PostgreSQL Data Types

INTEGER/BIGINT/BIGSERIAL, VARCHAR/TEXT, BOOLEAN, and DATE/TIMESTAMP /TIMESTAMPTZ -- how each one maps to its corresponding Java field type (Long/String /boolean/LocalDateTime) in this project's own real Topic/TopicTranslation/Question entities. The 4th lesson in the PostgreSQL Foundations category.

Beginner 20 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/postgresql-data-types

---

"Databases, Schemas, Tables, and Basic SQL Syntax" read `BIGSERIAL`, `VARCHAR(255)`, and `TIMESTAMP` in this project's own `CREATE TABLE` statements without stopping to ask what each type actually guarantees, or how it ends up as a `Long`, a `String`, or a `LocalDateTime` back in Java. This lesson answers that — using this project's own real columns, not invented ones.

Numeric Types: integer, bigint, and This Project's Own Choice of BIGSERIAL

PostgreSQL's two everyday whole-number types are `INTEGER` (4 bytes, roughly ±2.1 billion) and `BIGINT` (8 bytes, roughly ±9.2 quintillion). `BIGSERIAL`, used for every `id` column in `V1__init_schema.sql`, isn't a distinct storage type — it's `BIGINT` plus an automatically created sequence that generates the next value, which is exactly what backs `GenerationType.IDENTITY` on this project's own `Topic.id`:

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

`BIGINT` / `BIGSERIAL` maps to Java's `Long`, `INTEGER` / `SERIAL` to `Integer` — this project's `topic.estimated_minutes` column (plain `INTEGER`, no auto-generation) is exactly why `Topic.estimatedMinutes` is declared `Integer`, not `Long`: it's a small, human-entered number, never an identity column, so `BIGINT`'s extra range buys nothing. Choosing `BIGINT` / `BIGSERIAL` for every primary key, even in a table that will never approach a billion rows, is a defensive default worth keeping — changing a primary key's type later, after foreign keys reference it, is far more disruptive than the few extra bytes `BIGINT` costs up front.

Text Types: varchar vs. text

PostgreSQL has three string types, and the difference is smaller than it looks: `VARCHAR(n)` enforces a maximum length of `n` characters; `VARCHAR` with no length and `TEXT` are functionally identical, unlimited-length strings. Internally, PostgreSQL stores all three the same way and applies no performance penalty to `TEXT` over a length-limited `VARCHAR` — unlike some other databases, where `TEXT` is a slower, separately stored type.

This project's own columns show both choices used deliberately: `topic.slug VARCHAR(255)` (from `V1__init_schema.sql`) caps a value that's used in URLs and file paths, where an unbounded length would be a bug, not a feature; `topic_translation.summary`, by contrast, is mapped in Java with an explicit `columnDefinition`:

```
@Column(columnDefinition = "TEXT")
private String summary;
```

A lesson summary has no natural length ceiling, so `TEXT` is the honest choice — no arbitrary cap to bump into later. Both map to Java's `String`; the type alone never tells you which one you're looking at without checking `columnDefinition` or the migration itself.

boolean

`BOOLEAN` stores exactly `true`, `false`, or `NULL` — no `0`/`1` integer substitute, unlike some databases. This project's `topic_translation.published` column is a real, plain `BOOLEAN NOT NULL`, mapped straightforwardly to Java's primitive `boolean`:

```
@Column(nullable = false)
private boolean published;
```

Note the `NOT NULL` here isn't optional in practice — a primitive `boolean` field in Java can never hold `null`, so if the column allowed it, a `NULL` value read from the database would have nowhere valid to go. This is a first, concrete look at a pattern worth remembering generally: a `NOT NULL` column and a non-nullable Java type (a primitive, as opposed to a boxed `Boolean` or `Integer`) need to agree, or reading a row can fail in ways that have nothing to do with application logic.

Date and Time Types: date, timestamp, and timestamptz

PostgreSQL has three commonly used temporal types. `DATE` stores a calendar date only — no time component. `TIMESTAMP` (short for `TIMESTAMP WITHOUT TIME ZONE`) stores a date and time with no time zone attached at all — just a naive point in time, as written. `TIMESTAMPTZ` (`TIMESTAMP WITH TIME ZONE`) stores a point in time that PostgreSQL always normalizes to UTC internally, converting to and from whatever time zone the connecting client is in.

This project's own `question` table (added in a later migration, after `V1__init_schema.sql`) uses plain `TIMESTAMP`:

```
ALTER TABLE question
    ADD COLUMN created_at TIMESTAMP NOT NULL DEFAULT now(),
    ADD COLUMN updated_at TIMESTAMP NOT NULL DEFAULT now();
```

mapped to `java.time.LocalDateTime` — a type that, not coincidentally, also carries no time zone of its own:

```
@Column(name = "created_at", nullable = false)
private LocalDateTime createdAt;
```

`LocalDateTime` for `TIMESTAMP`, and (were this project to use it) `java.time.Instant` or `OffsetDateTime` for `TIMESTAMPTZ`, is the natural pairing in each direction — mixing them (mapping a `TIMESTAMPTZ` column to `LocalDateTime`, say) silently discards the time zone information PostgreSQL was tracking. `now()`, seen above, is a PostgreSQL built-in function returning the current transaction's timestamp — not a value supplied by the application, which is why `created_at` / `updated_at` get a real default even for rows inserted directly by SQL, independent of any Java code path.

Reading a Real Column's Type with psql

`\d <table>` (already used in "Connecting to PostgreSQL" and "Databases, Schemas, Tables, and Basic SQL Syntax") is the fastest way to check a column's actual type without opening a migration file at all:

```
learning=# \d topic_translation
                 Table "public.topic_translation"
      Column      |          Type          | Collation | Nullable | Default
------------------+------------------------+-----------+----------+---------
 id               | bigint                 |           | not null |
 topic_id         | bigint                 |           | not null |
 language         | character varying(5)   |           | not null |
 title            | character varying(255) |           | not null |
 summary          | text                   |           |          |
 seo_title        | character varying(255) |           |          |
 seo_description  | character varying(500) |           |          |
 published        | boolean                |           | not null |
```

Note that `psql` reports `character varying(255)` rather than `VARCHAR(255)` and `bigint` rather than `BIGINT` — PostgreSQL's internal type names are lowercase and occasionally longer than the SQL keyword used to declare them; both refer to the identical type.

From SQL Type to Java Field: How Hibernate Bridges the Gap

Every mapping shown above — `BIGINT` ↔ `Long`, `VARCHAR` / `TEXT` ↔ `String`, `BOOLEAN` ↔ `boolean`, `TIMESTAMP` ↔ `LocalDateTime` — is applied by Hibernate automatically, without a single explicit type-conversion annotation anywhere in `Topic`, `TopicTranslation`, or `Question`. This is a direct consequence of the JDBC driver and Hibernate dialect layer that "JPA, Hibernate, and Spring Data JPA" already covered — this lesson isn't introducing new machinery, only naming, concretely, which PostgreSQL type each of this project's real Java field types corresponds to on the other side of that layer. The one place this project does make a mapping explicit rather than relying on inference is exactly the `TEXT` case above (`columnDefinition = "TEXT"`) — because Hibernate's own default for a bare `String` field is a length-capped `VARCHAR`, not `TEXT`, so an unbounded column has to be requested deliberately.

Common Misconceptions

"`VARCHAR(255)` is faster than `TEXT`." Not in PostgreSQL — internally identical, with identical performance; the length limit is purely a data-integrity choice, not a performance one. **"A `BOOLEAN` column can be `0` or `1`."** No — PostgreSQL's `BOOLEAN` is a real three-valued type (`true` / `false` / `NULL`); `0` / `1` are integers, a different type entirely, even though some client libraries accept them as loose input. **"`TIMESTAMP` includes time zone handling."** It's the opposite — plain `TIMESTAMP` explicitly has no time zone; `TIMESTAMPTZ` is the one that does, despite `TIMESTAMP` being the more commonly reached-for name.

Best Practices

- Default to `BIGINT` / `BIGSERIAL` for primary keys even when a table will stay small — this project does so uniformly in `V1__init_schema.sql`, and it costs nothing meaningful up front while avoiding a disruptive later migration.
- Prefer `TEXT` for genuinely unbounded content (like `topic_translation.summary`) and reserve `VARCHAR(n)` for values with a real, meaningful upper bound (like a `slug` used in a URL) — let the length limit express a business rule, not a guess.
- Match a `NOT NULL` column to a Java primitive (`boolean`, `int`, `long`) and a nullable column to the corresponding boxed type (`Boolean`, `Integer`, `Long`) — this project's `published` field is a clean example of the primitive side of that rule.
- Choose `TIMESTAMPTZ` over plain `TIMESTAMP` for any new time-tracking column in a project that might ever run across time zones — treat this project's own use of plain `TIMESTAMP` for `question.created_at` / `updated_at` as a real, existing trade-off worth being aware of, not a pattern to copy without thinking about it.

Common Mistakes

- Assuming `VARCHAR` without a length behaves differently from `TEXT` — they don't, in PostgreSQL specifically; carrying that assumption over from another database leads to unnecessary micro-optimization.
- Mapping a `TIMESTAMPTZ` column to `LocalDateTime` (or a `TIMESTAMP` column to `Instant` / `OffsetDateTime`) — both compile and often even run without error, but silently lose or fabricate time zone information at read or write time.
- Forgetting that a Java primitive field cannot hold a `NULL` read from a nullable column — the failure surfaces as an unchecked exception at the Hibernate/JDBC layer, far from the column definition that actually caused it.
- Picking `INTEGER` for a primary key to "save space," then hitting real range limits (or a disruptive type-widening migration) once the table grows — the bytes saved rarely justify the risk for an `id` column specifically.

Summary, Cheat Sheet, and Glossary

Summary

- `INTEGER` / `BIGINT` map to Java's `Integer` / `Long`; `BIGSERIAL` is `BIGINT` plus an auto-incrementing sequence, backing `GenerationType.IDENTITY` on this project's own primary keys.
- `VARCHAR(n)`, unbounded `VARCHAR`, and `TEXT` are stored identically in PostgreSQL and all map to Java's `String` — only the length cap differs, and it should express a real constraint, not a performance choice.
- `BOOLEAN` is a genuine three-valued type (`true` / `false` / `NULL`), mapping cleanly to a Java primitive `boolean` only when the column is `NOT NULL`.
- `TIMESTAMP` carries no time zone and maps to `LocalDateTime`; `TIMESTAMPTZ` does carry one (normalized to UTC internally) and maps to `Instant` / `OffsetDateTime` — this project's own `question` table uses plain `TIMESTAMP`.
- Every SQL-to-Java type mapping shown here is handled automatically by Hibernate's JDBC dialect layer (already covered in "JPA, Hibernate, and Spring Data JPA") — no explicit conversion code exists anywhere in this project's own entities.

Cheat Sheet

```
INTEGER / SERIAL     ↔ Integer
BIGINT  / BIGSERIAL   ↔ Long
VARCHAR(n) / TEXT     ↔ String
BOOLEAN               ↔ boolean (if NOT NULL) / Boolean (if nullable)
DATE                  ↔ LocalDate
TIMESTAMP             ↔ LocalDateTime
TIMESTAMPTZ           ↔ Instant / OffsetDateTime
```

```
\d <table>     -- see a table's real column types (psql)
```

Glossary

- **BIGSERIAL**: `BIGINT` plus an auto-incrementing sequence PostgreSQL creates and manages implicitly — the SQL-level mechanism behind `GenerationType.IDENTITY`.
- **TEXT**: an unbounded-length string type, stored identically to `VARCHAR` internally in PostgreSQL.
- **TIMESTAMPTZ**: a timestamp type that tracks time zone information, normalized to UTC internally and converted for the connecting client.
- **now()**: a PostgreSQL built-in function returning the current transaction's timestamp, usable directly as a column `DEFAULT`.