Open your .env file and set your PostgreSQL connection string:

DATABASE_URL="postgresql://user:password@localhost:5432/yourdbname"

- Replace user, password, and yourdbname with your actual PostgreSQL credentials.

PostgreSQL Notes

---

1) Quick intro

- **PostgreSQL** = powerful open-source relational DBMS.
- Uses **SQL** (ANSI standard) plus many advanced features: `JSON/JSONB`, `ARRAY`, `UUID`, custom types, window functions, full-text search, stored procedures, etc.

2) Installation & tools (short)

- Windows / macOS: installers available (Postgres.app on macOS).
- Linux (Debian/Ubuntu): `sudo apt install postgresql postgresql-contrib`
- Tools: `psql` (CLI), **pgAdmin** (GUI), **DBeaver** (multi-DB GUI).

Start/stop (Linux example):

```
sudo systemctl start postgresql
sudo systemctl enable postgresql
```

Connect with psql as default user:

```
sudo -u postgres psql
-- or from local: psql -U youruser -d yourdb -h localhost -p 5432
```

3) psql meta-commands (very useful)

Run inside `psql`:

```
\l           -- list databases
\c dbname    -- connect to db
\dt          -- list tables
\d tablename -- describe table (columns, types, constraints)
\df          -- list functions
\?           -- help for psql commands
\q           -- quit
```

4) Create DB / User / Role / Permissions

```
-- create role (user) with password
CREATE ROLE sagar WITH LOGIN PASSWORD 'strongpass';

-- create database owned by role
CREATE DATABASE practice_db OWNER sagar;

-- grant privileges (if needed)
GRANT ALL PRIVILEGES ON DATABASE practice_db TO sagar;
```

---

5) Basic schema & CRUD (examples)

```
-- create table
CREATE TABLE students (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email TEXT UNIQUE,
  age INT,
  enrolled BOOLEAN DEFAULT TRUE,
  created_at TIMESTAMP DEFAULT NOW()
);

-- insert
INSERT INTO students (name, email, age) VALUES
('Sagar', 'sagar@example.com', 21),
('Maya', 'maya@example.com', 23);

-- select
SELECT * FROM students;
SELECT name, age FROM students WHERE age > 21 ORDER BY age DESC
LIMIT 5;

-- update
UPDATE students SET age = age + 1 WHERE id = 1;

-- delete
DELETE FROM students WHERE id = 2;
```

---

6) Important Data types

- `SERIAL` / `BIGSERIAL` — auto increment integer (backed by sequence)
- `INTEGER`, `BIGINT`, `SMALLINT`
- `VARCHAR(n)`, `TEXT`
- `BOOLEAN`
- `TIMESTAMP`, `DATE`, `TIME`
- `NUMERIC(p,s)` — precise decimals (money)
- `UUID` — universally unique id (use `gen_random_uuid()` from `pgcrypto` or `uuid-ossp`)
- `JSON`, `JSONB` — JSONB recommended for indexing and performance
- `ARRAY` — e.g., `TEXT[]`
- `ENUM` — custom enum types

Example JSONB:

```
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  data JSONB
);

INSERT INTO products (data) VALUES
('{"name":"shoe","sizes":[38,39,40],"stock":10}');
SELECT data->>'name' AS name FROM products;
```

---

7) Constraints & Keys

- `PRIMARY KEY`, `UNIQUE`, `NOT NULL`, `CHECK`, `FOREIGN KEY`

```
ALTER TABLE students ADD CONSTRAINT chk_age CHECK (age >= 0);
```

Foreign key:

```
CREATE TABLE courses (
  id SERIAL PRIMARY KEY,
  student_id INT REFERENCES students(id) ON DELETE CASCADE,
  course_name VARCHAR(100)
);
```

---

8) Indexes & when to use

- Create index to speed up queries on columns used in WHERE/JOIN/ORDER BY:

```
CREATE INDEX idx_students_name ON students(name);
```

- Use `EXPLAIN ANALYZE <query>` to see query plan and whether index used.
- Avoid useless indexes (costly on writes).

---

9) Joins & subqueries

```
-- inner join
SELECT s.name, c.course_name
FROM students s
JOIN courses c ON s.id = c.student_id;

-- left join
SELECT s.name, c.course_name
```

```
FROM students s
LEFT JOIN courses c ON s.id = c.student_id;

-- subquery
SELECT name FROM students WHERE id IN (SELECT student_id FROM
courses WHERE course_name='Math');
```

---

10) Aggregation, GROUP BY, HAVING

```
SELECT COUNT(*) AS total, AVG(age) AS avg_age FROM students;
SELECT student_id, COUNT(*) AS course_count FROM courses GROUP BY
student_id HAVING COUNT(*) > 1;
```

---

11) Window functions (powerful)

```
SELECT id, name, age,
       ROW_NUMBER() OVER (ORDER BY age DESC) AS rn,
       AVG(age) OVER () AS overall_avg
FROM students;
```

---

12) Transactions & concurrency

- Use transactions to group operations:

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
-- or ROLLBACK to cancel
```

- Isolation levels: `READ COMMITTED` (default), `REPEATABLE READ`, `SERIALIZABLE`. Use when needed to avoid race conditions.

---

13) Stored functions & procedures

- Create functions (SQL or PL/pgSQL):

```
CREATE FUNCTION add_numbers(a INT, b INT) RETURNS INT AS $$
BEGIN
  RETURN a + b;
END;
$$ LANGUAGE plpgsql;

SELECT add_numbers(2,3);
```

---

14) Views & materialized views

- View = virtual table

```
CREATE VIEW active_students AS SELECT id, name FROM students WHERE
enrolled;
```

- Materialized view = stores results (refresh manually)

```
CREATE MATERIALIZED VIEW mv_popular_courses AS
SELECT course_name, COUNT(*) cnt FROM courses GROUP BY course_name;
REFRESH MATERIALIZED VIEW mv_popular_courses;
```

---

15) Full-text search (basic)

```
ALTER TABLE articles ADD COLUMN tsv tsvector;
UPDATE articles SET tsv = to_tsvector(coalesce(title,'') || ' ' ||
coalesce(content,''));
CREATE INDEX idx_articles_tsv ON articles USING GIN(tsv);
```

```
SELECT * FROM articles WHERE tsv @@ to_tsquery('postgres &
tutorial');
```

---

16) JSONB tips

- Query nested JSONB items:

```
SELECT data->'sizes' AS sizes, data->>'name' AS name FROM products;
-- existence operator
SELECT * FROM products WHERE data @> '{"stock":10}';
```

- Index JSONB with GIN for containment queries:

```
CREATE INDEX idx_products_data ON products USING GIN (data);
```

---

17) Backup & restore

- Backup:

```
pg_dump -U sagar -d practice_db > practice_db.sql
-- for whole cluster: pg_dumpall > alldb.sql
```

- Restore:

```
psql -U sagar -d practice_db < practice_db.sql
```

- Binary backup via `pg_basebackup` for replication.

---

18) Roles & security best practices

- Use roles with least privileges.
- Don’t use superuser for app connections.
- Use `pg_hba.conf` to configure authentication (md5, scram-sha-256).
- Keep backups, use SSL for remote connections.

---

19) Useful built-in functions & operators (cheat sheet)

- Date/time: `NOW()`, `CURRENT_DATE`, `AGE(timestamp)`
- String: `LENGTH()`, `UPPER()`, `LOWER()`, `CONCAT()`
- Aggregates: `COUNT()`, `SUM()`, `AVG()`, `MIN()`, `MAX()`
- JSON: `->`, `->>`, `#>` (path), `@>` (contains)
- Array: `array_length`, `unnest()`
- Sequence: `nextval('seqname')`, `currval()`

---

20) Performance tips

- Index columns used frequently in WHERE/JOIN.
- Use `EXPLAIN ANALYZE` to understand slow queries.
- Be cautious with `SELECT *` on large tables.
- Use connection pooling (PgBouncer) for web apps.

- Vacuum and analyze: `VACUUM` / `VACUUM ANALYZE` to maintain stats (or enable autovacuum).

---

21) Example mini schema + queries (put it all together)

```
-- create users and posts
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  username VARCHAR(50) UNIQUE NOT NULL,
  joined TIMESTAMP DEFAULT NOW()
);

CREATE TABLE posts (
  id SERIAL PRIMARY KEY,
  user_id INT REFERENCES users(id),
  title TEXT,
  body TEXT,
  tags TEXT[],
  created_at TIMESTAMP DEFAULT NOW()
);

-- insert sample
INSERT INTO users (username) VALUES ('sagar'), ('maya');
INSERT INTO posts (user_id, title, tags) VALUES (1, 'Intro to PG',
ARRAY['postgres','sql']), (2, 'JS Tips', ARRAY['javascript']);

-- query: top posters
SELECT u.username, COUNT(p.id) AS posts
FROM users u
LEFT JOIN posts p ON u.id = p.user_id
GROUP BY u.username ORDER BY posts DESC;
```

---

22) Common pitfalls & notes

- `SERIAL` creates a sequence — if you restore and insert explicit ids, sequence may need `setval()` to avoid conflicts.
- `TEXT` vs `VARCHAR(n)` — `TEXT` has no length limit. `VARCHAR(n)` enforces max length.
- Case sensitivity: unquoted identifiers are folded to lowercase. Use double quotes `"MyTable"` to preserve case (avoid this unless needed).
- JSONB is good, but if data has relational nature prefer normalized tables.

---

23) Learning practice suggestions

- Recreate the example mini schema and practice queries.
- Use `EXPLAIN ANALYZE` on slow queries.
- Build a small CRUD app (Node.js + `pg` or Python + `psycopg2`) to practice connections and transactions.
- Try one feature per day: JSONB, window functions, materialized views, etc.

---

24) Quick resources (no links here — search for these)

- Official PostgreSQL docs (search "PostgreSQL documentation") — excellent, definitive.
- Tutorials: "Postgres tutorial" / "psql cheat sheet" / "Postgres JSONB examples".
- Practice: use pgAdmin or DBeaver and a toy database.
