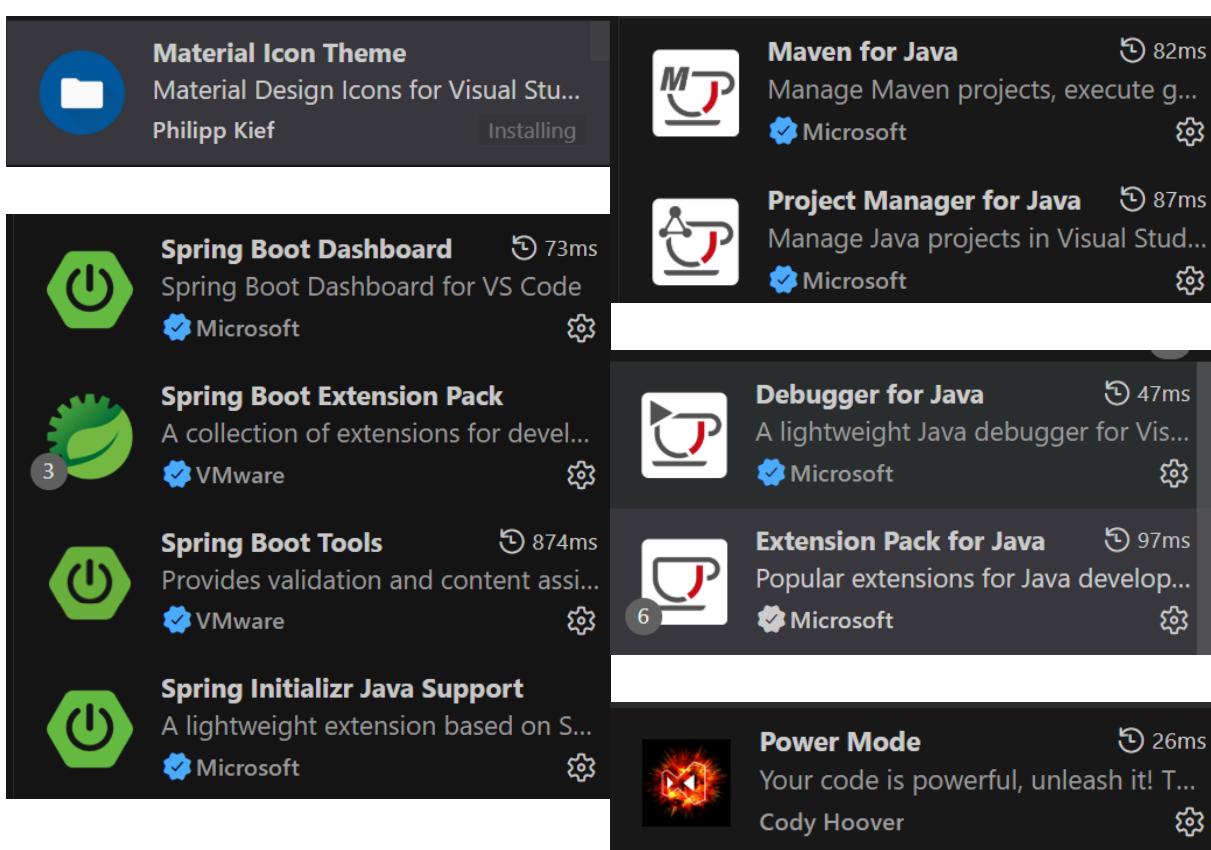


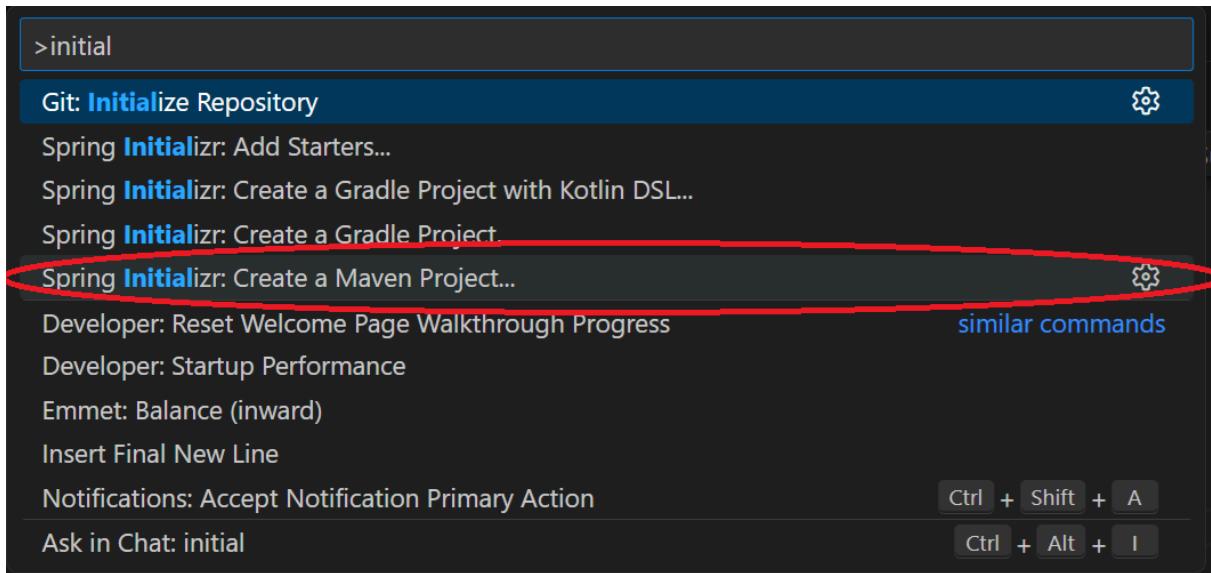
# 유비샘 1차 세미나

2026-02-03

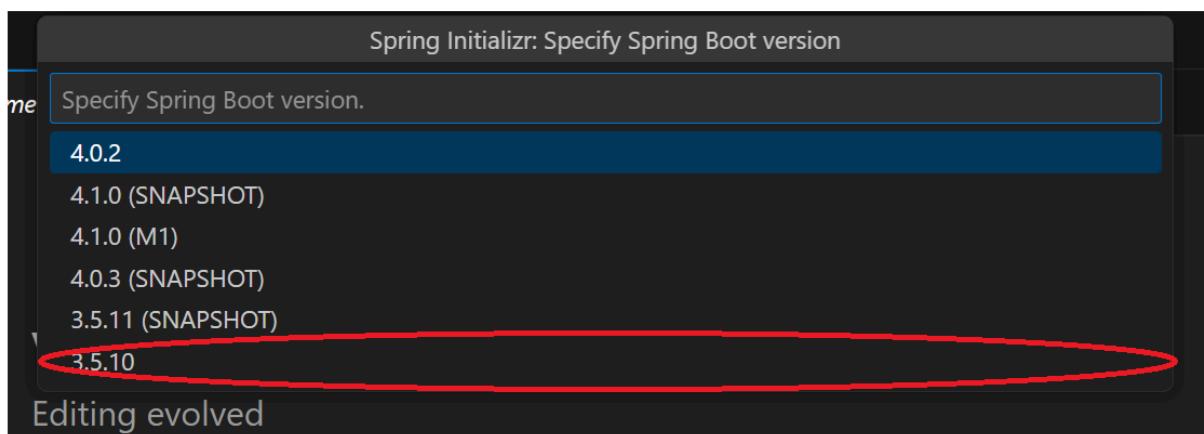
- <https://code.visualstudio.com/docs>
  - 위 docs에는 여러 언어들의 대한 설명이 많으므로 참조하면 좋습니다.
- VS Code 다운 후 확장팩 설치



- Ctrl + Shift + P 를 누른후 initial 을 치면 Create a Maven Project가 나옵니다.



- 최신 버전의 바로 아래 메이저 버전을 권장합니다.



그 후 Java → com.ubisam → example1 → com.ubisam.example1 → Jar  
→ 17 혹은 21을 선택합니다.

- Dependencies 설정은 아래와 같습니다.
  - Rest Repositories - Controller 없이 CRUD Rest API를 자동으로 만들어주는 의존성입니다.
  - Spring Data JPA - Java의 ORM 기술인 JPA(Java Persistence API)를 더 쉽고 편리하게 해줍니다.
  - Lombok - 어노테이션을 기반으로 코드를 자동완성 해주는 라이브러리입니다.
  - HyperSQL Database - 순수 자바로 만들어진 DBMS입니다.

- 해당 의존성 주입 후 서버를 실행했을 시 JDBC 연결 부분입니다.

```
2026-02-04T09:29:05.061+09:00 [INFO] 31869 --- [example1] [main] com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Added connection org.hsqldb.jdbc.JDBCConnection@218f2f51
```

- JDBC란?



Java DataBase Connectivity의 약자로 Java 기반 애플리케이션의 데이터를 데이터베이스에 저장, 업데이트하거나 데이터베이스에 저장된 데이터를 Java에서 사용할 수 있도록 하는 자바 API입니다.

The screenshot shows a list of Java dependencies under the heading 'Selected'. The listed dependencies are:

- ✓ Lombok Developer Tools  
Java annotation library which helps to reduce boilerplate code.
- ✓ HyperSQL Database SQL  
Lightweight 100% Java SQL Database Engine.
- ✓ Spring Data JPA SQL  
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- ✓ Rest Repositories Web  
Exposing Spring Data repositories over REST via Spring Data REST.

- 추가하신 의존성은 아래 구조 기준 pom.xml 파일을 보시면  
<dependencies></dependencies> 안에 내용들로 확인이 가능합니다.

```
<dependencies> Add Spring Boot Starters...
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>

<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>
```

- application.properties 설정 및 설명입니다.

```

spring.application.name=example1

# === JPA / Hibernate: SQL 출력 및 포맷 설정 ===
# 콘솔에 실행되는 SQL을 출력합니다.
spring.jpa.show-sql=true
# 출력되는 SQL을 읽기 쉽게 포맷합니다.
spring.jpa.properties.hibernate.format_sql=true
# Hibernate의 show_sql 속성도 명시적으로 활성화합니다.
spring.jpa.properties.hibernate.show_sql=true

# === DDL 자동 적용 ===
# 개발용: 엔티티 변경 시 스키마를 자동으로 업데이트합니다. (주의: 운영환경에서는 신중히 사용)
spring.jpa.hibernate.ddl-auto=update

# === 로깅 레벨 설정 ===
# 애플리케이션 전체 기본 로그 레벨을 INFO로 설정합니다.
logging.level.root=INFO
# Hibernate가 출력하는 SQL을 INFO 레벨로 기록합니다.
logging.level.org.hibernate.SQL=INFO
# 바인딩(파라미터) 로그를 TRACE로 기록합니다. (자세한 파라미터 값 확인용)
logging.level.org.hibernate.orm.jdbc.bind=TRACE

```

- `spring.jpa.show-sql=true` — 애플리케이션 실행 시 실행되는 SQL을 콘솔에 출력합니다.
  - `spring.jpa.properties.hibernate.format_sql=true` — 출력되는 SQL을 사람이 읽기 쉽게 포맷합니다.
  - `spring.jpa.properties.hibernate.show_sql=true` — Hibernate 레벨에서 SQL 출력 활성화
  - `spring.jpa.hibernate.ddl-auto=update` — 엔티티 변경에 따라 DB 스키마를 자동으로 업데이트합니다.
  - `logging.level.root=INFO` — 전체 애플리케이션의 기본 로그 레벨을 INFO로 설정합니다.
  - `logging.level.org.hibernate.SQL=INFO` — Hibernate가 로그로 출력하는 SQL을 INFO 레벨로 기록합니다.
  - `logging.level.org.hibernate.orm.jdbc.bind=TRACE` — SQL 바인딩(파라미터) 값을 자세히 출력합니다.
- 
- maven 의 구조입니다.

## 루트

- `HELP.md` — 프로젝트별 간단 설명/도움말 입니다.

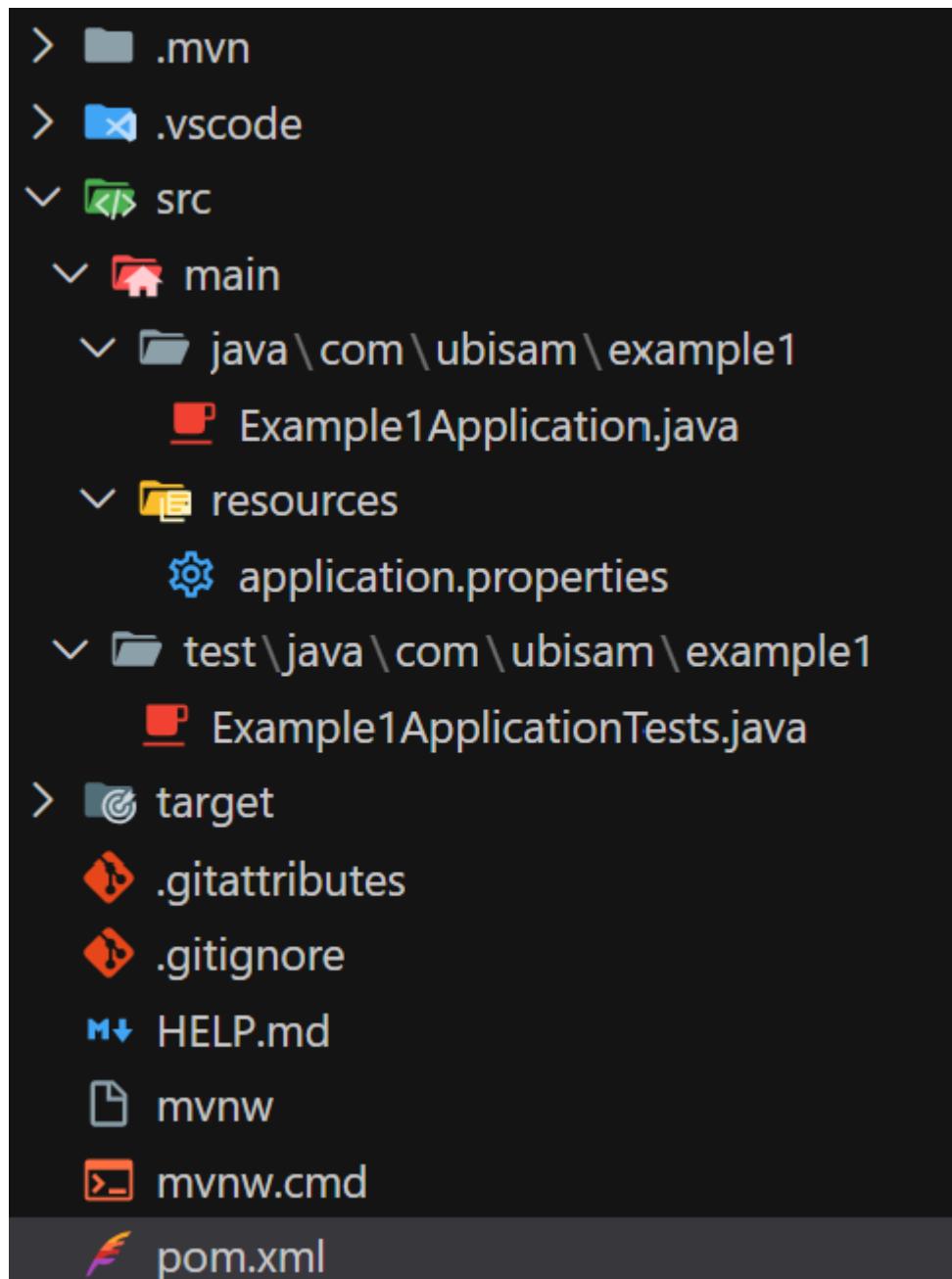
- `mvnw`, `mvnw.cmd` — **Maven Wrapper**(환경에 상관없이 Maven 명령 실행)입니다.

Windows에서는 `mvnw.cmd` 사용합니다.

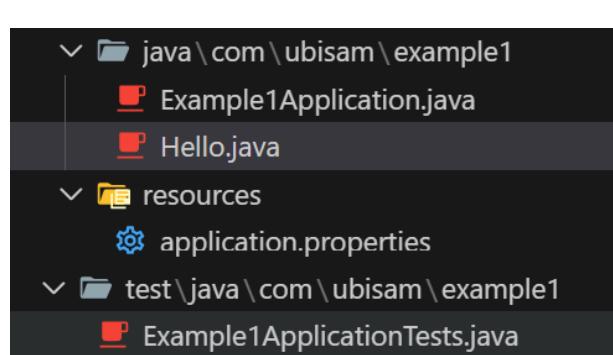
- `pom.xml` — Maven 빌드를 설정(의존성, 플러그인, 빌드 프로필 등)합니다.



- `src/main/java/...`
    - `com/ubisam/example1/Example1Application.java` — 애플리케이션 **메인 클래스**입니다.
  - `src/main/resources/`
    - `application.properties` — 런타임을 설정(포트, DB 연결 등)합니다.
  - `src/test/java/...`
    - `com/ubisam/example1/Example1ApplicationTests.java` — 단위/통합 테스트 클래스입니다.
-



## 실제 코드 테스트 부문입니다.



- 루트에 Hello.java 를 생성합니다.
  - 이때 자바코드는 항상 대문자로 시작합니다.
  - 이는 Java 공식 네이밍 컨벤션입니다.

```
package com.ubisam.example1;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.Id;
import lombok.Data;

@Entity
@Data
public class Hello {

    @Id
    @GeneratedValue
    private Long id;
    private String name;
    private String email;
}
```

- 해당 규칙을 PascalCase(UpperCase)라고 명명합니다.

- @Entity - 해당 클래스가 JPA 엔티티임을 선언합니다. 또한 Hibernate가 이 클래스를 데이터베이스 테이블로 변환합니다.
- @Data - 자동으로 getter, setter, toString, equals, hashCode 메서드를 생성합니다.

(해당 메서드는 아래에서 추가 설명 덧붙이겠습니다)

- @Id - 이 필드가 엔티티의 기본키(PK)임을 지정합니다.
- @GeneratedValue - 기본키 값을 자동으로 생성하도록 지정합니다. 또한 새 레코드 생성 시 id가 자동으로 증가합니다.

만약 id 자동증가를 원하지 않을 경우 빼면 됩니다. 대신 직접 id를 짜줘야합니다.



getter, setter, toString, equals, hashCode 메서드란?

1. getter 메서드 - 필드 값을 읽기 위한 메서드입니다.

private 필드에 접근할 수 있게 public 메소드로 노출하며

형식은 get + 필드명(첫글자 대문자) 입니다.

- 아래 코드는 위 Hello.java 코드 사진을 기준으로 getter를 작성한 예시입니다.

```
public Long getId(){
    return this.id;
}
public String getName(){
    return this.name;
}
```

```
public String getEmail(){
    return this.email;
}
```

2. setter 메서드 - 필드 값을 수정하기 위한 메서드입니다.

필드 값을 변경할 때 사용합니다.

형식은 set + 필드명(첫글자 대문자) 입니다.

- 아래 코드는 위 Hello.java 코드 사진을 기준으로 setter를 작성한 예시입니다.

```
public void setId(Long id){
    this.id = id;
}
public void setName(String name){
    this.name = name;
}
public void setEmail(String email){
    this.email = email;
}
```

3. toString 메서드 - 객체의 정보를 문자열로 변환하는 메서드입니다.

로깅이나 디버깅 시 객체의 상태를 쉽게 확인할 수 있습니다.

- 아래는 해당 메서드를 사용하여 System.out.println(hello); 시 출력되는 예시입니다.

```
@Override
public String toString() {
    return "Hello{" +
        "id=" + id +
        ", name='" + name + '\'' +
        ", email='" + email + '\'' +
        '}';
}
// 출력 예: Hello{id=1, name='name1', email='abc@example.co
m'}
```

4. equals 메서드 - 두 객체가 같은지 비교하는 메서드입니다.

같은 필드 값을 가진 두 객체가 동등한지 판단합니다.

데이터베이스에서 조회한 같은 id의 객체들을 비교할 때 유용합니다.

- 아래는 해당 메서드를 사용한 예시입니다.

```
@Override
public boolean equals(Object o) {
    if (this == o) {
        return true; // 같은 객체 참조면 true
    }
    if (o == null || getClass() != o.getClass()) {
        return false; // null이거나 다른 타입이면 false
    }

    Hello hello = (Hello) o;
    if (id != null ? !id.equals(hello.id) : hello.id != null) {
        return false;
    }
    if (name != null ? !name.equals(hello.name) : hello.name != null) {
        return false;
    }
    return email != null ? email.equals(hello.email) : hello.email == null;
}

// 사용 예:
// Hello h1 = new Hello();
// h1.setId(1L);
// h1.setName("name1");
// Hello h2 = new Hello();
// h2.setId(1L);
// h2.setName("name1");
// h1.equals(h2) → true (필드값이 같음)
```

## 5. hashCode 메서드 - 객체의 해시값을 반환하는 메서드입니다.

HashMap, HashSet 같은 해시 기반 컬렉션에서 객체를 빠르게 찾기 위한 메서드입니다.

`equals`가 `true`인 두 객체는 반드시 같은 `hashCode` 값을 가져야 합니다.

- 아래는 해당 메서드를 사용한 예시입니다.

```
@Override
public int hashCode() {
    // 해시값을 누적해 나가는 변수입니다.
    // 첫 번째 필드(id)의 해시코드를 기반으로 시작합니다.
    // 만약 id가 null일 경우 0으로 시작해 NullPointerException
    // 을 방지합니다.
    // id가 null이 아닐 경우 id.hashCode()를 사용합니다.
    int result = id != null ? id.hashCode() : 0;

    // 지금까지 만든 해시값(result)에 "가중치"를 주고,
    // 다음 필드(name)의 해시코드를 섞어 넣습니다.
    result = 31 * result + (name != null ? name.hashCode()
        : 0);

    // 위와 동일한 방식으로 email까지 섞습니다.
    // 즉, 해시는 (id → name → email) 순서로 누적해서 만들어지고,
    // 각 단계마다 "31배"를 해서 이전 결과와 다음 필드가 잘 섞이도록
    // 합니다.
    result = 31 * result + (email != null ? email.hashCode()
        : 0);

    // 최종적으로 누적된 result를 반환합니다.
    // 이 값이 HashMap/HashSet 같은 해시 기반 컬렉션에서
    // "어느 버킷(bucket)에 들어갈지"를 결정하는 핵심값이 됩니다.
    return result;
}
// 사용 예:
// Set<Hello> set = new HashSet<>();
// set.add(hello1); // hashCode 기반으로 저장됩니다.
// set.contains(hello1); // hashCode로 빠르게 검색합니다.
```

```

@Entity
@Data
@Table(name = "t_str")
public class Hello {

    @Id
    @GeneratedValue
    private Long id;
    private String name;
    private String email;

}

```

- 해당 테이블 명을 설정합니다.
- 테이블 명은 t\_str 입니다.

- ORM은 **Object-Relational Mapping** 의 약자로  
객체와 관계형 데이터베이스의 데이터를 자동으로 매핑해주는 것을 말합니다.
- Java ORM을 JPA(Java Persistent API) 라고 합니다.
- 쉽게 설명드리면 ORM과 JPA는 설계도이며 실제 구현 제품 이름이 Hibernate입니다.



ORM (개념)

  └ JPA (자바 표준 규칙)

    └ Hibernate (JPA 구현체)

```

main
└ java\com\ubisam\example1
  └ HelloRepository.java
resources
└ application.properties
test\java\com\ubisam\example1
└ Example1ApplicationTests.java

```

- Hello.java 와 같은 위치에  
HelloRepository.java를 생성합니다.

```

package com.ubisam.example1;

import org.springframework.data.jpa.repository.JpaRepository;

public interface HelloRepository extends JpaRepository<Hello, Long>{
}

```

- 그 후 HelloRepository 클래스를 interface로 변환합니다.
- extends 를 통해 JpaRepository를 상속받고 제네릭 타입은 Hello, 그리고 Hello 객체의 PK인 id의 타입인 Long을 넣습니다.
- 단, Hello가 @Entity이며 PK필드가 @Id Long id 여야 합니다.

## 테이블이 만들어지는지 확인하기

- 완료 후에는 터미널에 ./mvnw spring-boot:run 혹은 .\mvnw spring-boot:run 을 하시면 실행이 됩니다.

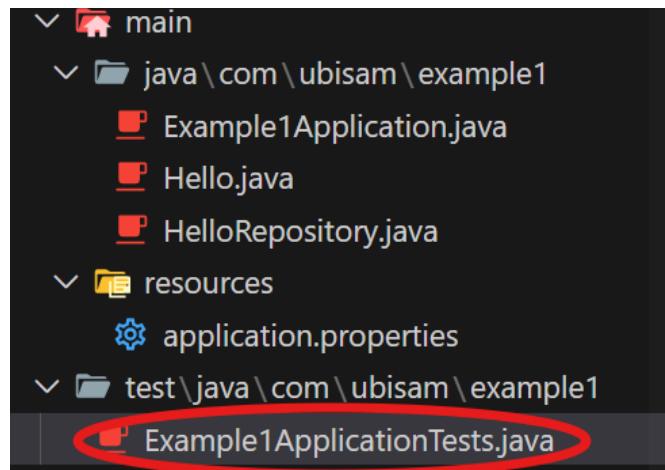
```

dataSource      : HikariPool-1 - Start completed.
2026-02-04T11:57.814+09:00  INFO 16784 --- [example1] [           main] org.hibernate.orm.connections.pooling : HHH0001005: Database info:
                           Database JDBC URL [Connecting through datasource 'HikariDataSource (HikariPool-1)']
                           Database driver: undefined/unknown
                           Database version: 2.7.3
                           Autocommit mode: undefined/unknown
                           Isolation level: undefined/unknown
                           Minimum pool size: undefined/unknown
                           Maximum pool size: undefined/unknown
2026-02-04T11:58.442+09:00  INFO 16784 --- [example1] [           main] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000489: No JTA platform available (set 'hibernate.transaction.jta.platform' to enable JTA platform integration)
Hibernate:
  create table t_str (
    id bigint not null,
    email varchar(255),
    name varchar(255),
    primary key (id)
  )
Hibernate:
  create sequence t_str_seq start with 1 increment by 50
2026-02-04T11:58.481+09:00  INFO 16784 --- [example1] [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2026-02-04T11:58.522+09:00  WARN 16784 --- [example1] [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2026-02-04T11:59.502+09:00  INFO 16784 --- [           main] o.s.b.a.e.AnnotationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.web.embedded.tomcat.TomcatWebServerApplicationContext@700e11f1: startup date [2026-02-04T11:59:50.502+09:00]; root

```

- 그 후 터미널을 올리다보면 t\_str 테이블이 생성된 것을 확인할 수 있습니다.

## 테스트 해보기



- 해당 파일 클릭 후 아래와 같이 코딩을 합니다.

```

package com.ubisam.example1;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class Example1ApplicationTests {

    @Autowired
    private HelloRepository helloRepository;

    @Test
    void contextLoads() {
        Hello h = new Hello();
        h.setName(name: "name1");
        h.setEmail(email: "abc@ubisam.com");

        helloRepository.save(h);
    }
}

```

- Spring 애플리케이션 컨텍스트를 로드하고 JPA 저장소를 사용하는 간단한 통합 테스트입니다.
  - @Autowired - 스프링 컨테이너에 있는 Bean 중에서 타입이 맞는 객체를 자동으로 찾아서 주입해줍니다.
  - CRUD를 수행하려면 실제 인스턴스가 필요하고 Spring이 런타임에 HelloRepository 구현체를 만들어서 빈으로 등록합니다.
- 해당 인스턴스를 가져오기 위해 아래 코드를 사용합니다.

```

@Autowire
private HelloRepository helloRepository;

```

- 해당 테스트 메서드는 Hello 객체를 생성한 후 setName, setEmail을 호출한 뒤 helloRepository.save(h)로 저장하는 작업을 합니다.

- 실행 후 터미널을 볼 때 아래 사진처럼 insert문이 실행되는것을 볼 수 있습니다.

```

LINKSRESOLVER      : Exposing 1 endpoint beneath base path /deacted
2026-02-04T12:02:56.154+09:00  INFO 14428 --- [example1] [           main] c.u.example1.Example1A
pplicationTests    : Started Example1ApplicationTests in 5.53 seconds (process running for 6.841)
Hibernate:
    call next value for t_str_seq
Hibernate:
    insert
    into
        t_str
        (email, name, id)
    values
        (?, ?, ?)
2026-02-04T12:02:56.608+09:00 TRACE 14428 --- [example1] [           main] org.hibernate.orm.jdbc
c.bind              : binding parameter (1:VARCHAR) <- [abc@ubisam.com]
2026-02-04T12:02:56.608+09:00 TRACE 14428 --- [example1] [           main] org.hibernate.orm.jdbc
c.bind              : binding parameter (2:VARCHAR) <- [name1]
2026-02-04T12:02:56.608+09:00 TRACE 14428 --- [example1] [           main] org.hibernate.orm.jdbc
c.bind              : binding parameter (3:BIGINT) <- [1]
-----
BUILD SUCCESS
-----
```

- //Read 밑에 코드를 작성합니다.
  - sysout 치고 ctrl + space하면 System.out.println() 이 자동으로 완성됩니다.

```

@Test
void contextLoads() {
    Hello h = new Hello();
    h.setName(name: "name1");
    h.setEmail(email: "abc@ubisam.com");

    helloRepository.save(h);

    //Read
    Optional<Hello> h2 = helloRepository.findById(1l);
    System.out.println("h.getId(): " + h.getId());
    System.out.println("h2.get().getId(): " + h2.get().getId());
}
```

- 여기서 왜 Optional을 쓰는지 의문이 들어 찾아봤습니다.
- Optional<Hello>를 쓴 이유는 findById를 통해
   
Id가 1인 값을 찾는건데 이때 DB상황은 2가지로 나뉩니다.
  1. ✓ ID=1 데이터 있음
  2. ✗ ID=1 데이터 없음

Optional<Hello>은 조회 결과가 없을 수도 있음을 “강제로” 처리하게 만들기 위함입니다.

만약 Optional을 사용하지 않은 코드라면 아래 코드처럼 사용하게 됩니다.

```
Hello h2 = helloRepository.findById(1L);
```

위 코드를 사용 시 코드 상에서 빨간 줄을 띄우게 됩니다.

- 정상적인 코드로 작동시키면 아래와 같은 결과가 나옵니다.

```
Hibernate:
    select
        h1_0.id,
        h1_0.email,
        h1_0.name
    from
        t_str h1_0
    where
        h1_0.id=?
2026-02-04T12:26:29.097+09:00 TRACE 7948 --- [example1] [main] org.hibernate.orm.jdbc.
bind          : binding parameter (1:BIGINT) <- [1]
h.getId(): 1
h2.getId(): 1
-----
BUILD SUCCESS
```

- h.getId()의 값이랑 h2.getId().getId()의 값이 1로 동일합니다.
- 이때 왜 1로 동일한지 의문이 들어 찾아봤습니다.

```
@Test
void contextLoads() {
    Hello h = new Hello();
    h.setName(name: "name1");
    h.setEmail(email: "abc@ubisam.com");

    helloRepository.save(h);

    //Read
    Optional<Hello> h2 = helloRepository.findById(1L);
    System.out.println("h.getId(): " + h.getId());
    System.out.println("h2.get().getId(): " + h2.get().getId());
}
```

- 아래 코드 실행 시 id = null; 상태입니다.

```
Hello h = new Hello();
```

- setName과 setEmail을 지나 save에 도달하게 되면
  1. ID할당: @GeneratedValue 설정에 따라 `h.id` = 1로 설정합니다.
  2. INSERT 실행: `insert into t_str (...) values (?, ?, ?)` 안에 바인딩 된 파라미터  
해당 코드 기준 각각 (1, name1, abc@ubisam.com)
  3. 해당 id를 다시 h 객체의 세팅하고 h는 영속 상태가 됩니다.
  4. 이 시점부터 `h.getId() == 1`이 됩니다.

```
helloRepository.save(h);
```



JPA는 바로 DB부터 가지 않습니다.

1. 객체 동일성 보장
2. 성능 최적화
3. 변경 감지 & 자동 동기화

를 위해서 라고 합니다.

- 영속성 컨텍스트(1차 캐시) 확인을 합니다.  
이미 ID=1 객체가 있으므로 DB 조회를 생략합니다.  
같은 객체를 반환합니다.
- `h == h2.get()` 의 값이 true로 나오게 됩니다.
- 즉, 아래 코드와 같은 형태로 볼 수 있습니다.

```
h.getId() == h2.get().getId()    // true
h == h2.get()                  // true
```

- 다음은 Update 코드입니다

```

//Read
Optional<Hello> h2 = helloRepository.findById(id: 1L);
// Hello h2 = helloRepository.findById(1L);
System.out.println("h.getId(): " + h.getId());
System.out.println("h2.get().getId(): " + h2.get().getId());

//Update
h.setName(name: "name2");
helloRepository.save(h);

```

- h객체의 name1을 name2로 update 합니다.
- 그 후 save를 호출합니다.
- insert문과 방식이 유사합니다.

```

h.getId(): 1
h2.getId(): 1
Hibernate:
    select
        h1_0.id,
        h1_0.email,
        h1_0.name
    from
        t_str h1_0
    where
        h1_0.id=?
2026-02-04T14:08:43.927+09:00 TRACE 4004 --- [example1] [           main] org.hibernate.orm.jdbc.b
ind           : binding parameter (1:BIGINT) <- [1]
Hibernate:
    update
        t_str
    set
        email=?,
        name=?
    where
        id=?
2026-02-04T14:08:43.932+09:00 TRACE 4004 --- [example1] [           main] org.hibernate.orm.jdbc.b
ind           : binding parameter (1:VARCHAR) <- [abc@ubisam.com]
2026-02-04T14:08:43.932+09:00 TRACE 4004 --- [example1] [           main] org.hibernate.orm.jdbc.b
ind           : binding parameter (2:VARCHAR) <- [name2]
2026-02-04T14:08:43.932+09:00 TRACE 4004 --- [example1] [           main] org.hibernate.orm.jdbc.b
ind           : binding parameter (3:BIGINT) <- [1]
-----
BUILD SUCCESS

```

- 실행 시 name에 바인딩된 파라미터가 name2로 변경된 것을 확인할 수 있습니다.
- 다음은 Delete 코드입니다.

```
//Update  
h.setName(name: "name2");  
helloRepository.save(h);  
  
//Delete  
helloRepository.delete(h);
```

- 이미 엔티티 인스턴스가 존재하므로 그걸 넘겨서 삭제합니다.
- 내부적으로는 해당 엔티티의 ID를 기준으로 DB에서 삭제가 실행됩니다.

```
bind          : binding parameter (1:BIGINT) <- [1]  
Hibernate:  
  delete  
  from  
    t_str  
  where  
    id=?  
2026-02-04T14:14:48.589+09:00 TRACE 19064 --- [example1] [main] org.hibernate.orm.jdbc.  
bind          : binding parameter (1:BIGINT) <- [1]  
-----  
BUILD SUCCESS
```

- 위 결과 사진을 보시면 바인딩 파라미터에 1이 들어가신 것을 확인하실 수 있습니다.
- 내부적으로 ID를 기준으로 삭제했다는 것을 보여줍니다.
- 다음은 Search 코드입니다.

```
//Delete  
helloRepository.delete(h);  
  
//Search  
List<Hello> r = helloRepository.findAll();  
  
System.out.println("r: " + r);
```

- 결과를 확인하기 위해  
System.out.println("r: " + r);  
을 추가했습니다.

```

Hibernate:
    delete
    from
        t_str
    where
        id=?
2026-02-04T14:26:31.274+09:00 TRACE 7464 --- [example1] [main] org.hibernate.orm.jdbc.b
ind           : binding parameter (1:BIGINT) <- [1]
Hibernate:
    select
        h1_0.id,
        h1_0.email,
        h1_0.name
    from
        t_str h1_0
r: []
-----
BUILD SUCCESS
-----
```

- 해당 코드를 실행 시 위와 같은 결과가 출력되는걸 확인하실 수 있습니다.
- select문이 정상적으로 작동하였고 r의 출력은 빈 리스트를 반환합니다.
- 위에서 delete 문을 통해 해당 레코드를 삭제했기 때문에 findAll()이 빈 리스트를 반환 합니다.

```

//Search
List<Hello> r = helloRepository.findAll();
System.out.println("r: " + r);

//Delete
helloRepository.delete(h);
```

- 이런식으로 코드의 위치를 변환하게 되면 정상적으로 결과를 보실 수 있습니다.

```

Hibernate:
    select
        h1_0.id,
        h1_0.email,
        h1_0.name
    from
        t_str h1_0
r: [Hello(id=1, name=name2, email=abc@ubisam.com)]
```

- 다음으로 복합키 설정입니다.

```

@Entity
@Data
@Table(name = "t_str")
public class Hello {

    // @Id
    // @GeneratedValue
    // private Long id;

    @EmbeddedId
    private Id id;
    private String name;
    private String email;

    @Data
    @Embeddable
    public class Id{
        private String id1;
        private String id2;
    }
}

```

- 기본키는 @EmbeddedId로 복합키 Id를 사용합니다.  
(id1, id2 필드로 구성)
- 또한 HelloRepository.java에 제네릭 타입도 변경해야합니다.

```

public interface HelloRepos
itory extends JpaRepository<Hello, Hello.id>{
}

```

Hibernate:

```

create table t_str (
    id1 varchar(255) not null,
    id2 varchar(255) not null,
    email varchar(255),
    name varchar(255),
    primary key (id1, id2)
)

```

- 쿼리는 이렇게 출력됩니다.



SQL 복잡도를 늘리는 방식은 지양해야 합니다.

- 이제 쿼리문을 추가하기 위해 HelloRepository.java로 이동합니다.

```

public interface HelloRepository extends JpaRepository<Hell
o, Long>{
    List<Hello> findByEmail(String email);
    List<Hello> findByNameAndEmail(String name, String emai
l);
    List<Hello> findByIdOrName(Long id, String name);
}

```

- 위와 같이 3개의 코드를 추가합니다.

- ai에게 해당 코드의 출력을 확인할 수 있는 코드를 받아

Example1ApplicationTests.java에 추가 후

테스트한 결과입니다.

- h = helloRepository.save(h); 와 //Read 사이에 넣어서 테스트를 하였습니다.

```

h = helloRepository.save(h);

System.out.println("== findByEmail ==");
helloRepository.findByEmail("abc@ubisam.com")
    .forEach(x -> System.out.println("id=" + x.getId()));

System.out.println("== findByNameAndEmail ==");
helloRepository.findByNameAndEmail("name1", "abc@ubisam.co
m")
    .forEach(x -> System.out.println("id=" + x.getId()));

System.out.println("== findByIdOrName ==");
helloRepository.findByIdOrName(h.getId(), "name1")
    .forEach(x -> System.out.println("id=" + x.getId()));

//Read

```

```

==== findByEmail ===
Hibernate:
    select
        h1_0.id,
        h1_0.email,
        h1_0.name
    from
        t_str h1_0
    where
        h1_0.email=?
2026-02-04T15:06:52.170+09:00 TRACE 32899 --- [example1] [           main] org.hibernate.orm.jdbc.bind : binding parameter (1:VARCHAR) <- [abc@ubisam.com]
id=1

```

```

==== findByNameAndEmail ===
Hibernate:
    select
        h1_0.id,
        h1_0.email,
        h1_0.name
    from
        t_str h1_0
    where
        h1_0.name=? and h1_0.email=?
2026-02-04T15:06:52.182+09:00 TRACE 32899 --- [example1] [           main] org.hibernate.orm.jdbc.bind : binding parameter (1:VARCHAR) <- [name1]
2026-02-04T15:06:52.182+09:00 TRACE 32899 --- [example1] [           main] org.hibernate.orm.jdbc.bind : binding parameter (2:VARCHAR) <- [abc@ubisam.com]
id=1

```

```
==== findByIdOrName ====
Hibernate:
    select
        h1_0.id,
        h1_0.email,
        h1_0.name
    from
        t_str h1_0
    where
        h1_0.id=?
        or h1_0.name=?
2026-02-04T15:06:52.184+09:00 TRACE 32899 --- [example1] [           main] org.hibernate.orm.jdbc.bind          : binding parameter (1:BIGINT) <- [1]
2026-02-04T15:06:52.184+09:00 TRACE 32899 --- [example1] [           main] org.hibernate.orm.jdbc.bind          : binding parameter (2:VARCHAR) <- [name
]
id=1
```

---

## 이렇게 2월 3일 세미나에 대한 내용 및 추가 정보공유 였습니다.

제 소스 파일은 github에 올려놨습니다.

<https://github.com/ubisam-heung/myownstudy/tree/main/Semina/example1>

긴 글 이였는데도 읽어주셔서 감사합니다 😊