**Day 26 Spring Data JPA.**

Agenda

What is JPA

JPA Implementations.

ORM Framework.

How ORM works.

Spring Boot JPA implementation.

Configuring Multiple SQL based DBs for different Environment.

Day 25 Revisit.

Centralized Logging using ELK Stack (Elastic Stack)

ELK – Elastic Search, Logstash & Kibana. (Running all 3 tools locally by installing and configuring it)

Executed a Project using Docker and Docker-compose.

Logging, Log Levels, Logging Config,

JDBC is a Specification (Connection, Driver, Statement, PreparedStatement,CallableStatement, ResultsetMetaData, DatabaseMetaDeta. JPA is a Specification. (Contains only Interfaces & Abstract classes).

**🧩 1. What is JPA?**

**JPA (Java Persistence API)** is a **specification** that defines how Java objects map to relational database tables.  
It abstracts away boilerplate JDBC code by handling ORM (Object Relational Mapping).

Think of it as a **contract** — not an implementation.

JPA redefines how we interact with DBs (SQL based)

MySQL

Postgres

MSSQL

DB2 etc.,

We write and execute queries in the database client or using programs.

**🧱 2. Common Implementations of JPA**

| **Implementation** | **Description** | **Notes** |
| --- | --- | --- |
| **Hibernate** | Most widely used JPA implementation. Mature and feature-rich. | Default choice in Spring Boot. |
| **EclipseLink** | Reference implementation for JPA (by Eclipse Foundation). | Used in some enterprise setups. |
| **OpenJPA** | Apache’s JPA implementation. | Less popular now. |
| **DataNucleus** | Supports non-relational datastores too. | Used in niche projects. |

Spring Data JPA builds on top of **Hibernate** by default, offering repository abstractions and query simplifications.

ORM – Object Relational Mapping (Java, SQL entity, mapping – connection)

If you want to try EclipseLink projects use Eclipse EE IDE.

**🧠 3. Why Use JPA?**

* Reduces boilerplate (no manual SQL or JDBC code).
* Enables database-agnostic development.
* Supports caching, lazy loading, and transaction management.
* Integrates seamlessly with **Spring Boot**.

**🛠️ 4. How to Implement JPA in Spring Boot**

1. **Add dependencies**
2. <dependency>
3. <groupId>org.springframework.boot</groupId>
4. <artifactId>spring-boot-starter-data-jpa</artifactId>
5. </dependency>
6. <dependency>
7. <groupId>com.h2database</groupId>
8. <artifactId>h2</artifactId>
9. <scope>runtime</scope>
10. </dependency>
11. **Configure application.properties**
12. spring.datasource.url=jdbc:h2:mem:testdb
13. spring.jpa.hibernate.ddl-auto=update
14. spring.jpa.show-sql=true
15. **Define Entity**
16. @Entity
17. @Table(name = "students")
18. public class Student {
19. @Id
20. @GeneratedValue(strategy = GenerationType.IDENTITY)
21. private Long id;
22. private String name;
23. }
24. **Create Repository**
25. public interface StudentRepository extends JpaRepository<Student, Long> {}
26. **Use in Service**
27. @Service
28. public class StudentService {
29. @Autowired
30. private StudentRepository repo;
31. public List<Student> getAll() {
32. return repo.findAll();
33. }
34. }

Hibernate uses HQL (Hibernate Query Lang)

**🔗 5. Relationships Between Entities**

| **Relationship** | **Annotation** | **Example** |
| --- | --- | --- |
| **One-To-One** | @OneToOne | Person ↔ Aadhaar (PAN/Passport) |
| **One-To-Many** | @OneToMany | Department → Students |
| **Many-To-One** | @ManyToOne | Students → Department |
| **Many-To-Many** | @ManyToMany | Students ↔ Courses |

Always define the **owning side** carefully using mappedBy.

**Example:**

@Entity

public class Department {

@Id

private Long id;

private String name;

@OneToMany(mappedBy = "department", cascade = CascadeType.ALL)

private List<Student> students;

}

@Entity

public class Student {

@Id

private Long id;

private String name;

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

}

**🧾 6. Key Annotations in JPA**

| **Annotation** | **Purpose** |
| --- | --- |
| @Entity | Marks class as a JPA entity |
| @Table | Maps entity to specific table |
| @Id | Defines primary key |
| @GeneratedValue | Auto-generates PK values |
| @Column | Customizes column mapping |
| @Transient | Excludes a field from persistence |
| @Lob | Large objects (e.g., text, blob) |
| @Enumerated | Persists enums |
| @Embedded / @Embeddable | For value objects |
| @Version | Optimistic locking |
| @NamedQuery | Defines static queries |

Users (Table Name) – select \* from users; (SQL) -- User (Java Class) – select u from User u; (HQL – Hibernate Query Lang) – DB independent.

**💬 7. Custom Queries and Joins**

**Derived Queries**

List<Student> findByDepartmentName(String name);

**JPQL Queries**

@Query("SELECT s FROM Student s WHERE s.department.name = :name")

List<Student> findByDept(@Param("name") String name);

**Native Queries**

@Query(value = "SELECT \* FROM students WHERE name = :name", nativeQuery = true)

List<Student> findByNative(@Param("name") String name);

**Joins**

@Query("SELECT s FROM Student s JOIN s.department d WHERE d.name = :deptName")

List<Student> findStudentsByDept(@Param("deptName") String deptName);

**⚙️ 8. Executing Stored Procedures & Functions**

**Method 1: Using @Procedure**

@Procedure(name = "get\_student\_count")

Integer getStudentCount();

**Method 2: Using @NamedStoredProcedureQuery**

@NamedStoredProcedureQuery(

name = "Student.getByDept",

procedureName = "get\_students\_by\_dept",

parameters = {

@StoredProcedureParameter(mode = ParameterMode.IN, name = "dept", type = String.class)

}

)

**Method 3: Using EntityManager**

StoredProcedureQuery query = entityManager

.createStoredProcedureQuery("get\_students\_by\_dept")

.registerStoredProcedureParameter("dept", String.class, ParameterMode.IN)

.setParameter("dept", "IT");

**🔍 9. Indexing & Performance**

* Use @Index in @Table
* @Table(name = "students", indexes = @Index(name = "idx\_student\_name", columnList = "name"))
* Index frequently searched columns.
* Prefer pagination (Pageable) over fetching large lists.
* Avoid **N+1 query problem** by using @EntityGraph or fetch = FetchType.LAZY.
* Use caching (spring.jpa.properties.hibernate.cache.use\_second\_level\_cache=true).
* Batch inserts/updates with spring.jpa.properties.hibernate.jdbc.batch\_size.

**🌿 10. Best Practices**

* Keep entities **pure** (avoid business logic inside them).
* Always use @Transactional in service layer, not in repository.
* Use DTOs for projections instead of exposing entities directly.
* Prefer **JPQL or Criteria** over native queries for portability.
* Clearly define **cascades and fetch types** to avoid unintended deletes or performance hits.
* Validate data with JSR-380 annotations (@NotNull, @Size, etc.).

Two types of Operations in DB

* 1. Read (It wont changes anything in DB)
  2. Modify (Insert/Delete/Update) [Changes/Modify the Table]

<https://hibernate.org/orm/>

EclipseLink Download Link --- <https://www.eclipse.org/downloads/download.php?file=/rt/eclipselink/releases/4.0.8/eclipselink-4.0.8.v20250923-f4449e70fd.zip>

<https://wiki.eclipse.org/EclipseLink/Examples/>

CREATE TABLE `eclipselink`.`employee` (

`id` INT NOT NULL AUTO\_INCREMENT,

`name` VARCHAR(150) NOT NULL,

`email` VARCHAR(250) NOT NULL,

`mobile` BIGINT(10) NULL,

PRIMARY KEY (`id`),

UNIQUE INDEX `email\_UNIQUE` (`email` ASC) VISIBLE,

UNIQUE INDEX `mobile\_UNIQUE` (`mobile` ASC) VISIBLE);

**🧩 1. Database Connection**

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=secret

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

If you’re using an in-memory database for testing:

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driver-class-name=org.h2.Driver

spring.h2.console.enabled=true

**⚙️ 2. Hibernate & JPA Core Settings**

# Show SQL statements in the console

spring.jpa.show-sql=true

# Format SQL output for readability

spring.jpa.properties.hibernate.format\_sql=true

# Choose how Hibernate handles schema creation

spring.jpa.hibernate.ddl-auto=update

# options: none | validate | update | create | create-drop

# "update" = safe for dev, avoid in production

# Specify the underlying JPA implementation

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

💡 hibernate.dialect helps Hibernate translate HQL into optimized SQL for your database.

**🧠 3. Naming Strategies**

By default, Hibernate converts camelCase field names into snake\_case columns.  
You can customize that:

# Disable automatic naming conversions

spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

**🧩 4. Connection Pool (HikariCP)**

Spring Boot uses **HikariCP** by default — it’s fast and production-ready.

spring.datasource.hikari.maximum-pool-size=10

spring.datasource.hikari.minimum-idle=5

spring.datasource.hikari.idle-timeout=30000

spring.datasource.hikari.max-lifetime=1800000

spring.datasource.hikari.connection-timeout=20000

**🧾 5. Logging & SQL Debugging**

logging.level.org.hibernate.SQL=DEBUG

logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE

This logs both SQL statements and their bound parameter values — helpful for debugging.

**📦 6. Caching (Optional)**

# Enable second-level cache

spring.jpa.properties.hibernate.cache.use\_second\_level\_cache=true

spring.jpa.properties.hibernate.cache.region.factory\_class=org.hibernate.cache.jcache.JCacheRegionFactory

spring.jpa.properties.javax.cache.provider=org.ehcache.jsr107.EhcacheCachingProvider

**🔍 7. Batch Processing & Performance**

# Optimize batch inserts/updates

spring.jpa.properties.hibernate.jdbc.batch\_size=50

spring.jpa.properties.hibernate.order\_inserts=true

spring.jpa.properties.hibernate.order\_updates=true

# Avoid fetching too many lazy associations at once

spring.jpa.properties.hibernate.default\_batch\_fetch\_size=16

**🧩 8. Transaction Management**

# Enable automatic transaction rollback on commit failure

spring.jpa.properties.hibernate.connection.provider\_disables\_autocommit=true

**🧠 Typical Minimal Config (Dev Mode)**

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

**🧩 application.yml (Full Example)**

spring:

datasource:

url: jdbc:mysql://localhost:3306/mydb

username: root

password: secret

driver-class-name: com.mysql.cj.jdbc.Driver

hikari:

maximum-pool-size: 10

minimum-idle: 5

idle-timeout: 30000 # 30s

max-lifetime: 1800000 # 30min

connection-timeout: 20000 # 20s

jpa:

show-sql: true

hibernate:

ddl-auto: update # none | validate | update | create | create-drop

properties:

hibernate:

dialect: org.hibernate.dialect.MySQL8Dialect

format\_sql: true

jdbc:

batch\_size: 50

order\_inserts: true

order\_updates: true

default\_batch\_fetch\_size: 16

cache:

use\_second\_level\_cache: true

region.factory\_class: org.hibernate.cache.jcache.JCacheRegionFactory

connection:

provider\_disables\_autocommit: true

open-in-view: false # good practice to disable in production

logging:

level:

org.hibernate.SQL: DEBUG

org.hibernate.type.descriptor.sql.BasicBinder: TRACE

**🧪 Lightweight Dev Config (H2 Example)**

spring:

datasource:

url: jdbc:h2:mem:testdb

driver-class-name: org.h2.Driver

h2:

console:

enabled: true

jpa:

hibernate:

ddl-auto: update

show-sql: true

properties:

hibernate:

format\_sql: true

For Multiple Environments

**🧱 1. Base Configuration — application.yml**

spring:

profiles:

active: dev # default active profile

jpa:

show-sql: true

hibernate:

ddl-auto: none # override in dev/test

open-in-view: false

properties:

hibernate:

format\_sql: true

jdbc:

batch\_size: 50

order\_inserts: true

order\_updates: true

default\_batch\_fetch\_size: 16

connection:

provider\_disables\_autocommit: true

logging:

level:

org.hibernate.SQL: DEBUG

org.hibernate.type.descriptor.sql.BasicBinder: TRACE

This file defines the *shared logic* — JPA, Hibernate behavior, logging, etc.  
Each environment adds its own connection details and schema strategy.

**🧪 2. Development — application-dev.yml**

spring:

datasource:

url: jdbc:h2:mem:devdb

driver-class-name: org.h2.Driver

username: sa

password:

hikari:

maximum-pool-size: 5

minimum-idle: 2

jpa:

hibernate:

ddl-auto: update # auto-create/update schema during dev

**Usage:**

-Dspring.profiles.active=dev

**🧩 3. Testing — application-test.yml**

spring:

datasource:

url: jdbc:h2:mem:testdb

driver-class-name: org.h2.Driver

username: sa

password:

jpa:

hibernate:

ddl-auto: create-drop # fresh schema each test run

h2:

console:

enabled: true

**🏭 4. Production — application-prod.yml**

spring:

datasource:

url: jdbc:mysql://prod-db-server:3306/mydb

username: prod\_user

password: ${DB\_PASSWORD}

driver-class-name: com.mysql.cj.jdbc.Driver

hikari:

maximum-pool-size: 20

minimum-idle: 10

idle-timeout: 30000

max-lifetime: 1800000

connection-timeout: 20000

jpa:

hibernate:

ddl-auto: validate # ensure schema matches entity mappings

jpa:

show-sql: false # avoid logging SQL in prod

properties:

hibernate:

dialect: org.hibernate.dialect.MySQL8Dialect

cache:

use\_second\_level\_cache: true

region.factory\_class: org.hibernate.cache.jcache.JCacheRegionFactory

**💡 How it Works**

* Spring Boot automatically loads:
  + application.yml (base)
  + application-{profile}.yml (overrides)
* Active profile is chosen by:
  + Environment variable: SPRING\_PROFILES\_ACTIVE=prod
  + JVM argument: -Dspring.profiles.active=prod
  + Or directly in application.yml