개요

> 정보 힌트 이 장에서는 Nest 프레임워크와의 Nest Devtools 통합에 대해 설명합니다. 개발자 도구 애플리케이션을 찾고 계신다면 개발자 도구 웹사이트를 방문하세요.

로컬 애플리케이션 디버깅을 시작하려면 다음과 같이 `main.ts` 파일을 열고 애플리케이션 옵션 개체에서 `스냅샷` 속성을 `true로` 설정하세요:

```
비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule, { 스냅
    샷: true,
  });
  await app.listen(3000);
}
```

이렇게 하면 프레임워크가 네스트 개발자 도구가 애플리케이션의 그래프를 시각화할 수 있도록 필요한 메타데이터를 수집하도

록 지시합니다. 다음으로 필요한 의존성을 설치하겠습니다:

```
$ npm i @nestjs/devtools-integration
```

> 경고 경고 애플리케이션에서 `@nestjs/graphql` 패키지를 사용하는 경우 최신 버전(`npm i @nestjs/graphql@11`)을 설치해야 합니다.

이 종속성을 설정했으면 `app.module.ts` 파일을 열고 방금 설치한 `DevtoolsModule을` 임포트해 보겠습니다:

```
모듈({ import:
  [
    DevtoolsModule.register({
      http: process.env.NODE_ENV !== 'production',
    }),
  ],
  컨트롤러: [AppController], 공급
  자: [앱서비스],
})
내보내기 클래스 AppModule {}
```

> 경고 경고 여기서 `NODE_ENV` 환경 변수를 확인하는 이유는 프로덕션 환경에서 이 모듈을 절대 사용해서는 안 되기 때문입니다!

개발자 도구 `모듈을` 가져오고 애플리케이션이 실행되면(`npm 실행 시작:dev`) 개발자 도구 URL로 이동하여 검사된 그래프를 볼 수 있어야 합니다.

> 정보 힌트 위의 스크린샷에서 볼 수 있듯이 모든 모듈은 `InternalCoreModule에` 연결됩니다. `InternalCoreModule은` 항상 루트 모듈로 임포트되는 글로벌 모듈입니다. 글로벌 노드로 등록되어 있기 때문에 Nest는 모든 모듈과 `InternalCoreModule` 노드 사이에 자동으로 에지를 생성합니다. 이제 그래프에서 전역 모듈을 숨기려면 사이드바의 "전역 모듈 숨기기" 체크박스를 사용하면 됩니다.

위에서 볼 수 있듯이 DevtoolsModule을 사용하면 애플리케이션이 포트 8000에 있는 추가 HTTP 서버를 노출하여 Devtools 애플리케이션이 앱을 인트로스펙티브하는 데 사용할 수 있습니다.

모든 것이 예상대로 작동하는지 다시 확인하려면 그래프 보기를 "클래스"로 변경합니다. 다음 화면이 표시됩니다:

특정 노드에 초점을 맞추려면 사각형을 클릭하면 그래프에 '포커스' 버튼이 있는 팝업 창이 표시됩니다. 사이드바에 있는 검색창을 사용하여 특정 노드를 찾을 수도 있습니다.

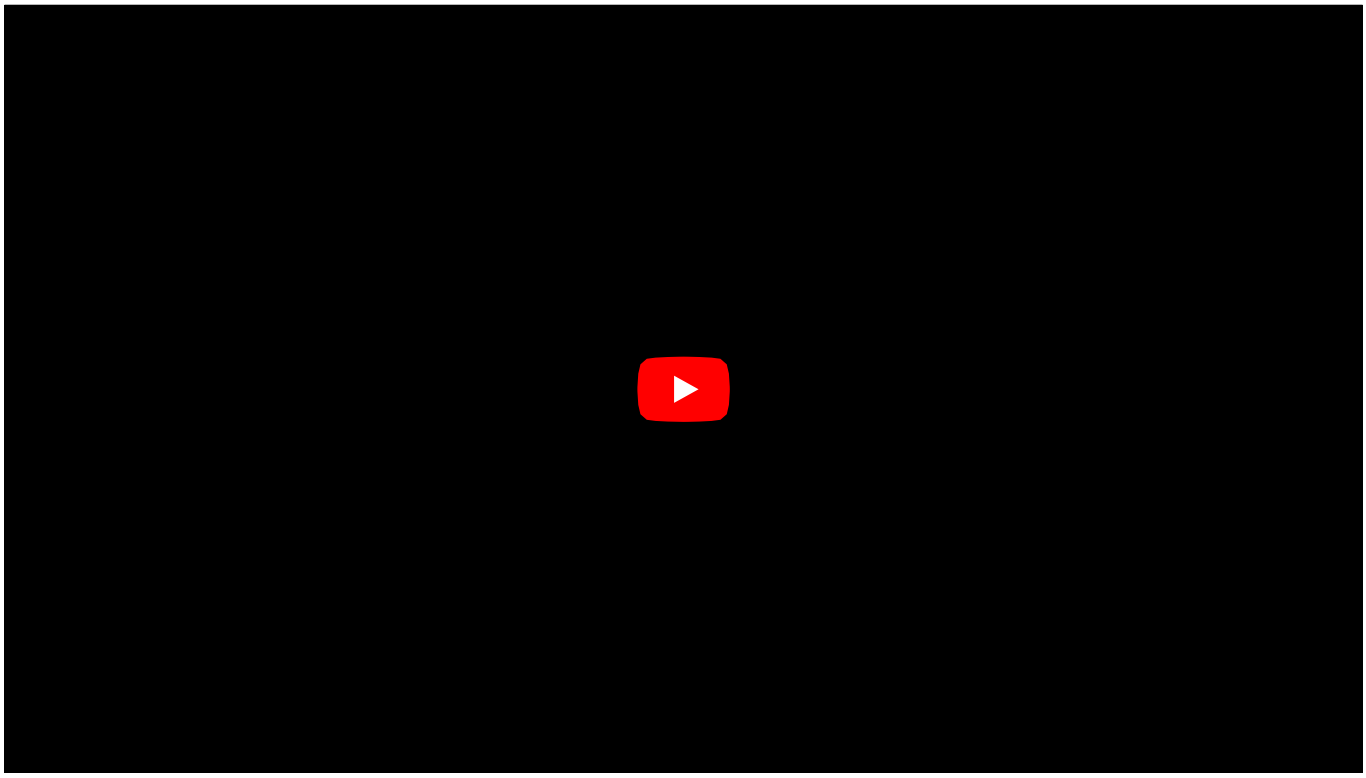
> 정보 힌트 검사 버튼을 클릭하면 애플리케이션이 특정 노드가 선택된 `/debug` 페이지로 이동합니다.

> 정보 힌트 그래프를 이미지로 내보내려면 그래프 오른쪽 모서리에 있는 PNG로 내보내기 버튼을 클릭합니다.

사이드바(왼쪽)에 있는 양식 컨트롤을 사용하여 예를 들어 특정 애플리케이션 하위 트리를 시각화하기 위해 가장자리 근접성을 제어할 수 있습니다:

이 기능은 팀에 새로운 개발자가 들어왔을 때 애플리케이션이 어떻게 구성되어 있는지 보여주고 싶을 때 특히 유용할 수 있습니다. 또한 이 기능을 사용하여 특정 모듈(예: `TasksModule`)과 모든 종속성을 시각화할 수 있으므로 대규모 애플리케이션을 더 작은 모듈(예: 개별 마이크로 서비스)로 세분화할 때 유용하게 사용할 수 있습니다.

이 동영상을 통해 그래프 탐색기 기능이 실제로 작동하는 모습을 확인할 수 있습니다:



"종속성을 해결할 수 없음" 오류 조사 중

> 정보 참고 이 기능은 `@nestjs/core` >= `v9.3.10`에서 지원됩니다.

아마도 가장 흔히 볼 수 있는 오류 메시지는 Nest가 공급자의 종속성을 해결할 수 없다는 것입니다. Nest 개발자 도구를 사용하면 문제를 쉽게 식별하고 해결 방법을 배울 수 있습니다.

먼저 `main.ts` 파일을 열고 다음과 같이 `부트스트랩()` 호출을 업데이트합니다:

```
bootstrap().catch((err) => {
  fs.writeFileSync('graph.json', PartialGraphHost.toString() ?? '');
  process.exit(1);
});
```

또한 `abortOnError를 false로` 설정해야 합니다:

```
const app = await NestFactory.create(AppModule, { 스냅샷:
  true,
  abortOnError: false, // <--- THIS
});
```

이제 애플리케이션이 "종속성을 해결할 수 없음" 오류로 인해 부트스트랩에 실패할 때마다 루트 디렉터리에서 그래프(부분 그래프를 나타내는) 파일인 `graph.json`을 찾을 수 있습니다. 그런 다음 이 파일을 개발자 도구로 끌어다 놓으면 됩니다(현재 모드를 "대화형"에서 "미리보기"로 전환해야 함):

업로드에 성공하면 다음과 같은 그래프와 대화창이 표시됩니다:

보시다시피 강조 표시된 `작업` 모듈이 우리가 살펴봐야 할 모듈입니다. 또한 대화 창에서 이 문제를 해결하는 방법에 대한 몇 가지 지침을 이미 볼 수 있습니다.

대신 '클래스' 보기로 전환하면 다음과 같은 내용이 표시됩니다:

이 그래프는 `태스크 서비스에` 주입하려는 `진단 서비스를 태스크 모듈의` 컨텍스트에서 찾을 수 없음을 보여줍니다.

모듈에진단 모듈을 가져와서 이 문제를 해결해야 할 것입니다! 경로 탐색기

경로 탐색기 페이지로 이동하면 등록된 모든 진입점을 볼 수 있습니다:

> 정보 힌트 이 페이지에는 HTTP 경로뿐만 아니라 다른 모든 진입점(예: 웹소켓, gRPC, GraphQL 리졸버 등)도 표시됩니다.

엔트리포인트는 호스트 컨트롤러에 따라 그룹화됩니다. 검색창을 사용하여 특정 엔트리포인트를 찾을 수도 있습니다.

특정 진입 지점을 클릭하면 흐름 그래프가 표시됩니다. 이 그래프는 진입점(예: 이 경로에 연결된 가드, 인터셉터, 파이프 등)의 실행 흐름을 보여줍니다. 이 그래프는 특정 경로에 대한 요청/응답 주기를 이해하거나 특정 가드/인터셉터/파이프가 실행되지 않는 이유를 해결할 때 특히 유용합니다.

샌드박스

자바스크립트 코드를 즉석에서 실행하고 애플리케이션과 실시간으로 상호작용하려면 샌드박스 페이지로 이동하세요:

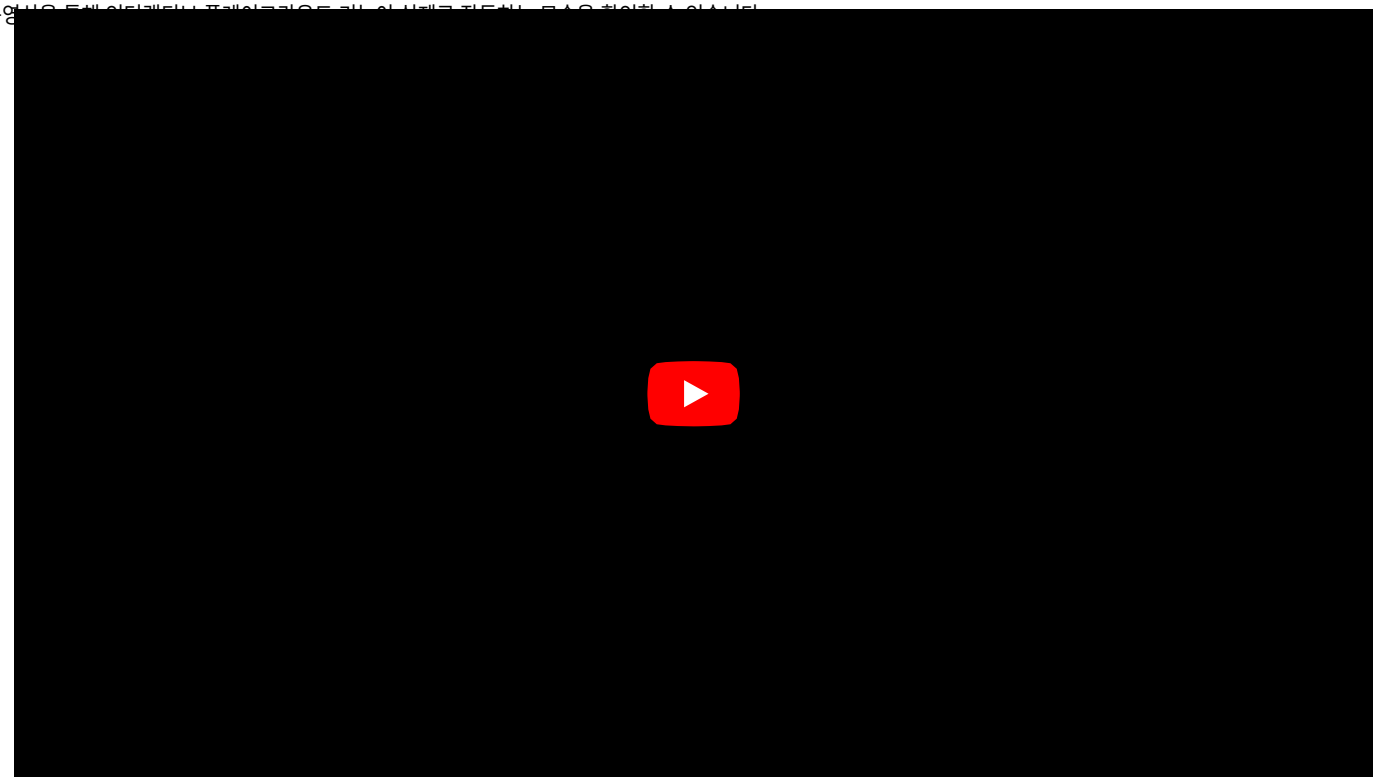
플레이그라운드를 사용하면 API 엔드포인트를 실시간으로 테스트하고 디버깅할 수 있으므로 개발자는 HTTP 클라이언트 등을 사용하지 않고도 문제를 빠르게 식별하고 해결할 수 있습니다. 또한 인증 계층을 우회할 수 있으므로 더 이상 추가 로그인 단계나 테스트 목적의 특수 사용자 계정이 필요하지 않습니다. 이벤트 기반 애플리케이션의 경우, 플레이그라운드에서 직접 이벤트를 트리거하고 애플리케이션이 이에 어떻게 반응하는지 확인할 수도 있습니다.

로그아웃되는 모든 항목은 플레이그라운드의 콘솔로 간소화되어 무슨 일이 일어나고 있는지 쉽게 확인할 수 있습니다.

애플리케이션을 다시 빌드하고 서버를 다시 시작할 필요 없이 코드를 즉시 실행하고 결과를 바로 확인할 수 있습니다.

> 정보 힌트 객체 배열을 예쁘게 표시하려면 `console.table()`(또는 그냥 `table()`) 함수를 사용하세요.

이 동영



부트스트랩 성능 분석기

모든 클래스 노드(컨트롤러, 공급자, 인핸서 등) 목록과 해당 인스턴스화 시간을 확인하려면 부트스트랩 성능 페이지로 이동하세요:

이 페이지는 애플리케이션의 부트스트랩 프로세스에서 가장 느린 부분을 식별하려는 경우(예: 서버리스 환경에서 중요한 애플리케이션의 시작 시간을 최적화하려는 경우)에 특히 유용합니다.

감사

자동 생성된 감사(애플리케이션이 직렬화된 그래프를 분석하는 동안 발생한 오류/경고/힌트)를 확인하려면 감사 페이지로 이동합니다:

> 정보 힌트 위의 스크린샷에는 사용 가능한 모든 감사 규칙이 표시되어 있지 않습니다.

이 페이지는 애플리케이션의 잠재적인 문제를 파악할 때 유용합니다.

정적 파일 미리보기

직렬화된 그래프를 파일에 저장하려면 다음 코드를 사용합니다:

```
await app.listen(3000); // OR await app.init()
fs.writeFileSync('./graph.json', app.get(SerializedGraph).toString());
```

> 정보 힌트 SerializedGraph는 `@nestjs/core` 패키지에서 내보냅니다.

그런 다음 이 파일을 끌어다 놓거나 업로드할 수 있습니다:

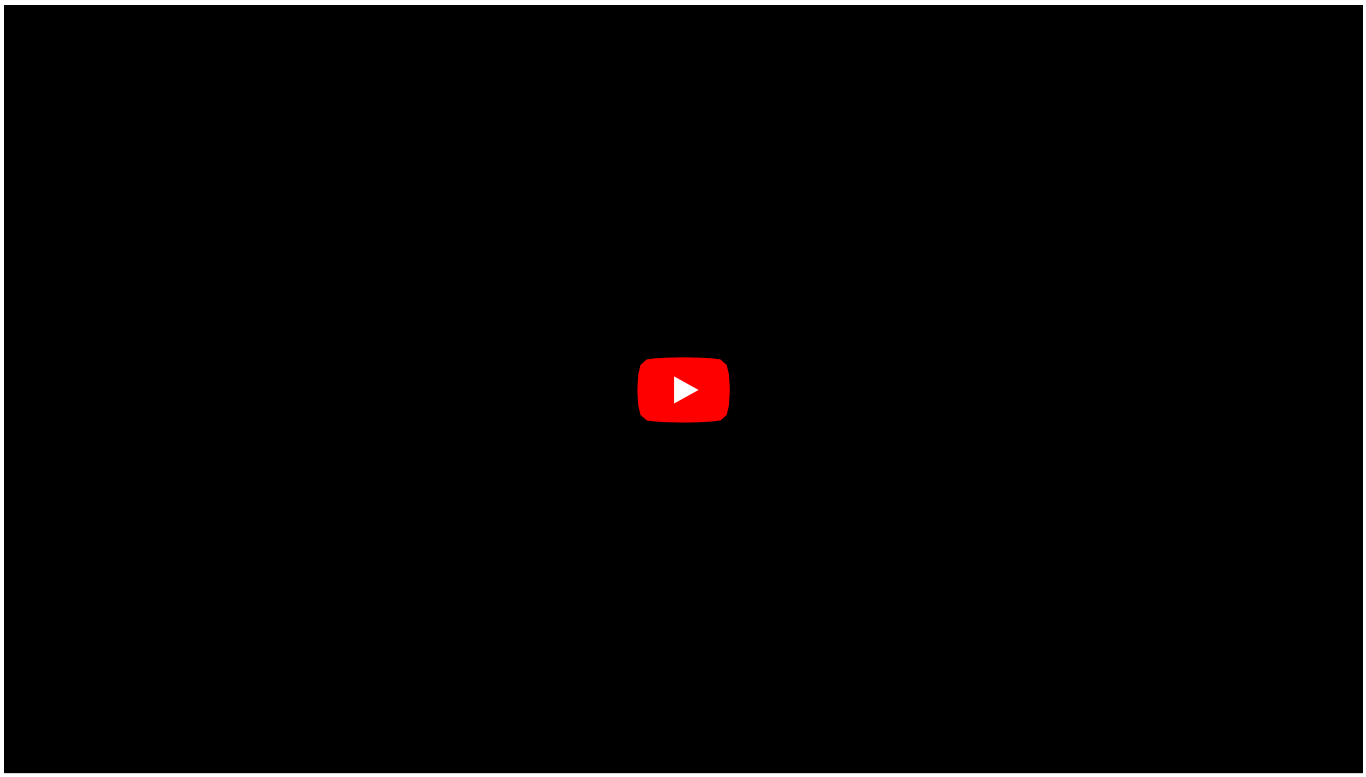
이 기능은 다른 사람(예: 동료)과 그래프를 공유하거나 오프라인에서 분석하고 싶을 때 유용합니다.

CI/CD 통합

> 정보 힌트 이 장에서는 Nest 프레임워크와의 Nest Devtools 통합에 대해 설명합니다. 개발자 도구 애플리케이션을 찾고 계신다면 개발자 도구 웹사이트를 방문하세요.

CI/CD 통합은 Enterprise 요금제를 사용하는 사용자에게 제공됩니다.

이 동영상을 통해 CI/CD 통합이 도움이 되는 이유와 방법을 알아보세요:



그래프 게시

먼저 애플리케이션 부트스트랩 파일(`main.ts`)을 다음과 같이 `GraphPublisher` 클래스(`@nestjs/devtools-integration`에서 내보낸 - 자세한 내용은 이전 장 참조)를 사용하도록 구성해 보겠습니다:

```
비동기 함수 부트스트랩() {
  const shouldPublishGraph = process.env.PUBISH_GRAPH === "true";

  const app = await NestFactory.create(AppModule, { 스냅
    샷: true,
    미리보기: shouldPublishGraph,
  });

  if (shouldPublishGraph) {
    await app.init();

    const publishOptions = { ... } // 참고: 이 옵션 객체는 사용 중인 CI/CD 제공업체에 따라 달라집니다.
    const graphPublisher = new GraphPublisher(app);
    await graphPublisher.publish(publishOptions);

    await app.close();
  } else {
    await app.listen(3000);
  }
}
```

보시다시피, 여기서는 `그래프` 퍼블리셔를 사용하여 직렬화된 그래프를 중앙 집중식 레지스트리에 게시하고 있습니다. `PUBLISH_GRAPH는` 그래프를 게시할지(CI/CD 워크플로), 아니면 일반 애플리케이션 부트스트랩에 게시할지 여부를 제어할 수 있는 사용자 정의 환경 변수입니다. 또한 여기서 `미리보기` 속성을 `true로` 설정했습니다. 이 플래

그를 활성화하면 애플리케이션이 미리보기 모드에서 부트스트랩되며, 이는 기본적으로 애플리케이션의 모든 컨트롤러, 인핸서 및 프로바이더의 생성자(및 수명 주기 후크)가 실행되지 않음을 의미합니다. 참고 - 필수는 아니지만 이 경우 CI/CD 파이프라인에서 애플리케이션을 실행할 때 데이터베이스 등에 연결할 필요가 없으므로 작업이 더 간단해집니다.

게시 옵션 개체는 사용 중인 CI/CD 제공업체에 따라 달라집니다. 가장 많이 사용되는 CI/CD 제공업체에 대한 지침은 이후 섹션에서 자세히 설명합니다.

그래프가 성공적으로 게시되면 워크플로 보기에 다음과 같은 출력이 표시됩니다:

그래프가 게시될 때마다 프로젝트의 해당 페이지에 새 항목이 표시됩니다:

보고서

중앙 집중식 레지스트리에 이미 해당 스냅샷이 저장되어 있는 경우 Devtools는 모든 빌드에 대해 보고서를 생성합니다. 예를 들어 그래프가 이미 게시된 `마스터` 브랜치에 대해 PR을 만들면 애플리케이션에서 차이점을 감지하고 보고서를 생성할 수 있습니다. 그렇지 않으면 보고서가 생성되지 않습니다.

보고서를 보려면 프로젝트의 해당 페이지로 이동합니다(조직 참조).

이 기능은 코드 검토 중에 눈에 띄지 않았을 수 있는 변경 사항을 식별하는 데 특히 유용합니다. 예를 들어, 누군가 깊게 중첩된 프로바이더의 범위를 변경했다고 가정해 보겠습니다. 이러한 변경 사항은 검토자가 즉시 알아차리지 못할 수도 있지만 Devtools를 사용하면 이러한 변경 사항을 쉽게 발견하고 의도적인 변경인지 확인할 수 있습니다. 또는 특정 엔드포인트에서 가드를 제거하면 보고서에서 해당 엔드포인트가 영향을 받는 것으로 표시됩니다. 해당 경로에 대한 통합 또는 e2e 테스트가 없다면 더 이상 보호되지 않는다는 사실을 알아차리지 못할 수 있으며, 알아차렸을 때는 이미 너무 늦을 수 있습니다.

마찬가지로 대규모 코드베이스에서 작업할 때 모듈을 전역으로 수정하면 그래프에 추가된 에지 수를 확인할 수 있으며, 이는 대부분의 경우 뭔가 잘못하고 있다는 신호입니다.

빌드 미리 보기

게시된 모든 그래프에 대해 시간을 거슬러 올라가서 미리보기 버튼을 클릭하여 이전 그래프의 모습을 미리 볼 수 있습니다. 또한 보고서가 생성된 경우 그래프에서 차이점이 강조 표시된 것을 볼 수 있습니다:

- 녹색 노드는 추가된 요소를 나타냅니다.
- 밝은 흰색 노드는 업데이트된 요소를 나타냅니다
- 빨간색 노드는 삭제된 요소를 나타냅니다.

아래 스크린샷을 참조하세요:

시간을 거슬러 올라가는 기능을 사용하면 현재 그래프와 이전 그래프를 비교하여 문제를 조사하고 해결할 수 있습니다. 설정 방법에 따라 모든 풀 리퀘스트(또는 모든 커밋)에 해당하는 스냅샷이 레지스트리에 저장되므로 시간을 거슬러 올라가서 변경된 내용을 쉽게 확인할 수 있습니다. Nest가 애플리케이션 그래프를 구성하는 방식을 이해하고 이를 시각화할 수 있는 기능을 갖춘 Devtools를 Git이라고 생각하세요.

통합: Github 액션

먼저 프로젝트의 `.github/workflows` 디렉터리에 새 Github 워크플로우를 생성하고, 예를 들어 `publish-graph.yml이라고` 부르겠습니다. 이 파일 안에 다음 정의를 사용하겠습니다:

```
이름: 개발자 도구

켜짐:
  푸시합니다:
    브랜치:
      - 마스터 풀
  _요청:
      브랜치:
      - '*'

작업:
  게시합니다:
    if: github.actor!= 'dependabot[bot]'
    name: 그래프 게시
    실행 중: 우분투 최신 단계:
      - 용도: 액션/체크아웃@v3
      - 사용: 동작/설정 노드@v3와 함께:
          노드-버전: '16' 캐시:
          'npm'
      - 이름: 설치 종속성 실행: npm ci
      - 이름: 설정 환경(PR)
        if: {{ '${{' }} github.event_name == 'pull_request' {{ '}}' }} 셸:
        bash
        실행합니다: |
          echo "COMMIT_SHA={{ '${{' }} github.event.pull_request.head.sha {{ '}}' }}" >>\${github_env}
      - 이름: 설정 환경(푸시)
```

```
        if: {{ '${{' }} github.event_name == 'push' {{ '}}' }}
        셸: bash
        실행합니다: |
          echo "COMMIT_SHA=\${GITHUB_SHA}" >> \${GITHUB_ENV}
      - 이름: 게시
        실행합니다: PUBLISH_GRAPH=true npm 실행 시
        작 환경:
          devtools_api_key: 변경_이_것을_귀하의_api_키로_변경합니다.
          리포지토리_이름: {{ '${{' }} github.event.repository.name {{ '}}' }}
          BRANCH_NAME: {{ '${{' }} github.head_ref || github.ref_name {{ '}}' }}
          TARGET_SHA: {{ '${{' }} github.event.pull_request.base.sha {{ '}}' }}
```

이상적으로 DEVTOOLS_API_KEY 환경 변수는 Github 시크릿에서 검색해야 하며, 자세한 내용은 여기를 참조하세요.

이 워크플로는 마스터 브랜치를대상으로 하는 각 풀 리퀘스트마다 실행되거나 마스터 브랜치에직접 커밋이 있는 경우 실행됩니다. 이 구성은 프로젝트에 필요한 모든 것에 맞춰 자유롭게 조정할 수 있습니다. 여기서 중요한 것은 그래프 퍼블리셔 클래스를 실행하는 데 필요한 환경 변수를 제공해야 한다는 것입니다.

하지만 이 워크플로를 사용하기 전에 업데이트해야 하는 변수가 하나 있는데, 바로 DEVTOOLS_API_KEY입니다. 이 페이지에서 프로젝트 전용 API 키를 생성할 수 있습니다.

마지막으로, 다시 main.ts 파일로 이동하여 이전에 비워둔 publishOptions 객체를 업데이트해 보겠습니다.

```
const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY, 리포지토리
  : process.env.REPOSITORY_NAME, 소유자:
  process.env.GITHUB_REPOSITORY_OWNER, sha:
  process.env.COMMIT_SHA,
  대상: process.env.TARGET_SHA,
  트리거: process.env.GITHUB_BASE_REF ? 'pull' : 'push', 브랜치:
  process.env.BRANCH_NAME,
};
```

최상의 개발자 환경을 위해 "Github 앱 통합" 버튼을 클릭하여 프로젝트의 Github 애플리케이션을 통합하세요(아래 스크린샷 참조). 참고 - 필수는 아닙니다.

이 통합 기능을 사용하면 풀 리퀘스트에서 바로 미리보기/리포트 생성 프로세스의 상태를 확인할 수 있습니다:

통합: Gitlab 파이프라인

먼저 프로젝트의 루트 디렉터리에 새 Gitlab CI 구성 파일을 생성하고 이름을 .gitlab-ci.yml로 지정합니다. 이 파일 안에 다음 정의를 사용하겠습니다:

```
const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY, 리포지토리
  : process.env.REPOSITORY_NAME, 소유자:
  process.env.GITHUB_REPOSITORY_OWNER, sha:
  process.env.COMMIT_SHA,
  대상: process.env.TARGET_SHA,
  트리거: process.env.GITHUB_BASE_REF ? 'pull' : 'push', 브랜치:
  process.env.BRANCH_NAME,
};
```

> 정보 힌트 이상적으로는 DEVTOOLS_API_KEY 환경 변수를 시크릿에서 검색해야 합니다.

이 워크플로는 마스터 브랜치를대상으로 하는 각 풀 리퀘스트마다 실행되거나 마스터 브랜치에직접 커밋이 있는 경우 실행됩니다. 이 구성은 프로젝트에 필요한 모든 것에 맞춰 자유롭게 조정할 수 있습니다. 여기서 중요한 것은 그래프 퍼블리셔 클래스가 실행되는 데 필요한 환경 변수를 제공해야 한다는 것입니다.

하지만 이 워크플로우를 사용하기 전에 업데이트해야 하는 변수(이 워크플로우 정의에서)가 하나 있는데, 바로 DEVTOOLS_API_KEY입니다. 이 페이지에서 프로젝트 전용 API 키를 생성할 수 있습니다.

마지막으로, 다시 `main.ts` 파일로 이동하여 이전에 비워둔 `publishOptions` 객체를 업데이트해 보겠습니다.

```
이미지: 노드:16단

계:
  - 빌드 캐시:
```

```
  키를 누릅니다:
    파일을 만듭니다:
      - 패키지-잠금.json 경로:
    - node_modules/

워크플로: 규칙:
    - if: CI_PIPELINE_SOURCE == "merge_request_event"
      when: 항상
    - if: CI_COMMIT_BRANCH == "마스터" && $CI_PIPELINE_SOURCE == "푸시" 언제
      : 항상
    - 언제: 절대

설치_종속성: 단계: 빌드
  스크립트:
    - npm CI

게시_그래프: 단계
  : 빌드 요구 사
  항:
    - 설치_종속성 스크립트:
  npm 실행 시작 변수:
    PUBLISH_GRAPH: 'true'
    devtools_api_key: 'change_this_to_our_api_key'
```

기타 CI/CD 도구

Nest Devtools CI/CD 통합은 원하는 모든 CI/CD 도구(예: Bitbucket Pipelines , CircleCI 등)와 함께 사용할 수 있으므로 여기에서 설명한 제공업체에 국한되지 않습니다.

다음 게시 옵션 개체 구성을 살펴보고 특정 커밋/빌드/PR에 대한 그래프를 게시하는 데 필요한 정보를 이해하세요.

```
const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY, 리포지토리
  : process.env.CI_PROJECT_NAME, 소유자:
  process.env.CI_PROJECT_ROOT_NAMESPACE, sha:
  process.env.CI_COMMIT_SHA,
  대상: process.env.CI_MERGE_REQUEST_DIFF_BASE_SHA,
  트리거: process.env.CI_MERGE_REQUEST_DIFF_BASE_SHA ? 'pull' : 'push',
  branch:
    process.env.CI_COMMIT_BRANCH ??
    process.env.CI_MERGE_REQUEST_SOURCE_BRANCH_NAME,
};
```

이 정보의 대부분은 CI/CD 기본 제공 환경 변수를 통해 제공됩니다(CircleCI 기본 제공 환경 목록 및 Bitbucket 변수 참조). 그래프 게시를 위한 파이프라인

구성의 경우, 다음 트리거를 사용하는 것이 좋습니다:

- 푸시 이벤트 - 현재 브랜치가 배포 환경(예: 마스터, 메인, 스테이징, 프로덕션 등)을 나타내는 경우에만 해당됩니다.
- 풀 리퀘스트 이벤트 - 항상 또는 대상 브랜치가 배포 환경을 나타내는 경우(위 참조)