Postgres

(https://projects.100xdevs.com/tracks/YOSAherHkqWXhOdlE4yE/sql-9)

What we will learn today?

Simple - SQL vs NoSQL, how to create Postgres Databases, How to do CRUD on them

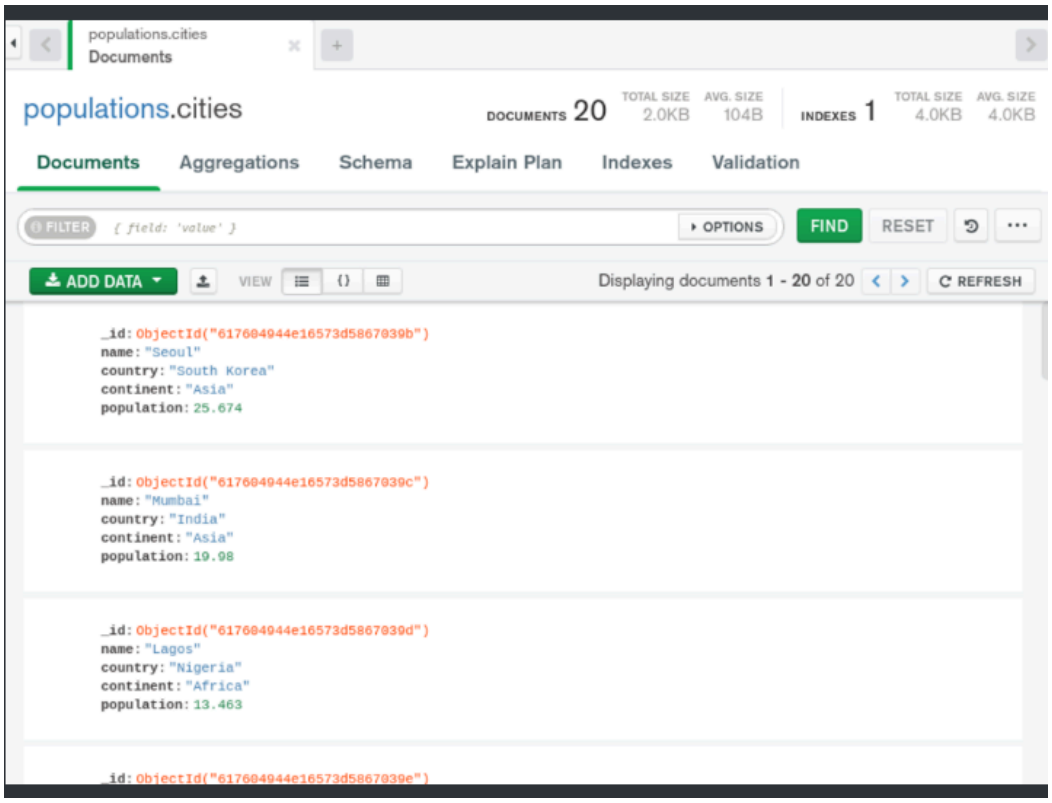
Advance - Relationships, Joins, Transactions

Types of Databases

There are a few type of databases, all service different types of use-cases

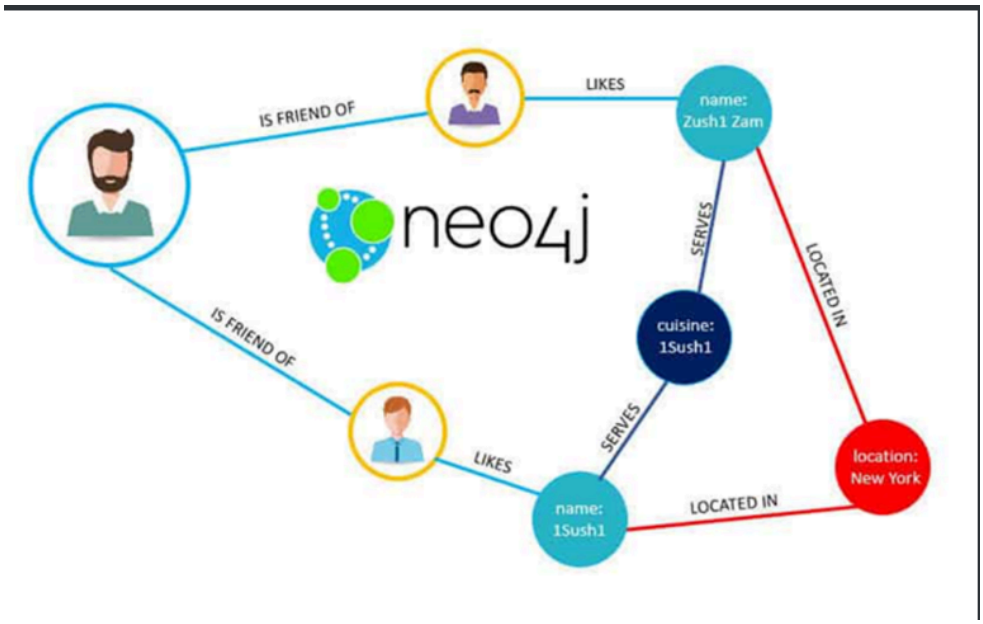
NoSQL Databases

1. Store data in a schema-less fashion. Extremely lean and fast way to store data.
2. Examples - MongoDB



Graph databases

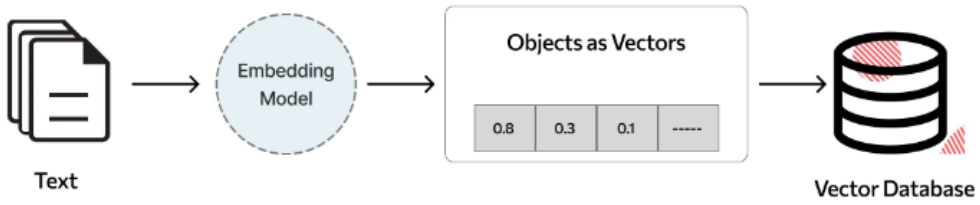
1. Data is stored in the form of a graph. Specially useful in cases where **relationships** need to be stored (social network)
2. Social network Website
3. Examples - Neo4j



Vector databases

1. Stores data in the form of vectors
2. Useful in Machine learning
3. Examples - Pinecone

Very useful for AI applications



Example : talk to pdf application

SQL databases

1. Stores data in form of rows
2. Most full stack applications will use this
3. Examples - MySQL, Postgres

Table: customers

customer_id	first_name	last_name	phone	country
1	John	Doe	817-646-8833	USA
2	Robert	Luna	412-862-0502	USA
3	David	Robinson	208-340-7906	UK
4	John	Reinhardt	307-242-6285	UK
5	Betty	Taylor	806-749-2958	UAE

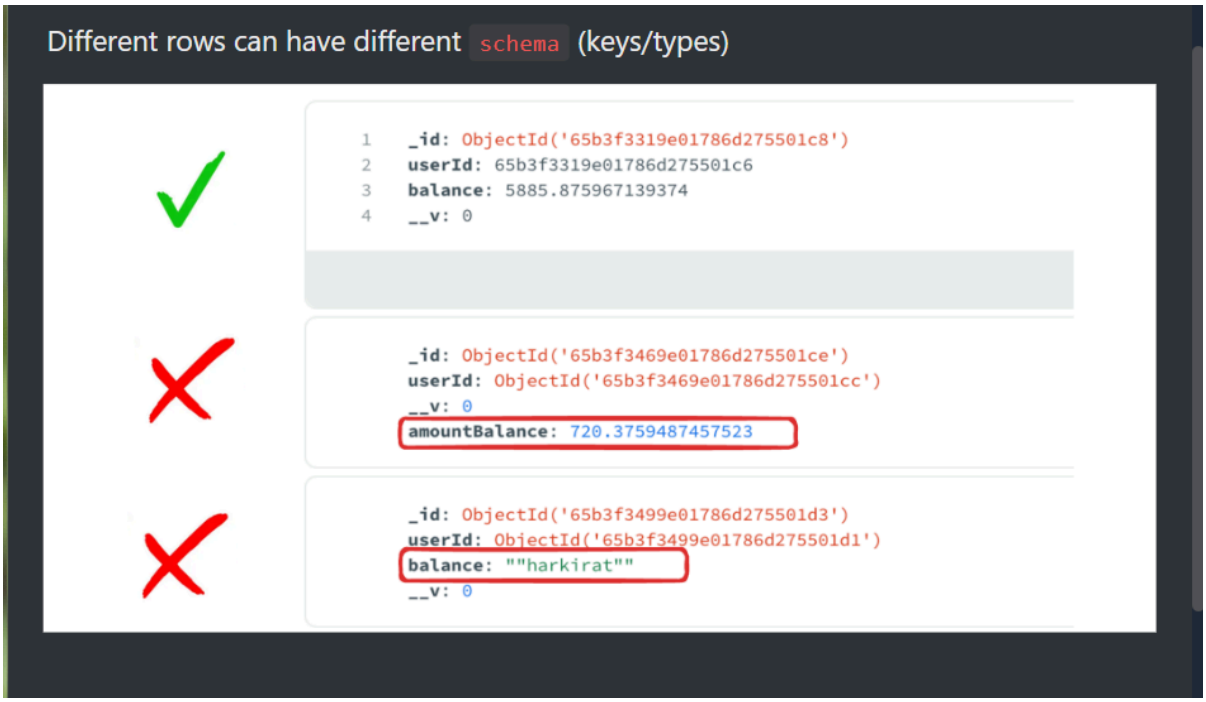
They are very strict.

We don’t usually store array etc in it

Why not NoSQL

MongoDB schemaless properties make it ideal to for bootstrapping a project fast. But as our app grows, this property makes it very easy for data to get corrupted Different rows can have different schema.

What is Schemaless?



Problems?

- Can lead to inconsistent database
- Can cause runtime errors
- Is too flexible for an app that needs strictness.

Upsides?

- Can move very fast
- Can change schema very easily.

You might think that `mongoose` does add strictness to the codebase because we used to define a schema there.
That strictness is present at the Node.js level, not at the DB level. You can still put in erroneous data in the database that doesn't follow that schema.

We can still put an erroneous data, Our database is not resisting
Our database should reject the erroneous data.

Our database should reject bad things.

Why SQL??

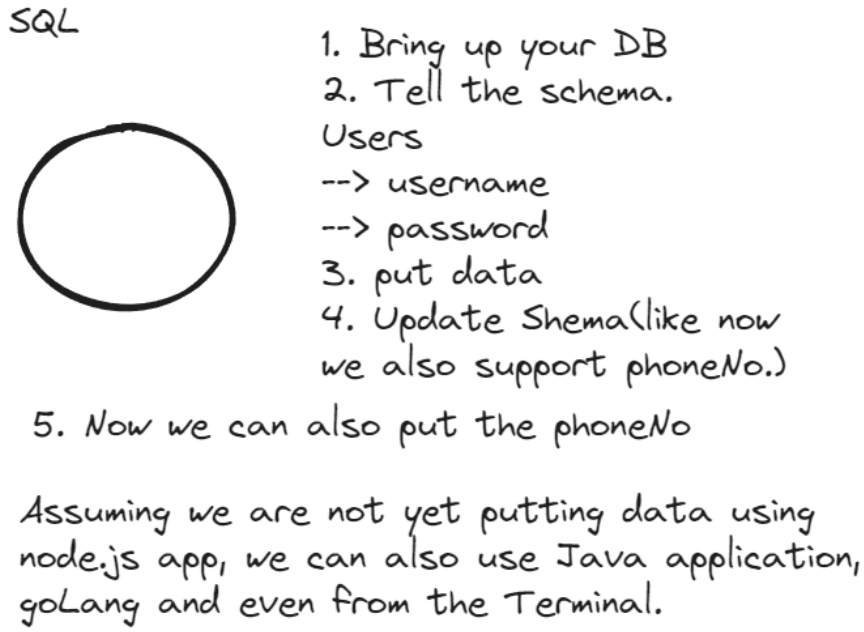
SQL databases have a strict schema. They require you to

1. Define your schema upfront

Whenever we have SQL we have to define shape of our application

2. Put in data that follows that schema
3. Update the schema as your app changes and perform migrations

Update your schema as application and user grows . We need to perform migrations and tell database that we have another column



So there are 4 parts when using an SQL database(not connecting it to Node.js,just running it and putting data in it.)

1. Running the database.
2. Using a library that let's you connect and put data in it.
3. Creating a table and defining its schema.
4. Run queries on the database to interact with the data (Insert/Update/Delete)

Let's Bootstrap simple SQL database

Using docker on windows

How to run postgrSQL in windows terminal(if you have docker installed).

- fist run docker gui application that help in running commands in terminal.
- After that run it with the docker instance by the help of following command .

-- for the first time if the image is not downloaded .

-- **docker run --name my-postgres1 -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres**.

-- if the docker image is there, prior to use the it can simply be runned by **docker run <image name>**.

- After that ,

-- use **docker exec -it my-postgres1 psql -U postgres -d postgres** this command in terminal .

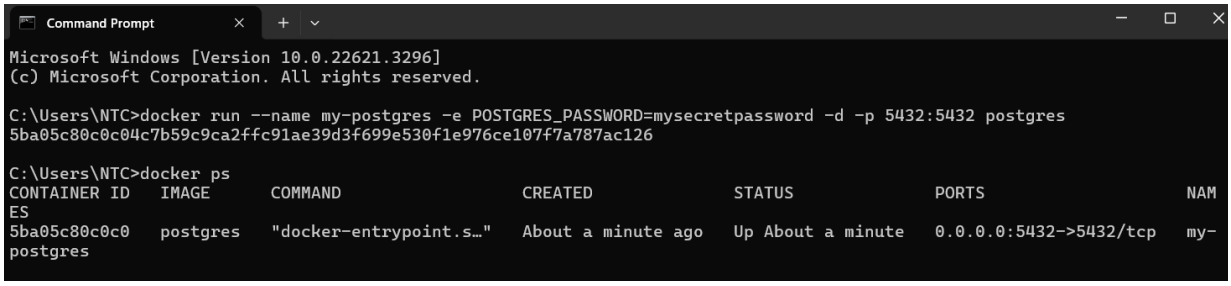
-- then enter the password and it will connect to localhost Postgress instance .

-- now you will be inside the postress command line that looks like **postgres-#** .

- U can check it by running **\\dt** , (the command to display all the tables.)

(will start the postgress locally)

docker ps to check whether their is something running on docker or not



Connection string

postgresql://postgres:mysecretpassword@localhost:5432/postgres?sslmode=disable

Neon DB

postgresql://neondb_owner:tWcwiYRK8A3U@ep-fancy-pine-a5vggaa1.us-east-2.aws.neon.tech/neondb?sslmode=require

User questions field

form_id	question_id	title
1	1	What is 2+2?

Options

Question_id	option_id	title

Using a library that let's you connect and put data in it.

1. **psql**

psql is a terminal-based front-end to PostgreSQL. It provides an interactive command-line interface to the PostgreSQL (or TimescaleDB) database. With psql, you can type in queries interactively, issue them to PostgreSQL, and see the query results.

CLI , (Command line interface/ Terminal)

If we installl postgress then we will get access to the psql

How to connect to your database?

psql Comes bundled with postgresql. You don't need it for this tutorial. We will directly be communicating with the database from Node.js

psql -h p-broken-frost-69135494.us-east-2.aws.neon.tech -d database1 -U 100xdevs

Running psql from docker

```
C:\Users\NTC>docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED
STATUS              PORTS                     NAMES
5ba05c80c0c0   postgres   "docker-entrypoint.s…"   About a minute ago
Up About a minute   0.0.0.0:5432->5432/tcp   my-postgres

C:\Users\NTC>docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED         STATU
S         PORTS                     NAMES
5ba05c80c0c0   postgres   "docker-entrypoint.s…"   6 minutes ago   Up 6
minutes   0.0.0.0:5432->5432/tcp   my-postgres

C:\Users\NTC>
```

Copy the container id 5ba05c80c0c0

docker exec -it e5f25d072f66 /bin/bash

```
C:\Users\NTC>docker exec -it 5ba05c80c0c0 /bin/bash
root@5ba05c80c0c0:/#
```

Now i am inside the container, we ssh into the server

We have psql inside the container

Now lets connect psql to the database

```
root@5ba05c80c0c0:/# psql
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQ
L.5432" failed: FATAL:  role "root" does not exist
root@5ba05c80c0c0:/#
```

psql -h localhost -d postgres -U postgres

```
root@5ba05c80c0c0:/# psql -h localhost -d postgres -U postgres
psql (16.2 (Debian 16.2-1.pgdg120+2))
Type "help" for help.

postgres=#
```

Now we can run commands specific to psql shell

Example : \dt\

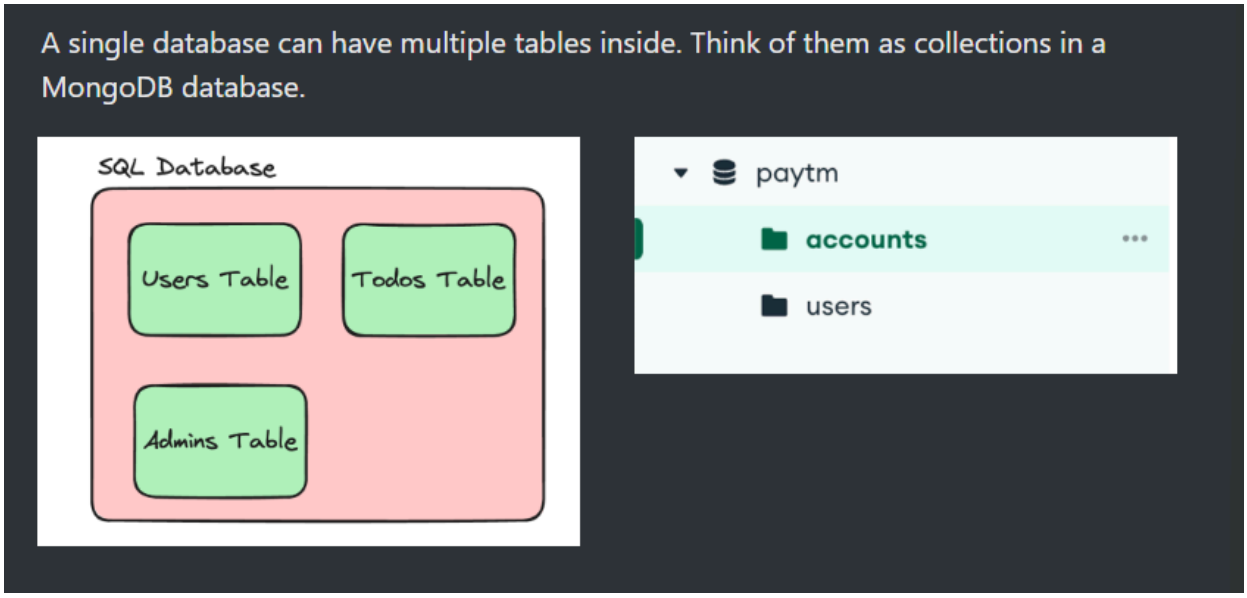
```
postgres=# \dt
Did not find any relations.
postgres=#
```

2. **pg**

pg is a **node.js** library that you can use in your backend app to store data in the Postgress DB (similar to mongoose).

Creating a table and defining it's schema

Tables in SQL



Until now, we have database that we can interact with. The next step in case of postgres is to define the **schema** of your tables.

SQL stands for Structured query language. It is a language in which you can describe what/how you want to put data in the database. (How to talk to database)

To create a table, the command to run is -

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
```

```
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
```

```
Did not find any relations.
postgres=# CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE
postgres=#
```

```
postgres=# \dt
         List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | users | table | postgres
(1 row)

postgres=#
```

There are a few parts of this SQL statement , let's decode them one by one.

1. CREATE TABLE users

CREATE TABLE users : this command initiates the creation of a new table in the database named **users.**

2. id SERIAL PRIMARY KEY

- **id**: The name of the first column in the **users** table, typically used as a unique identifier for each row (user). Similar to **_id** in mongoDB
- **SERIAL** : A PostgreSQL -specific data type for creating an auto-increment integer. Every time a new row is inserted, this value automatically increments, ensuring each user has a unique **id.**

- **PRIMARY KEY:** This constraint specifies that the **id** column is the primary key for the table, meaning it uniquely identifies each row. Values in this column must be unique and not null.
  Whenever creating a table we have to define a primary key which uniquely identifies a user.
  User might want to change his phoneNo, email etc.

3. email VARCHAR(255) UNIQUE NOT NULL,

- **email**: The name of the second column, intended to store the user's username.
- **VARCHAR(50)**: A variable character string data type that can store up to 50 characters. It's used here to limit the length of the username.
- **UNIQUE**: This constraint ensures that all values in the **username** column are unique across the table. No two users can have the same username.
- **NOT NULL**: This constraint prevents null values from being inserted into the **username** column. Every row must have a username value.

4. password VARCHAR(255) NOT NULL
   Can be non unique

5. Created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP

- **created_at**: The name of the fifth column, intended to store the timestamp when the user was created.
- **TIMESTAMP WITH TIME ZONE**: This data type stores both a timestamp and a time zone, allowing for the precise tracking of when an event occurred, regardless of the user's or server's time zone.

- **DEFAULT CURRENT_TIMESTAMP:** This default value automatically sets the **created_at** column to the date and time at which the row is inserted into the table, using the current timestamp of the database server.

Interacting with the Database

There are 4 things you'd like to do with a database

1. **INSERT**

```
INSERT INTO users (username, email, password)
VALUES ('username_here', 'user@example.com', 'user_password');
```

‘

We didn't have to specify the id because it auto increments.

2. **UPDATE**

```
UPDATE users
SET password = 'new_password'
WHERE email = 'user@example.com';
```

3. **DELETE**

```
DELETE FROM users
WHERE id = 1;
```

4. **SELECT**

```
SELECT * FROM users
WHERE id = 1;
```

```
postgres=# INSERT INTO users (username, email, password)
VALUES ('username12_here', 'user1@exampl2e.com', 'user_password12');
INSERT 0 1
postgres=# \dt
         List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | users | table | postgres
(1 row)

postgres=# SELECT * FROM users;
 id |    username     |       email        |    password     |          created_at
----+-----------------+--------------------+-----------------+-------------------------------
  2 | username1_here  | user@exampl2e.com  | user_password1  | 2024-03-30 18:37:01.711504+00
  1 | username_here   | user@example.com   | new_password    | 2024-03-30 18:32:47.450488+00
  3 | username12_here | user1@exampl2e.com | user_password12 | 2024-03-30 18:38:46.593723+00
(3 rows)

postgres=#
```

Whenever we say delete in we will basically create a new row .

A real full stack will do this from Node.js

How to do queries from a Node.js app?

In the end, postgres exposes a protocol that someone needs to talk to be able to send these commands (update, delete) to the database.

psql is one such library that takes commands from your terminal and sends it over to the database.

To do the same in a Node.js , you can use one of many **Postgres clients**

pg library

https://www.npmjs.com/package/pg

Non-Blocking(it will not block when we insert something) PostgreSQL client for Node.js

Documentation: https://node-postgres.com/

Connecting

```
import { Client } from 'pg'

const client = new Client({
  host: 'my.database-server.com',
  port: 5334,
```

```
  database: 'database-name',
  user: 'database-user',
  password: 'secretpassword!!',
})

client.connect()
```

Querying:

```
const result = await client.query('SELECT * FROM USERS;')
console.log(result)
```

```
// write a function to create a users table in your database.
import { Client } from 'pg'

const client = new Client({
  connectionString:
"postgresql://postgres:mysecretpassword@localhost/postgres"
})

async function createUsersTable() {
    await client.connect()
    const result = await client.query(`
        CREATE TABLE users (
            id SERIAL PRIMARY KEY,
            username VARCHAR(50) UNIQUE NOT NULL,
            email VARCHAR(255) UNIQUE NOT NULL,
            password VARCHAR(255) NOT NULL,
            created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
        );
    `)
    console.log(result)
}

createUsersTable();
```

Creating a simple Node.js app

1. Initialize an empty typescript project

npm init -y

npx tsc --init

npm i pg

2. Change the rootDir and outDir in tsconfig.json

"rootDir": "./src",

"outDir": "./dist",

3. Install the pg library and it's types (because we are using TS)

npm install pg

npm install @types/pg

4. Create a simple Node.js app that lets you put data

- CDN does scalability
- Indexing in database , What algorithm to be used.

```
//  Write a function to create a users table in a Database
import { Client } from 'pg'

const client = new Client({

connectionString:"postgresql://postgres:mysecretpassword@localhost/postgre
s"
})

// `` help us write multiple line string

async function createUsersTable() {
    await client.connect()
```

```
    const result = await client.query(`
        CREATE TABLE users (
            id SERIAL PRIMARY KEY,
            username VARCHAR(50) UNIQUE NOT NULL,
            email VARCHAR(255) UNIQUE NOT NULL,
            password VARCHAR(255) NOT NULL,
            created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
        );
    `)
    console.log(result)
}

createUsersTable();
```

```
C:\Users\NTC\Desktop\Dev\Week12>node dist/index.js
Result {
  command: 'CREATE',
  rowCount: null,
  oid: null,
  rows: [],
  fields: [],
  _parsers: undefined,
  _types: TypeOverrides {
    _types: {
      getTypeParser: [Function: getTypeParser],
      setTypeParser: [Function: setTypeParser],
      arrayParser: [Object],
      builtins: [Object]
    },
    text: {},
    binary: {}
  },
  RowCtor: null,
  rowAsArray: false,
  _prebuiltEmptyResultObject: null
}
```

What if we try to again create the table.

We will receive an error message

```
C:\Users\NTC\Desktop\Dev\Week12>node dist/index.js
C:\Users\NTC\Desktop\Dev\Week12\node_modules\pg\lib\client.js:526
          Error.captureStackTrace(err);
                ^

error: relation "users2" already exists
    at C:\Users\NTC\Desktop\Dev\Week12\node_modules\pg\lib\client.js:526:17
    at process.processTicksAndRejections (node:internal/process/task_queues
:95:5) {
  length: 100,
  severity: 'ERROR',
  code: '42P07',
  detail: undefined,
  hint: undefined,
  position: undefined,
  internalPosition: undefined,
  internalQuery: undefined,
  where: undefined,
  schema: undefined,
  table: undefined,
  column: undefined,
  dataType: undefined,
  constraint: undefined,
  file: 'heap.c',
  line: '1146',
  routine: 'heap_create_with_catalog'
}
```

Create a function that let's you insert data into a table. Make it async, make sure client.connect resolves before u do the insert.

Connection string has all the details of ours and what database we are connecting to.

We give this to psql too.

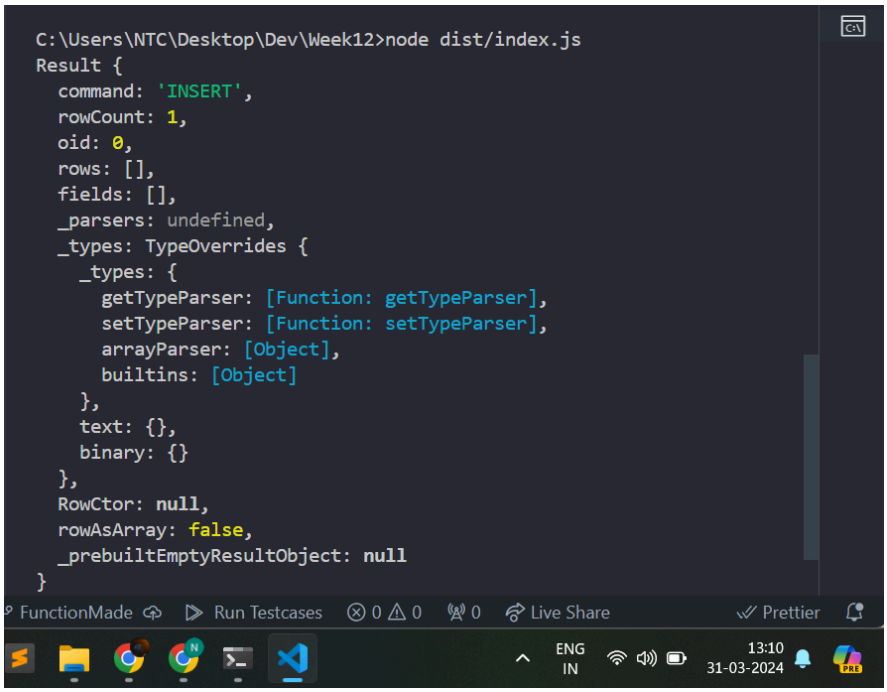
```
//  Write a function to create a users table in a Database
import { Client } from 'pg'

const client = new Client({

connectionString:"postgresql://postgres:mysecretpassword@localhost/postgre
s"
})

async function insertData() {
    try{
        await client.connect() //Ensure client connection is establishing
        const insertQuery = "INSERT INTO users (username, email, password)
VALUES ('username2','user@ascdv.com','user_password')";
        const res = await client.query(insertQuery);
```

```
        console.log('Insertion success',res);//Output insertion result
    }catch(err){
        console.error('Error during the insertion',err);
    }finally{
        await client.end();//Close the client connection
    }
}
insertData();
```




```
// write a function to create a users table in your database.
import { Client } from 'pg'

const client = new Client({
  connectionString:
"postgresql://postgres:mysecretpassword@localhost/postgres"
})

async function insertUsersData(username:string,
password:string,email:string) {
    await client.connect()
    const result = await client.query(`
       INSERT INTO users2 (username, password, email)
```

```
        VALUES ('${username}','${password}','${email}')
    `)
    console.log(result)
    // this is insecure way of writing query
}

insertUsersData(
    "user",
    "123",
    "nishantthapa@asffvdsa.com"
)
```

Username, password and email are provided by user.

If someone come and give erroneous input like

“; DELETE * FROM users”

This is called SQL Injection.

(should’nt pass as SQL syntax)

This is an **insecure** way to store data in your tables.

When you expose this functionality eventually via HTTP someone can do an SQL INJECTION to get access to your data/ delete your data.

More secure way to store data.

Update the code so you don’t put user provided fields in the SQL string

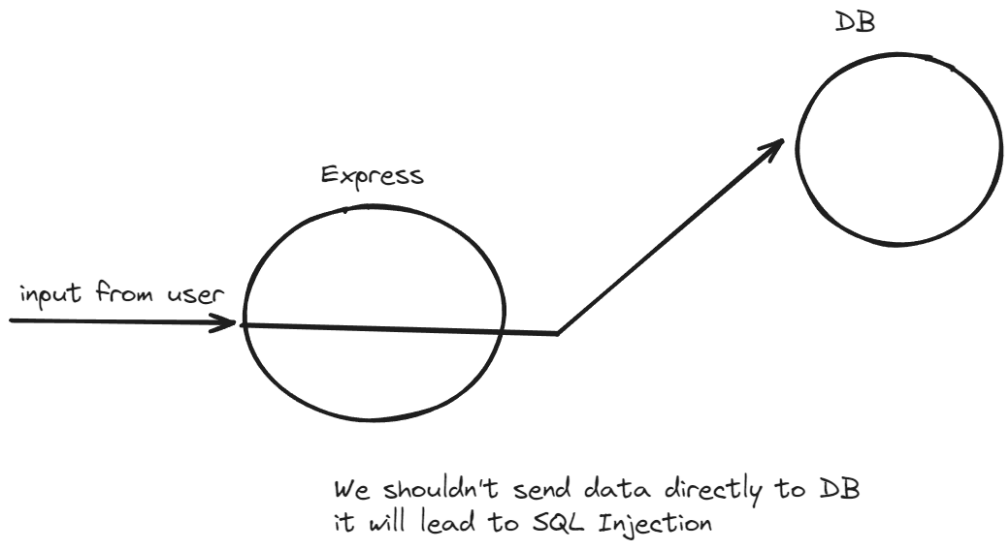
To avoid this we should use different syntax

const result = client.query(`

INSERT INTO users (username, password, email)

VALUES ($1,$2,$3)

`,[username, password, email])



What are user provided strings?

In your final app, this insert statement will be done when a user signs up on your app.

Email, username, password are all user provided strings

What is SQL string??

const insertQuery = "INSERT INTO users (username, email, password) VALUES ('username2', 'user3@example.com', 'user_password');";

const insertQuery = 'INSERT INTO example_table(column1, column2) VALUES($1, $2)';

const res = await client.query(insertQuery, [column1Value, column2Value]);

Solution:

```
import { Client } from 'pg';

// Async function to insert data into a table
async function insertData(username: string, email: string, password: string) {
```

```
  const client = new Client({
    host: 'localhost',
    port: 5432,
    database: 'postgres',
    user: 'postgres',
    password: 'mysecretpassword',
  });

  try {
    await client.connect(); // Ensure client connection is established
    // Use parameterized query to prevent SQL injection
    const insertQuery = "INSERT INTO users (username, email, password)
VALUES ($1, $2, $3)";
    const values = [username, email, password];
    const res = await client.query(insertQuery, values);
    console.log('Insertion success:', res); // Output insertion result
  } catch (err) {
    console.error('Error during the insertion:', err);
  } finally {
    await client.end(); // Close the client connection
  }
}

// Example usage
insertData('username5', 'user5@example.com',
'user_password').catch(console.error);
```

Query Data

Write a function getUser That lets you fetch data from he database given a email as input

```
import { Client } from 'pg';

// Async function to fetch user data from the database given an email
async function getUser(email: string) {
    const client = new Client({
        host: 'localhost',
        port: 5432,
        database: 'postgres',
```

```
        user: 'postgres',
        password: 'mysecretpassword',
    });


  try {
    await client.connect(); // Ensure client connection is established
    const query = 'SELECT * FROM users WHERE email = $1';
    const values = [email];
    const result = await client.query(query, values);

    if (result.rows.length > 0) {
      console.log('User found:', result.rows[0]); // Output user data
      return result.rows[0]; // Return the user data
    } else {
      console.log('No user found with the given email.');
      return null; // Return null if no user was found
    }
  } catch (err) {
    console.error('Error during fetching user:', err);
    throw err; // Rethrow or handle error appropriately
  } finally {
    await client.end(); // Close the client connection
  }
}

// Example usage
getUser('user3@example.com').catch(console.error);
```

Relationships and Transactions

Relationship let you store data in different tables and relate it with each other.

Relationship in MongoDB

Since mongoDb is noSQL database you can store any shape of data in it.

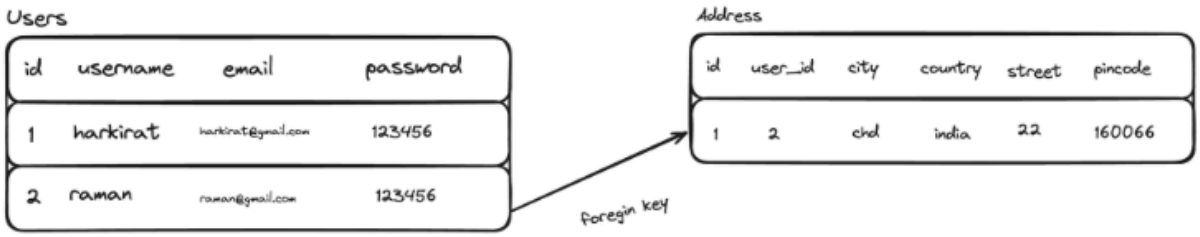
If i ask you to store a users details along with their address, you can store it in an object that has the address details

```
_id: ObjectId('65b3f3469e01786d275501ce')
address: Object
    street: "1 West HW"
    city: "Chandigarh"
    country: "India"
    pincode: "160066"
email: "harkirat@gmail.com"
name: "harkirat"
```

User having multiple address. It will be array of objects. We can shove entire into single row

Relationships in SQL

Since SQL can not store objects as such, we need to define two different tables to store this data in.



This is called a **relationship,** which means that the **Address** table is related to the **Users** table.

We can shove everything of address in a table , but this approach fails when a single user has multiple address

When defining the table, you need to define the relationship

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
```

```
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE addresses (
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL,
    city VARCHAR(100) NOT NULL,
    country VARCHAR(100) NOT NULL,
    street VARCHAR(255) NOT NULL,
    pincode VARCHAR(20),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

Relationship and foreign keys .

Foreign key : key in second table which help us uniquely identify in table 1.

SQL query

To insert the address of a user –

```
INSERT INTO addresses (user_id, city, country, street, pincode)
VALUES (1, 'New York', 'USA', '123 Broadway St', '10001');
```

Now if you want to get the address of a user given an id, you can run the following query -

```
SELECT city, country, street, pincode
FROM addresses
WHERE user_id = 1;
```

Extra - Transactions in SQL

Good question to have at this point is what queries are run when the user signs up and sends both their information and their address in a single request.

Do we send two SQL queries into the database? What if one of the queries (address query for example) fails?
This would require transactions in SQL to ensure either both the user information and address goes in, or neither does

SQL Query

```sql
BEGIN; -- Start transaction

INSERT INTO users (username, email, password)
VALUES ('john_doe', 'john_doe1@example.com', 'securepassword123');

INSERT INTO addresses (user_id, city, country, street, pincode)
VALUES (currval('users_id_seq'), 'New York', 'USA', '123 Broadway St',
'10001');

COMMIT;
```

We say to SQL that we will send many SQL query to you don't just commit when you get a SQL query

Node js

```ts
import { Client } from 'pg';

async function insertUserAndAddress(
    username: string,
    email: string,
    password: string,
    city: string,
    country: string,
    street: string,
    pincode: string
) {
    const client = new Client({
        host: 'localhost',
        port: 5432,
        database: 'postgres',
        user: 'postgres',
```

```
        password: 'mysecretpassword',
    });

    try {
        await client.connect();

        // Start transaction
        await client.query('BEGIN');

        // Insert user
        const insertUserText = `
            INSERT INTO users (username, email, password)
            VALUES ($1, $2, $3)
            RETURNING id;
        `;
        const userRes = await client.query(insertUserText, [username,
email, password]);
        const userId = userRes.rows[0].id;

        // Insert address using the returned user ID
        const insertAddressText = `
            INSERT INTO addresses (user_id, city, country, street,
pincode)
            VALUES ($1, $2, $3, $4, $5);
        `;
        await client.query(insertAddressText, [userId, city, country,
street, pincode]);

        // Commit transaction
        await client.query('COMMIT');

        console.log('User and address inserted successfully');
    } catch (err) {
        await client.query('ROLLBACK'); // Roll back the transaction on
error
        console.error('Error during transaction, rolled back.', err);
        throw err;
    } finally {
        await client.end(); // Close the client connection
    }
```

```
}

// Example usage
insertUserAndAddress(
    'johndoe',
    'john.doe@example.com',
    'securepassword123',
    'New York',
    'USA',
    '123 Broadway St',
    '10001'
);
```

Joins

Defining relationship is easy.

What's hard is joining data from two(or more) tables together.

For example? If I ask you to fetch me a users details and their address, what SQL would you run??

Approach 1:

```
-- Query 1: Fetch user's details
SELECT id, username, email
FROM users
WHERE id = YOUR_USER_ID;

-- Query 2: Fetch user's address
SELECT city, country, street, pincode
FROM addresses
WHERE user_id = YOUR_USER_ID;
```

We don't know that anything has changed between the Query 1 and Query 2

Approach 2:

```
SELECT users.id, users.username, users.email, addresses.city,
addresses.country, addresses.street, addresses.pincode
```

```
FROM users
JOIN addresses ON users.id = addresses.user_id
WHERE users.id = '1';
```

```
SELECT u.id, u.username, u.email, a.city, a.country, a.street, a.pincode
FROM users u
JOIN addresses a ON u.id = a.user_id
WHERE u.id = YOUR_USER_ID;
```

Now lets try converting the same in our node app

Approach 1 (not good)

```
import { Client } from 'pg';
// Async function to fetch user details and address separately
async function getUserDetailsAndAddressSeparately(userId: string) {
    const client = new Client({
        host: 'localhost',
        port: 5432,
        database: 'postgres',
        user: 'postgres',
        password: 'mysecretpassword',
    });
    try {
        await client.connect();
        // Fetch user details
        const userDetailsQuery = 'SELECT id, username, email FROM users
WHERE id = $1';
        const userDetails = await client.query(userDetailsQuery,
[userId]);
        // Fetch user address
        const userAddressQuery = 'SELECT city, country, street, pincode
FROM addresses WHERE user_id = $1';
        const userAddress = await client.query(userAddressQuery,
[userId]);
        if (userDetails.rows.length > 0) {
            console.log('User found:', userDetails.rows[0]);
            console.log('Address:', userAddress.rows.length > 0 ?
userAddress.rows[0] : 'No address found');
```

```
            return { user: userDetails.rows[0], address:
userAddress.rows.length > 0 ? userAddress.rows[0] : null };
        } else {
            console.log('No user found with the given ID.');
            return null;
        }
    } catch (err) {
        console.error('Error during fetching user and address:', err);
        throw err;
    } finally {
        await client.end();
    }
}
getUserDetailsAndAddressSeparately("1");
```

Approach 2:

```
import { Client } from 'pg';

// Async function to fetch user data and their address together
async function getUserDetailsWithAddress(userId: string) {
    const client = new Client({
        host: 'localhost',
        port: 5432,
        database: 'postgres',
        user: 'postgres',
        password: 'mysecretpassword',
    });

    try {
        await client.connect();
        const query = `
            SELECT u.id, u.username, u.email, a.city, a.country, a.street,
a.pincode
            FROM users u
            JOIN addresses a ON u.id = a.user_id
            WHERE u.id = $1
        `;
        const result = await client.query(query, [userId]);
```

```
        if (result.rows.length > 0) {
            console.log('User and address found:', result.rows[0]);
            return result.rows[0];
        } else {
            console.log('No user or address found with the given ID.');
            return null;
        }
    } catch (err) {
        console.error('Error during fetching user and address:', err);
        throw err;
    } finally {
        await client.end();
    }
}
getUserDetailsWithAddress("1");
```

Benefits of using a join –

1. Reduced latency
2. Simplified Application logic
3. Transactional Integrity

Types of Join

1. **INNER JOIN**

Returns rows when there is at least one match in both tables. If there is no match, the rows are not returned. It's the most common type of join.

Use Case: Find All Users With Their Addresses. If a user hasn't filled their address, that user shouldn't be returned

```
SELECT users.username, addresses.city, addresses.country,
addresses.street, addresses.pincode
FROM users
INNER JOIN addresses ON users.id = addresses.user_id;
```

2. **LEFT JOIN**

Return all rows from the left table, and the matched rows from the right table.

Use case: to list all users from your databases alon with their address information (if theyve provided it)

. you'd use a LEFT JOIN. Users without an address will still appear in your query result, but the address fields will be NULL for them.

```
SELECT users.username, addresses.city, addresses.country,
addresses.street, addresses.pincode
FROM users
LEFT JOIN addresses ON users.id = addresses.user_id;
```

3. **RIGHT JOIN**

Returns all rows from the right table, and the matched rows from the left table.

Use case - Given the structure of the database, a RIGHT JOIN would be less common since the addresses table is unlikely to have entries not linked to a user due to the foreign key constraint. However, if you had a situation where you start with the addresses table and optionally include user information, this would be the theoretical use case.

```
SELECT users.username, addresses.city, addresses.country,
addresses.street, addresses.pincode
FROM users
RIGHT JOIN addresses ON users.id = addresses.user_id;
```

4. **FULL JOIN**

Returns rows when there is a match in one of the tables. It effectively combines the results of both LEFT JOIN and RIGHT JOIN.

Use case - A FULL JOIN would combine all records from both users and addresses, showing the relationship where it exists. Given the constraints, this might not be as relevant because every address should be linked to a user, but if there were somehow orphaned records on either side, this query would reveal them.

```
SELECT users.username, addresses.city, addresses.country,
addresses.street, addresses.pincode
FROM users
```

```
FULL JOIN addresses ON users.id = addresses.user_id;
```