APPENDIX

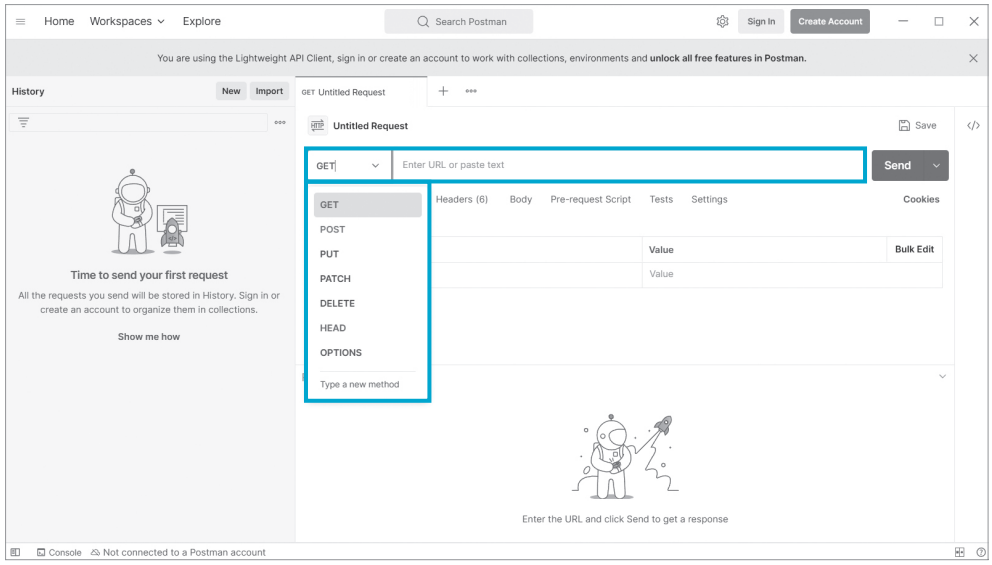
API 테스트를 위한 툴, Postman

Postman은 실무에서 많이 사용하는 HTTP 클라이언트로서, 개발 과정에서 API 테스트를 위한 툴로 사용된다. HTTP 트랜잭션과 관련된 Postman의 사용 방법을 알아보자.

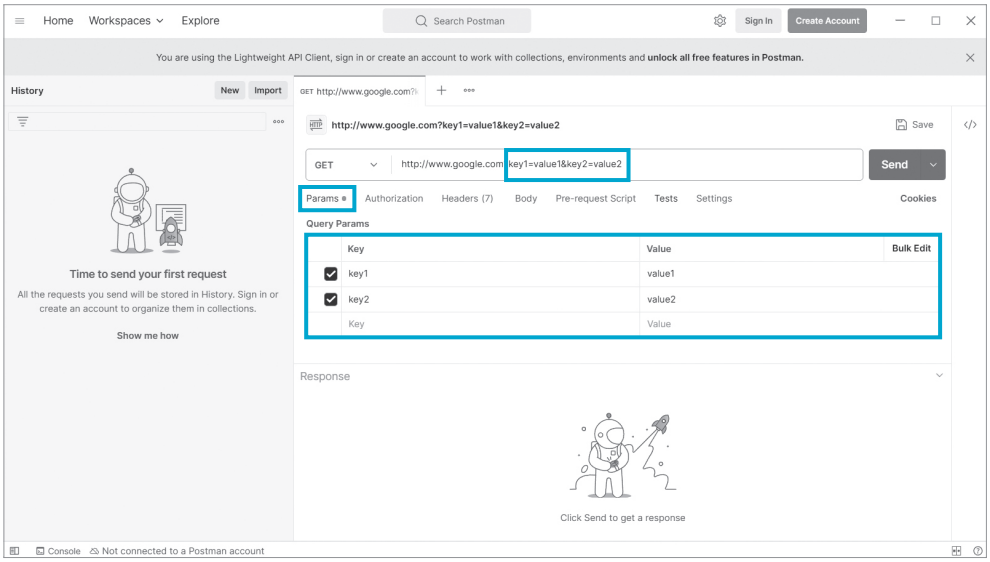
요청 관련 사용 방법

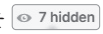
Postman을 실행하면 다음과 같은 기본 창 구성을 확인할 수 있다. 지속적으로 업데이트되므로 일부 UI가 변경될 수 있다. 기본적인 기능과 위치를 확인해 학습에 참고하자.

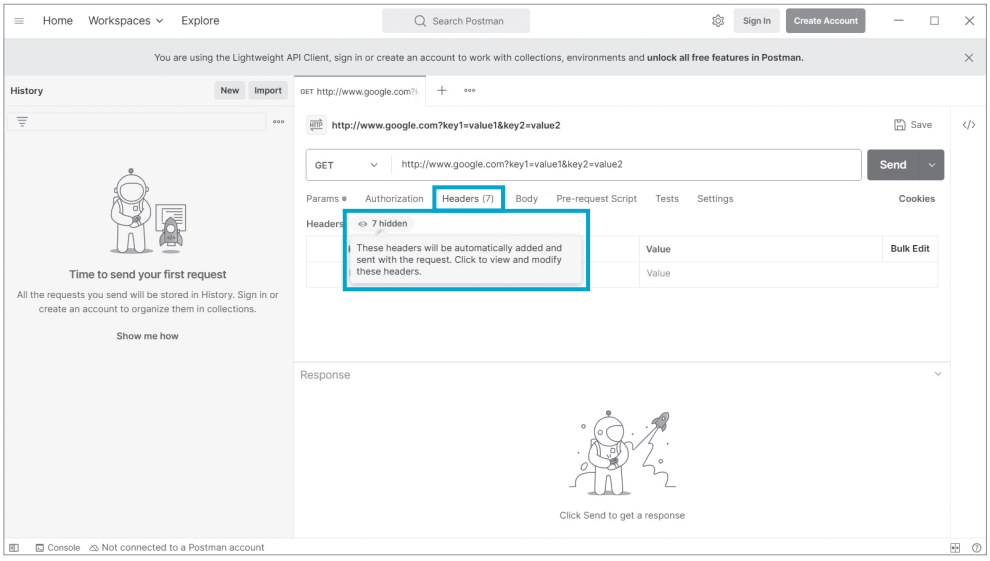
상단 탭에 있는 ⊕ 아이콘을 클릭해 새로운 Request를 추가하면 가장 위쪽에서 요청할 메서드와 URL을 지정할 수 있다.



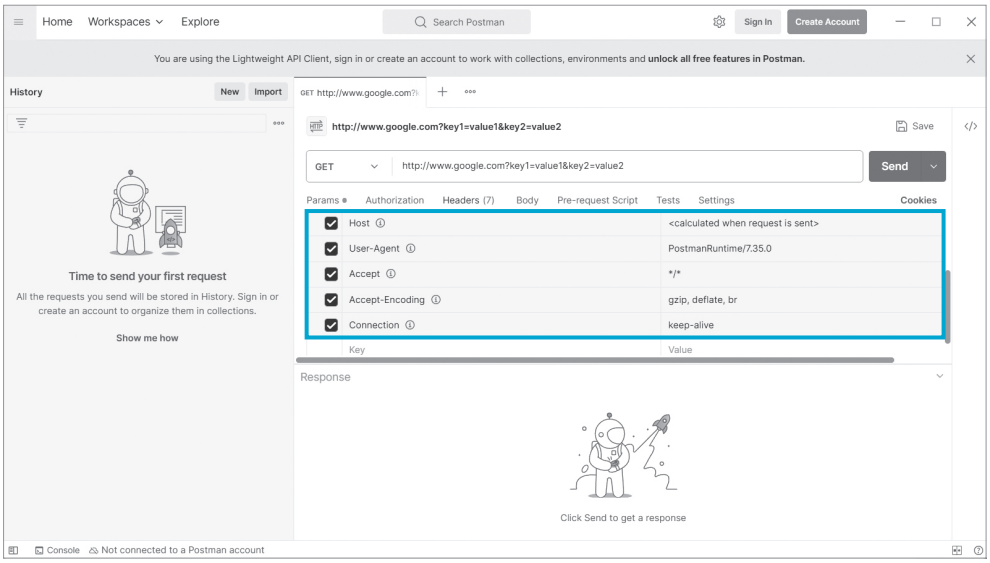
그 밑에는 요청에 사용할 수 있는 여러 가지 탭이 있다. 첫 번째 [Params] 탭에서는 다음과 같이 쿼리 파라미터를 사용하여 요청할 내용을 지정하며, [Authorization] 탭에서는 웹 애플리케이션의 인증과 인가에 대한 보안을 설정한다. [Authorization]을 비롯해 [Pre-request Script], [Tests], [Settings] 탭은 개발에 유용한 기능이지만 심화 내용에 해당하므로 자세한 정보는 검색을 통해 보완하자.



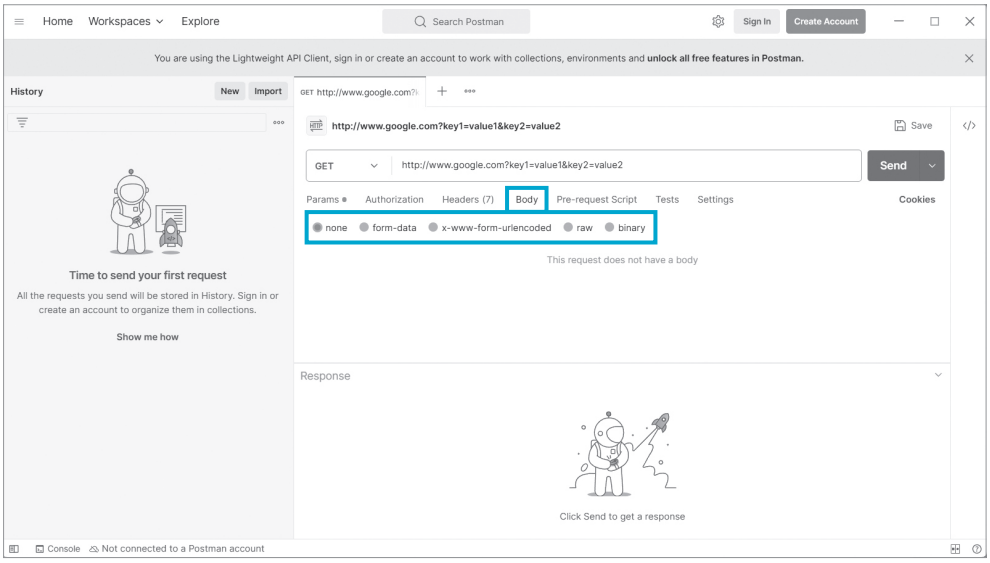
[Headers] 탭에서는 요청 헤더를 지정한다. 처음에는 Postman에서 기본 요청 헤더를 숨겨 두기 때문에 다음과 같이 아무 헤더도 보이지 않는다. 사용자가 설정하는 헤더만 우선적으로 보여 주기 위함이다. 표 위에 있는  아이콘을 누르면 숨겨진 헤더를 확인할 수 있다. 예상할 수 있듯 아이콘에 있는 숫자(7)는 숨겨진 헤더의 수이다.



숨겨져 있던 헤더들은 Postman에서 자동으로 채워 주는 값이다. [Body] 탭에서 다른 방식으로 데이터를 전송하면 필요한 헤더가 [Headers] 탭에 자동으로 설정될 것이다.

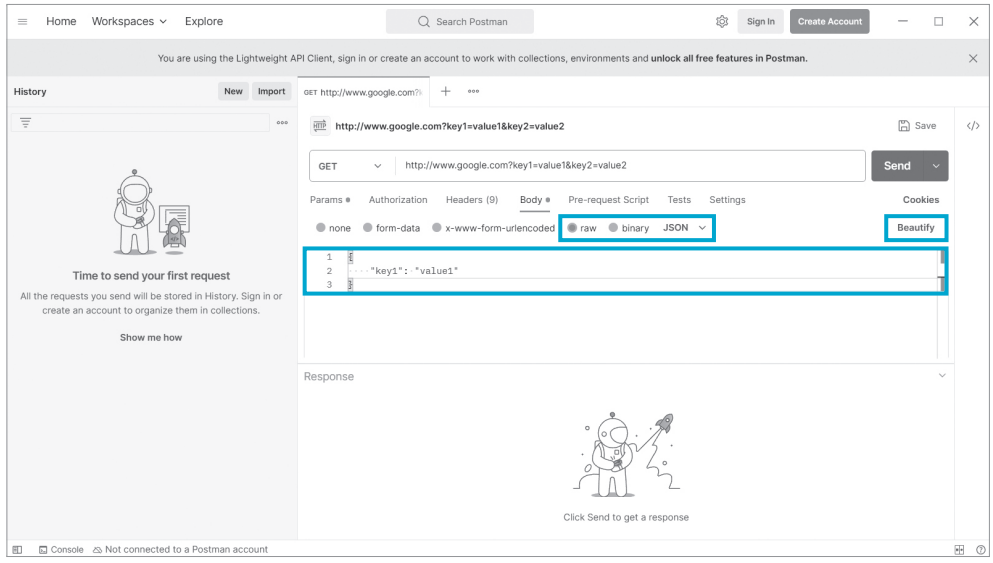


또한 [Body] 탭에서는 요청 바디로 보낼 데이터를 지정할 수 있다. 각각의 데이터는 다음과 같이 구분된다.

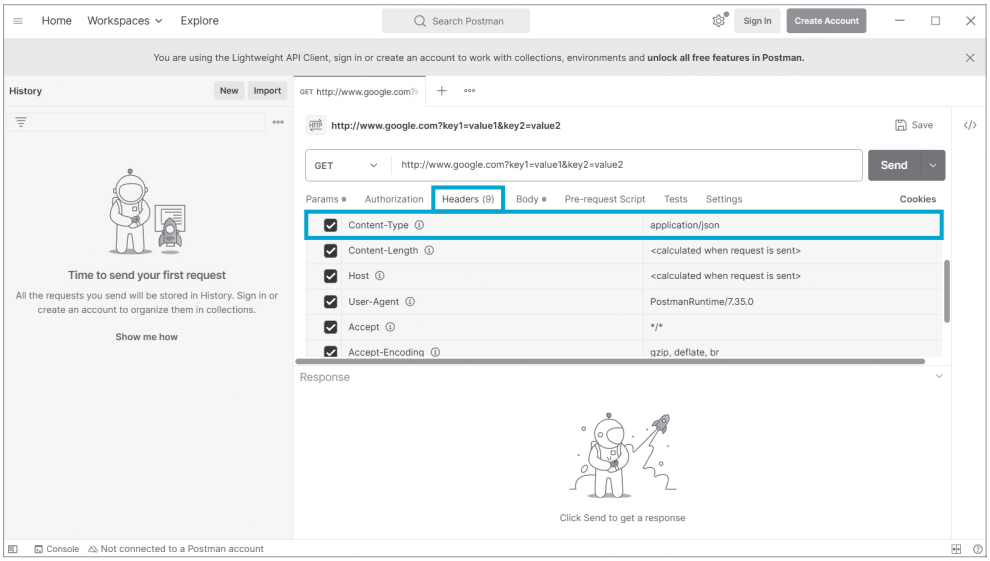


- **none**: 요청 바디를 사용하지 않는다.
- **form-data**: 'Content-Type: multipart/form-data'로 보내는 멀티 파트 요청이다.
- **x-www-form-urlencoded**: form 태그 요청의 기본값으로, 'Content-Type : application/x-www-form-urlencoded'로 보내는 요청이다. 요청 시 파라미터는 URL 인코딩된다.
- **raw**: Body를 그대로 보내는 요청 방식으로, Text, JavaScript, JSON, HTML, XML 등으로 보낼 수 있다.
- **binary**: 파일을 전송할 수 있는 옵션으로, 파일 업로드 등을 테스트한다.
- **GraphQL**: 사용자가 저장소에 저장된 데이터를 조회하는 것처럼 조회 조건을 지정해서 보내는 요청 방식이다. 자세한 내용은 구글에서 'GraphQL'을 검색해 보완하자.

그럼 raw에서 JSON으로 설정했을 때의 요청 방법을 살펴보자. 먼저 [raw]에 체크하고 더보기 목록에서 'JSON'을 선택한다. 포맷만 맞게 '{"key1":"value1"}'이라고 입력하고 오른쪽에 있는 'Beautify'를 클릭하면 줄을 바꾸거나 띄어쓰기를 하지 않아도 보기 좋게 정렬된 JSON을 확인할 수 있다.



JSON으로 바디를 입력한 상태에서 [Headers] 탭을 클릭하면 [Content-Type]이 'application/json'으로 설정되어 있는 것을 확인할 수 있다.



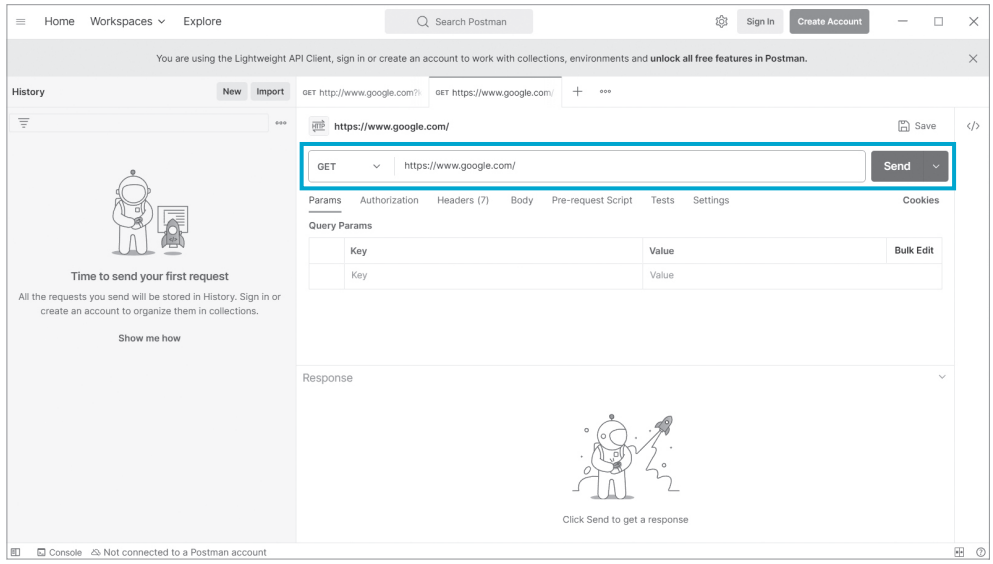
마지막으로 유용하게 사용할 만한 기능을 소개한다. 입력 창 오른쪽에 있는 코드(</>) 아이콘을 클릭하면 나타나는 [Code snippet] 창에서는 해당 요청을 여러 가지 언어로 보낼 수 있는 코드를 자동 생성한다. Postman에서 제공하는 HTTP 요청의 헤더와 바디를 그대로 보내려면 또 다른 HTTP 클라이언트나 텔넷Telnet 클라이언트가 있어야 하므로 HTTP는 보통 참고용으로만 사용한다. 실제로는 사용하고자 하는 언어로 코드를 변환하거나 cURL로 변환하여 사용한다.

NOTE 환경에 따라 윈도우 운영체제에서는 작은 따옴표나 개행 문자 등의 문자열 처리가 달라 정상적으로 실행되지 않을 수 있다. Postman을 통해 이런 기능을 사용할 수 있다는 점 정도만 기억하자.

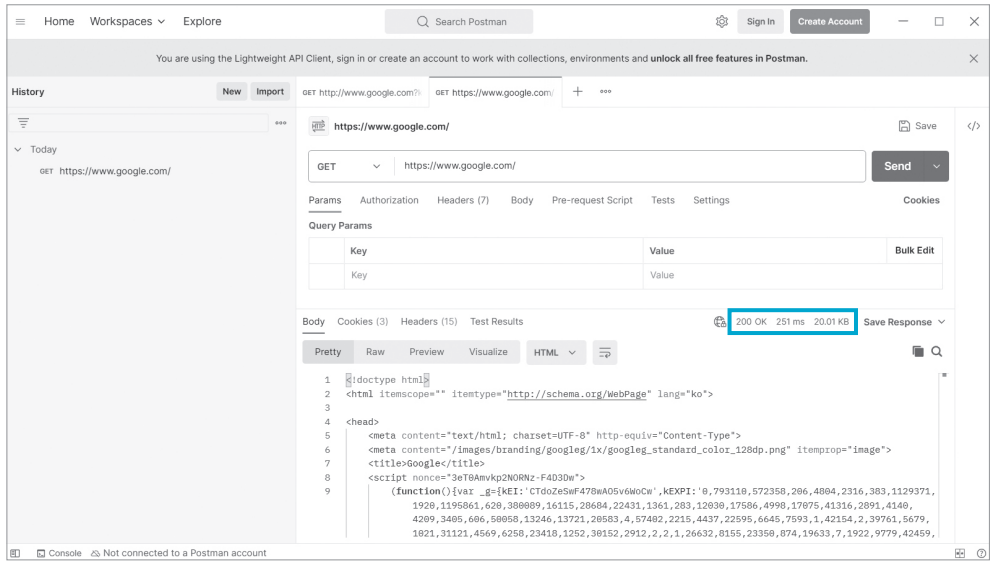
응답 관련 사용 방법

HTTP 트랜잭션에서와 마찬가지로 응답과 관련된 사용 방법은 간단하다. HTTP 트랜잭션에서 확인했던 내용을 Postman에서는 어떻게 확인하는지에 초점을 맞춰 실습해 보자.

Postman 상단 탭에 있는 + 아이콘을 클릭해 새로운 요청 탭을 연 다음, 메서드는 'GET', URL은 'https://www.google.com/'이라고 입력하고 [Send]를 누른다.

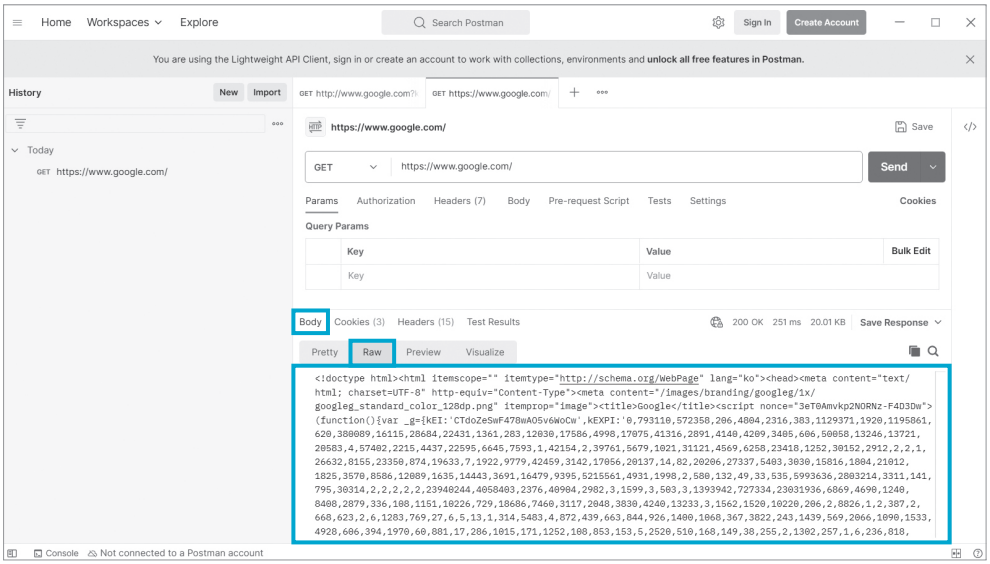


그러면 추가된 응답 관련 정보를 확인할 수 있다. 응답 영역의 오른쪽 위에는 공통으로 노출되는 정보가 있다. 사용자 환경에 따라 내용은 조금 다를 수 있다. '200 OK'는 상태 코드, '251ms'는 요청 후 응답이 오는데 걸린 시간을 나타낸다. 1ms는 1/1000초이므로 251ms는 0.251초이다. 또한 옆에 있는 '20.01KB'는 해당 응답의 크기로, 헤더와 바디를 더한 크기를 나타낸다. 개발 과정에서는 응답의 크기보다 상태 코드와 응답이 오는 데 걸리는 시간에 대한 정보를 더 많이 활용하게 될 것이다.

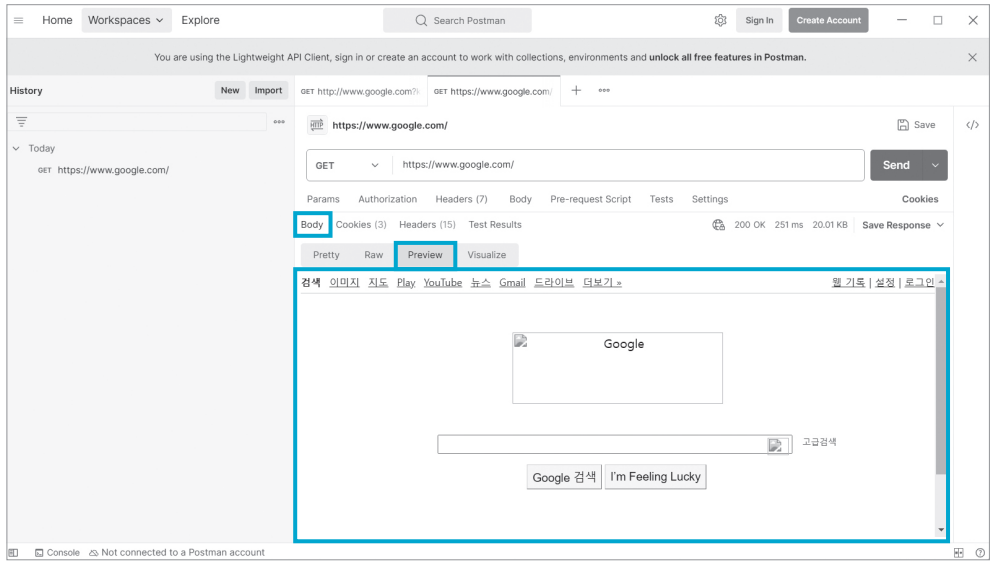


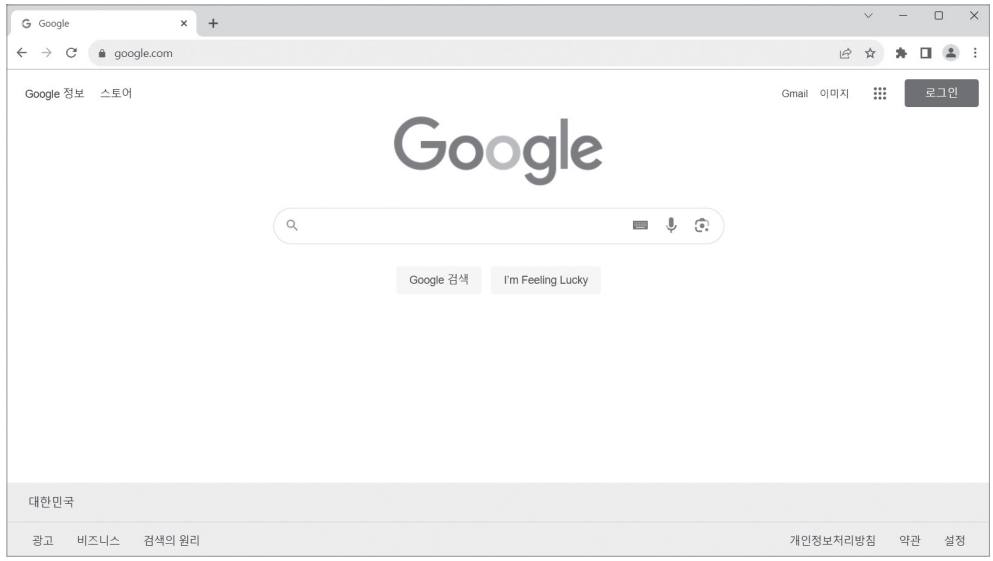
이어서 [Body], [Headers] 탭을 살펴보자. [Cookies] 탭은 요청에서 설명했던 [Authorization]과 마찬가지로, 인증과 인가에 주로 사용되는 '헤더의 일부'이다.

[Body] 탭은 개발자 도구에 있던 [Preview]와 [Response] 탭과 비슷하다. [Raw] 메뉴는 [Response]와 비슷하게 있는 그대로의 데이터를 보여 주고, [Preview] 메뉴는 실제 웹 브라우저와 동일한 화면을 보여 준다. [Pretty] 메뉴를 클릭하면 [Raw] 메뉴에 있는 그대로의 코드나 JSON을 보기 좋게 보여 준다.



[Preview] 메뉴를 클릭하면 실제 웹 브라우저에서 보이는 구글 검색 페이지와는 달리 글씨체도, 이미지도 정상적으로 표시되지 않고 있다. 그 이유는 'https://www.google.com/'에 대해서만 HTTP 트랜잭션이 발생했고, HTML 페이지에 포함된 CSS, 자바스크립트, 이미지 등에 대한 HTTP 트랜잭션은 발생하지 않았기 때문이다. 또한 [Visualize] 탭과 스크립트를 활용해 보기 좋게 응답을 보여 주는 방법도 있지만 심화 내용에 해당하므로 자세한 정보는 검색을 통해 보완하자.





정상적인 구글 검색 페이지

이 정도만 알고 있어도 Postman을 실습에 활용하기에는 충분하므로 무작정 사용법을 외우지 않아도 된다. Postman은 HTTP를 사용하기 위한 클라이언트 중 하나이다. 먼저 HTTP에서 우리가 하려는 행위가 무엇인지를 생각해 보자.

APPENDIX

B

과제 테스트 최종 문서화하기

과제 테스트 제출을 위한 문서 작성하기

애플리케이션을 개발하는 것 만큼이나 중요한 것이 바로 개발한 내용을 문서화하는 일이다. 문서화의 기본은 문서를 보는 사람이 해당 문서를 통해 유의미한 정보를 얻을 수 있도록 작성하는 것이다. 지금부터 책에서 개발한 상품 관리 애플리케이션을 과제로 제출하는 방법을 알아보자.

메일로 과제 제출하기

과제 테스트를 제출하려면 가장 먼저 과제 테스트가 제시된 이메일이나 과제 깃허브 레포지토리의 README.md 파일을 잘 읽어 봐야 한다. 해당 파일에는 과제에 대한 구체적인 제출 방법과 제출 시 반드시 지켜야 할 내용이 나와 있다.

보통 이메일로 과제를 제출하는 경우, 과제를 완료한 인텔리제이 프로젝트를 그대로 압축하여 보내게 된다. 그럼 면접관은 여러분이 보낸 인텔리제이 프로젝트를 열어서 과제를 확인하고, 실행할 것이다. 이때 흔히 문제가 되는 부분은 지원자가 프로젝트를 실행했던 환경과 면접관이 프로젝트를 실행하는 환경이 다를 수 있다는 점이다. 이는 다음과 같은 경우들이 대표적이다.

1. 자바 버전이 일치하지 않는 경우
2. Profile이 일치하지 않는 경우
3. 애플리케이션 실행에 데이터베이스 같은 외부 요소가 필요한 경우

1 자바 버전이 일치하지 않는 경우

과제 요구사항에서 자바의 버전을 지정했다면 반드시 해당 버전을 사용해야 한다. 자의적으로 해석하여 다른 버전을 사용해서는 안 된다. 만약 요구사항에서 자바의 버전을 지정해 주지 않았다면, 사용하고 싶은 버전을 사용하면 된다. 물론 그 버전을 선택한 이유를 면접에서 물어볼 수도 있으므로 적절한 답변을 준비해야 한다. 자바 버전은 인텔리제이 프로젝트의 pom.xml에서 확인할 수 있지만, 가급적 과제를 제출하는 메일의 내용에 사용한 자바의 버전을 명시해 주는 편이 좋다.

❷ Profile이 일치하지 않는 경우

과제 테스트를 제출할 때는 가능하면 새로 프로젝트를 열게 된 사람이 빠르고, 안전하게 애플리케이션을 실행하여 테스트 코드를 돌려 볼 수 있도록 준비해 두어야 한다. 따라서 Profile은 기본(Default)으로 바로 실행될 수 있도록 구성해야 한다. 간혹 application.properties가 아니라 인텔리제이 설정에서 실행 시 사용하는 Profile을 지정하는 경우가 있는데, 이렇게 사용할 Profile을 지정하면 프로젝트를 열게 된 다른 사람은 어떤 Profile이 적용되고 있는지 헷갈릴 수 있다. 이와 같이 불가피한 이유로 면접관이 Profile을 바꿔서 테스트해야 한다면, 이 내용 역시 반드시 제출하는 메일에 명시해야 한다.

❸ 애플리케이션 실행에 데이터베이스 같은 외부 요소가 필요한 경우

상품 관리 애플리케이션에서 사용할 Profile을 prod로 설정하는 경우에는 반드시 데이터베이스가 있어야 애플리케이션이 정상적으로 실행된다. 만약 요구사항에 의해 애플리케이션 외부에 있는 요소를 사용하는 과제를 제출하는 경우에는 외부 요소를 여러분의 PC와 똑같이 설정하는 방법을 제시해 주는 것이 좋다. 책에서 만든 상품 관리 애플리케이션의 데이터베이스를 예로 들어 보면, 도커로 데이터베이스를 실행시키고 도커에 접속하여 테이블을 생성했던 방법을 제출하는 메일에 설명해 주는 것이다.

다음과 같이 이러한 주의점을 모두 고려해 과제 제출 메일을 작성해 보자.

과제 제출 메일 예시

자바 버전은 JDK17을 사용했습니다. 구체적으로는 OpenJDK 중 Temurin JDK를 사용했습니다.
https://adoptium.net/temurin/releases/

Profile은 test Profile로 바로 리스트를 사용하는 상품 관리 애플리케이션을 실행시킬 수 있도록 구성했고, 만약 데이터베이스를 사용하는 Profile로 실행시키려면 application.properties에서 'spring.profiles.active' 값을 prod로 변경해 주세요.

Profile을 prod로 변경하는 경우 데이터베이스를 사용하게 되고, 데이터베이스는 다음과 같이 실행시킬 수 있습니다.

1. 도커로 MySQL 데이터베이스 실행: docker run --name some-mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=hanbit -d mysql:8.0.29 --character-set-server=utf8mb4 --collation-server=utf8mb4_general_ci
2. 데이터베이스 접속: mysql -u root -p
3. 스키마 생성: CREATE SCHEMA product_management;

```
4. 스키마 사용: USE product_management;
5. products 테이블 생성:
   CREATE TABLE products (
       id BIGINT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    price INT NOT NULL,
    amount INT NOT NULL
);
```

데이터베이스를 실행하고 테이블을 추가하는 과정을 과제로 제출해야 한다면 스크립트를 작성하여 면접관이 쉽게 실행시킬 수 있도록 제공하는 편이 좋다. 필수는 아니지만, 이와 같이 사소한 디테일은 개발자다운 모습을 어필하기에 좋다. 물론 환경의 차이 때문에 데이터베이스를 사용하는 과제가 흔히 출제되지는 않는다. 앞서 레포지토리 코드를 구현했던 것처럼 데이터베이스가 하는 역할을 컬렉션으로 그대로 대체할 수 있기도 하고, 데이터베이스를 사용하지 않는 과제로도 충분히 검증할 수 있기 때문이다.

추가로 이와 같은 내용을 메일로 답변하다 보면, 디테일한 내용이 메일 본문에 들어가게 되므로 면접관이 과제의 전체 맥락을 파악하기 어려워질 우려가 있다. 따라서 과제에 대한 설명이 필요하다면, 가급적 메일 본문보다는 별도의 PDF 문서를 만들어 첨부하자. 간혹 HWP 같은 확장자로 파일을 첨부해서 보내는 경우가 있는데, 정부 기관과 협업하는 회사가 아니라면 HWP 파일은 접근성이 떨어지므로 HWP로는 보내지 말아야 한다. 반면, 웹 브라우저에서도 열리는 PDF는 파일을 확인하기 위해 별도의 툴을 설치할 필요가 없어 편리하다.

노션Notion 링크를 첨부하는 방법도 있다. 이러면 이미 과제를 제출한 상태에서도 면접관이 열람하기 전까지는 내용을 수정할 수 있다는 장점이 있다. 다만, 애초에 노션에 작성하지 않았다면 PDF로 제출하는 편이 더 편하다.

그럼 깃허브로 과제를 제출할 때는 어떻게 문서화해야 할까? 작성할 내용은 메일로 제출하는 경우와 크게 다르지 않다. 제출하는 방법을 알아보자.

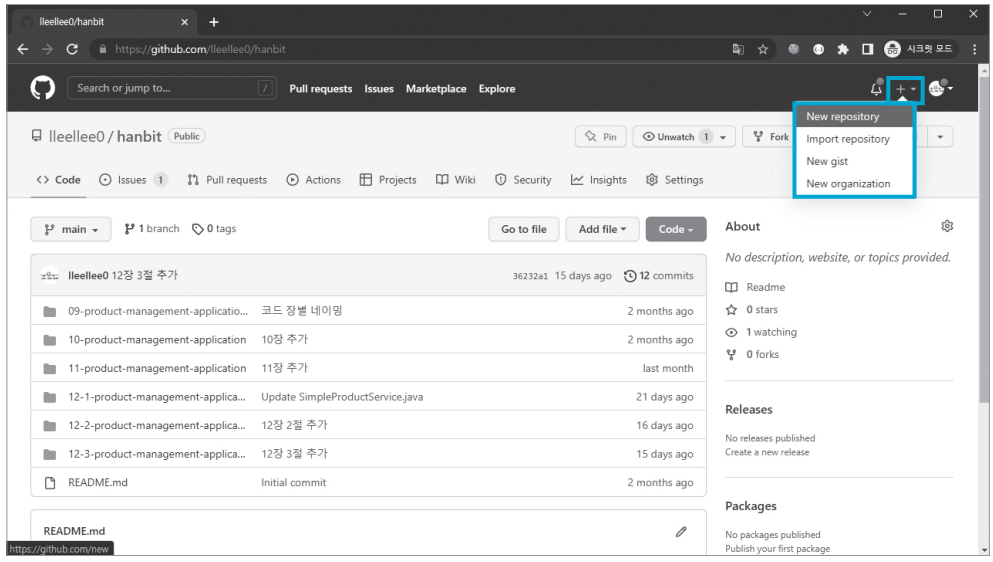
깃허브로 과제 제출하기

깃허브를 통한 과제 제출은 단순히 메일에 프로젝트를 압축하여 첨부하는 것과는 다르다. 또한 과제에 대한 설명을 작성하는 방법도 메일과는 다르기 때문에 익숙하지 않을 것이다. 하나씩 차근차근 알아보자.

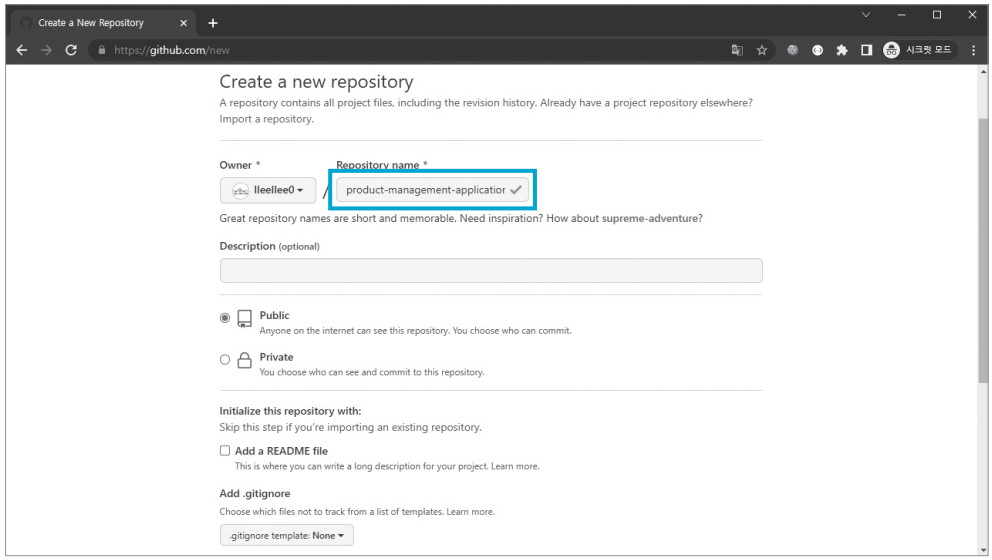
깃허브 레포지토리 만들기

먼저 해야 할 일은 깃허브 레포지토리를 만드는 것이다. 우선 깃허브 계정을 생성해야 한다. 어려운 과정이 아니므로 깃허브 웹사이트(github.com)에 접속해 계정을 생성하고 로그인해 보자.

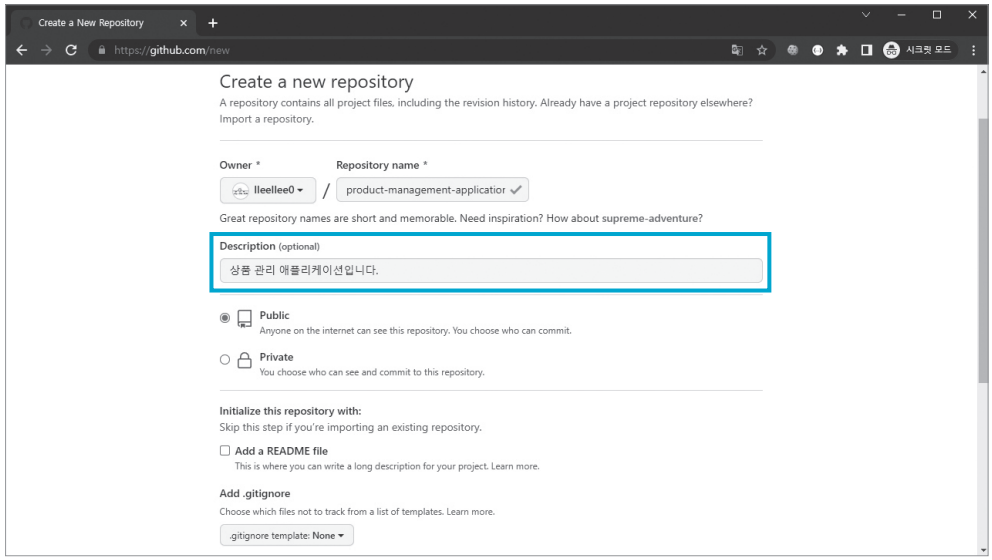
01. 로그인에 성공하면 깃허브 페이지 오른쪽 상단에 있는 버튼(+)을 클릭해 [New repository]를 선택한다.



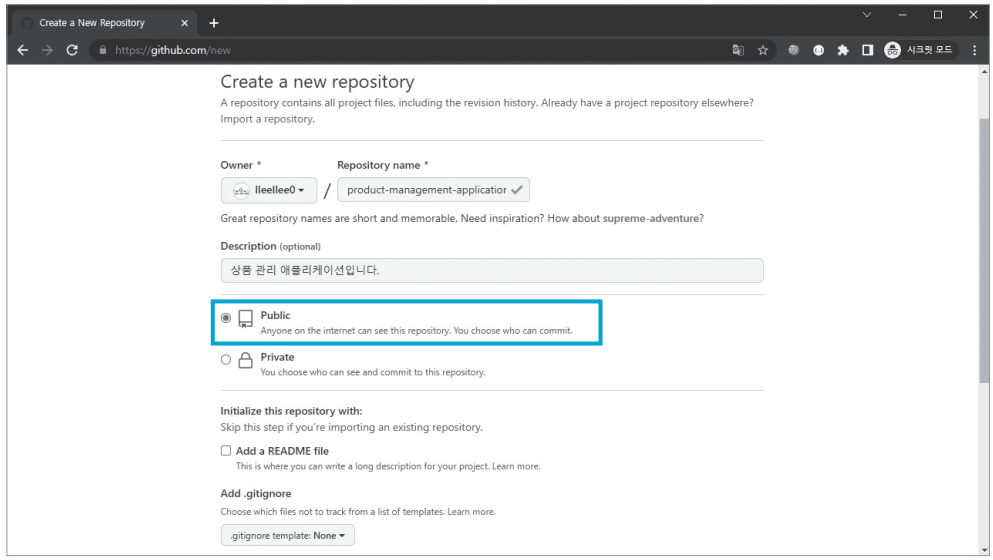
02. 다음과 같은 레포지토리 추가 페이지의 [Repository name]에 애플리케이션의 이름을 작성한다. 만약 과제로 제시된 이름이 있다면, 해당 과제 이름을 사용하면 된다. 여기에서는 상품 관리 애플리케이션을 만들었으므로 'product-management-application'이라고 입력했다.



03. [Description]에는 해당 레포지토리에 대한 설명을 입력한다. 한글로 입력해도 되므로 해당 레포지토리를 설명할 수 있는 내용을 자유롭게 입력하면 된다.

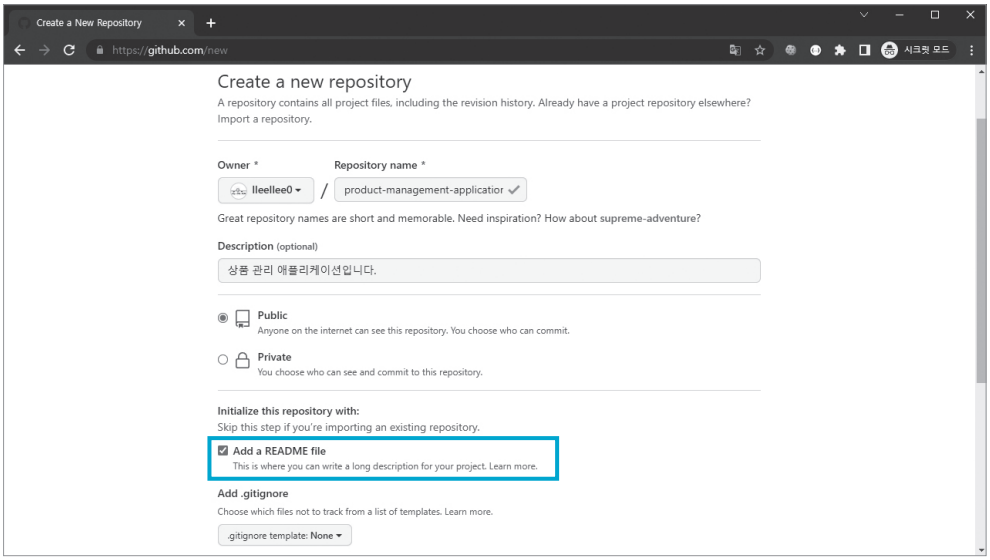


04. [Public]과 [Private]으로 공개 범위를 설정할 수 있는데, Public으로 설정하면 모든 사람에게 해당 레포지토리가 공개되므로 누구나 해당 레포지토리를 볼 수 있다. 여기서는 Public으로 생성하여 진행한다.

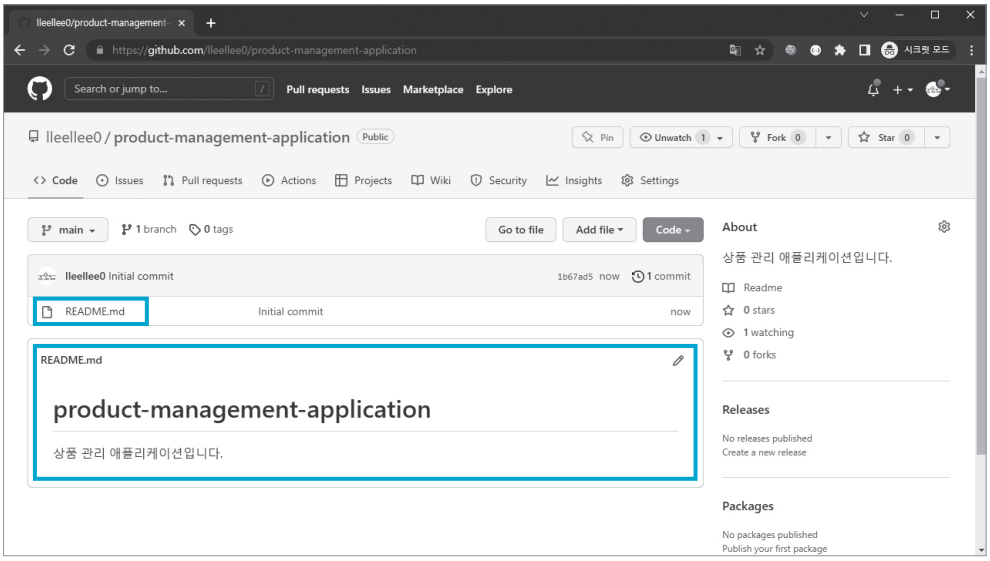


NOTE 일반적으로 과제 테스트를 진행할 때는 Private으로 레포지토리를 생성한 후 면접관에게 초대 링크를 전송하거나, 반대로 면접관이 Private으로 레포지토리를 생성한 후 면접자를 해당 레포지토리로 초대하여 과제를 진행한다.

05. 다음으로 [Initialize this repository with:]라는 항목에서는 'Add a README file'이라는 항목만 체크한 후 [Create repository]를 선택해 레포지토리를 생성한다.



06. 다음과 같이 생성된 레포지토리 페이지에서 박스로 표시된 부분이 바로 애플리케이션 개발 내용을 확인할 수 있는 'README.md' 파일이다.

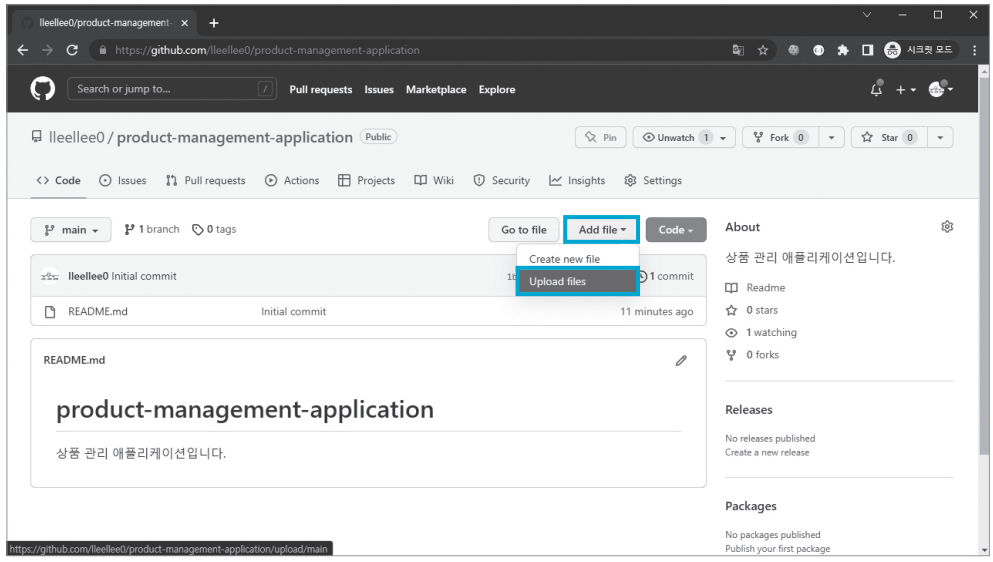


이 파일에 내용을 작성하기 위해서는 '마크다운Markdown'이라는 문법을 사용해야 한다. README.md의 뒤에 붙어 있는 '.md'가 바로 마크다운 파일이라는 의미이다. 마크다운 문법과 README.md 파일 작성은 23쪽에서 살펴본다.

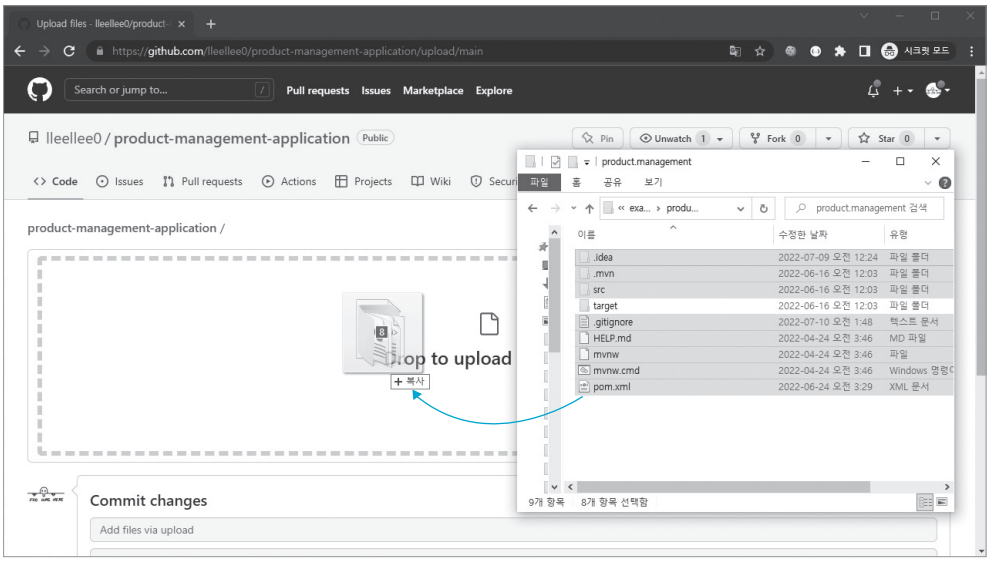
깃허브 레포지토리에 코드 업로드하기

이어서 레포지토리에 코드를 업로드해 과제를 첨부하는 방법을 알아보자.

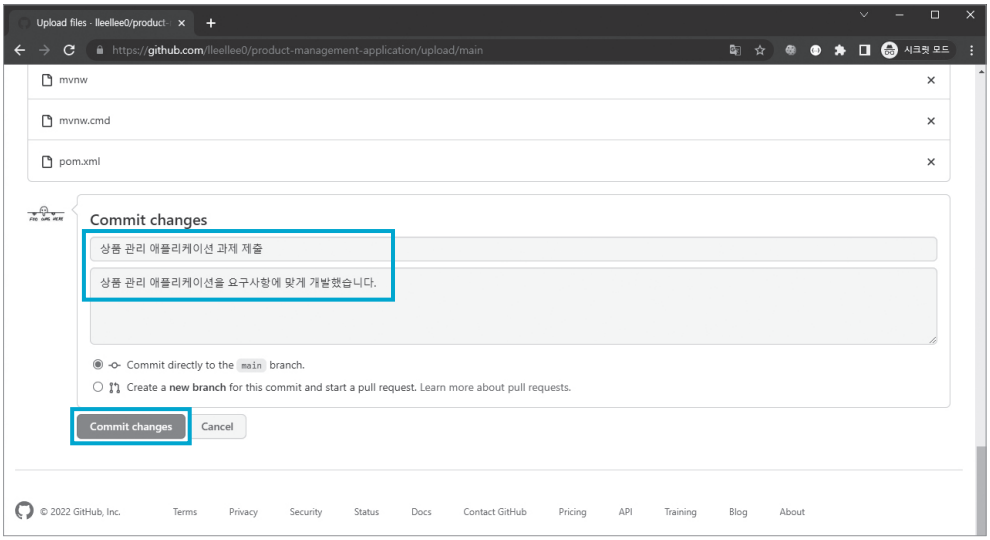
01. 생성한 레포지토리 페이지에서 [Add file] 버튼을 클릭해 [Upload files]를 선택한다.



02. 다음과 같이 레포지토리에 추가하려는 파일을 드래그하여 추가한다. 프로젝트 폴더에 있는 상품 관리 애플리케이션 파일을 추가하면 된다. **주의할 점은 target 폴더를 빼고 추가해야 한다는 것이다.**

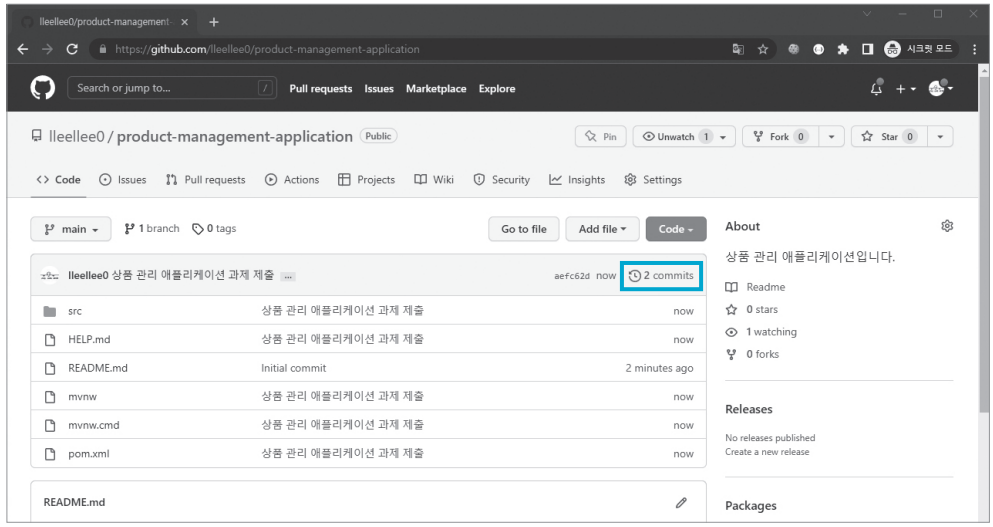


03. 파일을 업로드하고 화면을 아래로 스크롤해 내리면 [Commit changes]라는 항목을 볼 수 있다. 2개의 입력란 중 첫 번째는 '커밋 요약Commit Summary'을 작성하는 부분이고, 두 번째는 '커밋 설명Commit Description'을 작성하는 부분이다. 여기서는 다음과 같이 내용을 작성하고 [Commit changes] 버튼을 클릭한다.

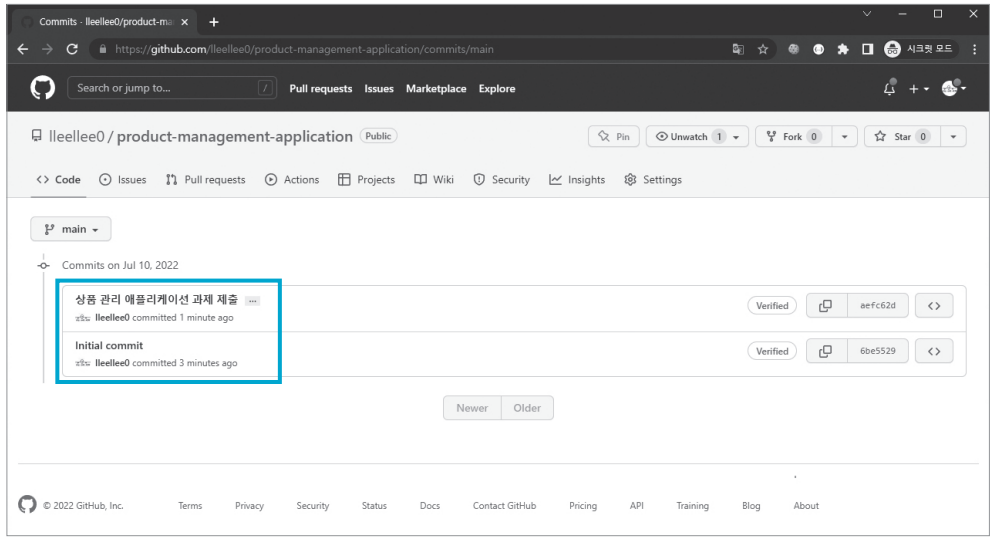


이렇게 코드 업로드를 완료했다. 이해를 돕기 위해 몇 가지 설명을 덧붙인다.

커밋Commit은 레포지토리에 코드가 변경되는 단위를 의미한다. 처음 레포지토리를 생성할 때 첫 번째 커밋이 생겼고, 그 다음으로 조금 전 프로젝트 코드를 업로드하면서 두 번째 커밋이 생겼다. 이 내용은 커밋 히스토리를 살펴보면 된다. 다음과 같이 '2 commits'라고 되어 있는 부분을 클릭해 보자.



프로젝트 코드를 올릴 때 작성했던 커밋 요약을 확인할 수 있으며, 더보기 버튼을 누르면 앞서 작성한 커밋의 설명도 볼 수 있다.



다시 파일을 드래그해 업로드하던 때로 돌아가 보자. [target] 디렉토리를 함께 업로드하지 않은 이유는 [target] 디렉토리의 역할을 생각해 보면 알 수 있다. [target] 디렉토리는 소스코드를 포함하고 있는 디렉토리가 아니라 애플리케이션이 빌드된 후 실행되기 위해 존재하는 .class 파일들이 위치하는 곳이다. 깃허브에 소스코드만 올려 두고, 프로젝트를 다운로드해서 빌드하면 똑같이 .class 파일을 생성할 수 있으므로 빌드 과정에서 생성되는 파일을 깃허브에 올릴 필요는 없다.

여기서 잠깐

.gitignore

[target] 디렉토리처럼 불필요한 파일을 넣지 않기 위해 사용할 수 있는 방법은 .gitignore 파일을 추가하는 것이다. 그러나 실습에서는 깃허브에서 직접 파일을 업로드하는 경우에는 적용되지 않기 때문에 .gitignore 파일을 추가하지 않았다. .gitignore 파일을 적용하려면 깃허브에 직접 파일을 업로드하는 것이 아니라, Git을 통해 로컬에서 커밋을 생성하고 깃허브에 업로드(푸시)해야 한다.

소스코드와 무관하거나, 소스코드가 빌드되는 과정에서 함께 생성되기 때문에 .gitignore 파일에는 깃허브에 업로드될 필요가 없는 파일도 넣어 주는 것이 좋다. 다음 파일은 인텔리제이에서 자동으로 생성해 주는 .gitignore 파일이다. 이 파일은 예시로만 참고하고 파일의 내용까지 외울 필요는 없다.

인텔리제이에서 자동으로 생성해 주는 .gitignore 파일

```
HELP.md
target/
!.mvn/wrapper/maven-wrapper.jar
!**/src/main/**/target/
!**/src/test/**/target/

### STS ###
.apt_generated
.classpath
.factorypath
.project
.settings
.springBeans
.sts4-cache

### IntelliJ IDEA ###
.idea
*.iws
```

```
*.iml
*.ipr

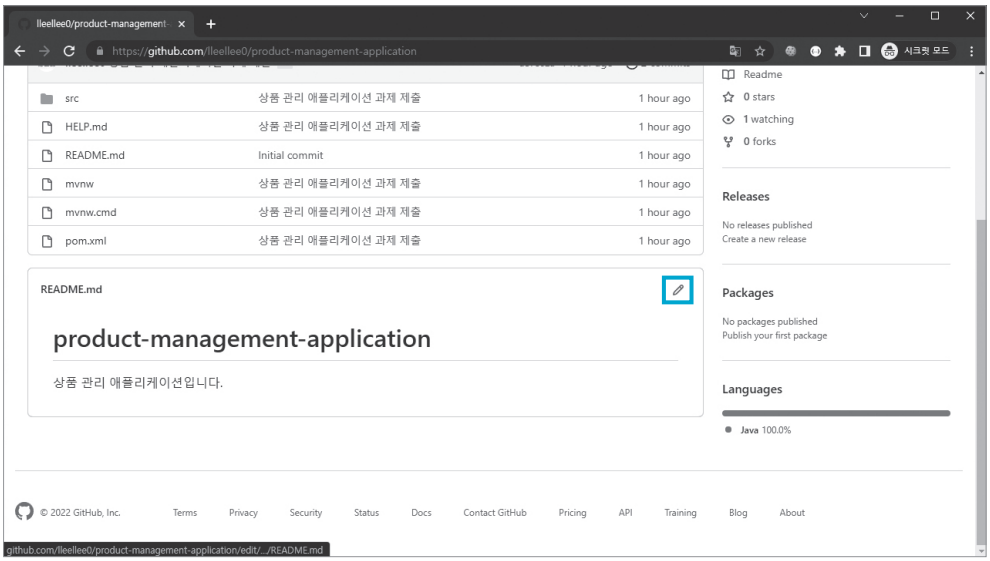
### NetBeans ###
/nbproject/private/
/nbbuild/
/dist/
/nbdist/
/.nb-gradle/
build/
!**/src/main/**/build/
!**/src/test/**/build/

### VS Code ###
.vscode/
```

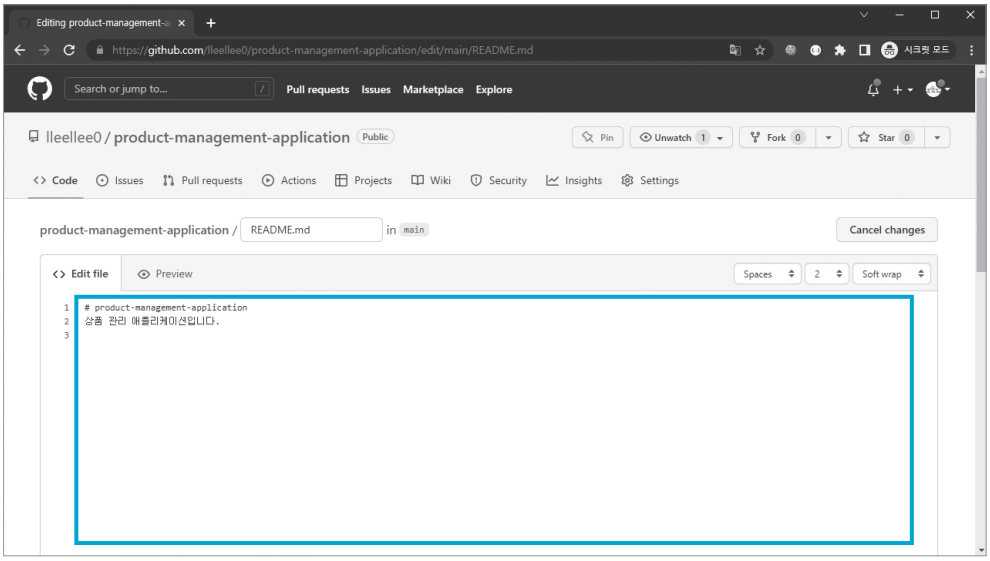
README.md 작성하기

이어서 과제에 대한 설명을 작성해 보자. 앞서 이야기했던 것처럼 마크다운이라는 문법으로 작성해야 하므로 구글에서 'markdown'이라고 검색하면 쉽게 다양한 문법을 찾을 수 있다. README.md 파일을 작성해 보면서 사용된 문법을 살펴보자.

01. 먼저 README.md 파일에서 연필 모양(✎) 아이콘을 클릭해 README.md 파일을 수정하자.



02. README.md 파일을 수정할 수 있는 입력 란에 다음과 같이 작성한다. 앞서 메일에 작성한 내용을 기반으로 마크다운을 적용한 것이다.



NOTE 앞서 레포지토리를 생성할 때 'Description'에 작성한 내용이 README.md 파일에 자동으로 입력된다.

README.md 작성 예시

```
# product-management-application

### JDK
자바 버전은 JDK17을 사용했습니다.

구체적으로는 OpenJDK 중 [Temurin JDK](https://adoptium.net/temurin/releases/)를 사용했습니다.

### Profile
Profile은 test Profile로 리스트를 사용하는 상품 관리 애플리케이션을 바로 실행시킬 수 있도록 구성했습니다. 만약 데이터베이스를 사용하는 Profile로 실행시키려면 application.properties에서 'spring.profiles.active' 값을 prod로 변경해 주세요.

Profile을 prod로 변경하는 경우 데이터베이스를 사용하게 되고, 데이터베이스는 다음과 같이 쉽게 실행시킬 수 있습니다.

#### 도커로 MySQL 데이터베이스 실행
```
docker run --name some-mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=hanbit -d
mysql:8.0.29 --character-set-server=utf8mb4 --collation-server=utf8mb4_general_ci
```

#### 데이터베이스 접속
```
mysql -u root -p
```

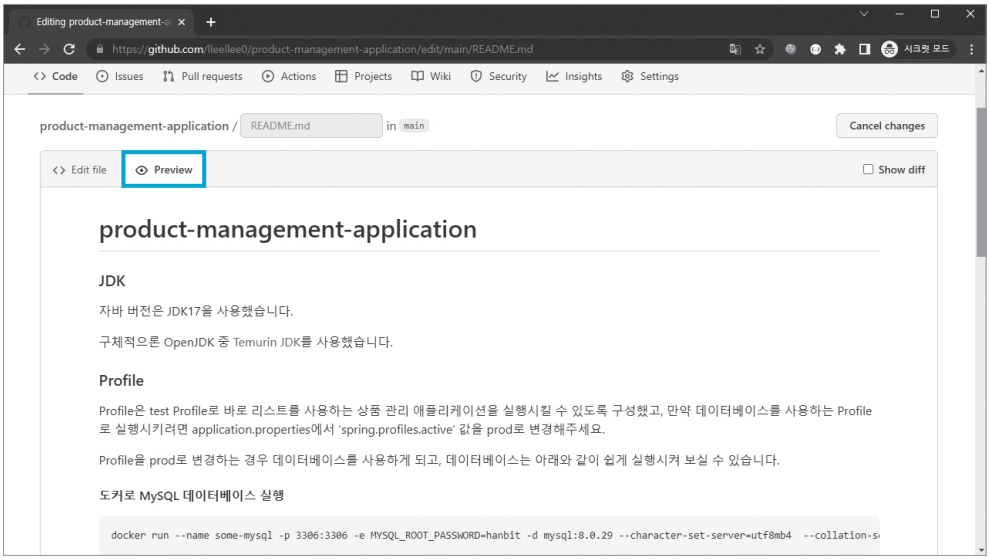
#### 스키마 생성
```
CREATE SCHEMA product_management;
```

#### 스키마 사용
```
USE product_management;
```
#### products 테이블 생성
```SQL
CREATE TABLE products (
    id BIGINT PRIMARY KEY NOT NULL AUTO_INCREMENT,
```
```

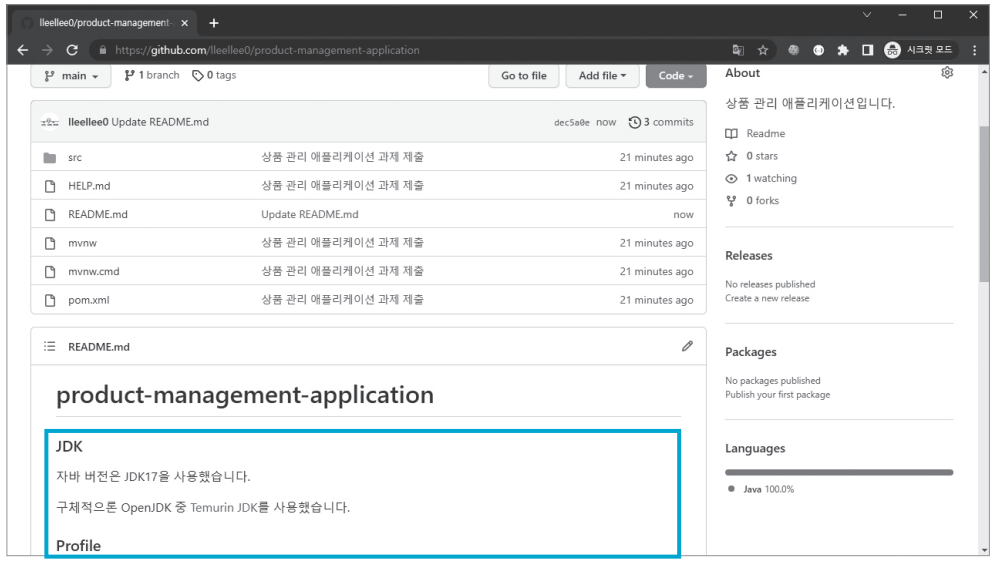
```
    name VARCHAR(100) NOT NULL,
    price INT NOT NULL,
    amount INT NOT NULL
);
```

NOTE 다음 링크에 이 내용을 그래도 작성해 두었다. 직접 입력하기 어렵다면 복사하여 사용하자.
https://raw.githubusercontent.com/lleellee0/java-for-backend/main/APPENDIX B/README.md

03. [Preview] 탭을 클릭하면 작성한 마크다운 파일을 미리보기할 수 있다. 다른 사람들에게 입력된 내용이 어떻게 보일지 미리 확인하는 것이다.



04. 이번에도 커밋 요약 메시지가 미리 들어가 있을 것이다. 아래로 스크롤해 [Commit chages] 버튼을 클릭하고 커밋을 완료하면 다음과 같이 레포지토리 메인 페이지에서 프로젝트에 대한 설명을 확인할 수 있다.



마크다운 문법

README.md 작성 예시에 사용한 마크다운 문법인 글머리, 링크, 코드에 대해 알아보자.

글머리

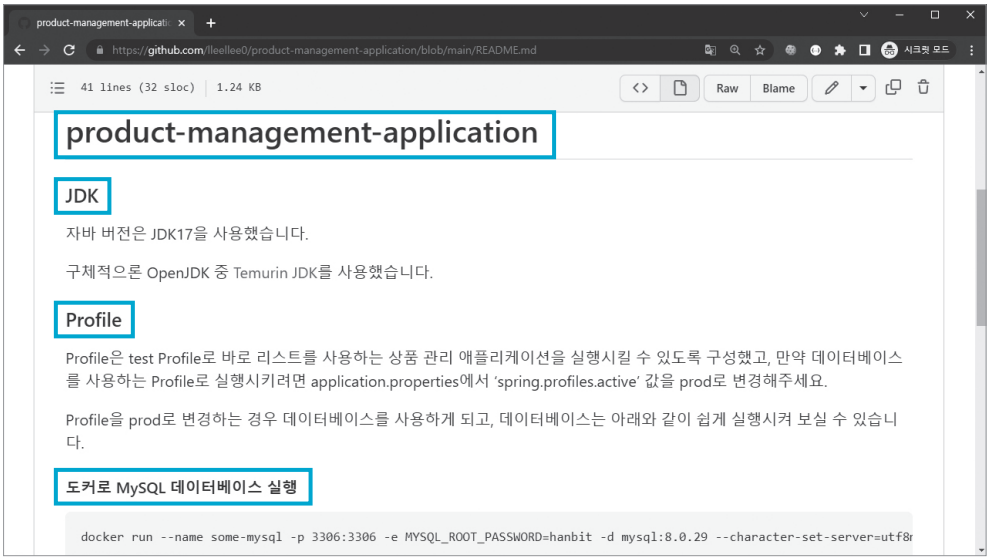
글머리는 #을 붙여서 표현한다. 앞서 예시에서는 세 가지 종류의 글머리를 사용했다. #을 한 번만 붙이기도 하고 세 번, 네 번 붙이기도 했다. 글머리는 총 6단계로 표현할 수 있는데, #의 개수가 늘어날수록 글머리의 크기는 작아진다. 즉, #은 1개부터 6개까지 붙일 수 있으며, 1~2개만 붙였을 때는 글머리 아래에 수평선이 표기된다.

```
# product-management-application

### JDK

### Profile

#### 도커로 MySQL 데이터베이스 실행
```

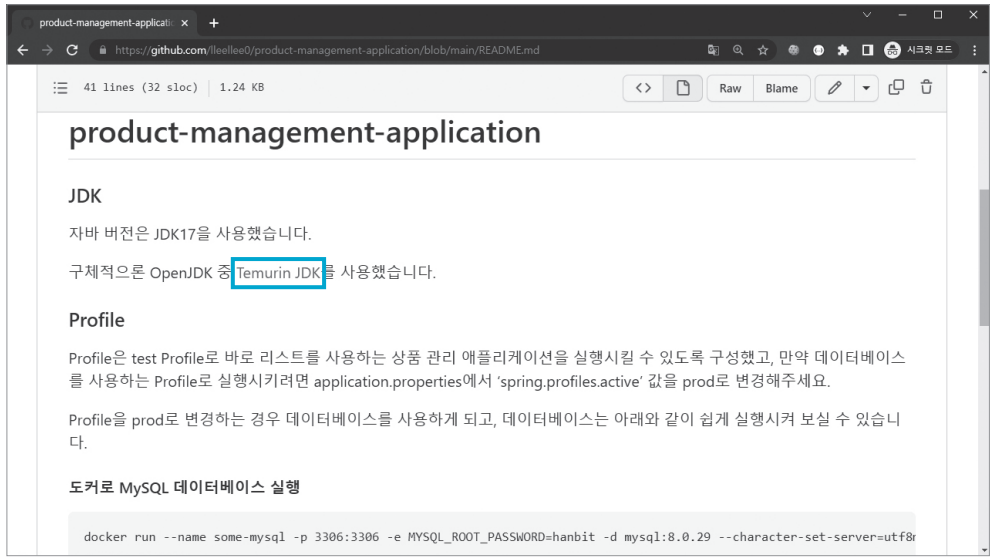


글머리 적용

링크

작성 예시에서는 'Temurin JDK'라는 글자를 대괄호로 감싸고, 이어서 소괄호로 감싼 URL을 입력했다. 이는 링크를 추가하는 방법이다.

```
구체적으론 OpenJDK 중 [Temurin JDK](https://adoptium.net/temurin/releases/)를 사용했습니다.
                      표시되는 글자          링크 주소
```



코드

코드는 백틱Backtick 문자 `를 코드의 시작과 끝에 각각 3개씩 붙여서 나타낸다. 백틱 문자는 키보드에서 물결표(~)와 같은 키에 위치한다. products 테이블을 생성하는 SQL 코드에서 백틱 문자 옆에 SQL이라고 적어 주면 해당 코드의 문법에 맞게 자동으로 하이라이트 표시가 된다.

````
#### 스키마 사용
```
USE product_management;
```

#### products 테이블 생성
```SQL
CREATE TABLE products (
    id BIGINT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    price INT NOT NULL,
    amount INT NOT NULL
);
```
````

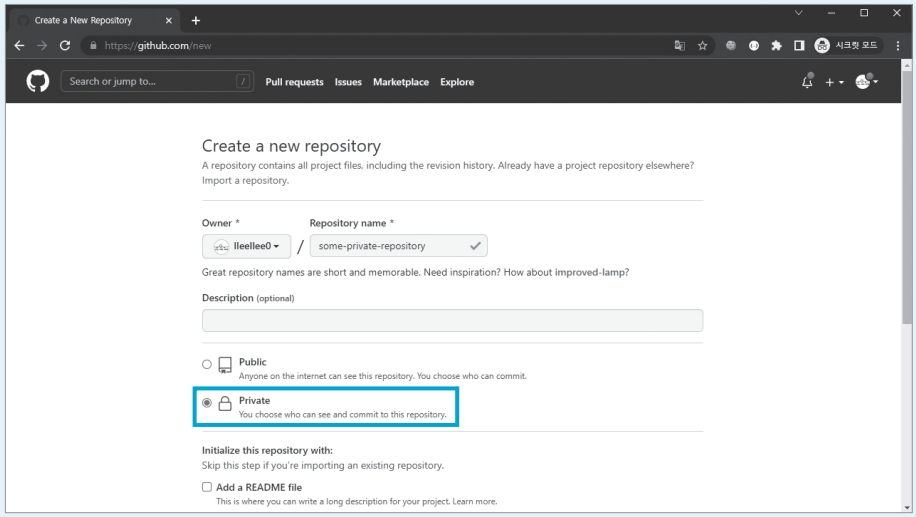
메일이나 깃허브를 통해 간단하게 과제를 제출하는 방법을 알아봤다. 모든 과제는 출제 가이드 대로 제출하는 것이 우선이다. 또한 깃과 깃허브는 이보다 훨씬 다양한 기능을 제공한다. 백엔드 개발자가 되기 위해 반드시 알아야 할 내용이므로 별도의 도서나 강의를 통해 학습을 보완하자.

여기서 잠깐

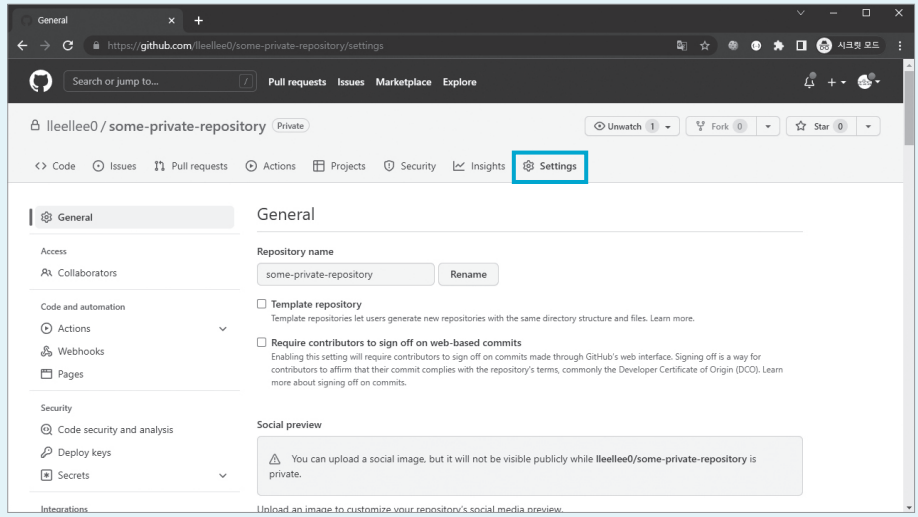
레포지토리를 Private으로 생성하고 면접관 초대하기

과제가 외부에 공개되기를 원하지 않는 경우 레포지토리를 Private으로 생성하라는 요구사항이 제시될 수 있다. 레포지토리를 Private으로 생성하는 경우 면접관을 초대해야 면접관이 과제를 볼 수 있으므로 Private 레포지토리 초대 방법을 알아 두자.

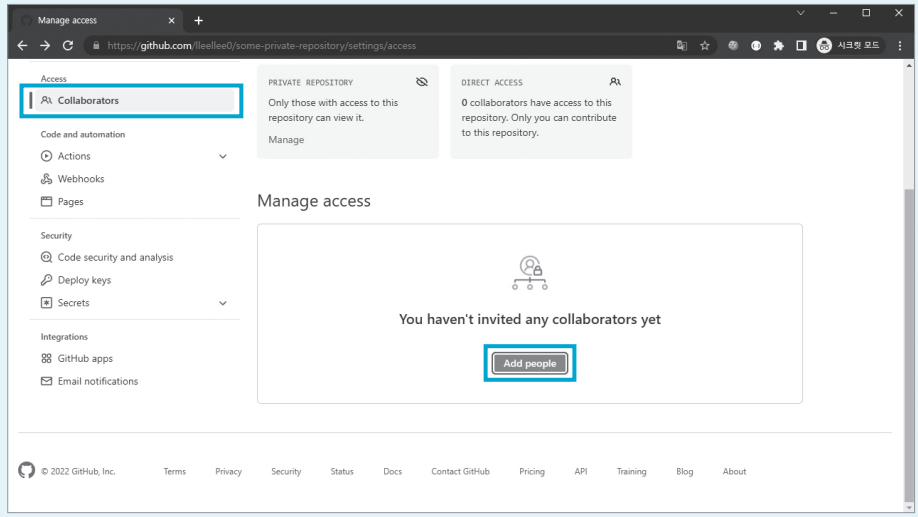
01. 레포지토리 생성 시 [Private]을 선택한다.



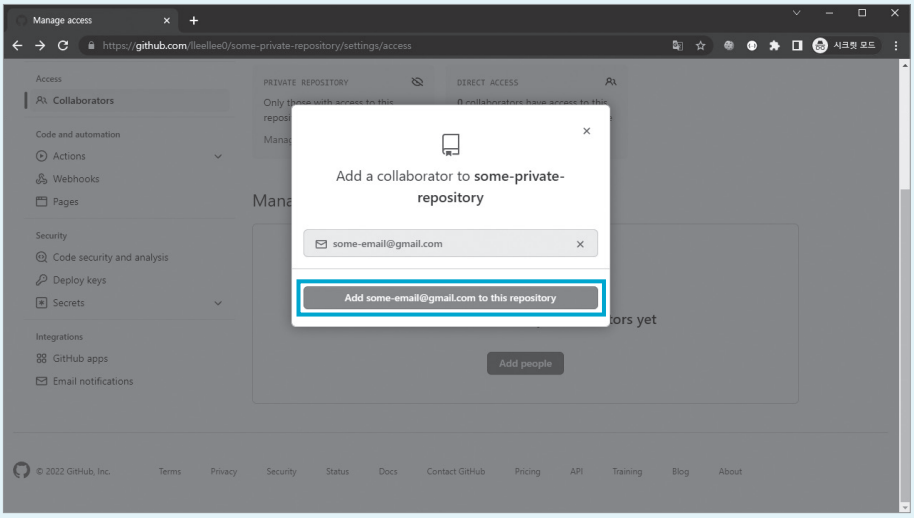
02. 생성된 레포지토리 페이지 상단에 있는 [Settings]을 클릭한다.



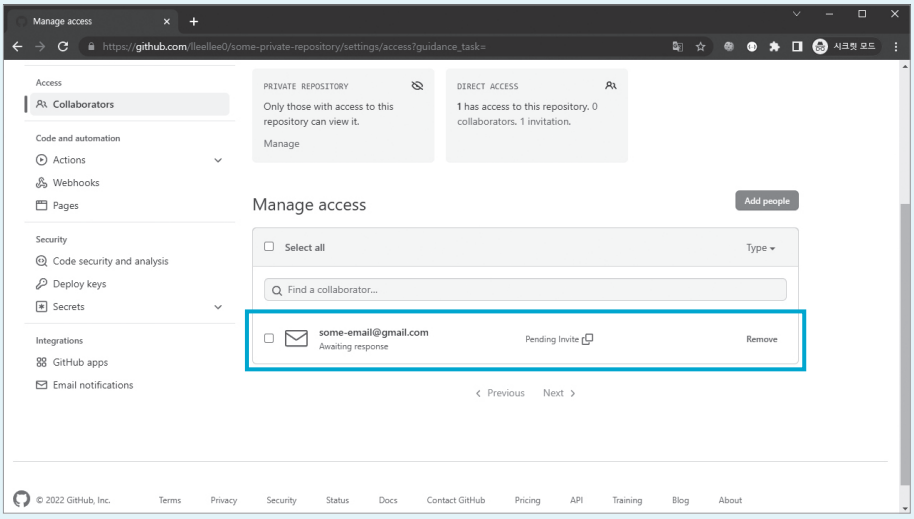
03. 왼쪽 메뉴에서 [Access] – [Collaborators]를 선택하고, [Manage access]의 [Add people] 버튼을 클릭한다.



04. 초대할 대상의 이메일 주소를 입력하고, [Add some 이메일 주소 to this Repository] 버튼을 클릭하여 초대를 완료한다.



05. 다음과 같이 초대한 이메일 주소가 표시되는 것을 확인할 수 있다.



방금처럼 메일 주소로 초대할 수도 있고, 깃허브 계정을 검색하여 초대할 수도 있다. 어떤 방법을 사용해도 괜찮지만, 메일 주소를 잘못 입력했을 수도 있기 때문에 상대방을 초대한 후에는 깃허브 초대 메일이 제대로 갔는지 면접관에게 이메일을 보내 확인해 보는 것이 좋다. 면접관이 해당 메일로 초대를 수락하면 그때부터 Private 레포지토리를 볼 수 있다.

과제의 관점으로 상품 관리 애플리케이션 만들기

책에서는 백엔드 애플리케이션 개발에 필요한 개념을 설명하기 위해 요구사항에는 명시되지 않을 불필요한 코드도 추가했다. 실제로 과제 테스트를 풀이하는 상황에는 맞지 않을 수 있으므로 과제 테스트를 제출하는 상황을 가정하면 어떤 식으로 상품 관리 애플리케이션을 만들 수 있을지 알아보자.

요구사항 분석과 구현

과제를 제시받으면 어디에서부터 과제를 시작해야 하는지를 먼저 파악해야 한다. 과제에 대한 프로젝트 파일이나 깃허브 레포지토리가 미리 제공됐을 수도 있다. 만약 그런 경우가 아니라면 책에서 했던 것처럼 스프링 이니셜라이저를 통해 스프링 부트 프로젝트를 생성하는 것부터 시작해야 한다.

프로젝트 생성

프로젝트를 생성한 후에는 당연히 인텔리제이로 프로젝트를 열어 줘야 한다. 그 후에는 4개의 패키지(application, domain, infrastructure, presentation)부터 만든다. 책에서 소개한 것처럼 패키지를 나누지 않아도 된다. 그러나 레이어드 아키텍처를 사용하면 레이어별로 역할을 나눠서 개발할 수 있고, 테스트 코드 작성에도 유리하다는 점을 기억해 두자.

데이터 클래스 및 필드 생성

패키지까지 생성한 후에는 요구사항의 어떤 내용에 주목해야 할까? 어떤 기능이 있는지 살펴보고 기능부터 구현할 수도 있지만, 백엔드 애플리케이션 개발에 입문하는 사람들이 가장 먼저 시작하기 좋은 지점은 바로 '데이터'이다. 즉, 상품 관리 애플리케이션에서는 '상품'을 중심으로 개발을 시작하는 편이 좋다. 따라서 Product 클래스부터 만들고 필드를 만들자. Getter, Setter, 생성자를 바로 추가하지 않고, 이들이 필요한 상황이 생겼을 때 하나씩 추가하자.

컨트롤러 생성

다음으로는 컨트롤러와 함께 Product에 대한 DTO를 만들자. 사실 테스트 코드 작성에 능숙하다면 컨트롤러보다 서비스부터 만들어서 더 빠르게 애플리케이션을 개발할 수도 있다. 그러나 아직 테스트 코드보다 Postman을 통한 테스트에 더 익숙하다면 컨트롤러를 먼저 만드는 것을 추천한다.

컨트롤러를 만들었다면 @RequestMapping을 추가해 주자. 지금까지 배운 내용을 떠올려 보면 어떤 URL 경로, 어떤 HTTP 메서드를 사용해야 할지 쉽게 정의할 수 있을 것이다. 아직 부족하다면 구글 등을 통해 'REST API의 URL 설계 방법'을 검색하여 여러 가지 사례를 확인해 보자.

이제 Postman에서 컨트롤러로 데이터를 전달하고, 반대로 컨트롤러에서 Postman 쪽으로 데이터를 반환할 수 있도록 코드를 구현하자. 이를 구현했다면 컨트롤러에서 서비스 쪽 코드를 호출할 수 있도록 서비스 쪽 코드를 추가하면 된다.

서비스 개발

다음은 서비스와 레포지토리를 함께 개발하는 것을 추천하는데, 레포지토리를 모킹하여 개발하는 것이 아니라면 실제 데이터가 저장되고 조회되는 것을 눈으로 확인하면서 개발하는 것이 편리하기 때문이다. 필자는 어떤 데이터에 대한 CRUD 기능을 구현해야 한다면, 보통 데이터를 추가하는 것부터 구현한다. 데이터를 조회하거나 수정하거나 삭제하려면 먼저 데이터가 저장되어 있어야 하기 때문이다.

데이터 추가 다음으로는 조회 기능을 구현하여 데이터 추가 기능이 잘 구현되었는지 확인해 보자. 수정, 삭제 기능을 구현할 때도 조회 기능을 통해 계속 확인하면서 개발하자.

레포지토리 개발

레포지토리는 애플리케이션 외부 요소에 의존하는 경향이 많으므로 테스트 코드 혹은 서로 다른 Profile인 경우, 서로 다른 구현체를 사용하게 되는 경우가 많다. 따라서 처음부터 인터페이스를 두고 구현하게 된다.

일반적으로 과제 테스트의 요구사항에서는 데이터베이스와 같은 외부 요소를 사용하기보다는 ListProductRepository와 같은 컬렉션을 활용하도록 제시하는 경우가 많다. 컬렉션을 활용한 레포지토리라고 하더라도 면접자가 코드에 어떻게 추상화를 적용했고, 어떻게 레포지토리가 확장될

것을 고려했는지 충분히 검증할 수 있기 때문이다. 컬렉션을 활용해 레포지토리를 구현할 때는 멀티 스레드 환경에서 동시성 문제가 발생하지 않도록(스레드 세이프하도록) 신경써서 코드를 작성하자.

컬렉션을 활용한 레포지토리 코드를 구현할 때 여러분은 아마 다음과 같은 작업을 반복하게 될 것이다(다음은 상품 수정 기능을 구현하는 중이라고 가정한 작업이다).

1. 코드 수정
2. 애플리케이션 재시작
3. 상품 추가 API 호출 (Postman)
4. 상품 수정 API 호출 (Postman)
5. 상품 조회 API 호출 (Postman)
6. 상품 정보가 잘 수정되었는지 5의 결과로 확인

즉, 상품 수정 기능을 개발하면서 애플리케이션도 재시작해 주고, 상품 추가 API와 상품 조회 API를 반복적으로 호출하며 테스트해야 하는 것이다. 이 작업을 테스트 코드로 만든다면 분명 빠르게 상품 수정 기능이 올바르게 구현되었는지 테스트해 볼 수 있다. 테스트 코드는 코드 변경에 대해 기존과 같은 동작을 하는지 보장하는 것 외에도 반복적이고 빠른 실행으로 개발의 속도를 올려줄 수 있다.

예외 정의

서비스와 레포지토리 코드를 어느 정도 구현해 줬다면, 예외를 정의할 차례이다. 요구사항에는 예외를 정의하라는 말이 없더라도 애플리케이션의 구현상 예외를 정의하는 것이 바람직하다. 그리고 예외에 대한 핸들러도 추가해 주자.

여기까지 구현했다면 애플리케이션을 돌아보고 여러 가지 방법으로 애플리케이션을 테스트해 보자. 요구사항을 잘못 이해하고 구현한 부분은 없는지, 빠뜨린 요구사항은 없는지 등을 확인해 봐야 한다.

다르게 구현할 부분

책에서 상품 관리 애플리케이션을 구현할 때 설명한 것과 조금 다르게 구현할 만한 몇 가지 부분을 짚어 본다. 크게 다음과 같은 세 가지 내용을 살펴보자.

- ListProductRepository처럼 List를 사용하여 Repository를 구현하기보다는 Map을 활용하여 Repository를 구현하면 어떨까?
- 데이터베이스를 사용하는 경우 SimpleProductService 쪽에 @Transactional 애너테이션을 활용하면 어떨까?
- 적절한 곳에 로그를 추가하면 어떨까?

Repository를 Map으로 구현

다음의 ListProductRepository 코드를 보면서 **왜 List를 활용하는 Repository가 아니라 Map을 활용하도록 구현하려고 하는지**에 대해 살펴보자.

ListProductRepository.java

```
(생략)

public class ListProductRepository implements ProductRepository {

    private List<Product> products = new CopyOnWriteArrayList<>();
    private AtomicLong sequence = new AtomicLong(1L);

    public Product add(Product product) {
        product.setId(sequence.getAndAdd(1L));

        products.add(product);
        return product;
    }

    public Product findById(Long id) {
        return products.stream()
            .filter(product -> product.sameId(id))
            .findFirst()
            .orElseThrow(() -> new EntityNotFoundException("Product를 찾지 못했습니다."));
    }

    public List<Product> findAll() {
        return products;
    }

    public List<Product> findByNameContaining(String name) {
```

```
        return products.stream()
            .filter(product -> product.containsName(name))
            .toList();
    }

    public Product update(Product product) {
        Integer indexToModify = products.indexOf(product);
        products.set(indexToModify, product);
        return product;
    }

    public void delete(Long id) {
        Product product = this.findById(id);
        products.remove(product);
    }

}
```

볼드로 표현한 부분에 주목해야 한다. 먼저 위쪽에 있는 findById에서 products가 만약 List가 아니라 Map이었다면, 상품 번호(id)를 Map의 Key로 사용했을 것이다. 그 후에는 stream을 생성하지 않고도 손쉽게 key를 통해 찾고 싶은 id의 Product 인스턴스를 조회할 수 있다. id를 통해 특정 Product를 찾아내는 요청이 자주 발생한다면 List보다 Map을 사용하는 편이 더 적절할 수 있다.

이보다는 아래쪽에 있는 코드가 더 문제가 되는데, 앞서 우리는 책에서 update 메서드가 스레드 세이프하지 않다고 배웠다. 만약 Map을 사용한다면 수시로 변경될 수 있는 List의 인덱스를 사용하지 않고 Key를 통해 바로 해당 Product를 수정할 수 있었을 것이다.

그럼 Map으로 변경한다면 어떤 구현체를 사용해야 할까? 여러 가지 방법이 있지만, 추천하는 방법은 ConcurrentHashMap을 사용하는 것이다. ConcurrentHashMap은 스레드 안전성이 있는 Map 구현체 중 하나이다. 물론 ConcurrentHashMap으로 변경하면 그 아래에 있는 메서드들도 함께 변경해야 한다. 이때 주의할 점은 findAll 메서드와 findByNameContaining 메서드의 반환 타입을 여전히 List〈Product〉인 상태로 만들어야 한다는 것이다. 그래야 인터페이스를 유지할 수 있기 때문이다.

SimpleProductService가 가진 메서드에 @Transactional 붙여 주기

다음으로 **SimpleProductService 쪽에 @Transactional 애너테이션을 추가하는 것**에 대해 알아보자. @Transactional 애너테이션은 앞서 테스트 코드를 작성할 때 테스트 메서드 위에 달아 줬다. 해당 애너테이션은 테스트가 실행되더라도, 데이터베이스에 테스트 데이터가 실제로 반영되는 것을 막아 주는 기능을 한다. 해당 기능은 트랜잭션을 활용하여 가능하며, 이것을 그대로 서비스에도 적용시킬 수 있다. 그 방법을 알아보자.

우선 다음과 같이 코드를 수정하면 prod 프로파일이 적용되고, 데이터베이스를 사용하게 된다.

application.properties

```
spring.profiles.active=prod
```

SimpleProductService.java

```
package kr.co.hanbit.product.management.application;

(생략)

import org.springframework.transaction.annotation.Transactional;

import java.util.List;

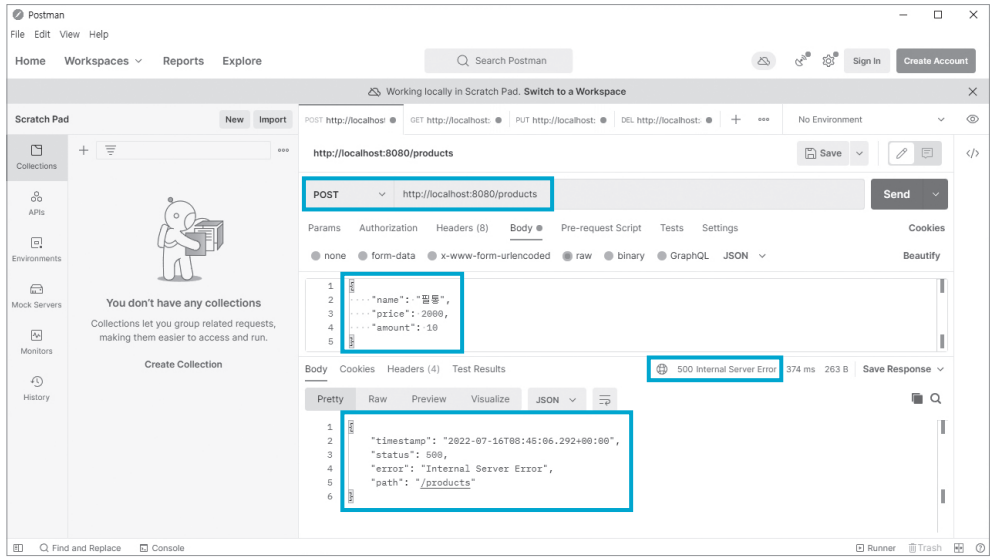
@Transactional(readOnly = true)
@Service
public class SimpleProductService {

(생략)

}
```

코드는 SimpleProductService의 클래스 위에 @Transactional 애너테이션을 추가해 줬다. 이렇게 클래스 위에 @Transactional 애너테이션을 선언하면 해당 클래스에 존재하는 모든 메서드에 @Transactional 애너테이션이 적용된 것과 같은 효과를 가진다. 여기에서 해당 애너테이션의 파라미터로 'readOnly = true'라는 옵션을 주었는데, 이 옵션은 해당 트랜잭션이 오직 읽기만 가능한 트랜잭션임을 의미한다. 만약 해당 트랜잭션 내에서 데이터를 수정하려고 하면 예외가 발생하고, 데이터가 수정되지 않는다.

이 상태에서 다음과 같이 애플리케이션을 시작하고 Postman으로 상품 데이터를 추가해 보자.



그러면 상품 추가가 실패하면서 다음과 같은 에러 로그가 나타난다.

에러 로그

```
2022-07-16 17:45:06.285 ERROR 82824 --- [nio-8080-exec-1] o.a.c.c.C.[.[.[/].
[dispatcherServlet]      : Servlet.service() for servlet [dispatcherServlet]
in context with path [] threw exception [Request processing failed; nested
exception is org.springframework.dao.TransientDataAccessResourceException:
PreparedStatementCallback; SQL [INSERT INTO products (name, price, amount) VALUES
(?, ?, ?)]; Connection is read-only. Queries leading to data modification are not
allowed; nested exception is java.sql.SQLException: Connection is read-only.
Queries leading to data modification are not allowed] with root cause

java.sql.SQLException: Connection is read-only. Queries leading to data
modification are not allowed
      at
(생략)
kr.co.hanbit.product.management.infrastructure.DatabaseProductRepository.
add(DatabaseProductRepository.java:35) ~[classes/:na]
```

여기에서 DatabaseProductRepository.java의 35번째 줄에 해당하는 코드는 다음 중 볼드로 표시된 부분이다. 해당 코드를 실행하면서 예외가 발생한 것이다.

DatabaseProductRepository.java의 add 메서드

```
public Product add(Product product) {
    KeyHolder keyHolder = new GeneratedKeyHolder();
    SqlParameterSource namedParameter = new BeanPropertySqlParameterSource(product);

    namedParameterJdbcTemplate.update("INSERT INTO products (name, price, amount)
VALUES (:name, :price, :amount)", namedParameter, keyHolder);

    Long generatedId = keyHolder.getKey().longValue();
    product.setId(generatedId);

    return product;
}
```

정상적으로 코드가 실행되도록 만들려면, 데이터의 수정이 필요한 곳에 'readOnly = true' 속성이 적용되지 않도록 변경해야 한다. 따라서 SimpleProductService의 add, update, delete 메서드에는 다음과 같이 'readOnly = false' 속성이 적용되도록 애너테이션을 달아 주자.

SimpleProductService.java

```
(생략)

@Transactional(readOnly = true)
@Service
public class SimpleProductService {

(생략)

    @Transactional(readOnly = false)
    public ProductDto add(ProductDto productDto) {
        Product product = ProductDto.toEntity(productDto);
        validationService.checkValid(product);

        Product savedProduct = productRepository.add(product);
        ProductDto savedProductDto = ProductDto.toDto(savedProduct);
        return savedProductDto;
    }
```

```
(생략)

    @Transactional(readOnly = false)
    public ProductDto update(ProductDto productDto) {
        Product product = ProductDto.toEntity(productDto);
        Product updatedProduct = productRepository.update(product);
        ProductDto updatedProductDto = ProductDto.toDto(updatedProduct);
        return updatedProductDto;
    }

    @Transactional(readOnly = false)
    public void delete(Long id) {
        productRepository.delete(id);
    }
}
```

클래스 위에 붙은 설정보다 메서드 위에 붙은 설정이 우선순위가 높으므로 세 가지 메서드에는 'readOnly = false'가 적용되어 정상적으로 데이터를 수정할 수 있다. 이제 애플리케이션을 재시작하고 상품 데이터를 추가해 보자. 이제 잘 추가될 것이다.

@Transactional 애너테이션은 스프링 프레임워크에서 제공해 주는 추상적 애너테이션이다. JdbcTemplate를 활용하여 데이터베이스를 사용할 때도, JPA를 활용하여 데이터베이스를 사용할 때도 동일한 애너테이션으로 트랜잭셔널한 처리를 지원한다.

로그 추가하기

마지막으로 적절한 곳에 로그를 추가하는 것인데, 여기에서 적절하다는 의미는 상황에 따라 달라질 수 있다. 우선 상품 관리 애플리케이션 내에서는 다음 두 가지 중 어떤 작업이 로그를 남기기에 더 적절할까?

- 상품을 조회하는 작업
- 상품을 추가하거나 수정하는 작업

필자라면 상품을 조회하는 것보다는 상품의 데이터가 수정되는 상품 추가나 상품 수정 쪽에 로그를 남기는 것에 우선순위를 둘 것 같다. 애플리케이션을 개발하고 서비스를 운영하다 보면 특정 데이터가 언제 수정되었는지 추적해야 하는 상황이 자주 발생하기 때문이다. 하지만 상품 조회는 데이터를

변경하지 않기 때문에 데이터가 언제 수정되었는지 추적할 이유가 없으므로 상품을 추가하거나 수정하는 작업에 대한 로그가 더 중요하다.

이런 의문이 들 수도 있다. '고민하지 말고 둘 다 로그를 남기면 되지 않나?' 맞는 말이다. 둘 다 로그를 남기는 것도 괜찮은 방법이다. 그러나 애플리케이션의 규모가 커지면 트래픽도 커지고, 로그에 기록되는 데이터도 그만큼 많아진다. 따라서 불필요한 로그를 남기면 저장 공간이 낭비되고, 문제 해결에 필요한 로그를 찾아내는 것도 어렵게 만든다. 따라서 필요한 로그와 불필요한 로그가 무엇인지 구분하여 필요한 로그를 남기도록 하자.

상황에 따라 적절한 로그가 달라진다고 했다. 상품을 조회하는 작업처럼 무언가를 조회하는 행위가 중요한 상황도 있다. 대표적으로 두 가지 상황을 예로 들 수 있다.

먼저 상품 조회 통계를 내고 이를 분석하기 위해 필요한 상황이 있다. 특정 고객이 커머스 서비스에서 특정 상품을 조회했다는 정보는 굉장히 중요하다. 특정 고객이 그 상품이나 그와 비슷한 상품에 관심이 있다는 것이고, 이것을 타겟팅하여 다른 상품을 해당 고객에게 추천해 줄 수 있기 때문이다.

또 다른 하나는 연봉이나 인사 정보와 같은 민감한 정보를 조회하는 상황이다. 이런 정보는 반드시 조회하는 것 자체를 로그로 남겨야 한다. 만약 로그를 남기지 않는다면 인가되지 않은 사람이 해당 정보를 열람하는 것을 알아챌 수 없거나 알아채는 데 오랜 시간이 걸릴 수 있기 때문이다.

SimpleProductService에 로그를 남기기

SimpleProductService에 로그를 남기는 방법을 간단히 소개한다. 스프링 부트 애플리케이션에서는 기본적으로 로그를 남기기 위해 Log4j라는 라이브러리를 사용한다. 좀 더 편리하게 사용하려면 Lombok이라는 의존성을 추가해 주고, '@Slf4j'라는 애터테이션을 활용하면 된다. Log4j를 직접 사용하는 방법은 간단하다. 다음과 같이 SimpleProductService에 코드를 추가해 주면 된다.

SimpleProductService.java

```
(생략)
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
(생략)

public class SimpleProductService {
```

```
    private final Logger logger = LogManager.getLogger(SimpleProductService.class);

    private ProductRepository productRepository;
    private ValidationService validationService;

(생략)

    @Transactional(readOnly = false)
    public ProductDto add(ProductDto productDto) {
        logger.info("상품을 추가합니다. productDto={}", productDto);

        Product product = ProductDto.toEntity(productDto);
        validationService.checkValid(product);

        Product savedProduct = productRepository.add(product);
        ProductDto savedProductDto = ProductDto.toDto(savedProduct);

        return savedProductDto;
    }

(생략)

    @Transactional(readOnly = false)
    public ProductDto update(ProductDto productDto) {
        logger.info("상품을 수정합니다. productDto={}", productDto);

        Product product = ProductDto.toEntity(productDto);
        Product updatedProduct = productRepository.update(product);
        ProductDto updatedProductDto = ProductDto.toDto(updatedProduct);
        return updatedProductDto;
    }

    @Transactional(readOnly = false)
    public void delete(Long id) {
        logger.info("상품을 삭제합니다. id={}", id);

        productRepository.delete(id);
    }
}
```

로그는 로거Logger를 통해 남길 수 있다. 로거를 사용하기 위해서는 SimpleProductService 클래스의 필드로 LoggerManager를 통해 Logger를 생성하여 주입받아야 한다. 그리고 로그를 남기는 곳에서는 생성한 로거를 사용하여 info라는 메서드로 로그를 남겨 준다. 코드에서 info라는 메서드는 로그의 레벨과 관련된 내용인데, 우선은 로그로 찍히는 메시지에 대해 알아보자.

로그 메시지에는 'productDto={}' 같이 중괄호가 들어가 있는 곳이 있다. 이 중괄호에는 뒤따르는 파라미터가 들어간다. 중괄호가 여러 개라면 뒷쪽에 오는 파라미터도 여러 개라는 뜻이다. 즉, 다음과 같이 중괄호와 파라미터가 순서대로 매핑된다.

```
logger.info("상품을 추가합니다. productDto={}", productDto);

logger.info("a={}, b={}", a, b);
```

중괄호와 파라미터 매핑 예시

애플리케이션을 재시작하고 직접 로그를 찍어 보자. 로그 자체는 잘 찍히지만, 다음과 같이 뭔가 이상하게 찍힌 부분이 보인다.

```
2022-07-17 02:24:32.618  INFO 74428 --- [nio-8080-exec-2] k.c.h.p.m.a.SimpleProdu
ctService          : 상품을 추가합니다. productDto=kr.co.hanbit.product.management.
presentation.ProductDto@439f2fac
```

productDto의 값이 이상하게 찍혀 있다는 것을 알 수 있다. 그 이유는 ProductDto를 문자열로 바꾸는 과정에서 toString이 오버라이딩되지 않았기 때문이다. 이렇게 클래스 이름과 골뱅이(@)에 뒤에 16진수 숫자가 오는 것은 Object 클래스에서 정의된 toString 메서드를 사용하고 있을 때의 특징이다. 따라서 우리는 다음과 같이 ProductDto의 toString을 오버라이딩해 줘야 한다.

ProductDto.java

```
(생략)

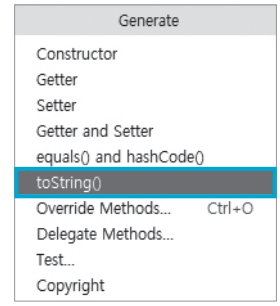
public class ProductDto {
```

```
(생략)

    @Override
    public String toString() {
        return "ProductDto{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", price=" + price +
                ", amount=" + amount +
                '}';
    }

}
```

이 메서드는 인텔리제이에서 제공해 주는 메서드 자동 생성 기능을 활용하여 생성한 것이다. ProductDto 클래스에서 마우스 오른쪽 버튼을 클릭하면 나오는 메뉴에서 [toString()]을 선택하여 간단하게 메서드를 추가할 수 있다.



애플리케이션을 재시작하고 요청해 보면 다음과 같이 로그가 잘 찍히는 것을 볼 수 있다. 상품 추가 요청이므로 id가 들어오지 않았기 때문에 null이 찍힌 것이다.

```
2022-07-17 02:34:19.163  INFO 5736 --- [nio-8080-exec-2] k.c.h.p.m.a.SimpleProductServi
ce          : 상품을 추가합니다. productDto=ProductDto{id=null, name='필통', price=2000,
amount=10}
```

끝으로 Log4j의 **로그 레벨**에는 DEBUG, INFO, WARN, ERROR, FATAL이 있다. 각각의 로그 레벨은 다음과 같은 의미로 사용하는 것을 권장한다.

- **DEBUG:** 애플리케이션을 디버깅할 때 필요한 정보를 기록하는 로그 레벨
- **INFO:** 애플리케이션에서 통상적으로 사용되는 정보를 기록하는 로그 레벨
- **WARN:** 잠재적으로 문제가 될 수 있는 상황에 대한 정보를 경고의 의미로 기록하는 로그 레벨

- **ERROR:** 해당 요청을 처리하는 데 문제가 되는 상황이지만, 애플리케이션 자체는 계속 실행될 수 있는 수준의 정보를 기록하는 로그 레벨
- **FATAL:** 애플리케이션이 중단될 수 있을 정도로 심각한 상황에 대한 정보를 기록하는 로그 레벨

각각의 로그 레벨을 명확하게 구분하는 기준은 없다. 각자가 적절한 로그 레벨을 지정해 주면 된다. 여러분의 판단을 도와줄 만한 예시를 살펴보자.

상품 번호(id)를 통해 상품을 조회하려고 하다가 실패하는 경우 EntityNotFoundException을 발생시켜 줬다. 이 경우 어떤 레벨의 로그를 찍어 줘야 할까? INFO 아니면 ERROR 정도가 적당할까? 어떤 레벨이 적절한지는 상황에 따라 달라질 수 있다. 만약 상품 번호를 사람이 입력하는 값으로 조회한다면 상품 번호는 언제든지 잘못 입력할 수 있다. 그런 경우라면 INFO나 WARN 레벨이 적절하지만, 상품 번호를 잘못 입력할 수 없는 상황, 코드나 서비스 등에 의해 상품이 조회되는 상황이라면 거기서 발생하는 예외는 ERROR 레벨이 적절할 것이다. 시스템상의 심각한 결함이 발생한 것일 수 있기 때문이다.

로그 레벨을 적절하게 적용하는 것이 중요한 이유는 문제가 될 만한 상황을 로그로 감지해내기 위함이다. 예를 들어 ERROR 레벨의 로그가 발생했을 때 개발자에게 즉시 알림을 보내도록 로그 시스템을 구성할 수 있다. 이러면 개발자는 평소에는 로그를 볼 필요가 없고, 알림을 받았을 때만 로그를 살펴보고 시스템에 발생한 문제를 해결할 수 있다. 만약 로그 레벨이 나뉘어 있지 않다면 이런 식으로 시스템을 구성할 수 없고, 매일 발생하는 로그 파일 속에서 시스템의 문제를 찾아 헤매야 한다.

질문 있습니다

예외에 대한 로그는 ERROR 레벨로 남겨야 하나요?

로그 레벨을 나눠서 로그를 남기다 보면, 분명 예외에 대해서 어떤 레벨의 로그를 남겨야 하는지에 대해 고민하는 순간이 오게 된다. 아니면 고민하지 않고 '예외=에러'라는 생각에 예외에 대한 로그를 그냥 ERROR로 남길 수도 있다. 그런데 예외에 대한 로그를 무조건 ERROR로 남기는 것이 적절할까?

당연히 아니다. 책에서 언급한 것처럼, ERROR 레벨 로그는 시스템에 문제가 생겨 개발자가 확인해 봐야 하거나 조치해야 하는 경우에 남기는 것이 좋다. 만약 모든 예외를 ERROR 레벨로 남긴다면 사용자가 잘못 입력하여 예외가 발생한 경우에도 ERROR 레벨로 로그가 남게 되고, 개발자가 직접 확인하고 개입해야 하는 상황만 자동으로 알림받도록 구성하기가 어렵다.

따라서 예외는 에러와 반드시 구분하도록 하자. 본문에서 언급한 것처럼 개발자의 확인 및 개입이 필요한 경우에만 ERROR 레벨의 로그로 남기고, 통상적으로 애플리케이션에서 발생할 수 있고 확인이 불필요한 예외는 INFO 레벨이나 WARN 레벨의 로그를 사용해야 한다.