Spring Data JPA Notes

Spring Boot with JPA & Hibernate – Core Notes

1. Introduction

- **JPA (Java Persistence API):** A specification for object-relational mapping (ORM) that allows developers to map Java objects to relational database tables.
- **Hibernate:** The most popular JPA implementation (provider). Handles SQL generation, object mapping, and database interactions.
- **Spring Data JPA:** A Spring module built on top of JPA, making repository/DAO layer development simple.

□ Together, **Spring Boot + Spring Data JPA + Hibernate** allows us to quickly build data-driven applications without writing boilerplate SQL.

---

2. Key Annotations

2.1 @Entity

- Declares a class as a **JPA entity (mapped to a database table)**.
- Each instance of the class = a row in the table.
- Must have:
  - A **no-args constructor** (public or protected).
  - A **primary key field** annotated with @Id.

□ **Example:**

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class Student {

    @Id
    private Long id;   // Primary key

    private String name;
    private String email;

    // Getters & setters
}
```

2.2 @Id

- Marks a field as the **primary key** of the entity.
- Required for every entity.
- Can be manually assigned or auto-generated using @GeneratedValue.

□ **Example:**

```
@Entity
public class Student {
    @Id
    private Long id;   // Manually assigned primary key
}
```

2.3 @GeneratedValue

- Used with @Id to **automatically generate primary key values**.
- Strategies:
    1. GenerationType.AUTO → Hibernate chooses best strategy (default).
    2. GenerationType.IDENTITY → Uses auto-increment column in DB.
    3. GenerationType.SEQUENCE → Uses database sequence (efficient for Oracle/Postgres).
    4. GenerationType.TABLE → Uses a table to generate IDs (rarely used).

□ **Example:**

```
@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;   // Auto-generated primary key

    private String name;
    private String email;
}
```

---

3. How It Works (Behind the Scenes)

1. Spring Boot app starts → Hibernate scans @Entity classes.
2. Hibernate maps them to DB tables (creates if spring.jpa.hibernate.ddl-auto=update).
3. @Id field becomes the **PRIMARY KEY** column.
4. @GeneratedValue strategy decides how IDs are generated.

---

4. Application Example

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/testdb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Entity Example

```
@Entity
public class Student {

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

    private String name;
    private String email;
}
```

Repository Example

```
import org.springframework.data.jpa.repository.JpaRepository;
```

```
public interface StudentRepository extends JpaRepository<Student, Long> {}
```

Service Example

```
@Service
public class StudentService {
   @Autowired
   private StudentRepository repo;

   public Student saveStudent(Student s) {
      return repo.save(s);
   }
}
```

---

5. Interview Insights

- **Q: What is the role of @Entity?**
  → Marks class as a JPA entity (mapped to DB table).
- **Q: Can an entity exist without @Id?**
  → No, every entity must have a primary key.
- **Q: Difference between GenerationType.IDENTITY vs SEQUENCE?**
  → IDENTITY uses auto-increment (good for MySQL).
  → SEQUENCE uses DB sequence objects (better performance in Oracle/Postgres).
- **Q: What if you don't specify @GeneratedValue?**
  → Then IDs must be manually assigned.

Spring Data JPA – Repositories

1. Introduction

In Spring Data JPA, **Repositories** are interfaces that handle all **data access operations**.
Instead of writing boilerplate DAO (Data Access Object) code, we just create an interface, and Spring generates the implementation automatically at runtime.

□ Benefits:

- No need to write SQL/JPQL for basic CRUD.
- Reduces boilerplate code.

- Supports custom queries with method names or @Query.

---

2. Repository Hierarchy (Simplified)

```
Repository (Marker Interface)
  └── CrudRepository
        └── PagingAndSortingRepository
              └── JpaRepository
```

---

3. CrudRepository

□ Definition

- Provides **CRUD (Create, Read, Update, Delete)** operations.
- Generic interface:

```
public interface CrudRepository<T, ID> extends Repository<T, ID> { ... }
```

- T → Entity type
- ID → Primary key type

□ Common Methods

Method	Description
save(S entity)	Save or update entity
findById(ID id)	Find entity by primary key
findAll()	Get all entities
deleteById(ID id)	Delete entity by primary key
deleteAll()	Delete all entities
count()	Get total records

□ Example

```
public interface StudentRepository extends CrudRepository<Student, Long> {}
```

Usage:

```
@Autowired
private StudentRepository repo;

public void demo() {
   // Save new student
   Student s = new Student();
   s.setName("John");
   repo.save(s);

   // Fetch all
   Iterable<Student> students = repo.findAll();

   // Find by ID
   Optional<Student> student = repo.findById(1L);

   // Delete
   repo.deleteById(1L);
}
```

---

4. JpaRepository

□ Definition

- Extends PagingAndSortingRepository (and indirectly CrudRepository).
- Provides **extra methods** for JPA-specific operations.
- Generic interface:

```
public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T,
ID> { ... }
```

□ Additional Features

- **Batch operations**
- **Flush and refresh**
- **Pagination and sorting** support
- Built-in methods for lists (instead of Iterable)

Common Methods

Method	Description
List<T> findAll()	Returns all entities as a List
List<T> findAll(Sort sort)	Returns all entities sorted
Page<T> findAll(Pageable pageable)	Returns entities in paginated form
void flush()	Flushes pending changes to DB
T saveAndFlush(T entity)	Save entity and immediately flush
void deleteInBatch(Iterable<T> entities)	Delete multiple entities in one query

□ Example

```
public interface StudentRepository extends JpaRepository<Student, Long> {}
```

Usage:

```
@Autowired
private StudentRepository repo;

public void demo() {
    // Save and flush
    Student s = new Student();
    s.setName("Alice");
    repo.saveAndFlush(s);

    // Pagination (page 0, size 2)
    Page<Student> page = repo.findAll(PageRequest.of(0, 2));

    // Sorting by name
    List<Student> sorted = repo.findAll(Sort.by("name"));
}
```

---

6. Interview Insights

- **Q: Why do we prefer JpaRepository over CrudRepository?**
  → Because it provides pagination, sorting, and JPA-specific methods.

- **Q: What happens if we don’t extend any Repository interface?**
  → We’d have to manually write DAO/EntityManager code for CRUD.
- **Q: Can we create custom queries in repositories?**
  → Yes, by using **method name convention** (e.g., findByName(String name)) or @Query.

Spring Data JPA – Important Annotations

1. @Table

- Defines the **table name** and details for the entity.
- Default → table name = class name (Student → student).

□ **Example:**

```
import jakarta.persistence.*;

@Entity
@Table(name = "students")   // Maps to table 'students'
public class Student {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private String name;
}
```

---

2. @Column

- Used to **customize column mapping**.
- Attributes:
  - name → column name
  - nullable → allows null or not
  - unique → enforces unique constraint
  - length → column length (for String)
  - updatable/insertable → include in SQL or not

□ **Example:**

```
@Entity
```

```
@Table(name = "students")
public class Student {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   @Column(name = "full_name", nullable = false, length = 100)
   private String name;

   @Column(unique = true)
   private String email;
}
```

---

3. Relationships

3.1 @OneToOne

- Defines **one-to-one relationship** between two entities.
- Example: One Student → One Address.

□ Example:

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

   private String name;

   @OneToOne(cascade = CascadeType.ALL)
   @JoinColumn(name = "address_id", referencedColumnName = "id")
   private Address address;
}

@Entity
public class Address {
   @Id
   @GeneratedValue
```

```
    private Long id;

    private String city;
    private String street;
}
```

□ Here, student table will have a foreign key column address_id.

3.2 @OneToMany

- Defines **one-to-many relationship**.
- Example: One Department → Many Students.

□ **Example:**

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

    private String name;

    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
    private List<Student> students = new ArrayList<>();
}

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

    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id")   // Foreign key in Student table
    private Department department;
}
```

□ department_id will be created in the student table.

3.3 @ManyToOne

- Opposite of @OneToMany.
- Example: Many students belong to one department.
  (Shown above in Student entity).

3.4 @ManyToMany

- Defines **many-to-many relationship**.
- Example: Many Students can enroll in many Courses.
- Requires a **join table**.

□ **Example:**

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

   @ManyToMany
   @JoinTable(
      name = "student_course",
      joinColumns = @JoinColumn(name = "student_id"),
      inverseJoinColumns = @JoinColumn(name = "course_id")
   )
   private List<Course> courses = new ArrayList<>();
}

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

```
    private String title;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students = new ArrayList<>();
}
```

□ A new join table student_course will be created with columns student_id and course_id.

---

4. Cascade Types

- **Cascade** = Defines how operations on parent affect child entities.
- Options:
  - PERSIST → Save child automatically when saving parent.
  - MERGE → Update child automatically.
  - REMOVE → Delete child when parent deleted.
  - ALL → Applies all operations.

---

5. Fetch Types

- **FetchType.LAZY (default for collections):** Loads related entities **only when accessed**.
- **FetchType.EAGER (default for @OneToOne, @ManyToOne):** Loads related entities **immediately**.

□ **Example:**

```
@OneToMany(mappedBy = "department", fetch = FetchType.LAZY)
private List<Student> students;
```

---

6. Interview Insights

- **Q: Difference between @OneToMany and @ManyToOne?**
  → @OneToMany is parent-side, @ManyToOne is child-side of the same relationship.

- **Q: Why use mappedBy in @OneToMany?**
  → To tell Hibernate that the foreign key is maintained by the other entity (avoids extra join table).
- **Q: Difference between EAGER and LAZY fetching?**
  → EAGER loads immediately (can cause performance issues), LAZY loads only when accessed (better performance).
- **Q: What happens if we don't use @JoinColumn?**
  → Hibernate creates a default join column like address_id.

Spring Data JPA – Query Methods & Custom Queries

1. Introduction

- Spring Data JPA allows querying the database in **two main ways**:
  1. **Derived Query Methods** → Based on method naming convention.
  2. **Custom Queries** → Using @Query annotation (JPQL or Native SQL).

□ This removes the need to write DAO classes manually.

---

2. Query Methods (Derived Queries)

Naming Convention

Spring Data JPA parses method names and generates SQL automatically.

Syntax:

```
findBy<PropertyName>
readBy<PropertyName>
getBy<PropertyName>
```

□ **Example:**

```
public interface StudentRepository extends JpaRepository<Student, Long> {
   List<Student> findByName(String name);
   Student findByEmail(String email);
   List<Student> findByAgeGreaterThan(int age);
   List<Student> findByCityAndAge(String city, int age);
}
```

Generated Queries (internally):

- findByName("John") → SELECT * FROM student WHERE name = 'John'
- findByAgeGreaterThan(20) → SELECT * FROM student WHERE age > 20
- findByCityAndAge("Pune", 22) → SELECT * FROM student WHERE city='Pune' AND age=22

Common Keywords in Method Names

Keyword	Meaning	Example
And	AND condition	findByNameAndCity
Or	OR condition	findByNameOrCity
Between	Range	findByAgeBetween(18,25)
LessThan, GreaterThan	Comparisons	findBySalaryGreaterThan(50000)
Like	Pattern matching	findByNameLike("%John%")
OrderBy	Sorting	findByAgeOrderByNameDesc
In	Matches list	findByCityIn(List<String> cities)
IsNull, IsNotNull	Null checks	findByEmailIsNull()

---

3. Custom Queries (@Query)

Sometimes naming conventions are not enough.
We can use **JPQL** (Java Persistence Query Language) or **Native SQL**.

3.1 JPQL Queries (@Query)

- JPQL works with **entity names and fields**, not table/column names.

□ Example:

```
public interface StudentRepository extends JpaRepository<Student, Long> {

    // JPQL query
    @Query("SELECT s FROM Student s WHERE s.name = ?1")
    List<Student> findByNameJPQL(String name);

    // Using named parameter
    @Query("SELECT s FROM Student s WHERE s.city = :city")
    List<Student> findByCity(@Param("city") String city);
}
```

□ Here, Student is the entity name, not the table.

3.2 Native Queries (nativeQuery = true)

- When JPQL is not enough, we can write **raw SQL**.

□ **Example:**

```
public interface StudentRepository extends JpaRepository<Student, Long> {

    @Query(value = "SELECT * FROM students WHERE city = ?1", nativeQuery = true)
    List<Student> findByCityNative(String city);
}
```

3.3 Modifying Queries (@Modifying)

- For **update or delete** queries, use @Modifying along with @Transactional.

□ **Example:**

```
@Transactional
@Modifying
@Query("UPDATE Student s SET s.city = :city WHERE s.id = :id")
int updateStudentCity(@Param("id") Long id, @Param("city") String city);

@Transactional
@Modifying
@Query("DELETE FROM Student s WHERE s.city = :city")
int deleteByCity(@Param("city") String city);
```

4. Example Usage in Service

```
@Service
public class StudentService {

   @Autowired
   private StudentRepository repo;

   public void demoQueries() {
      repo.findByName("John");
      repo.findByAgeGreaterThan(18);
      repo.findByCity("Pune");
      repo.findByCityNative("Mumbai");
      repo.updateStudentCity(1L, "Delhi");
   }
}
```

5. Interview Insights

- **Q: Difference between JPQL and Native SQL?**
  → JPQL uses **entities and fields**, is database-independent.
  → Native SQL uses **table/column names**, is database-specific.
- **Q: Can we write update/delete queries with @Query?**
  → Yes, but must use @Modifying + @Transactional.
- **Q: When should we use derived methods vs @Query?**
  → Use derived queries for simple cases, @Query for complex logic or joins.
- **Q: What if a method name is too long in derived queries?**
  → Use @Query for better readability.

Spring Data JPA – Pagination & Sorting

1. Why Pagination & Sorting?

- In real-world applications, data sets can be **huge** (thousands or millions of rows).
- Instead of fetching all rows at once, we fetch **small chunks (pages)** for performance.
- Sorting helps order results efficiently.

□ Spring Data JPA provides **built-in support** for pagination & sorting via:

- Pageable (for pagination)
- Sort (for sorting)
- Page and Slice (to represent result subsets)

---

2. Pageable (Pagination)

□ Interface:

```
Pageable pageable = PageRequest.of(pageNumber, pageSize);
```

- pageNumber → starts from 0
- pageSize → number of records per page

□ Repository Usage:

```
public interface StudentRepository extends JpaRepository<Student, Long> {
    Page<Student> findByCity(String city, Pageable pageable);
}
```

□ Example:

```
@Autowired
private StudentRepository repo;

public void demoPagination() {
    Pageable pageable = PageRequest.of(0, 3);  // Page 0, size 3
    Page<Student> page = repo.findByCity("Pune", pageable);

    System.out.println("Total Elements: " + page.getTotalElements());
    System.out.println("Total Pages: " + page.getTotalPages());
    page.getContent().forEach(System.out::println);
}
```

---

3. Sort (Sorting)

□ Usage:

```
Sort sort = Sort.by("name").ascending();
List<Student> students = repo.findAll(sort);
```

Multiple fields:

```
Sort sort = Sort.by("city").descending().and(Sort.by("name").ascending());
```

---

4. Combining Pagination & Sorting

□ Example:

```
Pageable pageable = PageRequest.of(0, 5, Sort.by("name").ascending());
Page<Student> page = repo.findAll(pageable);

page.getContent().forEach(System.out::println);
```

□ Here: Fetch **first 5 students sorted by name ASC**.

---

6. Example Repository

```
public interface StudentRepository extends JpaRepository<Student, Long> {
   Page<Student> findByCity(String city, Pageable pageable);
   Slice<Student> findByAgeGreaterThan(int age, Pageable pageable);
}
```

---

7. Interview Insights

- **Q: What is the difference between Page and Slice?**
  → Page gives content + total count + total pages.
  → Slice gives only content + next-page info (faster).
- **Q: How does Spring Data JPA implement pagination internally?**
  → It uses LIMIT and OFFSET in SQL queries.

- **Q: Can we sort and paginate together?**
  → Yes, use PageRequest.of(page, size, Sort).
- **Q: Which fetch strategy is best for large datasets?**
  → Use Page or Slice instead of fetching all records.

Spring Data JPA – Database Configuration

1. Introduction

Spring Boot makes it easy to configure databases.

- By default, it uses **H2 in-memory DB** if no external DB is configured.
- For MySQL/PostgreSQL, we just add the **JDBC driver dependency** and configure application.properties (or application.yml).

□ The **main configs** needed:

1. spring.datasource.url → JDBC connection string
2. spring.datasource.username & spring.datasource.password
3. spring.jpa.hibernate.ddl-auto → Schema management (create, update, validate)
4. spring.jpa.show-sql → Print SQL queries in console

---

2. Common JPA Properties

Property	Description	Example
spring.datasource.url	JDBC URL	jdbc:mysql://localhost:3306/testdb
spring.datasource.username	DB username	root
spring.datasource.password	DB password	admin
spring.datasource.driver-class-name	JDBC driver (optional, auto-detected)	com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto	Schema strategy (create,	update

Property	Description	Example
	update, validate, none)	
spring.jpa.show-sql	Show SQL queries in logs	true
spring.jpa.properties.hibernate.dialect	SQL dialect	org.hibernate.dialect.MySQL8Dialect

---

3. MySQL Configuration

□ Dependency (Maven)

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

□ application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/testdb?useSSL=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

---

4. PostgreSQL Configuration

□ Dependency (Maven)

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
```

```
</dependency>
```

□ **application.properties**

```
spring.datasource.url=jdbc:postgresql://localhost:5432/testdb
spring.datasource.username=postgres
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

---

5. H2 Database (In-Memory)

□ Best for testing, development, and quick demos.

□ **Dependency (Maven)**

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

□ **application.properties**

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

□ **H2 Console (Web UI)**

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

□ Access at: http://localhost:8080/h2-console

---

6. Schema Generation Strategies (ddl-auto)

Value	Description
create	Drops and recreates schema at startup (data lost each restart).
update	Updates schema without dropping (recommended for dev).
validate	Validates schema against entity mappings (no changes).
none	Disables auto schema management.

7. Interview Insights

- **Q: Difference between H2 and MySQL/Postgres?**
  → H2 is in-memory, fast, used for dev/testing. MySQL/Postgres are persistent relational databases for production.
- **Q: What happens if ddl-auto=create?**
  → Tables are dropped and recreated each app restart (not suitable for prod).
- **Q: Why specify Hibernate dialect?**
  → To optimize SQL generation for specific database features.
- **Q: How does Spring Boot auto-detect DB?**
  → By checking which JDBC driver is on the classpath.

Transactions in Spring Data JPA (@Transactional)

1. What is a Transaction?

- A **transaction** is a unit of work that is either **fully completed** or **fully rolled back**.
- Ensures **data integrity** in case of failures (system crash, exception, etc.).
- Follows **ACID properties**:
  - **Atomicity** – All or nothing.
  - **Consistency** – DB moves from one valid state to another.
  - **Isolation** – Transactions don’t interfere with each other.
  - **Durability** – Once committed, changes persist.

2. Why Transactions in Spring?

- Database operations like INSERT, UPDATE, DELETE must be **atomic**.
- Prevents **partial updates**.
- Example:
    - Money transfer between two accounts.
    - Debit must rollback if credit fails.

---

3. @Transactional Annotation

- Defined in org.springframework.transaction.annotation.Transactional.
- Used on **methods** or **classes**.
- Automatically starts and commits/rolls back a transaction.

Example:

```
@Service
public class AccountService {

    @Autowired
    private AccountRepository accountRepository;

    @Transactional
    public void transferMoney(Long fromId, Long toId, Double amount) {
        Account from = accountRepository.findById(fromId).orElseThrow();
        Account to = accountRepository.findById(toId).orElseThrow();

        from.setBalance(from.getBalance() - amount);
        to.setBalance(to.getBalance() + amount);

        accountRepository.save(from);
        accountRepository.save(to);

        // If exception occurs, transaction rolls back automatically
    }
}
```

---

4. Transactional Rollback

- By default: rolls back **RuntimeException** and **Error**.
- Can configure to rollback for specific exceptions.

Example:

```
@Transactional(rollbackFor = Exception.class)
public void doSomething() {
   // Will rollback for checked exceptions too
}
```

---

5. Propagation Types

Defines how a method participates in a transaction.

- REQUIRED (default) → Use existing transaction or create new.
- REQUIRES_NEW → Always starts a new transaction.
- MANDATORY → Must run inside existing transaction, else error.
- SUPPORTS → Use transaction if available, else run without.
- NOT_SUPPORTED → Run without transaction.
- NEVER → Throws error if transaction exists.
- NESTED → Executes within a nested transaction.

---

6. Isolation Levels

Controls how one transaction is **isolated** from others.

- READ_UNCOMMITTED → Can read uncommitted data (**dirty read**).
- READ_COMMITTED → Prevents dirty reads (default in many DBs).
- REPEATABLE_READ → Prevents dirty & non-repeatable reads.
- SERIALIZABLE → Highest isolation, but lowest performance.

Example:

```
@Transactional(isolation = Isolation.REPEATABLE_READ)
public void processOrder() {
   // Safe from dirty/non-repeatable reads
}
```

---

7. Best Practices

- Use @Transactional only on **Service Layer**, not Controller.
- Keep transactional methods **short & focused**.
- Don't call @Transactional methods **internally** in the same class (self-invocation issue).
- Configure proper **isolation** & **rollback rules** for performance.

Database Migrations with Flyway & Liquibase

1. □ Why Database Migrations?

When working with Spring Boot + JPA:

- Database schema evolves (new tables, columns, constraints).
- Manual changes are risky and error-prone.
- Migration tools (Flyway, Liquibase) automate schema evolution with version-controlled scripts.

□ Benefits:

- Version control for DB schema
- Easy rollback and tracking
- Ensures consistency across environments (dev, test, prod)

---

2. Flyway in Spring Boot

□ What is Flyway?

- Flyway is a **database migration tool**.
- Uses **SQL scripts or Java classes** to manage DB schema.
- Default location: src/main/resources/db/migration/

□ Naming Convention

- Scripts must follow:
  V<version_number>__<description>.sql

Example:

```
V1__Create_users_table.sql
V2__Add_email_to_users.sql
```

□ Example: Flyway SQL Migration

src/main/resources/db/migration/V1__Create_users_table.sql

```
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);
```

src/main/resources/db/migration/V2__Add_age_to_users.sql

```
ALTER TABLE users ADD age INT;
```

□ Configuration in application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=root

spring.flyway.enabled=true
spring.flyway.baseline-on-migrate=true
```

□ On app startup, Flyway runs migrations in order.

---

3. Liquibase in Spring Boot

□ What is Liquibase?

- Alternative to Flyway.
- Supports **XML, YAML, JSON, SQL** for schema changes.
- Provides more **advanced rollback** and changelog management.

□ Liquibase Files

Default: src/main/resources/db/changelog/db.changelog-master.yaml

Example:

```yaml
databaseChangeLog:
  - changeSet:
      id: 1
      author: prathamesh
      changes:
        - createTable:
            tableName: users
            columns:
              - column:
                  name: id
                  type: BIGINT
                  autoIncrement: true
                  constraints:
                    primaryKey: true
              - column:
                  name: name
                  type: VARCHAR(100)
                  constraints:
                    nullable: false
              - column:
                  name: email
                  type: VARCHAR(100)
                  constraints:
                    unique: true
                    nullable: false
  - changeSet:
      id: 2
      author: prathamesh
      changes:
        - addColumn:
            tableName: users
            columns:
              - column:
                  name: age
                  type: INT
```

□ Configuration in application.properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/mydb
spring.datasource.username=postgres
spring.datasource.password=admin

spring.liquibase.change-log=classpath:db/changelog/db.changelog-master.yaml
```

□ On startup, Liquibase applies changes defined in changelogs.

---

4. Flyway vs Liquibase

Feature	Flyway	Liquibase
Migration format	SQL (preferred), Java	SQL, XML, YAML, JSON
Learning curve	Easier	Steeper
Rollback	Limited	Strong rollback support
Community	Large, widely used	Strong, enterprise-focused
Best for	Simple migrations	Complex migrations with rollback

---

5. Best Practices

- Keep migrations in **version control (Git)**.
- Always use **incremental migrations**, not modifying old ones.
- Use **separate profiles** for dev/test/prod DBs.
- For big teams, **Liquibase** offers better tracking; for small projects, **Flyway** is simpler.