	SQL	SQL Database	95	
	Node	SQL with Node	121	
	Node	SQL + Node (CRUD op.)	131	
		(+ Express)		

SQL Database

* Database

→ It is a collection of data in a format that can be easily accessed.

→ Benefits:
- can store large data
- features like security, scalability, etc.
- easier to insert, update or delete data

* SQL v/s NoSQL

SQL	NoSQL
Relational Database (data stored in Tables)	Non Relational Database (data stored in document/ key-value / graphs, etc.)
eg mySQL, Oracle, PostgreSQL, etc	eg MongoDB, Cassandra, Neo4j, etc.

* SQL - Structured Query Language

→ SQL is a programming language used to interact with relational databases.

* Installation - MySQL

- visit www.mysql.com/downloads/
- scroll down to "MySQL Community (GPL) Downloads" and click on this link
- click on "MySQL Installer for Windows"
- click on "download" button (2.4mb msi file)
- click on "No thanks, just start my download" link

- "install" downloaded file
- select "full" & click on "Next"
- click on "Execute", "Execute", "Next", "Next", ...
- enter root password for when we have to access MySQL server
  (Here, I am using = mysql@12345)

- click on "Next", "Next", ..., "Execute", "Finish", "Next", ...
- enter root password we had just entered and
- click on "Next"
- click on "Execute", "Finish", "Execute"

- Uncheck "Start MySQL shell after setup"
- click on "Finish"

MySQL Workbench software

- On left side panel, first icon is for "MySQL Connections", click on ⊕ icon

- "setup New Connection" window will open
  connection Name = New Connection
  Password = click on "Store in Vault" button & enter root password.

- click on "Test Connection" button. It has to be dialog box for "successfully made the MySQL Connection" & click on "ok" button.

- click on connection, we have just made.

- Now, you can see some windows such as Navigator, Query 1 (file) with editor in which we will write commands (code), outputs, etc.

Note: To execute selected code in editor, press ctrl + enter

* Create a database

← database name
```
- CREATE DATABASE college;
```

→ In Navigator Panel ⇒ Schemas tab, click on ↻ (refresh) icon at the top-right corner, you can see our "college" named database.

* Delete a database

```
- DROP DATABASE college;
```

* Use a database

→ To create anything in the database, we have to select that database.

```
- USE college;
```

* Create a Table

```
- CREATE TABLE table_name (
    column_name1 datatype constraint,
    column_name2 datatype constraint,
    column_name3 datatype constraint
  );
```

e.g.

```
CREATE TABLE student (
    rollno INT,
    name VARCHAR(30),
    age INT
);
```

* Insert data into the table

```
- INSERT INTO student
  VALUES
  (101, "adam", 25),
  (102, "eve", 24);
```

* show Table

```
- SELECT * FROM student;
```

→ you can see student named table in "Result Grid" window

```
rollno    name    age
101       adam    25
102       eve     24
```

* Database Queries

→ CREATE DATABASE db_name;
CREATE DATABASE IF NOT EXISTS db_name;

→ DROP DATABASE db_name;
DROP DATABASE IF EXISTS db_name;

→ SHOW DATABASES;

→ SHOW TABLES;

* Table Queries

→ Create
→ Insert
→ Update
→ Alter
→ Truncate
→ Delete

* Datatypes

→ CHAR = string (0-255), can store characters of fixed length
CHAR (50)

→ VARCHAR = string (0-255), can store characters of upto given length
VARCHAR (50)

→ BLOB = string (0-65535), can store binary large object
BLOB (1000)

→ INT = integer (-2,147,483,648 to 2,147,483,647)

INT

→ TINYINT = integer (-128 to 127). If we use "TINYINT UNSIGNED" then range will be (0 - 255).
→ TINYINT

→ BIGINT = integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
BIGINT

→ BIT = can store x-bit values. x can range from 1 to 64.
BIT (2)

→ FLOAT = decimal number with precision to 23 dijits
FLOAT

→ DOUBLE = decimal number with 24 to 53 digits
DOUBLE

→ BOOLEAN = boolean values 0 or 1
BOOLEAN

→ DATE = date in format of YYYY-MM-DD
ranging from 1000-01-01 to 9999-12-31
DATE

→ YEAR = year in 4 digits format ranging from
1901 to 2155
YEAR

* Constraints

→ Rules for data in the table

NOT NULL	column can not have a null value
UNIQUE	all values in column are different
DEFAULT	sets the default value of a column
CHECK	it can limit the values allowed in a column

e.g.

```
- name VARCHAR(30) NOT NULL

- email VARCHAR(50) UNIQUE

- following INT DEFAULT 0

- salary INT DEFAULT 25000

- age INT
  CONSTRAINT CHECK ( age >= 13 )

- CONSTRAINT age-check CHECK ( age >= 18 AND city = "Delhi")

- age INT CHECK ( age >= 18 )

- age INT,
  CHECK (age >= 18 )
```

* Key Constraints

→ Keys are special columns in the tables

⇒ Primary Key

→ Primary key constraint makes a column unique & not null but used only for one column.

→ It is a column (or set of columns) in a table that uniquely identifies each row (a unique id).

→ There is only 1 PK & it should be NOT NULL.

e.g.

```
- CREATE TABLE temp (
      id INT NOT NULL,
      PRIMARY KEY (id)
  );
```

e.g.

```
- CREATE TABLE temp (
      id INT PRIMARY KEY
  );
```

⇒ Foreign Key

→ Foreign Key prevents actions that would destroy links between tables

→ A Foreign Key is a column (or set of columns) in a table that refers to the primary key in another table.

→ FKs can have duplicate & null values.

→ There can be multiple FKs.

e.g.

```
- CREATE TABLE temp (
      temp-id  INT PRIMARY KEY,
      cust-id  INT,
      FOREIGN KEY (cust-id) REFERENCES  temp1(id)
  );
```

↑ id is PRIMARY KEY in temp1 table

* Visualise Tables (EER Diagram)

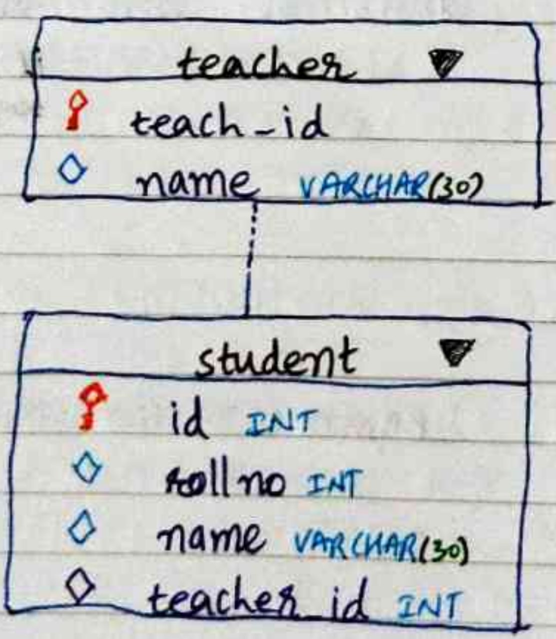
→ Database menu ⇒ Reverse Engineer ⇒ Next, Next, L.., Execute

e.g.

```
- CREATE TABLE teacher (
      teach-id  INT PRIMARY KEY,
      name  VARCHAR (30)
  );

  CREATE TABLE student (
      id  INT PRIMARY KEY,
      rollno  INT,
      name  VARCHAR (30),
      teacher-id  INT,
     FOREIGN KEY (teacher-id) REFERENCES teacher (teach-id)
  );
```

EER Diagram:



* Insert data into Table

```
- INSERT INTO user
  (id, age, name, email, followers, followings)
  VALUES
  (1, 14, "abc", "abc@email.com", 105, 10),
  (5, 22, "xyz", "xyz@email.com", 120, 50);
```

* Select Command

- selects & shows data from the DB.

  syntax : SELECT col1, col2 FROM table-name ;

  syntax (to show all) : SELECT * FROM table_name ;

⇒ DISTINCT keyword used to show distinct values from duplicate values from a table column

- SELECT DISTINCT age FROM user;

  output : If age column has 14, 15, 15, 14, 16 values then output will be :

```
14
15
16
```

* Where Clause

→ to define some conditions

syntax:

- SELECT col1, col2 FROM table-name
WHERE conditions;

e.g.
```
SELECT * FROM user
WHERE followers >= 200;
```

e.g.
```
SELECT name followers
FROM user
WHERE followers >= 200 ;
```

* Operators in Where Clause

→ Arithmetic Operators

+	addition
-	subtraction
*	multiplication
/	division
%	modulus

→ Comparison Operators

=, != , >, >=, <, <=

→ Logical Operators

AND, OR, NOT, IN, BETWEEN, ALL, LIKE, ANY

→ Bitwise Operators

&	Bitwise AND
|	Bitwise OR

* Logical Operators

AND to check for both conditions to be true
OR to check for one of the conditions to be true
BETWEEN selects for a given range
IN matches any value in the list
NOT to negate the given condition

e.g.
```
SELECT name, age
FROM user
WHERE age > 18  AND  followers > 200 ;
```

e.g.
```
SELECT name, age
FROM user
WHERE age BETWEEN 15 AND 17 ;
```
→ 15, 16, 17

e.g.
```
SELECT name, age, email
FROM user
WHERE email IN ("abc@email.com", "def@email.com");
```
↑ NOT to negate this condition

* Limit clause

→ sets an upper limit on number of (tuples) rows to be returned

```
- SELECT col1, col2
  FROM table_name
  LIMIT number;
```

e.g.
```
SELECT name, age, email
FROM user
LIMIT 2;
```

e.g.
```
SELECT *
FROM user
WHERE age > 14
LIMIT 3;
```

* Order by clause

→ To sort in ascending (ASC) or descending (DESC) order

```
- SELECT col1, col2
  FROM table_name
  ORDER BY col_name(s) ASC;
```

e.g.
```
SELECT * FROM user
ORDER BY followers DESC;
```

↑ here, if do not write anything then by default it is ASC.

* Aggregate Functions

→ Aggregate functions perform a calculation on a set of values, and returns a single value.

- COUNT()
- MAX()
- MIN()
- SUM()
- AVG()

eg SELECT max(followers) FROM user;

eg SELECT count(age) FROM user
   WHERE age > 18;

* Group By Clause

→ Groups rows that have the same values into summary rows.

→ It collects data from multiple records and groups the result by one or more column

- SELECT col1, col2 FROM table_name
  GROUP BY col_name(s);

→ Generally, we use group by clause with some aggregation function.

e.g.
```
SELECT age, count(id)
FROM user
GROUP BY age;
```

output:
```
14  2
15  3
16  2
17  1
```

* Having Clause

→ Similar to where; i.e. applies some conditions on rows, But it is used when we want to apply any conditions after grouping.

```
- SELECT col1, col2
  FROM table_name
  GROUP BY col_name(s)
  HAVING condition;
```

→ WHERE is for the table, HAVING is for a group
→ Grouping is necessary for HAVING.

e.g.
```
SELECT age, max(followers)
FROM user
GROUP BY age
HAVING max(followers) > 200;
```

* General Order

```
- SELECT column(s)
  FROM table-name
  WHERE condition
  GROUP BY column(s)
  HAVING condition
  ORDER BY column(s)   DESC;
```

* Update Table Rows

→ To update existing rows

```
- UPDATE table_name
  SET col1 = val1,   col2 = val2
  WHERE condition;
```

```
e.g. UPDATE user
     SET followers = 600
     WHERE age > 18;
```

→ If SQL gives an error when above code executes, then run following line of code :

```
SET SQL_SAFE_UPDATES = 0;
```

* Delete Table Rows

→ To delete existing rows

```
- DELETE FROM table_name
  WHERE condition;
```

eg
```
DELETE FROM user
WHERE age = 14;
```

* Alter Table Queries

→ To change the schema

⇒ ADD Column

```
- ALTER TABLE table-name
  ADD COLUMN column-name datatype constraint;
```

⇒ DROP Column

```
- ALTER TABLE table_name
  DROP COLUMN column-name;
```

⇒ RENAME Table

```
- ALTER TABLE table-name
  RENAME TO new-table-name;
```

⇒ CHANGE column (rename) :

- ALTER TABLE table-name
  CHANGE COLUMN old-name new-name new-datatype new-constraint;

⇒ MODIFY column (modify datatype / constraint)

- ALTER TABLE table-name
  MODIFY col_name new-datatype new-constraint ;

* Truncate Table Query

→ To delete table data

- TRUNCATE TABLE table-name ;

* SQL in Terminal

→ Now, we will use VS code Terminal instead of SQL Workbench software.

In Terminal

```
- HP@DP MINGW64 ~/Desktop/demo/SQL-CLASS
  $ /usr/local/mysql/bin/mysql -u root -p
                       ↑ If this command gives an error then

  $ /c/"Program Files"/MySQL/"MySQLServer 8.0"/bin/mysql -u root -p
  Enter password: mysql@12345
  Welcome to the MySQL monitor. Commands end with
  ; or \g.
  Your SQL Connection id is 42.
  Server version: 8.0.34 MySQL Community Server - GPL

  Copyright (c) 2000, 2023, Oracle and/or its affiliates.

  Oracle is a .....

  Type "help;" or "\h" for help. Type "\c" to clear
  the current input statement.

- mysql>
- mysql > SHOW DATABASES;

    --Database---
    information_schema
    insta
    mysql
    performance_schema
    sakila
    sys
    --world--------
    7 rows in set (0.00 sec)

- mysql > CREATE DATABASE delta-app;

- mysql > USE delta-app;
  Database changed

- mysql > CREATE TABLE temp ( id INT PRIMARY KEY );

- mysql > SHOW TABLES;

    _Tables_in_delta_app__
    ---temp---------
    1 row in set (0.00 sec)
```

* Source .sql file

In demo/SQL-CLASS/ schema.sql

```
- CREATE TABLE user (
     id VARCHAR(50) PRIMARY KEY,
    username VARCHAR(50) UNIQUE,
    email VARCHAR(50) UNIQUE NOT NULL,
     password VARCHAR(50) NOT NULL
  );
```

In Terminal

```
- mysql > source schema.sql;
  Query OK, 0 rows affected (0.01 sec)

- mysql > SHOW TABLES;

    _Tables_in_delta_app_
         temp
    ____user__________
    2 rows in set (0.00 sec)
```

SQL with Node

* Faker Package (@faker-js/faker)

→ To generate fake data

Note: From this page forward in examples, we are assuming that we have two ~~data~~ tables (temp & user) in database (delta_app) as per page no. 118 & 119.

In Terminal

```
- HP@DP MINGW64 ~/Desktop/demo/SQL_CLASS
  $ npm init -y
  :
```

```
- $ npm i @faker-js/faker
  :
```

In SQL_CLASS/ index.js

```
- const { faker } = require ("@faker-js/faker");

let getRandomUser = () => {

   return {
      userId: faker.string.uuid(),
      username: faker.internet.userName(),
      email: faker.internet.email(),
      avatar: faker.image.avatar(),
      password: faker.internet.password(),
      birthdate: faker.date.birthdate(),
      registeredAt: faker.date.past()
   };
};

console.log(getRandomUser());
```

In Terminal

```
- $ node index.js
{
 userId: '3a3b047c-97c1-abf0-8046-50347ac681sc',
 username: 'Rory_Johns',
 email: 'Jason_Braku17@hotmail.com',
 avatar: 'https://avatars.githubusercontent.com/u/18632780',
 password: 'dvHQVbdolaws3Va',
 birthdate: '1947-09-30T21:21:27.378Z',
 registeredAt: 2023-02-23T12:17:23.713Z
}
```

* MySQL Package (mysql2)

→ To connect Node with MySQL

In Terminal

```
- HP@DP MINGW64 ~/Desktop/demo/SQL-CLASS
  $ npm i mysql2
  :
```

In SQL-CLASS/index.js

```
- const mysql = require("mysql2");

const connection = mysql.createConnection( {

      host: "localhost",
      user: "root",
      password: "mysql@12345",
      database: "delta_app"
});

let q = "SHOW TABLES";
```

P.T.O. →

```
try {
  connection.query(q, (err, result) => {

      if (err)  throw err;

      console.log(result);
  });

} catch (err) {

     console.log(err);
}

connection.end();
```

In <u>Terminal</u>

```
- $ node index.js
  [ { Tables_in_delta_app: 'temp' },
    { Tables_in_delta_app: 'user' }  ]

  $ ▮
```

↑ If we had only one table then output will be [{ ... }]

↑ If we didn't write connection.end() then cursor won't appear here but after output (cursor will be there).

⇒ In REST (Web app), we need to have connection with database, so need to end when we working with express.

* INSERT into TABLE (single row)

In index.js

table in database

```
- let q = " INSERT INTO user (id, username, email, password)
          VALUES (?, ?, ?, ?)";

let user = [ "123", "123_newuser", "abc@gmail.com", "abc" ];

try {
   connection. query ( q, user, (err, result) => {

       if (err) throw err;

       console.log (result);
   });
} catch (err) {

     console.log (err);
}
```

In Terminal

```
- $ node index.js
ResultSetHeader {
   fieldCount: 0,
   affectedRows: 1,
   insertId: 0,
   info: " ",
   serverStatus: 2,
   warningStatus: 0,
   changedRows: 0
}
```

* INSERT into Table (multiple rows)

In index.js

```
- let users = [
    ["123b", "123_newuserb", "abc@gmail.comb", "abcb"],
    ["123c", "123_newuserc", "abc@gmail.comc", "abcc"]
  ];

                              table of database
  let q = " INSERT INTO user ( id, username, email, password)
          VALUES ? ";

  connection. query ( q; [users], (err, result) => {
      .....
  });
```

* INSERT into Table (Bulk data using faker)

In index.js

```
- const { faker } = require ( "@faker-js/faker");
  const mysql = require ( "mysql2");

  const connection = mysql.createConnection( {

        host: "localhost",
        user: "root",
        password: "mysql@12345",
        database: "delta_app"
  });

  let getRandomUser = () => {

     return [
        faker.datatype.uuid(),
        faker.~~datat~~ internet.userName(),
        faker.internet.email(),
        faker.internet.password()
     ];
  };

  let data = [];

  for (let i = 1, i <= 100, i++) {

       data.push(getRandomUser());
  }
```

```
                                        ↙ table in database
let q = " INSERT INTO user (id, username, email, password)
         VALUES ?";

try {
    connection.query( q, [data], (err, result) => {

        if (err) throw err;

        console.log( result);
    });
} catch (err) {

    console.log(err);
}

connection.end();
```

In Terminal

```
- $ node index.js
ResultSetHeader {
    fieldCount: 0,
    affectedRows: 100,
    insertId: 0,
    info: 'Records: 100, Duplicates: 0, Warnings: 0',
    serverStatus: 2,
    warningStatus: 0,
    changedRows: 0
}
```

CRUD Operations (SQL + Ex. + Node)

→ SQL database = We have database = "dalta-app"
table = "user"
Total Records = 103

→ packages = express,
uuid,
faker (optional if we want to add bulk fake data),
ejs,
method-override,
mysql2

→ Home Route

GET	/	show no. of users in DB	home.ejs

→ Show Route

GET	/user	show all users data	users.ejs

→ Edit Route

GET	/user/:id/edit	edit form	edit.ejs
PATCH	/user/:id	update in database	

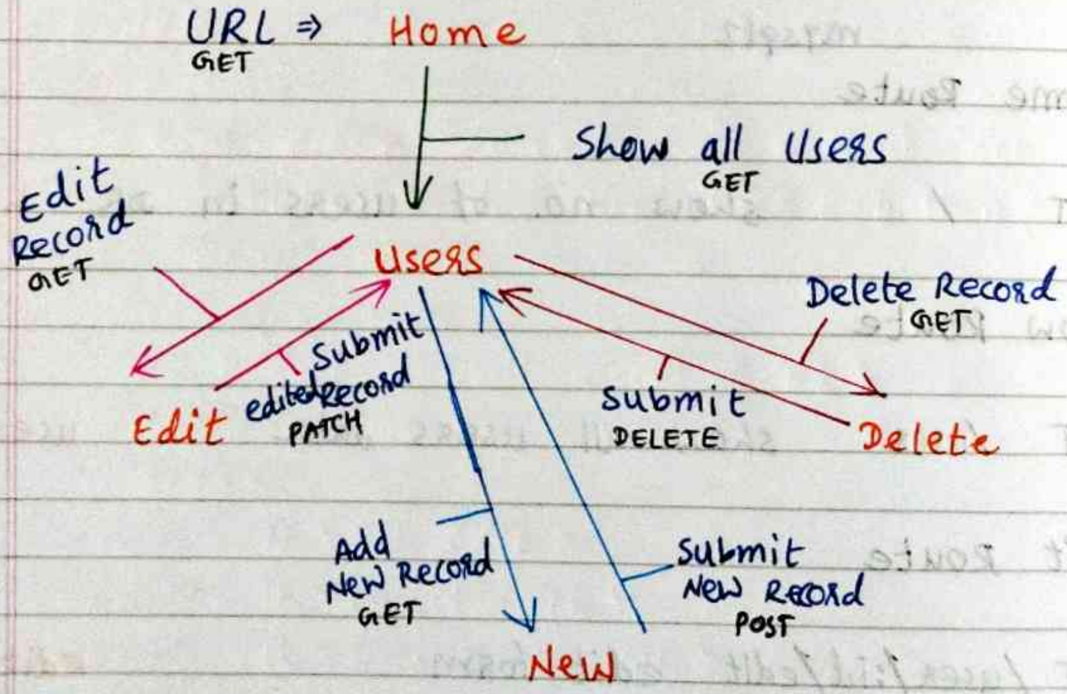
→ Delete Route

GET	/user/:id/delete	delete form	delete.ejs
DELETE	/user/:id	delete record in DB	

→ Add Route

GET /user/new new user addition form new.ejs

POST /user new user data store in DB



* All ".ejs" files of "views" folder

⇒ home.ejs

```
!DOCTYPE html
:

   <title> Home Page </title>
</head>
< body>

   <h2> Total Numbers of Users are: <%= count %>
   </h2>

   <form action="/user" method="get">
    <button> click me to see All users </button>
   </form>

</ body >
</ html >
```

⇒ users.ejs

```
<!DOCTYPE html>
 :
     <title> All Users </title>
     <style>
          table, th, td {
               border: 1px solid black;
          }
     </style>
</head>
<body>

     <form action="/user/new" method="get">
          <button> Add New User </button>
     </form>

     <h2> List of all users </h2>
     <table>
       <tr>
          <th> Id </th>
          <th> Email </th>
          <th> Username </th>
       </tr>

       <% for (const user of users ) { %>

               <tr>
                    <td><%= user.id %> </td>
                    <td><%= user.email %> </td>
                    <td><%= user.username %> </td>
```

```
            < td >
                < form action="/user/<%= user.id %>/edit"
                            method="get" >
                        <button> Edit username</button>
                </ form >
            </td>

            < td >
                < form action="/user/<%= user.id %>/delete"
                            method= "get" >
                        < button > Delete Record </button >
                </ form>
            </td>
        </ tr >
    <%  }  %>
  </ table >

</ body >
</ html >
```

⇒ new.ejs

```
<!DOCTYPE html>
:
    <title> Add New User </title>
</head>
<body>

  <form action="/user" method="post">
      <input  name="email"
              type="email"
              placeholder="enter email" >
      <input  name="username"
              type="text"
              placeholder="enter username">
      <input  name="password"
              type="password"
              placeholder="enter password" >
      <button> submit </button>
  </form>
</body>
</html>
```

⇒ edit.ejs

```
<!DOCTYPE html>
 :
    <title> Edit Page </title>
</head>
< body >

    <h2> You are about to edit this user: <%= user.email %>
    </h2>

    <form action="/user/<%= user.id %>?_method=PATCH"
          method = "post">
        <textarea name "username"
            <%= user.username %>
        </textarea>
        < input name="password"
                type = "password"
                placeholder="enter password" >
        <button> submit </button>

    </form>

</body>
</html>
```

⇒ delete.ejs

```
 !DOCTYPE html
 :
  <title> Delete Page </title>
</head>
<body>

    <h2> You are about to Delete this user:
        <%= user . username  %>  with email:
        <%= user . email     %>
    </h2>

    <form action="/user/<%= user.id %>?_method=DELETE"
          method= "post" >
      <input  name="password"
              type= "password"
              placeholder="enter password" >
      <button> Delete </button>
    </form>

</body>
</html>
```

* "index.js" is in "SQL-CLASS" folder

⇒ index.js

```
const mysql = require("mysql2");
const express = require("express");
const app = ~~require~~ express();
const path = require("path");
const methodOverride = require("method-override");
const { v4: uuidv4 } = require("uuid");

app.set ("view engine", "ejs" );
app.set ("views", path.join(__dirname, "/views"));

app.use (express.urlencoded({extended: true}) );
app.use ( methodOverride ("_method") );

app.listen("8080", () => {
    console.log("~~Server~~ App is listening on port 8080");
});

const connection = mysql.createConnection( {

        host: "localhost",
        user: "root",
        password: "mysql@12345",
        database: "delta_app"
});
```

P.T.O →

```
// Home Route

app.get("/", (req, res) => {
                                                table name
    let q = `SELECT count(*) FROM user`;

    try {
        connection.query(q, (err, result) => {

            if (err) throw err;

            let count = result[0]["count(*)"];
            res.render("home.ejs", {count});
        });
    } catch (err) {
        console.log(err); res.send("Something wrong");
    }
});

// Show Route

app.get("/user", (req, res) => {

    let q = `SELECT * FROM user`;
    try {
        connection.query(q, (err, users) => {
            if (err) throw err;
            res.render("users.ejs", {users});
        });
    } catch(err) {
        console.log(err);
        res.send("something went wrong");
    }
});
```

```
// Edit Route

app.get("/user/:id/edit", (req, res) => {
  let { id } = req.params;
                                                      id in database table
  let q = `SELECT * FROM user WHERE id="${id}" `;
                                                      variable of this block
  try {
     connection.query(q, (err, result) => {
         if (err)  throw  err;
         let  user = result[0];
         res.render("edit.ejs", { user });
     });
  } catch(err) {
     console.log(err);
     res.send("something went wrong");
  }
});
```

P.T.O. →

```
// Edit Route - update database

app.patch("/user/:id", (req, res) => {

    let { id } = req.params;
    let {password: formPass, username: newUsername } = req.body;

    let q = `SELECT * FROM user WHERE id="${ id }" `;

    try {

        connection.query(q, (err, result) => {

            if (err)  throw err;

            let user = result[0];

            if (formPass  !=  user.password) {

                res.send("WRONG PASSWORD");
            } else {

                let q2 = `UPDATE user SET
                          username = "${newUsername}"
                          WHERE id = "${id}" `;

                connection.query(q2, (err, result)=>{
                    if(err) throw err;
                    res.redirect("/user");
                });
            }
        });
    } catch(err) {
            console.log(err);
            res.send("something went wrong");
    }
});
```

// Delete Route

```
app.get("/user/:id/delete", (req, res) => {

    let { id } = req.params;

    let q = `SELECT * FROM user WHERE id="${id}" `;

    try {

        connection.query(q, (err, result) => {

            if (err)   throw err;

            let user = result[0];
            res.render("delete.ejs", {user});
        });
    } catch (err) {

        console.log(err);
        res.send("Something went wrong");
    }
});
```

// Delete Route - Update database

```
app.delete("/user/:id", (req,res) => {

   let {id} = req.params;
   let {password: formPass} = req.body;

   let q = `SELECT * FROM user WHERE id="${id}"`;

   try {
      connection.query(q, (err, result) => {

         if(err) throw err;

         let user = result[0];

         if(formPass != user.password) {
              res.send("WRONG password");
         } else {

            let q2 = `DELETE FROM user WHERE
                      id="${id}"`;
            connection.query(q2, (err, result) => {

                if(err) throw err;

                res.redirect("/user");
            });
         }
   } catch(err) {  console.log(err);
                   res.send("something went wrong");
   }
});
```

```
// Add new Record Route

app.get("/user/new", (req, res) => {

    res.render("new.ejs");
});

// add new record Route - add in database

app.post("/user", (req, res) => {

    let { username, email, password } = req.body;
    let id = uuidv4();

    let q=`INSERT INTO user (id, username, email,
           password) VALUES ("${id}", "${username}",
           "$ {email}", "${password}")`;

    try {
        connection.query( q , (err, result) => {

            if(err) throw err;

            res.redirect("/user");
        });
    } catch(err) {

        console.log(err);
        res.send("Something went wrong");
    }
});
```

* start the server

-
  ```
  HP@DP MINGW64 ~/Desktop/demo/SQL_CLASS
  $ nodemon index.js
  :
  App is listening on port 8080
  ```