서버리스

서버리스 컴퓨팅은 클라우드 제공업체가 고객을 대신하여 서버를 관리하면서 온디맨드 방식으로 머신 리소스를 할당하는 클라우드 컴퓨팅 실행 모델입니다. 앱을 사용하지 않을 때는 앱에 할당된 컴퓨팅 리소스가 없습니다. 가격은 애플리케이션(소스)이 소비하는 실제 리소스 양을 기준으로 책정됩니다.

서버리스 아키텍처를 사용하면 애플리케이션 코드의 개별 기능에만 집중할 수 있습니다. AWS Lambda, Google Cloud Functions, Microsoft Azure Functions와 같은 서비스가 모든 물리적 하드웨어, 가상 머신 운영 체제, 웹 서버 소프트웨어 관리를 처리합니다.

> 정보 힌트 이 장에서는 서버리스 기능의 장단점을 다루거나 클라우드 제공업체의 세부 사항을 자세히 다루지 않습니다.

콜드 스타트

콜드 스타트란 오랜만에 코드를 실행하는 것을 말합니다. 사용하는 클라우드 제공업체에 따라 코드를 다운로드하고 런타임을 부트스트랩하는 것부터 최종적으로 코드를 실행하는 것까지 여러 가지 작업이 포함될 수 있습니다. 이 프로세스는 언어, 애플리케이션에 필요한 패키지 수 등 여러 요인에 따라 상당한 지연 시간을 추가합니다.

콜드 스타트는 중요하며, 우리가 통제할 수 없는 상황도 있지만 가능한 한 짧게 만들기 위해 우리가 할 수 있는 일은 많습니다.

Nest는 복잡한 엔터프라이즈 애플리케이션에 사용하도록 설계된 완전한 프레임워크라고 생각할 수 있지만, 훨씬 "간단한" 애플리케이션(또는 스크립트)에도 적합합니다. 예를 들어, 독립형 애플리케이션 기능을 사용하면 간단한 워커, CRON 작업, CLI 또는 서버리스 기능에서 Nest의 DI 시스템을 활용할 수 있습니다.

벤치마크

서버리스 함수의 맥락에서 Nest 또는 기타 잘 알려진 라이브러리(예: `express`)를 사용하는 데 드는 비용을 더 잘 이해하기 위해 Node 런타임이 다음 스크립트를 실행하는 데 얼마나 많은 시간이 필요한지 비교해 보겠습니다:

```
// #1 Express
'express'에서 *를 express로 가져옵니다;

비동기 함수 부트스트랩() { const
  app = express();
  app.get('/', (req, res) => res.send('Hello world!'));
  await new Promise<void>((resolve) => app.listen(3000, resolve));
}
부트스트랩();

// #2 Nest (@nestjs/platform-express 사용)
'@nestjs/core'에서 { NestFactory } 임포트;
'./app.module'에서 { AppModule } 임포트;
```

```
비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule, { logger: ['error'] });
  await app.listen(3000);
}
부트스트랩();

// #3 Nest를 독립형 애플리케이션(HTTP 서버 없음)으로 사용하려면
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'./app.module'에서 { AppModule }을 가져오고,
'./app.service'에서 { AppService }를 가져옵니다;

비동기 함수 부트스트랩() {
  const app = await NestFactory.createApplicationContext(AppModule, {
    logger: ['error'],
  });
  console.log(app.get(AppService).getHello());
}
부트스트랩();

// #4 원시 Node.js 스크립트 비동
기 함수 bootstrap() {
  console.log('안녕하세요!');
}
부트스트랩();
```

이 모든 스크립트에는 `tsc`(TypeScript) 컴파일러를 사용했기 때문에 코드가 번들되지 않은 상태로 유지됩니다(`웹팩은` 사용되지 않음).

Express	0.0079초(7.9ms)
Nest(`@nestjs/platform-express` 포함)	0.1974초 (197.4ms)
Nest(독립 실행형 애플리케이션)	0.1117초 (111.7ms)
원시 Node.js 스크립트	0.0071초(7.1ms)

> 정보 노트 기계: MacBook Pro Mid 2014, 2.5GHz 쿼드코어 인텔 코어 i7, 16GB 1600MHz DDR3, SSD.

이제 모든 벤치마크를 반복하되 이번에는 웹팩(Nest CLI가 설치되어 있는 경우 `nest build --webpack`을 실행하

면 됩니다)을 사용하여 애플리케이션을 단일 실행 가능한 JavaScript 파일로 번들링해 보겠습니다. 하지만 Nest CLI와 함께 제공되는 기본 `웹팩` 구성을 사용하는 대신 다음과 같이 모든 종속 요소(`node_modules`)를 함께 번들로 묶어 보겠습니다:

```
module.exports = (옵션, 웹팩) => { const
  lazyImports = [
    '@nestjs/microservices/microservices-module',
    '@nestjs/websockets/socket-module',
  ];

  반환 {
```

```
      ...옵션,
      외부: [], 플러그인: [
        ...options.plugins,
        새로운 웹팩.무시 플러그인({
          checkResource(resource) {
            if (lazyImports.includes(resource)) {
              try {
                require.resolve(resource);
              } catch (err) {
                return true;
              }
            }
            거짓을 반환합니다;
          },
        }),
      ],
    };
  };
```

> 정보 힌트 Nest CLI가 이 구성을 사용하도록 지시하려면 프로젝트의 루트 디렉터리에 새 `webpack.config.js` 파일을 만드세요.

이 구성을 통해 다음과 같은 결과를 얻었습니다: Express

	0.0068초
(6.8ms)	
Nest(`@nestjs/platform-express` 포함)	0.0815초
(81.5ms) Nest(독립 실행형 애플리케이션)	0.0319초
(31.9ms)	
원시 Node.js 스크립트	0.0066초(6.6ms)

> 정보 노트 기계: MacBook Pro Mid 2014, 2.5GHz 쿼드코어 인텔 코어 i7, 16GB 1600MHz DDR3, SSD.

> 정보 힌트 추가 코드 최소화 및 최적화 기법(웹팩 플러그인 등 사용)을 적용하여 더욱 최적화할 수 있습니다.

보시다시피 컴파일 방식(그리고 코드 번들링 여부)은 매우 중요하며 전체 시작 시간에 큰 영향을 미칩니다. 웹팩을 사용하면 독립형 Nest 애플리케이션(하나의 모듈, 컨트롤러, 서비스가 포함된 스타터 프로젝트)의 부트스트랩 시간을 평균 32ms까지 단축할 수 있으며, 일반 HTTP, 익스프레스 기반 NestJS 앱의 경우 81.5ms까지 단축할

수 있습니다.

예를 들어 10개의 리소스(`$ nest g 리소스` 스키마 = 10개의 모듈, 10개의 컨트롤러, 10개의 서비스, 20개의 DTO 클래스, 50개의 HTTP 엔드포인트 + `AppModule을` 통해 생성됨)가 있는 더 복잡한 Nest 애플리케이션의 경우, MacBook Pro Mid 2014, 2.5GHz 쿼드 코어 인텔 코어 i7, 16GB 1600MHz DDR3, SSD에서 전체 시작은 약 0.1298초(129.8ms)입니다. 모놀리식 애플리케이션을 서버리스 기능으로 실행하는 것은 일반적으로 큰 의미가 없으므로 이 벤치마크는 애플리케이션이 성장함에 따라 부트스트랩 시간이 어떻게 증가할 수 있는지를 보여주는 예시라고 생각하면 됩니다.

런타임 최적화

지금까지 컴파일 시간 최적화에 대해 살펴보았습니다. 이는 애플리케이션에서 프로바이더를 정의하고 네스트 모듈을 로드하는 방식과 무관하지 않으며, 애플리케이션이 커질수록 필수적인 역할을 합니다.

예를 들어 데이터베이스 연결이 비동기 공급자로 정의되어 있다고 가정해 보겠습니다. 비동기 공급자는 하나 이상의 비동기 작업이 완료될 때까지 애플리케이션 시작을 지연하도록 설계되었습니다. 즉, 서버리스 함수가 데이터베이스에 연결하는 데 평균 2초가 걸리는 경우(부트스트랩에서), 엔드포인트는 (콜드 스타트이고 애플리케이션이 아직 실행 중이 아닌 경우) 응답을 다시 보내는 데 최소 2초(연결이 설정될 때까지 기다려야 하므로)가 더 필요합니다.

보시다시피, 부트스트랩 시간이 중요한 서버리스 환경에서는 프로바이더를 구성하는 방식이 다소 달라집니다. 또 다른 좋은 예는 특정 시나리오에서만 캐싱을 위해 Redis를 사용하는 경우입니다. 이 경우, 특정 함수 호출에 필요하지 않더라도 부트스트랩 시간이 느려질 수 있으므로 Redis 연결을 비동기 공급자로 정의해서는 안 됩니다.

또한 이 장에서 설명한 대로 `LazyModuleLoader` 클래스를 사용하여 전체 모듈을 지연 로드할 수도 있습니다. 여기에서도 캐싱이 좋은 예입니다. 애플리케이션에 내부적으로 Redis에 연결하고 또한 `CacheService`를 내보내 Redis 스토리지와 상호 작용하는 `CacheModule`이 있다고 가정해 보겠습니다. 모든 잠재적 함수 호출에 필요하지 않은 경우, 온디맨드 방식으로 느리게 로드하면 됩니다. 이렇게 하면 캐싱이 필요하지 않은 모든 호출에 대해 시작 시간(콜드 스타트 발생 시)을 단축할 수 있습니다.

```
if (request.method === RequestMethod[RequestMethod.GET]) {
  const { CacheModule } = await import('./cache.module');
  const moduleRef = await this.lazyModuleLoader.load(() => CacheModule);

  const { CacheService } = await import('./cache.service');
  const cacheService = moduleRef.get(CacheService);

  cacheService.get(ENDPOINT_KEY)를 반환합니다;
}
```

또 다른 좋은 예는 특정 조건(예: 입력 인수)에 따라 다른 작업을 수행할 수 있는 웹훅 또는 워커입니다. 이러한 경우 라우트 핸들러 내부에 특정 함수 호출에 적합한 모듈을 느리게 로드하는 조건을 지정하고 다른 모든 모듈을 느리게 로드할 수 있습니다.

```
if (workerType === WorkerType.A) {
  const { WorkerAModule } = await import('./worker-a.module');
  const moduleRef = await this.lazyModuleLoader.load(() => WorkerAModule);
  // ...
} else if (workerType === WorkerType.B) {
  const { WorkerBModule } = await import('./worker-b.module');
  const moduleRef = await this.lazyModuleLoader.load(() => WorkerBModule);
  // ...
}
```

통합 예시

애플리케이션의 엔트리 파일(일반적으로 `main.ts` 파일)의 모양은 여러 요인에 따라 달라지므로 모든 시나리오에 적합한 단일 템플릿은 없습니다. 예를 들어, 서버리스 기능을 가동하는 데 필요한 초기화 파일은 클라우드 제공업체(AWS, Azure, GCP 등)에 따라 다릅니다. 또한 여러 경로/엔드포인트로 일반적인 HTTP 애플리케이션을 실행할지, 단일 경로만 제공할지(또는 코드의 특정 부분만 실행할지) 여부에 따라 애플리케이션 코드의 모양이 달라집니다(예를 들어, 함수별 엔드포인트 접근 방식의 경우 HTTP 서버 부팅, 미들웨어 설정 등 대신 `NestFactory.createApplicationContext를` 사용할 수 있습니다.).

예시를 보여드리기 위해 Nest(@nestjs/platform-express를 사용하여 전체 기능을 갖춘 HTTP 라우터 전체를 스핀업)를 서버리스 프레임워크(이 경우 AWS Lambda를 대상으로 함)와 통합해 보겠습니다. 앞서 언급했듯이 선택한 클라우드 제공업체와 기타 여러 요인에 따라 코드가 달라집니다.

먼저 필요한 패키지를 설치해 보겠습니다:

```
$ npm i @vendia/serverless-express aws-lambda
$ npm i -D @타입스/aws-lambda 서버리스-오프라인
```

> 정보 힌트 개발 주기를 단축하기 위해 AWS λ 및 API 게이트웨이를 에뮬레이트하는 `서버리스-오프라인` 플러그인을 설치합니다.

설치 프로세스가 완료되면 서버리스 프레임워크를 구성하기 위해 serverless`.yml` 파일을 생성해 보겠습니다:

```
서비스: 서버리스 예제 플러그인:
  - 서버리스 오프라인

공급자: 이름:
  AWS
  런타임: nodejs14.x

기능: 메인:
    핸들러: dist/main.handler 이벤트:
      - http:
          메서드: 모든 경
          로: /
      - http:
          메서드를 사용합니다:
          모든 경로: '{프록시
          +}'
```

> 정보 힌트 서버리스 프레임워크에 대해 자세히 알아보려면 공식 문서를 참조하세요.

이제 `main.ts` 파일로 이동하여 필요한 상용구로 부트스트랩 코드를 업데이트할 수 있습니다:

```
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@vendia/serverless-express'에서 서버리스익스프레스 임포트; 'aws-
lambda'에서 { 콜백, 컨텍스트, 핸들러 } 임포트; './app.module'에서 {
앱모듈 } 임포트;

렛 서버: 핸들러;

비동기 함수 부트스트랩(): Promise<Handler> { const app
  = await NestFactory.create(AppModule); await
  app.init();

  const expressApp = app.getHttpAdapter().getInstance();
  return serverlessExpress({ app: expressApp });
}

const 핸들러를 내보냅니다: 핸들러 = async ( 이벤트:
  any,
  컨텍스트: 컨텍스트, 콜백:
  콜백,
) => {
  서버 = 서버 ?? (await bootstrap()); 반환 서버(
  이벤트, 컨텍스트, 콜백);
};
```

> 정보 힌트 여러 서버리스 함수를 생성하고 이들 간에 공통 모듈을 공유하려면 CLI 모노레포 모드를 사용하는 것이 좋습니다.

> 경고 경고 `@nestjs/swagger` 패키지를 사용하는 경우 서버리스 기능의 맥락에서 제대로 작동하도록 하기 위해 몇 가지 추가 단계가 필요합니다. 자세한 내용은 이 스레드를 확인하세요.

그런 다음 `tsconfig.json` 파일을 열고 `esModuleInterop` 옵션을 활성화하여 `@vendia/serverless-express` 패키지가 제대로 로드되도록 합니다.

```
{
  "컴파일러옵션": {
    ...
    "esModuleInterop": true
  }
}
```

이제 애플리케이션을 빌드하고(네스트 `빌드` 또는 `tsc 사용) 서버리스` CLI를 사용하여 람다 함수를 로컬에서 시작할 수 있습니다:

```
$ npm 실행 빌드
npx 서버리스 오프라인
```

애플리케이션이 실행 중이면 브라우저를 열고 `http://localhost:3000/dev/[ANY_ROUTE]`로 이동합니다(여기서 `[ANY_ROUTE]`는 애플리케이션에 등록된 모든 엔드포인트).

위 섹션에서는 웹팩을 사용하여 앱을 번들링하면 전체 부트스트랩 시간에 상당한 영향을 미칠 수 있음을 보여주었습니다. 하지만 이 예제에서 작동하도록 하려면 `webpack.config.js` 파일에 몇 가지 추가 구성을 추가해야 합니다. 일반적으로 `핸들러` 함수가 선택되도록 하려면 `출력.libraryTarget` 속성을 `commonjs2로` 변경해야 합니다.

```
반환 {
  ...옵션,
  외부: [], 출력: {
    ...options.output,
    libraryTarget: 'commonjs2',
  },
  // ... 나머지 구성
};
```

이제 `$ nest build` --webpack을 사용하여 함수의 코드를 컴파일할 수 있습니다(그런 다음 `npx 서버리스` 오프라인으로 테스트합니다).

또한 프로덕션 빌드를 축소할 때 클래스명을 그대로 유지하기 위해 `terser- webpack-plugin` 패키지를 설치하고 해당 구성을 재정의하는 것이 좋습니다(빌드 프로세스가 느려지므로 필수는 아님). 그렇게 하지 않으면 애플리케이션 내에서 `클래스 유효성 검사기를` 사용할 때 잘못된 동작이 발생할 수 있습니다.

```
const TerserPlugin = require('terser-webpack-plugin');

return {
  ...옵션,
  외부: [], 최적화: {
    최소화합니다: [
      새로운
        TerserPlugin({
        terserOptions: {
          keep_classnames: true,
        },
      }),
    ],
  },
  출력합니다: {
    ...options.output,
    libraryTarget: 'commonjs2',
```

```
  },
  // ... 나머지 구성
};
```

독립 실행형 애플리케이션 기능 사용

또는 함수를 매우 가볍게 유지하고 HTTP 관련 기능(라우팅뿐만 아니라 가드, 인터셉터, 파이프 등)이 필요하지 않은 경우, 다음과 같이 전체 HTTP 서버를 실행하는 대신 `NestFactory.createApplicationContext`(앞서 언급한 대로)를 사용할 수 있습니다(그리고 내부적으로 `표현):`

```
@@파일명(메인)
'@nestjs/common'에서 { HttpStatus }를 임포트하고
, '@nestjs/core'에서 { NestFactory }를 임포트합
니다;
'aws-lambda'에서 { Callback, Context, Handler }를 가져오고,
'./app.module'에서 { AppModule }을 가져옵니다;
'./app.service'에서 { AppService }를 가져옵니다;

const 핸들러를 내보냅니다: 핸들러 = 비동기 ( 이벤트:
  any,
  컨텍스트: 컨텍스트, 콜백:
  콜백,
) => {
  const appContext = await
NestFactory.createApplicationContext(AppModule);
  const appService = appContext.get(AppService);

  반환 {
    body: appService.getHello(),
    statusCode: HttpStatus.OK,
  };
};
```

> 정보 힌트 NestFactory.createApplicationContext는 컨트롤러 메서드를 인핸서(가드, 인터셉터 등)로 래핑하지 않는다는 점에 유의하세요. 이를 위해서는 `NestFactory.create` 메서드를 사용해야 합니다.

또한 `이벤트` 객체를 처리하고 입력 값과 비즈니스 로직에 따라 해당 값을 반환할 수 있는 `이벤트 서비스` 공급자에게 이벤트 객체를 전달할 수도 있습니다.

```
const 핸들러를 내보냅니다: 핸들러 = 비동기 ( 이벤트:
  any,
  컨텍스트: 컨텍스트, 콜백:
  콜백,
) => {
  const appContext = await
NestFactory.createApplicationContext(AppModule);
  const eventsService = appContext.get(EventsService);
```

```
  반환 이벤트서비스.프로세스(이벤트);
};
```

HTTP 어댑터

Nest 애플리케이션 컨텍스트 내에서 또는 외부에서 기본 **HTTP** 서버에 액세스해야 하는 경우가 있습니다.

모든 네이티브(플랫폼별) **HTTP** 서버/라이브러리(예: Express 및 Fastify) 인스턴스는 어댑터로 래핑됩니다. 어댑터는 애플리케이션 컨텍스트에서 검색할 수 있고 다른 제공업체에 삽입할 수 있는 전역적으로 사용 가능한 제공업체로 등록됩니다.

외부 애플리케이션 컨텍스트 전략

애플리케이션 컨텍스트 외부에서 HttpAdapter에 대한 참조를 가져 오려면

메서드를 호출합니다.

```
@@파일명()
const app = await NestFactory.create(AppModule);
const httpAdapter = app.getHttpAdapter();
```

상황에 맞는 전략

애플리케이션 컨텍스트 내에서 HttpAdapterHost에 대한 참조를 얻으려면 다른 기존 공급자와 동일한 기술(예: 생성자 주입 사용)을 사용하여 주입합니다.

```
@@파일명()
내보내기 클래스 CatsService {
  constructor(private adapterHost: HttpAdapterHost) {}
}
@@스위치 @디펜던시(HttpAdapterHost) 내보
내기 클래스 CatsService {
  constructor(adapterHost) {
    this.adapterHost = adapterHost;
  }
}
```

> 정보 힌트 HttpAdapterHost는 @nestjs/core 패키지에서 가져옵니다.

HttpAdapterHost는 실제 HttpAdapter가 아닙니다. 실제 HttpAdapter 인스턴스를 가져오려면 httpAdapter 프로퍼티에 액세스하면 됩니다.

```
const adapterHost = app.get(HttpAdapterHost);
const httpAdapter = adapterHost.httpAdapter;
```

`httpAdapter`는 기본 프레임워크에서 사용하는 HTTP 어댑터의 실제 인스턴스입니다. `ExpressAdapter` 또는 `FastifyAdapter`의 인스턴스입니다(두 클래스 모두 `AbstractHttpAdapter`를 확장합니다).

어댑터 객체는 HTTP 서버와 상호 작용할 수 있는 몇 가지 유용한 메서드를 노출합니다. 그러나 라이브러리 인스턴스(예: Express 인스턴스)에 직접 액세스하려면 `getInstance()` 메서드를 호출하세요.

```
const 인스턴스 = httpAdapter.getInstance();
```

글로벌 접두사

HTTP 애플리케이션에 등록된 모든 경로에 접두사를 설정하려면 `INestApplication` 인스턴스의 `setGlobalPrefix()` 메서드를 사용합니다.

```
const app = await NestFactory.create(AppModule);
app.setGlobalPrefix('v1');
```

다음 구문을 사용하여 글로벌 접두사에서 경로를 제외할 수 있습니다:

```
app.setGlobalPrefix('v1', {
  제외: [{ 경로: 'health', 메서드: RequestMethod.GET }],
});
```

또는 경로를 문자열로 지정할 수도 있습니다(모든 요청 메서드에 적용됨):

```
app.setGlobalPrefix('v1', { 제외: ['cats'] });
```

> 정보 힌트 `경로` 속성은 경로 정규식 패키지를 사용하여 와일드카드 매개변수를 지원합니다. 참고: 와일드카드 별표 `*를 사용할` 수 없습니다. 대신 매개변수(예: `(.*)`, `:splat*`)를 사용해야 합니다.

생체

원시 요청 본문에 액세스하는 가장 일반적인 사용 사례 중 하나는 웹훅 서명 검증을 수행하는 것입니다. 일반적으로 웹훅 서명 유효성 검사를 수행하려면 직렬화되지 않은 요청 본문이 HMAC 해시를 계산하는 데 필요합니다.

> 경고 경고 이 기능은 내장된 글로벌 본문 구문 분석기 미들웨어가 활성화된 경우에만 사용할 수 있습니다. 즉, 앱을 만들 때 `bodyParser: false`를 전달하지 않아야 합니다.

Express와 함께 사용

먼저 Nest Express 애플리케이션을 만들 때 이 옵션을 활성화합니다:

```
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/platform-express'에서 { NestExpressApplication } 유형 가져오기;
'./app.module'에서 { AppModule } 가져오기;

// "부트스트랩" 함수에서
const app = await NestFactory.create<NestExpressApplication>(AppModule, {
  rawBody: true,
});
await app.listen(3000);
```

컨트롤러에서 원시 요청 본문에 액세스하려면 요청에 `원시 요청` 필드를 노출하기 위한 편의 인터페이스 RawBodyRequest가 제공됩니다(인터페이스 `RawBodyRequest` 유형 사용):

```
'@nestjs/common'에서 { Controller, Post, RawBodyRequest, Req }를 임포트하고,
'express'에서 { Request }를 임포트합니다;

@Controller('cats') 클
래스 CatsController {
  @Post()
  create(@Req() req: RawBodyRequest<Request>) {
    const raw = req.rawBody; // `버퍼`를 반환합니다.
  }
}
```

다른 구문 분석기 등록하기

기본적으로 `json` 및 `urlencoded` 구문 분석기만 등록되어 있습니다. 다른 구문 분석기를 즉시 등록하려면 명시적으로 등록해야 합니다.

예를 들어 `텍스트` 구문 분석기를 등록하려면 다음 코드를 사용할 수 있습니다:

```
app.useBodyParser('text');
```

> 경고 `NestFactory.create` 호출에 올바른 애플리케이션 유형을 제공하고 있는지 확인합니다. Express 애플리케이션의 경우 올바른 유형은 `NestExpressApplication`입니다. 그렇지 않으면 `.useBodyParser` 메서드를 찾을 수 없습니다.

본문 구문 분석기 크기 제한

애플리케이션에서 Express의 기본 `100KB보다` 큰 본문을 구문 분석해야 하는 경우 다음을 사용하세요:

```
app.useBodyParser('json', { limit: '10MB' });
```

`.useBodyParser` 메서드는 애플리케이션 옵션에서 전달되는 `rawBody` 옵션을 존중합니다. Fastify와 함께

사용

먼저 Nest Fastify 애플리케이션을 생성할 때 옵션을 활성화합니다:

```
'@nestjs/core'에서 { NestFactory }를 가져옵니다.
  FastifyAdapter,
  NestFastifyApplication,
}를 '@nestjs/platform-fastify'에서 가져옵니다;
'./app.module'에서 { AppModule }을 가져옵니다;

// "부트스트랩" 함수에서
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  새로운 FastifyAdapter(),
  {
    rawBody: true,
  },
);
await app.listen(3000);
```

컨트롤러에서 원시 요청 본문에 액세스하려면 요청에 `원시 요청` 필드를 노출하기 위한 편의 인터페이스 RawBodyRequest가 제공됩니다(인터페이스 `RawBodyRequest` 유형 사용):

```
'@nestjs/common'에서 { Controller, Post, RawBodyRequest, Req }를 임포트하고,
'fastify'에서 { FastifyRequest }를 임포트합니다;

@Controller('cats') 클
래스 CatsController {
  @Post()
  create(@Req() req: RawBodyRequest<FastifyRequest>) {
    const raw = req.rawBody; // `버퍼`를 반환합니다.
  }
}
```

다른 구문 분석기 등록하기

기본적으로 `application/json` 및 `application/x-www-form-urlencoded` 구문 분석기만 등록되어 있습니다. 다른 구문 분석기를 즉시 등록하려면 명시적으로 등록해야 합니다.

예를 들어 `텍스트/일반` 구문 분석기를 등록하려면 다음 코드를 사용할 수 있습니다:

```
app.useBodyParser('text/plain');
```

> 경고 `NestFactory.create` 호출에 올바른 애플리케이션 유형을 제공하고 있는지 확인합니다. Fastify 애플리케이션의 경우 올바른 유형은 `NestFastifyApplication`입니다. 그렇지 않으면 `.useBodyParser` 메서드를 찾을 수 없습니다.

본문 구문 분석기 크기 제한

애플리케이션에서 Fastify의 기본 1MiB보다 큰 본문을 구문 분석해야 하는 경우 다음을 사용하세요:

```
const bodyLimit = 10_485_760; // 10MiB
app.useBodyParser('application/json', { bodyLimit });
```

`.useBodyParser` 메서드는 애플리케이션 옵션에서 전달된 `rawBody` 옵션을 존중합니다.

하이브리드 애플리케이션

하이브리드 애플리케이션은 HTTP 요청을 수신할 뿐만 아니라 연결된 마이크로서비스를 사용하는 애플리케이션입니다. `INestApplication` 인스턴스는 `connectMicroservice()` 메서드를 통해 `INestMicroservice` 인스턴스와 연결할 수 있습니다.

```
const app = await NestFactory.create(AppModule);
const microservice = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.TCP,
});

await app.startAllMicroservices();
await app.listen(3001);
```

여러 마이크로서비스 인스턴스를 연결하려면 각 마이크로서비스에 대해 `connectMicroservice()`를 호출합니다:

```
const app = await NestFactory.create(AppModule);
// 마이크로서비스 #1
const microserviceTcp = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.TCP,
  옵션: { port:
    3001,
  },
});
// 마이크로서비스 #2
const microserviceRedis = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.REDIS,
  옵션: {
    호스트: 'localhost',
    포트: 6379,
  },
});

await app.startAllMicroservices();
await app.listen(3001);
```

여러 마이크로서비스가 있는 하이브리드 애플리케이션에서 `@MessagePattern()`을 하나의 전송 전략(예: **MQTT**)에만 바인딩하려면, 모든 기본 제공 전송 전략이 정의된 열거형인 `Transport` 유형의 두 번째 인수를 전달하면 됩니다.

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

```
  메시지 패턴({ cmd: 'time.us' }, Transport.TCP)
  getTCPDate(@Payload() 데이터: 숫자[]) {
    새 Date().toLocaleTimeString(...)을 반환합니다;
  }
  @@switch
  @Bind(Payload(), Ctx())
  @MessagePattern('time.us.*', Transport.NATS)
  getDate(data, context) {
    console.log(`Subject: ${context.getSubject()}`); // 예: "time.us.east"
    return new Date().toLocaleTimeString(...);
  }
  @Bind(Payload(), Ctx())
  메시지 패턴({ cmd: 'time.us' }, Transport.TCP) getTCPDate(데이
  터, 컨텍스트) {
    새 Date().toLocaleTimeString(...)을 반환합니다;
  }
```

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

구성 공유

기본적으로 하이브리드 애플리케이션은 기본(HTTP 기반) 애플리케이션에 대해 구성된 전역 파이프, 인터셉터, 가드 및 필터를 상속하지 않습니다. 주 애플리케이션에서 이러한 구성 속성을 상속하려면 다음과 같이 `connectMicroservice()` 호출의 두 번째 인수(선택적 옵션 개체)에 `inheritAppConfig` 속성을 설정하세요:

```
const microservice = app.connectMicroservice<MicroserviceOptions>(
  {
    전송: Transport.TCP,
  },
  { inheritAppConfig: true },
);
```

HTTPS

HTTPS 프로토콜을 사용하는 애플리케이션을 만들려면 `NestFactory` 클래스의 `create()` 메서드에 전달된 옵션 객체에서 `httpsOptions` 속성을 설정합니다:

```
const httpsOptions = {
  키: fs.readFileSync('./secrets/private-key.pem'),
  인증: fs.readFileSync('./secrets/public-certificate.pem'),
};
const app = await NestFactory.create(AppModule, {
  httpsOptions,
});
await app.listen(3000);
```

FastifyAdapter를 사용하는 경우 다음과 같이 애플리케이션을 생성합니다:

```
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  새로운 FastifyAdapter({ https: httpsOptions }),
);
```

여러 대의 동시 서버

다음 레시피는 여러 포트(예: 비-HTTPS 포트와 HTTPS 포트)에서 동시에 수신 대기하는 Nest 애플리케이션을 인스턴스화하는 방법을 보여줍니다.

```
const httpsOptions = {
  키: fs.readFileSync('./secrets/private-key.pem'),
  인증: fs.readFileSync('./secrets/public-certificate.pem'),
};

const server = express();
const app = await NestFactory.create(
  AppModule,
  새로운 ExpressAdapter(서버),
);
await app.init();

const httpServer = http.createServer(server).listen(3000);
const httpsServer = https.createServer(httpsOptions, server).listen(443);
```

`http.createServer` / `https.createServer`를 직접 호출했기 때문에 NestJS는 `app.close` / on 종료 신호를

호출할 때 이를 닫지 않습니다. 이 작업을 직접 수행해야 합니다:

```
@Injectable()
내보내기 클래스 ShutdownObserver는 OnApplicationShutdown을 구현합니다 {
  private httpServers: http.Server[] = [];

  public addHttpServer(server: http.Server): void {
    this.httpServers.push(server);
  }

  공용 비동기 onApplicationShutdown(): Promise<void> { await
    Promise.all(
      this.httpServers.map((server) => new
        Promise((resolve, reject) => {
          server.close((error) => {
            if (error) {
              reject(error);
            } else {
              resolve(null);
            }
          });
        })
      ),
    );
  }
}

const shutdownObserver = app.get(ShutdownObserver);
shutdownObserver.addHttpServer(httpServer);
shutdownObserver.addHttpServer(httpsServer);
```

> 정보 힌트 ExpressAdapter는 `@nestjs/platform-express` 패키지에서 가져옵니다. 이 패키지의 `http` 및 `https` 패키지는 네이티브 Node.js 패키지입니다.

> 경고 이 레시피는 GraphQL 구독에서는 작동하지 않습니다.

요청 수명 주기

Nest 애플리케이션은 요청 수명 주기라고 하는 시퀀스에 따라 요청을 처리하고 응답을 생성합니다. 미들웨어, 파이프, 가드, 인터셉터를 사용하면 특히 글로벌, 컨트롤러 수준, 경로 수준 구성 요소가 작용하기 때문에 요청 수명 주기 동안 특정 코드가 실행되는 위치를 추적하기가 어려울 수 있습니다. 일반적으로 요청은 미들웨어를 통해 가드, 인터셉터, 파이프로 이동한 다음 응답이 생성될 때 반환 경로에서 다시 인터셉터로 이동합니다.

미들웨어

미들웨어는 특정 순서로 실행됩니다. 먼저 Nest는 전역으로 바인딩된 미들웨어(예: `app.use로` 바인딩된 미들웨어)를 실행한 다음 경로에 따라 결정되는 모듈 바인딩된 미들웨어를 실행합니다. 미들웨어는 바인딩된 순서대로 순차적으로 실행되며, 이는 Express의 미들웨어가 작동하는 방식과 유사합니다. 여러 모듈에 바인딩된 미들웨어의 경우 루트 모듈에 바인딩된 미들웨어가 먼저 실행되고, 그 다음 모듈이 import 배열에 추가되는 순서대로 미들웨어가 실행됩니다.

경비병

가드 실행은 글로벌 가드부터 시작하여 컨트롤러 가드, 라우팅 가드 순으로 진행됩니다. 미들웨어와 마찬가지로 가드는 바인딩된 순서대로 실행됩니다. 예를 들어

```
UseGuards(Guard1, Guard2)
@Controller('cats')
내보내기 클래스 CatsController {
  constructor(private catsService: CatsService) {}

  사용가드(Guard3)
  @Get()
  getCats(): Cats[] {
    this.catsService.getCats()를 반환합니다;
  }
}
```

가드1은 `가드2보다` 먼저 실행되며, 둘 다 `가드3보다` 먼저 실행됩니다.

> 정보 힌트 전역 바인딩과 컨트롤러 또는 로컬 바인딩에 대해 말할 때 가드(또는 다른 컴포넌트)가 바인딩되는 위치에 차이가 있습니다. `app.useGlobalGuard()`를 사용하거나 모듈을 통해 컴포넌트를 제공하는 경우 전역적으로 바인딩됩니다. 그렇지 않으면 데코레이터가 컨트롤러 클래스 앞에 오는 경우 컨트롤러에 바인딩되고, 데코레이터가 경로 선언을 진행하는 경우 경로에 바인딩됩니다.

인터셉터

인터셉터는 대부분 가드와 동일한 패턴을 따르지만, 한 가지 예외가 있습니다. 인터셉터가 RxJS Observables를 반환할 때, 관찰 가능 항목은 선입선출 방식으로 해결됩니다. 따라서 인바운드 요청은 표준 글로벌, 컨트롤러, 라우트 수준 확인을 거치지만, 요청의 응답 측(즉, 컨트롤러 메서드 핸들러에서 반환된 후)은 라우트에서 컨트롤러로, 글로벌로 확인됩니다. 또한,

파이프, 컨트롤러 또는 서비스에서 발생하는 모든 오류는 인터셉터의 `catchError` 연산자에서 읽을 수 있습니다
.

파이프

파이프는 표준 글로벌-컨트롤러-경로 바인딩 순서를 따르며, `@UsePipes()` 매개변수와 관련하여 동일한 선입선출 방식을 따릅니다. 그러나 경로 매개변수 수준에서는 여러 개의 파이프가 실행 중인 경우 마지막 매개변수에서 첫 번째 매개변수까지 순서대로 실행됩니다. 이는 경로 수준 및 컨트롤러 수준 파이프에도 적용됩니다. 예를 들어 다음과 같은 컨트롤러가 있다고 가정해 보겠습니다:

```
사용파이프(일반검증파이프) @컨트롤러('cats')
내보내기 클래스 CatsController {
  constructor(private catsService: CatsService) {}

  사용파이프(경로특정파이프) @패치(':id')
  업데이트캣(
    Body() 본문: UpdateCatDTO, @Param()
    매개변수: UpdateCatParams, @Query()
    쿼리: UpdateCatQuery,
  ) {
    이.catsService.updateCat(body, params, query)을 반환합니다;
  }
}
```

로 설정하면 `쿼리`, `매개변수`, `본문` 객체에 대해 GeneralValidationPipe가 실행된 후 동일한 순서를 따르는 `RouteSpecificPipe로` 이동합니다. 매개변수별 파이프가 있는 경우 컨트롤러 및 경로 수준 파이프 다음에 매개변수별 파이프가 실행됩니다(다시 말해서 마지막 매개변수부터 첫 번째 매개변수까지).

필터

필터는 전역 먼저 해결하지 않는 유일한 구성 요소입니다. 대신 필터는 가능한 가장 낮은 수준부터 해결하므로 실행은 경로 바인딩 필터에서 시작하여 컨트롤러 수준, 그리고 마지막으로 전역 필터로 진행됩니다. 예외는 필터 간에 전달될 수 없으며, 경로 수준 필터가 예외를 포착하면 컨트롤러 또는 전역 수준 필터는 동일한 예외를 포착할 수 없습니다. 이와 같은 효과를 얻을 수 있는 유일한 방법은 필터 간에 상속을 사용하는 것입니다.

> 정보 힌트 필터는 요청 프로세스 중에 잡히지 않은 예외가 발생하는 경우에만 실행됩니다. 시도/잡기로 잡힌 예외와 같이 잡힌 예외는 예외 필터가 실행되도록 트리거되지 않습니다. 잡히지 않은 예외가 발생하면 나머지 라이프사이클은 무시되고 요청은 바로 필터로 건너뛰게 됩니다.

요약

일반적으로 요청 라이프사이클은 다음과 같습니다:

. 수신 요청

. 미들웨어

- 2.1. 글로벌 바인딩 미들웨어
- 2.2. 모듈 바인딩 미들웨어

. Guards

- 3.1 글로벌 가드
- 3.2 컨트롤러 가드
- 3.3 경로 가드

. 인터셉터(사전 컨트롤러)

- 4.1 글로벌 인터셉터
- 4.2 컨트롤러 인터셉터
- 4.3 경로 인터셉터

. 파이
프

- 5.1 글로벌 파이프
- 5.2 컨트롤러 파이프
- 5.3 파이프 라우팅
- 5.4 매개변수 파이프 라우팅

. 컨트롤러(메서드 핸들러)

. 서비스(있는 경우)

. 인터셉터(요청 후)

- 8.1 경로 인터셉터
- 8.2 컨트롤러 인터셉터
- 8.3 글로벌 인터셉터

. 예외 필터

- 9.1 경로
- 9.2 컨트롤러
- 9.3 글로벌

. 서버 응답

일반적인 오류

NestJS로 개발하는 동안 프레임워크를 익히면서 다양한 오류가 발생할 수 있습니다. "종속성을 해결할 수 없음" 오류

> 정보 힌트 "종속성을 해결할 수 없음" 오류를 쉽게 해결하는 데 도움이 되는 NestJS 개발자 도구를 확인하세요.

아마도 가장 일반적인 오류 메시지는 Nest가 공급자의 종속성을 해결할 수 없다는 것입니다. 오류 메시지는 일반적으로 다음과 같이 표시됩니다:

```
Nest가 <제공자>(?)의 종속성을 해결할 수 없습니다. 인덱스 [<인덱스>]의 인수
<unknown_token>을 사용할 수 있는지 확인하세요.
<모듈> 컨텍스트.

잠재적인 솔루션:
- <모듈>이 유효한 NestJS 모듈인가요?
- <unknown_token>이 공급자라면 현재 <모듈>의 일부인가요?
- <unknown_token>이 별도의 @Module에서 내보낸 경우 해당 모듈을 <module> 내에서 가
져올 수 있나요?
  모듈({
    임포트합니다: [ /* <unknown_token>이 포함된 모듈 */ ]입니다.
  })
```

오류의 가장 일반적인 원인은 모듈의 `공급자` 배열에 `<공급자>가` 없는 경우입니다. 공급자가 실제로 `공급자` 배열에 있고 표준 NestJS 공급자 관행을 따르고 있는지 확인하세요.

몇 가지 일반적인 문제가 있습니다. 하나는 `import` 배열에 프로바이더를 넣는 경우입니다. 이 경우 오류에 `<module>이` 있어야 할 위치에 공급자 이름이 표시됩니다.

개발 중에 이 오류가 발생하면 오류 메시지에 언급된 모듈을 살펴보고 해당 모듈의 `공급자를` 살펴보세요. `공급자` 배열의 각 공급자에 대해 모듈이 모든 종속성에 액세스할 수 있는지 확인하세요. 종종 `프로바이더가` "기능 모듈"과 "루트 모듈"에 중복되는 경우가 있는데, 이는 Nest가 프로바이더를 두 번 인스턴스화하려고 시도한다는 의미입니다. 대부분의 경우 <공급자>가 복제되는 대신 "루트 모듈"의 `가져오기` 배열에 추가되어야 합니다.

위의 `<unknown_token>이` 문자열 `종속성인` 경우 순환 파일 가져오기가 있을 수 있습니다. 이는 생성자에서 프로바이더가 서로 종속되는 대신 두 파일이 서로를 임포트하는 것을 의미하기 때문에 아래의 순환 종속성과는 다릅니다. 일반적인 경우는 모듈 파일에서 토큰을 선언하고 프로바이더를 임포트하고, 프로바이더는 모듈 파일에서 토큰 상수를 임포트하는 경우입니다. 배럴 파일을 사용하는 경우, 배럴 임포트가 이러한 순환 임포트를 생성하지 않도록 하세요.

위의 `<unknown_token>이` 문자열 `Object인` 경우 적절한 공급자 토큰이 없는 유형/인터페이스를 사용하여 주입하고 있다는 의미입니다. 이 문제를 해결하려면 클래스 참조를 가져오고 있는지 확인하거나 `@Inject()` 데코레이터와 함께 사용자 지정 토큰을 사용하세요. 사용자 정의 공급자 페이지를 읽어보세요.

또한 NestJS에서는 자체 주입이 허용되지 않으므로 공급자를 자체적으로 주입하지 않았는지 확인해야 합니다.

이 경우 `<unknown_token>은 <provider>와` 같을 가능성이 높습니다.

모노레포 설정인 경우 위와 동일한 오류가 발생할 수 있지만 코어 공급자의 경우

ModuleRef를 `<알 수 없는_토큰>으로 저장합니다`:

```
Nest가 <제공자>(?)의 종속성을 해결할 수 없습니다.
인덱스 [<인덱스>]의 ModuleRef 인수가 <모듈> 컨텍스트에서 사용 가능한지 확인하세요.
...
```

프로젝트에서 다음과 같이 `@nestjs/core` 패키지의 두 노드 모듈을 로드할 때 이런 문제가 발생할 수 있습니다
:

```
.
├── package.json
├── 앱
    │└── API
        │└── node_modules
            │└── @nestjs/bull
                │└── node_modules
                    │└── @nestjs/core
└── node_modules
    ├── (기타 패키지)
    └── @nestjs/core
```

솔루션:

- Yarn 워크스페이스의 경우, nohoist 기능을 사용하여 `@nestjs/core` 패키지를 들어 올리지 않도록 하세요.
- pnpm 워크스페이스의 경우, 다른 모듈의 피어 종속성 및 `"의존성 메타"`에 `@nestjs/core를` 설정

  합니다: `{{ '{' }}"다른 모듈 이름": {{ '{' }}"injected": true{{ '}}' }}를` 모듈

  을 임포트한 앱 package.json에 추가합니다. 참조: 의존성 메타인젝트

"순환 종속성" 오류

애플리케이션에서 순환 종속성을 피하기 어려운 경우가 있습니다. Nest가 이러한 문제를 해결할 수 있도록 몇
가지 단계를 수행해야 합니다. 순환 종속성으로 인해 발생하는 오류는 다음과 같습니다:

```
Nest가 <모듈> 인스턴스를 만들 수 없습니다.
<모듈> "가져오기" 배열의 인덱스 [<index>]에 있는 모듈이 정의되지 않았습니다.

잠재적 원인:
- 모듈 간의 순환 종속성. 이를 방지하려면 forwardRef()를 사용하세요. 자세히 보기:
https://docs.nestjs.com/fundamentals/circular-dependency
- 인덱스 [<index>]의 모듈이 "정의되지 않음" 유형입니다. 가져오기 확인
```

```
문과 모듈의 유형에 따라 다릅니다.

범위 [<모듈_임포트_체인>]]
# 예제 체인 AppModule -> FooModule
```

모듈 파일에서 상수를 내보내고 서비스 파일에서 가져오는 것과 같이 두 공급자가 서로 의존하거나 상수에 대해 서로 의존하는 타입스크립트 파일에서 순환 종속성이 발생할 수 있습니다. 후자의 경우 상수를 위한 별도의 파일을 생성하는 것이 좋습니다. 전자의 경우 순환 종속성에 대한 가이드를 따르고 모듈과 공급자가 모두 `forwardRef`로 표시되어 있는지 확인하세요.

종속성 오류 디버깅

종속성이 올바른지 수동으로 확인하는 것 외에도 Nest 8.1.0부터는 `NEST_DEBUG` 환경 변수를 진실로 해석하는 문자열로 설정하여 Nest가 애플리케이션의 모든 종속성을 해석하는 동안 추가 로깅 정보를 얻을 수 있습니다.

위 이미지에서 노란색 문자열은 주입되는 종속성의 호스트 클래스, 파란색 문자열은 주입된 종속성의 이름 또는 주입 토큰, 보라색 문자열은 종속성을 검색 중인 모듈입니다. 이 문자열을 사용하면 일반적으로 종속성 주입 문제가 발생하는 이유와 종속성 해결 과정을 추적할 수 있습니다.

"파일 변경 감지됨"이 끝없이 반복됨

TypeScript 버전 4.9 이상을 사용하는 Windows 사용자는 이 문제가 발생할 수 있습니다. 이 문제는 감시 모드에서 애플리케이션을 실행하려고 할 때 발생합니다(예: `npm 실행 start:`dev에서 로그 메시지가 끝없이 반복되는 것을 볼 수 있습니다):

```
XX:XX:XX AM - 파일 변경이 감지되었습니다. 증분 컴파일 시작 중... XX:XX:XX AM - 0 오류 발견. 파일 변경을 감시 중입니다.
```

NestJS CLI를 사용하여 감시 모드에서 애플리케이션을 시작할 때 `tsc -- watch를` 호출하여 수행되며, TypeScript 버전 4.9부터 파일 변경을 감지하는 새로운 전략이 사용되어 이 문제의 원인이 될 수 있습니다. 이 문제를 해결하려면 다음과 같이 `"compilerOptions"` 옵션 뒤에 tsconfig.json 파일에 설정을 추가해야 합니다:

```
  "watchOptions": {
    "watchFile": "고정 폴링 간격"
  }
```

이 옵션은 일부 컴퓨터에서 문제를 일으킬 수 있는 파일 시스템 이벤트(새로운 기본 방법) 대신 폴링 방법을 사용하여 파일 변경 사항을 확인하도록 TypeScript에 지시합니다. `"watchFile"` 옵션에 대한 자세한 내용은 TypeScript 문서에서 확인할 수 있습니다.