PostgreSQL and the Relational Model

What a relational database actually is, what PostgreSQL is specifically, the Spring Boot → Spring Data JPA → Hibernate → SQL → PostgreSQL stack, and how a call to this project's own real TopicRepository travels through it -- no SQL syntax yet, just the mental model. The 1st lesson in both the PostgreSQL course and its PostgreSQL Foundations category.

Beginner 15 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/postgresql-and-the-relational-model

---

This is the first lesson of the new PostgreSQL course, and the first lesson of its "PostgreSQL Foundations" category. You already know Java, Spring Boot, and Spring Data JPA — "JPA, Hibernate, and Spring Data JPA" even built a mental model of JPA (a specification) → Hibernate (an implementation) → Spring Data JPA (a repository abstraction). That model stopped one layer short of the database itself. This course starts exactly there: not at SQL syntax yet, but at what a relational database actually is, and what PostgreSQL specifically is — the mental model everything else in this course will be hung on.

What Is a Relational Database?

A relational database stores data as tables of rows and columns, and — the part that gives it its name — lets those tables REFER to one another through shared values, instead of duplicating data or nesting it inside a single giant structure. A `topic` row doesn't repeat its category's name and description inline; it stores a `category_id` that points at a row in a separate `category` table. "Relational" describes exactly this: data organized as separate, related tables, not data related by family or proximity in some looser sense.

Why Does It Exist?

Before this model became standard, applications typically stored data in whatever shape suited one specific program — a flat file, a custom binary format, data nested arbitrarily deep inside a single record. That worked until a second program needed the same data, or a piece of data needed to be found by something other than the exact way it was originally stored, or two pieces of data needed to stay consistent with each other as both changed. The relational model, and a database built to enforce it, solves that directly: data is stored once, in a shape that doesn't assume how it will be queried later, with the database itself — not each individual application — responsible for keeping related pieces consistent.

History

The relational model itself was proposed by Edgar F. Codd, an IBM researcher, in a 1970 paper — a genuinely new idea at the time, describing data purely in terms of tables and mathematical relations between them, independent of how it was physically stored on disk. PostgreSQL's own history traces back to a research project at UC Berkeley in the mid-1980s, led by Michael Stonebraker, explicitly named POSTGRES as a successor to an earlier Berkeley database (Ingres) — the name literally means "after Ingres." It gained SQL support and its current name, PostgreSQL, in the mid-1990s, and has been developed as an open-source project ever since.

What Is PostgreSQL, Specifically?

PostgreSQL is one specific, open-source implementation of a relational database — the same relationship "JPA, Hibernate, and Spring Data JPA" already covered for JPA and Hibernate (a specification vs. one of its implementations) shows up again here, one layer down: SQL itself is a broadly standardized language, and PostgreSQL is one real, running piece of software that implements it (alongside others — MySQL, Oracle Database, SQL Server). What sets PostgreSQL apart, and what makes it worth learning specifically rather than "just SQL in general," is covered progressively throughout this course — strict standards compliance, a genuinely extensible type system (covered in "PostgreSQL-Specific Data Types: UUID, JSON/JSONB, and Arrays" later in this course), and a specific, well-documented approach to transactions and concurrency (covered in full in "Transactions and Concurrency in PostgreSQL," the last lesson in this course). For now, the important fact is simpler: PostgreSQL is what actually stores and retrieves this platform's own data, right now, underneath every `TopicRepository` call this project's Java code makes.

Tables, Rows, and Columns: The Core Mental Model

A table is a named collection of records that all share the same shape — the same set of columns, each with its own name and type. A row is one single record within that table — one specific topic, one specific category. A column is one named, typed slot every row in that table has a value for (or explicitly has no value for, when the column allows it).

```
table: category
+----+-----------------------+--------------------+
| id | name                  | slug               |
+----+-----------------------+--------------------+
| 1  | Spring Data JPA       | spring-data-jpa    |
| 2  | Advanced Spring       | advanced-spring    |
+----+-----------------------+--------------------+

table: topic
+----+-------------+--------------------------------------+
| id | category_id | slug                                 |
+----+-------------+--------------------------------------+
| 1  | 1           | jpa-hibernate-and-spring-data-jpa    |
| 2  | 1           | entities-and-repositories            |
+----+-------------+--------------------------------------+
```

This is deliberately drawn from this project's own real schema, not an invented example — `topic`'s `category_id` column is exactly the "refer to another table by a shared value" idea "What Is a Relational Database?" described, with nothing invented for the sake of the example.

Where PostgreSQL Fits in a Spring Boot Application

Every layer between a Java method call and an actual stored row is worth naming explicitly, since each one hands off to the next.

```
Spring Boot
     ↓
Spring Data JPA
     ↓
Hibernate
     ↓
   SQL
     ↓
PostgreSQL
     ↓
Tables / Indexes / Constraints / Transactions
```

Spring Boot auto-configures the pieces above it (already covered in "Spring Boot Auto-Configuration & Properties" and referenced again in "JPA, Hibernate, and Spring Data JPA"). Spring Data JPA generates a working repository implementation from a declared interface. Hibernate turns a repository call into real SQL. SQL is the language that SQL statement is written in. PostgreSQL is the actual running database process that receives that SQL, executes it, and returns a result — and everything the rest of this course covers (tables, indexes, constraints, transactions) is what PostgreSQL itself is doing underneath that execution.

From a Repository Method to a Database Row

Making this concrete with a call this project's own code actually makes helps the whole stack diagram click into place at once.

```
topicRepository.findBySlug("records")
     ↓  (Spring Data JPA derives a query from the method name --
     ↓   see "Query Methods and JPQL with @Query")
Hibernate builds a SQL statement
     ↓
SELECT * FROM topic WHERE slug = 'records'
     ↓  (PostgreSQL receives and executes this SQL)
PostgreSQL scans the "topic" table, finds the matching row
     ↓
one row: {id: 8, category_id: 4, slug: 'records', ...}
     ↓
Hibernate maps that row back onto a real Topic object
     ↓
topicRepository.findBySlug("records") returns it
```

Nothing here is new mechanism — "JPA, Hibernate, and Spring Data JPA" already walked through this exact flow, for this exact method. What's new in THIS course is what happens starting at the `SELECT` line: PostgreSQL receiving that statement, deciding how to actually find the matching row (a subject "Indexes and Query Performance with EXPLAIN," later in this course, covers directly), and returning it.

ACID: A First Look

Four letters are worth knowing by name this early, even though none of them are taught in depth until "Transactions and Concurrency in PostgreSQL," the final lesson in this course: Atomicity (a group of changes either all happen or none do), Consistency (the database never ends up in a state that violates its own rules, like a foreign key pointing at a row that doesn't exist), Isolation (one transaction doesn't see another transaction's unfinished, uncommitted work), and Durability (once a change is committed, it survives a crash). PostgreSQL provides all four — this is one of the concrete answers to "why PostgreSQL specifically" from earlier in this lesson — but exactly how, and what that means practically for concurrent access, real locking, and Spring's own `@Transactional` (already covered in "Transaction Management"), is deliberately left for later, once real SQL and real concurrent scenarios exist to demonstrate it with.

Best Practices

- Keep the five-layer stack in mind whenever a Spring Data JPA operation feels opaque — every layer hands off to a specific, nameable next one, all the way down to PostgreSQL.

- Think of a table's columns as a shape every row must fit, not a loose bag of fields — this is what makes a relational database fundamentally different from storing arbitrary nested data.
- Resist the urge to duplicate data across tables "for convenience" — the whole point of `category_id` pointing at `category` instead of repeating its data is a single source of truth.
- When something goes wrong in a Spring Data JPA call, remember PostgreSQL itself is a real, inspectable system underneath it — later lessons in this course teach exactly how to look at what it's actually doing.

Common Mistakes

- Treating "the database" as an implementation detail Hibernate fully hides — it doesn't; this entire course exists because understanding PostgreSQL itself pays off even in a Spring-heavy codebase.
- Assuming "relational" means "related data" in a loose, everyday sense, rather than the specific idea of separate tables connected through shared column values.
- Confusing SQL (the language) with PostgreSQL (one specific program that runs SQL) — the same specification-vs-implementation distinction already covered for JPA and Hibernate.
- Expecting a full explanation of ACID, transactions, or locking this early — this lesson intentionally only names them; "Transactions and Concurrency in PostgreSQL" is where they're taught for real.

Summary, Cheat Sheet, and Glossary

Summary

- A relational database stores data as tables that refer to one another through shared column values, instead of duplicating or arbitrarily nesting data.
- PostgreSQL is one specific, open-source implementation of a relational database — the same specification-vs-implementation relationship already covered for JPA and Hibernate, one layer down.
- A table is a named collection of same-shaped rows; a row is one record; a column is one named, typed slot every row has a value for.
- The full stack from a Java method call to a stored row is: Spring Boot → Spring Data JPA → Hibernate → SQL → PostgreSQL → tables/indexes/constraints/transactions.
- ACID (Atomicity, Consistency, Isolation, Durability) is named here at a high level only — taught in full in "Transactions and Concurrency in PostgreSQL."

Cheat Sheet

```
Spring Boot
     ↓
Spring Data JPA        (repository.findBySlug("records"))
     ↓
Hibernate               (generates the SQL)
     ↓
   SQL                  (SELECT * FROM topic WHERE slug = 'records')
     ↓
PostgreSQL              (executes it, finds the row)
     ↓
Tables / Indexes / Constraints / Transactions
```

Glossary

- **Relational database**: a database storing data as tables that refer to one another through shared column values.
- **PostgreSQL**: one specific, open-source implementation of a relational database, built around SQL.
- **Table / row / column**: a named collection of same-shaped records / one record / one named, typed slot every row has a value for.
- **ACID**: Atomicity, Consistency, Isolation, Durability — the four guarantees a relational database provides for transactions, covered in full later in this course.