DAY 6-7(10/10/2025)
JPA AND HIBERNATE ORM FUNDAMENTALS

1. What is JPA?

- JPA stands for **Java Persistence API**.
- It is a **specification (set of rules)** that defines how Java objects (entities) can be stored in and retrieved from a database.
- JPA is **not a framework** — it only provides interfaces and annotations.
- To actually perform the operations, JPA needs a **provider (implementation)** such as Hibernate.

Example analogy:

- JPA → Blueprint (rules).
- Hibernate → Actual builder (implementation).

2. Why do we need JPA?

Without JPA:

- You would write raw SQL queries (SELECT, INSERT, UPDATE, DELETE).
- You would manually handle connections, statements, and mappings.

With JPA:

- You can work with **Java objects** instead of SQL.
- JPA automatically maps the objects to database tables.
- Reduces boilerplate code and improves maintainability.

3. What is ORM?

- ORM stands for **Object Relational Mapping**.
- It is the technique of mapping **Java objects to database tables**.
- Each object corresponds to a row in the table, and each field corresponds to a column.

Example:

Java Class (Object)	Database Table
User class	users table
id field	id column
name field	name column

This mapping is handled automatically by ORM tools like Hibernate.

4. What is Hibernate?

- Hibernate is the **most popular ORM framework** and the **main JPA provider** used in Spring Boot.

- It implements all JPA rules and provides extra features (caching, lazy loading, etc.).
- Converts JPA annotations into actual SQL queries and executes them on the database.

5. Key Features of Hibernate

1. **Automatic Table Mapping**
    - Maps Java classes to tables using annotations like @Entity, @Table.
2. **Automatic SQL Generation**
    - You don't need to write SQL manually; Hibernate generates it.
3. **Transaction Management**
    - Handles commits, rollbacks, and session management.
4. **Caching**
    - Reduces database hits by storing frequently used data in memory.
5. **Lazy Loading**
    - Loads related data only when it's needed.
6. **HQL / JPQL Support**
    - Allows you to write queries using class and field names, not table names.

6. Core JPA Annotations (Basic)

Annotation	Meaning
@Entity	Marks the class as a JPA entity (mapped to a table)
@Table(name="table_name")	Defines the table name
@Id	Defines the primary key
@GeneratedValue	Auto-generates primary key value
@Column(name="column_name")	Maps a field to a specific column

Example:

```
@Entity
@Table(name = "users")
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   @Column(nullable = false)
   private String name;
   private String email;
}
```

This automatically creates a table users with columns id, name, and email.

7. Lifecycle of an Entity

1. **Transient** – Object created but not associated with a database.
2. **Persistent** – Object is associated with a database (saved).

3. **Detached** – Object was saved but now not managed by Hibernate session.
4. **Removed** – Object is deleted from database.

8. JPA Configuration in Spring Boot

Spring Boot automatically configures JPA when you include:
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
runtimeOnly 'com.h2database:h2' // or MySQL connector
In application.properties:
spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

9. Hibernate Architecture (Simplified)

Java Application
↓
JPA (Specification)
↓
Hibernate (Implementation)
↓
JDBC (Driver)
↓
Database (MySQL / H2 / PostgreSQL)

Hibernate converts Java entity operations into SQL and communicates with the database through JDBC.

10. Summary

Concept	Description
JPA	Specification for ORM in Java
Hibernate	Implementation of JPA
ORM	Maps Java objects to database tables
Entity	Java class representing a table
Persistence	Saving objects to the database
Hibernate handles	SQL generation, transaction, caching, lazy loading

ENTITY MAPPING ANNOTATIONS

1. What is Entity Mapping?

- **Entity mapping** is the process of linking a **Java class** and its fields to a **database table** and its columns.
- It is the core idea of **Object Relational Mapping (ORM)**.
- Spring Data JPA (through Hibernate) uses **annotations** to define how this mapping happens.

2. @Entity

- Marks a **Java class** as a JPA **entity** (a class that maps to a table).
- Every JPA entity must have this annotation.
- The table name is automatically taken from the class name (if @Table is not specified).

Example:

```
@Entity
public class User {
   // fields, getters, setters
}
```

This maps to a database table named **user** (or **users** depending on naming strategy).

3. @Table

- Defines the **table name** and additional table-level details (like schema).
- Optional — if not specified, JPA uses the class name as the table name.

Example:

```
@Entity
@Table(name = "users")
public class User {
   // fields
}
```

→ Creates or maps to a table named **users**.

4. @Id

- Marks a field as the **primary key** of the table.
- Every entity must have **one and only one** field annotated with @Id.

Example:

```
@Id
private Long id;
```

5. @GeneratedValue

- Used with @Id to **auto-generate primary key values**.
- The strategy decides how the value is generated.

Strategies:

Strategy	Description
IDENTITY	Uses database auto-increment column
SEQUENCE	Uses a sequence (common in PostgreSQL, Oracle)
TABLE	Uses a separate table to generate IDs

AUTO	Hibernate chooses the strategy automatically

Example:

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

6. @Column

- Used to map a **Java field to a specific column** in the table.
- Optional — JPA maps fields automatically if names match, but you can customize using this annotation.

Example:

```
@Column(name = "user_name", nullable = false, unique = true)
private String name;
```

Common properties:

Property	Meaning
name	Custom column name
nullable	Whether column can be null
unique	Whether column should be unique
length	Length for string columns
columnDefinition	Custom SQL column definition

7. @Transient

- Marks a field that **should not be stored in the database**.
- Useful for temporary or computed values.

Example:

```
@Transient
private int tempValue;
```

This field will not be persisted in the database.

8. Example – Full Entity

```
@Entity
@Table(name = "users")
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   @Column(name = "user_name", nullable = false)
   private String name;
   @Column(unique = true)
   private String email;
```

```
    private Double balance;  // auto-mapped column (same name)
    @Transient
    private boolean loggedIn;  // not saved in DB
}
```

Resulting Table Structure (simplified):

Column Name	Type	Constraints
id	bigint	Primary Key, Auto Increment
user_name	varchar	Not Null
email	varchar	Unique
balance	double	—
loggedIn	—	Not stored

RELATIONSHIP MAPPINGS IN JPA

Relationship mappings define how **entities are connected** in the database.

1. @OneToOne

- Represents a **one-to-one relationship** between two entities.
- Each row in table A maps to **exactly one row** in table B.

Example:

```
@Entity
public class Wallet {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToOne
    @JoinColumn(name = "user_id")  // foreign key in wallet table
    private User user;
}
```

Notes:

- @JoinColumn specifies the **foreign key** column.
- By default, fetch type is EAGER (loads related entity immediately).

2. @OneToMany

- Represents a **one-to-many relationship**.
- One entity is related to **many entities**.

Example:

```
@Entity
public class User {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List<Wallet> wallets;
}
```

Notes:

- mappedBy tells JPA that the **other side owns the relationship**.
- cascade controls operations like persist, remove, etc.
- Default fetch type is LAZY (related entities loaded on demand).

3. @ManyToOne

- Represents a **many-to-one relationship**.
- Many rows in table A map to **one row** in table B.

Example:

```
@Entity
public class Wallet {
    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;
}
```

Notes:

- Often used with @OneToMany on the other side.
- Default fetch type is EAGER.

4. @ManyToMany

- Represents a **many-to-many relationship**.
- Requires a **join table** to map the association.

Example:

```
@Entity
public class User {
    @ManyToMany
    @JoinTable(
        name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id")
    )
    private Set<Role> roles;
}
```

Notes:

- @JoinTable defines the **join table and columns**.
- Both sides can have @ManyToMany.

5. Fetch Strategies

- **EAGER**: Load related entity **immediately**.
- **LAZY**: Load related entity **on-demand** (when accessed).

Default:

Relationship	Fetch Type
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

6. Cascade Types

- Controls how operations on **parent entity** affect **related entities**.
- Common types:
Type	Description
ALL	All operations cascade
PERSIST	Persist related entities
MERGE	Merge changes to related entities
REMOVE	Delete related entities
REFRESH	Refresh related entities
DETACH	Detach related entities from persistence context

REPOSITORY PATTERN & SPRING DATA JPA REPOSITORIES

1. Repository Pattern

- **Purpose:** Abstracts data access logic from business logic.
- **Benefits:**
  - Encapsulates CRUD operations.
  - Makes code **cleaner** and **maintainable**.
  - Easier to **test** (can mock repository in unit tests).

Key Idea: Your service interacts with a **repository interface**, not directly with the database.

2. Spring Data JPA Repositories

Spring Data JPA provides **ready-to-use repository interfaces** that implement CRUD operations.

Common Repository Interfaces:

Interface	Description
CrudRepository<T, ID>	Basic CRUD operations (save, findById, delete, etc.)
JpaRepository<T, ID>	Extends CrudRepository, adds JPA-specific operations (flush, saveAll, deleteInBatch)
PagingAndSortingRepository<T, ID>	Adds pagination and sorting support

3. Defining a Repository

Example – User Repository:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
   // Custom queries can be added here
}
```

- JpaRepository<User, Long> → User is entity, Long is primary key type.
- No implementation class needed; Spring generates it at runtime.

4. Using Repository in Service

Example:

```
@Service
public class UserService {
   private final UserRepository userRepository;
   public UserService(UserRepository userRepository) {
      this.userRepository = userRepository;
   }
   public List<User> getAllUsers() {
      return userRepository.findAll();
   }
   public User createUser(User user) {
      return userRepository.save(user);
   }
}
```

- findAll() → Fetch all users.
- save() → Insert or update entity.
- Spring handles the database interactions internally.

5. Custom Query Methods

- Spring Data JPA can generate queries automatically based on **method names**.

Example:

List<User> findByName(String name);
List<User> findByEmailContaining(String keyword);

- findByName → SQL: SELECT * FROM user WHERE name = ?
- findByEmailContaining → SQL: SELECT * FROM user WHERE email LIKE %?%

6. Using @Query for Complex Queries

- For more complex queries, use **JPQL or native SQL** with @Query.

@Query("SELECT u FROM User u WHERE u.wallet.balance > :amount")
List<User> findUsersWithHighBalanceWallets(Double amount);

- :amount → Parameter binding.
- Can use JPQL (entity-based) or nativeQuery=true for raw SQL.

CUSTOM QUERY METHODS & JPQL (JAVA PERSISTENCE QUERY LANGUAGE)

1. Custom Query Methods

- Spring Data JPA can **automatically generate SQL queries** from method names.
- This eliminates the need to write boilerplate queries for simple operations.

Examples:
List<User> findByName(String name);
List<User> findByEmailContaining(String keyword);
List<Wallet> findByBalanceGreaterThan(Double amount);
Explanation:

Method Name	Meaning/SQL Equivalent
findByName	SELECT * FROM user WHERE name = ?
findByEmailContaining	SELECT * FROM user WHERE email LIKE %?%
findByBalanceGreaterThan	SELECT * FROM wallet WHERE balance > ?

2. JPQL (Java Persistence Query Language)

- **JPQL** is similar to SQL but works with **entity objects**, not tables.
- Queries are written in terms of **Java entities and their fields**.

Example:

@Query("SELECT w FROM Wallet w WHERE w.balance > :amount")
List<Wallet> findRichWallets(@Param("amount") Double amount);

- Wallet → Entity class
- w.balance → Entity field
- :amount → Named parameter

3. Native Queries

- For complex or database-specific queries, use nativeQuery=true:

@Query(value = "SELECT * FROM wallets w WHERE w.balance > :amount", nativeQuery = true)
List<Wallet> findRichWalletsNative(@Param("amount") Double amount);

- Directly uses SQL syntax.
- Can be useful for joins or database functions not supported in JPQL.

4. Parameter Binding

- **Named Parameters:** :paramName → @Param("paramName")
- **Positional Parameters:** ?1, ?2, ...

Example:
@Query("SELECT u FROM User u WHERE u.email = ?1 AND u.name = ?2")
User findByEmailAndName(String email, String name);

CASCADE OPERATIONS & FETCH STRATEGIES

1. Cascade Operations

- Cascade determines **what happens to related entities** when performing operations (persist, merge, remove, etc.) on a parent entity.
- Defined using cascade attribute in relationship annotations:

@OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
private List<Wallet> wallets;

Common Cascade Types:

Type	Description
PERSIST	Save the child entities automatically when parent is saved
MERGE	Update child entities automatically when parent is updated
REMOVE	Delete child entities automatically when parent is deleted
REFRESH	Reload child entities from the database when parent is refreshed
DETACH	Detach child entities when parent is detached
ALL	Apply all cascade operations

Example:

```
User user = new User();
Wallet wallet = new Wallet();
user.setWallets(List.of(wallet));
entityManager.persist(user); // wallet is automatically persisted
```

2. Fetch Strategies

- Fetch strategies define **how related entities are loaded** from the database.

Types:

Strategy	Description	Default
EAGER	Loads the related entity immediately with the parent	@ManyToOne, @OneToOne default
LAZY	Loads the related entity **on demand** when accessed	@OneToMany, @ManyToMany default

Example:

```
@OneToMany(mappedBy = "user", fetch = FetchType.LAZY)
private List<Wallet> wallets;
```

- **EAGER:** Fetch all wallets when user is fetched
- **LAZY:** Fetch wallets only when getWallets() is called