Constraints and Keys

PRIMARY KEY, FOREIGN KEY and referential integrity, ON DELETE CASCADE vs. RESTRICT (with a real contrast from this project's own quiz_question_link table), composite UNIQUE, CHECK constraints, and how BIGSERIAL PRIMARY KEY connects to GenerationType.IDENTITY. The 5th lesson in the PostgreSQL Foundations category.

Beginner 20 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/constraints-and-keys

---

"Databases, Schemas, Tables, and Basic SQL Syntax" already used `PRIMARY KEY`, `REFERENCES ... ON DELETE CASCADE`, `NOT NULL`, and `UNIQUE` while reading `V1__init_schema.sql`, but only as syntax to recognize, with the full mechanics explicitly deferred to this lesson. This is where each one is taken apart properly — what it actually enforces, what happens when it's violated, and where this project's own real schema shows more than one option in practice.

PRIMARY KEY: What It Actually Enforces

`PRIMARY KEY` is really two constraints bundled together: `NOT NULL` (a primary key column can never be empty) plus `UNIQUE` (no two rows can share the same value) — and PostgreSQL automatically builds an index on it, which "Indexes and Query Performance," later in this course, explains the significance of. A table can have at most one primary key, though that key can span multiple columns (a composite primary key) — this project doesn't use one anywhere, since every table's identity is a single generated `id` column, but the option exists for cases where a row's identity is naturally a combination of values rather than a single generated number.

FOREIGN KEY and Referential Integrity

A `FOREIGN KEY` (introduced with `REFERENCES`) constrains a column's values to only those that already exist as a primary key (or a unique column) in another table. This is what PostgreSQL calls **referential integrity**: it's not possible, at the database level, for `category.course_id` to hold a value that doesn't correspond to a real row in `course` — not because application code checks it, but because PostgreSQL itself rejects the `INSERT` or `UPDATE` outright. Trying to insert a `category` row with a `course_id` of `9999` when no such course exists fails with a real, specific error (`violates foreign key constraint`), regardless of what Java code did or didn't validate beforehand.

ON DELETE Behavior: CASCADE vs. RESTRICT, a Real Contrast in This Project

A foreign key alone doesn't say what should happen when the row it points to is deleted — that's what `ON DELETE` specifies, and this project's own schema uses two different answers, deliberately, for two different reasons.

`V1__init_schema.sql` uses `ON DELETE CASCADE` throughout the content hierarchy:

```
category_id BIGINT NOT NULL REFERENCES category (id) ON DELETE CASCADE
```

Deleting a `category` automatically deletes every `topic` that references it, which cascades further into every `topic_translation` and `code_example` — deleting one row intentionally deletes an entire dependent subtree, which is the right behavior here because a `topic` genuinely has no meaning without its `category`.

This project's `quiz_question_link` table, added much later, makes the opposite choice on purpose, with its own real migration comment explaining why:

```
-- question_id: ON DELETE RESTRICT KASITLI -- bir soru, canlı bir sabit quiz'in
-- parçası olduğu sürece hard-delete edilemez.
quiz_id     BIGINT NOT NULL REFERENCES quiz (id) ON DELETE CASCADE,
question_id BIGINT NOT NULL REFERENCES question (id) ON DELETE RESTRICT
```

`ON DELETE RESTRICT` does the opposite of `CASCADE` — it *blocks* the delete outright: as long as a `question` is linked into any published quiz, attempting `DELETE FROM question WHERE id = ...` fails with an error instead of silently removing the link along with it. Note that the same table uses `CASCADE` for `quiz_id` (deleting a `Quiz` should take its own links with it) and `RESTRICT` for `question_id` (deleting a `Question` should not silently corrupt a quiz that depends on it) — the two columns intentionally disagree, because the two relationships mean different things. A third option, `ON DELETE SET NULL`, sets the foreign key column to `NULL` instead of deleting or blocking — not used anywhere in this project, since every foreign key here is `NOT NULL` and therefore can't hold `NULL` to begin with.

NOT NULL and UNIQUE (Single-Column and Composite)

`NOT NULL` and `UNIQUE` were already read as syntax in "Databases, Schemas, Tables, and Basic SQL Syntax"; the one addition worth making here is the distinction between a single-column and a composite version of each. A single-column `UNIQUE`, like `course.slug UNIQUE`, forbids any two rows in the whole table from sharing that value. A composite (table-level) `UNIQUE`, like this project's real

```
CONSTRAINT uq_category_course_slug UNIQUE (course_id, slug)
```

forbids duplicates only across the *combination* of both columns — two different courses are free to each have a category with the slug `fundamentals`, but the same course can't have two. `NOT NULL` doesn't have a composite form — it's always evaluated one column at a time — but it very often shows up alongside a composite `UNIQUE`, exactly as it does here, since a composite uniqueness check on a column that could be `NULL` runs into the `NULL`-related surprise covered below.

CHECK Constraints

A `CHECK` constraint enforces an arbitrary boolean condition on a column's value, evaluated on every `INSERT` and `UPDATE` — for example, `estimated_minutes INTEGER CHECK (estimated_minutes > 0)` would reject any row where that value is zero or negative. This project's actual `topic.estimated_minutes` column has no `CHECK` constraint today — it's a real, honest gap: nothing at the database level currently stops a migration from inserting a negative value; only application-level care (and, so far, correct migrations) has kept every row valid. It's a fair illustration of what a `CHECK` constraint is *for*: turning an assumption that currently holds only by convention into one PostgreSQL itself refuses to let a row violate.

Naming Constraints Explicitly

`PRIMARY KEY`, `NOT NULL`, and a single-column `UNIQUE`/`REFERENCES` all get an automatically generated name if none is given, which is why this project's migrations never bother naming those. Table-level constraints spanning more than one column are different — this project always names them explicitly, as in `CONSTRAINT uq_category_course_slug UNIQUE (course_id, slug)` — because an unnamed composite constraint gets an auto-generated name (like `category_course_id_slug_key`) that's harder to recognize in an error message or a later `ALTER TABLE ... DROP CONSTRAINT` than a deliberately chosen one.

From SERIAL/IDENTITY to GenerationType.IDENTITY

"PostgreSQL Data Types" already covered `BIGSERIAL` as `BIGINT` plus an auto-incrementing sequence, and "Entities and the Repository Abstraction" already covered `GenerationType.IDENTITY` on the Java side — the two are the same mechanism, described from opposite ends of the JDBC boundary, and nothing new needs repeating here. What's worth adding now, with `PRIMARY KEY` fully understood, is the complete picture: `BIGSERIAL PRIMARY KEY` is really three things layered together — a `BIGINT` column, a sequence generating its default value, and a `PRIMARY KEY` constraint making that value both `NOT NULL` and unique. A modern alternative syntax, `GENERATED ALWAYS AS IDENTITY`, does the same job with slightly different semantics (it refuses an explicit value on `INSERT` by default, where `BIGSERIAL` allows one) — this project uses `BIGSERIAL` throughout, and the distinction between the two identity strategies matters more once UUID primary keys enter the picture, which "PostgreSQL-Specific Data Types," later in this course, returns to.

Common Misconceptions

"A foreign key automatically deletes related rows." Only if `ON DELETE CASCADE` says so — the default, with no `ON DELETE` clause at all, is effectively `RESTRICT`: the delete is blocked. **"`UNIQUE` also implies `NOT NULL`."** It doesn't — a `UNIQUE` column can hold multiple `NULL` rows, since PostgreSQL never considers one `NULL` equal to another `NULL`, including for uniqueness checks; that's exactly why `NOT NULL` and `UNIQUE` are written as two separate constraints rather than one covering both. **"A composite `UNIQUE (a, b)` is the same as two separate `UNIQUE` constraints on `a` and `b`."** Not remotely — a composite constraint only rejects a duplicate of the *combination*; two separate single-column constraints would each reject duplicates of that column alone, a much stricter rule.

Best Practices

- Choose `ON DELETE CASCADE` only when a child row genuinely has no meaning without its parent (like `topic` without `category`); choose `RESTRICT` when a delete should force a conscious decision instead of silently rippling — this project's own `quiz_question_link` table makes both choices correctly, for different columns, in the same `CREATE TABLE`.
- Name every multi-column constraint explicitly (`CONSTRAINT name UNIQUE (...)`), following this project's own `uq_category_course_slug` pattern — it turns a future error message or `DROP CONSTRAINT` into something readable instead of an auto-generated identifier.
- Reach for a `CHECK` constraint whenever a column's validity depends only on its own value (like a minimum for `estimated_minutes`) rather than relying purely on application code, which every future direct SQL `INSERT` — including from a migration — would otherwise bypass.
- Treat a composite `UNIQUE` and `NOT NULL` as a pair worth reviewing together — a nullable column inside a composite unique constraint can silently defeat the uniqueness check for exactly the rows where it matters least to notice.

Common Mistakes

- Assuming the absence of an `ON DELETE` clause means "nothing happens" — it means the delete itself is blocked, which surfaces as a constraint-violation error the first time it's attempted, often in production rather than during development.
- Relying on `UNIQUE` alone to guarantee a value is present, then being surprised that several rows all have `NULL` in that "unique" column simultaneously.
- Writing a `CHECK` constraint that references another table's data (`CHECK` conditions can only see the current row) instead of realizing that cross-row or cross-table validation needs a trigger or application-level check instead.
- Letting an auto-generated constraint name (from an unnamed composite constraint) leak into a user-facing error message without translating it — a name like `category_course_id_slug_key` means little without the schema in front of you.

Summary, Cheat Sheet, and Glossary

Summary

- `PRIMARY KEY` bundles `NOT NULL` plus `UNIQUE` and adds an automatic index; a table has at most one, but it can span multiple columns.
- `FOREIGN KEY` (`REFERENCES`) enforces referential integrity at the database level, independent of application code; `ON DELETE CASCADE` / `RESTRICT` / `SET NULL` decide what happens to dependent rows — this project's real `quiz_question_link` table uses `CASCADE` and `RESTRICT` side by side, deliberately.
- `UNIQUE` has single-column and composite (table-level, multi-column) forms; a composite `UNIQUE` only rejects duplicates of the full combination, not of either column alone.
- `CHECK` enforces an arbitrary condition on a single row's own values — this project's `estimated_minutes` column is a real example of a column that currently has none.
- `BIGSERIAL PRIMARY KEY` layers three things (a `BIGINT` column, a sequence, and a `PRIMARY KEY` constraint) — the same mechanism as `GenerationType.IDENTITY`, already covered on the Java side in "Entities and the Repository Abstraction."

Cheat Sheet

```
id         BIGSERIAL PRIMARY KEY,
parent_id  BIGINT NOT NULL REFERENCES parent (id) ON DELETE CASCADE,
child_id   BIGINT NOT NULL REFERENCES child (id) ON DELETE RESTRICT,
slug       VARCHAR(255) NOT NULL UNIQUE,
minutes    INTEGER CHECK (minutes > 0),
CONSTRAINT uq_parent_slug UNIQUE (parent_id, slug)
```

Glossary

- **Referential integrity**: the database-enforced guarantee that a foreign key column only ever holds values that exist as a real row elsewhere.
- **ON DELETE CASCADE**: deleting the referenced row automatically deletes every row that references it.
- **ON DELETE RESTRICT**: deleting the referenced row is blocked outright while any row still references it.
- **Composite constraint**: a `UNIQUE` , `PRIMARY KEY` , or `CHECK` that spans more than one column, evaluated against the combination rather than each column independently.
- **CHECK constraint**: a boolean condition on a single row's own values, enforced by PostgreSQL on every insert and update.