Spring Boot 3 - Database Access with Hibernate / JPA CRUD

Hibernate / JPA Overview

- What is **Hibernate?**
  - A framework for **persisting/saving Java objects** in a **database**
- Benefits of **Hibernate**
  - **Hibernate handles** all of the **low-level SQL**
  - **Minimizes** the amount of **JDBC code** you have to develop
  - **Hibernate** provides the **Object-to-Relational Mapping (ORM)**
- **Object-to-Relational Mapping (ORM)**
  - The developer defines **mapping** between **Java classes** and **database tables**
- What is **JPA**?
  - **Jakarta Persistence API (JPA)**
    - Previously known as **Java Persistence API**
    - Standard API for **Object-to-Relational Mapping (ORM)**
  - Merely a **specification**
    - Defines a **set of interfaces**
    - **Requires** an **implementation** to be usable
- Benefits of **JPA**
  - By having a **standard API**, you're **not locked** to a **vendor's implementation**
  - Maintain **portable**, **flexible code** by coding to **JPA spec (interfaces)**
  - Can **theoretically switch vendor implementations**
    - For example, you can switch to Vendor XYZ without vendor lock-in if Vendor ABC stops supporting their product
- **JPA/Hibernate CRUD Apps**
  - **C**reate objects
  - **R**ead objects

    - **U**pdate objects
    - **D**elete objects

Hibernate, JPA, and JDBC

- **Hibernate / JPA** uses **JDBC** for all **database communications**

Setting Up Development Environment

- **MySQL Database**
    - Two **components**
        - **MySQL Database Server**
        - **MySQL Workbench**
- **MySQL Database Server**
    - The **main engine** of the database
    - **Stores data** for the database
    - Supported **CRUD** features on the data
- **MySQL Workbench**
    - A **client GUI** for **interacting** with the **database**
    - Create **database schemas** and **tables**
    - **Execute SQL queries** to retrieve data
    - Perform **inserts**, **updates**, and **deletes** on data
    - Handle **administrative functions** such as creating users

Setting Up Spring Boot Project

- Automatic **Data Source Configuration**
    - In **Spring Boot**, **Hibernate** is the default implementation of **JPA**
    - **EntityManager** is the **main component** for creating **queries**
    - **EntityManager** is from **Jakarta Persistence API (JPA)**
    - Based on **configurations**, **Spring Boot** will **automatically create** the **beans**
        - You can then **inject** these into your app
- **Spring Boot - Auto Configuration**
    - **Spring Boot** will **automatically configure** your data source for you
    - Based on entries from the **Maven pom** file
        - **JDBC Driver → mysql-connector-j**

        - **Spring Data (ORM) → spring-boot-starter-data-jpa**
    - **DB** connection info from **application.properties**

JPA Annotations

- **JPA Development Process - To-do List**
    - **Annotate** Java class
    - Develop **Java** code to perform **database operations**
- **Entity** class
    - **Java class** that is **mapped** to a **database table**
    - At a **minimum**, the **Entity class**
        - Must be annotated with **@Entity**
        - Must a have a **public** or **protected no-argument constructor**
            - The class **can have other constructors**
- **Java Annotations**
    - Step 1: **Map class** to **database table**
    - Step 2: **Map fields** to **database columns**
- **@Column** Annotation
    - This is an **optional annotation**
    - If **not specified**, the **column name** is the **same name** as the **Java field**
        - **Not recommended**
            - If you **refactor** the **Java** code, then it **won't match** existing **database columns**
            - This is a **breaking change** and you will need to **update database columns**
    - The same applies to **@Table → Database table name** is the **same** as the **Java class**
- **Primary Key**
    - **Uniquely** identifies each **row** in a **table**
    - Must be a **unique value**
    - **Cannot** contain **null** values
- **ID Generation Strategies**

Name	Description
GenerationType.AUTO	Pick an **appropriate strategy** for the **particular database**
GenerationType.IDENTITY	Assign primary keys using **database identity column**; generally **recommended**
GenerationType.SEQUENCE	Assign primary keys using a **database sequence**
GenerationType.TABLE	Assign primary keys using an **underlying database table** to **ensure uniqueness**

- **Custom Generation Strategy**
  - Create implementation of **org.hibernate.id.IdentifierGenerator**
  - Override **public Serializable generate(...)**

Saving a Java Object with JPA

- Student Data **Access Object**
  - Responsible for **interfacing** with the **database**
  - This is a common **design pattern** → **D**ata **A**ccess **O**bject **(DAO)**
  - Needs a **JPA Entity Manager**
  - **JPA Entity Manager** is the **main component** for **saving/retrieving entities**
- **JPA Entity Manager**
  - Needs a **Data Source**
  - The **Data Source** defines **database connection info**
  - **JPA Entity Manager** and **Data Source** are **automatically created** by **Spring Boot**
    - Based on the file → **application.;properties**
  - We can **autowire/inject** the **JPA Entity Manager** into our **Student DAO**

- **Student DAO Development Process**
  - Step 1: **Define DAO interface**
  - Step 2: **Define DAO implementation → Inject the entity manager**
  - Step 3: **Update the main app**
- **Spring @Transactional**
  - **Automatically begin/end** a transaction for you **JPA** code
    - **No need** to do this **explicitly**
- Specialized **Annotations** for **DAOs**
  - **@Repository**
  - Applied to **DAO implementations**
  - **Spring** will **automatically register** the **DAO implementation**
    - Thanks to **component-scanning**
  - **Spring** also provides **translation** of any **JDBC-related exceptions**

Reading Objects with JPA

- **Development Process**
  - Add new **method** to **DAO interface**
  - Add new **method** to **DAO implementation**
  - Update **main app**

Querying Objects with JPA

- **JPA Query Language (JPQL)**
  - **Query language** for **retrieving objects**
  - Similar to **SQL**
  - Based on **entity name** and **entity fields**
- **Named Parameters**
- **Development Process**
  - Add new **method** to **DAO interface**
  - Add new **method** to **DAO implementation**
  - Update **main app**

Updating Objects with JPA

- **Update** using **merge**
- **Development Process**

    - Add new **method** to **DAO interface**
    - Add new **method** to **DAO implementation**
    - Update **main app**

Deleting Objects with JPA

- **.executeUpdate()** to the end of the **query**
- **Development Process**
    - Add new **method** to **DAO interface**
    - Add new **method** to **DAO implementation**
    - Update **main app**

Create Database Tables from Java Code

- **Create Database Tables**
    - **JPA/Hibernate** can **automatically create database tables**
    - **Create tables** based on **Java** code with **JPA/Hibernate annotations**
    - Useful for **development** and **testing**
- **Configuration**
    - In **application.properties**
        - **spring.jpa.hibernate.deel-auto=create**
    - **JPA/Hibernate** will **drop tables**, then **create** them from scratch

Property Value	Property Description
none	**No action** will be performed
create-only	Database tables are only **created**
drop	Database tables are **dropped**
create	Database tables are **dropped**, followed by database tables **creation**

create-drop	Database tables are **dropped**, followed by database tables **creation**. On application **shutdown**, **drop** the database tables
validate	**Validate** the database table **schema**
update	**Update** the database table **schema**

- Basic Projects
  - Auto-Configuration → **spring.jpa.hibernate.ddl-auto=create**
  - Database tables are **doppped** first and then **created** from scratch
  - **When database tables are dropped, all data is lost**
  - If you want to **create tables once** and **keep the data**
    - Use **update**
    - **spring.jpa.hibernate.ddl-auto=update**
    - Will **alter database schema** based on latest code updates
- Use Case
  - **Database integration testing** with **in-memory databases**
  - **Basic**, small hobby projects
- **Recommendation**
  - **Don't recommend auto generation for enterprise**, real-time projects
    - You **can easily drop production data** if you're not careful
  - **SQL scripts** are **recommended**
    - Corporate DBAs prefer SQL scripts for **governance** and **code review**
    - SQL scripts can be **customized** and **fine-tuned** for complex database designs
    - SQL scripts can be **version-controlled**
    - SQL scripts can also **work with schema migration tools** such as Liquibase and Flyway