# **1. What Is Spring Data JPA?**

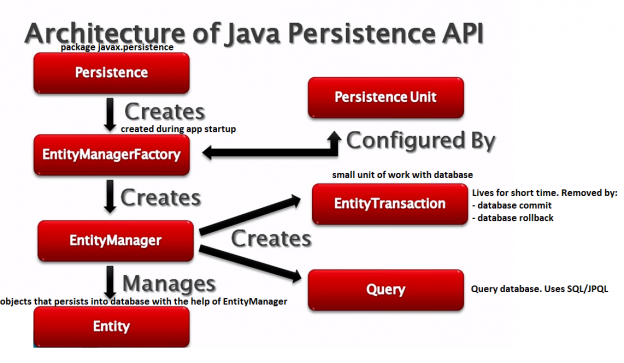
Spring Data JPA builds on top of the Java Persistence API (JPA) to dramatically simplify data access layers:

* **Repository abstraction**: you write interfaces, not implementations
* **Query derivation**: method names become queries
* **Paging & sorting** built-in
* Integration with Spring’s **transaction management**, **dependency injection**, and **auditing**

**2. Core Components**

| **Component** | **Role** |
| --- | --- |
| **Entity classes** | - Plain Java classes annotated with @Entity, @Table, etc., modeling database tables. |
| **EntityManagerFactory** | - Bootstrapped by Spring from your spring.datasource + JPA properties. |
| **EntityManager** | - JPA’s core API for CRUD, queries, and managing the persistence context. |
| **PlatformTransactionManager** | - Binds JPA transactions to Spring’s @Transactional. |
| **Repository interfaces** | - Extend JpaRepository<T,ID> (or CrudRepository) to gain save(), findAll(), etc. |
| **RepositoryFactoryBean** | - Under the hood, Spring creates a proxy/implementation of your interface at startup. |
| **Query/Specification API** | - Support for method-name queries, @Query, and the JPA Criteria/Specification API. |
| **Auditing support** | - @CreatedDate, @LastModifiedDate via @EnableJpaAuditing. |

**3. Architectural Flow**



1. **Startup**
   * Spring Boot auto-configures EntityManagerFactory, TransactionManager, and creates repository proxies.
2. **At runtime**
   * You @Autowired MyEntityRepository repo;
   * Calling repo.findByLastName("Smith") invokes the proxy, which looks up the query (derived or @Query), obtains an EntityManager, starts a transaction (if needed), executes the JPQL/SQL, maps rows to entities, and returns results.
3. **Transactions**
   * Methods in repositories are by default read-only transactions (or read-write for save()).
   * Services can use @Transactional to combine multiple repository calls in one atomic unit.

**4. When & Where to Use Spring Data JPA**

* **CRUD-Heavy Domains**  
  Admin back-offices, content management, user directories.
* **Rapid Prototyping**  
  Stand up data access layers with zero boilerplate—ideal for MVPs and greenfield services.
* **Microservices**  
  Each service with its own data model uses Spring Data JPA for consistency and rapid development.
* **Complex Querying**  
  Leverage method-name queries, JPQL, the Criteria API, or QueryDSL (via Spring Data extensions) for dynamic filtering.

**5. Real-World Examples**

**Example 1: E-Commerce Product Catalog**

public interface ProductRepository extends JpaRepository<Product, Long> {

List<Product> findByCategoryOrderByPriceAsc(String category);

@Query("SELECT p FROM Product p WHERE p.stockQty < :threshold")

List<Product> findLowStock(@Param("threshold") int threshold);

}

* **Use case**: Homepage shows “Top 10 cheapest electronics” via findByCategory…, inventory dashboard flags low-stock items.

**Example 2: Banking Customer Service**

public interface AccountRepository extends JpaRepository<Account, Long> {

Optional<Account> findByAccountNumber(String acctNo);

Page<Account> findByCustomerId(Long customerId, Pageable page);

}

* **Use case**: Customer portal pages list accounts by page, back-office looks up by account number; no DAOs to write.

**Example 3: IoT Sensor Data Storage**

public interface SensorDataRepository extends JpaRepository<SensorData, UUID> {

List<SensorData> findByDeviceIdAndTimestampBetween(

String deviceId, Instant from, Instant to);

}

* **Use case**: Analytics UI queries a time window of telemetry for a given device; Spring Data JPA handles paging and sorting.

**6. Tips & Best Practices**

* **DTO Projections**  
  Use interface-based projections (List<IPersonSummary> findAllBy()) to fetch only needed columns.
* **Bulk Operations**  
  For large deletes/updates, use @Modifying @Query to avoid loading entities.
* **Avoid N+1 Queries**  
  Use @EntityGraph or fetch joins in JPQL to eagerly load relationships where needed.
* **Custom Extensions**  
  For logic beyond CRUD, define a custom repository implementation:

MyRepo extends JpaRepository<T,ID>, MyRepoCustom

class MyRepoImpl implements MyRepoCustom { /\* custom methods \*/ }

## **1. Why Spring Data MongoDB Came**

1. **NoSQL Popularity**  
   As applications moved beyond relational schemas, document stores like MongoDB gained traction for their schema flexibility, horizontal scalability, and rich JSON‐style queries.
2. **Boilerplate Reduction**  
   Raw MongoDB Java driver code requires manual mapping between org.bson.Document and POJOs, plus verbose CRUD boilerplate. Spring Data MongoDB’s repository abstraction and mapping layer eliminate that.
3. **Consistent Spring Programming Model**  
   Brings Spring Data’s familiar **Repository** pattern, annotation‐driven mapping, and transaction support (where available) to MongoDB, so developers don’t juggle two radically different APIs in polyglot persistence systems.

**2. Core Rules & Features**

| **Feature** | **Description & Rules** |
| --- | --- |
| **MongoTemplate** | Low‐level operations (findOne, insert, update, remove) on POJOs. |
| **Repository Interfaces** | Extend MongoRepository<T,ID> (or PagingAndSortingRepository) to get save(), findAll(), etc. |
| **@Document** | Marks a class as a MongoDB document (collection name defaults to class name). |
| **@Id** | Indicates the primary identifier field (maps to MongoDB \_id). |
| **Derived Query Methods** | Define methods like List<User> findByLastName(String lastName);—Spring derives the MongoDB query. |
| **Custom Queries** | Use @Query("{ 'age' : { $gt: ?0 }}") for JSON‐style MongoDB queries or Query objects in MongoTemplate. |
| **Pagination & Sorting** | Pass a Pageable to repository methods, or use skip()/limit() on Query. |
| **Auditing Support** | @CreatedDate, @LastModifiedDate with @EnableMongoAuditing. |
| **Transactions (4.0+)** | Multi‐document ACID transactions supported on replica sets / sharded clusters; enabled via MongoTransactionManager. |

**3. When & Where to Use**

* **Schema‐Flexible Domains**  
  When your data model evolves frequently (e.g. user‐generated content, product catalogs with varying attributes).
* **High‐Throughput Logging / Event Streams**  
  Append‐only sensor data, application logs, or analytics events where query patterns favor document lookups over complex joins.
* **Geo‐Spatial & Full‐Text Search**  
  Built‐in support for geospatial indexes and text search makes MongoDB ideal for location‐based services or content search.
* **Polyglot Persistence**  
  In microservices, use MongoDB for certain bounded contexts (e.g., session store, shopping cart) alongside RDBMS.
* **Rapid Prototyping**  
  No schema migrations: simply start writing to collections; you can add new fields without downtime.

**4. Real-World Examples**

**Example 1: E-Commerce Product Catalog**

* **Domain:** Products have varying attributes (size, color, specs).
* **Entity:**

@Document("products")

public class Product {

@Id private String id;

private String name;

private String category;

private Map<String,Object> attributes; // flexible

private BigDecimal price;

// getters/setters…

}

* **Repository:**

public interface ProductRepository extends MongoRepository<Product,String> {

List<Product> findByCategory(String category);

@Query("{ 'attributes.?0': ?1 }")

List<Product> findByAttribute(String key, Object value);

}

* **Use Case:** Catalog service that lets new product types roll out without schema changes.

**Example 2: IoT Sensor Data Ingestion**

* **Domain:** High‐volume telemetry from devices; each record has timestamp, deviceId, readings.
* **Entity:**

@Document("sensorData")

public class SensorData {

@Id private String id;

private String deviceId;

private Instant timestamp;

private Map<String,Double> readings;

}

* **Template Usage:**

Query q = Query.query(Criteria.where("deviceId").is(deviceId))

.addCriteria(Criteria.where("timestamp").gte(start).lte(end))

.with(Sort.by("timestamp").ascending());

List<SensorData> window = mongoTemplate.find(q, SensorData.class);

* **Use Case:** Real-time dashboards that plot sensor metrics over sliding time windows.

**Example 3: User Session Store with Transactions**

* **Domain:** Web sessions stored in MongoDB, occasionally updated across multiple collections in one transaction.
* **Configuration:**

@Bean

MongoTransactionManager transactionManager(MongoDatabaseFactory factory) {

return new MongoTransactionManager(factory);

}

* **Transactional Service:**

@Service

public class SessionService {

@Autowired private MongoTemplate template;

@Transactional

public void transferSessionData(String fromId, String toId) {

Session from = template.findById(fromId, Session.class);

Session to = template.findById(toId, Session.class);

to.mergeData(from.getData());

template.save(to);

template.remove(from);

}

}

* **Use Case:** Safely migrate or merge sessions across shards without losing data.

**5. Best Practices**

1. **Model Documents Thoughtfully**  
   Align collections with query patterns—embed vs reference decisions impact performance (denormalize for read speed).
2. **Use Indexes**  
   Define indexes via @Indexed or in code to speed up common queries (e.g. deviceId, geospatial fields).
3. **Handle Large Results**  
   Use pagination (Pageable) or cursors (template.stream(Query, Class)) to avoid OOM on large collections.
4. **Leverage Audit Fields**  
   Enable Mongo auditing to auto‐populate createdAt/updatedAt.
5. **Monitor & Scale**  
   Use MongoDB’s built-in monitoring or Spring Actuator metrics to track connection pools, query latencies, and consider sharding for write scale.

# **✅ Spring Data JPA Quiz**

**1. What does Spring Data JPA provide over standard JPA?**  
A. A new ORM framework  
B. Support for JDBC templates  
C. Abstractions for data access layers with built-in repository implementations  
D. Database encryption

**Answer:** C. Abstractions for data access layers with built-in repository implementations

**2. Which interface is the base for Spring Data JPA repositories?**  
A. JpaContext  
B. JpaRepository  
C. EntityManager  
D. CrudDao

**Answer:** B. JpaRepository

**3. What does the @Entity annotation do?**  
A. Configures the database  
B. Marks a class as a Spring component  
C. Maps a class to a database table  
D. Configures a controller

**Answer:** C. Maps a class to a database table

**4. Which annotation is used to define the primary key in an entity class?**  
A. @PrimaryKey  
B. @Id  
C. @Column  
D. @PK

**Answer:** B. @Id

**5. What is the purpose of the @GeneratedValue annotation?**  
A. Marks a field as optional  
B. Specifies a column’s name  
C. Indicates that the primary key should be auto-generated  
D. Disables JPA validation

**Answer:** C. Indicates that the primary key should be auto-generated

**6. What does the findByUsername(String username) method imply in a repository interface?**  
A. A custom SQL query  
B. A hardcoded value  
C. A derived query using field name username  
D. An HTTP GET method

**Answer:** C. A derived query using field name username

**7. What annotation is used to create a custom JPQL or native SQL query?**  
A. @SQLQuery  
B. @Query  
C. @CustomQuery  
D. @NamedQuery

**Answer:** B. @Query

**8. What annotation is required for transactional methods in Spring Data JPA?**  
A. @Repository  
B. @Transactional  
C. @Service  
D. @Scope

**Answer:** B. @Transactional

**9. Which method retrieves all rows from a table in a JpaRepository?**  
A. readAll()  
B. getAll()  
C. fetch()  
D. findAll()

**Answer:** D. findAll()

**10. What relationship does @OneToMany represent?**  
A. One object has one property  
B. One object is part of many tables  
C. One entity is related to many of another entity  
D. Many entities are embedded

**Answer:** C. One entity is related to many of another entity

**11. What is the default fetch type for @ManyToOne relationships?**  
A. EAGER  
B. LAZY  
C. NONE  
D. FETCH

**Answer:** A. EAGER

**12. Which annotation is used to avoid circular references in bi-directional relationships when serializing JSON?**  
A. @JsonLoop  
B. @JsonIgnore  
C. @JsonBackReference and @JsonManagedReference  
D. @JsonCircular

**Answer:** C. @JsonBackReference and @JsonManagedReference

**13. What is the role of EntityManager in JPA?**  
A. It loads Spring Beans  
B. It handles security  
C. It manages persistence operations (CRUD)  
D. It validates Java classes

**Answer:** C. It manages persistence operations (CRUD)

**14. What does the save() method in JpaRepository do?**  
A. Only updates an existing entity  
B. Persists a new entity or updates an existing one  
C. Logs a record  
D. Deletes the record

**Answer:** B. Persists a new entity or updates an existing one

**15. How would you define a native SQL query in Spring Data JPA?**  
A. Use @Native  
B. Use @Query(value = "...", nativeQuery = true)  
C. Use @SQL  
D. Use @JPQL(native=true)

**Answer:** B. Use @Query(value = "...", nativeQuery = true)

## **Assignment of this week:**

You’re building a microservice for an online bookstore. All book data—titles, authors, categories, reviews—will be stored in MySql / H2. You need to implement CRUD operations, plus a couple of “real-world” queries (e.g. find books by category, list top-rated books).

**🎯 Requirements**

1. **Spring Boot REST Service**
   * Use Spring Boot with spring-boot-starter-data-jpa and spring-boot-starter-web.
2. **Domain Model**

@Document("books")

public class Book {

@Id String id;

String title;

String author;

String category; // e.g. "Fiction", "Science", "History"

List<Review> reviews; // nested documents

}

public class Review {

String reviewer;

int rating; // 1–5

String comment;

}

1. **Repository**

public interface BookRepository extends MongoRepository<Book,String> {

List<Book> findByCategory(String category);

@Query("{ 'reviews.rating': { $gte: ?0 } }")

List<Book> findByMinReviewRating(int minRating);

}

1. **REST Endpoints**
   * POST /books — create a new book
   * GET /books — list all books
   * GET /books/{id} — get a book by ID
   * PUT /books/{id} — update book metadata (title/author/category)
   * DELETE /books/{id} — remove a book
   * GET /books/category/{category} — list by category
   * GET /books/top-rated/{rating} — list books having at least one review ≥ rating
2. **Sample Data Initialization**  
   On startup, load three books with mixed categories and reviews (via a CommandLineRunner).
3. **Validation & Error Handling**
   * 400 on bad input (e.g. missing title)
   * 404 if book not found

**🛠️ Sample Inputs & Expected Outputs**

1. **Create a Book**

**Request**

POST /books

Content-Type: application/json

{

"title": "Clean Code",

"author": "Robert C. Martin",

"category": "Programming",

"reviews": [

{"reviewer":"Alice","rating":5,"comment":"Must-read!"},

{"reviewer":"Bob","rating":4,"comment":"Very helpful."}

]

}

**Response**

HTTP/1.1 201 Created

{

"id": "5f8d0d55b54764421b7156c1",

"title": "Clean Code",

"author": "Robert C. Martin",

"category": "Programming",

"reviews": [ … ]

}

1. **List Books by Category**

**Request**

GET /books/category/Programming

**Response**

HTTP/1.1 200 OK

[

{

"id": "5f8d0d55b54764421b7156c1",

"title": "Clean Code",

"author": "Robert C. Martin",

"category": "Programming",

"reviews": [ … ]

},

{

"id": "5f8d0e33b54764421b7156c2",

"title": "Effective Java",

"author": "Joshua Bloch",

"category": "Programming",

"reviews": [ … ]

}

]

1. **Find Top-Rated Books**

**Request**

GET /books/top-rated/5

**Response**

HTTP/1.1 200 OK

[

{

"id": "5f8d0d55b54764421b7156c1",

"title": "Clean Code",

"author": "Robert C. Martin",

"category": "Programming",

"reviews": [

{"reviewer":"Alice","rating":5,"comment":"Must-read!"}, …

]

}

]

1. **Update a Book**

**Request**

PUT /books/5f8d0d55b54764421b7156c1

Content-Type: application/json

{

"title": "Clean Code (2nd Edition)",

"author": "Robert C. Martin",

"category": "Software Engineering"

}

**Response**

HTTP/1.1 200 OK

{

"id": "5f8d0d55b54764421b7156c1",

"title": "Clean Code (2nd Edition)",

"author": "Robert C. Martin",

"category": "Software Engineering",

"reviews": [ … ]

}

1. **Delete a Book**

**Request**

DELETE /books/5f8d0d55b54764421b7156c1

**Response**

HTTP/1.1 204 No Content

**🔧 Deliverables**

* Maven/Gradle Spring Boot project
* Properly annotated domain classes
* Repository interfaces with derived and @Query methods
* REST controllers with above endpoints
* CommandLineRunner to seed sample data
* application.properties pointing to an embedded or local MySql or H2

# **Example Project:**

**Step-by-Step Implementation Example**

**Step 1: Create a Spring Boot Project**

Use [IntelliJ IDEA](https://www.geeksforgeeks.org/springboot/how-to-create-a-spring-boot-project-with-intellij-idea/) or [Spring Initializr.](https://start.spring.io/) Add the following dependencies:

* Spring Web
* Spring Data JPA
* H2 Database
* Lombok

**Step 2: Project Structure**

Create 4 packages as listed below and create some classes and interfaces inside these packages as seen in the below image

* entity
* repository
* service
* controller

**Step 3. Entity Class**

Inside the entity package create a simple [POJO class](https://www.geeksforgeeks.org/advance-java/pojo-vs-java-beans/) inside the Department.java file.

**Step 4: Repository Interface**

Inside the repository package create a simple interface and name the interface as DepartmentRepository. This interface is going to extend the JpaRepository as we have discussed above.

**Step 5: Service Layer**

Inside the service package create one interface named as DepartmentService and one class named as DepartmentServiceImpl

**Step 6: Controller Layer**

Inside the controller package create one class named as DepartmentController.

**Step 7: application.properties**

Below is the code for the application.properties file

server.port=8082

# H2 Database

spring.h2.console.enabled=true

spring.datasource.url=jdbc:h2:mem:dcbapp

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

**Step 8: Run Application**

Now run your application and let's test the endpoints in Postman and also refer to our H2 Database.

**Testing the Endpoint in Postman**

**Endpoint 1:** POST - http://localhost:8082/departments/

**Endpoint 2:** GET - http://localhost:8082/departments/

**Endpoint 3:** PUT - <http://localhost:8082/departments/1>

**Endpoint 4:** DELETE - <http://localhost:8082/departments/1>

## **Above project is using H2 Db from RAM we can switch to MySql Db by doing below changes**

**📦 Dependencies**

Since H2 is embedded, you don’t need to install anything.  
For MySQL, add this dependency in your build.gradle:

implementation 'mysql:mysql-connector-java:8.0.33'  
  
  
  
In Maven, you add dependencies inside the <dependencies> section of your pom.xml.  
For MySQL connector, add this:

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.33</version>

</dependency>

application.properties

# ===============================

# MySQL Database Configuration

# ===============================

spring.datasource.url=jdbc:mysql://localhost:3306/dcbapp?useSSL=false&serverTimezone=UTC&allowPublicKeyRetrieval=true

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.username=root

spring.datasource.password=your\_password

# JPA / Hibernate

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true  
  
  
Even though you’re using **Spring Data JPA**, under the hood it **uses Hibernate as the default JPA provider** (unless you explicitly switch to EclipseLink, OpenJPA, etc.).

So those properties (spring.jpa.database-platform, spring.jpa.hibernate.ddl-auto, spring.jpa.show-sql) are **actually Hibernate-related**, but they are exposed through Spring Boot’s spring.jpa.\* abstraction.

👉 Meaning: **You are using Spring JPA, but Hibernate is still the engine running it.**

Let me re-explain your configs in **Spring Data JPA terms** (not just Hibernate):

**1. spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect**

* **JPA meaning**: Tells JPA provider (Hibernate by default) which SQL dialect to use for query translation.
* **Without this property**, Spring Boot usually auto-detects the dialect from your DB connection. But explicitly setting it avoids mistakes.
* Example:  
  JPA query like
* @Query("SELECT s FROM Student s WHERE s.age > 20")

gets translated into **MySQL 8 syntax**:

select student0\_.id as id1\_0\_, student0\_.age as age2\_0\_, student0\_.name as name3\_0\_

from student student0\_

where student0\_.age>20;

**2. spring.jpa.hibernate.ddl-auto=update**

* **JPA meaning**: Tells the persistence provider (Hibernate) how to handle schema generation.
* Values:
  + none → JPA won’t touch schema.
  + validate → JPA validates schema matches entities.
  + update → JPA updates schema (adds missing columns/tables).
  + create → JPA drops & recreates schema at startup.
  + create-drop → Same as create, but also drops on shutdown.
* Example:
  + If you add a new field in your entity:
  + private String email;
  + With update, JPA will issue:
  + alter table student add column email varchar(255);

**3. spring.jpa.show-sql=true**

* **JPA meaning**: Logs SQL queries executed by the persistence provider (Hibernate).
* Example:  
  If you do:
* studentRepository.findAll();

You’ll see:

select student0\_.id as id1\_0\_, student0\_.age as age2\_0\_, student0\_.name as name3\_0\_ from student student0\_;

✅ **Key Point**:

* You are using **Spring Data JPA**.
* JPA is just a **specification (an API)**.
* Hibernate is the **default implementation** that Spring Boot wires in.
* These configs are Spring Boot’s way of customizing Hibernate **through JPA abstraction**.