중 하나)이 필요합니다. 그렇지 않으면 TypeScript에서 비공개 클래스 정의 사용에 대해 불만을 표시합니다. 권장: 아무 것도 사용하지 말고 인터페이스를 정의하세요.
- 유형은 @nestjs/common 패키지에서 가져옵니다.
- isAbstract: true 속성은 이 클래스에 대해 SDL(스키마 정의 언어 문)을 생성하지 않아야 함을 나타냅니다. 다른 유형에 대해서도 이 속성을 설정하여 SDL 생성을 억제할 수 있습니다.

베이스 리졸버의 구체적인 하위 클래스를 생성하는 방법은 다음과 같습니다:

```
@Resolver((of) => Recipe)
export class RecipesResolver extends BaseResolver(Recipe) {
  constructor(private recipesService: RecipesService) {
    super();
  }
}
```

이 구성은 다음과 같은 SDL을 생성합니다:

```
유형 쿼리 {
  findAllRecipe: [Recipe!]!
}
```

제네릭

위에서 제네릭의 한 가지 용도를 살펴봤습니다. 이 강력한 타입스크립트 기능은 유용한 추상화를 만드는 데 사용할 수 있습니다. 예를 들어 이 문서를 기반으로 커서 기반 페이지 매김을 구현한 샘플은 다음과 같습니다:

```
'@nestjs/graphql'에서 { Field, ObjectType, Int }를 가져오고,
'@nestjs/common'에서 { Type }을 가져옵니다;

인터페이스 IEdgeType<T> { 커
  서: 문자열;
  노드: T;
}

내보내기 인터페이스 IPaginatedType<T> { 가
  장자리: IEdgeType<T>[];
  노드를 반환합니다: T[];
  totalCount: 숫자;
  hasNextPage: 부울;
}

내보내기 함수 Paginated<T>(classRef: Type<T>): Type<IPaginatedType<T>> {
  @ObjectType(`${classRef.name}Edge`)
```

```
  추상 클래스 EdgeType {
    @Field((type) => String)
    cursor: string;

    @Field((type) => classRef) 노
    드: T;
  }

  객체 유형({ isAbstract: true })
  추상 클래스 PaginatedType 구현 IPaginatedType<T> { @Field((type) =>
    [EdgeType], { nullable: true })
    에지: EdgeType[];

    @Field((type) => [classRef], { nullable: true }) 노
    드: T[];

    @Field((type) => Int)
    totalCount: 숫자;

    @Field()
    hasNextPage: 부울;
  }
  PaginatedType을 Type<IPaginatedType<T>>로 반환합니다;
}
```

위의 기본 클래스가 정의되었으므로 이제 이 동작을 상속하는 특수 유형을 쉽게 만들 수 있습니다. 예를 들어

```
객체 유형()
PaginatedAuthor 클래스 Paginated(Author) 확장 {}
```

스키마 우선

이전 장에서 언급했듯이, 스키마 우선 접근 방식에서는 SDL에서 스키마 유형을 수동으로 정의하는 것으로 시작합니다(자세히 보기). 다음 SDL 유형 정의를 고려하세요.

> 정보 힌트 이 장에서는 편의를 위해 모든 SDL을 한 위치(예: 하나의

```
type Author {
  id: Int!
  이름: 성: 문자열입니다:
  문자열 글입니다: [포스트
  ]
```

```
}

유형 Post {
  id: Int!
  title: 문자열! 투표:
  Int
}

유형 쿼리 {
  author(id: Int!): 저자
}
```

스키마 우선 해결자

위의 스키마는 단일 쿼리인 `author(id: Int!)`를 노출합니다: `Author`.

> 정보 힌트 GraphQL 쿼리에 대한 자세한 내용은 여기를 참조하세요.

이제 작성자 쿼리를 해결하는 `AuthorsResolver` 클래스를 만들어 보겠습니다:

```
@@파일명(저자/저자.해결자) @Resolver('저자')
내보내기 클래스 AuthorsResolver { 생성자(
    비공개 작성자 서비스: AuthorsService, 비공개
    postsService: PostsService,
  ) {}

  쿼리()
  async author(@Args('id') id: number) {
    this.authorsService.findOneById(id)를 반환합니다;
  }

  @ResolveField()
  async posts(@Parent() author) {
    const { id } = author;
    return this.postsService.findAll({ authorId: id });
  }
}
```

> 정보 힌트 모든 데코레이터(예: `@Resolver`, `@ResolveField`, `@Args` 등)는 `nestjs/graphql` 패키지.

> 경고 `AuthorsService` 및 `PostsService` 클래스 내부의 로직은 필요에 따라 단순하거나 정교하게 만들 수 있습니다. 이 예제의 요점은 리졸버를 구성하는 방법과 리졸버가 다른 공급자와 상호 작용하는 방법을 보여주기 위한 것입니다.

`Resolver()` 데코레이터는 필수입니다. 이 데코레이터는 클래스 이름이 포함된 선택적 문자열 인수를 받습니다. 이 클래스 이름은 클래스에 `@ResolveField()` 데코레이터가 포함될 때마다 Nest에 해당 클래스가

장식된 메서드는 부모 유형(현재 예제에서는 `Author` 유형)과 연관되어 있습니다. 또는 클래스 상단에 `@Resolver()`를 설정하는 대신 각 메서드에 대해 이 작업을 수행할 수 있습니다:

```
Resolver('Author')
@ResolveField()
async posts(@Parent() author) {
  const { id } = author;
  return this.postsService.findAll({ authorId: id });
}
```

이 경우(메서드 레벨의 `@Resolver()` 데코레이터), 클래스 내에 여러 개의 `@ResolveField()` 데코레이터가 있는 경우 모든 데코레이터에 `@Resolver()`를 추가해야 합니다. 이는 추가 오버헤드가 발생하므로 모범 사례로 간주되지 않습니다.

> 정보 힌트 `@Resolver()`에 전달된 클래스 이름 인수는 쿼리(`@Query()` 데코레이터) 또는 돌연변이(`@Mutation()` 데코레이터)에 영향을 주지 않습니다.

> 경고 메서드 수준에서 `@Resolver` 데코레이터를 사용하는 것은 코드 우선 접근 방식에서 지원되지 않습니다.

위의 예에서 `@Query()` 및 `@ResolveField()` 데코레이터는 메서드 이름에 따라 GraphQL 스키마 유형에 연결됩니다. 예를 들어 위의 예제에서 다음 구성을 고려해 보겠습니다:

```
쿼리()
async author(@Args('id') id: number) {
  this.authorsService.findOneById(id)를 반환합니다;
}
```

이렇게 하면 스키마에서 작성자 쿼리에 대한 다음 항목이 생성됩니다(쿼리 유형은 메서드 이름과 동일한 이름을 사용함):

```
유형 쿼리 {
  author(id: Int!): 저자
}
```

일반적으로는 리졸버 메서드에 `getAuthor()` 또는 `getPosts()`와 같은 이름을 사용하여 이를 분리하는 것을 선호

합니다. 아래 그림과 같이 데코레이터에 매핑 이름을 인자로 전달하면 쉽게 분리할 수 있습니다:

```
@@파일명(저자/저자.해결자) @Resolver('저자')
내보내기 클래스 AuthorsResolver { 생성자(
    비공개 authorsService: AuthorsService,
```

```
    비공개 게시 서비스: 게시 서비스,
  ) {}

  @Query('author')
  async getAuthor(@Args('id') id: number) {
    return this.authorsService.findOneById(id);
  }

  @ResolveField('posts')
  async getPosts(@Parent() author) {
    const { id } = author;
    return this.postsService.findAll({ authorId: id });
  }
}
```

> 정보 힌트 네스트 CLI는 모든 상용구 코드를 자동으로 생성하는 제너레이터(회로도)를 제공하여 이 모든 작업을 피하고 개발자 환경을 훨씬 더 간단하게 만들 수 있도록 도와줍니다. 이 기능에 대한 자세한 내용은 여기에서 확인하세요.

유형 생성

스키마 우선 접근 방식을 사용하고 유형 생성 기능을 활성화했다고 가정하면(이전 장에 표시된 것처럼 `출력As: 'class'` 사용), 애플리케이션을 실행하면 다음 파일이 생성됩니다(`GraphQLModule.forRoot()` 메서드에서 지정한 위치에). 예를 들어, `src/graphql.ts에 있습니다`:

```
@@파일명(그래프명) 내보내기
(클래스 저자 {
  id: 숫자; 이름?: 문자
  열; 성?: 문자열;
  posts?: Post[];
})
export class Post {
  id: 숫자; title:
  문자열; votes?: 숫
  자;
}

내보내기 추상 클래스 IQuery {
  추상 저자(ID: 숫자): 저자 | 약속<저자>;
}
```

인터페이스를 생성하는 기본 기법 대신 클래스를 생성하면 스키마 우선 접근 방식과 함께 선언적 유효성 검사 데코레이터를 사용할 수 있으며, 이는 매우 유용한 기법입니다(자세히 읽기). 예를 들어, 아래와 같이 생성된 `CreatePostInput` 클래스에 `클래스 유효성` 검사 데코레이터를 추가하여 `제목` 필드에 최소 및 최대 문자열 길이를 적용할 수 있습니다:

```
'class-validator'에서 { MinLength, MaxLength }를 가져옵니다;

export 클래스 CreatePostInput {
  @MinLength(3)
  @MaxLength(50)
  title: 문자열;
}
```

> 경고 주의 입력(및 매개변수)의 자동 유효성 검사를 사용하려면 `ValidationPipe`를 사용하세요. 유효성 검사에 대한 자세한 내용은 여기를, 파이프에 대한 자세한 내용은 여기를 참조하세요.

그러나 자동 생성된 파일에 데코레이터를 직접 추가하면 파일이 생성될 때마다 덮어쓰게 됩니다. 대신 별도의 파일을 생성하고 생성된 클래스를 확장하기만 하면 됩니다.

```
'class-validator'에서 { MinLength, MaxLength }를 가져오고,
'../../graphql.ts'에서 { Post }를 가져옵니다;

export 클래스 CreatePostInput extends Post {
  @MinLength(3)
  @MaxLength(50)
  title: 문자열;
}
```

GraphQL 인수 데코레이터

전용 데코레이터를 사용하여 표준 GraphQL 리졸버 인자에 액세스할 수 있습니다. 아래는 Nest 데코레이터와 이들이 나타내는 일반 아폴로 매개변수를 비교한 것입니다.

루트() 및 @부모()	루트/부모
@컨텍스트(파라미터?: 문자열)	context /context[파라미터]
@정보(파라미터?: 문자열)	정보/정보[파라미터]
@아르그(파라미터?: 문자열)	args/ args[파라미터]

이러한 인수의 의미는 다음과 같습니다:

- `root`: 상위 필드에 있는 리졸버에서 반환된 결과를 포함하는 객체 또는 최상위 `쿼리` 필드의 경우 서버

구성에서 전달된 `rootValue`입니다.

- `컨텍스트`: 특정 쿼리의 모든 리졸버가 공유하는 객체로, 일반적으로 요청별 상태를 포함하는 데 사용됩니다.
- `정보`: 쿼리 실행 상태에 대한 정보를 포함하는 객체입니다.
- `args`: 쿼리에서 필드에 전달된 인수가 있는 객체입니다.

모듈

위의 단계를 완료하면, 리졸버 맵을 생성하는 데 필요한 모든 정보를 GraphQLModule에 선언적으로 지정했습니다. 그래프QL모듈은 리플렉션을 사용하여 데코레이터를 통해 제공된 메타 데이터를 인트로스펙트하고 클래스를 올바른 리졸버 맵으로 자동 변환합니다.

처리해야 할 다른 유일한 작업은 리졸버 클래스(AuthorsResolver)를 제공하고(즉, 일부 모듈에 공급자로 나열) 모듈(AuthorsModule)을 어딘가에 가져와야 Nest가 이를 활용할 수 있습니다.

예를 들어 이 컨텍스트에 필요한 다른 서비스도 제공할 수 있는 AuthorsModule에서 이 작업을 수행할 수 있습니다. AuthorsModule을 루트 모듈이나 루트 모듈에서 임포트한 다른 모듈 등 어딘가에 임포트해야 합니다.

```
@@파일명(authors/authors.module)
@Module({
  수입: [PostsModule],
  공급자: [AuthorsService, AuthorsResolver],
})
내보내기 클래스 AuthorsModule {}
```

> 정보 힌트 소위 도메인 모델별로 코드를 구성하는 것이 도움이 됩니다(REST API에서 엔트리 포인트를 구성하는 방식과 유사). 이 접근 방식에서는 도메인 모델을 나타내는 Nest 모듈 내에 모델(ObjectType 클래스), 리졸버 및 서비스를 함께 보관하세요. 이러한 모든 구성 요소를 모듈당 하나의 폴더에 보관하세요. 이렇게 하고 Nest CLI를 사용하여 각 요소를 생성하면 Nest가 이러한 모든 부분을 자동으로 연결(적절한 폴더에 파일 찾기, 공급자 및 가져오기 배열에 항목 생성 등)해 줍니다.

돌연변이

GraphQL에 대한 대부분의 논의는 데이터 가져오기에 초점을 맞추고 있지만, 완전한 데이터 플랫폼에는 서버 측 데이터를 수정할 수 있는 방법도 필요합니다. REST에서는 모든 요청이 서버에 부작용을 일으킬 수 있지만, 모범 사례에 따르면 GET 요청에서 데이터를 수정하지 않는 것이 좋습니다. GraphQL도 비슷합니다. 기술적으로는 모든 쿼리가 데이터 쓰기를 유발하도록 구현될 수 있습니다. 그러나 REST와 마찬가지로 쓰기를 유발하는 모든 작업은 변형을 통해 명시적으로 전송해야 한다는 규칙을 준수하는 것이 좋습니다(자세한 내용은 여기를 참조하세요).

공식 아폴로 문서에서는 `upvotePost()` 변이 예제를 사용합니다. 이 변이는 게시물의 `투표` 속성 값을 증가시키는 메서드를 구현합니다. Nest에서 이와 동등한 변형을 만들려면 `@Mutation()` 데코레이터를 사용하겠습니다.

코드 우선

이전 섹션에서 사용한 `AuthorResolver에` 다른 메서드를 추가해 보겠습니다(리졸버 참조).

```
@Mutation(returns => Post)
async upvotePost(@Args({ name: 'postId', type: () => Int }) postId:
number) {
  this.postsService.upvoteById({ id: postId })를 반환합니다;
}
```

> 정보 힌트 모든 데코레이터(예: `@Resolver`, `@ResolveField`, `@Args` 등)는 `nestjs/graphql` 패키지.

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
유형 돌연변이 { upvotePost(postId:
  Int!): Post
}
```

`upvotePost()` 메서드는 `postId(Int)`를 인자로 받아 업데이트된 `Post` 엔티티를 반환합니다. 리졸버 섹션에서 설명한 이유 때문에 예상 유형을 명시적으로 설정해야 합니다.

돌연변이가 객체를 인자로 받아야 하는 경우 입력 유형을 만들 수 있습니다. 입력 유형은 인자로 전달할 수 있는 특수한 종류의 객체 유형입니다(자세한 내용은 여기를 참조하세요). 입력 유형을 선언하려면 `@InputType()` 데코레이터를 사용합니다.

```
'@nestjs/graphql'에서 { InputType, Field } 임포트;

@InputType()
내보내기 클래스 UpvotePostInput {
  @Field()
  postId: 번호;
}
```

> 정보 힌트 `@InputType()` 데코레이터는 옵션 객체를 인수로 받으므로 예를 들어 입력 유형에 대한 설명을 지정할 수 있습니다. TypeScript의 메타데이터 반영 시스템 제한으로 인해 `@Field` 데코레이터를 사용하여 유형을 수동으로 표시하거나 CLI 플러그인을 사용해야 합니다.

그런 다음 리졸버 클래스에서 이 유형을 사용할 수 있습니다:

```
@Mutation(returns => Post)
async upvotePost(
  @Args('upvotePostData') upvotePostData: 업보트포스트입력,
) {}
```

스키마 우선

이전 섹션에서 사용한 `AuthorResolver`를 확장해 보겠습니다(리졸버 참조).

```
@Mutation()
async upvotePost(@Args('postId') postId: number) {
  return this.postsService.upvoteById({ id: postId });
}
```

위에서 비즈니스 로직이 게시물을 쿼리하고 해당 `투표` 속성을 증가시키는 등의작업이 `PostsService`로 옮겨졌다고 가정했습니다. `PostsService` 클래스 내부의 로직은 필요에 따라 단순하거나 정교하게 만들 수 있습니다. 이 예제의 요점은 리졸버가 다른 공급자와 상호 작용하는 방법을 보여주기 위한 것입니다.

마지막 단계는 기존 유형 정의에 돌연변이를 추가하는 것입니다.

```
type Author {
  id: Int!
  이름: 성: 문자열입니다:
  문자열 글입니다: [포스트
  ]
}

유형 Post {
  id: Int!
  제목: 문자열 투표:
  Int
}

유형 쿼리 {
  author(id: Int!): 저자
}
```

유형 돌연변이 {

```
  upvotePost(postId: Int!): Post
}
```

`upvotePost(postId: Int!):` 이제 애플리케이션의 GraphQL API의 일부로 `포스트` 변형을 호출할 수 있습니다.

구독

쿼리를 사용하여 데이터를 가져오고 변형을 사용하여 데이터를 수정하는 것 외에도 GraphQL 사양은 구독이라는 세 번째 작업 유형을 지원합니다. GraphQL 구독은 서버에서 서버의 실시간 메시지를 수신하도록 선택한 클라이언트로 데이터를 푸시하는 방법입니다. 구독은 클라이언트에 전달할 필드 집합을 지정한다는 점에서 쿼리와 유사하지만, 단일 응답을 즉시 반환하는 대신 서버에서 특정 이벤트가 발생할 때마다 채널이 열리고 결과가 클라이언트로 전송됩니다.

구독의 일반적인 사용 사례는 새 개체 생성, 업데이트된 필드 등 특정 이벤트에 대해 클라이언트 측에 알리는 것입니다(자세한 내용은 여기를 참조하세요).

Apollo 드라이버로 구독 사용 설정하기

구독을 사용하도록 설정하려면 `installSubscriptionHandlers` 속성을 `true로` 설정합니다.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  installSubscriptionHandlers: true,
}),
```

> 경고 `설치 구독 핸들러` 구성 옵션은 최신 버전의 Apollo 서버에서 제거되었으며 이 패키지에서도 곧 더 이상 사용되지 않을 예정입니다. 기본적으로 installSubscriptionHandlers는 `구독-트랜스포트-ws`(자세히 보기)를 사용하도록 대체하지만, 대신 `graphql-ws`(자세히 보기) 라이브러리를 사용할 것을 강력히 권장합니다.

대신 `graphql-ws` 패키지를 사용하도록 전환하려면 다음 구성을 사용합니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  구독을 추가합니다: {
    'graphql-ws': true
  },
}),
```

> 정보 힌트 예를 들어 이전 버전과의 호환성을 위해 두 패키지(`구독-트랜스포트-ws` 및 `그래프ql-ws`)를 동시에 사용할 수도 있습니다.

코드 우선

코드 퍼스트 접근 방식을 사용하여 구독을 생성하려면 `@Subscription()` 데코레이터(`@nestjs/graphql` 패키지에서 내보낸 것)와 간단한 게시/구독 API를 제공하는 `graphql-subscriptions` 패키지의 `PubSub` 클래스를 사용합니다.

다음 구독 핸들러는 다음을 호출하여 이벤트 구독을 처리합니다.

`PubSub#asyncIterator`. 이 메서드는 단일 인자, `트리거 이름(triggerName)`을 받습니다.

이벤트 주제 이름입니다.

```
const pubSub = new PubSub();

@Resolver((of) => Author)
export class AuthorResolver {
  // ...
  구독((반환값) => 댓글) commentAdded() {
    pubSub.asyncIterator('commentAdded')를 반환합니다;
  }
}
```

> 정보 힌트 모든 데코레이터는 `@nestjs/graphql` 패키지에서 내보내지만 `PubSub` 클래스는 `graphql-subscriptions` 패키지에서 내보냅니다.
> 경고 참고 PubSub는 간단한 `게시` 및 `구독 API를` 노출하는 클래스입니다. 여기에서 자세히 알아보세요.
> Apollo 문서에서는 기본 구현이 프로덕션에 적합하지 않다고 경고하고 있습니다(여기에서 자세히 읽어보세요). 프로덕션 앱은 외부 스토어에서 지원하는 `PubSub` 구현을 사용해야 합니다(자세한 내용은 여기를 참조하세요).

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
유형 구독 { commentAdded():
  코멘트!
}
```

구독은 정의상 구독의 이름을 키로 하는 단일 최상위 프로퍼티를 가진 객체를 반환한다는 점에 유의하세요. 이 이름은 구독 핸들러 메서드의 이름에서 상속되거나(예: 위의 `commentAdded),` 아래 표시된 것처럼 키 `이름을` 두 번째 인수로 하는 옵션을 `@Subscription()` 데코레이터에 전달하여 명시적으로 제공될 수 있습니다.

```
구독(반환값 => 댓글, { 이름: '댓글 추가됨',
})
subscribeToCommentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

이 구성은 이전 코드 샘플과 동일한 SDL을 생성하지만 메서드 이름을 구독에서 분리할 수 있습니다.

게시

이제 이벤트를 게시하기 위해 `PubSub#publish` 메서드를 사용합니다. 이 메서드는 오브젝트 그래프의 일부가 변경되었을 때 클라이언트 측 업데이트를 트리거하기 위해 변이 내에서 자주 사용됩니다. 예를 들어

```
@@파일명(posts/posts.resolver)
@Mutation(returns => Post)
비동기 추가 코멘트(
  @Args('postId', { type: () => Int }) postId: 숫자,
  @Args('comment', { type: () => Comment }) comment: CommentInput,
) {
  const newComment = this.commentsService.addComment({ id: postId, comment
});
  pubSub.publish('commentAdded', { commentAdded: newComment });
  return newComment;
}
```

PubSub#publish 메서드는 첫 번째 매개변수로 트리거 이름(이벤트 주제 이름이라고 생각하면 됩니다)을, 두 번째 매개변수로 이벤트 페이로드를 받습니다. 앞서 언급했듯이 구독은 정의에 따라 값을 반환하며 그 값은 모양을 갖습니다. 댓글 추가 구독에 대해 생성된 SDL을 다시 살펴보세요:

```
유형 구독 { commentAdded():
  코멘트!
}
```

이는 구독이 최상위 프로퍼티 이름이 commentAdded이고 값이 Comment 객체인 객체를 반환해야 한다는 것을 알려줍니다. 여기서 주의해야 할 중요한 점은 PubSub#publish 메서드가 내보내는 이벤트 페이로드의 모양이 구독에서 반환할 것으로 예상되는 값의 모양과 일치해야 한다는 것입니다. 따라서 위의 예제에서 pubSub.publish('commentAdded', {{ '{' }} commentAdded: newComment {{ '}' }}) 문은 적절한 모양의 페이로드가 포함된 commentAdded 이벤트를 게시합니다. 이러한 모양이 일치하지 않으면 GraphQL 유효성 검사 단계에서 구독이 실패합니다.

구독 필터링

특정 이벤트를 필터링하려면 필터 속성을 필터 함수로 설정합니다. 이 함수는 배열 필터에 전달된 함수와 유사하게 작동합니다. 이 함수에는 이벤트 페이로드가 포함된 페이로드(이벤트 게시자가 보낸 것)와 구독 요청 중에 전달된 인수를 받는 변수의 두 가지 인수가 필요합니다. 이 함수는 이 이벤트를 클라이언트 리스너에 게시할지 여부를 결정하는 부울을 반환합니다.

```
구독(반환 => 댓글, { 필터: (페이로드, 변수)
  =>)
    payload.commentAdded.title === variables.title,
})
commentAdded(@Args('title') title: string) {
  return pubSub.asyncIterator('commentAdded')를
  반환합니다;
}
```

구독 페이로드 변경

게시된 이벤트 페이로드를 변경하려면 `resolve` 속성을 함수로 설정합니다. 함수는 이벤트 게시자가 전송한 이벤트 페이로드를 수신하고 적절한 값을 반환합니다.

```
구독(반환 => 댓글, { 해결: 값 => 값,
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

> 경고 `resolve` 옵션을 사용하는 경우 언래핑된 페이로드를 반환해야 합니다(예: 이 예제에서는 `{{ '{' }} commentAdded: newComment {{ '}' }}` 객체를 반환해야 합니다.)

주입된 공급자에 액세스해야 하는 경우(예: 외부 서비스를 사용하여 데이터 유효성을 검사하는 경우) 다음 구성을 사용합니다.

```
구독(반환값 => 댓글, { resolve(this:
  AuthorResolver, value) {
    // "this"는 "AuthorResolver" 반환 값의 인스턴스를 참조합니
    다;
  }
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

필터에서도 동일한 구조가 작동합니다:

```
구독(반환 => 댓글, {
  filter(this: AuthorResolver, payload, variables) {
    // "this"는 "AuthorResolver"의 인스턴스를 참조합니다. 반환
    payload.commentAdded.title === variables.title;
  }
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

스키마 우선

Nest에서 이와 동등한 구독을 생성하기 위해 `@Subscription()` 데코레이터를 사용하겠습니다.

```
const pubSub = new PubSub();
```

```
@Resolver('Author')
내보내기 클래스 AuthorResolver {
  // ... 구독()
  commentAdded() {
    pubSub.asyncIterator('commentAdded')를 반환합니다;
  }
}
```

컨텍스트 및 인수를 기반으로 특정 이벤트를 필터링하려면 필터 속성을 설정합니다.

```
구독('commentAdded', { 필터: (페이로드,
  변수) =>)
    payload.commentAdded.title === variables.title,
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

게시된 페이로드를 변경하려면 해결 함수를 사용할 수 있습니다.

```
구독('commentAdded', { resolve: value
  => value,
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

삽입된 공급자에 액세스해야 하는 경우(예: 외부 서비스를 사용하여 데이터 유효성을 검사하는 경우) 다음 구성을 사용하세요:

```
구독('commentAdded', { resolve(this:
  AuthorResolver, value) {
    // "this"는 "AuthorResolver" 반환 값의 인스턴스를 참조합니
    다;
  }
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

필터에서도 동일한 구조가 작동합니다:

```
구독('commentAdded', {
  filter(this: AuthorResolver, payload, variables) {
```

```
    // "this"는 "AuthorResolver"의 인스턴스를 참조합니다. 반환
    payload.commentAdded.title === variables.title;
  }
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

마지막 단계는 유형 정의 파일을 업데이트하는 것입니다.

```
type Author {
  id: Int!
  이름: 성: 문자열입니다:
  문자열 글입니다: [포스트
  ]
}

유형 Post {
  id: Int!
  제목: 문자열 투표:
  Int
}

유형 쿼리 {
  author(id: Int!): 저자
}

유형 코멘트 { id:
  문자열 내용입니다:
  문자열
}

유형 구독 {
  commentAdded(title: 문자열!): 코멘트
}
```

이렇게 해서 하나의 `commentAdded(제목: 문자열!)`를 만들었습니다: `댓글` 구독을 생성했습니다. 전체 샘플 구현은 여기에서 확인할 수 있습니다.

PubSub

위에서 로컬 `PubSub` 인스턴스를 인스턴스화했습니다. 선호하는 접근 방식은 PubSub를 프로바이더로 정의하

고 생성자(`@Inject()` 데코레이터 사용)를 통해 인스턴스를 주입하는 것입니다. 이렇게 하면 전체 애플리케이션에서 인스턴스를 재사용할 수 있습니다. 예를 들어, 다음과 같이 프로바이더를 정의한 다음 필요한 곳에 `'PUB_SUB'`를 주입합니다.

```
{
  제공: 'PUB_SUB',
```

```
    사용값: 새로운 PubSub(),
  }
```

구독 서버 사용자 지정

구독 서버를 사용자 지정하려면(예: 경로 변경) `구독` 옵션 속성을 사용합니다.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  구독을 설정합니다: {
    'graphql-ws': { 경로:
      '/graphql'
    },
  }
}),
```

구독에 `graphql-ws` 패키지를 사용하는 경우 `구독-트랜스포트-ws를 구독-트랜스포트-ws로` 대체합니다.

키를 사용하여 다음과 같이 설정합니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  구독을 추가합니다: {
    'graphql-ws': {
      경로: '/graphql'
    },
  }
}),
```

웹소켓을 통한 인증

사용자가 인증되었는지 확인하는 것은 `구독` 옵션에서 지정할 수 있는 `onConnect` 콜백 함수 내에서 수행할 수 있습니다.

onConnect는 첫 번째 인자로 `연결` 매개변수인

`구독 클라이언트` (자세히 읽기).

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  구독을 설정합니다: {
    '구독-트랜스포트-ws': { onConnect:
      (connectionParams) => {
        const authToken = connectionParams.authToken;
        if (!isValid(authToken)) {
          새로운 오류('토큰이 유효하지 않습니다')를 던집니다;
        }
        // 토큰에서 사용자 정보 추출
```

```
          const user = parseToken(authToken);
          // 나중에 컨텍스트에 추가하기 위해 사용자 정보를 반환합니다 { 사용자
          }를 반환합니다;
        },
      }
    },
    컨텍스트: ({ 연결 }) => {
      // connection.context는 "onConnect" 콜백이 반환한 것과 동일합니다.
    },
  }),
```

이 예의 인증 `토큰은` 연결이 처음 설정될 때 클라이언트에 의해 한 번만 전송됩니다. 이 연결로 이루어진 모든 구독은 동일한 인증 토큰을 가지며, 따라서 동일한 사용자 정보를 갖게 됩니다.

> 경고 `구독-transport-ws에` 연결이 `onConnect` 단계를 건너뛸 수 있는 버그가 있습니다(자세한 내용 참조). 사용자가 구독을 시작할 때 `onConnect가` 호출되었다고 가정해서는 안 되며 항상 컨텍스트가 채워져 있는지 확인해야 합니다.

`graphql-ws` 패키지를 사용하는 경우, `onConnect` 콜백의 서명이 약간 달라집니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  구독을 추가합니다: {
    'graphql-ws': {
      onConnect: (context: Context<any>) => {
        const { connectionParams, extra } = 컨텍스트;
        // 사용자 유효성 검사는 위 예제와 동일하게 유지됩니다.
        // 그래프QL-WS와 함께 사용할 경우 추가 컨텍스트 값을 추가 필드에 저장해야 합니다.
        extra.user = { user: {} };
      },
    },
  },
  context: ({ extra }) => {
    // 이제 추가 필드를 통해 추가 컨텍스트 값에 액세스할 수 있습니다.
  },
});
```

Mercurius 드라이버로 구독 활성화하기

구독을 활성화하려면 `구독` 속성을 `true로` 설정합니다.

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  구독: true,
}),
```

> 정보 힌트 옵션 개체를 전달하여 사용자 지정 이미터를 설정하고 들어오는 연결의 유효성을 검사하는 등의 작업을 수행할 수도 있습니다. 여기에서 자세히 알아보세요(구독 참조).

코드 우선

코드 퍼스트 접근 방식을 사용하여 구독을 생성하려면 @Subscription() 데코레이터(@nestjs/graphql 패키지에서 내보낸 것)와 간단한 게시/구독 API를 제공하는 mercurius 패키지의 PubSub 클래스를 사용합니다.

다음 구독 핸들러는 PubSub#asyncIterator를 호출하여 이벤트 구독을 처리합니다. 이 메서드는 이벤트 주제 이름에 해당하는 단일 인자 triggerName을 받습니다.

```
@Resolver((of) => Author)
export class AuthorResolver {
  // ...
  구독((반환값) => 댓글) commentAdded(@Context('pubsub')
  pubSub: PubSub) {
    pubSub.subscribe('commentAdded')를 반환합니다;
  }
}
```

> 정보 힌트 위 예제에서 사용된 모든 데코레이터는 @nestjs/graphql에서 내보낸 것입니다. 패키지로 내보내는 반면 PubSub 클래스는 mercurius 패키지에서 내보냅니다.

> 경고 참고 PubSub는 간단한 게시 및 구독 API를 노출하는 클래스입니다. 사용자 지정 PubSub 클래스를 등록하는 방법은 이 섹션을 참조하세요.

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
유형 구독 { commentAdded():
  코멘트!
}
```

구독은 정의상 구독의 이름을 키로 하는 단일 최상위 프로퍼티를 가진 객체를 반환한다는 점에 유의하세요. 이 이름은 구독 핸들러 메서드의 이름에서 상속되거나(예: 위의 commentAdded), 아래 표시된 것처럼 키 이름을 두 번째 인수로 하는 옵션을 @Subscription() 데코레이터에 전달하여 명시적으로 제공될 수 있습니다.

```
구독(반환값 => 댓글, { 이름: '댓글 추가됨',
})
subscribeToCommentAdded(@Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded')를 반환합니다;
}
```

이 구성은 이전 코드 샘플과 동일한 SDL을 생성하지만 메서드 이름을 구독에서 분리할 수 있습니다.

게시

이제 이벤트를 게시하기 위해 `PubSub#publish` 메서드를 사용합니다. 이 메서드는 오브젝트 그래프의 일부가 변경되었을 때 클라이언트 측 업데이트를 트리거하기 위해 변이 내에서 자주 사용됩니다. 예를 들어

```
@@파일명(posts/posts.resolver)
@Mutation(returns => Post)
비동기 추가 코멘트(
  @Args('postId', { type: () => Int }) postId: 숫자,
  @Args('comment', { type: () => Comment }) comment: CommentInput,
  @Context('pubsub') pubSub: PubSub,
) {
  const newComment = this.commentsService.addComment({ id: postId, comment
});
  await pubSub.publish({
    topic: 'commentAdded',
    payload: {
      댓글 추가됨: 새로운 댓글
    }
  });
  새로운 댓글을 반환합니다;
}
```

앞서 언급했듯이 구독은 정의에 따라 값을 반환하며 그 값에는 모양이 있습니다. `댓글 추가` 구독에 대해 생성된 SDL을 다시 살펴보세요:

```
유형 구독 { commentAdded():
  코멘트!
}
```

이는 구독이 최상위 프로퍼티 이름이 commentAdded이고 값이 `Comment` 객체인 객체를 반환해야 한다는 것을 알려줍니다. 여기서 주의해야 할 중요한 점은 `PubSub#publish` 메서드가 내보내는 이벤트 페이로드의 모양이 구독에서 반환할 것으로 예상되는 값의 모양과 일치해야 한다는 것입니다. 따라서 위의 예제에서는 `pubSub.publish({{ '{' }} 주제: 'commentAdded', payload: {{ '{' }} commentAdded: newComment {{ '}' }} {{ '}' }})` 문은 적절한 모양의 페이로드가 포함된 `commentAdded` 이벤트를 게시합니다. 이러한 모양이 일치하지 않으면 GraphQL 유효성 검사 단계에서 구독이 실패합니다.

구독 필터링

특정 이벤트를 필터링하려면 `필터` 속성을 필터 함수로 설정합니다. 이 함수는 배열 `필터에` 전달된 함수와 유사하

게 작동합니다. 이 함수에는 이벤트 페이로드가 포함된 페이로드(이벤트 게시자가 보낸 대로)와 구독 요청 중에 전달된 인수를 받는 변수의 두 가지 인수가 필요합니다. 이 함수는 이 이벤트를 클라이언트 리스너에 게시할지 여부를 결정하는 부울을 반환합니다.

```
구독(반환 => 댓글, { 필터: (페이로드, 변수)
  =>)
    payload.commentAdded.title === variables.title,
})
commentAdded(@Args('title') title: 문자열, @Context('pubsub') pubSub:
PubSub) {
  pubSub.subscribe('commentAdded')를 반환합니다;
}
```

주입된 공급자에 액세스해야 하는 경우(예: 외부 서비스를 사용하여 데이터 유효성을 검사하는 경우) 다음 구성을 사용합니다.

```
구독(반환 => 댓글, {
  filter(this: AuthorResolver, payload, variables) {
    // "this"는 "AuthorResolver"의 인스턴스를 참조합니다. 반환
    payload.commentAdded.title === variables.title;
  }
})
commentAdded(@Args('title') title: 문자열, @Context('pubsub') pubSub:
PubSub) {
  pubSub.subscribe('commentAdded')를 반환합니다;
}
```

스키마 우선

Nest에서 이와 동등한 구독을 생성하기 위해 `@Subscription()` 데코레이터를 사용하겠습니다.

```
const pubSub = new PubSub();

@Resolver('Author')
내보내기 클래스 AuthorResolver {
  // ... 구독()
  commentAdded(@Context('pubsub') pubSub: PubSub) {
    return pubSub.subscribe('commentAdded')를 반환합니다;
  }
}
```

컨텍스트 및 인수를 기반으로 특정 이벤트를 필터링하려면 `필터` 속성을 설정합니다.

```
구독('commentAdded', { 필터: (페이로드,
  변수) =>)
    payload.commentAdded.title === variables.title,
})
commentAdded(@Context('pubsub') pubSub: PubSub) {
```

```
    pubSub.subscribe('commentAdded')를 반환합니다;
  }
```

주입된 공급자에 액세스해야 하는 경우(예: 외부 서비스를 사용하여 데이터 유효성을 검사하는 경우) 다음 구성을 사용합니다:

```
구독('commentAdded', {
  filter(this: AuthorResolver, payload, variables) {
    // "this"는 "AuthorResolver"의 인스턴스를 참조합니다. 반환
    payload.commentAdded.title === variables.title;
  }
})
commentAdded(@Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded')를 반환합니다;
}
```

마지막 단계는 유형 정의 파일을 업데이트하는 것입니다.

```
type Author {
  id: Int!
  이름: 성: 문자열입니다:
  문자열 글입니다: [포스트
  ]
}

유형 Post {
  id: Int!
  제목: 문자열 투표:
  Int
}

유형 쿼리 {
  author(id: Int!): 저자
}

유형 Comment {
  id: 문자열 내용입
  니다: 문자열
}

유형 구독 {
  commentAdded(title: 문자열!): 코멘트
}
```

이렇게 해서 하나의 `commentAdded(제목: 문자열!)`를 만들었습니다: `댓글` 구독을 생성합니다.

PubSub

위의 예에서는 기본 `PubSub` 이미터(mqemitter)를 사용했습니다. (프로덕션의 경우) 선호하는 접근 방식은 `mqemitter-redis를` 사용하는 것입니다. 또는 사용자 정의 `PubSub` 구현을 제공할 수도 있습니다(자세한 내용은 여기를 참조하세요).

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  구독입니다: {
    emitter: require('mqemitter-redis')({
      port: 6579,
      호스트: '127.0.0.1',
    }),
  },
});
```

웹소켓을 통한 인증

사용자가 인증되었는지 확인하는 것은 `구독` 옵션에서 지정할 수 있는 `verifyClient` 콜백 함수 내에서 수행할 수 있습니다.

`verifyClient는` 요청의 헤더를 검색하는 데 사용할 수 있는 `정보` 객체를 첫 번째 인수로 받습니다.

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  구독입니다: {
    verifyClient: (정보, 다음) => {
      const authorization = info.req.headers?.authorization as string;
      if (!authorization?.startsWith('Bearer ')) {
        반환 다음(거짓);
      }
      next(true);
    },
  }
}),
```

스칼라

GraphQL 객체 유형에는 이름과 필드가 있지만, 어느 시점에서는 이러한 필드가 구체적인 데이터로 해석되어야 합니다. 스칼라 유형이 바로 쿼리의 리프를 나타내는 역할을 합니다(자세한 내용은 여기를 참조하세요). GraphQL에는 다음과 같은 기본 유형이 포함되어 있습니다: `Int`, `Float`, `String`, `Boolean` 및 `ID`. 이러한 기본 제공 유형 외에도 사용자 정의 원자 데이터 유형(예: `날짜`)을 지원해야 할 수도 있습니다.

코드 우선

코드 우선 접근 방식은 5개의 스칼라를 제공하며, 이 중 3개는 기존 GraphQL 유형의 간단한 별칭입니다.

- `ID`(`GraphQLID의` 별칭) - 객체를 리페치하거나 캐시의 키로 자주 사용되는 고유 식별자를 나타냅니다.
- `Int`(`GraphQLInt의` 별칭) - 부호화된 32비트 정수
- `부동` 소수점(`GraphQLFloat의` 별칭) - 부호화된 배정밀도 부동 소수점 값
- `GraphQLISODateTime` - UTC 기준 날짜-시간 문자열(기본적으로 `날짜` 유형을 나타내는 데 사용됨)
- `그래프QL타임스탬프` - 날짜와 시간을 UNIX 시대 시작부터 밀리초 수로 나타내는 부호화된 정수입니다.

`날짜` 유형을 나타내기 위해 기본적으로 `GraphQLISODateTime`(예: `2019-12-03T09:54:33Z`)이 사용됩니다. 대신 `GraphQL타임스탬프를` 사용하려면 다음과 같이 `빌드스케마옵션` 객체의 `dateScalarMode를` `'timestamp'`로 설정합니다:

```
GraphQLModule.forRoot({
  buildSchemaOptions: {
    dateScalarMode: '타임스탬프',
  }
}),
```

마찬가지로 `숫자` 유형을 나타내는 데 기본적으로 GraphQLFloat가 사용됩니다. `GraphQLInt를` 사용하려면 대신 다음과 같이 `buildSchemaOptions` 객체의 `numberScalarMode를` `'정수'`로 설정합니다:

```
GraphQLModule.forRoot({
  buildSchemaOptions: {
    numberScalarMode: '정수',
  }
}),
```

또한 사용자 정의 스칼라를 만들 수도 있습니

다. 기본 스칼라 재정의

`Date` 스칼라에 대한 사용자 정의 구현을 만들려면 새 클래스를 만들면 됩니다.

```
'@nestjs/graphql'에서 { Scalar, CustomScalar }를 임포트하고
, 'graphql'에서 { Kind, ValueNode }를 임포트합니다;

@Scalar('Date', (type) => Date)
export class DateScalar 구현 CustomScalar<number, Date> {
  description = '날짜 사용자 정의 스칼라 유형';

  parseValue(value: 숫자): Date {
    반환 새로운 날짜(값); // 클라이언트로부터의 값
  }

  serialize(value: Date): number {
    반환 value.getTime(); // 클라이언트로 전송된 값
  }

  parseLiteral(ast: ValueNode): Date {
    if (ast.kind === Kind.INT) {
      새로운 Date(ast.value)를 반환합니다;
    }
    널을 반환합니다;
  }
}
```

이 준비가 완료되면 `DateScalar를` 공급자로 등록합니다.

```
모듈({
  공급자: [날짜스칼라],
})
내보내기 클래스 CommonModule {}
```

이제 클래스에서 `날짜` 유형을 사용할 수 있습니다.

```
필드() 생성Date: 날짜;
```

사용자 정의 스칼라 가져오기

사용자 정의 스칼라를 사용하려면 해당 스칼라를 가져와 리졸버로 등록합니다. 데모 목적으로 `graphql-type-json` 패키지를 사용하겠습니다. 이 npm 패키지는 `JSON` GraphQL 스칼라 유형을 정의합니다.

패키지를 설치하는 것으로 시작하세요:

```
$ npm i --save graphql-type-json
```

패키지가 설치되면 사용자 정의 리졸버를 `forRoot()` 메서드에 전달합니다:

```
'graphql-type-json'에서 GraphQLJSON import;

@Module({
  임포트합니다: [
    GraphQLModule.forRoot({
      리졸버: { JSON: GraphQLJSON },
    }),
  ],
})
내보내기 클래스 AppModule {}
```

이제 클래스에서 `JSON` 유형을 사용할 수 있습니다.

```
필드((유형) => GraphQLJSON) 정보:
JSON;
```

유용한 스칼라를 보려면 graphql-scalars 패키지를 살펴보세요. 사용자 지정 스

칼라 만들기

사용자 정의 스칼라를 정의하려면 새 `GraphQLScalarType` 인스턴스를 만듭니다. 사용자 정의 `UUID` 스칼라를 생성하겠습니다.

```
const regex = /^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-
[0-9a-f]
{12}$/i;

함수 validate(uuid: unknown): string | never {
  if (typeof uuid !== "string" || !regex.test(uuid)) {
    throw new Error("유효하지 않은 uuid");
  }
  반환 UUID;
}

export const CustomUuidScalar = new GraphQLScalarType({
  name: 'UUID',
  description: '간단한 UUID 구문 분석기',
  serialize: (value) => validate(value),
  parseValue: (value) => validate(value),
  parseLiteral: (ast) => validate(ast.value)
})
```

사용자 정의 리졸버를 `forRoot()` 메서드에 전달합니다:

```
모듈({ import:
  [
    GraphQLModule.forRoot({
      리졸버: { UUID: CustomUuidScalar },
```

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

이제 클래스에서 `UUID` 유형을 사용할 수 있습니다.

```
@Field((type) => CustomUuidScalar)
uuid: 문자열;
```

스키마 우선

사용자 정의 스칼라를 정의하려면(스칼라에 대한 자세한 내용은 여기를 참조하세요) 유형 정의와 전용 리졸버를 생성합니다. 여기서는 (공식 문서에서와 마찬가지로) 데모 목적으로 `graphql-type-json` 패키지를 사용하겠습니다. 이 npm 패키지는 `JSON` GraphQL 스칼라 타입을 정의합니다.

패키지를 설치하는 것으로 시작하세요:

```
$ npm i --save graphql-type-json
```

패키지가 설치되면 사용자 정의 리졸버를 `forRoot()` 메서드에 전달합니다:

```
'graphql-type-json'에서 GraphQLJSON import;

@Module({
  임포트합니다: [
    GraphQLModule.forRoot({
      typePaths: ['./**/*.graphql'], 리졸버:
      { JSON: GraphQLJSON },
    }),
  ],
})
내보내기 클래스 AppModule {}
```

이제 유형 정의에 `JSON` 스칼라를 사용할 수 있습니다:

```
스칼라 JSON 유

형 Foo {
  필드입니다: JSON
}
```

스칼라 유형을 정의하는 또 다른 방법은 간단한 클래스를 만드는 것입니다. `Date` 타입으로 스키마를 개선하고 싶다고 가정해 보겠습니다.

```
'@nestjs/graphql'에서 { Scalar, CustomScalar }를 임포트하고
, 'graphql'에서 { Kind, ValueNode }를 임포트합니다;

@Scalar('Date')
export class DateScalar 구현 CustomScalar<number, Date> {
  description = '날짜 사용자 정의 스칼라 유형';

  parseValue(value: 숫자): Date {
    반환 새로운 날짜(값); // 클라이언트로부터의 값
  }

  serialize(value: Date): number {
    반환 value.getTime(); // 클라이언트로 전송된 값
  }

  parseLiteral(ast: ValueNode): Date {
    if (ast.kind === Kind.INT) {
      새로운 Date(ast.value)를 반환합니다;
    }
    널을 반환합니다;
  }
}
```

이 준비가 완료되면 `DateScalar`를 공급자로 등록합니다.

```
모듈({
  공급자: [날짜스칼라],
})
내보내기 클래스 CommonModule {}
```

이제 유형 정의에서 `Date` 스칼라를 사용할 수 있습니다.

```
스칼라 날짜
```

기본적으로 모든 스칼라에 대해 생성되는 타입스크립트 정의는 `임의의` 것으로, 특별히 타입 안전하지 않습니다. 그러나 유형 생성 방법을 지정할 때 Nest가 사용자 정의 스칼라에 대한 유형을 생성하는 방법을 구성할 수 있습니다:

```
'@nestjs/graphql'에서 { GraphQLDefinitionsFactory }를 가져오고,
'path'에서 { join }를 가져옵니다;

const definitionsFactory = 새로운 GraphQLDefinitionsFactory();
```

```
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  경로: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
  기본 스칼라 타입: '알 수 없음',
  customScalarTypeMapping: {
    DateTime: 'Date',
    BigNumber: '_BigNumber',
  },
  추가 헤더: "'bignumber.js'에서 _BigNumber 가져오기",
});
```

> 정보 힌트 또는 유형 참조를 대신 사용할 수도 있습니다: `DateTime: Date`. 이 경우 `GraphQLDefinitionsFactory`는 지정된 유형(`Date.name`)의 이름 속성을 추출하여 TS 정의를 생성합니다. 참고: 기본 제공 유형이 아닌 유형(사용자 정의 유형)에 대해서는 가져오기 문을 추가해야 합니다.

이제 다음과 같은 GraphQL 사용자 정의 스칼라 유형이 주어집니다:

```
스칼라 날짜/시간 스칼
라 빅넘버 스칼라 페이
로드
```

이제 다음과 같이 생성된 타입스크립트 정의가 `src/graphql.ts`에 표시됩니다:

```
'bignumber.js'에서 _BigNumber를 임포트하고,
내보내기 유형 DateTime = 날짜를 내보냅니다;
내보내기 유형 BigNumber = _BigNumber; 내보내
기 유형 페이로드 = 알 수 없음;
```

여기서는 `customScalarTypeMapping` 프로퍼티를 사용하여 사용자 정의 스칼라에 대해 선언하려는 유형의 맵을 제공했습니다. 또한 이러한 유형 정의에 필요한 임포트를 추가할 수 있도록 `additionalHeader` 프로퍼티를 제공했습니다. 마지막으로, `customScalarTypeMapping`에 지정되지 않은 모든 사용자 정의 스칼라가 `아무 것도` 아닌 알 수 `없음으로 별칭이` 지정되도록 `defaultScalarType`을 `'unknown'`으로 추가했습니다 (TypeScript는 유형 안전성 강화를 위해 3.0부터 사용을 권장하고 있습니다).

> 정보 힌트 순환형 참조를 피하기 위해 `bignumber.js`에서 `_BigNumber`를 가져왔음을 참고하세요.

지시어

지시어는 필드 또는 조각 포함에 첨부할 수 있으며 서버가 원하는 방식으로 쿼리 실행에 영향을 줄 수 있습니다(자세한 내용은 여기를 참조하세요). GraphQL 사양은 몇 가지 기본 지시어를 제공합니다:

- `include(if: 부울)` - 인수가 참인 경우에만 이 필드를 결과에 포함시킵니다.
- `@skip(if: 부울)` - 인수가 참이면 이 필드를 건너뜁니다.
- `사용 중단됨(이유: 문자열)` - 메시지와 함께 필드를 사용 중단된 것으로 표시합니다.

지시어는 `@` 문자가 앞에 오는 식별자이며, 선택적으로 GraphQL 쿼리 및 스키마 언어의 거의 모든 요소 뒤에 나타날 수 있는 명명된 인수 목록이 뒤따릅니다.

사용자 지정 지시문

아폴로/머큐리우스가 지시문을 만나면 어떤 일이 일어나야 하는지 지시하려면 트랜스포머 함수를 만들 수 있습니다. 이 함수는 `mapSchema` 함수를 사용하여 스키마의 위치(필드 정의, 유형 정의 등)를 반복하고 해당 변환을 수행합니다.

```
'@graphql- tools/utils'에서 { getDirective, MapperKind, mapSchema }
를 가져옵니다;
'graphql'에서 { defaultFieldResolver, GraphQLSchema }를 가져옵니다;

내보내기 함수 upperDirectiveTransformer( 스키마:
  GraphQLSchema,
  지시어 이름: 문자열,
) {
  return mapSchema(schema, {
    [MapperKind.OBJECT_FIELD]: (fieldConfig) => {
      const upperDirective = getDirective(
        스키마,
        fieldConfig, 지시어
        이름,
      )?.[0];

      if (upperDirective) {
        const { resolve = defaultFieldResolver } = fieldConfig;

        // 원래 리졸버를 *먼저* 다음과 같은 함수로 바꿉니다.
통화
        // 원래 리졸버를 호출한 다음 결과를 대문자로 변환합니다. fieldConfig.resolve =
        async 함수(source, args, context, info).
{
          const result = await resolve(source, args, context, info);
          if (typeof result === 'string') {
            결과값을 반환합니다;
          }
          결과를 반환합니다;
        };
        필드 컨피그를 반환합니다;
      }
    },
```

```
  });
}
```

이제 `GraphQLModule#forRoot에서 상위 지시어 트랜스포머` 변환 함수를 적용합니다.

메서드를 사용하여 `변환 스키마` 함수를 사용합니다:

```
GraphQLModule.forRoot({
  // ...
  transformSchema: (스키마) => upperDirectiveTransformer(스키마, 'upper'),
});
```

등록이 완료되면 `@upper` 지시문을 스키마에서 사용할 수 있습니다. 그러나 지시문을 적용하는 방식은 사용하는 접근 방식(코드 우선 또는 스키마 우선)에 따라 달라집니다.

코드 우선

코드 우선 접근 방식에서는 `@Directive()` 데코레이터를 사용하여 지시문을 적용합니다.

```
@디렉티브('@어퍼') @필드()
제목: 문자열;
```

> 정보 힌트 `@Directive()` 데코레이터는 `@nestjs/graphql` 패키지에서 내보냅니다.

지시어는 필드, 필드 해석기, 입력 및 객체 유형은 물론 쿼리, 변이 및 구독에도 적용할 수 있습니다. 다음은 쿼리 핸들러 수준에서 적용된 지시문의 예입니다:

```
지시어('@deprecated(reason: "이 쿼리는 다음 버전에서 제거될 예정입니다.")')
쿼리(반환값 => 작성자, { 이름: '작성자' })
async getAuthor(@Args({ name: 'id', type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

> 경고 `@Directive()` 데코레이터를 통해 적용된 지시문은 생성된 스키마 정의 파일에 반영되지 않습니다.

마지막으로 `GraphQLModule에서` 다음과 같이 지시문을 선언해야 합니다:

```
GraphQLModule.forRoot({
  // ...,
  transformSchema: schema => upperDirectiveTransformer(schema, 'upper'),
  buildSchemaOptions: {
    지시어: [
```

```
        새로운 그래프QL디렉티브({ 이
          름: 'upper',
          위치: [DirectiveLocation.FIELD_DEFINITION],
        }),
      ],
    },
  }),
```

> 정보 힌트 그래프QL디렉티브와 디렉티브 위치는 모두 그래프큐엘에서 내보낸다.
>
> 패키지입니다.

스키마 우선

스키마 우선 접근 방식에서는 SDL에서 직접 지시문을 적용합니다.

```
지시문 @upper는 FIELD_DEFINITION 유형

Post {
  id: Int!
  제목: String! 상위 투표:
  Int
}
```

인터페이스

많은 타입 시스템과 마찬가지로 GraphQL은 인터페이스를 지원합니다. 인터페이스는 유형이 인터페이스를 구현하기 위해 포함해야 하는 특정 필드 집합을 포함하는 추상 유형입니다(자세한 내용은 여기를 참조하세요).

코드 우선

코드 우선 접근 방식을 사용하는 경우, `@nestjs/graphql에서` 내보낸 `@InterfaceType()` 데코레이터로 주석이 달린 추상 클래스를 생성하여 GraphQL 인터페이스를 정의합니다.

```
'@nestjs/graphql'에서 { 필드, ID, 인터페이스 유형 } 임포트; @인터페이스 유형

()
내보내기 추상 클래스 Character {
  @Field((type) => ID)
  ID: 문자열;

  @Field() 이름:

  문자열;
}
```

> 경고 경고 TypeScript 인터페이스는 GraphQL 인터페이스를 정의하는 데 사용할 수 없습니다.

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
인터페이스 Character {
  id: ID!
  이름: 문자열!
}
```

이제 `캐릭터` 인터페이스를 구현하려면 `구현` 키를 사용합니다:

```
객체 유형({
  구현합니다: () => [문자],
})
내보내기 클래스 Human 구현 문자 { id: 문자열;
  이름: 문자열;
}
```

> 정보 힌트 `@ObjectType()` 데코레이터는 `@nestjs/graphql` 패키지에서 내보냅니다.

라이브러리에서 생성된 기본 `resolveType()` 함수는 리졸버 메서드에서 반환된 값을 기반으로 유형을 추출합니다. 즉, 클래스 인스턴스를 반환해야 합니다(리터럴 JavaScript 객체를 반환할 수 없음).

사용자 정의된 `resolveType()` 함수를 제공하려면 다음과 같이 `@InterfaceType()` 데코레이터에 전달된 옵션 객체에 `resolveType` 속성을 전달합니다:

```
@InterfaceType({
  resolveType(book) {
    if (book.colors) { 컬러
      링북을 반환합니다;
    }
    교과서를 반환합니다;
  },
})
내보내기 추상 클래스 Book { @Field((type)
  => ID)
  ID: 문자열;

  @Field()
  title: 문자열;
}
```

인터페이스 리졸버

지금까지는 인터페이스를 사용하여 필드 정의만 객체와 공유할 수 있었습니다. 실제 필드 리졸버 구현도 공유하려면 다음과 같이 전용 인터페이스 리졸버를 생성하면 됩니다:

```
'@nestjs/graphql'에서 { Resolver, ResolveField, Parent, Info }를 가져옵니다;

@Resolver(type => Character) // 알림: Character는 인터페이스 내보내기 클래스
CharacterInterfaceResolver {
  @ResolveField(() => [문자]) 친구(
    부모() 문자, // 문자를 구현하는 확인된 객체 @정보() { 부모 유형 }, // 구현하는 객
    체의 유형
캐릭터
    @Args('search', { type: () => String }) searchTerm: 문자열,
  ) {
    // 캐릭터의 친구 반환 []을 가져옵
    니다;
  }
}
```

이제 `친구` 필드 리졸버는 `Character를` 구현하는 모든 객체 유형에 대해 자동으로 등록됩니다.

인터페이스

입니다. 스키

마 우선

스키마 우선 접근 방식으로 인터페이스를 정의하려면 SDL로 GraphQL 인터페이스를 생성하기만 하면 됩니다.

```
인터페이스 Character {
  id: ID!
  이름: 문자열!
}
```

그런 다음 빠른 시작 챕터에 표시된 대로 타이핑 생성 기능을 사용하여 해당 타입스크립트 정의를 생성할 수 있습니다:

```
내보내기 인터페이스 Character {
  id: 문자열;
  이름: 문자열;
}
```

인터페이스는 리졸버 맵에 추가 resolveType 필드를 추가하여 인터페이스가 어떤 유형으로 해석할지 결정해야 합니다. `CharactersResolver` 클래스를 생성하고 다음과 같이 정의해 보겠습니다. `resolveType` 메서드를 정의해 보겠습니다:

```
@Resolver('Character')
내보내기 클래스 CharactersResolver {
  @ResolveField()
  __resolveType(value) {
    if ('age' in value) {
      사람을 반환합니다;
    }
    널을 반환합니다;
  }
}
```

> 정보 힌트 모든 데코레이터는 `@nestjs/graphql` 패키지에서 내보냅니다.

유니온

유니온 유형은 인터페이스와 매우 유사하지만 유형 간에 공통 필드를 지정할 수 없습니다(자세한 내용은 여기를 참조하세요). 유니온은 단일 필드에서 분리된 데이터 유형을 반환하는 데 유용합니다.

코드 우선

GraphQL 유니온 유형을 정의하려면 이 유니온이 구성될 클래스를 정의해야 합니다. Apollo 설명서의 예제에 따라 두 개의 클래스를 만들겠습니다. 첫째, `Book`:

```
'@nestjs/graphql'에서 { Field, ObjectType } 임포트; @ObjectType()
내보내기 클래스 Book {
  @Field()
  제목: 문자열;
}
```

그런 다음 작성합니다:

```
'@nestjs/graphql'에서 { Field, ObjectType } 임포트; @ObjectType()
내보내기 클래스 Author {
  @Field()
  이름: 문자열;
}
```

이 준비가 완료되면 `@nestjs/graphql` 패키지에서 내보낸 `createUnionType` 함수를 사용하여 `ResultUnion` 유니온을 등록합니다:

```
export const ResultUnion = createUnionType({
  name: 'ResultUnion',
  유형: () => [저자, 책] const,
});
```

> 경고 `createUnionType` 함수의 `types` 프로퍼티가 반환하는 배열에 const 어설션을 지정해야 합니다. const 어설션을 제공하지 않으면 컴파일 시 잘못된 선언 파일이 생성되어 다른 프로젝트에서 사용할 때 오류가 발생합니다.

```
@Query(returns => [ResultUnion])
const enum AllowedTypes = Result.types {
```

이제 쿼리에서 `ResultUnion을` 참조할 수 있습니다:

```
    반환 [새로운 저자(), 새로운 책()];
  }
```

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
유형 저자 { 이름: 문
  자열!
}

유형 Book {
  title: 문자열!
}
유니온 ResultUnion = 저자 | 도서 유형 쿼

리 {
  검색: [ResultUnion!]!
}
```

라이브러리에서 생성된 기본 `resolveType()` 함수는 리졸버 메서드에서 반환된 값을 기반으로 유형을 추출합니다. 즉, 리터럴 JavaScript 객체 대신 클래스 인스턴스를 반환해야 합니다.

사용자 정의된 `resolveType()` 함수를 제공하려면 다음과 같이 `createUnionType()` 함수에 전달된 옵션 객체에 `resolveType` 속성을 전달합니다:

```
export const ResultUnion = createUnionType({
  name: 'ResultUnion',
  types: () => [Author, Book] as const,
  resolveType(value) {
    if (value.name) {
      Author를 반환합니다;
    }
    if (value.title) { 반환
      책;
    }
    널을 반환합니다;
  },
});
```

스키마 우선

```
유형 저자 { 이름: 문
  자열!
```

스키마 우선 접근 방식에서 유니온을 정의하려면 SDL을 사용하여 GraphQL 유니온을 생성하기만 하면 됩니다.

```
}

유형 Book {
  title: 문자열!
}

유니온 ResultUnion = 저자 | 책
```

그런 다음 빠른 시작 챕터에 표시된 대로 타이핑 생성 기능을 사용하여 해당 타입스크립트 정의를 생성할 수 있습니다:

```
내보내기 클래스 Author { 이름
  : 문자열;
}

내보내기 클래스 Book { 제
  목: 문자열;
}
```

```
내보내기 유형 ResultUnion = 저자 | 책;
```

유니온에는 리졸버 맵에 추가 _____resolveType 필드를 추가하여 유니온이 어떤 유형으로 해석할지 결정해야 합니다. 또한 `ResultUnionResolver` 클래스는 모든 모듈에서 프로바이더로 등록해야 합니다. `ResultUnionResolver` 클래스를 생성하고 `__resolveType` 메서드를 정의해 보겠습니다.

```
@Resolver('ResultUnion')
export 클래스 ResultUnionResolver {
  @ResolveField()
  __resolveType(value) {
    if (value.name) {
      '저자'를 반환합니다;
    }
    if (value.title) { '
      책'을 반환합니다;
    }
    널을 반환합니다;
  }
}
```

> 정보 힌트 모든 데코레이터는 `@nestjs/graphql` 패키지에서 내보냅니다.

열거형

열거형은 허용되는 특정 값 집합으로 제한되는 특별한 종류의 스칼라입니다(자세한 내용은 여기를 참조하세요).

이를 통해 다음을 수행할 수 있습니다:

- 이 유형의 인수가 허용된 값 중 하나인지 확인합니다.

- 필드가 항상 유한한 값 집합 중 하나라는 것을 유형 시스템을 통해 전달합니다. 먼저 코드화합니다

.

코드 우선 접근 방식을 사용하는 경우, 간단히 TypeScript를 생성하여 GraphQL 열거형 유형을 정의합니다.
열거형.

```
export enum AllowedColor {
  RED,
  녹색,
  파란색,
}
```

이 작업을 완료한 후 `@nestjs/graphql` 패키지에서 내보낸 `registerEnumType` 함수를 사용하여 `AllowedColor` 열거형을 등록합니다:

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
});
```

이제 유형에서 `허용된 색상을` 참조할 수 있습니다:

```
@Field(유형 => 허용된 색상)
favoriteColor: 허용된 색상;
```

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
열거형 허용된 색상 { 빨간
  색
  녹색
  파란색
}
```

열거형에 대한 설명을 제공하려면 `설명` 프로퍼티를 `registerEnumType()`에 전달합니다.
함수입니다.

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
  설명: '지원되는 색상입니다.',
});
```

열거형 값에 대한 설명을 제공하거나 값을 더 이상 사용되지 않는 것으로 표시하려면 값맵을 전달합니다.

속성을 다음과 같이 변경합니다:

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
  설명: '지원되는 색상입니다.', valuesMap: {
    RED: {
      설명: '기본 색상입니다.',
    },
    BLUE: {
      deprecationReason: '너무 파랗다.',
    },
  },
});
```

이렇게 하면 SDL에 다음과 같은 GraphQL 스키마가 생성됩니다:

```
"""
지원되는 색상입니다. """
enum AllowedColor {
  """
  기본 색상입니다. """
  레드
  그린
  BLUE @deprecated(이유: "너무 파란색입니다.")
}
```

스키마 우선

스키마 퍼스트 접근 방식에서 열거자를 정의하려면 SDL로 GraphQL 열거자를 생성하기만 하면 됩니다.

```
열거형 허용된 색상 { 빨간
  색
  녹색
  파란색
}
```

그런 다음 빠른 시작 챕터에 표시된 대로 타이핑 생성 기능을 사용하여 해당 타입스크립트 정의를 생성할 수 있습

니다:

```
내보내기 열거형 허용된 색상 { 빨
    간색
```

```
  녹색

  파란색
}
```

때때로 백엔드에서 공개 API와 내부적으로 열거형에 대해 다른 값을 강제하는 경우가 있습니다. 이 예제에서는 API에 `RED가` 포함되어 있지만, 리졸버에서는 `#f00을` 대신 사용할 수 있습니다(자세한 내용은 여기를 참조하세요). 이렇게 하려면 `AllowedColor` 열거형에 대한 리졸버 객체를 선언합니다:

```
내보내기 const allowedColorResolver: 레코드<허용된 색의 유형 키, any>
= {
  빨간색: '#f00',
};
```

> 정보 힌트 모든 데코레이터는 `@nestjs/graphql` 패키지에서 내보냅니다.

그런 다음 이 리졸버 객체를 `GraphQLModule#forRoot()`의 `리졸버` 프로퍼티와 함께 사용합니다.

메서드에 대해 다음과 같이 설명합니다:

```
GraphQLModule.forRoot({
  resolvers: {
    허용된 색상: 허용된 색상 해결기,
  },
});
```

필드 미들웨어

> 경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

필드 미들웨어를 사용하면 필드가 확인되기 전이나 후에 임의의 코드를 실행할 수 있습니다. 필드 미들웨어를 사용하여 필드의 결과를 변환하거나, 필드의 인수를 검증하거나, 필드 수준 역할을 확인할 수도 있습니다(예: 미들웨어 함수가 실행되는 대상 필드에 액세스하는 데 필요한 경우).

여러 미들웨어 함수를 필드에 연결할 수 있습니다. 이 경우 이전 미들웨어가 다음 미들웨어를 호출하기로 결정한 체인을 따라 순차적으로 호출됩니다. `미들웨어` 배열에 있는 미들웨어 함수의 순서가 중요합니다. 첫 번째 리졸버는 "가장 바깥쪽" 레이어이므로 가장 먼저 그리고 마지막으로 실행됩니다(`그래프 쿼리 미들웨어` 패키지와 유사하게). 두 번째 리졸버는 "두 번째 외부" 계층이므로 두 번째로 실행되고 두 번째에서 마지막으로 실행됩니다.

시작하기

필드 값을 클라이언트로 다시 보내기 전에 기록하는 간단한 미들웨어를 만드는 것부터 시작해 보겠습니다:

```
'@nestjs/graphql'에서 { FieldMiddleware, MiddlewareContext, NextFn
}을 가져옵니다;

const loggerMiddleware: FieldMiddleware = async (
  ctx: MiddlewareContext,
  다음: 다음Fn,
) => {
  const value = await next();
  console.log(value);
  반환 값입니다;
};
```

> 정보 힌트 MiddlewareContext는 GraphQL 리졸버 함수가 일반적으로 수신하는 것과 동일한 인수(`{{ '{' }} source, args, context, info {{ '}' }}`)로 구성된 객체이며, NextFn은 스택의 다음 미들웨어(이 필드에 바인딩된) 또는 실제 필드 리졸버를 실행할 수 있게 해주는 함수입니다.

> 경고 필드 미들웨어 함수는 매우 가볍게 설계되었으며 데이터베이스에서 데이터를 검색하는 등 잠재적으로 시간이 많이 소요되는 작업을 수행해서는 안 되므로 종속성을 주입하거나 Nest의 DI 컨테이너에 액세스할 수 없습니다. 데이터 소스에서 외부 서비스를 호출하거나 데이터를 쿼리해야 하는 경우, 루트 쿼리/변이 핸들러에 바인딩된 가드/인터셉터에서 이를 수행하고 필드 미들웨어 내에서 액세스할 수 있는 `컨텍스트` 객체에 할당해야 합니다(특히, `MiddlewareContext` 객체에서).

필드 미들웨어는 `필드 미들웨어` 인터페이스와 일치해야 합니다. 위의 예제에서는 먼저 실제 필드 리졸버를 실행하고 필드 값을 반환하는 `next()` 함수를 실행한 다음, 이 값을 터미널에 로깅합니다. 또한 미들웨어 함수에서 반환된 값은 이전 값을 완전히 재정의하므로 변경을 수행하고 싶지 않으므로 원래 값을 반환하기만 하면 됩니다.

이렇게 하면 다음과 같이 `@Field()` 데코레이터에 미들웨어를 직접 등록할 수 있습니다:

```
객체 유형()
내보내기 클래스 레시피 {
  필드({ 미들웨어: [loggerMiddleware] }) 제목: 문
  자열;
}
```

이제 `레시피` 개체 유형의 `제목` 필드를 요청할 때마다 원래 필드 값이 콘솔에 기록됩니다.

> 정보 힌트 확장 기능을 사용하여 필드 수준 권한 시스템을 구현하는 방법을 알아보려면 이 섹션을 참조하세요.

> 경고 필드 미들웨어는 `ObjectType` 클래스에만 적용할 수 있습니다. 자세한 내용은 이 이슈를 확인하세요.

또한 위에서 언급했듯이 미들웨어 함수 내에서 필드 값을 제어할 수 있습니다. 데모를 위해 레시피의 제목(있는 경우)을 대문자로 표시해 보겠습니다:

```
const value = await next();
return value?.toUpperCase();
```

이 경우 요청 시 모든 제목이 자동으로 대문자로 바뀝니다.

마찬가지로 필드 미들웨어를 사용자 지정 필드 리졸버에 바인딩할 수 있습니다(`@ResolveField()` 데코레이터)를 다음과 같이 변경합니다:

```
@ResolveField(() => String, { middleware: [loggerMiddleware] })
title() {
  '자리 표시자'를 반환합니다;
}
```

> 경고 필드 리졸버 수준에서 인핸서가 활성화된 경우(자세히 읽기) 필드 미들웨어 기능은 메서드에 바인딩된 인터셉터, 가드 등보다 먼저 실행됩니다(쿼리 또는 변이 처리기에 등록된 루트 수준 인핸서 이후).

글로벌 필드 미들웨어

미들웨어를 특정 필드에 직접 바인딩하는 것 외에도 하나 또는 여러 개의 미들웨어 함수를 전역적으로 등록할 수도 있습니다. 이 경우 개체 유형의 모든 필드에 자동으로 연결됩니다.

```
GraphQLModule.forRoot({
  autoSchemaFile: 'schema.gql',
  buildSchemaOptions: {
    fieldMiddleware: [loggerMiddleware],
```

```
    },
  }),
```

> 정보 힌트 전역으로 등록된 필드 미들웨어 함수는 로컬로 등록된 함수(특정 필드에 직접 바인딩된 함수)보다 먼저 실행됩니다.

매핑된 유형

> 경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

CRUD(만들기/읽기/업데이트/삭제)와 같은 기능을 구축할 때 기본 엔티티 유형에서 변형을 구성하는 것이 유용할 때가 많습니다. Nest는 유형 변환을 수행하는 여러 유틸리티 함수를 제공하여 이 작업을 더 편리하게 만듭니다.

부분

입력 유효성 검사 유형(데이터 전송 개체 또는 DTO라고도 함)을 구축할 때 동일한 유형에 대해 만들기 및 업데이트 변형을 구축하는 것이 유용한 경우가 많습니다. 예를 들어, 생성 변형은 모든 필드를 필수로 설정하고 업데이트 변형은 모든 필드를 선택 사항으로 설정할 수 있습니다.

Nest는 이 작업을 더 쉽게 수행하고 상용구를 최소화하기 위해 `PartialType()` 유틸리티 함수를 제공합니다.

`PartialType()` 함수는 입력 유형의 모든 속성이 선택 사항으로 설정된 유형(클래스)을 반환합니다. 예를 들어 다음과 같은 create 유형이 있다고 가정해 보겠습니다:

```
입력 유형()
CreateUserInput 클래스 {
  @Field()
  이메일: 문자열;

  @Field() 비밀번호:
  문자열;

  필드() 이름: 문자열입
  니다;
}
```

기본적으로 이러한 필드는 모두 필수입니다. 필드는 동일하지만 각 필드가 선택 사항인 유형을 만들려면 클래스 참조(`CreateUserInput`)를 인수로 전달하는 `PartialType()`을 사용합니다:

```
입력 유형()
export class UpdateUserInput extends PartialType(CreateUserInput) {}
```

> 정보 힌트 `PartialType()` 함수는 `@nestjs/graphql` 패키지에서 가져온 것입니다.

`PartialType()` 함수는 데코레이터 팩토리에 대한 참조인 선택적 두 번째 인수를 받습니다. 이 인수는 결과(자식) 클래스에 적용되는 데코레이터 함수를 변경하는 데 사용할 수 있습니다. 지정하지 않으면 자식 클래스는 부모 클래스(첫 번째 인수에서 참조된 클래스)와 동일한 데코레이터를 효과적으로 사용합니다. 위 예제에서는 `@InputType()` 데코레이터로 어노테이션된 `CreateUserInput을` 확장하고 있습니다. UpdateUserInput도 `@InputType()`으로 장식된 것처럼 취급되기를 원하므로 두 번째 인수로 `InputType을` 전달할 필요가 없습니다. 부모와 자식

유형이 다른 경우(예: 부모가 `@ObjectType으로` 장식된 경우) 두 번째 인수로 `InputType을` 전달합니다. 예를 들어

```
입력 유형()
export class UpdateUserInput extends PartialType(User, InputType) {}
```

선택

`PickType()` 함수는 입력 유형에서 속성 집합을 선택하여 새 유형(클래스)을 구성합니다. 예를 들어 다음과 같은 유형으로 시작한다고 가정해 보겠습니다:

```
입력 유형()
CreateUserInput 클래스 {
  @Field()
  이메일: 문자열;

  @Field() 비밀번호:
  문자열;

  필드() 이름: 문자열입
  니다;
}
```

이 클래스에서 `PickType()` 유틸리티 함수를 사용하여 프로퍼티 집합을 선택할 수 있습니다:

```
입력 유형()
내보내기 클래스 UpdateEmailInput extends PickType(CreateUserInput, [
  '이메일,
const) {}
```

> 정보 힌트 `PickType()` 함수는 `@nestjs/graphql` 패키지에서 가져온 것입니다.

생략

`OmitType()` 함수는 입력 유형에서 모든 속성을 선택한 다음 특정 키 집합을 제거하여 유형을 구성합니다. 예를 들어 다음과 같은 유형으로 시작한다고 가정해 보겠습니다:

```
입력 유형()

CreateUserInput 클래스 {
  @Field()
  이메일: 문자열;

  @Field() 비밀번호:
  문자열;
```

```
  필드() 이름: 문자열입
  니다;
}
```

아래와 같이 이메일을 제외한 모든 속성을 가진 파생 유형을 생성할 수 있습니다. 이 구조에서 OmitType의 두 번째 인수는 속성 이름의 배열입니다.

```
입력 유형()
내보내기 클래스 UpdateUserInput extends OmitType(CreateUserInput, [ '
  이메일,
const) {}
```

> 정보 힌트 OmitType() 함수는 @nestjs/graphql 패키지에서 가져온 것입니다.

교차로

IntersectionType() 함수는 두 유형을 하나의 새로운 유형(클래스)으로 결합합니다. 예를 들어 다음과 같은 두 가지 유형으로 시작한다고 가정해 보겠습니다:

```
입력 유형()
CreateUserInput 클래스 {
  @Field()
  이메일: 문자열;

  @Field() 비밀번호:
  문자열;
}

객체 유형()
내보내기 클래스 AdditionalUserInfo {
  @Field()
  이름: 문자열입니다;

  @Field() 성: 문자열
  ;
}
```

두 유형의 모든 속성을 결합한 새로운 유형을 생성할 수 있습니다.

```
입력 유형()

내보내기 클래스 UpdateUserInput extends IntersectionType(
  CreateUserInput,
  추가 사용자 정보,
) {}
```

> 정보 힌트 `IntersectionType()` 함수는 `@nestjs/graphql` 패키지에서 가져온 것입니다.

구성

유형 매핑 유틸리티 함수는 컴포저블이 가능합니다. 예를 들어 다음은 `이메일을` 제외한 `CreateUserInput` 유형의 모든 속성을 가진 유형(클래스)을 생성하며, 이러한 속성은 선택 사항으로 설정됩니다:

```
입력 유형()
내보내기 클래스 UpdateUserInput extends PartialType(
  OmitType(CreateUserInput, ['email'] as const),
) {}
```

아폴로 플러그인

플러그인을 사용하면 특정 이벤트에 대한 응답으로 사용자 지정 작업을 수행하여 Apollo Server의 핵심 기능을 확장할 수 있습니다. 현재 이러한 이벤트는 GraphQL 요청 수명 주기의 개별 단계와 Apollo Server 자체의 시작에 해당합니다(자세한 내용은 여기를 참조하세요). 예를 들어 기본 로깅 플러그인은 Apollo Server로 전송되는 각 요청과 관련된 GraphQL 쿼리 문자열을 로깅할 수 있습니다.

사용자 지정 플러그인

플러그인을 만들려면 `@nestjs/apollo` 패키지에서 내보낸 `@Plugin` 데코레이터로 주석이 달린 클래스를 선언하세요. 또한 코드 자동 완성을 개선하려면 `@apollo/server` 패키지에서 `ApolloServerPlugin` 인터페이스를 구현하세요.

```
'@apollo/server'에서 { ApolloServerPlugin, GraphQLRequestListener }
를 가져옵니다;
'@nestjs/apollo'에서 { Plugin }을 가져옵니다;

플러그인()
로깅 플러그인 내보내기 클래스 ApolloServerPlugin 구현 {
  async requestDidStart(): Promise<GraphQLRequestListener<any>> {
    console.log('요청 시작');
    반환 {
      async willSendResponse() { console.log('응
        답을 보냅니다');
      },
    };
  }
}
```

이 작업을 완료하면 `로깅 플러그인을` 공급자로 등록할 수 있습니다.

```
모듈({
  제공자: [로깅 플러그인],
})
내보내기 클래스 CommonModule {}
```

Nest는 플러그인을 자동으로 인스턴스화하여 아폴로 서버에 적용합니다. 외부 플

러그인 사용

기본으로 제공되는 플러그인은 여러 가지가 있습니다. 기존 플러그인을 사용하려면 해당 플러그인을 가져와서 다음 위치에 추가하면 됩니다.

플러그인 배열:

```
GraphQLModule.forRoot({
  // ...
```

```
    플러그인: [ApolloServerOperationRegistry({ /* 옵션 */})]
  }),
```

> 정보 힌트 `ApolloServerOperationRegistry` 플러그인은 `@apollo/server- 플러그인-운영-레지스트리` 패키지에서 내보내집니다.

머큐리우스 플러그인

기존 머큐리우스 전용 Fastify 플러그인 중 일부는 플러그인 트리에서 머큐리우스 플러그인(여기에서 자세히 읽어보세요) 다음에 로드해야 합니다.

> 경고 경고 머큐리업로드는 예외이므로 메인 파일에 등록해야 합니다.

이를 위해 `머큐리우스 드라이버는` 선택적 `플러그인` 구성 옵션을 노출합니다. 이는 플러그인과 `옵션이라는` 두 가지 속성으로 구성된 객체 배열을 나타냅니다. 따라서 캐시 플러그인을 등록하면 다음과 같이 표시됩니다:

```
GraphQLModule.forRoot({
  driver: MercuriusDriver,
  // ... 플러
  그인: [
    {
      플러그인: 캐시, 옵션: {
        ttl: 10,
        policy: {
          쿼리: { add:
            true
          }
        }
      },
    }
  ]
}),
```

복잡성

> 경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

쿼리 복잡도를 사용하면 특정 필드의 복잡도를 정의하고 최대 복잡도를 가진 쿼리를 제한할 수 있습니다. 간단한 숫자를 사용하여 각 필드의 복잡도를 정의하는 것이 좋습니다. 일반적인 기본값은 각 필드에 복잡도 `1`을 지정하는 것입니다. 또한 복잡도 추정기를 사용하여 GraphQL 쿼리의 복잡도 계산을 사용자 지정할 수 있습니다. 복잡성 추정기는 필드의 복잡성을 계산하는 간단한 함수입니다. 규칙에 복잡도 추정기를 원하는 수만큼 추가한 다음 차례로 실행할 수 있습니다. 숫자 복잡도 값을 반환하는 첫 번째 추정기가 해당 필드의 복잡도를 결정합니다.

`nestjs/graphql` 패키지는 비용 분석 기반 솔루션을 제공하는 그래프 쿼리 복잡성 같은 도구와 매우 잘 통합됩니다. 이 라이브러리를 사용하면 실행 비용이 너무 많이 드는 것으로 간주되는 GraphQL 서버에 대한 쿼리를 거부할 수 있습니다.

설치

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

```
$ npm install --save graphql-query-complexity
```

시작하기

설치 프로세스가 완료되면 `ComplexityPlugin` 클래스를 정의할 수 있습니다:

```
"@nestjs/graphql"에서 { GraphQLSchemaHost }를 가져오고,
"@nestjs/apollo"에서 { Plugin }을 가져옵니다;
가져 오기 {
  ApolloServerPlugin,
  GraphQLRequestListener,
'apollo-server-plugin-base'에서 } import
{ GraphQLError } from 'graphql'; import
{
  fieldExtensionsEstimator,
  getComplexity,
  simpleEstimator,
그래프 쿼리 복잡도'에서 }를 가져옵니다;

플러그인()
export class ComplexityPlugin 구현 ApolloServerPlugin { constructor(private
  gqlSchemaHost: GraphQLSchemaHost) {}

  async requestDidStart(): Promise<GraphQLRequestListener> {
    const maxComplexity = 20;
    const { 스키마 } = this.gqlSchemaHost;

    반환 {
```

```
      async didResolveOperation({ request, document }) {
        const complexity = getComplexity({
          스키마,
          작업 이름: 요청.작업 이름, 쿼리: 문서,
          변수: 요청.변수, 추정자: [
            fieldExtensionsEstimator(),
            simpleEstimator({ defaultComplexity: 1 }),
          ],
        });
        if (complexity > maxComplexity) {
          throw new GraphQLError(
            쿼리가 너무 복잡합니다: ${complicity}. 허용되는 최대 복잡도:
${maxComplexity}`,
          );
        }
        console.log('쿼리 복잡성:', 복잡성);
      },
    };
  }
}
```

데모 목적으로 허용되는 최대 복잡도를 `20으로` 지정했습니다. 위의 예제에서는 `단순` 추정기와 `필드 확장` 추정기라는 두 가지 추정기를 사용했습니다.

- `simpleEstimator`: 단순 추정기는 각 필드에 대해 고정된 복잡도를 반환합니다.
- `fieldExtensionsEstimator`: 필드 확장 추정기는 스키마의 각 필드에 대한 복잡도 값을 추출합니다.

> 정보 힌트 모든 모듈의 공급자 배열에 이 클래스를 추가하는 것을 잊지 마세요.

필드 수준의 복잡성

이 플러그인을 사용하면 이제 `복잡도를` 지정하여 모든 필드에 대한 복잡도를 정의할 수 있습니다.

프로퍼티를 `@Field()` 데코레이터에 전달하면 다음과 같이 됩니다:

```
@Field({ complexity: 3 })
title: 문자열;
```

또는 추정기 함수를 정의할 수도 있습니다:

```
@Field({ complexity: (options: ComplexityEstimatorArgs) => ... })
title: 문자열;
```

쿼리/변이 수준 복잡성

또한 `@Query()` 및 `@Mutation()` 데코레이터에는 다음과 같이 지정된 `복잡성` 속성이 있을 수 있습니다:

```
쿼리({ 복잡성: (옵션: ComplexityEstimatorArgs) => options.args.count
* options.childComplexity }) 항목(@Args('count') count: 숫자) {
  return this.itemsService.getItems({ count });
}
```

확장 기능

> 경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

확장은 유형 구성에서 임의의 데이터를 정의할 수 있는 고급 저수준 기능입니다. 특정 필드에 사용자 지정 메타데이터를 첨부하면 보다 정교하고 일반적인 솔루션을 만들 수 있습니다. 예를 들어, 확장 기능을 사용하면 특정 필드에 액세스하는 데 필요한 필드 수준 역할을 정의할 수 있습니다. 이러한 역할은 런타임에 반영되어 호출자가 특정 필드를 검색할 수 있는 충분한 권한을 가지고 있는지 여부를 결정할 수 있습니다.

사용자 지정 메타데이터 추가

필드에 대한 사용자 지정 메타데이터를 첨부하려면 `@Extensions()` 데코레이터를 내보낸 후 `nestjs/graphql` 패키지.

```
@Field()
확장 기능({ 역할: 역할.관리자 }) 비밀번호: 문
자열;
```

위의 예에서는 `역할` 메타데이터 속성에 `Role.ADMIN` 값을 할당했습니다. `Role은` 시스템에서 사용 가능한 모든 사용자 역할을 그룹화하는 간단한 TypeScript 열거형입니다.

필드에 메타데이터를 설정하는 것 외에도 클래스 수준 및 메서드 수준(예: 쿼리 핸들러)에서 `@Extensions()` 데코레이터를 사용할 수 있다는 점에 유의하세요.

사용자 지정 메타데이터 사용

사용자 정의 메타데이터를 활용하는 로직은 필요에 따라 얼마든지 복잡해질 수 있습니다. 예를 들어 메서드 호출당 이벤트를 저장/기록하는 간단한 인터셉터 또는 필드를 검색하는 데 필요한 역할을 호출자 권한(필드 수준 권한 시스템)과 일치시키는 필드 미들웨어를 만들 수 있습니다.

예시를 위해 사용자의 역할(여기에 하드코딩됨)을 대상 필드에 액세스하는 데 필요한 역할과 비교하는 `checkRoleMiddleware를` 정의해 보겠습니다:

```
export const checkRoleMiddleware: FieldMiddleware = async (
  ctx: 미들웨어 컨텍스트,
  다음: 다음Fn,
) => {
  const { 정보 } = CTX;
  const { extensions } = info.parentType.getFields()[info.fieldName];

  /**
   * 실제 애플리케이션에서 "userRole" 변수
   * 은 호출자(사용자)의 역할을 나타내야 합니다(예: "ctx.user.role").
   */
  const userRole = Role.USER;
  if (userRole === extensions.role) {
```

```
      // 또는 그냥 "널을 반환"하여 무시할 수 있습니다
      .
        `사용자에게 "${info.fieldName}" 필드에 액세스할 수 있는 충분한 권한이
  없습니다.`,
      );
    }
    다음()을 반환합니다;
  };
```

이 준비가 완료되면 다음과 같이 비밀번호 필드에 대한 미들웨어를 등록할 수 있습니다:

```
@Field({ 미들웨어: [checkRoleMiddleware] })
@Extensions({ 역할: Role.ADMIN })
비밀번호: 문자열;
```

CLI 플러그인

> 경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

TypeScript의 메타데이터 반영 시스템에는 몇 가지 제약이 있어 클래스가 어떤 프로퍼티로 구성되어 있는지 확인하거나 특정 프로퍼티가 선택 사항인지 필수 사항인지 인식할 수 없습니다. 하지만 이러한 제약 조건 중 일부는 컴파일 시점에 해결할 수 있습니다. Nest는 필요한 상용구 코드의 양을 줄이기 위해 TypeScript 컴파일 프로세스를 개선하는 플러그인을 제공합니다.

> 정보 힌트 이 플러그인은 옵트인입니다. 원하는 경우 모든 데코레이터를 수동으로 선언하거나 필요한 곳에 특정 데코레이터만 선언할 수 있습니다.

개요

GraphQL 플러그인이 자동으로 실행됩니다:

- HideField를 사용하지 않는 한 모든 입력 객체, 객체 유형 및 인수 클래스 속성에 `@Field를` 주석으로 추가합니다.
- 물음표에 따라 `널` 가능 속성을 설정합니다(예: `이름?: 문자열이` 설정됩니다). `무효화 가능: 참`)
- 유형에 따라 `유형` 속성을 설정합니다(배열도 지원).
- 댓글을 기반으로 프로퍼티에 대한 설명을 생성합니다(`introspectComments가 true로` 설정된 경우).

플러그인에서 분석하려면 파일 이름에 다음 접미사 중 하나가 포함되어야 한다는 점에 유의하세요: `['.input.ts', '.args.ts', '.entity.ts', '.model.ts']`(예: `author.entity.ts`). 다른 접미사를 사용하는 경우 `typeFileNameSuffix` 옵션을 지정하여 플러그인의 동작을 조정할 수 있습니다(아래 참조).

지금까지 배운 내용을 바탕으로 GraphQL에서 유형을 어떻게 선언해야 하는지 패키지에 알리기 위해 많은 코드를 복제해야 합니다. 예를 들어 간단한 `Author` 클래스를 다음과 같이 정의할 수 있습니다:

```
@@파일명(authors/models/author.model) @ObjectType()
export class Author {
  @Field(type => ID)
  id: number;

  @Field({ nullable: true })
  firstName?: 문자열;

  @Field({ nullable: true })
  lastName?: 문자열;

  필드(유형 => [게시물]) 게시물
  : Post[];
}
```

중간 규모의 프로젝트에서는 큰 문제가 되지 않지만, 클래스의 수가 많아지면 장황해지고 유지 관리가 어려워집니다.

GraphQL 플러그인을 활성화하면 위의 클래스 정의를 간단하게 선언할 수 있습니다:

```
@@파일명(authors/models/author.model) @ObjectType()
export class Author {
  @Field(type => ID)
  id: 숫자;
  firstName?: 문자열;
  lastName?: 문자열;
  posts: Post[];
}
```

플러그인은 추상 구문 트리를 기반으로 적절한 데코레이터를 즉석에서 추가합니다. 따라서 코드 곳곳에 흩어져 있는 `@Field` 데코레이터로 고생할 필요가 없습니다.

> 정보 힌트 플러그인은 누락된 GraphQL 프로퍼티를 자동으로 생성하지만, 재정의해야 하는 경우 `@Field()`를 통해 명시적으로 설정하기만 하면 됩니다.

댓글 성찰

댓글 인트로스펙션 기능을 활성화하면 CLI 플러그인이 댓글을 기반으로 필드에 대한 설명을 생성합니다.

예를 들어 `역할` 속성을 예로 들어 보겠습니다:

```
/**
 * 사용자 역할 목록
 */
@Field(() => [문자열], {
  설명: '사용자 역할 목록'
})
역할: 문자열[];
```

설명 값을 복제해야 합니다. `introspectComments`를 활성화하면 CLI 플러그인이 이러한 설명을 추출하여 속성에 대한 설명을 자동으로 제공할 수 있습니다. 이제 위의 필드는 다음과 같이 간단하게 선언할 수 있습니다:

```
/**
 * 사용자 역할 목록
 */
역할: 문자열[];
```

CLI 플러그인 사용

플러그인을 활성화하려면 `nest-cli.json`(Nest CLI를 사용하는 경우)을 열고 다음 `플러그인을` 추가합니다.
구성합니다:

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "src", "컴파일러옵션": {
    "plugins": ["@nestjs/graphql"]
  }
}
```

옵션 속성을 사용하여 플러그인의 동작을 사용자 정의할 수 있습니다.

```
"플러그인": [
  {
    "name": "@nestjs/graphql", "옵션":
    {
      "typeFileNameSuffix": [".input.ts", ".args.ts"],
      "introspectComments": true
    }
  }
]
```

옵션 속성은 다음 인터페이스를 충족해야 합니다:

```
export interface PluginOptions {
  typeFileNameSuffix?: string[];
  introspectComments?: boolean;
}
```

옵션	기본값	설명
typeFileName접미사	['.input.ts', '.args.ts', '.entity.ts', '.model.ts']	GraphQL 유형 파일 접미사
introspectComments 거짓		true로 설정하면 플러그인이 다음을 생성합니다. 댓글을 기반으로 한 속성 설명

CLI를 사용하지 않고 대신 사용자 정의 웹팩 구성을 사용하는 경우 이 플러그인을 ts-loader와 함께 사용할 수 있습니다:

```
getCustomTransformers: (program: any) => ({{
  이전: [require('@nestjs/graphql/plugin').before({}, program)]
}),
```

SWC 빌더

표준 설정(비모노레포)의 경우 SWC 빌더에서 CLI 플러그인을 사용하려면 여기에 설명된 대로 유형 검사를 사용하도록 설정해야 합니다.

```
nest start -b swc --type-check
```

모노레포 설정의 경우 여기 지침을 따르세요.

```
$ npx ts-node src/generate-metadata.ts
# 또는 npx ts-node apps/{YOUR_APP}/src/generate-metadata.ts
```

이제 아래와 같이 직렬화된 메타데이터 파일을 `GraphQLModule` 메서드를 통해 로드해야 합니다:

```
'./metadata'에서 메타데이터 가져오기; // <-- "플러그인 메타데이터 생성기"에 의해 자동 생성된 파일입니다.

GraphQLModule.forRoot<...>({
  ..., // 기타 옵션 메타데이터
  ,
}),
```

ts-jest와 통합(e2e 테스트)

이 플러그인을 활성화한 상태에서 e2e 테스트를 실행할 때 스키마 컴파일과 관련된 문제가 발생할 수 있습니다. 예를 들어 가장 일반적인 오류 중 하나는 다음과 같습니다:

```
개체 유형 <이름>은 하나 이상의 필드를 정의해야 합니다.
```

이 문제는 `jest` 구성이 `@nestjs/graphql/plugin` 플러그인을 어디에도 가져오지 않기 때문에 발생합니다.

이 문제를 해결하려면 e2e 테스트 디렉터리에 다음 파일을 생성하세요:

```
const transformer = require('@nestjs/graphql/plugin');

module.exports.name = 'nestjs-graphql-transformer';
// 아래 구성을 변경할 때마다 버전 번호를 변경해야 합니다. 그렇지 않으면 jest가 변
경 사항을 감지하지 못합니다 module.exports.version = 1;

module.exports.factory = (cs) => {
  return transformer.before(
    {
```

```
      // @nestjs/graphql/플러그인 옵션(비워둘 수 있음)
    },
    cs.program, // 이전 버전의 Jest의 경우 "cs.tsCompiler.program" (<=
v27)
  );
};
```

이 설정이 완료되면 `jest` 구성 파일에서 AST 트랜스포머를 가져옵니다. 기본적으로(스타터 애플리케이션에서) e2e 테스트 구성 파일은 `테스트` 폴더 아래에 있으며 이름은 `jest-e2e.json`입니다.

```
{
  ... // 기타 구성 "전역": {
    "ts-jest": {
      "astTransformers": {
        "전에": ["<위에서 생성한 파일 경로>"]
      }
    }
  }
}
```

jest@^29를 사용하는 경우 이전 접근 방식이 더 이상 사용되지 않으므로 아래 스니펫을 사용하세요.

```
{
  ... // 기타 구성 "변환": {
    "^.+\\.(t|j)s$": [
      "ts-jest",
      {
        "astTransformers": {
          "전에": ["<위에서 생성한 파일 경로>"]
        }
      }
    ]
  }
}
```

공유 모델

> 경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

프로젝트 백엔드에 타입스크립트를 사용할 때의 가장 큰 장점 중 하나는 일반적인 타입스크립트 패키지를 사용하여 타입스크립트 기반 프론트엔드 애플리케이션에서 동일한 모델을 재사용할 수 있다는 점입니다.

하지만 문제가 있습니다. 코드 우선 접근 방식을 사용하여 생성된 모델은 GraphQL 관련 데코레이터로 많이 장식되어 있습니다. 이러한 데코레이터는 프론트엔드에서 관련성이 없어 성능에 부정적인 영향을 미칩니다.

모델 심 사용

이 문제를 해결하기 위해 NestJS는 웹팩(또는 이와 유사한) 구성을 사용하여 원래 데코레이터를 비활성 코드로 대체할 수 있는 "shim"을 제공합니다. 이 shim을 사용하려면 `@nestjs/graphql` 패키지와 shim 사이에 별칭을 구성합니다.

예를 들어 웹팩의 경우 이 문제는 다음과 같이 해결됩니다:

```
해결합니다: { // 참조: https://webpack.js.org/configuration/resolve/
  alias: {
      "@nestjs/graphql": path.resolve(__dirname,
"../node_modules/@nestjs/graphql/dist/extra/graphql-model-shim")
  }
}
```

> 정보 힌트 TypeORM 패키지에는 여기에서 찾을 수 있는 유사한 심이 있습니다.

기타 기능

GraphQL 세계에서는 인증이나 작업의 부작용과 같은 문제를 처리하는 것에 대해 많은 논쟁이 있습니다. 비즈니스 로직 내부에서 처리해야 할까요? 고차 함수를 사용하여
권한 부여 로직으로 쿼리 및 변형을 개선해야 하나요? 아니면 스키마 지시문을 사용해야 할까요? 이러한 질문에 대한 정답은 하나도 없습니다.

Nest는 가드 및 인터셉터와 같은 크로스 플랫폼 기능으로 이러한 문제를 해결하도록 지원합니다. Nest의 철학은 중복을 줄이고 잘 구조화되고 가독성이 높으며 일관된 애플리케이션을 만드는 데 도움이 되는 도구를 제공하는 것입니다.

개요

GraphQL을 사용하면 표준 가드, 인터셉터, 필터 및 파이프를 다른 RESTful 애플리케이션과 동일한 방식으로 사용할 수 있습니다. 또한 사용자 지정 데코레이터 기능을 활용하여 자신만의 데코레이터를 쉽게 만들 수 있습니다. 샘플 GraphQL 쿼리 핸들러를 살펴보겠습니다.

```
쿼리('author') @사용가드
(AuthGuard)
async getAuthor(@Args('id', ParseIntPipe) id: number) {
  return this.authorsService.findOneById(id)를 반환합니다;
}
```

보시다시피 GraphQL은 HTTP REST 핸들러와 동일한 방식으로 가드 및 파이프 모두에서 작동합니다. 따라서 인증 로직을 가드로 옮길 수 있으며, REST와 GraphQL API 인터페이스 모두에서 동일한 가드 클래스를 재사용할 수도 있습니다. 마찬가지로 인터셉터도 두 가지 유형의 애플리케이션에서 동일한 방식으로 작동합니다:

```
돌연변이() @사용인터셉터(이벤트인터셉터)
async upvotePost(@Args('postId') postId: number) {
  return this.postsService.upvoteById({ id: postId });
}
```

실행 컨텍스트

```
'@nestjs/common'에서 { CanActivate, ExecutionContext, Injectable }을
임포트합니다;
'@nestjs/graphql'에서 { GqlExecutionContext }를 임포트합니다;
```

GraphQL은 들어오는 요청에서 다른 유형의 데이터를 수신하기 때문에 가드와 인터셉터에서 수신하는 실행 컨텍스트는 GraphQL과 REST에서 다소 다릅니다. GraphQL 리졸버에는 `루트`, `args`, `컨텍스트`, `정보 등` 고유한 인자 집합이 있습니다. 따라서 가드와 인터셉터는 일반 ExecutionContext를 `GqlExecutionContext로` 변환해야 합니다. 이것은 간단합니다:

```
@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const ctx = GqlExecutionContext.create(context);
    return true;
  }
}
```

`GqlExecutionContext.create()`가 반환하는 GraphQL 컨텍스트 객체에는 각 GraphQL 리졸버 인자에 대한 get 메서드가 노출됩니다(예: `getArgs()`, `getContext()` 등). 변환이 완료되면 현재 요청에 대한 GraphQL 인수를 쉽게 선택할 수 있습니다.

예외 필터

Nest 표준 예외 필터는 GraphQL 애플리케이션과도 호환됩니다. `ExecutionContext`와 마찬가지로 GraphQL 앱은 `ArgumentsHost` 객체를 `GqlArgumentsHost` 객체로 변환해야 합니다.

```
@Catch(HttpException)
export class HttpExceptionFilter implements GqlExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const gqlHost = GqlArgumentsHost.create(host); 반환
    예외;
  }
}
```

> 정보 힌트 GqlExceptionFilter와 GqlArgumentsHost는 모두 `nestjs/graphql` 패키지.

REST의 경우와 달리 기본 `응답` 객체를 사용하여 응답을 생성하지 않습니다. 사용자 지정 데코레이터

앞서 언급했듯이 사용자 지정 데코레이터 기능은 GraphQL 리졸버에서 예상대로 작동합니다.

```
export const User = createParamDecorator( (데이
  터: 알 수 없음, ctx: ExecutionContext) =>
    GqlExecutionContext.create(ctx).getContext().user,
);
@Mutation()
비동기 업보트포스트(
  @User() 사용자: 사용자 엔티티,
```

`사용자()` 사용자 지정 데코레이터를 다음과 같이 사용합니다:

```
    @Args('postId') postId: 숫자,
  ) {}
```

> 정보 힌트 위의 예에서는 `사용자` 개체가 GraphQL 애플리케이션의 컨텍스트에 할당되어 있다고 가정했습니다.

필드 리졸버 수준에서 인핸서를 실행합니다.

GraphQL 컨텍스트에서 Nest는 필드 수준에서 인핸서(인터셉터, 가드 및 필터의 총칭)를 실행하지 않으며, 최상위 수준인 `@Query()`/@Mutation`()` 메서드에 대해서만 실행합니다(이 문제를 참조하세요). `GqlModuleOptions에서 fieldResolverEnhancers` 옵션을 설정하여 `@ResolveField()`로 주석이 달린 메서드에 대해 인터셉터, 가드 또는 필터를 실행하도록 Nest에 지시할 수 있습니다. `'인터셉터'`, `'가드'` 및/또는 `'필터'` 목록을 적절히 전달하세요:

```
GraphQLModule.forRoot({
  fieldResolverEnhancers: ['인터셉터']
}),
```

> 경고 필드 확인자에 인핸서를 활성화하면 많은 레코드를 반환하고 필드 확인자가 수천 번 실행될 때 성능 문제가 발생할 수 있습니다. 따라서 `fieldResolverEnhancers를` 활성화할 때는 필드 해석기에 꼭 필요하지 않은 인핸서의 실행을 건너뛰는 것이 좋습니다. 다음 도우미 함수를 사용하여 이 작업을 수행할 수

```
export 함수 isResolvingGraphQLField(context: ExecutionContext): boolean
{
  if (context.getType<GqlContextType>() === 'graphql') {
    const gqlContext = GqlExecutionContext.create(context);
    const info = gqlContext.getInfo();
    const parentType = info.parentType.name;
    반환 부모 유형 !== '쿼리' && 부모 유형 !== '돌연변이';
  }
  거짓을 반환합니다;
}
```

사용자 지정 드라이버 만들기

Nest는 두 가지 공식 드라이버를 기본으로 제공합니다: nestjs/apollo와 `@nestjs/mercurius`, 그리고 개발자가 새로운 커스텀 드라이버를 빌드할 수 있는 API를 제공합니다. 사용자 정의 드라이버를 사용하면 모든

GraphQL 라이브러리를 통합하거나 기존 통합을 확장하여 추가 기능을 추가할 수 있습니다.

예를 들어 `express-graphql` 패키지를 통합하려면 다음과 같은 드라이버 클래스를 만들 수 있습니다:

```
'@nestjs/graphql'에서 { AbstractGraphQLDriver, GqlModuleOptions }를 임포트하고,
'express-graphql'에서 { graphqlHTTP }를 임포트합니다;
```

```
ExpressGraphQLDriver 클래스는 AbstractGraphQLDriver를 확장합니다 {
  async start(옵션: GqlModuleOptions<any>): Promise<void> {
    options = await this.graphQlFactory.mergeWithSchema(options);

    const { httpAdapter } = this.httpAdapterHost;
    httpAdapter.use(
      '/graphql',
      graphqlHTTP({
        스키마: options.schema,
        graphiql: true,
      }),
    );
  }

  async stop() {}
}
```

그런 다음 다음과 같이 사용하세요:

```
GraphQLModule.forRoot({
  드라이버: ExpressGraphQLDriver,
});
```

연맹

페더레이션은 모놀리식 GraphQL 서버를 독립적인 마이크로서비스로 분할하는 수단을 제공합니다. 페더레이션은 게이트웨이와 하나 이상의 페더레이션 마이크로서비스라는 두 가지 구성 요소로 이루어져 있습니다. 각 마이크로서비스는 스키마의 일부를 보유하며 게이트웨이는 스키마를 클라이언트에서 사용할 수 있는 단일 스키마로 병합합니다.

아폴로 문서를 인용하자면, 페더레이션은 이러한 핵심 원칙에 따라 설계되었습니다:

- 그래프 작성은 선언적이어야 합니다. 페더레이션을 사용하면 명령형 스키마 스티칭 코드를 작성하는 대신 스키마 내에서 선언적으로 그래프를 작성할 수 있습니다.
- 코드는 유형이 아닌 관심사별로 분리해야 합니다. 사용자나 제품처럼 중요한 유형의 모든 측면을 한 팀이 제어할 수 없는 경우가 많으므로 이러한 유형의 정의는 중앙 집중식보다는 여러 팀과 코드베이스에 분산되어야 합니다.
- 그래프는 고객이 이해하기 쉽도록 단순해야 합니다. 연합된 서비스를 함께 사용하면 클라이언트에서 어
- 떻게 소비되는지 정확하게 반영하는 완전한 제품 중심의 그래프를 형성할 수 있습니다. 이는 사양을 준수하는 언어의 기능만 사용하는 GraphQL일 뿐입니다. JavaScript뿐만 아니라 모든 언어가 페더레이션을 구현할 수 있습니다.

> 경고 경고 연합은 현재 구독을 지원하지 않습니다.

다음 섹션에서는 게이트웨이와 두 개의 페더레이션 엔드포인트로 구성된 데모 애플리케이션을 설정해 보겠습니다: 사용자 서비스 및 게시물 서비스입니다.

아폴로와의 연합

필요한 종속성을 설치하는 것으로 시작하세요:

```
$ npm install --save @apollo/federation @apollo/subgraph
```

스키마 우선

"사용자 서비스"는 간단한 스키마를 제공합니다. `key` 지시문에 주목하세요. 이 지시문은 Apollo 쿼리 플래너에 특정 `User` 인스턴스를 지정하면 해당 인스턴스를 가져올 수 있다고 지시합니다. 또한 쿼리 유형을 확장한다는 점

에 유의하세요.

```
type User @key(fields: "id") {
  id: ID!
  이름: 문자열!
}

확장 유형 Query {
  getUser(id: ID!): User
}
```

리졸버는 `resolveReference()`라는 메서드 하나를 추가로 제공합니다. 이 메서드는 관련 리소스에 사용자 인스턴스가 필요할 때마다 Apollo 게이트웨이에 의해 트리거됩니다. 나중에 포스트 서비스에서 이에 대한 예를 살펴보겠습니다. 이 메서드에는 `@ResolveReference()` 데코레이터를 사용하여 주석을 달아야 한다는 점에 유의하세요.

```
'@nestjs/graphql'에서 { Args, Query, Resolver, ResolveReference }를 가져오고,
'./users.service'에서 { UsersService }를 가져옵니다;

@Resolver('User')
사용자 해결자 클래스 내보내기 {
  constructor(private usersService: UsersService) {}

  쿼리()
  getUser(@Args('id') id: 문자열) {
    this.usersService.findById(id)를 반환합니다;
  }

  @ResolveReference()
  resolveReference(reference: { __typename: 문자열; id: 문자열 }) {
    return this.usersService.findById(reference.id);
  }
}
```

마지막으로, GraphQLModule을 등록하고

`ApolloFederationDriver` 드라이버를 구성 개체에 추가합니다:

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 가져오고,
'./users.resolver'에서 { UsersResolver }를 가져옵니다;


모듈({ import:
  [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      typePaths: ['**/*.graphql'],
    }),
  ],
  제공자: [UsersResolver],
})
내보내기 클래스 AppModule {}
```

코드 우선

먼저 사용자 엔티티에 몇 가지 데코레이터를 추가합니다.

```
'@nestjs/graphql'에서 { 지시어, 필드, ID, 객체 유형 } 임포트; @ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((유형) => ID)
  id: 숫자;

  @Field() 이름:
  문자열;
}
```

리졸버는 `resolveReference()`라는 메서드 하나를 추가로 제공합니다. 이 메서드는 관련 리소스에 사용자 인스턴스가 필요할 때마다 Apollo 게이트웨이에 의해 트리거됩니다. 나중에 포스트 서비스에서 이에 대한 예를 살펴보겠습니다. 이 메서드에는 `@ResolveReference()` 데코레이터를 사용하여 주석을 달아야 한다는 점에 유의하세요.

```
'@nestjs/graphql'에서 { Args, Query, Resolver, ResolveReference } 가져오기;
'./user.entity'에서 { User } 가져오기;
'./users.service'에서 { UsersService }를 가져옵니다;

@Resolver((of) => User)
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query((returns) => User)
  getUser(@Args('id') id: number): User {
    this.usersService.findById(id)를 반환합니다;
  }

  @ResolveReference()
  resolveReference(reference: { __유형명: 문자열; id: 숫자 }): User { return
    this.usersService.findById(reference.id);
  }
}
```

마지막으로, GraphQLModule을 등록하고

`ApolloFederationDriver` 드라이버를 구성 개체에 추가합니다:

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'./users.resolver'에서 { UsersResolver }를 가져옵니다;
'./users.service'에서 { UsersService } import; // 이 예제에는 포함되지 않았습니
다.
```

```
모듈({ import:
  [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: true,
    }),
  ],
  제공자: [UsersResolver, UsersService],
})
내보내기 클래스 AppModule {}
```

코드 우선 모드에서는 여기에서, 스키마 우선 모드에서는 여기에서 작동하는 예제를 확인할 수

있습니다. 페더레이션 예제**:** Posts

게시물 서비스는 `getPosts` 쿼리를 통해 집계된 게시물을 제공해야 하지만, `사용자` 유형을 `user.posts` 필드와 함께 입력합니다.

스키마 우선

"게시물 서비스"는 `확장` 키워드로 표시하여 스키마에서 `사용자` 유형을 참조합니다. 또한 `사용자` 유형에 대해 하나의 추가 속성(`게시물`)을 선언합니다. User 인스턴스를 일치시키는 데 사용되는 `@key` 지시어와 `id` 필드가 다른 곳에서 관리됨을 나타내는 `@external` 지시어에 주목하세요.

```
type Post @key(fields: "id") {
  id: ID!
  제목: String!
  body: 문자열! 사
  용자: 사용자
}

확장 유형 사용자 @key(fields: "id") {
  id: ID! 외부
  게시물: [게시물]
}

확장 유형 Query { getPosts:
  [Post]
}
```

다음 예제에서 포스트 리졸버는 다음을 포함하는 참조를 반환하는 getUser() 메서드를 제공합니다. ___유형 이름과 애플리케이션이 참조를 확인하는 데 필요할 수 있는 몇 가지 추가 속성(이 경우 id)이 포함된 참조를 반환하는 메서드를 제공합니다. _____타입네임은 GraphQL 게이트웨이에서 사용자 유형을 담당하는 마이크로서비스를 정확히 찾아내고 해당 인스턴스를 검색하는 데 사용됩니다. 위에서 설명한 "사용자 서비스"는 resolveReference() 메서드를 실행할 때 요청됩니다.

```
'@nestjs/graphql'에서 { Query, Resolver, Parent, ResolveField } 가져오기;
'./posts.service'에서 { PostsService } 가져오기;
'./posts.interfaces'에서 { Post }를 가져옵니다;

@Resolver('Post')
내보내기 클래스 PostsResolver {
  constructor(private postsService: PostsService) {}

  @Query('getPosts')
  getPosts() {
    this.postsService.findAll()을 반환합니다;
  }

  ResolveField('user')
  getUser(@Parent() post: Post) {
    반환 { _____유형명: '사용자', id: post.userId };
  }
}
```

마지막으로, '사용자 서비스' 섹션에서 한 것과 유사하게 `GraphQLModule을` 등록해야 합니다.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 가져오고,
'./posts.resolver'에서 { PostsResolver }를 가져옵니다;

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      typePaths: ['**/*.graphql'],
    }),
  ],
  공급자: [게시물 해결자],
})
내보내기 클래스 AppModule {}
```

코드 우선

먼저 `User` 엔티티를 나타내는 클래스를 선언해야 합니다. 엔티티 자체는 다른 서비스에 있지만, 여기서는 이 엔티티를 사용(정의 확장)할 것입니다. `extends` 및 `@external` 지시어에 유의하세요.

```
'@nestjs/graphql'에서 { 지시문, 객체 유형, 필드, ID } 가져오기;
'./post.entity'에서 { Post } 가져오기;
```

```
객체유형() @디렉티브('@확장') @디렉티브
('@키(필드: "id")') 내보내기 클래스
사용자 {
  @Field((type) => ID) @디
  렉티브('@외부') id: 숫자;

  @Field((type) => [Post])
  posts? Post[];
}
```

이제 다음과 같이 `User` 엔티티에서 확장에 해당하는 리졸버를 만들어 보겠습니다:

```
'@nestjs/graphql'에서 { 부모, ResolveField, 해결자 }를 가져오고,
'./posts.service'에서 { PostsService }를 가져옵니다;
'./post.entity'에서 { Post } 가져오기;
'./user.entity'에서 { User } 가져오기;

@Resolver((of) => User)
export class UsersResolver {
  생성자(비공개 읽기 전용 postsService: PostsService) {}

  @ResolveField((of) => [Post])
  public posts(@Parent() user: User): Post[] {
    return this.postsService.forAuthor(user.id);
  }
}
```

또한 `Post` 엔티티 클래스를 정의해야 합니다:

```
'@nestjs/graphql'에서 { 지시어, 필드, ID, Int, 객체 유형 } 가져오기;
'./user.entity'에서 { 사용자 } 가져오기;

객체 유형() @디렉티브('@키(필드:
"id")') 내보내기 클래스 Post {
  @Field((유형) => ID)
  id: 숫자;

  @Field()
  title: 문자열;

  @Field((유형) => Int)
  authorId: 숫자;

  @Field((type) => User)
  user?: 사용자;
}
```

그리고 그 해결사:

```
'@nestjs/graphql'에서 { Query, Args, ResolveField, Resolver, Parent
}를 임포트합니다;
'./posts.service'에서 { PostsService } 가져오기;
'./post.entity'에서 { Post } 가져오기;
'./user.entity'에서 { User }를 가져옵니다;

@Resolver((of) => Post)
export class PostsResolver {
  생성자(비공개 읽기 전용 postsService: PostsService) {}

  @Query((returns) => Post)
  findPost(@Args('id') id: number): Post {
    이.postsService.findOne(id)을 반환합니다;
  }

  @Query((returns) => [Post])
  getPosts(): Post[] {
    this.postsService.all()을 반환합니다;
  }

  @ResolveField((of) => User)
  user(@Parent() post: Post): any {
    반환 { _____유형명: '사용자', id: post.authorId };
  }
}
```

마지막으로 모듈에 함께 묶습니다. 스키마 빌드 옵션에서 `User가` 고아(외부) 유형임을 지정하는 것에 주목하세요.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;
'./posts.resolvers'에서 { PostsResolvers }를 가져오고,
'./users.resolvers'에서 { UsersResolvers }를 가져옵니다;
'./posts.service'에서 { PostsService } 가져오기; // 예제에는 포함되지 않았습니다.

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: true,
      buildSchemaOptions: {
        고아 유형: [사용자],
      },
```

```
    }),
  ],
  공급자: [게시자 해결자, 사용자 해결자, 게시자 서비스],
})
내보내기 클래스 AppModule {}
```

코드 우선 모드의 경우 여기에서, 스키마 우선 모드의 경우 여기에서 작동하는 예제를 확인할 수 있습니다. 페

더레이션 예제: 게이트웨이

필요한 종속성을 설치하는 것으로 시작하세요:

```
npm install --save @apollo/gateway
```

게이트웨이는 엔드포인트 목록을 지정해야 하며 해당 스키마를 자동으로 검색합니다. 따라서 게이트웨이 서비스의 구현은 코드 접근 방식과 스키마 우선 접근 방식 모두 동일하게 유지됩니다.

```
'@apollo/gateway'에서 { IntrospectAndCompose }를 가져옵니다;
'@nestjs/apollo'에서 { ApolloGatewayDriver, ApolloGatewayDriverConfig }를 가져옵
니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloGatewayDriverConfig>({
      driver: ApolloGatewayDriver,
      서버: {
        // ... 아폴로 서버 옵션 cors:
        true,
      },
      게이트웨이: {
        supergraphSdl: new IntrospectAndCompose({
          subgraphs: [
            { name: 'users', url: 'http://user-service/graphql' },
            { name: 'posts', url: 'http://post-service/graphql' },
          ],
        }),
      },
    }),
  ],
})
내보내기 클래스 AppModule {}
```

코드 우선 모드의 경우 여기에서, 스키마 우선 모드의 경우 여기에서 작업 예제를 확인할 수 있습니다.

머큐리우스와의 연맹

필요한 종속성을 설치하는 것으로 시작하세요:

```
$ npm install --save @apollo/subgraph @nestjs/mercurius
```

> 정보 참고 서브그래프 스키마를 `빌드하려면 @apollo/subgraph` 패키지가 필요합니다(`빌드Sub그래프스키마`, `프린트Sub그래프스키마` 함수).

스키마 우선

"사용자 서비스"는 간단한 스키마를 제공합니다. `key` 지시문에 주목하세요. 이 지시문은 사용자가 `ID를` 지정하면 `사용자` 인스턴스를 가져올 수 있도록 Mercurius 쿼리 플래너에 지시합니다. 또한 `쿼리` 유형을 `확장한다는` 점에 유의하세요.

```
type User @key(fields: "id") {
  id: ID!
  이름: 문자열!
}

확장 유형 Query {
  getUser(id: ID!): User
}
```

리졸버는 `resolveReference()`라는 메서드 하나를 추가로 제공합니다. 이 메서드는 관련 리소스에 사용자 인스턴스가 필요할 때마다 머큐리우스 게이트웨이에 의해 트리거됩니다. 나중에 포스트 서비스에서 이에 대한 예를 살펴보겠습니다. 이 메서드에는 `@ResolveReference()` 데코레이터로 주석을 달아야 한다는 점에 유의하세요.

```
'@nestjs/graphql'에서 { Args, Query, Resolver, ResolveReference }를 가져오고,
'./users.service'에서 { UsersService }를 가져옵니다;

@Resolver('User')
사용자 해결자 클래스 내보내기 {
  constructor(private usersService: UsersService) {}

  쿼리()
  getUser(@Args('id') id: 문자열) {
    this.usersService.findById(id)를 반환합니다;
  }

  @ResolveReference()
  resolveReference(reference: { __typename: 문자열; id: 문자열 }) {
    return this.usersService.findById(reference.id);
  }
}
```

마지막으로, GraphQLModule을 등록하고

`MercuriusFederationDriver` 드라이버를 구성 개체에 추가합니다:

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
}를 '@nestjs/mercurius'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 가져오고,
'./users.resolver'에서 { UsersResolver }를 가져옵니다;

모듈({ import:
  [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      typePaths: ['**/*.graphql'],
      federationMetadata: true,
    }),
  ],
  제공자: [UsersResolver],
})
내보내기 클래스 AppModule {}
```

코드 우선

먼저 `사용자` 엔티티에 몇 가지 데코레이터를 추가합니다.

```
'@nestjs/graphql'에서 { 지시어, 필드, ID, 객체 유형 } 임포트; @ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((유형) => ID)
  id: 숫자;

  @Field() 이름:
  문자열;
}
```

리졸버는 `resolveReference()`라는 메서드 하나를 추가로 제공합니다. 이 메서드는 관련 리소스에 사용자 인스턴스가 필요할 때마다 머큐리우스 게이트웨이에 의해 트리거됩니다. 나중에 포스트 서비스에서 이에 대한 예를 살펴보겠습니다. 이 메서드에는 `@ResolveReference()` 데코레이터로 주석을 달아야 한다는 점에 유의하세

요.

```
'@nestjs/graphql'에서 { Args, Query, Resolver, ResolveReference } 가져오기;
'./user.entity'에서 { User } 가져오기;
'./users.service'에서 { UsersService }를 가져옵니다;
```

```
@Resolver((of) => User)
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query((returns) => User)
  getUser(@Args('id') id: number): User {
    this.usersService.findById(id)를 반환합니다;
  }

  @ResolveReference()
  resolveReference(reference: { __유형명: 문자열; id: 숫자 }): User { return
    this.usersService.findById(reference.id);
  }
}
```

마지막으로, GraphQLModule을 등록하고

`MercuriusFederationDriver` 드라이버를 구성 개체에 추가합니다:

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
}를 '@nestjs/mercurius'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'./users.resolver'에서 { UsersResolver }를 가져옵니다;
'./users.service'에서 { UsersService } import; // 이 예제에는 포함되지 않았습니
다.

모듈({ import:
  [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      autoSchemaFile: true, 페더레이션
      메타데이터: true,
    }),
  ],
  제공자: [UsersResolver, UsersService],
})
내보내기 클래스 AppModule {}
```

페더레이션 예시: 글

게시물 서비스는 `getPosts` 쿼리를 통해 집계된 게시물을 제공해야 하지만, 사용자

유형을 `user.posts` 필드와 함께 입력합니다.

스키마 우선

"게시물 서비스"는 `확장` 키워드로 표시하여 스키마에서 `사용자` 유형을 참조합니다. 또한 `사용자` 유형에 대한 추가 속성(`게시물`)을 하나 더 선언합니다. 매칭에 사용되는 `@key` 지시문에 유의하세요.

인스턴스 및 @external 지시문을 사용하여 ID 필드가 다른 곳에서 관리됨을 나타냅니다.

```
type Post @key(fields: "id") {
  id: ID!
  제목: String!
  body: 문자열! 사
  용자: 사용자
}

확장 유형 사용자 @key(fields: "id") {
  id: ID! 외부
  게시물: [게시물]
}

확장 유형 Query { getPosts:
  [Post]
}
```

다음 예제에서 포스트 리졸버는 다음을 포함하는 참조를 반환하는 getUser() 메서드를 제공합니다. ___유형 이름과 애플리케이션이 참조를 확인하는 데 필요할 수 있는 몇 가지 추가 속성(이 경우 id)이 포함된 참조를 반환하는 메서드를 제공합니다. _____타입네임은 GraphQL 게이트웨이에서 사용자 유형을 담당하는 마이크로서비스를 정확히 찾아내고 해당 인스턴스를 검색하는 데 사용됩니다. 위에서 설명한 "사용자 서비스"는 resolveReference() 메서드를 실행할 때 요청됩니다.

```
'@nestjs/graphql'에서 { Query, Resolver, Parent, ResolveField } 가져오기;
'./posts.service'에서 { PostsService } 가져오기;
'./posts.interfaces'에서 { Post }를 가져옵니다;

@Resolver('Post')
내보내기 클래스 PostsResolver {
  constructor(private postsService: PostsService) {}

  @Query('getPosts')
  getPosts() {
    this.postsService.findAll()을 반환합니다;
  }

  ResolveField('user')
  getUser(@Parent() post: Post) {
    반환 { _____유형명: '사용자', id: post.userId };
  }
}
```

마지막으로, '사용자 서비스' 섹션에서 한 것과 유사하게 `GraphQLModule을` 등록해야 합니다.

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
```

```
}를 '@nestjs/mercurius'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 가져오고,
'./posts.resolver'에서 { PostsResolver }를 가져옵니다;

모듈({ import:
  [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      페더레이션 메타데이터: true, 유형 경로
      : ['**/*.graphql'],
    }),
  ],
  공급자: [게시물 해결자],
})
내보내기 클래스 AppModule {}
```

코드 우선

먼저 `User` 엔티티를 나타내는 클래스를 선언해야 합니다. 엔티티 자체는 다른 서비스에 있지만, 여기서는 이 엔티티를 사용(정의 확장)할 것입니다. `extends` 및 `@external` 지시어에 유의하세요.

```
'@nestjs/graphql'에서 { 지시문, 객체 유형, 필드, ID } 가져오기;
'./post.entity'에서 { Post } 가져오기;

객체유형() @디렉티브('@확장') @디렉티브
('@키(필드: "id")') 내보내기 클래스
사용자 {
  @Field((type) => ID) @디
  렉티브('@외부') id: 숫자;

  @Field((type) => [Post])
  posts? Post[];
}
```

이제 다음과 같이 `User` 엔티티에서 확장에 해당하는 리졸버를 만들어 보겠습니다:

```
'@nestjs/graphql'에서 { 부모, ResolveField, 해결자 }를 가져오고,
'./posts.service'에서 { PostsService }를 가져옵니다;
'./post.entity'에서 { Post } 가져오기;
'./user.entity'에서 { User } 가져오기;

@Resolver((of) => User)
export class UsersResolver {
  생성자(비공개 읽기 전용 postsService: PostsService) {}

  @ResolveField((of) => [Post])
```

```
  public posts(@Parent() user: User): Post[] {
    return this.postsService.forAuthor(user.id);
  }
}
```

또한 포스트 엔티티 클래스를 정의해야 합니다:

```
'@nestjs/graphql'에서 { Directive, Field, ID, Int, ObjectType } 가져오기;
'./user.entity'에서 { User } 가져오기;

객체 유형() @디렉티브('@키(필드:
"id")') 내보내기 클래스 Post {
  @Field((유형) => ID)
  id: 숫자;

  @Field()
  title: 문자열;

  @Field((유형) => Int)
  authorId: 숫자;

  @Field((type) => User)
  user?: 사용자;
}
```

그리고 그 해결사:

```
'@nestjs/graphql'에서 { Query, Args, ResolveField, Resolver, Parent
}를 임포트합니다;
'./posts.service'에서 { PostsService } 가져오기;
'./post.entity'에서 { Post } 가져오기;
'./user.entity'에서 { User }를 가져옵니다;

@Resolver((of) => Post)
export class PostsResolver {
  생성자(비공개 읽기 전용 postsService: PostsService) {}

  @Query((returns) => Post)
  findPost(@Args('id') id: number): Post {
    이.postsService.findOne(id)을 반환합니다;
  }

  @Query((returns) => [Post])
  getPosts(): Post[] {
    this.postsService.all()을 반환합니다;
  }

  @ResolveField((of) => User)
```

```
  user(@Parent() post: Post): any {
    반환 { _____유형명: '사용자', id: post.authorId };
  }
}
```

마지막으로 모듈에 함께 묶습니다. 스키마 빌드 옵션에서 `User가` 고아(외부) 유형임을 지정하는 것에 주목하세요.

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
}를 '@nestjs/mercurius'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;
'./posts.resolvers'에서 { PostsResolvers }를 가져오고,
'./users.resolvers'에서 { UsersResolvers }를 가져옵니다;
'./posts.service'에서 { PostsService } 가져오기; // 예제에는 포함되지 않았습니다.

모듈({ import:
  [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      autoSchemaFile: true,
      federationMetadata: true,
      buildSchemaOptions: {
        고아 유형: [사용자],
      },
    }),
  ],
  공급자: [게시자 해결자, 사용자 해결자, 게시자 서비스],
})
내보내기 클래스 AppModule {}
```

페더레이션 예제**:** 게이트웨이

게이트웨이는 엔드포인트 목록을 지정해야 하며 해당 스키마를 자동으로 검색합니다. 따라서 게이트웨이 서비스의 구현은 코드 접근 방식과 스키마 우선 접근 방식 모두 동일하게 유지됩니다.

```
import {
  MercuriusGatewayDriver,
  MercuriusGatewayDriverConfig,
}를 '@nestjs/mercurius'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;

모듈({ import:
  [
```

```
    GraphQLModule.forRoot<MercuriusGatewayDriverConfig>({
      driver: MercuriusGatewayDriver,
      게이트웨이: {
        services: [
          { name: 'users', url: 'http://user-service/graphql' },
          { name: 'posts', url: 'http://post-service/graphql' },
        ],
      },
    }),
  ],
})
내보내기 클래스 AppModule {}
```

연맹 2

아폴로 문서를 인용하자면, 페더레이션 2는 원래 아폴로 페더레이션(이 문서에서는 페더레이션 1이라고 함)에서 개발자 환경을 개선한 것으로, 대부분의 원래 슈퍼그래프와 하위 호환됩니다.

> 경고 Mercurius는 페더레이션 2를 완전히 지원하지 않습니다. 페더레이션 2를 지원하는 라이브러리 목록은 여기에서 확인할 수 있습니다.

다음 섹션에서는 이전 예제를 페더레이션 2로 업그레이드하겠습니다. 페더레이션

예제: 사용자

페더레이션 2의 한 가지 변경 사항은 엔티티에 원본 하위 그래프가 없으므로 쿼리를 확장할 필요가 없다는 것입니다. 를 더 이상 사용할 수 없습니다. 자세한 내용은 아폴로 페더레이션 2 문서에서 엔티티 주제를

참조하세요. 스키마 먼저

스키마에서 확장 키워드를 제거하기만 하면 됩니다.

```
type User @key(fields: "id") {
  id: ID!
  이름: 문자열!
}

유형 쿼리 {
  getUser(id: ID!): User
}
```

코드 우선

페더레이션 2를 사용하려면 `자동 스키마 파일` 옵션에서 페더레이션 버전을 지정해야 합니다.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
```

```
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'./users.resolver'에서 { UsersResolver }를 가져옵니다;
'./users.service'에서 { UsersService } import; // 이 예제에는 포함되지 않았습니
다.

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: { 페
        더레이션: 2,
      },
    }),
  ],
  제공자: [UsersResolver, UsersService],
})
내보내기 클래스 AppModule {}
```

페더레이션 예시**:** 글

위와 같은 이유로 사용자 및 쿼리를 더 이상 확장할 필요가 없습니다. 스키마 먼저

스키마에서 확장 지시문과 외부 지시문을 간단히 제거할 수 있습니다.

```
type Post @key(fields: "id") {
  id: ID!
  제목: String!
  body: 문자열! 사
  용자: 사용자
}

type User @key(fields: "id") {
  id: ID!
  게시물: [게시물]
}

유형 Query {
  getPosts: [Post]
}
```

코드 우선

사용자 엔터티를 더 이상 확장하지 않으므로 사용자에서 확장 및 외부 지시문을 제거하기만 하면 됩니다.

```
'@nestjs/graphql'에서 { 지시문, 객체 유형, 필드, ID } 가져오기;
'./post.entity'에서 { Post } 가져오기;

객체 유형() @디렉티브('@키(필드:
"id")') 내보내기 클래스 사용자 {
  @Field((유형) => ID)
  id: 숫자;

  @Field((type) => [Post])
  posts? Post[];
}
```

또한 사용자 서비스와 마찬가지로 `GraphQLModule에서` 페더레이션 2를 사용하도록 지정해야 합니다.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;
'./posts.resolvers'에서 { PostsResolvers }를 가져오고,
'./users.resolvers'에서 { UsersResolvers }를 가져옵니다;
'./posts.service'에서 { PostsService } 가져오기; // 예제에는 포함되지 않았습니다.

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: { 페
        더레이션: 2,
      },
      빌드 스키마 옵션: { 고아 유형: [사용자
        ],
      },
    }),
  ],
  공급자: [게시자 해결자, 사용자 해결자, 게시자 서비스],
})
내보내기 클래스 AppModule {}
```

v10에서 v11로 마이그레이션

이 장에서는 `@nestjs/graphql` 버전 10에서 버전 11로 마이그레이션하기 위한 일련의 지침을 제공합니다. 이번 주요 릴리스의 일환으로 Apollo 드라이버가 Apollo Server v4(v3 대신)와 호환되도록 업데이트되었습니다. 참고: Apollo Server v4에는 몇 가지 중요한 변경 사항(특히 플러그인 및 에코시스템 패키지 관련)이 있으므로 그에 따라 코드베이스를 업데이트해야 합니다. 자세한 내용은 Apollo Server v4 마이그레이션 가이드를 참조하세요.

아폴로 패키지

`apollo-server-express` 패키지를 설치하는 대신 `@apollo/server`를 설치해야 합니다:

```
$ npm 제거 아폴로 서버-익스프레스
npm 설치 @apollo/server
```

Fastify 어댑터를 사용하는 경우 `@as-integrations/fastify` 패키지를 대신 설치해야 합니다:

```
$ npm 제거 아폴로-서버-패스티파이
npm install @apollo/server @as-integrations/fastify
```

머큐리우스 패키지

머큐리우스 게이트웨이는 더 이상 `머큐리우스` 패키지의 일부가 아닙니다. 대신 `mercuriusjs/gateway` 패키지를 별도로 다운로드하세요:

```
$ npm 설치 @mercuriusjs/gateway
```

마찬가지로 페더레이션 스키마를 생성하려면 `@mercuriusjs/federation` 패키지를 설치해야 합니다:

```
$ npm 설치 @mercuriusjs/federation
```

v9에서 v10으로 마이그레이션

이 장에서는 `@nestjs/graphql` 버전 9에서 버전 10으로 마이그레이션하기 위한 일련의 지침을 제공합니다. 이번

메이저 버전 릴리즈의 초점은 플랫폼에 구애받지 않는 더 가벼운 코어 라이브러리를 제공하는 것입니다.

"드라이버" 패키지 소개

최신 버전에서는 `@nestjs/graphql` 패키지를 몇 개의 개별 라이브러리로 분리하여 프로젝트에서 Apollo(`@nestjs/apollo`), Mercurius(`@nestjs/mercurius`) 또는 다른 GraphQL 라이브러리를 사용할지 선택할 수 있도록 결정했습니다.

즉, 이제 애플리케이션에서 사용할 드라이버를 명시적으로 지정해야 합니다.

```
// 이전
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;

모듈({ import:
  [
    GraphQLModule.forRoot({
      autoSchemaFile: 'schema.gql',
    }),
  ],
})
내보내기 클래스 AppModule {}

// 이후
'@nestjs/apollo'에서 { ApolloDriver, ApolloDriverConfig }를 임포트하고
, '@nestjs/common'에서 { Module }을 임포트합니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      자동 스키마 파일: 'schema.gql',
    }),
  ],
})
내보내기 클래스 AppModule {}
```

플러그인

Apollo 서버 플러그인을 사용하면 특정 이벤트에 대한 응답으로 사용자 정의 작업을 수행할 수 있습니다. 이 기능은 Apollo 전용 기능이기 때문에 `@nestjs/graphql에서` 새로 생성된 `@nestjs/apollo` 패키지로 옮겼으므로 애플리케이션에서 임포트를 업데이트해야 합니다.

```
// 이전
'@nestjs/graphql'에서 { Plugin }을 가져옵니다;

// 이후
'@nestjs/apollo'에서 { Plugin }을 가져옵니다;
```

지시어

`schemaDirectives` 기능이 `@graphql- tools/schema` 패키지의 v8에서 새로운 Schema 지시문 API로 대체되었습니다.

```
```

```
// 이전
'@graphql-tools/utils'에서 { SchemaDirectiveVisitor }를 가져오고,
'graphql'에서 { defaultFieldResolver, GraphQLField }를 가져옵니다;

export class UpperCaseDirective extends SchemaDirectiveVisitor {
  visitFieldDefinition(field: GraphQLField<any, any>) {
    const { resolve = defaultFieldResolver } = field;
    field.resolve = async 함수 (...args) {
      const result = await resolve.apply(this, args);
      if (typeof result === 'string') {
        결과값을 반환합니다;
      }
      결과를 반환합니다;
    };
  }
}

// 이후
'@graphql- tools/utils'에서 { getDirective, MapperKind, mapSchema }를 가져
옵니다;
'graphql'에서 { defaultFieldResolver, GraphQLSchema }를 가져옵니다;

내보내기 함수 upperDirectiveTransformer( 스키마:
  GraphQLSchema,
  지시어 이름: 문자열,
) {
  return mapSchema(schema, {
    [MapperKind.OBJECT_FIELD]: (fieldConfig) => {
      const upperDirective = getDirective( 스
        키마,
        fieldConfig, 지시어 이
        름,
      )?.[0];

      if (upperDirective) {
        const { resolve = defaultFieldResolver } = fieldConfig;


통화


{                 }
              },
            });
          }
```

```
// 원래 리졸버를 *먼저* 다음과 같은 함수로 바꿉니다.
(source, args, context, info); if (typeof result === 'string') {
  결과값을 반환합니다;
}
결과를 반환합니다;
};
필드 컨피그를 반환합니다;

// 원래 리졸버를 호출한 다음 결과를 대문자로 변환합니다.
fieldConfig.resolve = async 함수(source, args, context, info).

  const result = await resolve
```

이 지시문 구현을 `@upper` 지시문이 포함된 스키마에 적용하려면

`변환 스키마` 함수입니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  ...
  transformSchema: schema => upperDirectiveTransformer(schema, 'upper'),
})
```

연맹

GraphQLFederationModule이 제거되고 해당 드라이버 클래스로 대체되었습니다:

```
// 이전에
GraphQLFederationModule.forRoot({
  autoSchemaFile: true,
});

// 이후 GraphQLModule.forRoot<ApolloFederationDriverConfig>({
  드라이버: ApolloFederationDriver,
  autoSchemaFile: true,
});
```

> 정보 힌트 `ApolloFederationDriver` 클래스와 ApolloFederationDriverConfig는 모두 `@nestjs/apollo` 패키지에서 내보냅니다.

마찬가지로 전용 `GraphQLGatewayModule을` 사용하는 대신 적절한 `드라이버를` 전달하기만 하면 됩니다.

클래스를 `GraphQLModule` 설정에 추가합니다:

```
// 이전에
GraphQLGatewayModule.forRoot({
  게이트웨이: {
    supergraphSdl: new IntrospectAndCompose({
      subgraphs: [
        { name: 'users', url: 'http://localhost:3000/graphql' },
        { name: 'posts', url: 'http://localhost:3001/graphql' },
      ],
    }),
  },
});

// 이후
GraphQLModule.forRoot<ApolloGatewayDriverConfig>({
  드라이버: 아폴로게이트웨이드라이버, 게이트웨
  이: {
    supergraphSdl: new IntrospectAndCompose({
```

```
        하위 그래프: [
          { name: 'users', url: 'http://localhost:3000/graphql' },
          { name: 'posts', url: 'http://localhost:3001/graphql' },
        ],
      }),
    },
  });
```

> 정보 힌트 `ApolloGatewayDriver` 클래스와 ApolloGatewayDriverConfig는 모두 `@nestjs/apollo` 패키지에서 내보냅니다.