TOPIC 3-9

1. Entity Classes and the `@Entity` Annotation

- **What is an Entity?** A plain Java object (POJO) whose instances correspond to rows in a database table.

- **Defining an Entity**

```
import javax.persistence.*;

@Entity                              // Marks this class as a JPA
entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstName;
    private String lastName;

    // getters & setters, no-arg constructor
}
```

- **Key Points**
    - Must have a no-arg constructor (can be `protected`).
    - Each entity class must have a primary key (`@Id`).
    - By default, class name → table name; field name → column name.

---

2. Primary Key Mapping: `@Id` and `@GeneratedValue`

- **`@Id`** Marks the field as the primary key.

- **`@GeneratedValue` Strategies**

```
@GeneratedValue(strategy = GenerationType.AUTO)        // Provider
picks best
@GeneratedValue(strategy = GenerationType.IDENTITY)    // DB auto-
increment
@GeneratedValue(strategy = GenerationType.SEQUENCE)    // Uses a DB
sequence
@GeneratedValue(strategy = GenerationType.TABLE)       // Uses a table
to generate PKs
```

- **Example**

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator="order_seq")
    @SequenceGenerator(name="order_seq",
sequenceName="order_sequence")
    private Long id;
    // ...
}
```

---

3. Field-to-Column Mapping with `@Column`

- **Basic Usage**

```
@Column(name = "first_name", length = 50, nullable = false, unique =
true)
private String firstName;
```

- **Attributes**
  - `name` – custom column name
  - `length` – for `VARCHAR` columns
  - `nullable` – `false` → `NOT NULL`
  - `unique` – adds unique constraint
  - `columnDefinition` – custom SQL fragment

---

4. Table Mapping with `@Table`

- **Default Behavior** Without `@Table`, JPA uses class name as table name.

- **Custom Table Mapping**

```
@Entity
@Table(name = "employees",
       schema = "hr",
       uniqueConstraints = {
         @UniqueConstraint(columnNames = {"email"})
       })
public class Employee { … }
```

- **Attributes**
  - `name` – table name
  - `schema` – DB schema
  - `catalog` – DB catalog
  - `uniqueConstraints` – multi-column unique constraints

---

5. Lifecycle of an Entity

5.1 Entity States

1. **New (Transient)**

    - Created with `new` but not yet attached.
    - No database representation.

2. **Managed (Persistent)**
    - Attached to a persistence context (via `persist` or retrieval).
    - Changes auto-detected and synchronized at flush/commit.

3. **Detached**
    - Was managed, then the persistence context closed or entity evicted.
    - Changes won't be synchronized unless re-merged.

4. **Removed**
    - Marked for deletion via `remove()`.
    - Deleted at flush/commit.

5.2 Common Operations

Operation	**Effect**
`persist(e)`	New → Managed. Schedules `INSERT`.
`find()`	Database → Managed. Retrieves entity by PK.
`merge(e)`	Detached → copies state to Managed instance; returns Managed.
`remove(e)`	Managed → Removed. Schedules `DELETE`.
`flush()`	Synchronizes in-memory changes to the database immediately.
`refresh(e)`	Overwrites entity state with database values.

6. `EntityManager` & Persistence Context

6.1 Understanding `EntityManager`

- **Role:** API for CRUD operations, queries, and transaction control.
- **Obtaining an `EntityManager`:**

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("unitName");
EntityManager em = emf.createEntityManager();
```

- **Persistence Context:**
    - The "first-level cache" where managed entities live.
    - Within a transaction, `EntityManager` guarantees identity: repeated `find()` calls return the same Java object.

6.2 `persistence.xml` and Configuration

```
<persistence xmlns="…">
  <persistence-unit name="unitName" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>com.example.Person</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:…"/>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <!-- ... -->
    </properties>
  </persistence-unit>
</persistence>
```

- In **Spring Boot**, much of this is auto-configured via `application.properties`.

---

7. JPQL and Native Queries

7.1 JPQL vs Native SQL

- **JPQL (Java Persistence Query Language)**
  - Object-oriented: queries on entities and their relationships.
  - Portable across databases.

```
List<Person> adults = em.createQuery(
    "SELECT p FROM Person p WHERE p.age >= :minAge", Person.class)
    .setParameter("minAge", 18)
    .getResultList();
```

- **Native SQL**
  - Database-specific SQL.

```
List<Object[]> rows = em.createNativeQuery(
    "SELECT first_name, last_name FROM person WHERE age >= ?")
    .setParameter(1, 18)
    .getResultList();
```

7.2 `@Query` in Spring Data JPA

```
public interface PersonRepository extends JpaRepository<Person, Long> {
  @Query("SELECT p FROM Person p WHERE p.lastName = :ln")
  List<Person> findByLastName(@Param("ln") String lastName);

  @Query(value = "SELECT * FROM person WHERE age > :age", nativeQuery =
true)
  List<Person> findOlderThan(@Param("age") int age);
}
```

7.3 Named Queries

- **Defined on Entity:**

```
@Entity
@NamedQuery(
  name = "Person.byName",
  query = "SELECT p FROM Person p WHERE p.firstName = :fn"
)
public class Person { … }
```

- **Usage:**

```
em.createNamedQuery("Person.byName", Person.class)
  .setParameter("fn", "Alice")
  .getResultList();
```

7.4 Criteria API (Basic Intro)

- **Type-safe, programmatic query builder.**

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Person> cq = cb.createQuery(Person.class);
Root<Person> root = cq.from(Person.class);
cq.select(root)
  .where(cb.equal(root.get("lastName"), "Doe"));
List<Person> results = em.createQuery(cq).getResultList();
```

📚 Further Reading & Resources

- **Official JPA Spec (JSR 338)**
- **Hibernate User Guide** (for provider-specific extensions)
- **Spring Data JPA Reference**

— **First-Level Cache** *(Persistence Context Cache)* —

1. **How It Works with `EntityManager` / Hibernate `Session`

    - Every `EntityManager` (or Hibernate `Session`) instance maintains its own first-level cache, also called the **persistence context**.
    - When you call `find()`, `persist()`, or `merge()`, the entity instance is stored in this cache.
    - Subsequent operations for the same entity (same type + primary key) within that `EntityManager` session hit the cache instead of issuing another SQL `SELECT`.
    - At transaction commit or on explicit `flush()`, changes are synchronized to the database, but the cached entities remain managed until you close the `EntityManager` or clear it.

2. **Identity Guarantee & Caching in the Same Transaction/Session**

    - **Identity Guarantee:** Within one `EntityManager` session, two retrievals of the same row always return the *same Java object instance*.

    - **Example:**

```
Person p1 = em.find(Person.class, 1L);
Person p2 = em.find(Person.class, 1L);
// p1 == p2 → true, because both refer to the same cached object
```

- This behavior prevents accidental data inconsistency and unnecessary round-trips to the database.

— **Second-Level Cache** *(Session Factory–Level Cache)* —

1. **Difference from First-Level Cache**

Aspect	First-Level Cache	Second-Level Cache
Scope	Single `EntityManager`/`Session`	Across multiple sessions/factories
Lifetime	Tied to persistence context	Lives as long as the SessionFactory (application start–stop)
Cached Data	Entity instances only	Entities, collections, query results (optional)
Configuration	Automatic, no extra setup	Manual: choose provider (e.g., EHCache, Infinispan)

2. **Enabling & Configuring (EHCache Example)**

- **Add dependency** (Maven example):

```
<dependency>
  <groupId>org.ehcache</groupId>
  <artifactId>ehcache</artifactId>
  <version>3.10.0</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-ehcache</artifactId>
  <version>${hibernate.version}</version>
</dependency>
```

- **Hibernate Configuration** (in `application.properties` or `persistence.xml`):

```
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
spring.jpa.properties.javax.cache.uri=classpath:ehcache.xml
```

- **`ehcache.xml`** (defining cache regions):

```xml
<ehcache:config
  xmlns:ehcache="http://www.ehcache.org/v3">
  <ehcache:cache alias="com.example.Person">
    <ehcache:heap unit="entries">1000</ehcache:heap>
    <ehcache:expiry>
      <ehcache:ttl unit="seconds">600</ehcache:ttl>
    </ehcache:expiry>
  </ehcache:cache>
</ehcache:config>
```

3. **Annotations: `@Cacheable` & `@Cache`**

    - **`@Cacheable`** (Hibernate-specific) on entity:

    ```java
    @Entity
    @Cacheable
    @org.hibernate.annotations.Cache(
      usage = CacheConcurrencyStrategy.READ_WRITE
    )
    public class Person { … }
    ```

    - You can also annotate associations (collections) to cache relationships.

    - **`region`** attribute (optional) lets you assign a logical cache region name.

4. **Cache Concurrency Strategies**

Strategy	**When to Use**	**Guarantees**
`READ_ONLY`	Data never changes (e.g., lookup tables)	Fastest, no locks
`READ_WRITE`	Data occasionally changes	Uses locks to maintain consistency
`NONSTRICT_READ_WRITE`	Stale data tolerance; small window of staleness acceptable	Relaxed consistency, no strict locks
`TRANSACTIONAL`	JTA environments with XA support	Full transactional guarantees (rarely used)

🔑 Key Takeaways

- **First-level cache** is automatic, per-session, and required for JPA identity guarantee.
- **Second-level cache** is optional, application-wide, and configured to improve read performance across sessions.
- Choose your cache strategy based on how often data changes and how strict your consistency requirements are.
- Always test with realistic loads: caching can boost performance but can also introduce complexity (stale data, locking overhead).

Below is an in-depth guide to JPA relationship mappings, covering the four association types, cascading, fetch strategies, and how to control ownership and join details.

---

1. One-to-One (`@OneToOne`)

▶ Definition

A one-to-one association means each row in table A corresponds to exactly one row in table B.

▶ Example: Shared Primary Key

```
@Entity
public class User {
  @Id
  private Long id;

  @OneToOne(mappedBy = "user", cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
  private Address address;
  // …
}

@Entity
public class Address {
  @Id
  private Long id;

  @OneToOne
  @MapsId                         // share primary key
  @JoinColumn(name = "id")
  private User user;
  // …
}
```

▶ Key Points

- **Owning side**: the entity with the foreign key (no `mappedBy`).
- **Inverse side**: uses `mappedBy` to point to the owning property.
- `@MapsId` shares the same PK value.

---

2. One-to-Many (`@OneToMany`) & Many-to-One (`@ManyToOne`)

These two annotations model the same relationship from different ends.

▶ Example: Customer ↔ Orders

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;

  @OneToMany(mappedBy = "customer",
             cascade = CascadeType.PERSIST,
             fetch = FetchType.LAZY)
  private List<Order> orders = new ArrayList<>();
  // …
}

@Entity
public class Order {
  @Id @GeneratedValue
  private Long id;

  @ManyToOne(fetch = FetchType.EAGER, optional = false)
  @JoinColumn(name = "customer_id")
  private Customer customer;
  // …
}
```

▶ Key Points

- **Owning side**: `@ManyToOne` (owns the foreign key column `customer_id`).
- **Inverse side**: `@OneToMany(mappedBy="…")`.
- By default `@OneToMany` is **LAZY**; `@ManyToOne` is **EAGER**.

---

3. Many-to-Many (`@ManyToMany`)

▶ Example: Students ↔ Courses

```
@Entity
public class Student {
  @Id @GeneratedValue
  private Long id;

  @ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE },
              fetch = FetchType.LAZY)
  @JoinTable(name = "student_course",
    joinColumns = @JoinColumn(name = "student_id"),
    inverseJoinColumns = @JoinColumn(name = "course_id"))
  private Set<Course> courses = new HashSet<>();
  // …
}

@Entity
public class Course {
  @Id @GeneratedValue
  private Long id;

  @ManyToMany(mappedBy = "courses")
  private Set<Student> students = new HashSet<>();
  // …
}
```

▶ Key Points

- Uses a **join table** (`student_course`) instead of a foreign-key column.
- Owning side is where you declare `@JoinTable`; inverse uses `mappedBy`.
- You can also customize join columns and add extra columns (with an entity for the join table).

---

4. Cascading Operations (`CascadeType`)

When you perform an operation on the parent, cascade instructs JPA to propagate it to related children.

CascadeType	Meaning
`PERSIST`	cascade `save`
`MERGE`	cascade `merge`
`REMOVE`	cascade `delete`
`REFRESH`	cascade `refresh`
`DETACH`	cascade `detach`
`ALL`	all of the above

```
@OneToMany(cascade = CascadeType.ALL)
private List<Order> orders;
```

---

5. Fetch Types: **EAGER** vs **LAZY**

Fetch Type	Definition	Default On
`LAZY`	Data is loaded **on demand** (proxy)	Collections (`@OneToMany`, `@ManyToMany`)
`EAGER`	Data is loaded **immediately** (join/fetch)	Single-valued (`@ManyToOne`, `@OneToOne`)

- **When to use LAZY:** large collections or graphs you don’t always traverse.
- **When to use EAGER:** mandatory relationships you always need.

---

6. Ownership & Join Annotations

▶ `mappedBy`

- Placed on the **inverse** side to point to the owning property.
- Tells JPA: “This side is not responsible for the FK/join-table.”

▶ `@JoinColumn`

- Used on the **owning** side of `@OneToOne`, `@ManyToOne`.
- Specifies the foreign key column name.

```
@ManyToOne
@JoinColumn(name = "dept_id", nullable = false)
private Department department;
```

▶ `@JoinTable`

- Used on **owning** side of `@ManyToMany` (or bidirectional `@OneToMany` workaround).
- Defines the join table and the join columns on both sides.

```
@ManyToMany
@JoinTable(name = "student_course",
  joinColumns =  @JoinColumn(name = "student_id"),
  inverseJoinColumns = @JoinColumn(name = "course_id"))
private Set<Course> courses;
```

📝 Putting It All Together

- **Decide the cardinality** (1-1, 1-M, M-1, M-M).

- **Choose the owning side** (the one with the FK or join-table).

- **Annotate**:

  - Owning: `@XToY` + `@JoinColumn` / `@JoinTable`
  - Inverse: `@XToY(mappedBy="…")`

- **Configure cascade** to propagate operations.

- **Set fetch** based on performance vs. convenience.

---

1. Embeddables and Value Types

1.1 `@Embeddable` and `@Embedded`

- **Purpose** Decompose a reusable group of fields into its own class (value type) rather than a full entity.

- **`@Embeddable`** Marks a class whose instances are stored as part of an owning entity's table.

  ```
  @Embeddable
  public class Address {
    private String street;
    private String city;
    private String postalCode;
  ```

```
  // constructors, getters, setters
}
```

- **`@Embedded`** Placed on the owning entity's field to include the embeddable's columns.

```
@Entity
public class Customer {
  @Id @GeneratedValue
  private Long id;

  @Embedded
  private Address address;
  // …
}
```

- **Attribute Overrides** Customize column names for an embedded type:

```
@Embedded
@AttributeOverrides({
  @AttributeOverride(name="city",
column=@Column(name="billing_city")),
  @AttributeOverride(name="postalCode",
column=@Column(name="billing_zip"))
})
private Address billingAddress;
```

1.2 Difference Between Entities and Value Types

Aspect	Entity	Value Type (Embeddable)
Identity	Has its own primary key (`@Id`)	No primary key
Lifecycle	Managed independently	Lifecycle bound to owning entity
Sharing	May be shared across relationships	Not shared—copied whenever used
Mutability	Can be mutable or immutable (with care)	Should be treated as immutable ideally

1.3 Collections of Value Types (`@ElementCollection`)

- **When to use** To store a collection of simple value types or embeddables.
- **Example: List of Strings**

```
@Entity
public class Book {
  @Id @GeneratedValue
  private Long id;

  @ElementCollection
  @CollectionTable(
    name="book_tags",
    joinColumns=@JoinColumn(name="book_id")
  )
  @Column(name="tag")
  private Set<String> tags = new HashSet<>();
  // …
}
```

- **Example: Collection of Embeddables**

```
@Entity
public class Order {
  @Id @GeneratedValue
  private Long id;

  @ElementCollection
  @CollectionTable(name = "order_items", joinColumns =
@JoinColumn(name="order_id"))
  private List<OrderItem> items = new ArrayList<>();
  // …
}

@Embeddable
public class OrderItem {
  private String productCode;
  private int quantity;
}
```

---

2. Transaction Management

2.1 Understanding `@Transactional`

- **Definition** A Spring annotation that defines the scope of a single database transaction.
- **Usage**

```
@Service
public class UserService {
  @Transactional
  public void createUserAndProfile(User user, Profile profile) {
    userRepository.save(user);
    profileRepository.save(profile);
  }
}
```

- **Key Behaviors**
  - **Begin** a transaction when the method starts.
  - **Commit** if the method completes normally.
  - **Rollback** on runtime (unchecked) exceptions by default.

2.2 Declarative Transaction Boundaries

- **Class vs. Method Level**

```
@Transactional  // Applies to all public methods
public class OrderService { … }
```

```
// or

public class OrderService {
  @Transactional  // Only this method is transactional
  public void placeOrder(...) { … }
}
```

- **Propagation** Determines how transactions behave when calling other transactional methods:

Propagation	Behavior
REQUIRED	Join existing or create new if none
REQUIRES_NEW	Suspend existing and create a new one
SUPPORTS	Join existing or run non-transactionally if none
MANDATORY	Must join existing; throw exception if none
NOT_SUPPORTED	Suspend existing and run non-transactionally
NEVER	Run non-transactionally; throw exception if a transaction exists
NESTED	Run within a nested transaction (using savepoints)

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void auditLog(...) { … }
```

2.3 Rollback and Commit Behavior

- **Default Rollback Rules**
  - Rollback on unchecked exceptions (`RuntimeException`, `Error`).
  - Commit on checked exceptions unless `rollbackFor` is specified.
- **Customizing Rollback**

```
@Transactional(
  rollbackFor = { IOException.class },
  noRollbackFor = { IllegalArgumentException.class }
)
public void riskyOperation() { … }
```

- **Isolation Levels** Control how this transaction is isolated from others:

Level	Guarantee
DEFAULT	Use the database’s default
READ_UNCOMMITTED	Allows dirty reads

Level	Guarantee
READ_COMMITTED	Prevents dirty reads
REPEATABLE_READ	Prevents non-repeatable reads
SERIALIZABLE	Full isolation, highest overhead

```
@Transactional(isolation = Isolation.SERIALIZABLE)
public void processPayment(...) { … }
```

📝 Best Practices

- **Keep transactional methods as small as possible.**
- **Avoid `@Transactional` on private methods** (Spring AOP proxies won’t intercept).
- **Use read-only transactions** for queries to hint optimizations:

```
@Transactional(readOnly = true)
public List<Product> listAll() { … }
```

- **Be explicit about rollback rules** if you throw checked exceptions that should trigger rollbacks.
