Promises in JavaScript

Promises are a **modern approach to handling asynchronous operations** in JavaScript. They help manage operations like fetching data, reading files, or executing timers without blocking the main thread.

1 What is a Promise?

A **Promise** is an object that represents the eventual **completion (success) or failure** of an asynchronous operation.

◇ Basic Syntax

```
let promise = new Promise((resolve, reject) => {
    // Asynchronous operation
    let success = true; // Change to false to simulate failure

    setTimeout(() => {
        if (success) {
            resolve("Operation Successful!"); // When successful
        } else {
            reject("Operation Failed!"); // When failed
        }
    }, 2000); // Simulating delay
});

promise
    .then((result) => console.log(result))  // Handle success
    .catch((error) => console.log(error));  // Handle failure
```

`resolve()` is called if the operation is successful.
`reject()` is called if the operation fails.

2 Promise States

A Promise can have **three states**:

State	Description
Pending	Initial state (operation not completed yet)
Fulfilled	Operation completed successfully (`resolve()`)
Rejected	Operation failed (`reject()`)

3 Chaining Promises

Promises allow **chaining** to execute multiple asynchronous operations **sequentially**.

Example: Fetching Data in Sequence

```
new Promise((resolve) => {
    setTimeout(() => resolve("Step 1 Complete"), 1000);
})
    .then((result) => {
        console.log(result);
        return new Promise((resolve) => setTimeout(() => resolve("Step 2
Complete"), 1000));
    })
    .then((result) => {
        console.log(result);
        return new Promise((resolve) => setTimeout(() => resolve("Step 3
Complete"), 1000));
    })
    .then((result) => console.log(result))
    .catch((error) => console.log("Error:", error));
```

Each `.then()` receives the previous Promise's result and returns a new Promise.
If any step **fails**, the `.catch()` block handles the error.

4 Handling Errors with `.catch()`

Errors in a Promise chain can be caught using `.catch()`.

Example: Handling Errors

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => reject("Something went wrong!"), 2000);
});

promise
    .then((result) => console.log(result))
    .catch((error) => console.error("Error:", error)); // Catches rejection
```

If `reject()` is called, the `.catch()` block executes.

5 `Promise.all()` – Run Multiple Promises in Parallel

When you need to **run multiple asynchronous tasks at the same time** and wait for all of them to complete, use `Promise.all()`.

Example: Running Multiple Promises

```
let p1 = new Promise((resolve) => setTimeout(() => resolve("Promise 1
Done"), 1000));
```

```
let p2 = new Promise((resolve) => setTimeout(() => resolve("Promise 2
Done"), 2000));
let p3 = new Promise((resolve) => setTimeout(() => resolve("Promise 3
Done"), 1500));

Promise.all([p1, p2, p3])
    .then((results) => console.log("All completed:", results))
    .catch((error) => console.log("Error:", error));
```

☑ If all promises **resolve**, `Promise.all()` returns an **array of results**.
❌ If **any promise rejects**, the entire `Promise.all()` fails.

6 `Promise.race()` – Get the First Resolved Promise

If you want to **return the first promise that resolves/rejects**, use `Promise.race()`.

◇ Example: Racing Promises

```
let p1 = new Promise((resolve) => setTimeout(() => resolve("Fastest
Promise"), 1000));
let p2 = new Promise((resolve) => setTimeout(() => resolve("Slower
Promise"), 2000));

Promise.race([p1, p2])
    .then((result) => console.log("Winner:", result))
    .catch((error) => console.log("Error:", error));
```

The **first promise** to resolve/reject determines the result.

7 Async and Await

`async` and `await` are modern ways to handle asynchronous code more cleanly, avoiding `.then()` chains.

Using Async/Await

- **`async`**: Declares a function as asynchronous.
- **`await`**: Waits for a Promise to resolve before continuing execution.

Using **`async/await`** makes promise handling easier and more readable.

◇ Example: Using `async/await` Instead of `.then()`

```
function getData() {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve("Data Fetched!");
```

```
        }, 2000);
    });
}

async function fetchData() {
    console.log("Fetching data...");
    let result = await getData(); // Waits for getData() to complete
    console.log(result);
}

fetchData();
```

Key Points:

1. `await` can only be used inside an `async` function.
2. **`await`** pauses execution until the promise resolves.
3. It makes the code look synchronous but executes asynchronously.

Summary

Feature	**`.then()` / `.catch()`**	**`async/await`**
Readability	Harder in complex cases	Easier & cleaner
Error Handling	Uses `.catch()`	Uses `try...catch`
Use Case	When chaining multiple promises	When writing cleaner async code

Key Points

Promises help handle asynchronous operations.
They have **three states**: `pending`, `fulfilled`, `rejected`.
Use **`.then()`** to handle success and **`.catch()`** to handle errors.
`Promise.all()` waits for all promises, while **`Promise.race()`** returns the first resolved one.
`async/await` makes handling promises cleaner and more readable.

Exercise 1: Simulating API Request

Create a function `fetchUserData(userId)` that **returns a Promise** and resolves after **2 seconds** with user data. If the `userId` is invalid (negative or zero), reject the Promise.

Requirements:

1. If `userId` is **valid**, resolve with `{ id: userId, name: "John Doe" }`.
2. If `userId` is **invalid** (`<= 0`), reject with `"Invalid User ID"`.
3. Call the function with a valid and invalid `userId` and handle both success & error.

Expected Output

```
Fetching user data...
User Data: { id: 1, name: "John Doe" }
```

or

```
Fetching user data...
Error: Invalid User ID
```

```
function fetchUserData(userId) {
    return new Promise((resolve, reject) => {
        console.log("Fetching user data...");
        setTimeout(() => {
            if (userId > 0) {
                resolve({ id: userId, name: "John Doe" });
            } else {
                reject("Invalid User ID");
            }
        }, 2000);
    });
}

// Test with valid and invalid userId
fetchUserData(1)
    .then((data) => console.log("User Data:", data))
    .catch((error) => console.log("Error:", error));
```

Exercise 2: Simulating a Delayed Order System

Write a function `placeOrder(item, delay)` that **returns a Promise** and resolves after `delay` milliseconds.

Requirements:

1. The function should **log** `"Order placed for ITEM_NAME"` immediately.
2. After `delay` milliseconds, the Promise should resolve with `"ITEM_NAME is ready!"`.
3. Use `async/await` to place an order for `"Pizza"` and log the result.

Expected Output

```
Order placed for Pizza
(Pizza is ready! after 3 seconds)
```

```
function placeOrder(item, delay) {
    console.log(`Order placed for ${item}`);
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve(`${item} is ready!`);
        }, delay);
    });
}

// Using async/await
async function orderFood() {
    let result = await placeOrder("Pizza", 3000);
    console.log(result);
}
```

```
orderFood();
```

Exercise 3: Fetching Multiple Data Using `Promise.all()`

You need to **fetch user info and orders simultaneously** using `Promise.all()`.

Requirements:

1. Create `fetchUser()` that resolves in **2 seconds** with `{ id: 1, name: "Alice" }`.
2. Create `fetchOrders()` that resolves in **3 seconds** with `[ "Laptop", "Phone" ]`.
3. Use `Promise.all()` to fetch **both** at the same time and display results.

Expected Output

```
Fetching user and orders...
User Data: { id: 1, name: "Alice" }
Orders: [ 'Laptop', 'Phone' ]
```

```
function fetchUser() {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve({ id: 1, name: "Alice" });
        }, 2000);
    });
}

function fetchOrders() {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve(["Laptop", "Phone"]);
        }, 3000);
    });
}

console.log("Fetching user and orders...");

Promise.all([fetchUser(), fetchOrders()])
    .then(([user, orders]) => {
        console.log("User Data:", user);
        console.log("Orders:", orders);
    })
    .catch((error) => console.log("Error:", error));
```

Exercise 4: Handling API Timeout using `Promise.race()`

Write a function `fetchDataWithTimeout(url, timeout)` that fetches data from an API, but if it takes longer than `timeout` milliseconds, it should **reject with `"Request Timeout!"`**.

Requirements:

1. Create a `fetchData(url)` function that resolves **in 4 seconds**.
2. Create a `timeoutPromise(ms)` function that rejects **after `ms` milliseconds**.
3. Use `Promise.race()` to **resolve with API data or reject with timeout**.
4. Test with a **3-second timeout** (should fail) and **5-second timeout** (should succeed).

Expected Output

```
Fetching data...
Error: Request Timeout!
```

or

```
Fetching data...
Data received: { message: "Hello, World!" }
```

Try it!

```
function fetchData(url) {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve({ message: "Hello, World!" });
        }, 4000);
    });
}

function timeoutPromise(ms) {
    return new Promise((_, reject) => {
        setTimeout(() => {
            reject("Request Timeout!");
        }, ms);
    });
}

async function fetchDataWithTimeout(url, timeout) {
    console.log("Fetching data...");
    try {
        let data = await Promise.race([fetchData(url),
timeoutPromise(timeout)]);
        console.log("Data received:", data);
    } catch (error) {
        console.log("Error:", error);
    }
}

// Test with 3-second and 5-second timeout
fetchDataWithTimeout("https://api.example.com", 3000); // Should timeout
fetchDataWithTimeout("https://api.example.com", 5000); // Should succeed
```

Challenge Exercise: Retry Failed API Request

Imagine an API call fails **sometimes** due to network issues. Write a function `fetchWithRetry(url, attempts)` that retries **up to 3 times** before failing.

Requirements:

1. Create `mockAPI()` that **fails randomly** (rejects 50% of the time).
2. If API fails, **retry** up to `attempts` times.
3. If all retries fail, reject with `"Failed after 3 attempts"`.

Expected Output

```
Fetching data...
Attempt 1 failed, retrying...
Attempt 2 failed, retrying...
Data received: { message: "Success!" }
```

or

```
Fetching data...
Attempt 1 failed, retrying...
Attempt 2 failed, retrying...
Attempt 3 failed, retrying...
Failed after 3 attempts
```

```
function mockAPI() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (Math.random() > 0.5) {
                resolve({ message: "Success!" });
            } else {
                reject("API Error");
            }
        }, 1000);
    });
}

async function fetchWithRetry(url, attempts) {
    console.log("Fetching data...");
    for (let i = 1; i <= attempts; i++) {
        try {
            let data = await mockAPI();
            console.log("Data received:", data);
            return;
        } catch (error) {
            console.log(`Attempt ${i} failed, retrying...`);
        }
    }
    console.log("Failed after", attempts, "attempts");
}

fetchWithRetry("https://api.example.com", 3);
```

Summary

Basic Promises → Fetch user data
Delays with `setTimeout()` → Order system
Parallel Requests → `Promise.all()`
Timeout Handling → `Promise.race()`
Retry Logic → Handling failures

`async/await` for Asynchronous Code Handling in JavaScript

`async/await` provides a cleaner and more readable way to handle asynchronous operations in JavaScript compared to Promises with `.then()` and `.catch()`. It allows writing asynchronous code in a synchronous style, making it easier to debug and maintain.

Basic Syntax of `async/await`

```
async function fetchData() {
    return "Data fetched!";
}

fetchData().then(console.log); // Output: Data fetched!
```

- The `async` keyword makes a function return a **Promise**.
- The `await` keyword pauses execution until the Promise is resolved.

Example 1: Using `await` to Fetch Data

Let's create a function that **fetches user data** after 2 seconds.

```
async function getUserData() {
    console.log("Fetching user data...");
    let user = await new Promise((resolve) => {
        setTimeout(() => {
            resolve({ id: 1, name: "Alice" });
        }, 2000);
    });
    console.log("User Data:", user);
}

getUserData();
```

Output

```
Fetching user data...
(User data appears after 2 seconds)
User Data: { id: 1, name: "Alice" }
```

`await` pauses execution until the user data is available. No need for `.then()`!

Example 2: Handling Errors with `try/catch`

Let's simulate a **failed API request** and properly handle the error.

```
async function fetchWithError() {
    try {
        console.log("Fetching data...");
        let data = await new Promise((_, reject) => {
            setTimeout(() => reject("API Error!"), 2000);
        });
        console.log("Data:", data);
    } catch (error) {
        console.log("Error:", error);
    }
}

fetchWithError();
```

Output

```
Fetching data...
Error: API Error!
```

Instead of `.catch()`, we use a `try/catch` block for **better error handling**.

Example 3: Fetching Multiple APIs Sequentially

If we fetch **user data** first, then **orders**, we must wait for the first to complete.

```
async function fetchUserAndOrders() {
    console.log("Fetching user...");
    let user = await new Promise((resolve) => {
        setTimeout(() => resolve({ id: 1, name: "Alice" }), 2000);
    });

    console.log("User:", user);

    console.log("Fetching orders...");
    let orders = await new Promise((resolve) => {
        setTimeout(() => resolve(["Laptop", "Phone"]), 3000);
    });

    console.log("Orders:", orders);
}

fetchUserAndOrders();
```

Output

```
Fetching user...
(User data appears after 2 seconds)
User: { id: 1, name: "Alice" }
Fetching orders...
(Orders appear after 3 seconds)
Orders: [ 'Laptop', 'Phone' ]
```

Each request **waits** for the previous one to complete before proceeding.

Example 4: Running Multiple Requests in Parallel using `Promise.all()`

Instead of waiting **sequentially**, fetch both **user and orders at the same time**.

```
async function fetchParallel() {
    console.log("Fetching user and orders...");

    let [user, orders] = await Promise.all([
        new Promise((resolve) => setTimeout(() => resolve({ id: 1, name:
"Alice" }), 2000)),
        new Promise((resolve) => setTimeout(() => resolve(["Laptop",
"Phone"]), 3000))
    ]);

    console.log("User:", user);
    console.log("Orders:", orders);
}

fetchParallel();
```

Output

```
Fetching user and orders...
(Both requests start at the same time)
(User appears after 2 seconds, orders after 3 seconds)
User: { id: 1, name: "Alice" }
Orders: [ 'Laptop', 'Phone' ]
```

`Promise.all()` runs multiple async tasks in parallel.

Example 5: Retrying API Calls Using `async/await`

If an API request **fails**, retry **up to 3 times**.

```
async function fetchWithRetry(attempts) {
    for (let i = 1; i <= attempts; i++) {
        try {
            console.log(`Attempt ${i}: Fetching data...`);
            let data = await new Promise((resolve, reject) => {
                setTimeout(() => {
                    Math.random() > 0.5 ? resolve("Success!") : reject("API
Error");
                }, 1000);
            });
            console.log("Data received:", data);
            return;
        } catch (error) {
            console.log(`Attempt ${i} failed. Retrying...`);
```

```
        }
    }
    console.log("All attempts failed.");
}

fetchWithRetry(3);
```

Output

```
Attempt 1: Fetching data...
(50% chance of failure)
Attempt 1 failed. Retrying...
Attempt 2: Fetching data...
(Success message if it works, otherwise retries up to 3 times)
```

`async/await` with loops makes retries easy!

Example 6: API Call with Timeout (`Promise.race()`)

Set a **timeout** for an API request. If it **takes too long**, cancel it.

```
function fetchData() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Fetched Data!"), 4000); // Takes 4
seconds
    });
}

function timeout(ms) {
    return new Promise((_, reject) => {
        setTimeout(() => reject("Request Timeout!"), ms);
    });
}

async function fetchDataWithTimeout(ms) {
    try {
        console.log("Fetching data...");
        let data = await Promise.race([fetchData(), timeout(ms)]);
        console.log("Data:", data);
    } catch (error) {
        console.log("Error:", error);
    }
}

fetchDataWithTimeout(3000); // Will timeout
fetchDataWithTimeout(5000); // Will succeed
```

Output

```
Fetching data...
Error: Request Timeout! (if timeout < API time)
Fetching data...
Data: Fetched Data! (if API is faster)
```

Use `Promise.race()` to cancel long-running requests.

Summary

Basic `async/await` → Handling async functions
Error handling with `try/catch` → Catching failed requests
Sequential vs Parallel requests → Using `Promise.all()`
Retrying failed requests → Using loops
Timeout for API requests → Using `Promise.race()`

Key Points

`async` makes a function return a **Promise**.
`await` **pauses execution** until the Promise resolves.
Use **`try/catch`** for error handling instead of `.catch()`.
Use **`Promise.all()`** for parallel requests.
Use **`Promise.race()`** for timeouts.

HTTP Server

With Node.js, you can create a web server without external libraries.

Basic Web Server

```
const http = require("http");

const server = http.createServer((req, res) => {
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Hello, Node.js Server!");
});

server.listen(3000, () => {
    console.log("Server running at http://localhost:3000");
});
```

Run:

```
node server.js
```

Open **`http://localhost:3000`** in your browser.

Common Status Codes

Status Code	Meaning
`200 OK`	Request was successful
`201 Created`	New resource successfully created
`400 Bad Request`	Client-side error in the request
`401 Unauthorized`	Authentication required
`403 Forbidden`	Client does not have permission
`404 Not Found`	Resource not found
`500 Internal Server Error`	Server-side error occurred

Different Status Codes

```
const http = require("http");

const server = http.createServer((req, res) => {
    if (req.url === "/success") {
        res.writeHead(200, { "Content-Type": "text/plain" });
        res.end("Request successful!");
    } else if (req.url === "/notfound") {
        res.writeHead(404, { "Content-Type": "text/plain" });
        res.end("Page not found!");
    } else {
        res.writeHead(500, { "Content-Type": "text/plain" });
        res.end("Internal Server Error");
    }
});

server.listen(3000, () => {
    console.log("Server running at http://localhost:3000");
});
```

Try this in the browser:

- **`http://localhost:3000/success`** → Returns `200 OK`
- **`http://localhost:3000/notfound`** → Returns `404 Not Found`
- **Any other URL** → Returns `500 Internal Server Error`

Disadvantages of Using HTTP Module Directly

- Manually handling **routes** (e.g., `/`, `/about`) is tedious.
- **Middleware** (e.g., parsing request body) must be implemented manually.
- No built-in support for **JSON responses**.

- **Error handling** needs additional effort.
- No support for **request logging, authentication, and static file serving**.

Express Server

```
const express = require("express");

const app = express();


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

    res.send("Welcome to the home page!");

});


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

    res.send("About page");

});


// Handle all other routes

app.use((req, res) => {

    res.status(404).send("Page not found");

});


app.listen(3000, () => {

    console.log("Express server running at http://localhost:3000");

});
```

Express makes API development easier!

Event-Driven Programming in Node.js

What is Event-Driven Programming?

Event-driven programming is a **programming paradigm** where the **flow of execution is determined by events** such as user actions (clicks, keypresses), sensor outputs, or messages from other programs.

Key Characteristics:

- **Non-blocking:** Code execution **waits for events** rather than running sequentially.
- **Highly scalable:** Ideal for handling **multiple concurrent operations.**
- **Efficient:** Uses an **Event Loop** to handle events asynchronously.

Real-world example: JavaScript in browsers uses events like `click`, `hover`, and `keydown`.

Event-Driven Model in Node.js

Node.js follows an **event-driven architecture**, where it listens for events and executes callback functions when those events occur.

- Uses the **EventEmitter** class from the `events` module.
- Event handlers are functions that execute when an event is triggered.

Example 1: Basic EventEmitter

```
const EventEmitter = require("events"); // Import events module
const myEmitter = new EventEmitter(); // Create an event emitter instance

// Register an event listener
myEmitter.on("greet", () => {
    console.log("Hello! Event triggered.");
});

// Emit (trigger) the event
myEmitter.emit("greet");
```

Output:

```
Hello! Event triggered.
```

Explanation:
1 `on("greet", callback)` → Registers an event listener for `"greet"`.
2 `emit("greet")` → Triggers the event, executing the callback function.

Example 2: Event with Arguments

We can **pass data** when emitting an event.

```
const EventEmitter = require("events");
const myEmitter = new EventEmitter();

myEmitter.on("order", (orderId, amount) => {
    console.log(`Order received! Order ID: ${orderId}, Amount:
$${amount}`);
});

myEmitter.emit("order", 12345, 250);
```

Output:

```
Order received! Order ID: 12345, Amount: $250
```

Use Case: Event-driven order processing in e-commerce apps.

Example 3: Handling Events Once (`once` Method)

```
const EventEmitter = require("events");
const myEmitter = new EventEmitter();

myEmitter.once("connect", () => {
    console.log("Connected to the server!");
});

myEmitter.emit("connect");
myEmitter.emit("connect"); // Will not trigger again
```

Output:

```
Connected to the server!
```

Use Case: Database connections (connect only once).

Example 4: Removing an Event Listener

```
const EventEmitter = require("events");
const myEmitter = new EventEmitter();

const logMessage = () => console.log("Logging event triggered!");

myEmitter.on("log", logMessage);
myEmitter.emit("log"); // Event runs

myEmitter.removeListener("log", logMessage);
myEmitter.emit("log"); // No output since event listener was removed
```

Output:

```
Logging event triggered!
```

Use Case: Unsubscribing from notifications.

How Node.js Uses Event-Driven Programming

1 **HTTP Servers** – Handle requests using events
2 **File System (fs module)** – Listen for file read/write events
3 **Database Queries** – Handle data retrieval asynchronously
4**WebSockets (real-time communication)** – Chat apps, multiplayer games

Example 5: Event-Driven HTTP Server

```
const http = require("http");
const EventEmitter = require("events");

const server = new EventEmitter();

// Listen for "request" event
server.on("request", (req, res) => {
    res.writeHead(200, { "Content-Type": "text/plain" });
    res.end("Hello, Event-Driven Server!");
});

// Create and start an HTTP server
http.createServer((req, res) => {
    server.emit("request", req, res); // Trigger "request" event
}).listen(3000, () => console.log("Server running on port 3000"));
```

Output:
When you visit `http://localhost:3000`, the browser displays:

```
Hello, Event-Driven Server!
```

Use Case: Handling web requests dynamically using event listeners.

Conclusion

Event-Driven Architecture makes Node.js highly efficient.
EventEmitter allows defining and handling events in an organized way.
Ideal for real-time applications like **chats, notifications, and streaming**.

Assignment: Online Order Processing System

Scenario:

You are building a **Food Delivery System** using **Node.js and Express** that processes orders asynchronously. The system should:

1. **Accept a new order** (via API).
2. **Process the order asynchronously** using **callbacks, Promises, and async/await**.
3. **Emit an event** when the order is ready.
4. **Retrieve the order status** using an API.

Assignment Requirements:

You need to implement the following functionalities using **Node.js, Express.js, Promises, Async/Await, and EventEmitter**:

1 Accept New Order (POST `/order`)

- Take order details (order ID, food item, customer name).
- Simulate order preparation **asynchronously** (use `setTimeout`).
- Use **callback** to notify when the order is received.

2 Process Order Asynchronously

- Simulate order preparation using **Promise** and **async/await**.
- The order should take 5 seconds to prepare.
- When the order is ready, emit an event using **EventEmitter**.

3 Retrieve Order Status (GET `/order/:id`)

- Check if the order is still being processed or is ready.

Connecting Node.js with SQL Server

To connect Node.js with **SQL Server**, we use the **mssql** package.

1 Install `mssql` package

```
npm install mssql
```

2 Configure Database Connection

Create a `dbConfig.js` file:

```
const sql = require("mssql");

const config = {
    user: "your_username",
    password: "your_password",
    server: "localhost", // or your server name
    database: "your_database",
    options: {
        encrypt: false, // Disable encryption for local servers
        trustServerCertificate: true,
    }
};

const poolPromise = new sql.ConnectionPool(config)
    .connect()
    .then(pool => {
        console.log("Connected to SQL Server");
        return pool;
    })
    .catch(err => console.error("Database Connection Failed! ", err));

module.exports = { sql, poolPromise };
```

3 Fetch Data from SQL Server

Create a `server.js` file:

```
const express = require("express");
const { sql, poolPromise } = require("./dbConfig");

const app = express();
app.use(express.json());

app.get("/users", async (req, res) => {
    try {
        const pool = await poolPromise;
        const result = await pool.request().query("SELECT * FROM Users");
        res.json(result.recordset);
    } catch (err) {
        res.status(500).send(err.message);
    }
});

app.listen(3000, () => {
    console.log("Server running on http://localhost:3000");
});
```

4 Insert Data into SQL Server

```
app.post("/users", async (req, res) => {
    try {
        const { name, email } = req.body;
        const pool = await poolPromise;
        await pool
            .request()
            .input("name", sql.NVarChar, name)
```

```
            .input("email", sql.NVarChar, email)
            .query("INSERT INTO Users (name, email) VALUES (@name,
@email)");

        res.send("User added successfully!");
    } catch (err) {
        res.status(500).send(err.message);
    }
});
```

5 Update Data

```
app.put("/users/:id", async (req, res) => {
  try {
    const { id } = req.params; // Get user ID from URL
    const { name, email } = req.body; // Get new data from request body
    const pool = await poolPromise;

    const result = await pool
      .request()
      .input("id", sql.Int, id)
      .input("name", sql.NVarChar, name)
      .input("email", sql.NVarChar, email)
      .query("UPDATE Users SET name = @name, email = @email WHERE id = @id");

    if (result.rowsAffected[0] > 0) {
      res.send(`User with ID ${id} updated successfully!`);
    } else {
      res.status(404).send("User not found!");
    }
  } catch (err) {
    res.status(500).send(err.message);
  }
});
```

5. Delete Data

```
app.delete("/users/:id", async (req, res) => {
    try {
        const { id } = req.params; // Get user ID from URL
        const pool = await poolPromise;

        const result = await pool
            .request()
            .input("id", sql.Int, id)
            .query("DELETE FROM Users WHERE id = @id");

        if (result.rowsAffected[0] > 0) {
            res.send(`User with ID ${id} deleted successfully!`);
        } else {
            res.status(404).send("User not found!");
        }
    } catch (err) {
        res.status(500).send(err.message);
    }
});
```