PostgreSQL Course

Flavio Copes

Updated August 3, 2026

Contents

Learn PostgreSQL from installation to production basics: psql, roles, databases, schemas, types, indexes, application pools, migrations, backups, and debugging.

What you'll learn: Build and operate a small PostgreSQL-backed application with clear roles, reliable migrations, bounded connections, useful indexes, and a restorable backup.

Start using PostgreSQL

Install PostgreSQL, connect with psql, and navigate a server confidently.

What PostgreSQL is

Understand the PostgreSQL client-server model and the PostgreSQL 18 baseline used throughout this course.

PostgreSQL is a relational database server. It stores related data in tables, and you work with that data through SQL.

The word **server** matters. PostgreSQL is not a file your application opens, the way SQLite is. It is a long-running process. The server owns the data files, manages concurrent work, and listens for connections from applications and tools.

That design is what makes PostgreSQL safe under load. Twenty application processes can read and write at the same time, and the server coordinates them. It enforces constraints, isolates transactions, and recovers cleanly after a crash.

Three things people mix up

When you start, three names blur together. Keep them separate:

- **PostgreSQL** is the server. It runs in the background and does the actual work.
- **SQL** is the language you send to PostgreSQL. `SELECT`, `INSERT`, `CREATE TABLE` are SQL.
- `psql` is one command-line client that sends SQL to the server. It is not the server. It is one program that talks to it.

Your Node.js application is another client. A GUI like TablePlus is a third. They all speak the same protocol to the same server.

One server, several databases

One server can hold several databases. A connection enters one database as one role, then works with tables and other objects inside that database.

You can see this from any client:

```
SELECT current_database(), current_user;
```

```
 current_database | current_user
------------------+--------------
```

```
 postgres         | flavio
```

Every statement you run happens in that context: this database, this role.

The version baseline

This course targets PostgreSQL 18. PostgreSQL ships a new major version about once a year and supports each one for five years.

Check what a server actually runs before trusting a tutorial written for an older release:

```
SELECT version();
```

```
 PostgreSQL 18.1 on aarch64-apple-darwin24.4.0, compiled by ...
```

A common early confusion: `psql --version` in your terminal reports the **client** version, not the server. The two can differ, for example when your laptop's `psql` connects to a hosted database. `SELECT version();` asks the server itself, so trust that one.

Install PostgreSQL 18 on macOS

Install the PostgreSQL 18 Homebrew formula without treating a major-version upgrade like an ordinary package update.

On macOS, install the PostgreSQL 18 formula with Homebrew:

```
brew install postgresql@18
brew services start postgresql@18
```

The first command installs the server and its client tools. The second registers PostgreSQL as a background service, so it starts now and again after every reboot. Homebrew also creates the initial data directory and a superuser role named after your macOS user, which is why you can connect right away without a password.

Use the official PostgreSQL downloads for Windows or Linux. The rest of the course works the same on every platform once the server runs.

Put the tools on your PATH

The versioned formula is keg-only, so Homebrew does not link `psql` into your default path. Add the formula's `bin` directory to your shell path if Homebrew asks you to:

```
echo 'export PATH="/opt/homebrew/opt/postgresql@18/bin:$PATH"' >> ~/.zshrc
```

Open a new terminal, then verify both programs:

```
psql --version
pg_config --version
```

Both commands should report PostgreSQL 18. If you see `psql: command not found`, the path line is missing or the shell has not reloaded it yet.

Verify the server is running

Check the service state:

```
brew services list
```

```
Name           Status  User    File
postgresql@18  started flavio  ~/Library/LaunchAgents/homebrew.mxcl.postgresql@18.plist
```

`started` in green means the server is up. `error` means it failed to boot; look at the log file under `/opt/homebrew/var/log/` before trying anything else.

Upgrades are not all equal

A minor update inside PostgreSQL 18, say 18.1 to 18.2, is an ordinary package update. It fixes bugs and security issues without touching your data format.

A future move from 18 to 19 is a database-cluster upgrade. The on-disk data layout can change between major versions, so the new server cannot just open the old files. Back up first and use `pg_upgrade`, dump and restore, or logical replication.

Do not uninstall the old server and hope its data follows the new package. That is the classic way people end up with a running PostgreSQL 19 and an invisible, orphaned PostgreSQL 18 data directory. The versioned formula name, `postgresql@18`, exists exactly so a casual `brew upgrade` cannot move you across a major version by accident.

Connect with psql

Open a PostgreSQL command-line session and identify the server, database, and role used by the connection.

`psql` is the standard PostgreSQL command-line client. You will use it through the whole course, so let's make the first connection deliberate instead of lucky.

Start with the maintenance database that PostgreSQL creates:

```
psql postgres
```

That single argument is the database name. Everything else falls back to defaults: the local Unix socket as the host, port 5432, and your operating-system username as the role. On a fresh Homebrew install those defaults work, which is convenient and also why many people never learn what they connected to.

Know what you connected to

Inside `psql`, inspect the real connection:

```
\conninfo
SELECT version();
```

```
You are connected to database "postgres" as user "flavio"
via socket in "/tmp" at port "5432".
```

`\conninfo` answers three questions at once: which database, which role, which server. `SELECT version();` tells you what the server is running. Make checking both a habit. Most ”my table disappeared” panics are really ”I am connected to the wrong database”.

A complete connection can specify each part explicitly:

```
psql -h localhost -p 5432 -U flavio -d postgres
```

`-h` forces a TCP connection instead of the socket, `-U` picks the role, `-d` picks the database. You will need these flags the first time you connect to a remote server, so it pays to try them locally first.

Type `\q` to leave the session.

The most common first-day error

Run `psql` with no arguments and you may see:

```
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed:
FATAL:  database "flavio" does not exist
```

Nothing is broken. Without a database name, `psql` tries a database named after your role, and nobody created one called `flavio`. Name the database and the error goes away: `psql postgres`.

One more distinction to keep sharp: the `psql` client is not the server. It is one program that talks to it. Closing `psql` does not stop PostgreSQL, and updating `psql` does not update the server.

Navigate psql

Use psql meta-commands to inspect the current connection without confusing them with SQL.

Inside `psql` you type two different languages, and telling them apart saves you a lot of confusion.

SQL statements such as `SELECT current_database();` go to the server. They end with a semicolon.

Commands beginning with a backslash belong to `psql` itself. They are called **meta-commands**, they run locally, and they need no semicolon. Try `\conninfo`, `\l`, `\dt`, `\d notes`, and `\q`.

The meta-commands you need first

A handful covers daily work:

```
\conninfo   current database, role, host, and port
\l          list databases
\dt         list tables in the current search path
\d notes    describe the notes table: columns, types, indexes
\du         list roles
\q          quit
```

`\d` is the one you will run most. Point it at a table and it shows every column, its type, its constraints, and the indexes on the table:

```
\d notes
```

```
                        Table "public.notes"
 Column |  Type  | Collation | Nullable |           Default
--------+--------+-----------+----------+-------------------------------
 id     | bigint |           | not null | generated by default as identity
 title  | text   |           | not null |
```

When the prompt looks stuck

Forget the semicolon on a SQL statement and `psql` waits for more input:

```
postgres=# SELECT current_database()
postgres-#
```

The prompt changed from `=#` to `-#`. That means the statement is not finished. Type `;` and press enter to run it, or `\r` to throw the unfinished statement away. This is the single most common way

beginners think `psql` froze.

The reverse mistake is harmless but worth knowing: meta-commands take effect immediately, so `\dt` runs the moment you press enter, semicolon or not.

Getting help without leaving the session

Two help commands cover both languages. Use `\?` for psql help: it lists every meta-command. Use `\h SELECT` for SQL help: it shows the syntax of the `SELECT` statement straight from the documentation. Swap in any SQL command name, like `\h CREATE TABLE`.

My advice is to keep one `psql` session open while you work through this course and check every change with a meta-command right after you make it.

List PostgreSQL databases

List databases with psql and recognize the maintenance and template databases in a new cluster.

Inside `psql`, list the databases with:

```
\l
```

```
                                 List of databases
   Name    | Owner  | Encoding | Locale | ... | Access privileges
-----------+--------+----------+--------+-----+-------------------
 notes_app | flavio | UTF8     | C      |     |
 postgres  | flavio | UTF8     | C      |     |
 template0 | flavio | UTF8     | C      |     | =c/flavio
 template1 | flavio | UTF8     | C      |     | =c/flavio
```

A brand-new cluster already contains three databases, and none of them is yours.

`postgres` is the normal maintenance database. It exists so administrators and tools always have somewhere to connect, even before any application database exists. That is why the earlier lessons started with `psql postgres`.

The other two are **templates**. PostgreSQL never creates a database from nothing: `CREATE DATABASE` clones an existing one. By default it copies `template1`, so anything you add to `template1` shows up in every database you create afterwards. `template0` stays clean for restores and for databases that need different locale settings.

That default has a sharp edge. Connect to `template1` by accident, create a table there, and every future database silently inherits it. If `\l` shows objects you never asked for in new databases, inspect `template1`.

Sizes and the SQL alternative

Use `\l+` when you also need sizes and tablespaces:

```
\l+
```

The extra columns show how much disk each database uses, which is the quickest way to spot the database that grew unexpectedly.

`\l` is a psql meta-command, so it only exists inside `psql`. From application code or another client, query the catalog directly:

```
SELECT datname FROM pg_database
WHERE datistemplate = false;
```

```
  datname
-----------
 postgres
 notes_app
```

The `datistemplate = false` filter hides `template0` and `template1`, matching what you usually mean by "my databases". Drop the filter to see everything, templates included.

Listing databases is also your verification step after `CREATE DATABASE` or a restore: if the new name does not appear in `\l`, the command did not do what you thought, and the next thing to check is which server and cluster you are connected to with `\conninfo`.

Switch PostgreSQL databases

Reconnect psql to another database and verify that the new connection uses the intended role.

A connection enters one database. There is no `USE database` that quietly changes context inside a session, the way MySQL does it. To move from `postgres` to `notes_app`, reconnect:

```
\connect notes_app
```

```
You are now connected to database "notes_app" as user "flavio".
```

`\c` is the short form of `\connect`, and you will see it in most documentation and tutorials.

You can also choose a role while switching:

```
\connect notes_app notes_app
```

The first argument is the database, the second is the role. This is how you test what your application role can actually see and do, without leaving your admin session behind for good.

It is a new connection, not a setting

Run `\conninfo` after switching. PostgreSQL opened a new connection; it did not change a database setting inside the old one.

That distinction has practical consequences. Session state does not travel across `\connect`. Anything you configured with `SET`, any open transaction, any temporary table: gone, because the session it lived in is gone. If you `SET search_path` and then switch databases, set it again.

When the switch fails

Point `\connect` at a database that does not exist and you get:

```
connection to server ... failed: FATAL:  database "notes_ap" does not exist
Previous connection kept
```

Read the last line. `psql` keeps your previous connection alive, so you are still connected to the old database. People miss that line, assume the switch happened, and run their next statements against the wrong database. After any failed `\connect`, run `\conninfo` before typing anything else.

The other failure you will meet once you lock down permissions in the next module:

```
FATAL:  permission denied for database "notes_app"
```

```
DETAIL:  User does not have CONNECT privilege.
```

That one is not a typo. The database exists, but the role you chose is not allowed in. It means your grants are working; connect as a role that holds `CONNECT` on that database instead.

Make the habit boring: switch, then verify with `\conninfo`, then work.

Check your understanding: PostgreSQL basics

Check the server and client roles, installation, connections, psql commands, database listing, and database switching.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Roles, databases, and schemas

Understand PostgreSQL organization and grant deliberate access.

Clusters, databases, and schemas

Understand PostgreSQL's hierarchy so a role, database, schema, and table do not blur into one concept.

PostgreSQL organizes everything in a strict hierarchy, and half of the confusing errors in this course dissolve once you can place each name on the right level.

One PostgreSQL server instance manages a **cluster** of databases. The cluster is the data directory plus the running server process: one port, one set of roles, one configuration. Everything Homebrew created when you installed PostgreSQL is one cluster.

A **database** lives inside the cluster. A connection enters one database at a time. You cannot join a table in `notes_app` with a table in another database from a plain connection; that needs extensions like `postgres_fdw`, which is a deliberate extra step, not the default.

Inside a database, **schemas** group objects such as tables and functions. A schema is a namespace, nothing more. `public` is the familiar default schema, not a separate database. When you write `app.notes`, you are naming schema `app` and table `notes` inside the current database.

A **table** belongs to one schema inside one database. The same table name can exist in several schemas without conflict.

Where roles fit

Roles break the nesting, and that surprises people. Roles belong to the **cluster**, so the same role name is visible from every database. You create a role once, then grant it different permissions in each database.

This is why `\du` shows the same list no matter which database you are connected to, and why dropping a database does not drop the roles that used it.

Verify each level yourself

The inspection commands map straight onto the hierarchy:

```
\l          -- databases in the cluster
\dn         -- schemas in the current database
\dt app.*   -- tables in one schema
\du         -- roles, shared by the whole cluster
```

And in plain SQL:

```
SELECT current_database(), current_schema(), current_user;
```

```
 current_database | current_schema | current_user
------------------+----------------+--------------
 notes_app        | public         | flavio
```

The classic mistake is creating a table in the wrong container: right database, wrong schema, or wrong database entirely. The table then ”does not exist” when the application looks for it. Before you blame the query, run the statement above and check which database and schema you are actually in. We will build this course's own `notes_app` database, `app` schema, and roles in the next lessons.

Separate owner and runtime roles

Use a non-login owner for migrations and a restricted login role for the running application.

PostgreSQL uses roles for both users and groups. There is no separate ”user” concept: a **role** with the `LOGIN` attribute can authenticate, and a role without it can own objects without becoming an application credential. `CREATE USER` is just `CREATE ROLE` with `LOGIN` added.

We will use that flexibility to solve a real problem. If the role your application connects with also owns every table, then anyone who steals the application's credentials can drop your schema. Ownership and runtime access do not need to travel together.

Two roles, two jobs

Create one role that owns the schema, and another that runs the application:

```
CREATE ROLE notes_owner NOLOGIN;
CREATE ROLE notes_app LOGIN;
```

`notes_owner` cannot log in at all. Try `psql -U notes_owner` and the server answers `FATAL: role "notes_owner" is not permitted to log in`. That is the point: an owner that is not a credential cannot be phished, leaked, or brute-forced.

`notes_app` is what your application will use. It needs a password. Set it with `\password notes_app` in `psql`, which prompts for the password interactively. This avoids putting the cleartext password in the command history, where `CREATE ROLE ... PASSWORD 'secret'` would leave it.

Verify both roles with `\du`:

```
                      List of roles
  Role name  |          Attributes
-------------+------------------------------
 notes_app   |
 notes_owner | Cannot login
```

Becoming the owner deliberately

Let your administrator become the owner role only while running migrations:

```
GRANT notes_owner TO CURRENT_USER
  WITH INHERIT FALSE, SET TRUE;
```

`SET ROLE notes_owner` now activates the owner deliberately. With `INHERIT FALSE`, your admin session does not carry the owner's privileges all day; it opts in for the migration and runs `RESET ROLE` after. That keeps a fat-fingered `DROP TABLE` in a routine session from touching objects it happens to own.

One more rule that trips people up: role attributes such as `LOGIN`, `CREATEDB`, and `CREATEROLE` are never inherited like table privileges. Granting a role membership in another role passes along its object permissions, not its attributes. If a role needs to create databases, that attribute goes on the role itself.

The next lessons give these two roles a database of their own and exactly the permissions each one needs.

Create a PostgreSQL database

Create the application database with the non-login owner role and connect to it explicitly.

Our application needs its own database. Create it while connected as an administrator:

```
CREATE DATABASE notes_app OWNER notes_owner;
```

Don't forget the semicolon. Without it, `psql` waits on a continuation prompt and nothing runs.

The `OWNER` clause matters. Without it, the database belongs to whoever ran the command, usually your personal admin role. We want `notes_owner` to own it, so ownership follows the role we created for exactly that job, not a person who might leave the project.

Behind the scenes, PostgreSQL does not create the database from nothing. It clones `template1`, which is why a "new" database already contains a `public` schema.

Verify the result before moving on:

```
\l notes_app
```

```
                      List of databases
   Name    |    Owner    | Encoding | ... | Access privileges
-----------+-------------+----------+-----+-------------------
 notes_app | notes_owner | UTF8     |     |
```

The owner column should say `notes_owner`. If it shows your admin role, fix it with `ALTER DATABASE notes_app OWNER TO notes_owner;`.

Connect to the new database

Creating a database does not move you into it. A PostgreSQL connection works inside one database, so reconnect:

```
\connect notes_app
```

`\connect` closes the current connection and opens another one. Confirm the result with `\conninfo` before creating schemas or tables. Skipping that check is how tables end up in `postgres`, the maintenance database, instead of the application database.

Two errors you may hit

If your role lacks the `CREATEDB` attribute, the server refuses:

```
ERROR:  permission denied to create database
```

That is a role-attribute problem, not a grants problem. An administrator fixes it with `ALTER ROLE flavio CREATEDB;` or runs the creation for you.

The second one bites people running migrations: `CREATE DATABASE` cannot run inside a transaction block. Migration tools often wrap every step in `BEGIN ... COMMIT`, and the statement fails with exactly that message. Create databases as a separate manual step, and let migrations manage what lives inside the database instead.

Grant PostgreSQL permissions

Keep ownership with the migration role while granting the runtime role only the database, schema, table, and sequence access it needs.

Permissions in PostgreSQL are layered. To read one row, the runtime role must be allowed into the database, allowed to use the schema, and allowed to select from the table. Miss any layer and the query fails, each with a different error message. We will grant the layers one at a time so you can recognize each failure later.

Connect to `notes_app`, become the owner, and create a dedicated schema:

```
SET ROLE notes_owner;
CREATE SCHEMA app AUTHORIZATION notes_owner;
RESET ROLE;
```

Application tables will live in `app`, owned by `notes_owner`. Keeping them out of `public` makes every grant explicit.

Let the runtime role connect and resolve names inside that schema:

```
REVOKE CONNECT ON DATABASE notes_app FROM PUBLIC;
GRANT CONNECT ON DATABASE notes_app TO notes_app;
GRANT USAGE ON SCHEMA app TO notes_app;
```

The `REVOKE` line matters because PostgreSQL grants `CONNECT` to everyone by default through the `PUBLIC` pseudo-role. After revoking it, only roles you name can even open a connection to this database.

`USAGE` does not grant access to tables. It only lets the role look up names in the schema. Grant row operations separately:

```
GRANT SELECT, INSERT, UPDATE, DELETE
ON ALL TABLES IN SCHEMA app TO notes_app;

GRANT USAGE, SELECT
ON ALL SEQUENCES IN SCHEMA app TO notes_app;
```

The sequence grant looks optional. It is not. Identity columns use sequences. Without sequence access, an insert can fail even when the role has `INSERT` on the table, with `ERROR: permission denied for sequence notes_id_seq`. That error confuses everyone the first time, because the grant they check is the table one.

Cover tables that do not exist yet

Existing grants do not cover tables created later. Every future migration would leave the runtime role locked out of the new tables. Set default privileges instead. Run these statements as `notes_owner`:

```
SET ROLE notes_owner;

ALTER DEFAULT PRIVILEGES IN SCHEMA app
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO notes_app;

ALTER DEFAULT PRIVILEGES IN SCHEMA app
GRANT USAGE, SELECT ON SEQUENCES TO notes_app;

RESET ROLE;
```

Default privileges apply to objects created by the role that ran the statement. That is why `SET ROLE notes_owner` comes first: migrations run as the owner, so the defaults must belong to the owner.

Verify the outcome from the runtime role's point of view:

```
SET ROLE notes_app;
SELECT count(*) FROM app.notes;
RESET ROLE;
```

The runtime role can now change rows, but it does not own tables and cannot drop them. A leaked application credential is a bad day, not a lost schema.

Schemas and search_path

Resolve unqualified table names deliberately and avoid granting broad access through a convenient default schema.

When you write `SELECT * FROM notes` without a schema, PostgreSQL has to decide which `notes` you mean. It looks for an unqualified name such as `notes` in each schema on the current `search_path`, in order, and uses the first match.

Check what your session resolves against right now:

```
SHOW search_path;
```

```
   search_path
-----------------
 "$user", public
```

That default means: first a schema named after the current role, if one exists, then `public`. Our tables live in `app`, which is on neither, so a bare `notes` fails with "relation does not exist" even though the table is right there.

Set the path for the runtime role

You could qualify every query, but for the application role it is cleaner to set the default once:

```
ALTER ROLE notes_app IN DATABASE notes_app
SET search_path = app, pg_catalog;
```

Every new connection made by `notes_app` to this database now resolves `notes` as `app.notes`. The setting applies at connection time, so already-open sessions keep their old path until they reconnect.

Verify it the way the application will experience it:

```
\connect notes_app notes_app
SHOW search_path;
SELECT count(*) FROM notes;
```

If the count works without a schema prefix, the path is right.

For migration and administration work, my advice is the opposite: qualify queries as `app.notes` when clarity matters. Admin sessions connect as different roles with different paths, and an explicit name cannot be resolved into the wrong table.

The security angle

The search path is also a security boundary. Name resolution happens in path order, so never put a schema writable by an untrusted role before trusted schemas in the path. A hostile object created there can shadow the table or function you meant, and your queries execute against the impostor.

The historical version of this problem is `public`. Fresh PostgreSQL 15+ databases do not give every role `CREATE` on `public`. Clusters upgraded from PostgreSQL 14 or older can keep the earlier grant, where any role could create objects in everyone's default path. Inspect it with `\dn+ public`, and remove it when present:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

With the runtime path set and `public` locked down, "which table did I just query" stops being a question you need to ask.

Check your understanding: roles, databases, and schemas

Check the PostgreSQL hierarchy, ownership, role login, grants, schema usage, and search-path behavior.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Tables and data

Use PostgreSQL types, identity columns, constraints, and transactions.

Choose PostgreSQL data types

Use PostgreSQL types that express the value your application really stores.

Use integer or numeric types for numbers, `text` for unrestricted strings, `boolean` for true and false, and date or timestamp types for time.

Choose a type from the meaning and operations of the value, not from the format your application happens to receive.

```
CREATE TABLE invoices (
```

```
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  customer_id uuid NOT NULL,
  total numeric(12, 2) NOT NULL CHECK (total >= 0),
  issued_on date NOT NULL,
  paid_at timestamptz,
  note text
);
```

`integer` and `bigint` are exact whole numbers. `numeric` is exact decimal arithmetic and fits money or measurements where rounding must be controlled. Floating-point types are faster for scientific approximations, but values such as `0.1` cannot always be represented exactly.

Use `date` for a calendar day and `timestamptz` for an instant that may be displayed in different time zones. PostgreSQL stores `timestamptz` as an absolute instant and converts it for the session time zone. `timestamp` without a time zone is appropriate for a local wall-clock value only when no global instant is intended.

`text` and unconstrained `varchar` have the same practical storage behavior in PostgreSQL. Add a length limit only when it is a real domain rule, not as a guessed optimization.

PostgreSQL also provides UUIDs, arrays, ranges, enums, network types, and JSONB. A specialized type earns its place when database operators, validation, constraints, or indexes simplify real queries. Otherwise, an ordinary column is easier to understand and migrate.

Types are the first integrity boundary. Storing dates as text, booleans as `0` and `1`, or money as floating point pushes validation and comparison bugs into every caller.

Try it: design columns for an event with a global start instant, a maximum attendee count, a network address, and a price. Explain which invalid values each chosen type rejects before application code runs.

Use identity primary keys

Create generated numeric identifiers with the SQL-standard identity syntax.

Most tables need a numeric identifier that the database assigns by itself. In PostgreSQL, the modern way to get one is an **identity column**.

Define a generated key like this:

```
CREATE TABLE notes (
  id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  title TEXT NOT NULL
);
```

Identity columns use a sequence underneath but keep the generation rule attached to the column definition. You never name the sequence, never wire up a default by hand, and the rule shows up clearly in `\d notes`.

Older tutorials teach `SERIAL` for this. It still works, and you will meet it in existing schemas, but it is a PostgreSQL shorthand that expands into a separate sequence plus a column default. Identity is the SQL-standard syntax, and it is what the PostgreSQL documentation recommends for new tables.

Use `BIGINT`, not `INT`. The cost is four bytes per row. The alternative is discovering, years in, that

a busy table ran out of 32-bit identifiers.

ALWAYS or BY DEFAULT

The generation rule comes in two strictness levels.

`GENERATED BY DEFAULT` fills the column when you leave it out, but accepts a value if you supply one. That flexibility helps when importing existing rows that already carry identifiers.

`GENERATED ALWAYS` rejects manual values outright:

```
INSERT INTO notes (id, title) VALUES (500, 'Manual id');
```

```
ERROR:  cannot insert a non-DEFAULT value into column "id"
HINT:  Use OVERRIDING SYSTEM VALUE to override.
```

That error is a feature. It stops application code from quietly inserting identifiers the database did not assign, which later collide with generated ones. My advice is `GENERATED ALWAYS` for new tables, and `BY DEFAULT` only when you have an import that genuinely needs it.

Verify the generation

Insert a row and read back the assigned key in one statement:

```
INSERT INTO notes (title)
VALUES ('Plan the week')
RETURNING id;
```

```
 id
----
  1
```

Run it again and you get `2`. Do not expect the numbers to be gapless, though: identifiers can skip values after rolled-back inserts, and that is normal. The sequence machinery behind this, including what happens to gaps and how imports can leave a sequence behind the table's real maximum, gets its own lesson next.

Use constraints and foreign keys

Put durable data rules in PostgreSQL so every application and script receives the same protection.

Use `NOT NULL`, `UNIQUE`, `CHECK`, primary keys, and foreign keys to reject invalid states.

```
CREATE TABLE projects (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  slug text NOT NULL UNIQUE,
  name text NOT NULL
);

CREATE TABLE tasks (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  project_id bigint NOT NULL REFERENCES projects(id) ON DELETE CASCADE,
  title text NOT NULL,
  progress integer NOT NULL CHECK (progress BETWEEN 0 AND 100)
);
```

These rules apply to every writer: the web application, an import script, a database console, and next year's replacement service. Application validation can produce friendlier messages, but it cannot provide the same final guarantee under concurrency.

`NOT NULL` says a value must exist. `CHECK` constrains values in a row. `UNIQUE` protects a candidate identifier across rows. A primary key is the table's main non-null unique identity. A foreign key prevents a child row from pointing at a missing parent.

Remember that a `CHECK` expression that evaluates to `NULL` passes. If a value is both required and constrained, use `NOT NULL` and `CHECK` together.

Choose foreign-key delete behavior deliberately:

- `RESTRICT` or `NO ACTION` rejects a parent deletion while children exist.
- `CASCADE` deletes dependent children too.
- `SET NULL` preserves the child but requires a nullable relationship.

`CASCADE` fits data that has no meaning alone, such as a task inside a deleted project. It is dangerous when one parent has millions of children or when the child represents an independent business record. Preview the affected rows and keep destructive operations observable.

PostgreSQL automatically indexes primary and unique keys, but it does not automatically index the referencing side of every foreign key. Add an index on `tasks(project_id)` when joins, parent deletion checks, or lookups need it.

Try it: attempt an invalid progress value, a duplicate slug, and a task with a missing project. Read each error and identify which invariant stopped the write.

Understand sequences

Know why generated identifiers can have gaps and how a sequence can fall behind imported rows.

A PostgreSQL sequence allocates values independently from a transaction. A rolled-back insert can leave a gap, and that is normal.

Identity columns commonly use a sequence behind the scenes:

```
CREATE TABLE notes (
  id bigint GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  title text NOT NULL
);
```

When PostgreSQL calls `nextval`, that value is consumed even if the transaction later rolls back or the insert conflicts. This prevents concurrent transactions from receiving the same value without making them wait for each other. It also means generated identifiers promise uniqueness, not a gapless business sequence.

Do not infer row count, ordering, or missing data from gaps. Use `created_at` for chronology and `count(*)` for a count. If invoice numbers must obey legal sequencing rules, model that requirement separately and understand the concurrency cost.

An import that supplies explicit identifiers can move the table ahead without moving its sequence. The next generated insert may then collide. Find the owned sequence and compare it with the data before changing anything:

```
SELECT pg_get_serial_sequence('notes', 'id');
```

```
SELECT max(id) FROM notes;
```

After confirming no concurrent import or insert is running, align the sequence deliberately:

```
SELECT setval(
  pg_get_serial_sequence('notes', 'id'),
  COALESCE((SELECT max(id) FROM notes), 1),
  EXISTS (SELECT 1 FROM notes)
);
```

The third argument matters. `true` means the next `nextval` advances beyond the supplied value; `false` means it returns that value first. The empty-table case above avoids skipping the initial value.

Never reset a live sequence merely to make IDs look consecutive. A lower value can collide with existing rows, and renumbering primary keys can break references, URLs, logs, and external systems.

Try it: insert a row inside a transaction, roll it back, then insert another row. Observe the gap and explain why it is safer than reclaiming the first value.

Use PostgreSQL transactions

Group changes and keep locks and connections for the shortest useful time.

Use `BEGIN`, `COMMIT`, and `ROLLBACK` around changes that form one unit.

```
BEGIN;

UPDATE accounts
SET balance = balance - 20
WHERE id = 1 AND balance >= 20;

UPDATE accounts
SET balance = balance + 20
WHERE id = 2;

COMMIT;
```

The transaction is atomic: both updates become visible together, or `ROLLBACK` discards them. Application code must also check that the debit updated exactly one row. Atomic execution does not make an incorrect business rule correct.

PostgreSQL's default `READ COMMITTED` isolation gives each statement a snapshot of committed data when that statement begins. A second query in the same transaction can therefore see a row another transaction committed in between. Stronger isolation levels change what can be observed, but may abort a transaction that must then be retried.

Concurrent transactions can still block one another when they update the same rows or request conflicting locks. Acquire rows in a consistent order to reduce deadlocks. When PostgreSQL detects a deadlock, it aborts one participant; the application should retry the whole transaction only when the operation is safe to repeat.

After any statement error, the transaction is aborted until `ROLLBACK` or `ROLLBACK TO SAVEPOINT`. Do not catch the application exception and continue sending unrelated SQL through the same failed transaction.

Keep the boundary small. Validate and compute before `BEGIN`, make the related database changes, then commit. Do not wait for email, payment, user input, or another network service while holding locks and a pooled connection. For external effects that must follow a commit, an outbox table can record work in the same transaction for a separate worker to deliver.

Try it: open two `psql` sessions, update the same row without committing in the first, and observe the second wait. Use `ROLLBACK` in the first session and confirm which value remains.

Do not wait for email, payment, or another network service inside a database transaction.

Check your understanding: tables and data

Check PostgreSQL types, identity columns, sequences, constraints, foreign keys, transactions, and locks.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Queries and performance

Use PostgreSQL query features and inspect plans before adding indexes.

Return changed rows

Use RETURNING to get generated or updated values without sending a second query.

After an insert, your application usually needs to know what the database created: the generated identifier, a default timestamp, a computed value. The naive pattern is to write, then immediately read back. PostgreSQL can return values directly from an insert, update, or delete instead:

```
INSERT INTO notes (title)
VALUES ('Plan the week')
RETURNING id, title;
```

```
 id |     title
----+---------------
  4 | Plan the week
(1 row)

INSERT 0 1
```

This keeps the change and its result in one statement. One round trip instead of two, and no race: a separate `SELECT max(id)` after the insert can pick up somebody else's row on a busy system. `RETURNING` gives you exactly the rows this statement touched.

It works on UPDATE and DELETE too

`RETURNING` accepts any expression over the affected rows, so updates can hand back their new values:

```
UPDATE notes
SET title = 'Plan Monday'
WHERE id = 4
RETURNING id, title;
```

And deletes can hand back what they removed:

```
DELETE FROM notes
WHERE created_at < now() - interval '1 year'
RETURNING id;
```

The delete variant doubles as an audit trail: you know precisely which rows a cleanup job removed, which beats logging "deleted some old rows".

Read the row count

Look at the status line psql prints, or the row count your driver reports. It tells you how many rows the statement actually changed, and `RETURNING` shows you which ones.

Zero rows is the quiet failure mode. An `UPDATE` whose `WHERE` matched nothing succeeds with an empty result, no error. Application code that assumes "no exception means it worked" ships bugs this way. Check that `RETURNING` produced the row you expected.

The opposite direction is more painful. Forget the `WHERE` entirely and the statement updates every row in the table, then dutifully returns all of them. If an update you expected to touch one row starts streaming back thousands, that flood of output is your earliest warning, and a transaction you can still `ROLLBACK` is your way out. Wrapping risky writes in an explicit transaction is covered in the transactions lesson; `RETURNING` is what makes the damage visible before you commit it.

Upsert with ON CONFLICT

Insert a row or handle a specific uniqueness conflict in one deliberate statement.

"Insert this row, unless it already exists, in which case update it" is one of the most common write patterns there is. Doing it as a `SELECT` followed by an `INSERT` or `UPDATE` is a race condition: two requests can both see "not there yet" and both insert. PostgreSQL solves it atomically with **upsert**.

Use `ON CONFLICT` against a real unique constraint:

```
INSERT INTO settings (user_id, theme)
VALUES (1, 'dark')
ON CONFLICT (user_id)
DO UPDATE SET theme = EXCLUDED.theme;
```

If no row with `user_id = 1` exists, this is a plain insert. If one exists, the unique constraint on `user_id` fires, and instead of an error PostgreSQL runs the `DO UPDATE` against the existing row.

`EXCLUDED` is the row you tried to insert. `EXCLUDED.theme` means "the theme value from my `VALUES` clause", so the same statement works no matter which value you pass in.

The conflict target is not optional decoration

`(user_id)` names which uniqueness rule you are handling, and PostgreSQL checks that a matching unique index or constraint really exists. Without one you get:

```
ERROR:  there is no unique or exclusion constraint matching the ON CONFLICT specification
```

That error is telling you something real: upsert is only meaningful against an actual constraint. If `user_id` is not unique, "the existing row" is not a well-defined idea, and the fix is to add the constraint, not to fight the syntax.

Name the conflict target. Do not silently swallow every possible error.

DO NOTHING, used narrowly

Sometimes the right reaction to a duplicate is to skip it, for example when replaying events that may already be recorded:

```
INSERT INTO events (event_id, payload)
VALUES ('evt_9f2c', '{"type":"signup"}')
ON CONFLICT (event_id) DO NOTHING;
```

Keep the explicit target even here. A bare `ON CONFLICT DO NOTHING` ignores conflicts on any constraint, which quietly discards rows for reasons you never anticipated.

Verify what an upsert actually did by adding `RETURNING id, theme` to the statement, or by checking the row count: `INSERT 0 1` means one row was written, `INSERT 0 0` after `DO NOTHING` means the row was skipped.

Use JSONB deliberately

Keep flexible nested data in JSONB without turning every relational value into an opaque document.

`jsonb` stores parsed JSON and supports operators and indexes for querying inside it.

It works well for optional metadata whose keys genuinely vary:

```
CREATE TABLE events (
  id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  kind text NOT NULL,
  occurred_at timestamptz NOT NULL,
  metadata jsonb NOT NULL DEFAULT '{}'::jsonb
);

INSERT INTO events (kind, occurred_at, metadata)
VALUES ('purchase', now(), '{"plan":"pro","source":"newsletter"}');
```

Query a text value with `->>`:

```
SELECT id
FROM events
WHERE metadata ->> 'source' = 'newsletter';
```

`->` returns JSON; `->>` returns text. This distinction affects comparisons and indexes. PostgreSQL also supports containment queries such as `metadata @> '{"plan":"pro"}'`.

A general GIN index can accelerate many containment and key-existence queries:

```
CREATE INDEX events_metadata_gin_idx
ON events USING gin (metadata);
```

For one frequent expression, a smaller expression index may be a better fit:

```
CREATE INDEX events_source_idx
ON events ((metadata ->> 'source'));
```

Use `EXPLAIN ANALYZE` with representative data before deciding. The broad GIN index costs more storage and write work, while an expression index serves a narrower query.

Keep ordinary columns for values you routinely filter, join, constrain, sort, or update independently. Hiding `customer_id`, status, or timestamps inside JSON weakens type checks and foreign keys and

makes every query repeat extraction logic. JSONB fits the flexible edge of a relational model, not the entire model by default.

Updates rewrite the JSONB value, so a large document that changes one small key can create unnecessary write amplification. Split frequently updated fields into columns or related tables.

Try it: store two event kinds with different metadata, query them by containment, inspect the plan before and after an index, then decide whether the measured workload justifies keeping it.

Read EXPLAIN ANALYZE

Compare estimated and actual work while remembering that the analyzed query really executes.

When a query is slow, guessing is expensive. PostgreSQL will tell you exactly what it did, if you ask.

`EXPLAIN` shows the chosen plan. `EXPLAIN ANALYZE` runs the statement and adds actual rows and timing. Use `BUFFERS` on a read to see cache and storage work:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, title
FROM app.notes
WHERE title = 'Plan the week';
```

```
Index Scan using notes_title_idx on notes
    (cost=0.29..8.30 rows=1 width=40)
    (actual time=0.041..0.043 rows=1 loops=1)
  Index Cond: (title = 'Plan the week'::text)
  Buffers: shared hit=3
Planning Time: 0.180 ms
Execution Time: 0.071 ms
```

Read it in this order. The node type first: `Index Scan` means the index found the row directly. Then compare estimated rows with actual rows; `rows=1` estimated against `rows=1` actual means the planner's statistics match reality. When the two diverge wildly, the plan was chosen on bad information. Then find the largest scan, repeated loop, sort, or buffer count. `shared hit=3` means three pages, all already in memory.

When the index is ignored

Here is the trap you will hit first in real code. Wrap the indexed column in a function and the plan changes:

```
EXPLAIN ANALYZE
SELECT id, title
FROM app.notes
WHERE lower(title) = 'plan the week';
```

```
Seq Scan on notes
    (cost=0.00..2041.00 rows=500 width=40)
    (actual time=0.011..14.2 rows=1 loops=1)
  Filter: (lower(title) = 'plan the week'::text)
  Rows Removed by Filter: 99999
```

`Seq Scan` plus a big `Rows Removed by Filter` is the signature: PostgreSQL read the whole table

to find one row. The index stores `title`, not `lower(title)`, so it cannot serve this predicate. Fix it with an expression index on `lower(title)`, or stop transforming the column in the query.

ANALYZE executes the query

`EXPLAIN ANALYZE` executes writes too. Analyzing an `UPDATE` really updates. Wrap a test write in a transaction and roll it back:

```
BEGIN;
EXPLAIN ANALYZE
UPDATE app.notes SET title = 'Plan Monday' WHERE id = 1;
ROLLBACK;
```

Do this on test data. Functions and sequence changes can have effects that a rollback does not undo. When you only need the plan and not the timings, plain `EXPLAIN` never executes anything and is always safe.

Add the index the query needs

Match index columns and order to a real filter, join, sort, or uniqueness requirement.

B-tree indexes cover most equality and range lookups. PostgreSQL also offers specialized index types for other operators and data types.

A useful index starts with a real query:

```
SELECT id, created_at, title
FROM notes
WHERE user_id = 42
ORDER BY created_at DESC
LIMIT 20;
```

A matching index is:

```
CREATE INDEX notes_user_created_idx
ON notes (user_id, created_at DESC);
```

A composite B-tree index is ordered by its leading columns. PostgreSQL can jump to one user's range and read recent rows in index order. The same index is much less useful for a query that only filters on `created_at`.

B-tree is the default and covers equality, ranges, and ordering. PostgreSQL also provides specialized types: GIN is common for JSONB containment, arrays, and full-text search; GiST supports operator families such as ranges and geometry; BRIN can be compact and effective on very large tables where values correlate with physical row order.

Partial indexes store only rows matching a predicate:

```
CREATE INDEX notes_unpublished_user_idx
ON notes (user_id)
WHERE published_at IS NULL;
```

They are valuable when the application repeatedly queries a small, stable subset. The query predicate must imply the index predicate for PostgreSQL to use it.

Indexes cost disk space, cache, and write work. Before adding one, capture the slow query and inspect `EXPLAIN (ANALYZE, BUFFERS)` on representative data. After adding it, confirm the selected

plan and real row counts. A sequential scan is often correct for a small table or a query returning much of the table.

In production, `CREATE INDEX` can block writes while building. `CREATE INDEX CONCURRENTLY` reduces that blocking but takes longer, performs more work, cannot run inside a transaction block, and can leave an invalid index after failure. Plan the operational path, not only the final schema.

Recheck the plan after adding an index and remove indexes the workload no longer uses.

Try it: compare the plans for the query above with no index, an index on `user_id`, and the compound index. Explain why the winner depends on both filtering and ordering.

Check your understanding: queries and performance

Check RETURNING, conflict handling, JSONB tradeoffs, EXPLAIN ANALYZE, estimates, and useful index design.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Applications and operations

Connect applications, control pools, migrate safely, restore backups, and debug failures.

Connect to PostgreSQL from Node.js

Optionally connect from Node.js with pg, a secret connection URL, bounded waits, and deliberate TLS settings.

The application lessons use Node.js as an optional track. You can follow the database and operations lessons without it.

Install the `pg` driver:

```
npm install pg
```

`pg` (also called node-postgres) is the standard low-level PostgreSQL driver for Node.js. Query builders like Kysely and ORMs like Drizzle sit on top of it, so what you learn here applies even when you add a layer later.

The connection URL

Everything the driver needs fits in one URL:

```
postgresql://notes_app:password@localhost:5432/notes_app
```

Read it left to right: role, password, host, port, database. It is the same information you have been giving `psql` with `-U`, `-h`, `-p`, and `-d`, packed into a string that fits in one environment variable.

Keep the connection URL in deployment secrets. Do not commit it, print it, or send it to browser code. The password is right there in the string, so treat the whole URL as a credential.

One encoding gotcha: passwords with special characters such as `@`, `:`, or `#` break the URL parsing. Percent-encode them, so `p@ss` becomes `p%40ss`. Authentication failures that only happen in production, after someone rotated the password, are this bug more often than not.

Verify the connection

Prove the URL works before building on it:

```
import pg from 'pg'

const pool = new pg.Pool({
  connectionString: process.env.DATABASE_URL,
})

const result = await pool.query('SELECT current_database(), current_user')
console.log(result.rows[0])
// { current_database: 'notes_app', current_user: 'notes_app' }

await pool.end()
```

If this prints the wrong database or role, fix it now. If it hangs and then throws `ECONNREFUSED`, the URL points at a host or port where nothing is listening; recheck host and port against `\conninfo` from a working `psql` session.

Hosted databases: two URLs and TLS

A hosted provider may give you separate direct and pooled URLs. The pooled one goes through the provider's connection pooler and is meant for application traffic; the direct one is for migrations and admin tasks. Use each for its job.

For a remote database, follow the provider's TLS instructions and verify its certificate. Do not silence certificate errors with `rejectUnauthorized: false`. That setting disables the check that you are talking to your real database and not an interceptor, which turns your connection encryption into decoration.

Budget connection pools

Limit connections across all application replicas and preserve room for workers, migrations, and emergency access.

Opening a PostgreSQL connection is not free. The server forks work for each one, and it enforces a hard ceiling: `max_connections`, 100 by default. A **connection pool** keeps a few connections open and hands them out to requests, so your application neither pays the setup cost per query nor stampedes the server.

The part people miss is that a pool is a budget, and the budget is cluster-wide.

Create one bounded pool in each application process:

```
import pg from 'pg'

const { Pool } = pg

export const pool = new Pool({
  connectionString: process.env.DATABASE_URL,
  max: 5,
  connectionTimeoutMillis: 5000,
  idleTimeoutMillis: 30000,
```

```
  statement_timeout: 10000,
  application_name: 'notes-web',
})
```

Each setting bounds a different failure. `max: 5` caps how many connections this process can hold. `connectionTimeoutMillis` stops a request from waiting forever for a free client. `idleTimeoutMillis` returns unused connections to the server. `statement_timeout` kills runaway queries instead of letting them occupy a connection for minutes. `application_name` labels these connections so you can recognize them later.

Do the multiplication

Five connections is an example budget. Eight replicas with a pool of five can still request forty connections. Autoscaling makes this worse: the platform adds replicas under load, and every new replica brings its whole pool allowance with it, right when the database is busiest.

Leave room for workers, migrations, and emergency access. If the application fleet can consume every available connection, the day something goes wrong is the day you cannot open a `psql` session to investigate.

Verify the budget from the server

Count what the server actually sees:

```
SELECT application_name, count(*)
FROM pg_stat_activity
WHERE datname = 'notes_app'
GROUP BY application_name;
```

```
 application_name | count
------------------+-------
 notes-web        |    10
 notes-worker     |     3
```

That `application_name` label now pays off: you can attribute every connection to a service and check the totals against your budget.

When the budget is blown, the symptom is `FATAL: sorry, too many clients already` from the server, or pool timeout errors inside the application. Do not fix that by raising `max`. Find which service is over budget or leaking first.

One more habit: call `await pool.end()` when a script or worker shuts down. Short-lived scripts that skip it leave connections lingering until the server times them out, and enough of those eat the budget too.

Parameterize PostgreSQL queries

Pass values separately with $1 placeholders and return generated data in the same statement.

Building SQL by gluing user input into a string is how SQL injection happens. If a note title arrives as `'); DROP TABLE app.notes; --`, string concatenation turns your insert into two statements, and the second one is the attacker's.

The fix is not escaping. It is keeping SQL and data on separate channels. Pass untrusted values separately:

```
const result = await pool.query(
  `INSERT INTO app.notes (title, body)
   VALUES ($1, $2)
   RETURNING id, title`,
  ['Plan the week', 'Choose three priorities']
)

console.log(result.rows[0])
// { id: 7, title: 'Plan the week' }
```

`$1` and `$2` are **placeholders**. The driver sends the SQL text and the values array to PostgreSQL as separate parts of the protocol. The server parses the statement first, then binds the values. By the time your data arrives, the statement's structure is fixed, so no input can add a second statement or a sneaky `OR 1=1`. The hostile title above just becomes a note with a weird name.

This is not a sanitization step you must remember to apply. It is the normal way to run queries with `pg`, and it should be the only way values reach your SQL.

`RETURNING` gives the application the generated identifier without a second query, so the insert and its result stay in one round trip.

What placeholders cannot do

Placeholders represent values. They cannot represent table names, column names, or sort directions. Try it and PostgreSQL rejects the statement with a syntax error, because those parts define the query's structure, and the structure must be known at parse time.

When the application needs a dynamic column or direction, choose those from a fixed allowlist:

```
const sortable = { created: 'created_at', title: 'title' }
const column = sortable[req.query.sort] ?? 'created_at'
const dir = req.query.dir === 'asc' ? 'ASC' : 'DESC'

const notes = await pool.query(
  `SELECT id, title FROM app.notes
   ORDER BY ${column} ${dir}
   LIMIT $1`,
  [20]
)
```

The interpolated pieces never contain user input, only values your own code chose from a closed set. The user picks a key; your code picks the SQL.

Verify the discipline holds with a search: any query in the codebase that concatenates a request value into SQL text is a finding, even when it ”looks safe today”.

Run an application transaction

Keep several queries on one checked-out client and always commit, roll back, and release it.

You know transactions from the SQL side: `BEGIN`, some statements, `COMMIT`. From application code there is one extra rule that changes everything.

A transaction belongs to one connection. `BEGIN` starts it on whichever connection carried that

statement, and only statements on that same connection are inside it. A pool, by design, spreads queries across connections. Those two facts collide.

Check out one client and use it for every statement:

```
const client = await pool.connect()

try {
  await client.query('BEGIN')

  const note = await client.query(
    'INSERT INTO app.notes (title) VALUES ($1) RETURNING id',
    ['Plan the week']
  )

  await client.query(
    'INSERT INTO app.note_tags (note_id, tag_id) VALUES ($1, $2)',
    [note.rows[0].id, 1]
  )

  await client.query('COMMIT')
} catch (error) {
  await client.query('ROLLBACK')
  throw error
} finally {
  client.release()
}
```

`pool.connect()` reserves one connection for you until you release it. Every `client.query()` rides that connection, so all four statements share the transaction.

The shape of the `try/catch/finally` is not boilerplate to trim. `COMMIT` on success. `ROLLBACK` on any error, so the connection goes back to the pool clean instead of carrying a broken transaction to its next borrower. And `client.release()` in `finally`, so the connection returns even when things go wrong.

The mistake that looks fine in testing

Do not call `pool.query()` inside this transaction. The pool may choose another connection. That statement then runs outside the transaction: it can commit on its own while your transaction rolls back, or read data your uncommitted transaction cannot see. Under low traffic the pool often reuses the same connection, so the bug hides through development and appears under production load.

The other classic failure is forgetting `release()` on some code path. Each forgotten client shrinks the pool by one until requests start timing out waiting for a connection.

Verify from another session

While the code between `BEGIN` and `COMMIT` runs, the server shows the reserved connection:

```
SELECT pid, state, query
FROM pg_stat_activity
WHERE application_name = 'notes-web';
```

A row with state `idle in transaction` is a transaction that is open while the application does non-database work. Brief is normal. Minutes long means some code path forgot to commit, roll back, or release, and it is holding locks the whole time.

Migrate PostgreSQL safely

Use expand-and-contract changes so old and new application versions can run during a deployment.

A migration and a deployment do not happen at the same instant. For a while, old and new application instances run against the same database, and both must keep working.

That rules out the obvious version of many changes. Rename a column in one step:

```
ALTER TABLE app.users
RENAME COLUMN name TO display_name;
```

The migration succeeds, and every still-running old instance starts failing with `column "name" does not exist`. The database did nothing wrong. The deployment model did.

Expand and contract

The safe pattern splits every breaking change into compatible steps. Add new structures before code requires them. Backfill in restartable batches, switch reads and writes in separate releases, then remove old structures after the compatibility window.

For the rename, that means: add `display_name` as a nullable column, deploy code that writes both columns, backfill old rows in batches, deploy code that reads the new column, and only then drop `name`. Each step works with the versions running before and after it. A failed deploy in the middle strands you in a state that still functions.

Verify the backfill before switching reads:

```
SELECT count(*) FROM app.users
WHERE display_name IS NULL AND name IS NOT NULL;
```

Zero means the new column is complete.

Bound the blast radius

Schema changes take locks, and a lock you wait for politely still ruins the site: every query queued behind your `ALTER TABLE` waits too. Bound how long a migration can wait or run:

```
SET lock_timeout = '2s';
SET statement_timeout = '5min';
```

With `lock_timeout`, a migration that cannot get its lock quickly fails fast and can retry off-peak, instead of stalling production traffic behind it. A failed migration is annoying. A hung one is an outage.

Large constraints, indexes, and table rewrites can lock or scan production data. Test the exact operation on a realistic copy and decide rollback before deployment. ”It ran instantly on my laptop” says nothing about a table with fifty million rows.

`CREATE INDEX CONCURRENTLY` reduces blocking on a live table, but it cannot run inside a transaction block, and migration tools wrap steps in transactions by default. Treat it as its own restartable migration step, with the tool's transaction wrapper disabled for that step. If it fails partway, it leaves an `INVALID` index behind; drop it and rerun rather than assuming the index works.

Back up and restore PostgreSQL

Create a custom-format dump and verify it by restoring a separate database.

A backup you have never restored is a hope, not a backup. The file can be incomplete, the schema can depend on roles that no longer exist, the application can choke on what comes back. So this lesson does both halves: dump, then prove the dump works.

Create a custom-format archive:

```
pg_dump --format=custom --file=notes.dump notes_app
```

`pg_dump` takes a consistent snapshot of one database while it stays online; readers and writers keep working during the dump. The custom format is compressed and, unlike a plain SQL file, lets `pg_restore` restore selectively: one table, schema only, data only.

Peek inside the archive without touching any database:

```
pg_restore --list notes.dump
```

A table of contents listing your schemas, tables, and indexes tells you the file is a readable archive. It does not yet prove the data restores.

Restore into a clean database

Restore it into a clean database, never over the source:

```
createdb --template=template0 notes_restore
pg_restore --exit-on-error --no-owner \
  --dbname=notes_restore notes.dump
```

`template0` gives you a genuinely empty database. `--exit-on-error` stops at the first failure instead of plowing through and leaving a half-restored database that looks fine. `--no-owner` skips restoring object ownership, which otherwise fails with `role "notes_owner" does not exist` when the target cluster lacks the original roles; everything becomes owned by the connecting role instead.

Verify the restored data:

```
psql notes_restore -c 'SELECT count(*) FROM app.notes'
```

Compare the count against the source. Better still, point a test instance of your application at `notes_restore` and click around. A database can restore perfectly and still not serve the application, for example when the dump predates a migration the code now requires.

What pg_dump does not cover

`pg_dump` backs up one database. Roles and other cluster-global objects are outside it. Back them up separately when you operate the cluster:

```
pg_dumpall --globals-only > globals.sql
```

Inspect and protect that file. It contains role definitions and can contain password hashes, so it deserves the same secrecy as any credential.

A successful dump job is not enough: restore it and run the application against the result. Put that restore drill on a schedule, because the failure mode of backups is silent, and you want to find it on a quiet afternoon rather than during the incident.

Fix a PostgreSQL connection failure

Diagnose service, socket, network, authentication, and database errors without reinstalling PostgreSQL or risking the data directory.

One day `psql` greets you with this:

```
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed:
No such file or directory
        Is the server running locally and accepting connections on that socket?
```

Start with the complete error. Do not reinstall PostgreSQL: that can hide the cause and put the existing cluster at risk. Reinstalling sometimes "works" by creating a fresh, empty data directory, which is a polite way of saying it can disconnect you from your data. The error text already narrows the problem down; use it.

Read the error as a layer diagnosis

Connection failures happen at one of a few layers, and each produces a distinct message.

A missing socket, like the error above, points to service or socket-directory configuration. The client looked for a local server and found nobody home. On a Homebrew machine that usually means the service is stopped or crashed, often after a `brew upgrade` that moved you across a major version so the new server refuses the old data directory.

"Connection refused" points to the server, address, port, or firewall. Something answered the network layer with "nothing is listening here". Wrong port, wrong host, or a server bound only to localhost while you connect from outside.

A `FATAL` authentication message means the server answered, so check the role, password, `pg_hba.conf`, TLS, and database name. This is progress: the server is fine, and you are negotiating with it.

Work through it locally

Check the client and Homebrew service:

```
psql --version
brew services list
```

If the service shows `error` or `stopped`, read the log before touching anything:

```
tail -50 /opt/homebrew/var/log/postgresql@18.log
```

A version-mismatch line such as `database files are incompatible with server` confirms the major-upgrade scenario, and the fix is `pg_upgrade` with the old binaries, not deletion.

Then try an explicit local connection:

```
psql -h localhost -p 5432 -U notes_app -d notes_app
```

Spelling out host, port, role, and database removes every default from the equation. If this works while plain `psql` fails, your problem was defaults, not the server.

Inspect Homebrew service logs before changing configuration. Change one layer at a time and retry the same command. If you change three things and it starts working, you do not know which change fixed it, and you may have broken something else quietly.

Fix “relation does not exist”

Check the current database, schema path, migrations, and PostgreSQL identifier folding before changing a query.

The error looks like this:

```
ERROR:  relation "notes" does not exist
LINE 1: SELECT * FROM notes;
```

PostgreSQL says ”relation does not exist” when it cannot resolve the table name in the current connection. Read that carefully: not ”the table does not exist anywhere”, but ”I cannot find it from where this connection stands”. The table is usually fine. The connection is looking in the wrong place.

Check where you are before changing the query

Check the connection and visible tables first:

```
SELECT current_database(), current_schema();
SHOW search_path;
\dt *.*
```

These four lines resolve most cases. `\dt *.*` lists tables in every schema, so you can see where the table actually lives.

Three outcomes, three fixes.

If `app.notes` exists but `notes` fails, fix the `search_path` or qualify the table as `app.notes`. The table is there; your session's path just does not include its schema.

If the table is absent from this database entirely, check `current_database()` against what you expected. With several environments and local clusters around, running migrations against `postgres` while the application connects to `notes_app` is an everyday mistake. Run the expected migrations against this database.

If the table exists with strange casing, keep reading.

The identifier folding trap

PostgreSQL folds unquoted identifiers to lowercase. `CREATE TABLE Notes` creates `notes`, while `CREATE TABLE "Notes"` creates a case-sensitive name that always needs quotes.

The painful combination is a table created with quotes, usually by an ORM or a GUI tool, then queried without them:

```
SELECT * FROM Notes;
-- ERROR:  relation "notes" does not exist

SELECT * FROM "Notes";
-- works
```

The unquoted `Notes` folds to `notes`, which does not exist; only `"Notes"` does. The error message even shows you the folded name in quotes, which is your clue.

My advice is to use lowercase unquoted names. Do not add quotes until you know the table was created with a quoted mixed-case name; adding quotes to a query against an ordinary lowercase

table creates the same error in reverse.

After any fix, verify with the failing statement itself, then with `\d app.notes` to confirm you are looking at the table you think you are.

Choose where PostgreSQL runs

Compare a managed service with operating PostgreSQL yourself before treating the monthly price as the whole cost.

Everything in this course so far ran on your laptop, and for development that is the right answer: fast, free, and yours to break. A deployed application needs a database reachable from wherever the application runs, and that is a real decision.

You have two families of options, and the difference between them is not features. It is whose pager rings.

Managed PostgreSQL

A managed provider normally handles installation, patching, monitoring, backups, and failover options. You get a connection URL and a dashboard; the provider keeps the server alive.

You still own schema changes, queries, access, capacity, and restore testing. No provider knows that your `notes` table needs an index on `user_id`, or that the runtime role should not own tables, or whether last night's backup actually restores into something your application can use. The lessons in this course stay your job on every host.

Managed costs more per month than raw compute. What you are buying is the operational work you no longer perform, and for most small teams that trade is worth it.

Self-hosted PostgreSQL

Self-hosting gives more control but also makes upgrades, storage, replication, security, and recovery your responsibility. On a $5 VPS you can run PostgreSQL next to your application and pay almost nothing.

The monthly price is not the whole cost. Price the hours: securing the server, applying updates, planning major-version upgrades, watching disk space, and rehearsing restores. If that work excites you, self-hosting is a great teacher. If it sounds like a distraction from your product, it is.

Choose the operational work you are prepared to perform, not only the database feature list.

Questions that decide it

A few concrete checks beat any comparison table:

- Can you restore yesterday's backup right now, and have you tried?
- Where must the data live, and does the host's region satisfy that?
- Does your platform run serverless functions? Those multiply connections, so a host with a built-in connection pooler saves you pain.
- What happens when the machine dies at 3am, and who notices?

Whichever way you choose, run the same verification: take a backup, restore it into a scratch database, and point the application at it. The failure mode of hosting decisions is assuming somebody else tested recovery. Providers back up; only you can prove the restore feeds your application.

Understand MVCC and routine maintenance

Recognize dead row versions, keep planner statistics current, and inspect active sessions before guessing at an operational problem.

PostgreSQL uses **MVCC**, multiversion concurrency control, so readers and writers can often work at the same time. Instead of overwriting a row in place, an update creates a new row version. The old version stays until no transaction needs it. Readers that started before your update keep seeing the old version; nobody blocks anybody for a plain read.

The price is garbage. Updated and deleted rows leave **dead row versions** behind, physically present in the table but invisible to new queries. A table with heavy update traffic can hold far more dead versions than live rows, wasting disk and slowing scans.

Autovacuum is the cleanup crew

Autovacuum reclaims reusable space and runs `ANALYZE` to refresh planner statistics. It wakes up on its own, checks which tables changed enough, and cleans them. Do not disable it as a performance shortcut. The workload spike you avoid today becomes table bloat and a bad query plan next month.

The thing that quietly defeats autovacuum is a long transaction. Cleanup can only remove versions no transaction might need, so one session sitting `idle in transaction` since this morning pins every dead row created since then, across the whole database.

Statistics have a manual lever too. Refresh statistics after a large test-data load:

```
ANALYZE app.notes;
```

The planner estimates row counts from statistics. Right after loading a million rows, the statistics still describe the empty table, and plans come out wrong until the next analyze.

Look before you guess

Check what maintenance has been doing:

```
SELECT relname, n_live_tup, n_dead_tup, last_autovacuum
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC;
```

A table with a large `n_dead_tup` and an old `last_autovacuum` is a table where cleanup is losing, and the next query tells you why. Inspect current sessions and waits:

```
SELECT pid, usename, application_name, state,
       wait_event_type, wait_event, query_start, query
FROM pg_stat_activity
WHERE datname = current_database();
```

Look for long `idle in transaction` sessions, old queries, and lock waits. That idle-in-transaction session is the classic finding: an application checked out a connection, started a transaction, and went off to do something else, holding locks and pinning dead rows the whole time.

Diagnose them before increasing connection limits or disabling maintenance. Operational ”fixes” applied without this look — more connections, less vacuuming — usually feed the underlying problem instead of solving it.

Build a PostgreSQL notes application

Build and verify a complete PostgreSQL notes application with separated ownership, safe queries, measured indexes, and a restored backup.

Let's finish with one runnable project.

Create the roles and database as the PostgreSQL administrator, using your real administrator role in the membership grant:

```
CREATE ROLE notes_owner NOLOGIN;
CREATE ROLE notes_app LOGIN;
\password notes_app
GRANT notes_owner TO CURRENT_USER
  WITH INHERIT FALSE, SET TRUE;
CREATE DATABASE notes_app OWNER notes_owner;
```

Connect to `notes_app`, then create the schema as its owner:

```
\connect notes_app
SET ROLE notes_owner;

CREATE SCHEMA app AUTHORIZATION notes_owner;

CREATE TABLE app.notes (
  id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  title TEXT NOT NULL,
  body TEXT,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE app.tags (
  id BIGINT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  name TEXT NOT NULL UNIQUE
);

CREATE TABLE app.note_tags (
  note_id BIGINT NOT NULL REFERENCES app.notes(id) ON DELETE CASCADE,
  tag_id BIGINT NOT NULL REFERENCES app.tags(id) ON DELETE CASCADE,
  PRIMARY KEY (note_id, tag_id)
);

CREATE INDEX notes_title_idx ON app.notes(title);
RESET ROLE;
```

Apply the runtime grants and default privileges from the permissions lesson. Set its path:

```
ALTER ROLE notes_app IN DATABASE notes_app
SET search_path = app, pg_catalog;
```

Install `pg`, set `DATABASE_URL`, and save this as `app.mjs`:

```
import pg from 'pg'
```

```
const pool = new pg.Pool({
  connectionString: process.env.DATABASE_URL,
  max: 5,
  connectionTimeoutMillis: 5000,
  statement_timeout: 10000,
  application_name: 'notes-lab',
})

try {
  const inserted = await pool.query(
    `INSERT INTO app.notes (title, body)
     VALUES ($1, $2)
     RETURNING id, title`,
    ['Plan the week', 'Choose three priorities']
  )

  const found = await pool.query(
    'SELECT id, title FROM app.notes WHERE id = $1',
    [inserted.rows[0].id]
  )

  console.log(found.rows[0])
} finally {
  await pool.end()
}
```

Run it with `node app.mjs`. Then connect as `notes_app` and confirm that `DROP TABLE app.notes` fails.

Inspect the title lookup with `EXPLAIN (ANALYZE, BUFFERS)`. Add an `archived_at TIMESTAMPTZ` column as an expand step, deploy code that tolerates it, and leave destructive cleanup for another migration.

Finally, dump `notes_app`, restore it into `notes_restore`, and query the inserted note there. The lab is complete when the runtime restriction, Node query, migration, plan, and restore all work.

Check your understanding: applications and operations

Check connection strings, pools, application transactions, migrations, restore tests, common errors, and hosting responsibilities.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/postgresql/.