

- **JPA 시작하기**

- **개발 환경 설정**

- - 이클립스 설치
 - - 프로젝트 Import 하기

- **H2 데이터베이스 설정**

- - H2 데이터베이스 설치 방법
 - - 예제 테이블 생성하기

- **라이브러리와 프로젝트 구조**

- - 메이븐과 사용 라이브러리 관리

- **객체 매핑하기 시작**

- **persistence.xml 설정**

- - 데이터베이스 방언(hibernate.dialect)

- **애플리케이션 개발**

- - 엔티티 매니저 설정
 - - 트랜잭션 관리
 - - 비즈니스 로직
 - - JPQL(Java Persistence Query Language)
 - - 정리

JPA 시작하기

테이블 하나를 등록/수정/삭제/조회하는 간단한 JPA 애플리케이션을 만들어보고 실제 JPA가 어떻게 동작하는지 알아보자.

개발 환경 설정

예제 프로젝트를 실행하려면 JDK 1.6 버전 이상 설치되어 있어야 한다.

다음 경로에서 예제 프로젝트를 내려받고 압축을 풀어두자.

내려받기 경로: <http://goo.gl/RuuDVI>

- 이클립스 설치

IDE는 이클립스를 사용하겠다. <http://www.eclipse.org> 에 접속해서 이클립스 최신 버전을 내려받고 설치하자. 현재 LUNA가 최신버전이다. 이때 될 수 있으면 Eclipse IDE for Java EE Developers 패키지로 내려받는 것을 권장하는데, 이 패키지를 사용하면 JPA로 개발할 때 편리한 도구들이 지원된다.

내려받기 경로: <http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunasr2>

- 프로젝트 Import 하기

이클립스를 실행한다.

메뉴에서 **File -> Import...** 를 선택한다.

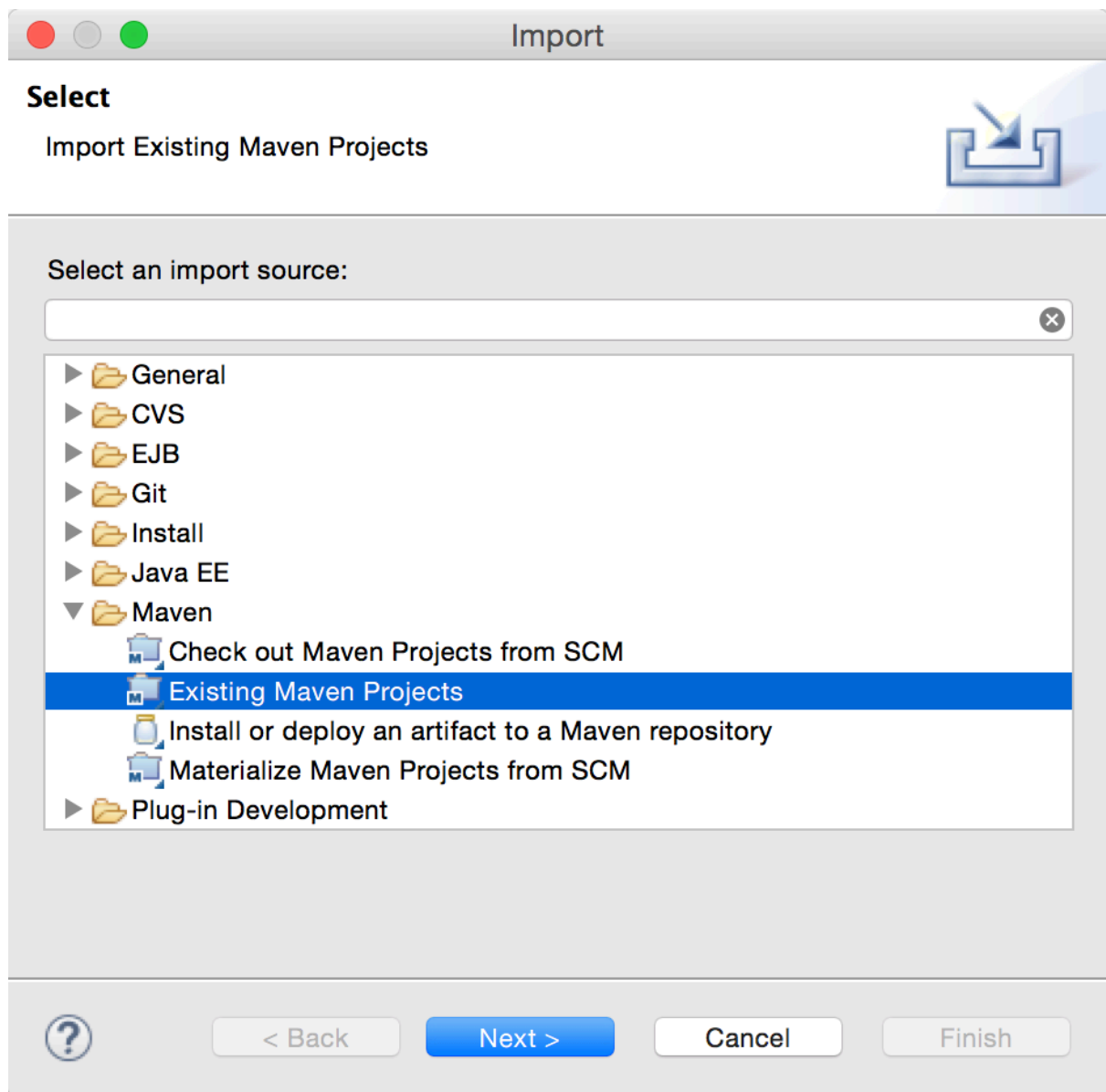


그림 - Import1

Maven -> Existing Maven Projects 를 선택하고 Next 버튼을 누른다.

02. JPA 시작하기

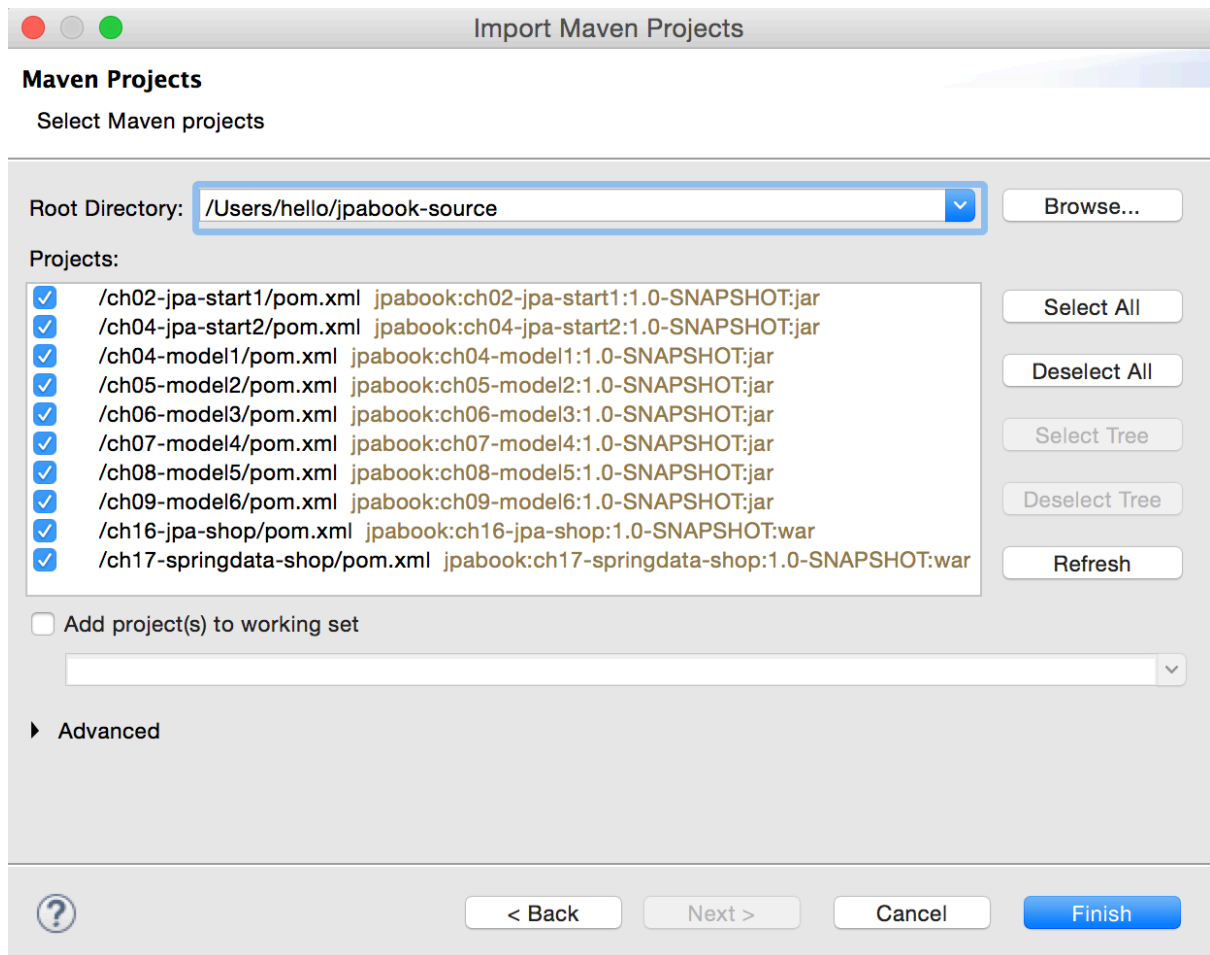


그림 - Import2

Browse... 버튼을 눌러서 예제가 있는 경로를 선택한다. 참고로 경로는 압축을 푼 경로를 선택하면 된다.

경로를 선택하면 모든 프로젝트에 대한 `/pom.xml` 이 나타난다. **Finish** 버튼을 선택하자.

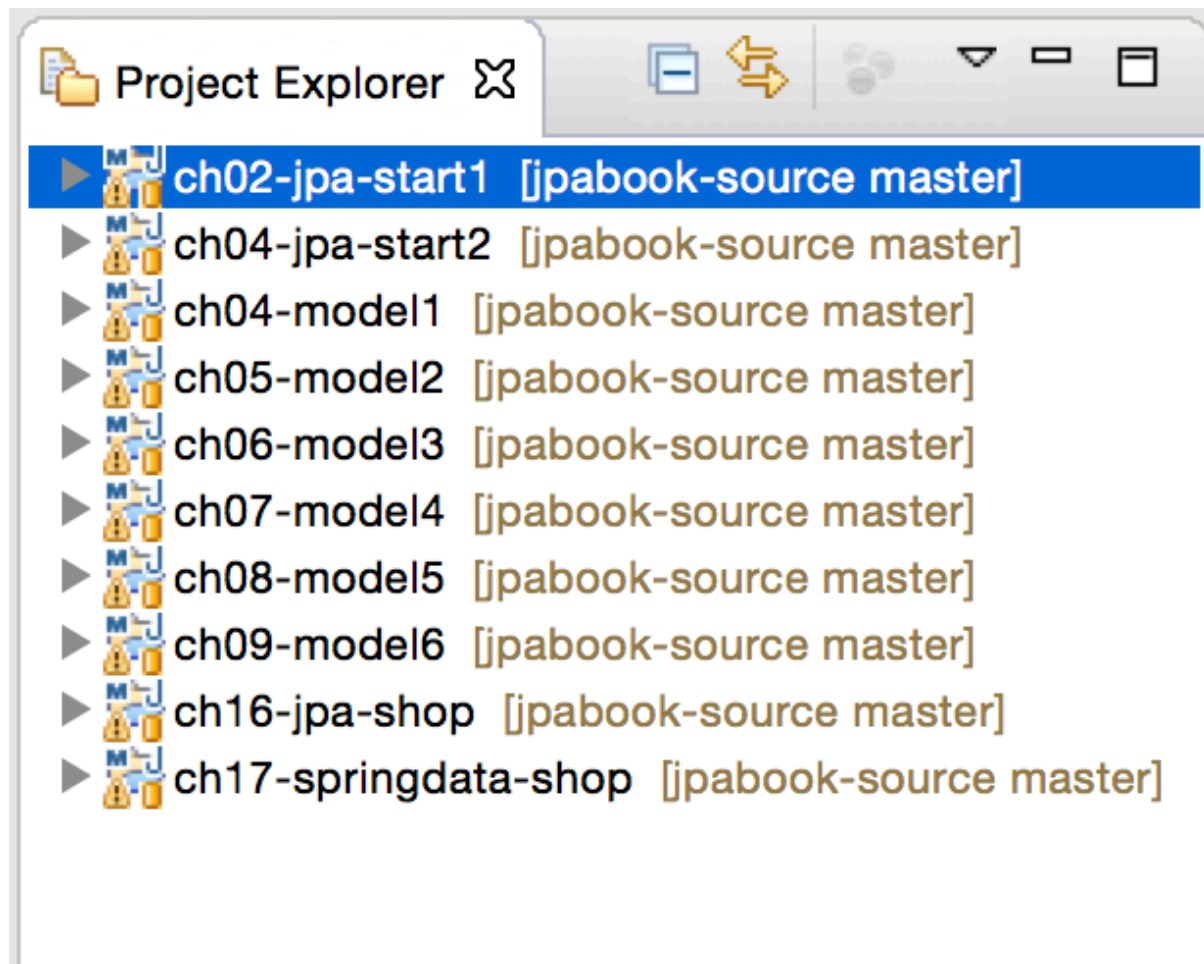


그림 - Import3

모든 예제 프로젝트가 프로젝트가 `Import` 되었다. 참고로 예제 프로젝트를 처음 `Import` 하면 메이븐 저장소에서 라이브러리를 내려받기 때문에 1 ~ 10분 정도 기다려야 한다.

처음 시작할 프로젝트는 `ch02-jpa-start1` 이다.

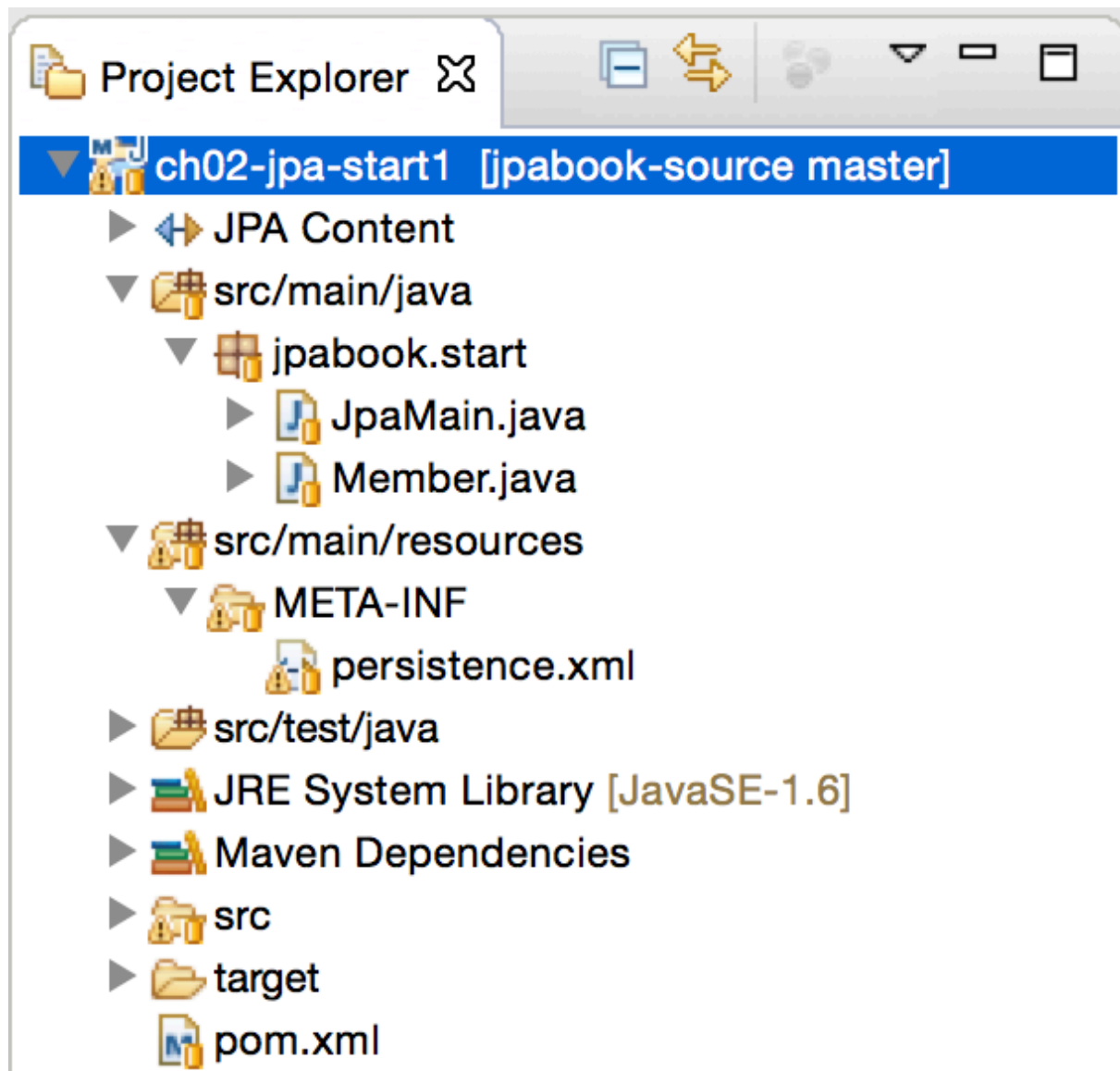


그림 - Import4

참고: 예제를 실행하려면 메이븐(Maven)이 설치되어 있어야 한다. 내려받은 이클립스 LUNA 버전에는 메이븐이 내장되어 있어서 별도로 설치하지 않아도 된다.

메이븐 오류 해결

만약 메이븐 관련 오류가 발생하면 다음 그림 처럼 프로젝트에서 마우스 오른쪽 버튼을 클릭해서 메뉴를 띄운 다음 **Maven -> Update Project...** 를 선택하자.

02. JPA 시작하기

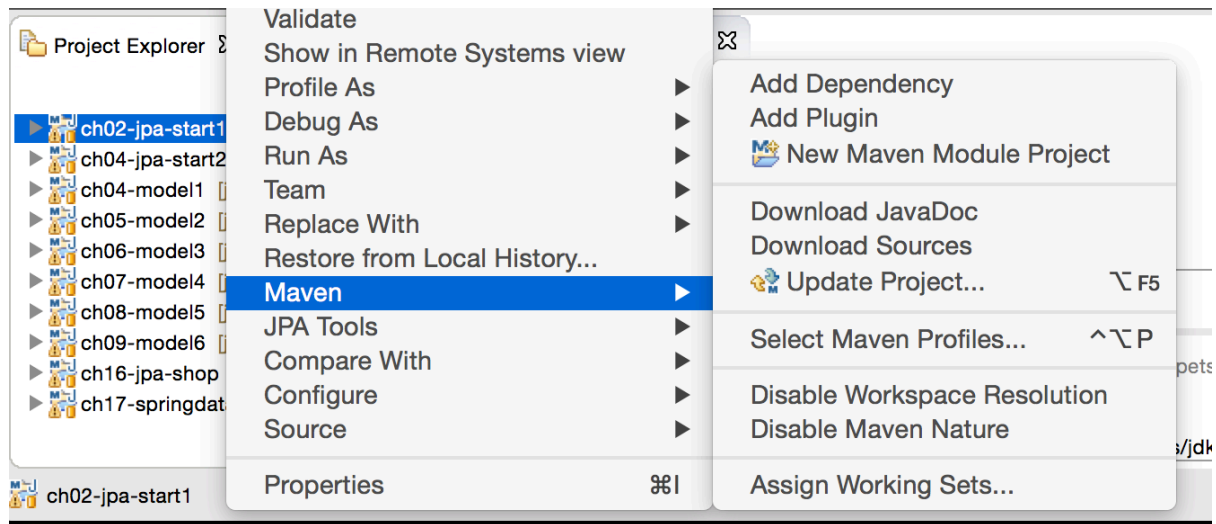


그림 - 메이븐 오류1

그리고 문제가 발생한 프로젝트를 선택한 다음 OK 버튼을 선택하면 메이븐 프로젝트를 초기화하고 구성한다.

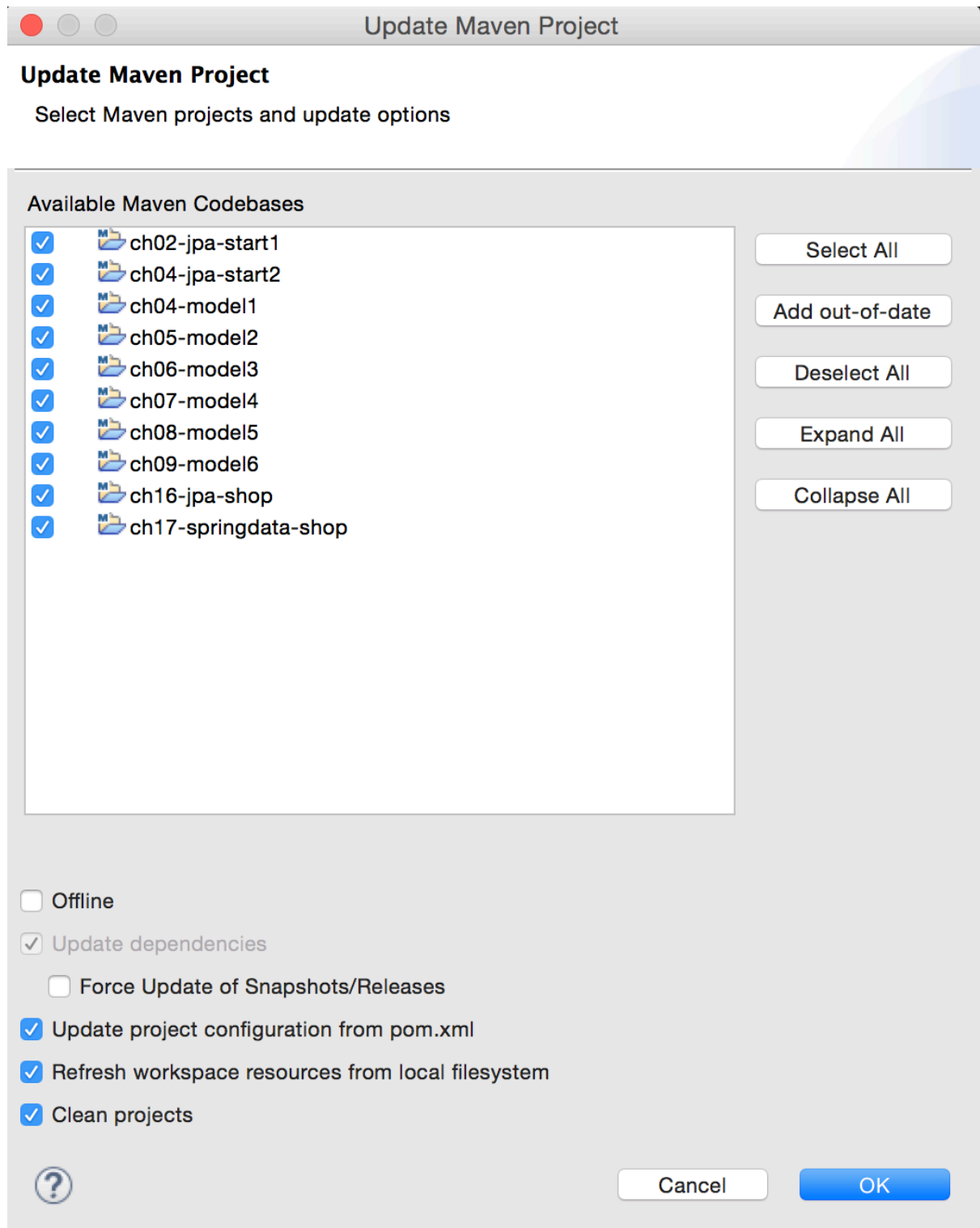


그림 - 메이븐 오류2

H2 데이터베이스 설정

예제는 MySQL이나 Oracle 데이터베이스를 사용해도 되지만 설치하는 부담이 크다. 따라서 설치가 필요없고 용량도 1.7M로 가벼운 H2 데이터베이스를 사용하겠다. 참고로 H2 데이터베이스는 자바가 설치되어 있어야 동작한다.

- H2 데이터베이스 설치 방법


`http://www.h2database.com` 에 들어가서 `All Platforms` 또는 `Platform-Independent zip` 을 내려받아서 압축을 풀자. 참고로 예제에서 사용한 버전은 `1.4.186` 이다.

내려받기 경로: `http://www.h2database.com/h2-2015-03-02.zip`


압축을 푼 곳에서 `bin/h2.sh` 를 실행하면 H2 데이터베이스를 서버 모드로 실행한다. (윈도우는 `h2.bat` 또는 `h2w.bat` 를 실행하면 된다.)

참고: H2 데이터베이스는 JVM 메모리 안에서 실행되는 임베디드 모드와 실제 데이터베이스처럼 별도의 서버를 띄워서 동작하는 서버 모드가 있다.

H2 데이터베이스를 서버 모드로 실행한 후에 웹 브라우저에서 `http://localhost:8082` 을 입력하면 H2 데이터베이스에 접속할 수 있는 화면이 나온다.

한국어  [설정](#) [도구](#) [도움말](#)

로그인

저장한 설정: Generic H2 (Server) 

설정 이름: Generic H2 (Server)

저장 삭제

드라이버 클래스: org.h2.Driver

JDBC URL: jdbc:h2:tcp://localhost/~ /test

사용자명: sa

비밀번호:

연결 연결 시험

그림 - H2 로그인 화면

화면 왼쪽 위를 보면 언어를 선택할 수 있는데 한국어를 선택하자.

02. JPA 시작하기

다음과 같이 입력하고 연결 버튼을 선택하자.

- 드라이버 클래스: `org.h2.Driver`
- JDBC URL: `jdbc:h2:tcp://localhost/~test`
- 사용자명: `sa`
- 비밀번호: 입력하지 않는다.

이렇게 하면 `test` 라는 이름의 데이터베이스에 서버 모드로 접근한다.

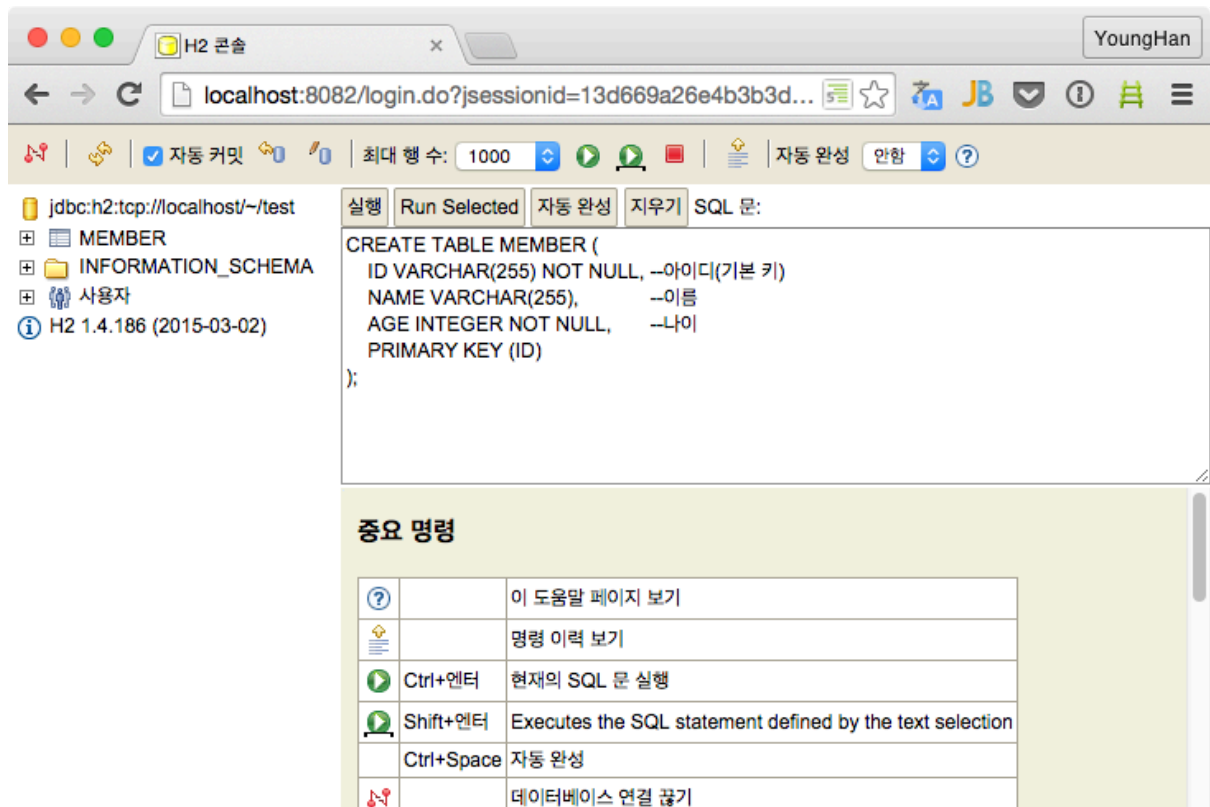


그림 - H2 화면

- 예제 테이블 생성하기

다음 SQL을 입력하고 실행 버튼을 선택하면 그림처럼 왼쪽 메뉴에서 생성된 `MEMBER` 테이블을 볼 수 있다.

```
CREATE TABLE MEMBER (  
  ID VARCHAR(255) NOT NULL, --아이디(기본 키)  
  NAME VARCHAR(255), --이름  
  AGE INTEGER NOT NULL, --나이  
  PRIMARY KEY (ID)  
)
```

라이브러리와 프로젝트 구조

책 전반에 걸쳐 JPA 구현체로 하이버네이트를 사용한다. (하이버네이트 버전 4.3.8.Final)

JPA 구현체로 하이버네이트를 사용하기 위한 핵심 라이브러리는 다음과 같다.

- `hibernate-core` : 하이버네이트 라이브러리
- `hibernate-entitymanager` : 하이버네이트가 JPA 구현체로 동작하도록 JPA 표준을 구현한 라이브러리
- `hibernate-jpa-2.1-api` : JPA 2.1 표준 API를 모아둔 라이브러리

예제에 사용할 프로젝트 구조는 다음과 같다.

```
src/main
├─ java
│   └─ jpabook/start
│       ├── JpaMain.java (실행 클래스)
│       └── Member.java (회원 엔티티)
├─ resources
│   └─ META-INF
│       └─ persistence.xml (JPA 설정 정보)
pom.xml
```

- 메이븐과 사용 라이브러리 관리

라이브러리는 메이븐을 사용해서 관리한다. 메이븐은 간단히 이야기해서 라이브러리를 관리해주는 도구인데 `pom.xml` 에 사용할 라이브러리를 적어주면 라이브러리를 자동으로 내려받아서 관리해준다.

메이븐 설정 파일인 `pom.xml` 을 보자.

===== `pom.xml` =====

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>

    <modelVersion>4.0.0</modelVersion>
    <groupId>jpabook</groupId>
    <artifactId>jpa-start</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
```

```

<!-- JPA, 하이버네이트 -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>4.3.8.Final</version>
</dependency>
<!-- H2 데이터베이스 -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>1.4.186</version>
</dependency>
</dependencies>

</project>

```

`<dependencies>` 에 사용할 라이브러리를 지정한다. `groupId` + `artifactId` + `version` 만 적어 주면 라이브러리(jar 파일)를 메이븐 공식 저장소에서 내려받아 라이브러리에 추가해준다.

프로젝트를 위해 많은 라이브러리가 필요하지만 핵심 라이브러리는 다음 2가지다.

핵심 라이브러리

- JPA, 하이버네이트(`hibernate-entitymanager`) : JPA 표준과 하이버네이트를 포함하는 라이브러리, `hibernate-entitymanager` 를 라이브러리로 지정하면 다음 중요 라이브러리도 함께 내려받는다.
 - `hibernate-core.jar`
 - `hibernate-jpa-2.1-api.jar`
- H2 데이터베이스 : H2 데이터베이스에 접속해야 하므로 `h2` 라이브러리도 지정했다.

메이븐(Maven)^[1]

최근 자바 애플리케이션은 대부분 메이븐을 사용해서 빌드한다. 메이븐은 크게 라이브러리 관리 기능과 빌드 기능을 제공한다. 참고로 최근에는 그래들(Gradle)의 사용이 점점 늘고 있다.

라이브러리 관리 기능

자바 애플리케이션을 개발하려면 jar 파일로 된 여러 라이브러리가 필요하다. 과거에는 이런 라이브러리를 직접 내려받아 사용했다. 메이븐은 사용할 라이브러리 이름과 버전만 명시하면 라이브러리를 자동으로 내려받고 관리해준다.

빌드 기능

애플리케이션을 직접 빌드하는 것은 상당히 고된 작업이다. 과거에는 Ant를 주로 사용했지만 개발자마다 작성하는 Ant 빌드 스크립트는 조금씩 다르다. 메이븐은 애플리케이션을 빌드하는 표준화된 방법을 제공한다.

객체 매핑하기 시작

먼저 다음 SQL을 실행해서 예제에서 사용할 회원 테이블을 만들자. H2 데이터베이스를 설정하면서 이미 생성해두었다면 생략해도 된다.

===== 회원 테이블 =====

```
CREATE TABLE MEMBER (
  ID VARCHAR(255) NOT NULL, --아이디(기본 키)
  NAME VARCHAR(255),        --이름
  AGE INTEGER,               --나이
  PRIMARY KEY (ID)
)
```

다음으로 애플리케이션에서 사용할 회원 클래스를 만들자.

===== 회원 클래스 =====

```
package jpabook.start;

public class Member {

    private String id;           //아이디
    private String username;     //이름
    private Integer age;         //나이

    //Getter, Setter
    public String getId() {return id;}
    public void setId(String id) {this.id = id;}

    public String getUsername() {return username;}
    public void setUsername(String username) {this.username = username;}

    public Integer getAge() {return age;}
    public void setAge(Integer age) {this.age = age;}
}
```

JPA를 사용하려면 가장 먼저 회원 클래스와 회원 테이블을 매핑해야 한다. 다음 매핑 정보 표를 참고하여 둘을 비교하면서 실제 매핑을 시작해보자.

표 - 매핑 정보

매핑 정보	회원 객체	회원 테이블
클래스와 테이블	Member	MEMBER
기본 키	id	ID
필드와 컬럼	username	NAME
필드와 컬럼	age	AGE

회원 클래스에 매핑 어노테이션을 추가하자.

===== 매핑 정보가 포함된 회원 클래스 =====

```
package jpabook.start;

import javax.persistence.*; /**

@Entity
@Table(name="MEMBER")
public class Member {

    @Id
    @Column(name = "ID")
    private String id;

    @Column(name = "NAME")
    private String username;

    //매핑 정보가 없는 필드
    private Integer age;
    ...
}
```

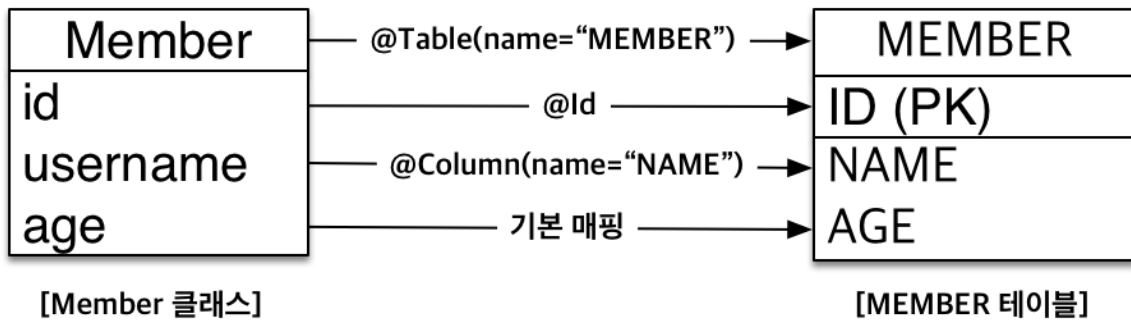


그림 - 클래스와 테이블 매핑

회원 클래스에 매핑 정보를 표시하는 어노테이션을 몇 개 추가했다. 여기서 `@Entity`, `@Table`, `@Column` 이 매핑 정보다. JPA는 매핑 어노테이션을 분석해서 어떤 객체가 어떤 테이블과 관계가 있는지 알아낸다.

회원 클래스에 사용한 매핑 어노테이션을 하나씩 살펴보자.

`@Entity`

이 클래스를 테이블과 매핑한다고 JPA에게 알려준다. 이렇게 `@Entity` 가 사용된 클래스를 **엔티티 클래스**라 한다.

`@Table`

엔티티 클래스에 매핑할 테이블 정보를 알려준다. 여기서는 `name` 속성을 사용해서 `Member` 엔티티를 `MEMBER` 테이블에 매핑했다. 이 어노테이션을 생략하면 클래스 이름을 테이블 이름으로 매핑한다.

`@Id`

엔티티 클래스의 필드를 테이블의 기본 키(Primary key)에 매핑한다. 여기서는 엔티티의 `id` 필드를 테이블의 `ID` 기본 키 컬럼에 매핑했다. 이렇게 `@Id` 가 사용된 필드를 식별자 필드라 한다.

`@Column`

필드를 컬럼에 매핑한다. 여기서는 `name` 속성을 사용해서 `Member` 엔티티의 `username` 필드를 `MEMBER` 테이블의 `NAME` 컬럼에 매핑했다.

매핑정보가 없는 필드

`age` 필드에는 매핑 어노테이션이 없다. 이렇게 매핑 어노테이션을 생략하면 필드명을 사용해서 컬럼명으로 매핑한다. 여기서는 필드명이 `age` 이므로 `age` 컬럼으로 매핑했다. 참고로 이 책에서는 데이터베이스가 대소문자를 구분하지 않는다고 가정한다. 만약 대소문자를 구분하는 데이터베이스를 사용하면 `@Column(name="AGE")` 처럼 명시적으로 매핑해야 한다.

이것으로 매핑 작업을 완료했다. 매핑 정보 덕분에 JPA는 어떤 엔티티를 어떤 테이블에 저장해야 하는지 알 수 있다.

다음으로 JPA를 실행하기 위한 기본 설정 파일인 `persistence.xml` 를 알아보자.

참고 : JPA 어노테이션의 패키지는 `javax.persistence` 이다.

persistence.xml 설정

JPA는 `persistence.xml` 을 사용해서 필요한 설정 정보를 관리한다. 이 설정 파일이 `META-INF/persistence.xml` 클래스 패스 경로에 있으면 별도의 설정 없이 JPA가 인식할 수 있다.

===== `persistence.xml` =====

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">
  <persistence-unit name="jpabook" > /**
    <properties>

      <!-- 필수 속성 -->
      <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
      <property name="javax.persistence.jdbc.user" value="sa"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:h2:tcp://localhost/~/test"/>
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.H2Dialect" />

      <!-- 옵션 -->
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.use_sql_comments" value="true" />
      <property name="hibernate.id.new_generator_mappings" value="true" />

    </properties>
  </persistence-unit>
</persistence>
```

`persistence.xml` 의 내용을 차근차근 분석해보자.

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">
```


설정 파일은 `persistence` 로 시작한다. 이곳에 XML 네임스페이스와 사용할 버전을 지정한다. JPA 2.1을 사용하려면 이 `xmlns` 와 `version` 을 명시하면 된다.

```
<persistence-unit name="jpabook" >
```

JPA 설정은 영속성 유닛(`persistence-unit`)이라는 것부터 시작하는데 일반적으로 연결할 데이터베이스당 하나의 영속성 유닛을 등록한다. 그리고 영속성 유닛에는 고유한 이름을 부여해야 하는데 여기서는 `jpabook` 이라는 이름을 사용했다.

다음으로 설정한 각각의 속성 값을 분석해보자.

```
<properties>
  <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
  ...
```

사용한 속성은 다음과 같다.

JPA 표준 속성

- `javax.persistence.jdbc.driver` : JDBC 드라이버
- `javax.persistence.jdbc.user` : 데이터베이스 접속 아이디
- `javax.persistence.jdbc.password` : 데이터베이스 접속 비밀번호
- `javax.persistence.jdbc.url` : 데이터베이스 접속 URL

하이버네이트 속성

- `hibernate.dialect` : 데이터베이스 방언(Dialect) 설정

이름이 `javax.persistence` 로 시작하는 속성은 JPA 표준 속성으로 특정 구현체에 종속되지 않는다. 반면에 `hibernate` 로 시작하는 속성은 하이버네이트 전용 속성이므로 하이버네이트에서만 사용할 수 있다.

사용한 속성을 보면 데이터베이스에 연결하기 위한 설정이 대부분이다. 여기서 가장 중요한 속성은 데이터베이스 방언을 설정하는 `hibernate.dialect` 다.

- 데이터베이스 방언(`hibernate.dialect`)

JPA는 특정 데이터베이스에 종속적이지 않은 기술이다. 따라서 다른 데이터베이스로 손쉽게 교체할 수 있는 장점이 있다. 그런데 각각의 데이터베이스가 제공하는 SQL 문법과 함수가 조금씩 다르다는 문제점이 있다.

데이터베이스마다 다른 점 예시

- 데이터 타입: 가변 문자 타입으로 MySQL은 `VARCHAR`, Oracle은 `VARCHAR2` 를 사용한다.
- 다른 함수명: 문자열을 자르는 함수로 SQL 표준은 `SUBSTRING()` 를 사용하지만 Oracle은 `SUBSTR()` 을 사용한다.
- 페이징 처리: MySQL은 `LIMIT` 를 사용하지만 Oracle은 `ROWNUM` 을 사용한다.

이처럼 SQL 표준을 지키지 않거나 특정 데이터베이스만의 고유한 기능을 JPA에서는 방언(Dialect)이라 한다. 애플리케이션 개발자가 특정 데이터베이스에 종속되는 기능을 많이 사용하면 나중에 데이터베이스를 교체하기가 어렵다. 하이버네이트를 포함한 대부분의 JPA 구현체들은 이런 문제를 해결하려고 다양한 데이터베이스 방언 클래스를 제공한다.

개발자는 JPA가 제공하는 표준 문법에 맞추어 JPA를 사용하면 되고, 특정 데이터베이스에 의존적인 SQL은 데이터베이스 방언이 처리해준다. 따라서 데이터베이스가 변경되어도 애플리케이션 코드를 변경할 필요 없이 데이터베이스 방언만 변경하면 된다. 참고로 데이터베이스 방언을 설정하는 방법은 JPA에 표준화되어 있지 않다.

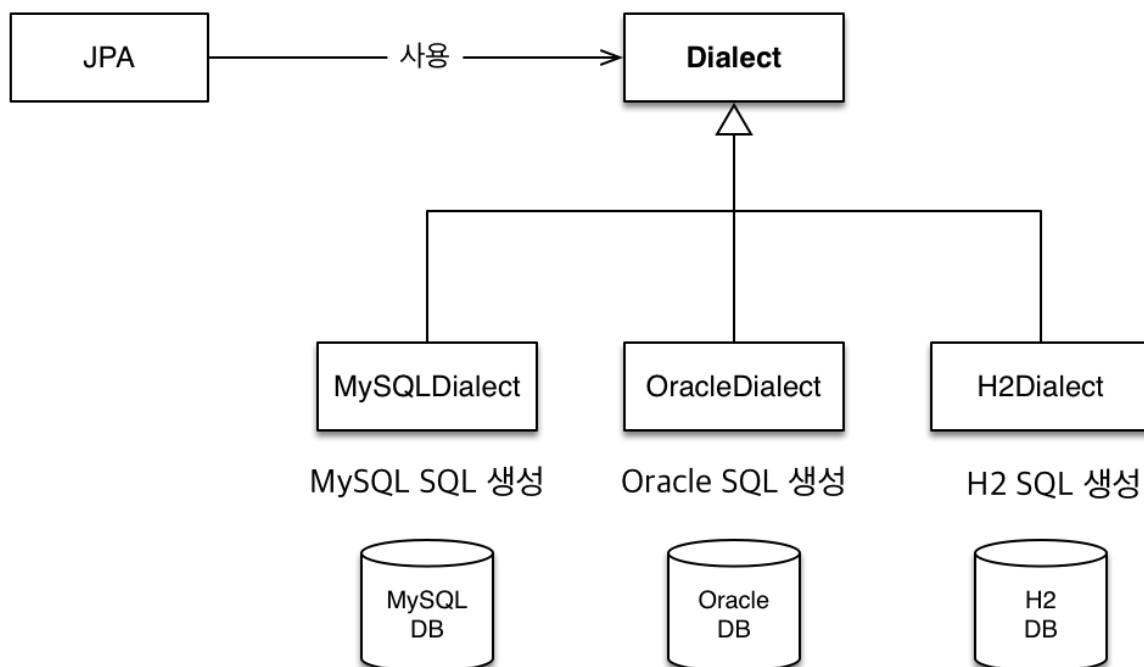


그림 - 방언

하이버네이트는 다양한 데이터베이스 방언을 제공한다. 대표적인 몇 개만 보자.

02. JPA 시작하기

- H2 : `org.hibernate.dialect.H2Dialect`
- Oracle 10g : `org.hibernate.dialect.Oracle10gDialect`
- MySQL : `org.hibernate.dialect.MySQL5InnoDBDialect`

여기서는 H2를 사용하므로 `org.hibernate.dialect.H2Dialect` 로 설정했다.

참고: 하이버네이트는 현재 45개의 데이터베이스 방언을 지원한다. 상세 항목은 다음 URL 을 참고하자. http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single/#configuration-optional-dialects

지금까지 `persistence.xml` 설정을 살펴보았다. 이제 이 정보를 바탕으로 실제 JPA를 사용해보자.

참고: 사용된 하이버네이트 전용 속성은 다음과 같다.

- `hibernate.show_sql` : 하이버네이트가 실행한 SQL을 출력한다.
- `hibernate.format_sql` : 하이버네이트가 실행한 SQL을 출력할 때 보기 쉽게 정렬한다.
- `hibernate.use_sql_comments` : 쿼리를 출력할 때 주석도 함께 출력한다.
- `hibernate.id.new_generator_mappings` : JPA 표준에 맞춘 새로운 키 생성 전략을 사용한다. 자세한 내용은 다음 장의 키 생성에서 설명하겠다.

애플리케이션 개발

애플리케이션 전체 코드는 다음과 같다.

===== 실행 코드 =====

```
package jpabook.start;

import javax.persistence.*;
import java.util.List;

public class JpaMain {

    public static void main(String[] args) {

        //[엔티티 매니저 팩토리] - 생성
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpabc
```

```
//[엔티티 매니저] - 생성
EntityManager em = emf.createEntityManager();
//[트랜잭션] - 획득
EntityTransaction tx = em.getTransaction();

try {

    tx.begin();      //[트랜잭션] - 시작
    logic(em);       //[비즈니스 로직 실행]
    tx.commit();     //[트랜잭션] - 커밋

} catch (Exception e) {
    tx.rollback();   //[트랜잭션] - 롤백
} finally {
    em.close();     //[엔티티 매니저] - 종료
}
emf.close();       //[엔티티 매니저 팩토리] - 종료

//[비즈니스 로직]
private static void logic(EntityManager em) {...}
}
```

코드는 크게 3부분으로 나뉘어 있다.

- 엔티티 매니저 설정
- 트랜잭션 관리
- 비즈니스 로직

엔티티 매니저 설정부터 살펴보자.

- 엔티티 매니저 설정

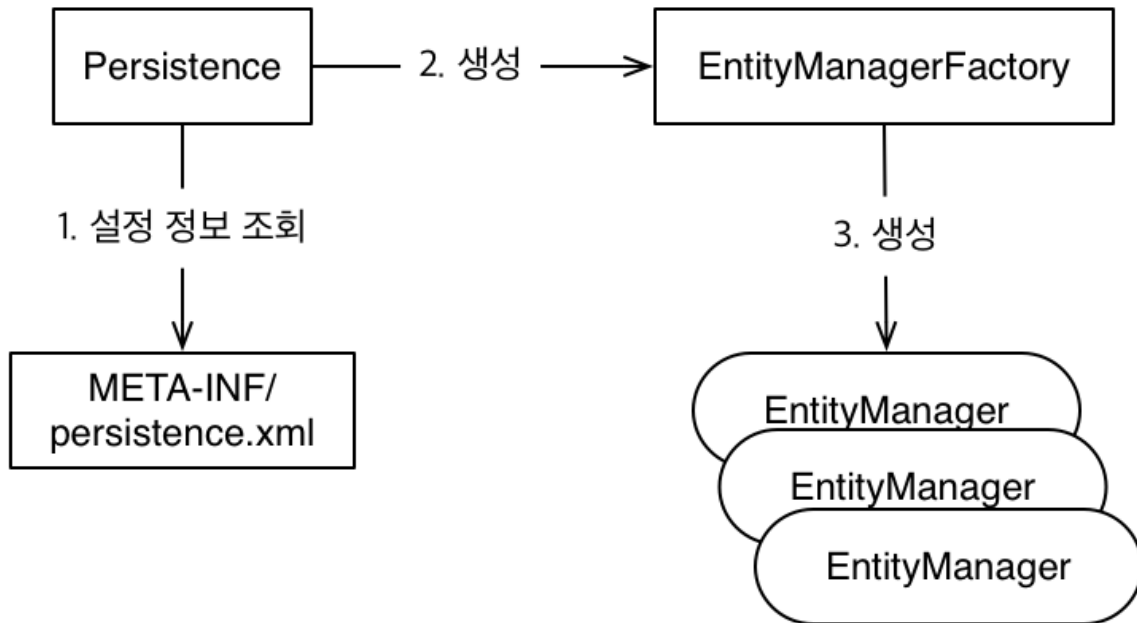


그림 - 엔티티 매니저 생성 과정

엔티티 매니저 팩토리 생성

JPA를 시작하려면 우선 `persistence.xml` 의 설정 정보를 사용해서 엔티티 매니저 팩토리를 생성해야 한다. 이때 `Persistence` 클래스를 사용하는데 이 클래스는 엔티티 매니저 팩토리를 생성해서 JPA를 사용할 수 있게 준비한다.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpabook");
```

이렇게 하면 `META-INF/persistence.xml` 에서 이름이 `jpabook` 인 영속성 유닛(`persistence-unit`)을 찾아서 엔티티 매니저 팩토리를 생성한다. 이때 `persistence.xml` 의 설정 정보를 읽어서 JPA를 동작시키기 위한 기반 객체를 만들고 JPA 구현체에 따라서는 데이터베이스 커넥션 풀도 생성하므로 엔티티 매니저 팩토리를 생성하는 비용은 아주 크다. 따라서 **엔티티 매니저 팩토리는 애플리케이션 전체에서 딱 한 번만 생성하고 공유해서 사용해야 한다.**

엔티티 매니저 팩토리를 만들었으니 이제 엔티티 매니저를 생성하자.

엔티티 매니저 생성

```
EntityManager em = emf.createEntityManager();
```

02. JPA 시작하기

엔티티 매니저 팩토리에서 엔티티 매니저를 생성한다. JPA의 기능 대부분은 이 엔티티 매니저가 제공한다. 대표적으로 **엔티티 매니저를 사용해서 엔티티를 데이터베이스에 등록/수정/삭제/조회할 수 있다**. 엔티티 매니저는 내부에 데이터소스(데이터베이스 커넥션)를 유지하면서 데이터베이스와 통신한다. 따라서 애플리케이션 개발자는 엔티티 매니저를 가상의 데이터베이스로 생각할 수 있다.

참고로 엔티티 매니저는 데이터베이스 커넥션과 밀접한 관계가 있으므로 스레드간에 공유하거나 재사용하면 안 된다.

사용이 끝난 엔티티 매니저는 반드시 종료해야 한다.

```
em.close(); //엔티티 매니저 종료
```

애플리케이션을 종료할 때 엔티티 매니저 팩토리도 종료해야 한다.

```
emf.close(); //엔티티 매니저 팩토리 종료
```

- 트랜잭션 관리

JPA를 사용하면 항상 트랜잭션 안에서 데이터를 변경해야 한다. 트랜잭션 없이 데이터를 변경하면 예외가 발생한다. 트랜잭션을 시작하려면 엔티티 매니저(`em`)에서 트랜잭션 API를 받아와야 한다.

```
EntityTransaction tx = em.getTransaction(); //트랜잭션 API
try {

    tx.begin(); //트랜잭션 시작
    logic(em); //비즈니스 로직 실행
    tx.commit(); //트랜잭션 커밋

} catch (Exception e) {
    tx.rollback(); //예외 발생시 트랜잭션 롤백
}
```

트랜잭션 API를 사용해서 비즈니스 로직이 정상 동작하면 트랜잭션을 커밋(commit)하고 예외가 발생하면 트랜잭션을 롤백(rollback)한다.

- 비즈니스 로직

02. JPA 시작하기

비즈니스 로직은 단순하다. 회원 엔티티를 하나 생성한 다음 엔티티 매니저를 통해 데이터베이스에 등록, 수정, 삭제, 조회한다.

```
public static void logic(EntityManager em) {

    String id = "id1";
    Member member = new Member();
    member.setId(id);
    member.setUsername("지한");
    member.setAge(2);

    //등록
    em.persist(member);

    //수정
    member.setAge(20);

    //한 건 조회
    Member findMember = em.find(Member.class, id);
    System.out.println("findMember=" + findMember.getUsername() +
        ", age=" + findMember.getAge());

    //목록 조회
    List<Member> members =
        em.createQuery("select m from Member m", Member.class)
            .getResultList();
    System.out.println("members.size=" + members.size());

    //삭제
    em.remove(member);
}
```

출력 결과

```
findMember=지한, age=20
members.size=1
```

비즈니스 로직을 보면 등록, 수정, 삭제, 조회 작업이 엔티티 매니저(em)를 통해서 수행되는 것을 알 수 있다. 엔티티 매니저는 객체를 저장하는 가상의 데이터베이스처럼 보인다. 코드를 분석해보자.

등록

```
String id = "id1";
```

02. JPA 시작하기

```
Member member = new Member();
member.setId(id);
member.setUsername("지한");
member.setAge(2);

//등록
em.persist(member);
```

엔티티를 저장하려면 엔티티 매니저의 `persist()` 메서드에 저장할 엔티티를 넘겨주면 된다. 예제를 보면 회원 엔티티를 생성하고 `em.persist(member)` 를 실행해서 엔티티를 저장했다. JPA는 회원 엔티티의 매핑 정보(어노테이션)를 분석해서 다음과 같은 SQL을 만들어 데이터베이스에 전달한다.

==== 실행된 SQL ====

```
INSERT INTO MEMBER (ID, NAME, AGE) VALUES ('id1', '지한', 2)
```

수정

```
//수정
member.setAge(20);
```

수정 부분을 보면 조금 이상하다. 엔티티를 수정한 후에 수정 내용을 반영하려면 `em.update()` 같은 메서드를 호출해야 할 것 같은데 단순히 엔티티의 값만 변경했다. JPA는 어떤 엔티티가 변경되었는지 추적하는 기능을 갖추고 있다. 따라서 `member.setAge(20)` 처럼 엔티티의 값만 변경하면 다음과 같은 UPDATE SQL을 생성해서 데이터베이스에 값을 변경한다. 사실 `em.update()` 라는 메서드도 없다.

==== 실행된 SQL ====

```
UPDATE MEMBER
  SET AGE=20, NAME='지한'
 WHERE ID='id1'
```

삭제

```
em.remove(member);
```


02. JPA 시작하기

엔티티를 삭제하려면 엔티티 매니저의 `remove()` 메서드에 삭제하려는 엔티티를 넘겨준다. JPA는 DELETE SQL을 생성해서 실행한다.

==== 실행된 SQL ====

```
DELETE FROM MEMBER WHERE ID = 'idl'
```

한 건 조회

```
//한 건 조회
Member findMember = em.find(Member.class, id);
```

`find()` 메서드는 조회할 엔티티 타입과 `@Id` 로 데이터베이스 테이블의 기본 키와 매핑한 식별자 값으로 엔티티 하나를 조회하는 가장 단순한 조회 메서드다. 이 메서드를 호출하면 SELECT SQL을 생성해서 데이터베이스에 결과를 조회한다. 그리고 조회한 결과 값으로 엔티티를 생성해서 반환한다.

==== 실행된 SQL ====

```
SELECT * FROM MEMBER WHERE ID='idl'
```

목록 조회

```
//목록 조회
TypedQuery<Member> query =
    em.createQuery("select m from Member m", Member.class);
List<Member> members = query.getResultList();
```

JPA를 사용하면 애플리케이션 개발자는 엔티티 객체를 중심으로 개발하고 데이터베이스에 대한 처리는 JPA에 맡겨야 한다. 바로 앞에서 살펴본 등록, 수정, 삭제, 한 건 조회 예를 보면 SQL을 전혀 사용하지 않았다. 문제는 검색 쿼리다. JPA는 엔티티 객체를 중심으로 개발하므로 검색을 할 때도 테이블이 아닌 엔티티 객체를 대상으로 검색해야 한다.

그런데 테이블이 아닌 엔티티 객체를 대상으로 검색하려면 데이터베이스의 모든 데이터를 애플리케이션으로 불러와서 엔티티 객체로 변경한 다음 검색해야 하는데, 이는 사실상 불가능하다.

애플리케이션이 필요한 데이터만 데이터베이스에서 불러오려면 결국 검색 조건이 포함된 SQL을 사용해야 한다. JPA는 JPQL이라는 쿼리 언어로 이런 문제를 해결한다.

- JPQL(Java Persistence Query Language)

JPA는 SQL을 추상화한 JPQL이라는 객체 지향 쿼리 언어를 제공한다. JPQL은 SQL과 문법이 거의 유사해서 SELECT, FROM, WHERE, GROUP BY, HAVING, JOIN 등을 사용할 수 있다. 둘의 가장 큰 차이점은 다음과 같다.

- JPQL은 **엔티티 객체**를 대상으로 쿼리한다. 쉽게 이야기해서 클래스와 필드를 대상으로 쿼리한다.
- SQL은 **데이터베이스 테이블**을 대상으로 쿼리한다.

예제에서 `select m from Member m` 이 바로 JPQL 이다. 여기서 `from Member` 는 회원 엔티티 객체를 말하는 것이지 `MEMBER` 테이블이 아니다. **JPQL은 데이터베이스 테이블에 대해 전혀 알지 못한다.**

JPQL을 사용하려면 먼저 `em.createQuery(JPQL, 반환 타입)` 메서드를 실행해서 쿼리 객체를 생성한 후 쿼리 객체의 `getResultList()` 메서드를 호출하면 된다. JPA는 JPQL을 분석해서 적절한 SQL을 만들어 데이터베이스에서 데이터를 조회한다.

==== JPA가 실행한 SQL ====

```
SELECT M.ID, M.NAME, M.AGE FROM MEMBER M
```

자세한 내용은 10장 JPQL에서 알아보자.

참고: JPQL은 대소문자를 명확하게 구분하지만, SQL은 관례상 대소문자를 구분하지 않고 사용하는 경우가 많다. 책에서는 JPQL과 SQL을 구분하려고 SQL은 될 수 있으면 대문자로 표현하겠다. 그리고 실제 JPA가 실행한 SQL에는 대소문자가 섞여 있고, 별칭도 알아보기 어려우므로 대문자로 고치고, 별칭도 읽기 쉽게 고치겠다.

- 정리

JPA가 반복적인 JDBC API와 결과 값 매핑을 처리해준 덕분에 코드량이 상당히 많이 줄어든 것은 물론이고 심지어 SQL도 작성할 필요가 없었다. 하지만 코드량을 줄이고 SQL을 자동 생성하는 것은 JPA가 제공하는 전체 기능 중 일부에 불과하다.

02. JPA 시작하기

1. <http://maven.apache.org> ↩