JPA and Spring Data JPA — Full Teacher-Style Notes with Code Examples

Overview

JPA (Java Persistence API) is a specification for mapping Java objects to relational database tables (ORM). Spring Data JPA builds on JPA to eliminate boilerplate by providing repository abstractions, derived queries, pagination, sorting, and integration with Spring Boot. Spring Data JPA focuses on a consistent programming model and repository support on top of a JPA provider like Hibernate.[1]

Core JPA Building Blocks

1) Entity and Table Mapping

- Use @Entity to mark a class as a persistent entity; optionally use @Table to customize the table name or schema.[2] [3]

```
import jakarta.persistence.*;

@Entity
@Table(name = "books", schema = "library")
public class Book {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(name = "title", nullable = false, length = 200)
  private String title;

  @Column(name = "isbn", unique = true, length = 20)
  private String isbn;

  // getters/setters
}
```

- @Id marks the primary key; @GeneratedValue selects ID generation strategy (AUTO, IDENTITY, SEQUENCE, TABLE) depending on DB and provider.[2] [3]

2) Column and Field Mapping

- @Column customizes column definition: name, length, nullable, unique, precision/scale.[3] [2]
- @Lob marks large text/binary fields (CLOB/BLOB).[2]
- @Transient excludes a field from persistence; useful for computed or non-persistent values. [2]

```
@Lob
@Column(name = "description")
private String description;

@Transient
private boolean highlighted;
```

3) Enumerations and Temporal Types

- @Enumerated(EnumType.STRING) or ORDINAL maps enums; STRING is safer for evolvable enums.[2]
- For legacy Date/Calendar, @Temporal(TemporalType.DATE/TIME/TIMESTAMP) clarifies precision (not needed for java.time types).[2]

```
public enum BookStatus { DRAFT, PUBLISHED }

@Enumerated(EnumType.STRING)
@Column(nullable = false)
private BookStatus status = BookStatus.DRAFT;
```

4) Embeddables and Embedded Types

- Use @Embeddable for value types and @Embedded to include them inside entities.[2]

```
@Embeddable
public class AuditInfo {
  private String createdBy;
  private Instant createdAt;
}

@Embedded
private AuditInfo audit;
```

5) Relationships

- @ManyToOne for many-to-one; the owning side holds the FK and often uses @JoinColumn. [3] [2]
- @OneToMany for one-to-many; typically mappedBy to avoid join table; consider fetch type and cascade.[2]
- @OneToOne for one-to-one; use @JoinColumn or mappedBy depending on ownership.[2]

- @ManyToMany for many-to-many; use @JoinTable to configure the link table; consider replacing with two one-to-many for richer models.[2]

```
@Entity
public class Author {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(nullable = false)
  private String name;

  @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval = true)
  private List<Book> books = new ArrayList<>();
}

@Entity
public class Book {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToOne(optional = false, fetch = FetchType.LAZY)
  @JoinColumn(name = "author_id", nullable = false)
  private Author author;

  @ManyToMany
  @JoinTable(
    name = "book_category",
    joinColumns = @JoinColumn(name = "book_id"),
    inverseJoinColumns = @JoinColumn(name = "category_id")
  )
  private Set<Category> categories = new HashSet<>();
}
```

- Ordering collections:
  - @OrderBy for JPQL-based ordering on fetch.[4] [2]
  - @OrderColumn to persist the order index in a column.[4] [2]

```
@OneToMany(mappedBy = "author")
@OrderBy("title ASC")
private List<Book> books;
```

6) Lifecycle and Listeners

- Register listeners with @EntityListeners; use callbacks like @PostLoad, @PrePersist, @PreUpdate, @PreRemove for auditing or validation tasks.[4] [2]

```
@EntityListeners(AuditListener.class)
@Entity
public class Book { ... }

public class AuditListener {
  @PrePersist
  public void beforeSave(Object entity) { /* audit */ }

  @PostLoad
  public void afterLoad(Object entity) { /* post-load hook */ }
}
```

7) Persistence Context and EntityManager (classic JPA)

- @PersistenceContext injects EntityManager in traditional JPA/Spring apps (less common when using Spring Data repositories).[5] [4] [2]

```
@Stateless
public class BookDao {
  @PersistenceContext
  private EntityManager em;

  public Book find(Long id) {
    return em.find(Book.class, id);
  }
}
```

Spring Data JPA Essentials

Spring Data JPA provides repository abstractions and query generation. It is typically enabled automatically in Spring Boot; otherwise, use @EnableJpaRepositories to scan repository packages.[1]

1) Repository Hierarchy and When to Use

- CrudRepository: basic CRUD use cases.[1]
- PagingAndSortingRepository: CRUD plus paging/sorting.[1]
- JpaRepository: adds JPA-specific methods (flush, batch, Example API); most common choice.[1]

```
public interface BookRepository extends JpaRepository<Book, Long> {
}
```

2) Derived Query Methods

Define methods by following naming conventions: findBy, readBy, getBy, existsBy, countBy, deleteBy, and combine with property names, comparisons, and keywords (And, Or, Between, LessThan, Like, In, OrderBy, etc.).[6] [1]

```
List<Book> findByTitle(String title);
List<Book> findByAuthorNameAndStatus(String authorName, BookStatus status);
Page<Book> findByTitleContainingIgnoreCase(String part, Pageable pageable);
```

```
boolean existsByIsbn(String isbn);
long countByStatus(BookStatus status);
```

3) @Query for Custom JPQL or Native SQL

- Use @Query for custom SELECT statements; @Param binds named parameters.[6] [1]

```
public interface BookRepository extends JpaRepository<Book, Long> {

  @Query("select b from Book b where b.author.name = :name and b.status = :status")
  List<Book> findByAuthorNameAndStatus(@Param("name") String name,
                                       @Param("status") BookStatus status);

  @Query(value = "select * from books b where b.isbn = :isbn", nativeQuery = true)
  Optional<Book> findByIsbnNative(@Param("isbn") String isbn);
}
```

- Spring Data also offers a composed @NativeQuery annotation in newer docs as a variant for native queries; typical practice remains @Query(nativeQuery=true).[6]

4) @Modifying for UPDATE/DELETE/INSERT/DDL

- Required for non-select queries alongside @Query; can return update counts; often combined with @Transactional at method or class level.[7] [8]

```
@Transactional
@Modifying
@Query("update Book b set b.status = :status where b.id = :id")
int updateStatus(@Param("id") Long id, @Param("status") BookStatus status);

@Transactional
@Modifying
@Query("delete from Book b where b.status = 'DRAFT'")
int deleteAllDrafts();

@Transactional
@Modifying
@Query(value = "alter table books add column archived boolean default false", nativeQuery
void addArchivedColumn();
```

- Persistence context may become stale after bulk operations; consider clearAutomatically or manual EntityManager clear/flush if needed.[7]

5) Pagination and Sorting

- Use Pageable and Sort with derived or @Query methods; repositories support Page<T> and Slice<T>.[1] [6]

```
Page<Book> findByStatus(BookStatus status, Pageable pageable);
```

```
// usage
Page<Book> page = repo.findByStatus(BookStatus.PUBLISHED, PageRequest.of(0, 20, Sort.by('
```

6) Transactions

- Repositories are typically transactional by default for write operations; annotate service methods with @Transactional for custom boundaries or readOnly optimizations.[1]

```
@Service
public class PublishingService {
  private final BookRepository repo;

  public PublishingService(BookRepository repo) { this.repo = repo; }

  @Transactional
  public void publish(Long id) {
    repo.updateStatus(id, BookStatus.PUBLISHED);
  }

  @Transactional(readOnly = true)
  public Page<Book> listPublished(Pageable pageable) {
    return repo.findByStatus(BookStatus.PUBLISHED, pageable);
  }
}
```

7) Enabling/Configuring Repositories

- In non-Boot apps: @EnableJpaRepositories(basePackages = "com.example.repo").[1]

```
@Configuration
@EnableJpaRepositories(basePackages = "com.example.repo")
public class JpaConfig {
  // DataSource, EntityManagerFactory, PlatformTransactionManager beans
}
```

When to Use Which Annotation — Practical Guidance

- @Entity: Every class persisted to a table.[3] [2]
- @Table: When table name/schema differs or need unique constraints at table level.[3] [2]
- @Id/@GeneratedValue: Always for PK; choose strategy based on DB (IDENTITY for MySQL, SEQUENCE for Postgres/Oracle with @SequenceGenerator).[3] [2]
- @Column: When you must rename a column, enforce length/precision, nullability, or uniqueness.[3] [2]
- Relationship annotations:
  - @ManyToOne on child with @JoinColumn for FK; default fetch is EAGER in JPA but consider LAZY for performance.[5] [2]

    - @OneToMany mappedBy on parent; prefer Set/List depending on need; consider orphanRemoval and cascade for aggregates.[2]
    - @OneToOne when true one-to-one; choose the owning side carefully (side holding FK).[2]
    - @ManyToMany with @JoinTable; consider replacing with explicit link entity for attributes (since many-to-many without attributes is often too limiting).[2]
- @Enumerated(EnumType.STRING): Prefer STRING for stability across enum changes.[2]
- @Lob: For large text/binary fields.[2]
- @Transient: For derived fields that should not persist.[2]
- @Embeddable/@Embedded: For reusable value objects (Address, Money, AuditInfo).[2]
- @EntityListeners and lifecycle callbacks: For auditing, timestamps, validation hooks without polluting business logic.[4] [2]
- Spring Data:
    - Extend JpaRepository by default for rich operations.[9] [1]
    - Use derived queries first; switch to @Query when expressions become too complex or when using joins/aggregations.[6] [1]
    - Use @Modifying for bulk updates/deletes or DDL; ensure proper transaction handling and persistence context clearing.[8] [7]
    - Use Pageable/Sort for paging/sorting at the database level, not in-memory.[6] [1]

End-to-End Example (Entity + Repository + Service)

```
// Domain
@Entity
@Table(name = "books")
public class Book {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Column(nullable = false, length = 200)
  private String title;

  @Column(unique = true, length = 20)
  private String isbn;

  @Enumerated(EnumType.STRING)
  @Column(nullable = false)
  private BookStatus status = BookStatus.DRAFT;

  @ManyToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "author_id", nullable = false)
  private Author author;

  @Lob
  private String description;
  @Embedded
  private AuditInfo audit;
}

// Repository
public interface BookRepository extends JpaRepository<Book, Long> {

  // Derived queries
  Optional<Book> findByIsbn(String isbn);
  List<Book> findByTitleContainingIgnoreCase(String part);
  Page<Book> findByStatus(BookStatus status, Pageable pageable);

  // Custom JPQL
  @Query("""
    select b from Book b
    join b.author a
    where a.name = :authorName and b.status = :status
  """)
  List<Book> findByAuthorAndStatus(@Param("authorName") String authorName,
                                   @Param("status") BookStatus status);

  // Bulk update
  @Transactional
  @Modifying
  @Query("update Book b set b.status = :newStatus where b.status = :oldStatus")
  int bulkUpdateStatus(@Param("oldStatus") BookStatus oldStatus,
                       @Param("newStatus") BookStatus newStatus);
}

// Service
@Service
public class BookService {
  private final BookRepository repo;

  public BookService(BookRepository repo) { this.repo = repo; }

  @Transactional(readOnly = true)
  public Page<Book> listPublished(int page, int size) {
    return repo.findByStatus(BookStatus.PUBLISHED, PageRequest.of(page, size, Sort.by("ti
  }

  @Transactional
  public int publishAllDrafts() {
    return repo.bulkUpdateStatus(BookStatus.DRAFT, BookStatus.PUBLISHED);
  }
}
```

Additional Notes and Best Practices

- Keep collections LAZY where possible to avoid N+1 issues; use fetch joins in @Query when needed.[6] [1]
- For bulk operations via @Modifying, consider clearAutomatically=true or manual EntityManager clear to avoid stale entities in the persistence context.[7]
- Prefer value objects with @Embeddable to model concepts like money, addresses, or audit metadata cleanly.[2]
- Use @OrderBy on collections when you always need a stable order at fetch time; for position-aware lists, @OrderColumn persists an index.[4] [2]
- Repositories can expose projections (interfaces/DTOs) in Spring Data JPA to fetch partial data via derived queries or @Query.[6] [1]

Quick Reference: Common Annotations and Use

- JPA core: @Entity, @Table, @Id, @GeneratedValue, @Column, @Lob, @Transient, @Enumerated, @Embeddable, @Embedded, @ManyToOne, @OneToMany, @OneToOne, @ManyToMany, @JoinColumn, @JoinTable, @OrderBy, @OrderColumn, @EntityListeners, @PrePersist, @PostLoad, @PersistenceContext.[5] [4] [3] [2]
- Spring Data JPA: @EnableJpaRepositories, @Repository (optional for interfaces), @Query, @Param, @Modifying, @Transactional; repository interfaces: CrudRepository, PagingAndSortingRepository, JpaRepository.[9] [7] [1] [6]

If specific domain cases are needed (soft deletes, auditing, DTO projections, specifications/criteria), detailed patterns can be added on request using the same building blocks above with Spring Data JPA's features such as Specifications and Query by Example.[1] [6]

⁂

1. https://docs.spring.io/spring-data/jpa/reference/index.html
2. https://dzone.com/articles/all-jpa-annotations-mapping-annotations
3. https://docs.jboss.org/hibernate/stable/annotations/reference/en/pdf/hibernate_reference.pdf
4. https://www.javaguides.net/2018/11/all-jpa-annotations-mapping-annotations.html
5. https://docs.redhat.com/en/documentation/red_hat_jboss_web_server/1.0/html/hibernate_annotations_reference_guide/entity-mapping
6. https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html
7. https://www.baeldung.com/spring-data-jpa-modifying-annotation
8. https://www.geeksforgeeks.org/advance-java/spring-data-jpa-modifying-annotation/
9. https://docs.spring.vmware.com/spring-data-jpa-distribution/docs/3.1.13/reference/html/index.html