Databases, Schemas, Tables, and Basic SQL Syntax

The server → database → schema → table hierarchy, CREATE TABLE syntax read through this project's own real V1__init_schema.sql, column definitions/constraints, SQL comments, and a first distinction between DDL and DML. The 3rd lesson in the PostgreSQL Foundations category.

Beginner 20 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/databases-schemas-tables-and-basic-sql

---

"Connecting to PostgreSQL" got a real session open with `psql`, pointed at this project's own `learning` database. This lesson looks at what's actually inside that database — not through Hibernate, not through a `CREATE TABLE` written for this lesson, but through this project's own real, running `V1__init_schema.sql` migration, read as SQL for the first time.

Databases, Schemas, and Tables: The Hierarchy

A single PostgreSQL server can host many **databases** — "Connecting to PostgreSQL" already saw two of them side by side (`learning` and `learning_test`), completely isolated from each other. Inside one database, tables aren't flat — they live inside a **schema**, a named namespace of tables (and other objects) within that database. A database can have several schemas; a schema can have many tables. The full hierarchy is: **server → database → schema → table**.

This project never creates its own schema — every `CREATE TABLE` in every migration, including `V1__init_schema.sql`, implicitly lands in a schema called `public`, the one schema PostgreSQL creates automatically in every new database. That's why `\dt` in "Connecting to PostgreSQL" listed `topic` / `category` / `course` without ever mentioning a schema name — `\dt` defaults to showing the `public` schema, and for this project, `public` is the only schema that has ever mattered.

```
learning=# \dn
  List of schemas
  Name  |  Owner
--------+----------
 public | learning
```

`\dn` — a `psql` meta-command not yet seen — lists every schema in the current database. For most single-application projects, exactly like this one, that list has one row.

CREATE TABLE: Reading This Project's Own Real Schema

This project's very first migration, `V1__init_schema.sql`, is real, unmodified SQL — the same file Flyway has run against every environment this project has ever had, from the first commit to today:

```
CREATE TABLE course
(
    id   BIGSERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    slug VARCHAR(255) NOT NULL UNIQUE
);

CREATE TABLE category
(
    id        BIGSERIAL PRIMARY KEY,
    course_id BIGINT       NOT NULL REFERENCES course (id) ON DELETE CASCADE,
    name      VARCHAR(255) NOT NULL,
    slug      VARCHAR(255) NOT NULL,
    CONSTRAINT uq_category_course_slug UNIQUE (course_id, slug)
);
```

`CREATE TABLE <name> ( <column definitions>, <table-level constraints> );` is the whole shape of the statement — a name, then a parenthesized, comma-separated list. Nothing here is invented for this lesson: this is the exact table Spring Data JPA's `Course` entity is mapped onto, and it's the exact table `psql`'s `\d course` (from "Connecting to PostgreSQL") would describe.

Column Definitions: Type, Constraints, Defaults

Each line inside the parentheses is one column: a name, a data type, then zero or more constraints, read left to right.

`id BIGSERIAL PRIMARY KEY` — `BIGSERIAL` is a PostgreSQL convenience type that auto-generates an increasing 64-bit integer (full data type coverage, including why this project uses `BIGSERIAL` specifically, is the next lesson's job); `PRIMARY KEY` marks this column as the row's unique identifier. `name VARCHAR(255) NOT NULL` — a variable-length string, capped at 255 characters, that can never be `NULL`. `slug VARCHAR(255) NOT NULL UNIQUE` — the same type and not-null rule, plus a constraint that no two rows in this table may share the same value. `course_id BIGINT NOT NULL REFERENCES course (id) ON DELETE CASCADE` — a foreign key: `course_id` must match an existing `id` in the `course` table, and `ON DELETE CASCADE` says that deleting a `course` row automatically deletes every `category` row that references it. `CONSTRAINT uq_category_course_slug UNIQUE (course_id, slug)` is a table-level constraint (spanning more than one column) rather than a column-level one — it says a `slug` only has

to be unique *within* a given `course_id`, not across the whole table. Full constraint mechanics — why `PRIMARY KEY` and `FOREIGN KEY` exist as concepts, what `ON DELETE CASCADE`'s alternatives are — belong to "Constraints and Keys," two lessons ahead; this lesson's job is only to read the syntax fluently enough to recognize each piece.

Statement Terminators, Case Sensitivity, and Identifiers

Every SQL statement in this project's migrations ends with a semicolon (`;`) — PostgreSQL reads statements up to that character, not up to a newline, which is why a single `CREATE TABLE` can safely span many lines. SQL keywords (`CREATE TABLE`, `NOT NULL`, `REFERENCES`) are conventionally written in uppercase, as this project's migrations do, but PostgreSQL doesn't actually require it — `create table` works identically. Unquoted identifiers (table and column names) are automatically lowercased by PostgreSQL regardless of how they're typed, which is exactly why this project's migrations write every table and column name in lowercase already — matching what PostgreSQL would do anyway, rather than fighting it. Wrapping an identifier in double quotes (`"Course"`) preserves its exact case and makes it case-sensitive from then on — this project never does this, and mixing quoted and unquoted identifiers is a common source of confusing "table not found" errors, covered in Common Mistakes below.

Comments in SQL

`V1__init_schema.sql` has a two-line SQL comment directly above `topic_translation`:

```
-- NOT: "published" burada YOK, kasıtlı olarak topic_translation seviyesinde —
-- bir dilin yayında, diğerinin taslak olabilmesi için.
```

`--` starts a single-line SQL comment, running to the end of that line — unrelated to Java's `//`, but serving the identical purpose. SQL also supports block comments (`/* ... */`), though this project's migrations only ever use `--`. This particular comment records exactly the same fact CLAUDE.md itself states about `topic_translation`'s design — a real example of documenting a non-obvious schema decision directly next to the SQL that makes it.

DDL vs. DML: A First Distinction

`CREATE TABLE` belongs to a category of SQL called **DDL** (Data Definition Language) — statements that define or change the *structure* of a database: `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`. Every one of this project's Flyway migrations, including `V1__init_schema.sql`, is DDL. A separate category, **DML** (Data Manipulation Language) — `INSERT`, `UPDATE`, `DELETE`, `SELECT` — reads and writes *rows* within a structure that DDL already created. This project's migrations occasionally mix both: `V1__init_schema.sql` is pure DDL, while a migration like `V402__connecting_to_postgresql_topic.sql` (creating this course's own topics) is pure DML, `INSERT`ing rows into a `topic` table that DDL already defined back in `V1`. The next four

lessons in this category — inserting/updating/deleting, then `SELECT`, then sorting /pagination, then joins — are entirely DML; this is the last DDL-focused lesson until "Constraints and Keys."

Common Misconceptions

"A database and a schema are the same thing." They're not — a database is the top-level container `psql`'s `\l` lists and `\c` switches between; a schema is a namespace *inside* one database, which `\dn` lists. This project happens to have exactly one schema (`public`) per database, which makes the distinction easy to miss, but a single database can hold many schemas. **"SQL keywords must be uppercase."** They don't have to be — PostgreSQL is case-insensitive for keywords; uppercase is purely a widely followed readability convention, one this project's own migrations follow consistently. **"Table and column names are case-sensitive, like Java identifiers."** Only if quoted — unquoted identifiers are folded to lowercase automatically, so `Course`, `COURSE`, and `course` all refer to the identical table unless one of them was created with double quotes.

Best Practices

- Read a `CREATE TABLE` statement top to bottom as a list of independent column definitions, then separately look for table-level `CONSTRAINT` lines at the end — trying to parse both in one pass is what makes a real migration file feel harder to read than it is.
- Stick to lowercase, unquoted identifiers, exactly as this project's migrations do — it sidesteps case-sensitivity confusion entirely rather than requiring quoting discipline everywhere a name is used.
- Write a SQL comment (`--`) directly above a column or constraint whose purpose isn't obvious from its name alone — `V1__init_schema.sql`'s own comment above `topic_translation` is a real example worth imitating.
- When looking at an unfamiliar table for the first time, reach for `psql`'s `\d <table>` (from "Connecting to PostgreSQL") before reading the migration file — it shows the table's *current* structure, after every later migration has potentially altered it, which a single old `CREATE TABLE` statement alone cannot.

Common Mistakes

- Assuming a `NOT NULL` column also somehow prevents duplicate values, or that `UNIQUE` also somehow prevents `NULL` — they're independent constraints; a `UNIQUE` column can still hold multiple `NULL`s (PostgreSQL treats `NULL` as never equal to another `NULL`, including for uniqueness).
- Quoting an identifier inconsistently — creating a table as `"Course"` and later querying `course` (unquoted, and therefore folded to lowercase) produces a genuine "relation does not exist" error, not a typo-style near-miss.

- Forgetting the semicolon at the end of a statement when writing SQL interactively in `psql` — without it, `psql` simply waits for more input on a continuation prompt instead of running anything.
- Treating every SQL statement as DML by default and being surprised that a `CREATE TABLE` inside a transaction can be rolled back just like an `INSERT` — PostgreSQL, unlike some databases, supports transactional DDL, a detail "Transactions and Concurrency in PostgreSQL" returns to later in this course.

Summary, Cheat Sheet, and Glossary

Summary

- The full hierarchy is server → database → schema → table; this project has one schema (`public`) per database, which `\dn` lists.
- `CREATE TABLE <name> ( <columns>, <constraints> );` is DDL — it defines structure, not rows; `V1__init_schema.sql` is real DDL already running in this project.
- Each column definition reads left to right: name, data type, then constraints (`NOT NULL` , `UNIQUE` , `PRIMARY KEY` , `REFERENCES ... ON DELETE ...`) — table-level constraints (like a composite `UNIQUE`) are listed separately, after the columns.
- SQL is keyword-case-insensitive and folds unquoted identifiers to lowercase; `--` starts a single-line comment.
- DDL (`CREATE` / `ALTER` / `DROP`) defines structure; DML (`INSERT` / `UPDATE` / `DELETE` / `SELECT`) reads and writes rows within it — the next several lessons are entirely DML.

Cheat Sheet

```
-- Minimal CREATE TABLE shape
CREATE TABLE table_name
(
    column_name TYPE constraint constraint,
    CONSTRAINT constraint_name CONSTRAINT_TYPE (column, ...)
);

-- Single-line comment
-- like this one
```

```
\dn            -- list schemas in the current database (psql)
\dt            -- list tables in the current schema (psql, from "Connecting to PostgreSQL")
\d <table>     -- describe a table's columns (psql, from "Connecting to PostgreSQL")
```

Glossary

- **Schema**: a named namespace of tables (and other objects) inside one database; this project uses only the default `public` schema.
- **DDL (Data Definition Language)**: SQL that defines or changes structure — `CREATE TABLE` , `ALTER TABLE` , `DROP TABLE` .

- **DML (Data Manipulation Language)**: SQL that reads or writes rows — `INSERT`, `UPDATE`, `DELETE`, `SELECT`.
- **Identifier**: a table or column name; unquoted identifiers are folded to lowercase, quoted ones (`"Name"`) preserve exact case and become case-sensitive.
- **Table-level constraint**: a constraint (like a composite `UNIQUE`) listed as its own line rather than attached to a single column, because it spans more than one column.