Java Technologies

Lecture 4

Java Persistence API & Spring Data JPA

Fall, 2025

Agenda

- The Persistence Layer, DBMS, Paradigms, Challenges
- Connecting to a DB, Conncetion Pools, Data Sources
- Object-Relational Mapping (ORM), Impedance Mismatch
- Java Persistence API (JPA)
- Entities, Annotations, Persistence Units & Contexts
- EntityManagers, The Lifecycle of an Entity, Callbacks
- Mapping Associations & Inheritance
- Java Persistence Query Language (JPQL), Criteria API
- JPA Performance Tunning
- Spring Data JPA, JpaRepository, Query Methods
- Configuring a DataSource, Pagination and Sorting
- Criteria Specifications, Auditing

The Persistence Layer

- **The bridge between business logic and data storage.**
- CRUD Operations
- Transaction Management
- Data Consistency
- Performance & Optimization
- Scalability & Flexibility
- Integrity & Security
- **Abstraction From Data Storage**
  - Independence from Technology
  - Separation of Concerns
  - Simplified Domain Models
  - Enhanced Testability

Database Management Systems

Database Type	Purpose	Example
Relational	Schema-based, ACID transactions	Oracle, PostgreSQL
Object-Oriented	Data as objects, no need for object-relational mapping	ObjectDB
Document Store	Flexible (JSON), scalable horizontally, evolving data	MongoDB
Key-Value Store	Fast lookups by key, caching, session data	Redis
Wide-Column	Time-series or large distributed datasets, write-heavy workloads	Cassandra
Graph	Optimized for relationship queries (social networks, fraud detection)	Neo4j
Multi-Model	Relational + NoSQL	YugabyteDB

Paradigms for Accessing RDBMS

Technology	Pros	Cons
JDBC (SQL-centric)	Full control, performance, no abstraction overhead	Low-level, verbose, error-prone
Spring JDBC	Template-based, reduces boilerplate	Still SQL-centric, not OO
JPA (ORM)	Maps objects to relations, less SQL, easy to maintain	Complexity, learning curve
Spring Data JPA	Declarative repository pattern, Less code	Framework "magic"
JOOQ	Type-safe SQL query builder: `select().from()`	SQL-Centric, learning curve
myBatis (Semi-ORM)	Write SQL in XML; map results to Java objects	More boilerplate than JPA
QueryDSL	Type-safe, fluent API for building queries over JPA, SQL, MongoDB	Build-time code generation step

Challenges When Working with Databases

- How to connect to a database?
- How many connections to make?
- When to close the connection(s)?
- How to map objects to relations?
- How to make sure that my queries are correct?
- How to share database config between multiple applications?
- How to coordinate and manage distributed transactions?
- How to ensure scalability and stability under high load?
- How to identify slow queries?
- How to abstract the application from the underlying database?
- How to evolve and maintain the database schema safely?

Connecting to a Database

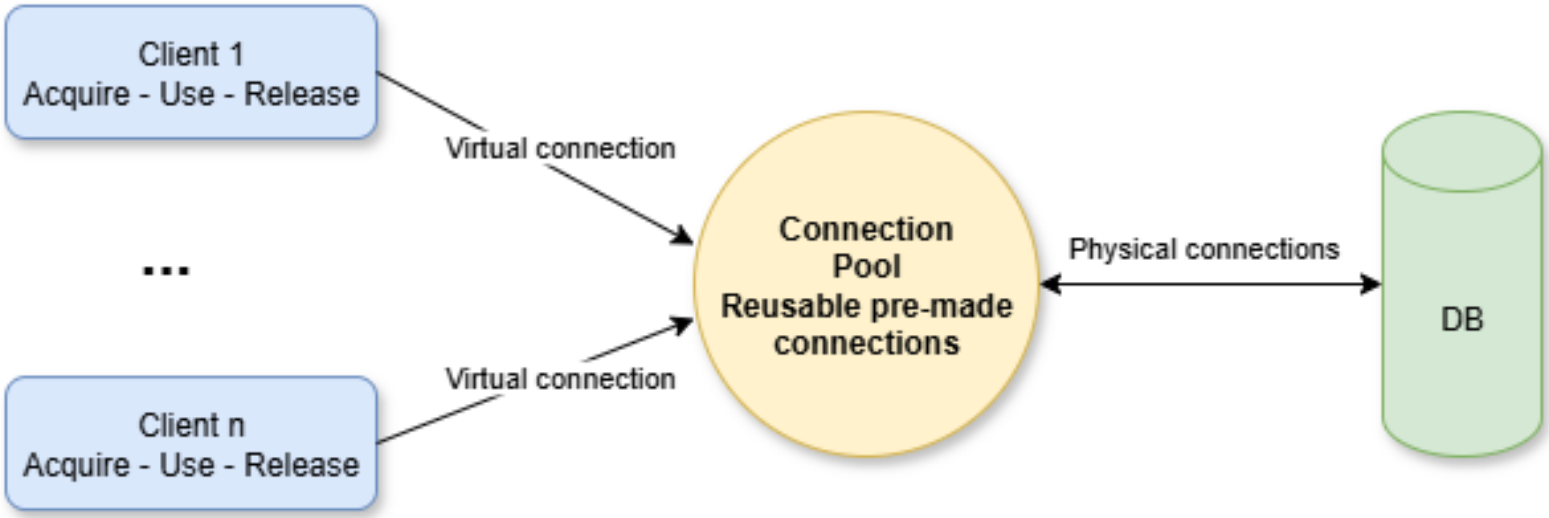
- All RDBMS technologies **use JDBC under the hood**.
- The `java.sql.Connection` interface describes a working session with a specific database.

```
String url = "jdbc:postgres://localhost/mydb"; ❌
Connection conn = DriverManager.getConnection(url, "user", "pwd");
```

- A `DataSource` is a factory for `Connection` objects.
  1. **Configure/Bind**: A `DataSource` is configured by a system administrator in an application server, or in a framework.
  2. **Lookup**: The `DataSource` object is retrieved from a naming service like JNDI (Java Naming and Directory Interface).
  3. **Get a Connection**: `dataSource.getConnection()` ✔
- Advantages of using `DataSource` objects:
  - Decouples configuration from code.
  - Connection pooling, Distributed transactions. 💡
  - Standard API for Java EE / Spring.

The Connection Pool Pattern

- **Connection Pool** is a creational pattern designed at **reducing the overhead** involved in performing database connections.



- Instead of opening a new connection for each request, the application requests a connection from the pool.
- Common configurations: initial and minimum pool size (e.g., 8 connections), maximum pool size (32), pool resize quantity (2), idle timeout (300 seconds), max wait time (60000 ms).

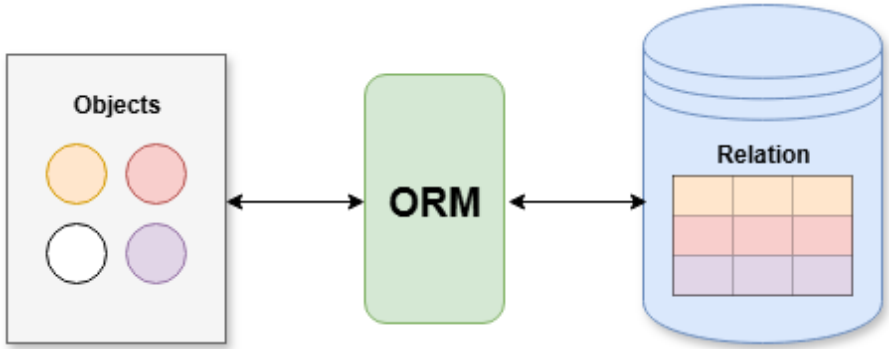
Example: Configuring a DataSource

HikariCP is a library offering a high-performance JDBC connection pooling mechanism. Used by default in Spring applications.

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:postgres://localhost/mydb");
config.setUsername("user");
config.setPassword("password");
// Fine tuning of the connection pool
config.setMaximumPoolSize(32);      // Max concurrent connections
config.setConnectionTimeout(30000); // Wait max 30s for a connection

//Create the DataSource object (connection pooling implementation)
DataSource dataSource = new HikariDataSource(config);
```

```
Connection conn = dataSource.getConnection(); //taken from the pool
con.close(); //returned to the pool
dataSource.close(); //closes the physical connections
```

Object-Relational Mapping (ORM)



- **Advantages**
  - Simplified development using automated conversions between objects and tables. No more SQL in the Java code.
  - Less code compared to embedded SQL and stored procedures.
  - Superior performance if object caching is used properly.
  - Applications are easier to maintain
- **Disadvantages**
  - The additional layer may slow execution sometimes
  - Defining the mapping may be difficult sometimes

Impedance Mismatch

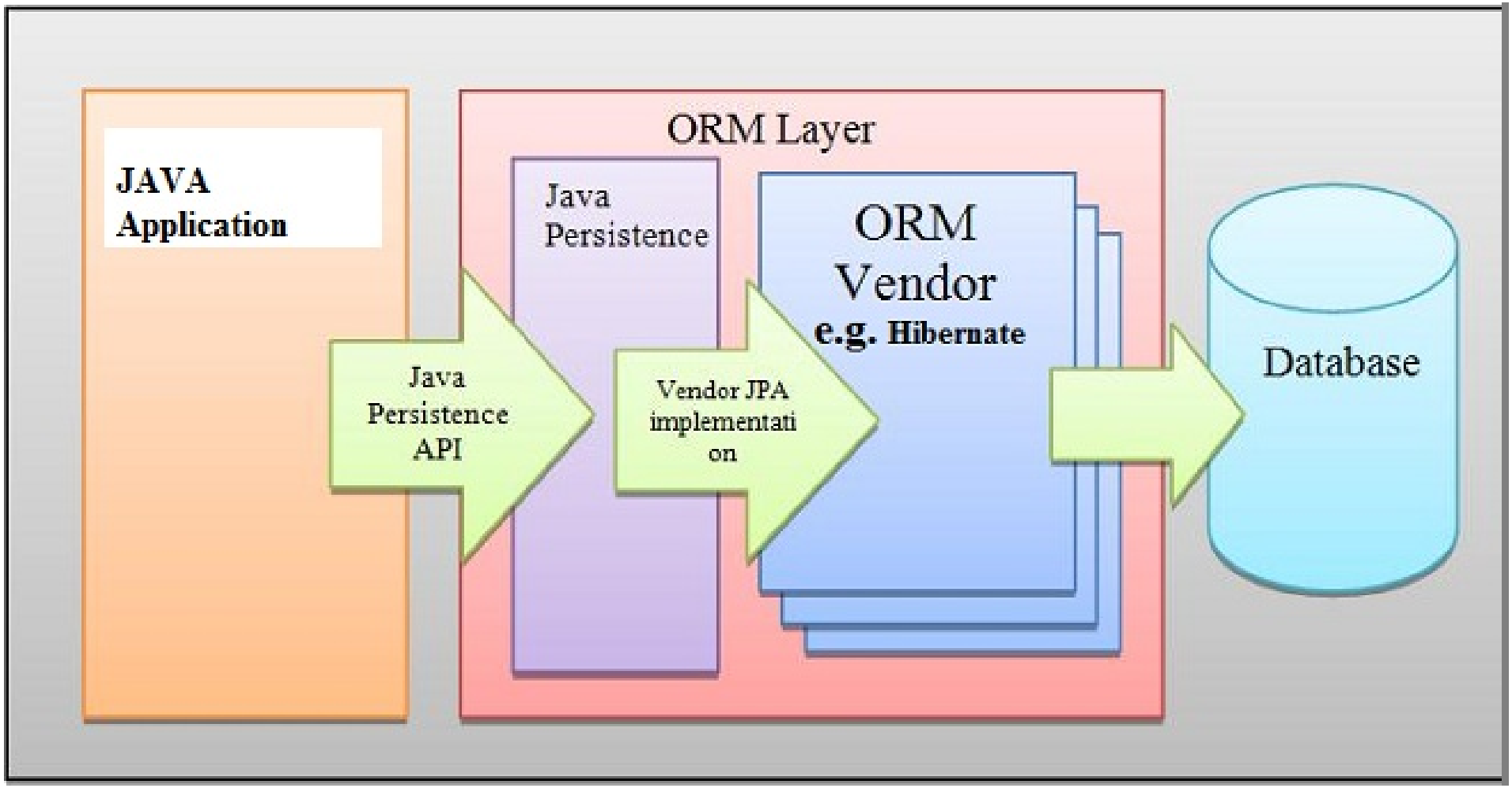
Graph of objects vs. Relations (sets of tuples)

- **What it means:** Difficulty mapping
  - object-oriented models
    $\rightarrow$ classes, inheritance, associations
  - to relational models
    $\rightarrow$ tables, foreign keys, joins.
- **Common Issues:**
  - Identity: `equals` vs. primary keys
  - Associations: references vs. foreign keys
  - Navigation: graph traversal vs. SQL queries
  - Inheritance: no direct relational equivalent
  - Granularity: the number of classes vs. tables
- "Impedance mismatch" is a conceptual gap that may lead to complexity and bugs if not handled properly.

Java Persistence API (JPA)

- A **standard**, high-level approach to object-relational mapping.
- It is defined by Java/Jakarta EE specifications.
  → specification, not implementation ⚠
- Consists of:
  - O/R **mapping metadata** (annotations).
  - Programming interface for managing **entities**.
  - A string-based **object-oriented query languages** (JPQL).
  - A **type-safe**, programmatic query building (Criteria API).
- Implemented by most of the Java ORM producers:
  - **EclipseLink**: The reference implementation of JPA.
    → GlassFish, Payara, WebLogic, WebSphere
  - **Hibernate**: Used by Spring Data JPA under the hood.
    → WildFly, JBoss EAP
  - **Apache OpenJPA**: TomEE (Tomcat + EE)

Which Implementation Should I Use?



Entities

- A **JPA entity** is a lightweight persistence domain object.
  - An entity class represents a table.
  - An entity instance corresponds to a row in that table.
- The mapping is defined using **persistence annotations**, using a "convention over configuration" approach.

```
@Entity
@Table(name = "persons") // ← may be omitted
public class Person {
    @Id
    private Integer id;
    private String name;
    // Constructors, getters, setters
}
```

- Entity classes are **POJOs** (regular Java classes).
- It must have a public or protected **no-arg constructor**.
- Every entity must have **a primary key**.

Persistence Annotations

```
@Entity
@Table(name = "PERSONS")
public class Person implements Serializable {

    @Id
    @SequenceGenerator(name = "sequence",
                       sequenceName = "persons_id_seq")
    @GeneratedValue(generator = "sequence")
    @Column(name = "PERSON_ID")
    private Integer id;

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

    @JoinColumn(name = "DEPT_ID")
    @ManyToOne
    private Department department;

    //...
} // In some contexts entity classes are required to be serializable
```

Persistence Units & Contexts

- A **persistence unit** defines the set of all entity classes that are managed by an application.
  - Defined at design time in `persistence.xml` 
  - `javax.persistence.PersistenceUnit`
  - This set of entity classes represents the data contained within a single data store.
  - An application may use multiple persistence units.

Persistence Units & Contexts

- A **persistence unit** defines the set of all entity classes that are managed by an application.
  - Defined at design time in `persistence.xml` 
  - `javax.persistence.PersistenceUnit`
  - This set of entity classes represents the data contained within a single data store.
  - An application may use multiple persistence units.

- A **persistence context** defines a set of entity instances managed at runtime by an **EntityManager**
  - It acts as a **first-level cache**, tracking changes to entities and synchronizing them with the database.
  - Within a persistence context, each database row is represented by only one object instance.
  - `javax.persistence.PersistenceContext`

persistence.xml (Local)

```
<persistence>
  <persistence-unit name="MyLocalPU"
                    transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <class>myapp.entity.Student</class>
    <class>myapp.entity.Project</class>

    <properties>
      <property name="javax.persistence.jdbc.driver"
                value="org.postgresql.Driver"/>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:postgresql://localhost/demo"/>
      <property name="javax.persistence.jdbc.user"
                value="username"/>
      <property name="javax.persistence.jdbc.password"
                value="secret"/>
    </properties>
  </persistence-unit>
</persistence>
```

persistence.xml (EE)

```
<?xml version="1.0" encoding="UTF-8"?>
  <persistence ...>
    <persistence-unit name="MyAppPU" transaction-type="JTA">

      <jta-data-source> jdbc/demo </jta-data-source>

      <exclude-unlisted-classes> false </exclude-unlisted-classes>

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

JTA (Java Transaction API)

- Allows applications to perform distributed transactions.
- Supports XA transactions (two-phase commit)
- Used mainly in Java EE application servers.
- Not usually needed in single-database Spring apps.

EntityManager

- The **interface** between the app and the persistence context.
- It handles CRUD, queries, and manages entities' lifecycle.
  → `persist, remove, find, refresh, createQuery`
- Provided by an `EntityManagerFactory`

```
// Plain JPA (Local)
EntityManagerFactory emf =
  Persistence.createEntityManagerFactory("myPU");
EntityManager em = emf.createEntityManager();
```

- In Java EE or Spring, an EM is injected when needed.

```
    @PersistenceContext // or @Autowired, or @Inject
    private EntityManager em;
```

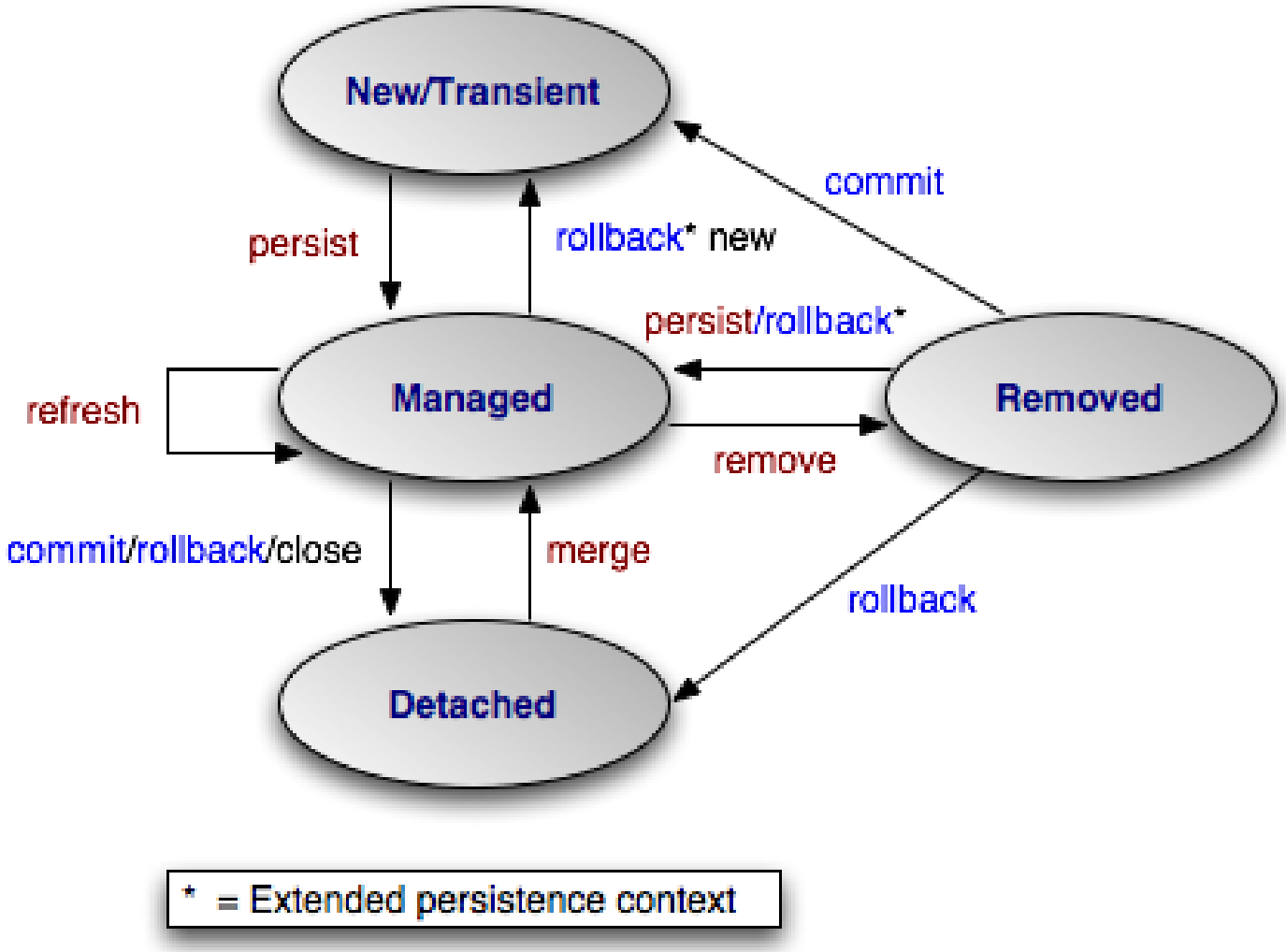
- An EMF is an expensive, thread-safe object.
- An EM is not expensive and not thread-safe.

Example: Using an EntityManager (Plain JPA)

```
em.getTransaction().begin();

Person joe = new Person("Joe");
System.out.println("ID:" + joe.getId()); //--> null
System.out.println("Managed:" + em.contains(joe)); //--> false

em.persist(joe); // INSERT
System.out.println("ID:" + joe.getId()); //--> 200
System.out.println("Managed:" + em.contains(joe)); //--> true

joe.setName("Joseph"); // UPDATE

int donaldId = 100;
Person donald = em.find(Person.class, donaldId); // SELECT
em.remove(donald); // DELETE

em.getTransaction().commit();
```

```
// Executing an object-oriented query
String jpql = "SELECT p FROM Person p";
List<Person> list = em.createQuery(jpql, Person.class).getResultList();
```

The Lifecycle of an Entity



JPA Lifecycle Callbacks

- Methods invoked automatically by the persistence provider during **specific events in an entity's lifecycle**.
- @PostLoad, @PrePersist, @PostPersist, @PreUpdate, @PostUpdate @PreRemove, @PostRemove
- Useful for auditing, validation, transformation, lazy initialization.

```
public class AuditingListener {
    @PrePersist
    public void setCreationDate(Object entity) {
        if (entity instanceof Auditable) {
            ((Auditable) entity).setCreatedAt(new Date());
        }
    }
}
```

```
@Entity
@EntityListeners(AuditingListener.class)
public class Person { ... }
```

Mapping Associations

- `@OneToOne, @OneToMany, @ManyToOne, @ManyToMany`
- Each association can be **unidirectional** or **bidirectional**.
- A unidirectional relationship has only an **owning side**.
  $\rightarrow$ The owning side is the entity that defines the actual mapping and is **responsible for updating** the table.
- A bidirectional relationship has both an owning side and an **inverse side** (passive side, contains a `@mappedBy`).

```
// In Order class ← inverse side
@OneToMany(cascade = CascadeType.ALL, mappedBy = "order")
public Set<OrderItem> items;

// In OrderItem class ← owning side
@ManyToOne
@JoinColumn( name = "order_id", referencedColumnName = "id")
public Order order;
```

- The many side of a many-to-one bidirectional relationship side is always the owning side.

Example: One-to-One

For each order, an invoice is created.

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

    @OneToOne(mappedBy = "order") // inverse side
    private Invoice invoice;
    //other properties, getters and setters
}
```

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

    @OneToOne
    @JoinColumn(name = "order_id") // owning side
    private Order order;
    // other properties, getters and setters
}
```

Example: One-to-Many, Many-To-One

Each order contains a set of items.

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

    @OneToMany(mappedBy = "order",
               cascade = CascadeType.ALL, orphanRemoval = true)
    private List<OrderItem> items = new ArrayList<>(); //...
}
```

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

    @ManyToOne
    @JoinColumn(name = "order_id") // owning side
    private Order order; //...
}
```

Example: Many-To-Many

Each supplier offers a set of products, each product may be offered by different suppliers.

```
// In Supplier class
    @ManyToMany
    @JoinTable(
        name = "suppliers_products",
        joinColumns = @JoinColumn(name = "supplier_id"),
        inverseJoinColumns = @JoinColumn(name = "product_id")
    )
    private List<Product> products = new ArrayList<>();

    // The table suppliers_products is also called a junction table.
```

```
// In Product class
    @ManyToMany(mappedBy = "products")
    private List<Supplier> suppliers = new ArrayList<>();
```

Mapping Associative Tables

- *A person may have different roles in various departments.*

```
persons_departments (person_id, department_id, role_id)
```

```
// In Person class
    List<Department> departments; // no role... ✗
    List<PersonDepartment> departments; ✓
```

```
// In Department class
    List<Person> persons; // no role... ✗
    List<PersonDepartment> persons; ✓
```

- A separate entity `PersonDepartment` is required to map the associative table. It must define a **composite primary key** described by a separate `@Embeddable` class.

```
    @EmbeddedId
    private PersonDepartmentId id;
```

The Associative Entity

```
@Entity
@Table(name = "persons_departments")
public class PersonDepartment {
    @EmbeddedId
    private PersonDepartmentId id;

    @ManyToOne
    @MapsId("personId")
    @JoinColumn(name = "person_id")
    private Person person;

    @ManyToOne
    @MapsId("departmentId")
    @JoinColumn(name = "department_id")
    private Department department;

    @ManyToOne
    @MapsId("roleId")
    @JoinColumn(name = "role_id")
    private Role role; //...
}
```

Mapping the Composite PK

```
@Embeddable
public class PersonDepartmentId {

    @Column(name = "person_id")
    private Integer personId;

    @Column(name = "department_id")
    private Integer departmentId;

    @Column(name = "role_id")
    private int roleId;

    //...
}
```

`@Embeddable` specifies a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity.

Mapping Inheritance

- Entities support **class inheritance**, polymorphic associations, and polymorphic queries.
- Entity classes can extend non-entity classes, and non-entity classes can extend entity classes.
- Entity classes can be **abstract**.
  `@MappedSuperclass` defines an abstract not-entity parent class.
- **Inheritance strategies** using `@Inheritance` annotation:
  - **`SINGLE_TABLE`**: One table for the entire class hierarchy (default). A discriminator column is used to distinguish which subclass each row represents.
  - **`JOINED`**: Each entity class, abstract or concrete, is mapped to its own table – common fields are not duplicated.
  - **`TABLE_PER_CLASS`** – each concrete class is mapped to its own table, – common fields are duplicated in each table.

Example: @MappedSuperclass

```java
@Entity
@MappedSuperclass
public class Person {
    @Id
    private int id;

    private String name; // ...
} // Provides common mappings for fields to its subclasses.
```

```java
@Entity
@Table(name = "STUDENTS")
public class Student extends Person {
    private int yearOfStudy; // ...
}
```

```java
@Entity
@Table(name = "LECTURERS")
public class Lecturer extends Person {
    private String position; // ...
}
```

Example: SINGLE_TABLE

```
@Entity
@Table(name = "PERSONS")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="person_type",
                     discriminatorType = DiscriminatorType.CHAR)
public class Person {
    @Id
    private int id;

    private String name; // ...
}
```

```
@Entity
@DiscriminatorValue("S")
public class Student extends Person { ... }
```

```
@Entity
@DiscriminatorValue("L")
public class Lecturer extends Person { ... }
```

Synchronizing Entity Data to the Database

- The state of entities is synchronized to the database when the transaction with which the entity is associated **commits**. Closing the EM without `commit` → `rollback`. ⚠
- If a managed entity is in a bidirectional relationship with another managed entity, the data will be persisted, **based on the owning side** of the relationship.
- To force synchronization of the managed entity to the data store, invoke **`flush()`**. If the entity is removed, calling flush will remove the entity data from the data store.
- Use **`refresh()`** to recreate the state of the instance from the database, overwriting changes made to the entity in the current transaction. Useful in case of **stale data**.

Java Persistence Query Language (JPQL)

- String-based, SQL-like, portable, **object-oriented queries** for entities and their persistent state.

```
SELECT p FROM Person p WHERE p.name LIKE '%Duke%'
```

- **Static (Named) Queries** are prefered over dynamic ones.

```
@NamedQuery(name="Person.findByName", // Static query ✔
    query="SELECT p FROM Person p WHERE p.name LIKE :name")
) // Parsed and compiled early; syntax errors are "detectable"
@Entity
@Table(name="persons")
class Person { ... }
```

```
List<Person> list = em.createNamedQuery("Person.findByName")
    .setParameter("name", name)
    .getResultList();
```

```
Query query = entityManager.createQuery( // Dynamic query ✘
    "SELECT p FROM Person p WHERE p.name LIKE '" + name + "'");
```

JPA Criteria API

- Portable **type-safe** queries for entities.
- Using the **Metamodel API** is **type-and-name** safe.
- Define queries in an **object-oriented** manner, via construction of a `CriteriaQuery` instance, rather than string-based.

```
public List<User> getAdultUsers() {
  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<Person> cq = cb.createQuery(Person.class);

  Root<Person> person = cq.from(Person.class);
  Predicate ageOver18 = cb.greaterThan(person.get("age"), 18);
  cq.select(person).where(ageOver18);

  return em.createQuery(cq).getResultList();
}
```

- Criteria API is the recommended solution for creating **dynamic queries** that dependend on runtime values.

JPA Performance Tunning

- **Fetching Strategies**
  - Lazy Loading (`FetchType.LAZY`) – Default for collections
  - Eager Loading (`FetchType.EAGER`)
- **The N+1 Query Problem** happens when fetching a parent entity causes N additional queries for its children.
- **Caching**
  - First-level cache (mandatory): EntityManager-level.
  - Second-level cache (optional): `@Cacheable` + Ehcache.
  - Query cache: Good for stable lookups.
- **Query Optimization**
  - DTO Projections: Fetch only the fields you need.
  - Pagination: Do not load in memory large result sets.
  - Read-Only Transactions: Allow for internal optimizations.
- **Logging and Monitoring**
  - SQL logging & Connection pool monitors

Spring Data JPA

- **Abstraction over JPA**.
- **Less boilerplate**: Just write interfaces.
- Repositories with **built-in CRUD**.
- **Derived query methods**.
- Spring Boot integration: `spring-boot-starter-data-jpa`
- Support for **advanced queries**: JPQL, Criteria, Specifications.

```java
public interface UserRepository extends JpaRepository<User, Long> {
    // Derived query
    Optional<User> findByEmail(String email);

    @Query("SELECT u FROM User u WHERE u.name LIKE %:name%") //JPQL
    List<User> findByName(@Param("name") String name);
}
```

```java
    @Autowired
    UserRepository userRepo; // ← Generates an implementation
```

Configuring a DataSource

- Single static configuration

```
# application.properties
spring.datasource.url=jdbc:postgresql://localhost/mydb
spring.datasource.username=postgres
spring.datasource.password=secret

spring.datasource.hikari.auto-commit=false
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.minimum-idle=5

spring.jpa.hibernate.ddl-auto=none
# options: create, create-drop, update, validate, none
spring.jpa.show-sql=true
```

- Multiple static configurations.
- Dynamic routing with runtime selection.
- Fully dynamic with unknown-at-startup tenants. 

JpaRepository

- `JpaRepository`
  extends `PagingAndSortingRepository`
  extends `CrudRepository`.
- Provides support for **CRUD operations**

```
save(), findById(), delete(), findAll()
```

- Provides support for **pagination & sorting**

```
findAll(Pageable pageable)
findAll(Sort sort)
```

- **Batch** operations, **Flush** & Entity management

```
saveAll(), deleteAllInBatch()
flush(), saveAndFlush()
```

- Spring Data JPA automatically wraps repository methods in transactions (read-only for queries by default).

Derived Queries

- Derived queries are methods in a repository interface whose **names encode the query automatically**.

```
  List<Person> findByName(String name);
  // Spring generates: SELECT p FROM persons p WHERE p.name = ?1
}
```

- Can be **combined**, define **range and comparison**, **string matching**, **sorting** and **limits**.

```
findByNameAndAge(String name, int age);
findByNameOrEmail(String name, String email);
findByAgeGreaterThan(int age);
findByEnrollmentDateBetween(LocalDate start, LocalDate end);
findByNameStartingWith(String prefix);
findByNameContainingIgnoreCase(String keyword);
findByAgeGreaterThanOrderByNameAsc(int age);
findTop3ByOrderByAgeAsc();
findTopByActiveTrueAndAgeGreaterThanAndScoreGreaterThanEqual...😄
```

- If the method name is wrong → application startup exception.

Calling Stored Procedures

- A stored procedure is a precompiled set of SQL statements that are stored in the database. Used to encapsulate complex business logic, performance, data security.
- Procedures can be declared in entity classes.

```
@Entity
@NamedStoredProcedureQuery(
    name = "Employee.calculateBonus",
    procedureName = "calculate_bonus",
    parameters = { ... } )
public class Employee { ...}
// JPA-standard, reusable, strong typing, centralized on entity.
```

- Stored procedures in a Spring `JpaRepository`.

```
public interface EmployeeRepository
        extends JpaRepository<Employee, Long> {

    @Procedure(name = "calculate_bonus")
    int calculateBonus(@Param("userId") Long employeeId);
} // Convenience, less verbose
```

Custom Repositories

We need a custom method in a JPA repository.

```
public interface CustomPersonRepository {
    void updateAge(Long id, int newAge);
}
```

```
@Repository
public interface PersonRepository
    extends JpaRepository<Person, Long>, CustomPersonRepository
```

```
@Repository
public class CustomRepoImpl implements CustomPersonRepository {
  @PersistenceContext private EntityManager em;

  @Override @Transactional
  public void updateAge(Long id, int newAge) {
    String stmt = "UPDATE Person p SET p.age = :age WHERE p.id = :id";
    em.createQuery(stmt).setParameter("age", newAge)
                        .setParameter("id", id).executeUpdate();
  }
}
```

Custom Repositories - Simple Solution

If your custom method is just an update/delete, you don't need a separate implementation class.

```
public interface PersonRepository
        extends JpaRepository<Person, Long> {

    @Modifying      // ← tells Spring this is not a SELECT
    @Transactional  // ← required for UPDATE / DELETE
    @Query("UPDATE Person p SET p.age = :age WHERE p.id = :id")
    void updateAge(Long id, int age);

    @Modifying
    @Transactional
    @Query("DELETE FROM Person p WHERE p.email = :email")
    int deleteByEmailAddress(@Param("email") String email);
}
```

What `@Transactional` Does

When you annotate a method/class with `@Transactional`, Spring:

1. Creates a **proxy** around your bean
   → using JDK dynamic proxies or CGLIB. 💡
2. **Intercepts** method calls.
3. Before method execution → starts or joins a transaction.
4. Executes your code using a JPA `EntityManager`.
5. On success → commit.
6. On exception → rollback (by default on runtime exceptions).

No need to call `commit()` or `rollback()` — the proxy does it.
Only public methods should be annotated with `@Transactional`.
Does not work for private or self-invoked method. ⚠
Transaction management is configurable:

```
@Transactional(rollbackFor = Exception.class)
@Transactional(readOnly = true)
```

Spring Transaction Propagation

```
@Transactional(propagation = Propagation.REQUIRED)
public void methodA() {
    otherBean.methodB();
}
```

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

Transaction Propagation Types:

- **REQUIRED**
- REQUIRES_NEW
- MANDATORY
- SUPPORTS
- NOT_SUPPORTED
- NEVER
- NESTED

Pagination and Sorting

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepo;

    public Page<User> getUsers
            (int page, int size, String sortBy) {

        Sort sort = Sort.by(sortBy).ascending();
        return userRepo.findAll(PageRequest.of(page, size, sort));
    }
}
```

Spring Data JPA translates this into a JPQL query.

- It uses query methods `setFirstResult, setMaxResults`.
- The generated SQL uses `OFFSET, LIMIT`.
- A **page** is a sublist of a list of objects, containing also information about its number, size, total elements, total pages.

Specifications (Dynamic Queries)

- Build dynamic, type-safe queries using **Criteria API**, easier.
- A **Specification** is a predicate (condition) that can be combined dynamically.

```
public interface UserRepository
  extends JpaRepository<User,Long>, JpaSpecificationExecutor<User> {}
```

```
public class UserSpecifications {
    public static Specification<User> ageGreaterThan(int age) {
        return (root, query, criteriaBuilder) ->
            criteriaBuilder.greaterThan(root.get("age"), age);
    } // and other specifications ...
}
```

```
  Specification<User> spec = Specification.where(null);
  if (minAge != null) {
    spec = spec.and(UserSpecifications.ageGreaterThan(minAge));
  } // and other specifications ...
  return userRepository.findAll(spec);
```

Auditing

- **Tracking changes** in your database.
- Automatically populates fields like `createdDate`, `lastModifiedDate`, `createdBy`, `lastModifiedBy`.
  1. **Enable auditing**: `@EnableJpaAuditing`
  2. Create an **auditable base class**, with annotations for dates and users. Auditable entity classes will extend from it.
  3. Provide an `AuditorAware` bean, to get the actual user.

```
@MappedSuperclass
@EntityListeners(AuditingEntityListener.class)
public abstract class Auditable {
    @CreatedDate
    @Column(updatable = false)
    private LocalDateTime createdDate;
    //...
}
```

```
@Entity
public class User extends Auditable { ... }
```