JPA, Hibernate, and Spring Data JPA

Starting from the problem of persisting a Java object into a database, this lesson builds the ORM → JPA (specification) → Hibernate (implementation) → Spring Data JPA (repository abstraction) → Spring Boot (auto-configuration) mental model using this project's own Topic/TopicRepository classes. The 1st lesson in the Spring Data JPA category.

Intermediate 25 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/jpa-hibernate-and-spring-data-jpa

---

You already know how to define a Java class, inject a dependency, and expose it over a REST endpoint. What you haven't seen yet is how a Java object actually gets INTO a database, and back out again, without you writing SQL by hand for every single field of every single class. That gap is what this whole category is about — and before touching a single repository interface, this first lesson builds the mental model underneath four names you'll keep seeing together: JPA, Hibernate, Spring Data JPA, and Spring Boot.

The Problem: A Java Object Is Not a Database Row

Start with the simplest possible question: how would you persist a Java object — say, a `Topic` with an `id` and a `title` — into a relational database?

```
public class PlainJavaTopicExample {

    // A plain Java object -- nothing here says anything about a database.
    // The JVM only knows how to keep this alive in memory; it has no idea
    // "topic" should correspond to a table, or that "title" should
    // correspond to a column.
    static class Topic {
        private Long id;
        private String title;

        Topic(Long id, String title) {
            this.id = id;
            this.title = title;
        }
    }

    public static void main(String[] args) {
        Topic topic = new Topic(1L, "Introduction to Generics");

        // Without any help, "saving" this means hand-writing SQL yourself,
        // and hand-writing it again for every other entity in the
        // application:
        //
        //   String sql = "INSERT INTO topic (id, title) VALUES (?, ?)";
        //   preparedStatement.setLong(1, topic.getId());
        //   preparedStatement.setString(2, topic.getTitle());
        //   preparedStatement.executeUpdate();
        //
        // This is exactly the repetitive, error-prone gap ORM exists to close.
        System.out.println(topic.title);
    }
}
```

Nothing about this `Topic` class knows it should end up as a row in a `topic` table. To actually save it, you'd write a `PreparedStatement`, map each field to a column by hand, and repeat that same mechanical work for every entity in the application — `Category`, `Course`, and everything else. Reading it back means the reverse: mapping each column of a `ResultSet` back onto a field, again by hand, again for every class. None of this is hard, exactly — it's just repetitive, easy to get subtly wrong, and it's the same shape of code over and over. That repetition is the problem the next four layers exist to solve, one at a time.

What Is ORM?

Object-Relational Mapping (ORM) is the general idea of automating exactly that translation — letting a tool read a class's structure and generate the SQL to store and retrieve it, instead of you writing that SQL by hand for every class. ORM itself isn't a Java-specific thing, or even a single tool; it's the concept. JPA, introduced next, is Java's standardized way of expressing that idea.

What Is JPA?

JPA (Jakarta Persistence API) is a SPECIFICATION — a set of interfaces and annotations (`@Entity` , `@Id` , `EntityManager` , and more) that describe HOW to do object-relational mapping in Java, without providing the actual code that does it. This is worth being precise about, because it answers a question beginners often get wrong: JPA is not a library you can run on its own — a specification has no runtime behavior at all. It only defines a contract; something else has to implement that contract.

What Is Hibernate?

Hibernate is that something else — a concrete IMPLEMENTATION of the JPA specification. When your code uses `@Entity` or calls a method JPA defines, it's Hibernate underneath that actually reads the annotation, generates the SQL, talks to the JDBC driver, and gets a row into PostgreSQL. Other JPA implementations exist (EclipseLink, for one), but Hibernate is by far the most common choice in real Spring Boot applications, including this one.

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

// The SAME class as PlainJavaTopicExample's Topic, with three annotations
// added. Nothing about the class's own Java behavior changed -- these
// annotations are metadata, read by JPA's implementation (Hibernate) to
// generate the INSERT/UPDATE/SELECT SQL that PlainJavaTopicExample had to
// write by hand.
@Entity
public class MinimalEntityExample {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    // A no-argument constructor is required -- JPA implementations create
    // entity instances via reflection before populating their fields. The
    // full reasoning behind this (and everything else about mapping an
    // entity correctly) is covered in "Entities and the Repository
    // Abstraction," the next lesson in this series.
    protected MinimalEntityExample() {
    }

    public MinimalEntityExample(String title) {
        this.title = title;
    }
}
```

The class is unchanged from `PlainJavaTopicExample` 's `Topic` — only three annotations were added. `@Entity` tells JPA (and, underneath, Hibernate) that this class maps to a table; `@Id`

marks which field is the primary key; `@GeneratedValue` says the database should generate that key's value. None of that SQL from the earlier example needs to be written anymore — Hibernate generates it from these annotations. Notice, too, the no-argument constructor `MinimalEntityExample` requires: Hibernate builds entity instances via reflection before populating their fields, which is also exactly why "Record" points out that a `record` — with no no-args constructor and no mutable fields — can't be used as a JPA entity at all.

> A no-args constructor is one of several requirements a class must satisfy to be a valid JPA entity — "Entities and the Repository Abstraction," next in this category, covers the rest (`@Table` , `@Enumerated` , `nullable` / `unique` constraints, and more) in full.

```
Java object
     ↓
    JPA
     ↓
 Hibernate
     ↓
    SQL
     ↓
PostgreSQL
```

What Is Spring Data JPA?

JPA plus Hibernate alone already solves the original problem — but using them directly still means writing an `EntityManager`-based class, by hand, for every entity, with the same handful of methods (`save` , `findById` , `findAll` , `delete`) repeated almost verbatim each time. Spring Data JPA is a REPOSITORY ABSTRACTION built on top of JPA that removes exactly that remaining repetition: you declare an interface, and Spring Data JPA generates a working implementation for you, at application startup.

```
Repository
     ↓
Spring Data JPA
     ↓
    JPA
     ↓
 Hibernate
     ↓
PostgreSQL
```

This is the layer this project's own code actually uses.

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

import java.util.Optional;

// This is a simplified, teaching-focused look at THIS PROJECT'S OWN
// TopicRepository (see src/main/java/com/cdurgun/learning/repository/
// TopicRepository.java) -- extending JpaRepository<Topic, Long> is the
// entire implementation. No class implements this interface by hand;
// Spring Data JPA generates a working implementation at runtime.
interface TopicRepositoryExample extends JpaRepository<TopicExample, Long> {

    // save(topic), findById(id), findAll(), deleteById(id), and more all
    // come from JpaRepository for free -- none of them are written here.

    // findBySlug(...) is NOT free the same way -- Spring Data JPA reads
    // this method's NAME and derives a "WHERE slug = ?" query from it.
    // Exactly how that derivation works is the subject of "Query Methods
    // and JPQL with @Query," the lesson after next -- for now, notice only
    // that no query, no SQL, no JPQL is written here, and yet this method
    // works.
    Optional<TopicExample> findBySlug(String slug);
}

class TopicExample {
    private Long id;
    private String slug;
}
```

`TopicRepositoryExample` (modeled directly on this project's real `TopicRepository`) declares nothing but an interface, yet `save(...)`, `findById(...)`, `findAll()`, and `deleteById(...)` all work — they come from `JpaRepository` for free. `findBySlug(...)` isn't inherited the same way; Spring Data JPA reads that METHOD NAME and derives the query behind it — exactly how is the subject of "Query Methods and JPQL with @Query," two lessons from now.

> Spring Data JPA does NOT replace JPA or Hibernate — it sits on top of both. Every query Spring Data JPA runs still goes through JPA's `EntityManager` and still gets turned into SQL by Hibernate underneath. Spring Data JPA's whole job is removing the repetitive boilerplate around that, not replacing the mechanism itself.

How the Four Layers Fit Together

Stacking the two diagrams above into one picture makes the whole relationship concrete:

```
Repository            (you write an interface)
     ↓
Spring Data JPA        (generates a working implementation)
     ↓
    JPA                (the specification: @Entity, EntityManager, ...)
     ↓
 Hibernate              (the implementation: turns JPA calls into SQL)
     ↓
    SQL
     ↓
PostgreSQL
```

Each layer solves the problem the layer below it left unsolved: JPA standardizes WHAT object-relational mapping should look like in Java; Hibernate actually DOES it; Spring Data JPA removes the repetitive boilerplate of using Hibernate/JPA directly, one repository interface at a time.

Where Spring Boot Fits

Spring Boot's role here is narrower than it might seem: it doesn't add another layer to the diagram above at all — it just wires the existing ones together automatically. Adding `spring-boot-starter-data-jpa` to the classpath is enough for Spring Boot to auto-configure a `DataSource`, an `EntityManagerFactory`, Hibernate as the JPA provider, and the infrastructure that turns your repository interfaces into real, working beans — all without a line of manual configuration. Exactly how that auto-configuration mechanism works (`@ConditionalOnClass`, auto-configuration classes, and so on) is already covered in "Spring Boot Auto-Configuration & Properties" — this lesson only needs the outcome: the plumbing between your code and PostgreSQL exists because Spring Boot assembled it, not because you wrote it.

A Small Example From This Project

Put together, here's what actually happens when this application starts up and later reads a topic by its slug — using the real `Topic`, `Category`, `Course`, and `TopicRepository` classes already in this codebase, not an invented example.

```
Topic.java (@Entity)
     ↓
Hibernate reads its annotations, maps it to the "topic" table
     ↓
TopicRepository extends JpaRepository<Topic, Long>
     ↓
Spring Data JPA generates a real implementation at startup
     ↓
topicRepository.findBySlug("records")
     ↓
Spring Data JPA derives a query from the method name
     ↓
JPA's EntityManager runs it
     ↓
Hibernate translates it to SQL
     ↓
PostgreSQL returns a row
     ↓
Hibernate maps that row back onto a real Topic object
```

Nowhere in this application's own source code is there a hand-written `SELECT * FROM topic WHERE slug = ?` for this — every step above happens because of the four layers this lesson just introduced, not because someone wrote that query by hand.

Common Misconceptions

A few distinctions worth stating directly, since the four names are easy to blur together: **JPA is not a library you install and run** — it's a specification with no behavior of its own; something must implement it. **Hibernate and JPA are not the same thing** — Hibernate is one (the most common) implementation of the JPA specification, not a synonym for it. **Spring Data JPA does not replace Hibernate** — it generates repository implementations that still go through JPA and still get executed as SQL by Hibernate underneath; nothing about Spring Data JPA bypasses either of them.

What Comes Next

This lesson deliberately stayed at the mental-model level — one minimal `@Entity` and one repository interface, no relationships, no custom queries, no deep dive into what makes a class a valid entity. "Entities and the Repository Abstraction," next in this category, picks up exactly there: what `@Id`, `@GeneratedValue`, `@Table`, `@Enumerated`, and a handful of other mapping annotations actually require, what the `Repository` → `CrudRepository` → `JpaRepository` interface hierarchy each adds, and what makes a class a well-formed entity in the first place.

Best Practices

- Keep the four-layer picture in mind whenever something in Spring Data JPA feels like "magic" — a repository method's behavior always traces back to JPA and Hibernate underneath, never to something Spring Data JPA invents on its own.
- Don't reach for Spring Data JPA's repository abstraction without understanding that JPA and Hibernate are still doing the real work beneath it — that understanding is what makes later, deeper topics (queries, fetching, the persistence context) make sense.
- When in doubt about which layer owns a behavior, ask: is this about WHAT mapping should look like (JPA), HOW it's actually executed (Hibernate), or removing BOILERPLATE around using them (Spring Data JPA)?

Common Mistakes

- Treating "JPA" and "Hibernate" as interchangeable words for the same thing, instead of a specification and one of its implementations.
- Assuming Spring Data JPA is a completely separate technology from JPA, rather than a thin abstraction layer built directly on top of it.
- Expecting Spring Boot to be doing something exotic under the hood — its role here is auto-configuring the same JPA/Hibernate/Spring Data JPA pieces you could otherwise wire up by hand, nothing more.
- Jumping straight to memorizing repository method names before understanding that they're generated from an underlying JPA/Hibernate mechanism, not the mechanism itself.

Summary, Cheat Sheet, and Glossary

Summary

- A plain Java object has no built-in way to become a database row — ORM is the general idea of automating that translation.
- JPA is a specification describing how object-relational mapping should work in Java; it defines a contract, not a runtime.
- Hibernate is a concrete implementation of JPA — it's what actually turns `@Entity`-annotated classes and JPA calls into SQL.
- Spring Data JPA is a repository abstraction built on top of JPA: you declare an interface, and it generates a working implementation, removing the boilerplate of using JPA /Hibernate directly.
- Spring Boot doesn't add a new layer — it auto-configures the `DataSource`, `EntityManagerFactory`, JPA provider, and repository infrastructure that connect the other three.

Cheat Sheet

```
Java object
     ↓
    JPA        (specification: @Entity, @Id, EntityManager, ...)
     ↓
 Hibernate      (implementation: generates the actual SQL)
     ↓
    SQL
     ↓
PostgreSQL

Repository
     ↓
Spring Data JPA (generates a working implementation of your interface)
     ↓
    JPA
     ↓
 Hibernate
     ↓
PostgreSQL
```

```
@Entity
public class Topic {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
}

interface TopicRepository extends JpaRepository<Topic, Long> {
    Optional<Topic> findBySlug(String slug); // no SQL written here
}
```

Glossary

- **ORM (Object-Relational Mapping)**: the general idea of automatically translating between Java objects and relational database rows/tables.
- **JPA (Jakarta Persistence API)**: a specification defining how object-relational mapping should work in Java — a contract, not a runtime implementation.
- **Hibernate**: a concrete implementation of the JPA specification; the most common one used in Spring Boot applications.
- **Spring Data JPA**: a repository abstraction built on top of JPA that generates a working implementation from a declared interface.
- **JPA provider**: the specific JPA implementation an application uses at runtime — Hibernate, in this project.