Let's make it **super simple and beginner-friendly**, so you can practice **Daily and Gradually Build Confidence**.

Here's a clear breakdown for **Day 1 and Day 2, Day3, Day4, Day 5, Day 6, Day 7 , Day 8**.

If you're just starting with **Express.js**, the key is **daily, small, consistent practice**—think of it like jogging for your brain. Here's a structured daily warm-up routine you can follow to reinforce your memory and skills:-

Day 1 – Hello World Server

Goal: Get a basic Express server running and respond to a simple route.

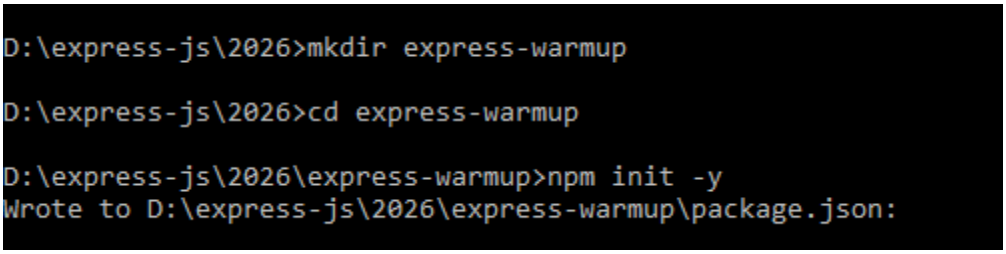
Steps:

1. **Create a project folder**

```
mkdir express-warmup
cd express-warmup
```

2. **Initialize npm**

```
npm init -y
```



3. **Install Express**

```
npm install express
```

```
D:\express-js\2026\express-warmup>npm install express

added 65 packages, and audited 66 packages in 7s

22 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

D:\express-js\2026\express-warmup>node app.js
Server running on http://localhost:3000
```

4. **Create a file** called `app.js`.

5. **Write the code**:

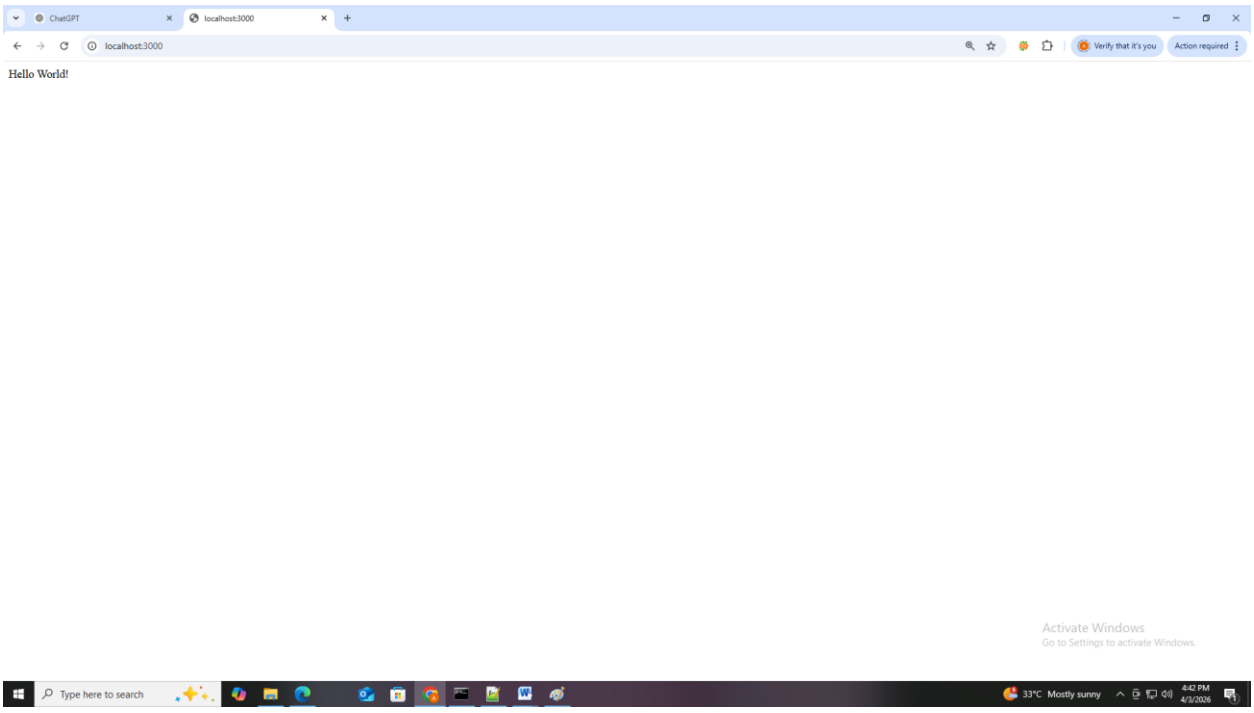
```
const express = require('express');
const app = express();

// Home route
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// Start server

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

6. **Run your server**

```
node app.js
```

```
D:\express-js\2026\express-warmup>node app.js
Server running on http://localhost:3000
```

7. **Test it:** Open your browser and go to `http://localhost:3000`. You should see **"Hello World!"**.



Day 2 – Add Another Route

Goal: Learn how to create multiple routes and send different responses.

Steps:

1. **Open `app.js` from Day 1**.
2. **Add a new route** for `/about`:

```
const express = require('express');
const app = express();

// Home route
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// About route
app.get('/about', (req, res) => {
  res.send('This is the About page.');
});

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

3. **Run the server**

```
node app.js
```

4. **Test the routes:**

   - `http://localhost:3000/` → "Hello World!"
   - `http://localhost:3000/about` → "This is the About page."

Tip: Try **adding one more route** like `/contact` as a tiny challenge.

Day 3 – JSON Response

Goal: Learn how to return JSON data from a route.

Steps:

1. **Open your `app.js`** file or create a new one if you want a fresh start.
2. **Add a new route `/api/data`** that sends JSON:

```
const express = require('express');
const app = express();

// Home route
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// About route
app.get('/about', (req, res) => {
  res.send('This is the About page.');
});

// JSON route
app.get('/api/data', (req, res) => {
```

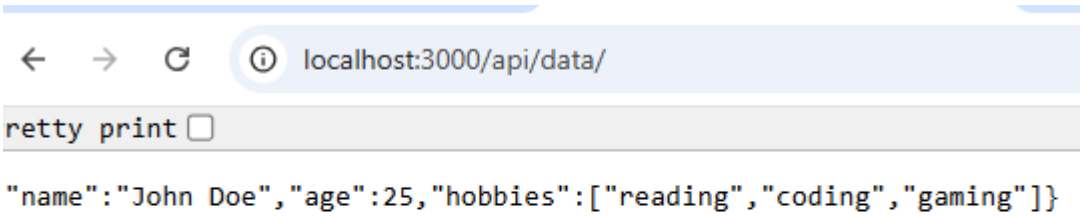
```
  const data = {
    name: "John Doe",
    age: 25,
    hobbies: ["reading", "coding", "gaming"]
  };
  res.json(data);
});

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

3. **Run the server:**

```
node app.js
```

4. **Test it:**
   Open `http://localhost:3000/api/data` in your browser or Postman—you should see a JSON object.

Warm-up challenge: Change the JSON object daily. Add another field like `city` or `profession`.



Day 4 – Query Parameters

Goal: Learn how to read data from the URL using query parameters.

Steps:

1. **Add a new route `/greet`** that takes a `name` query parameter in app.js file :

```
// Greet route with query parameter
app.get('/greet', (req, res) => {
```

```
  const name = req.query.name || "Guest"; // Default if no name provided
  res.send(`Hello, ${name}! Welcome to Express.`);
});
```

Then your app.js file will be :-

```
const express = require('express');
const app = express();

// Home route
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// About route
app.get('/about', (req, res) => {
  res.send('This is the About page.');
});

// JSON route
app.get('/api/data', (req, res) => {
  const data = {
    name: "John Doe",
    age: 25,
    hobbies: ["reading", "coding", "gaming"]
  };
  res.json(data);
});

// Greet route with query parameter
app.get('/greet', (req, res) => {
  const name = req.query.name || "Guest"; // Default if no name provided
  res.send(`Hello, ${name}! Welcome to Express.`);
});

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

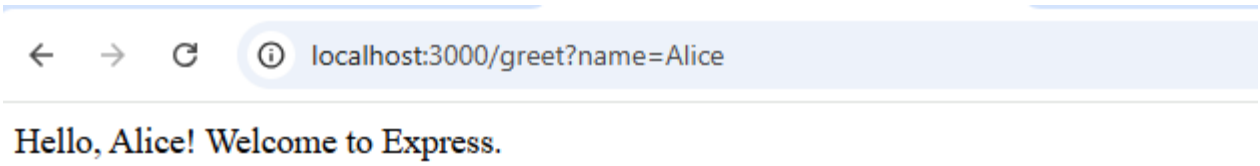
2. **Run the server** again:

```
node app.js
```

3. **Test it:**

- Open `http://localhost:3000/greet?name=Alice` → "Hello, Alice! Welcome to Express."
- Open `http://localhost:3000/greet` → "Hello, Guest! Welcome to Express."

Warm-up challenge: Try adding multiple query parameters, like `age` or `city`, and return a personalized message.



Day 5 – Route Parameters

Goal: Learn how to use dynamic parts of a URL (route parameters) to respond differently based on input.

Steps:

1. **Open your `app.js` file.**
2. **Add a new route `/users/:id`:**

```
// Route with parameter
app.get('/users/:id', (req, res) => {
  const userId = req.params.id; // Extract parameter from URL
  res.send(`You requested info for user with ID: ${userId}`);
});
```

Then your app.js file code will be :-

```
const express = require('express');
const app = express();

// Home route
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// About route
```

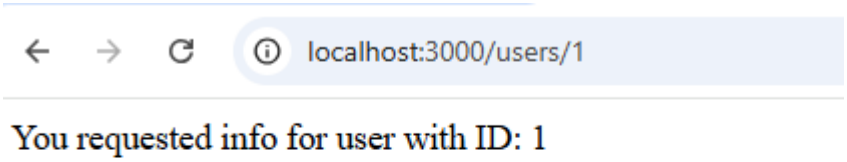
```
app.get('/about', (req, res) => {
  res.send('This is the About page.');
});

// JSON route
app.get('/api/data', (req, res) => {
  const data = {
    name: "John Doe",
    age: 25,
    hobbies: ["reading", "coding", "gaming"]
  };
  res.json(data);
});

// Greet route with query parameter
app.get('/greet', (req, res) => {
  const name = req.query.name || "Guest"; // Default if no name provided
  res.send(`Hello, ${name}! Welcome to Express.`);
});

// Route with parameter
app.get('/users/:id', (req, res) => {
  const userId = req.params.id; // Extract parameter from URL
  res.send(`You requested info for user with ID: ${userId}`);
});

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

3. **Run the server:**

```
node app.js
```

4. **Test it:**

- `http://localhost:3000/users/1` → "You requested info for user with ID: 1"
- `http://localhost:3000/users/42` → "You requested info for user with ID: 42"



Day 6 – Middleware Practice

Goal: Learn how to use middleware to process requests before reaching the route handler.

Steps:

1. **Add a simple logging middleware**:

```
// Logging middleware
app.use((req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} request to
${req.url}`);
  next(); // Pass control to the next handler
});
```

2. **Test your middleware:**

- Every time you make a request to any route (like `/` or `/about`), it should log the request in the console.

3. **Add another middleware for a specific route**:

```
// Route-specific middleware
function checkSecret(req, res, next) {
  const secret = req.query.secret;
```

```
  if(secret === '123') {
    next(); // User allowed
  } else {
    res.send('Access denied. Provide the correct secret.');
  }
}

app.get('/secret', checkSecret, (req, res) => {
  res.send('Welcome to the secret page!');
});
```

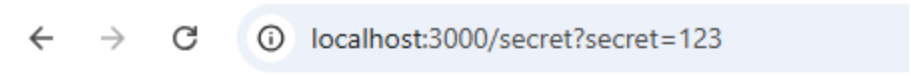
Your app.js file code will be like this :-

```
const express = require('express');
const app = express();

// Home route
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// About route
app.get('/about', (req, res) => {
  res.send('This is the About page.');
});

// JSON route
app.get('/api/data', (req, res) => {
  const data = {
    name: "John Doe",
    age: 25,
    hobbies: ["reading", "coding", "gaming"]
  };
  res.json(data);
});

// Greet route with query parameter
app.get('/greet', (req, res) => {
  const name = req.query.name || "Guest"; // Default if no name provided
  res.send(`Hello, ${name}! Welcome to Express.`);
});

// Route with parameter
app.get('/users/:id', (req, res) => {
  const userId = req.params.id; // Extract parameter from URL
  res.send(`You requested info for user with ID: ${userId}`);
});

// Route-specific middleware
function checkSecret(req, res, next) {
  const secret = req.query.secret;
  if(secret === '123') {
    next(); // User allowed
  } else {
    res.send('Access denied. Provide the correct secret.');
  }
```

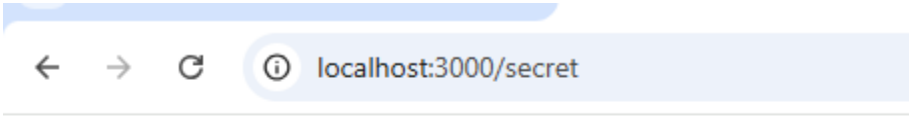
```
}

// Secret route using middleware
app.get('/secret', checkSecret, (req, res) => {
  res.send('Welcome to the secret page!');
});
// Start server
app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

4. **Run and test:**

- `http://localhost:3000/secret?secret=123` → Allowed
- `http://localhost:3000/secret` → Denied



Welcome to the secret page!



Access denied. Provide the correct secret.

Let's break this **step by step**, because understanding middleware is crucial in Express.

---

1. The Middleware Function

```
function checkSecret(req, res, next) {
  const secret = req.query.secret;
  if(secret === '123') {
    next(); // User allowed
  } else {
    res.send('Access denied. Provide the correct secret.');
  }
}
```

- **`req`**: the request object. Contains info about the incoming request (URL, query params, headers, etc.).
- **`res`**: the response object. Used to send a response back to the client.
- **`next`**: a function that tells Express to **move on to the next middleware or route handler**.

Inside the function:

1. `const secret = req.query.secret;`
   - This reads a **query parameter** named `secret` from the URL.
   - Example: `/secret?secret=123` → `req.query.secret` will be `"123"`.
2. `if(secret === '123') { next(); }`
   - If the secret is exactly `"123"`, call `next()`.
   - This allows the request to continue to the **actual route handler** (`(req, res) => {...}`).
3. `else { res.send('Access denied...'); }`
   - If the secret is wrong or missing, send an **immediate response**.
   - The request stops here and does **not** reach the route handler.

---

2. Using the Middleware on a Route

```
app.get('/secret', checkSecret, (req, res) => {
  res.send('Welcome to the secret page!');
});
```

- **`/secret`** is the route URL.
- **`checkSecret`** is the middleware function applied **before the route handler**.
- **`(req, res) => {...}`** is the actual route handler that runs **only if `next()` was called**.

Flow:

1. Client requests `/secret?secret=123`.
2. Express calls `checkSecret(req, res, next)`.
   - Secret matches `"123"` → `next()` called.
3. Express then runs the route handler → sends `"Welcome to the secret page!"`.

If the secret is missing or wrong:

1. Client requests `/secret` or `/secret?secret=abc`.
2. `checkSecret` sees the secret is not `"123"`.
3. Sends `"Access denied..."` → route handler **never runs**.

Key Points:

- Middleware can **control access**, modify requests/responses, or log info.
- `next()` is crucial—without it, the next handler never runs.
- Route-specific middleware only affects the routes you attach it to.

Day 7 – Add POST Request Handling

Goal: Learn how to accept **data from the client** using POST requests and `express.json()`.

Steps:

1. **Update your file** (or create `app.js`)

```
const express = require('express');
const app = express();

// Middleware to parse JSON body
app.use(express.json());

// POST route to create a new user
app.post('/users', (req, res) => {
  const user = req.body; // { name: "Alice", age: 25 }
  res.json({
    message: "User created successfully",
    user
  });
});

app.listen(3000, () => console.log('Day 8 app running on
http://localhost:3000'));
```

and run

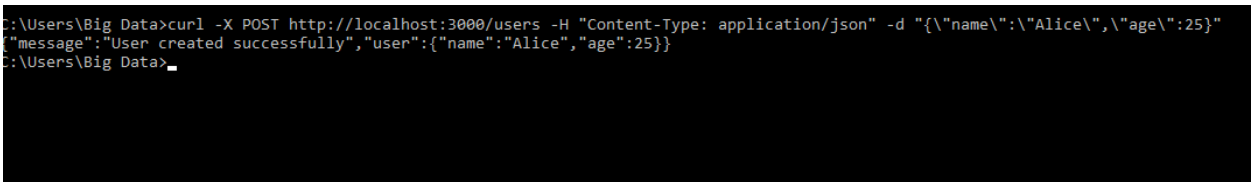
node app.js file

and open your cmd in windows :-

Here's the correct command for **Windows Command Prompt**:

```
curl -X POST http://localhost:3000/users -H "Content-Type: application/json" -d "{\"name\":\"Alice\",\"age\":25}"
```

in windows pc in cmd as shown below.:-



Full working Day 8 “users API” (GET + POST)

```
const express = require('express');
const app = express();

app.use(express.json()); // Parse JSON body

// In-memory users
let users = [];

// GET all users
app.get('/users', (req, res) => {
  res.json(users);
});

// POST create user
app.post('/users', (req, res) => {
  const user = req.body;
  user.id = users.length ? users[users.length - 1].id + 1 : 1;
  users.push(user);
  res.json({ message: "User created successfully", user });
});

app.listen(3000, () => console.log("Day 8 app running on
http://localhost:3000"));
```

---

4. Testing (Windows CMD)

Create user:

```
curl -X POST http://localhost:3000/users -H "Content-Type: application/json" -d "{\"name\":\"Alice\",\"age\":25}"
```

Get all users:

```
curl http://localhost:3000/users
```

 You should now see:

```
[
  { "id": 1, "name": "Alice", "age": 25 }
]
```

Common mistakes that break GET /users:

1. Forgetting `app.use(express.json())` → only affects POST/PUT, not GET.
2. Overwriting `users` accidentally in another route.
3. Sending `res.send` in a middleware **before GET route**.

Perfect! Let's create a **full working CRUD app** that includes **GET all users, GET single user, POST, PUT, DELETE**, and is **Windows CMD friendly** so all your curl commands will work. This will ensure **Day 8–9 practice works perfectly**.

Full CRUD App – Windows CMD Ready(app.js)

```
// app.js
const express = require('express');
const app = express();

// Middleware to parse JSON body
app.use(express.json());

// In-memory users array
let users = [
  { id: 1, name: "Alice", age: 25 },
  { id: 2, name: "Bob", age: 30 }
];

// ------------------
// GET all users
app.get('/users', (req, res) => {
  res.json(users);
});

// GET single user by ID
app.get('/users/:id', (req, res) => {
  const id = Number(req.params.id);
  const user = users.find(u => u.id === id);
  if(user) {
    res.json(user);
  } else {
    res.status(404).json({ error: "User not found" });
  }
```

```
});



// ------------------
// CREATE user (POST)
app.post('/users', (req, res) => {
  const user = req.body;
  user.id = users.length ? users[users.length - 1].id + 1 : 1;
  users.push(user);
  res.json({ message: "User created successfully", user });
});



// ------------------
// UPDATE user (PUT)
app.put('/users/:id', (req, res) => {
  const id = Number(req.params.id);
  const index = users.findIndex(u => u.id === id);
  if(index !== -1) {
    users[index] = { id, ...req.body };
    res.json({ message: "User updated successfully", user: users[index] });
  } else {
    res.status(404).json({ error: "User not found" });
  }
});

// ------------------
// DELETE user (DELETE)
app.delete('/users/:id', (req, res) => {
  const id = Number(req.params.id);
  const index = users.findIndex(u => u.id === id);
  if(index !== -1) {
    const deleted = users.splice(index, 1);
    res.json({ message: "User deleted successfully", user: deleted[0] });
  } else {
    res.status(404).json({ error: "User not found" });
  }
});

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

Run it:-

Node app.js

Windows CMD curl Commands

1️⃣ Get all users

```
curl http://localhost:3000/users
```

2️⃣ Get single user

```
curl http://localhost:3000/users/1
```

3️⃣ Create user

```
curl -X POST http://localhost:3000/users -H "Content-Type: application/json" -d "{\"name\":\"Charlie\",\"age\":28}"
```

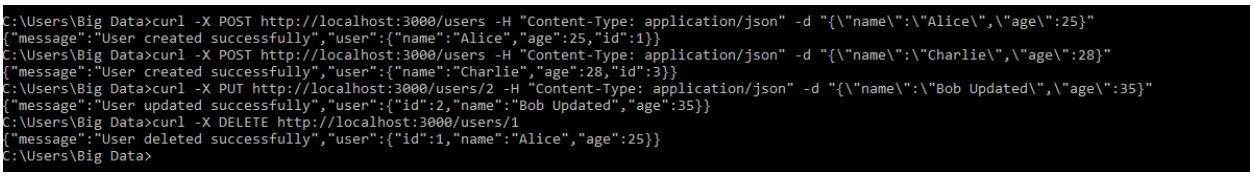
4️⃣ Update user

```
curl -X PUT http://localhost:3000/users/2 -H "Content-Type: application/json" -d "{\"name\":\"Bob Updated\",\"age\":35}"
```

5️⃣ Delete user

```
curl -X DELETE http://localhost:3000/users/1
```

as shown below windows Cmd :-



Let’s go **line by line in simple words.**

---

1. Import and Setup Express

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

- Imports the **Express library**.
- `require('express')` loads it from `node_modules`.

```
const app = express();
```

- Creates your **Express app (server)**.
- `app` is what you use to define routes and middleware.

---

2. Middleware

```
app.use(express.json());
```

- This is **middleware**.
- It tells Express:
  □ "If request has JSON data, convert it into JavaScript object"
- Without this, `req.body` will be **undefined** in POST/PUT.

---

3. In-Memory Database

```
let users = [
  { id: 1, name: "Alice", age: 25 },
  { id: 2, name: "Bob", age: 30 }
];
```

- This is a **fake database** stored in memory.
- `users` is just a JavaScript array.
- Data will reset when server restarts.

---

4. GET All Users

```
app.get('/users', (req, res) => {
  res.json(users);
});
```

- `app.get()` → handles **GET requests**.
- `/users` → URL endpoint.

Inside function:

- `req` → request from client
- `res` → response to client

```
res.json(users);
```

- Sends users array as **JSON response**.

---

5. GET Single User

```
app.get('/users/:id', (req, res) => {
```

- `:id` is a **route parameter** (dynamic value).

```
const id = Number(req.params.id);
```

- Gets `id` from URL.
- Converts string → number.

```
const user = users.find(u => u.id === id);
```

- Searches array for matching user.
- `.find()` returns first match.

```
if(user) {
  res.json(user);
}
```

- If found → send user data.

```
else {
  res.status(404).json({ error: "User not found" });
}
```

- If not found → send **404 error**.

---

6. CREATE User (POST)

```
app.post('/users', (req, res) => {
```

- Handles **POST request** (create data).

```
const user = req.body;
```

- Gets data from request body (JSON).

```
user.id = users.length ? users[users.length - 1].id + 1 : 1;
```

This line is important □

- If users exist:
  □ Take last user's ID and add 1
- If empty:
  □ Start with ID = 1

```
users.push(user);
```

- Adds new user to array.

```
res.json({ message: "User created successfully", user });
```

- Sends success message + created user.

---

7. UPDATE User (PUT)

```
app.put('/users/:id', (req, res) => {
```

- Handles **update request**.

```
const id = Number(req.params.id);
```

- Get ID from URL.

```
const index = users.findIndex(u => u.id === id);
```

- Finds position (index) in array.

```
if(index !== -1) {
```

- If user exists (`-1` means not found)

```
users[index] = { id, ...req.body };
```

- Updates user.
- `...req.body` = spread operator (copies fields)
- Example:

  ```
  { id: 2, name: "New", age: 40 }
  ```

```
res.json({ message: "User updated successfully", user: users[index] });
```

- Sends updated user.

```
else {
  res.status(404).json({ error: "User not found" });
}
```

- If user doesn't exist → error.

---

8. DELETE User

```
app.delete('/users/:id', (req, res) => {
```

- Handles delete request.

```
const id = Number(req.params.id);
```

- Get ID.

```
const index = users.findIndex(u => u.id === id);
```

- Find user position.

```
if(index !== -1) {
```

- If found:

```
const deleted = users.splice(index, 1);
```

- Removes user from array.
- Returns deleted user in array.

```
res.json({ message: "User deleted successfully", user: deleted[0] });
```

- Sends deleted user.

```
else {
  res.status(404).json({ error: "User not found" });
}
```

- If not found → error.

---

9. Start Server

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

- Starts server on port **3000**.
- Callback runs when server starts.

□ Simple Summary

- `app.get()` → Read data
- `app.post()` → Create data
- `app.put()` → Update data
- `app.delete()` → Delete data
- `req` → incoming request
- `res` → outgoing response
- `req.params` → URL values

- `req.body` → JSON data
- `res.json()` → send JSON