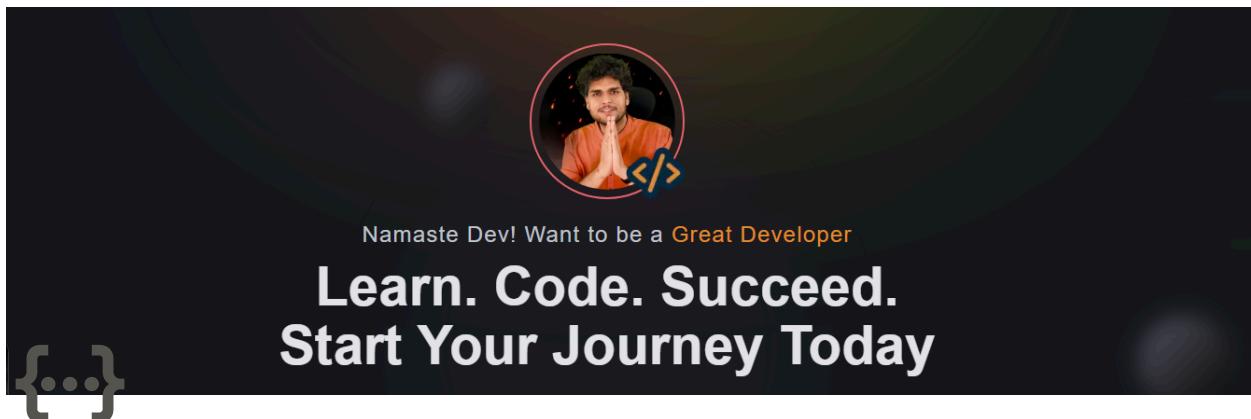


Route Handler : a function or block of code that executes when a specific URL (route) in a web application is accessed.



# Episode-04 | Routing and Request Handlers

## Pushing Code to GitHub

First, **initialize Git** in your project folder:

```
git init
```

**Important:** `.gitignore`

Add `node_modules` to your `.gitignore` file.

We **never push** `node_modules` to GitHub → it can be installed anytime using:

```
npm install
```

## Creating and Pushing the Repo

1. Create a GitHub repository ([I named mine devtinder](#)).
2. You can now **clone it** or **push an existing repo** using commands:

```
git add .
git commit -m "Initial commit"
git branch -M main
git remote add origin <your-repo-url>
git push -u origin main
```

Now, **all your code will be on GitHub.**



## Homework

Pause here and **push your code to GitHub.**

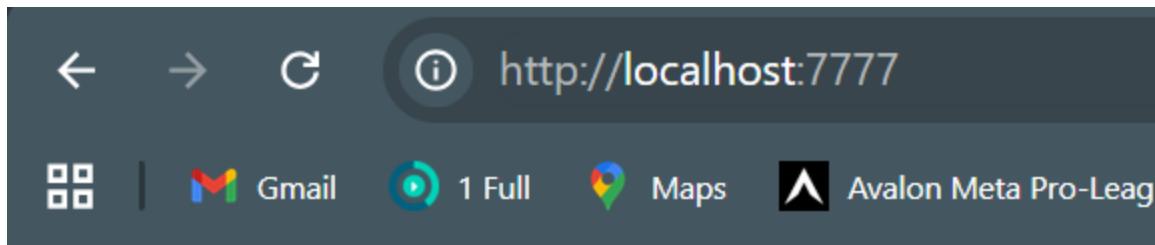
Next, I'll show you something a bit tricky { I want your 100% focus for this part }

## Understanding Routes in Express

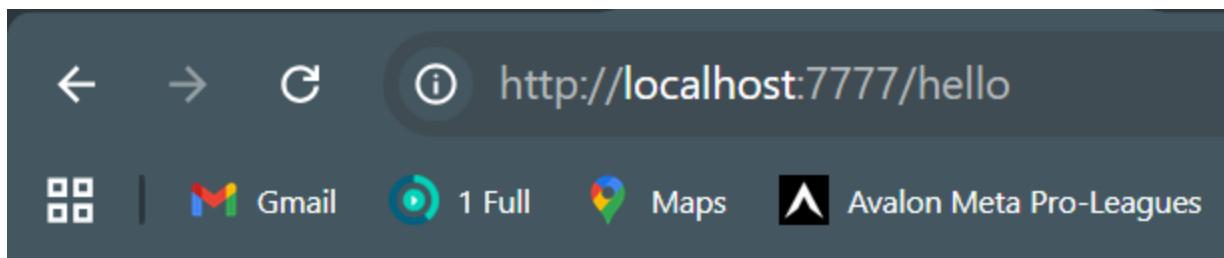
```
const express = require("express");
const app = express();
app.use("/", (req, res) => {
```

```
res.send("Namaste Akshay!!");  
});  
  
app.use("/hello", (req, res) => {  
  res.send("hello hello hello!!");  
});  
  
app.use("/test", (req, res) => {  
  res.send("Hello from the server!!");  
});  
  
app.listen(7777, () => {  
  console.log("Server is successfully listening on port 7777");  
});
```

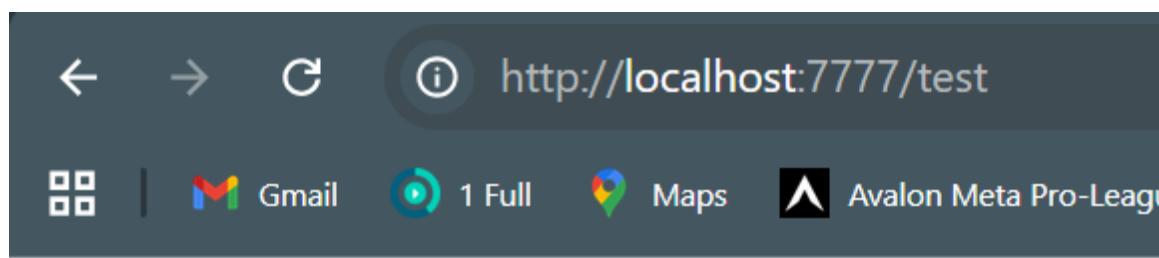
Let's check the `/` route.



now , Let's check the `/hello` route.

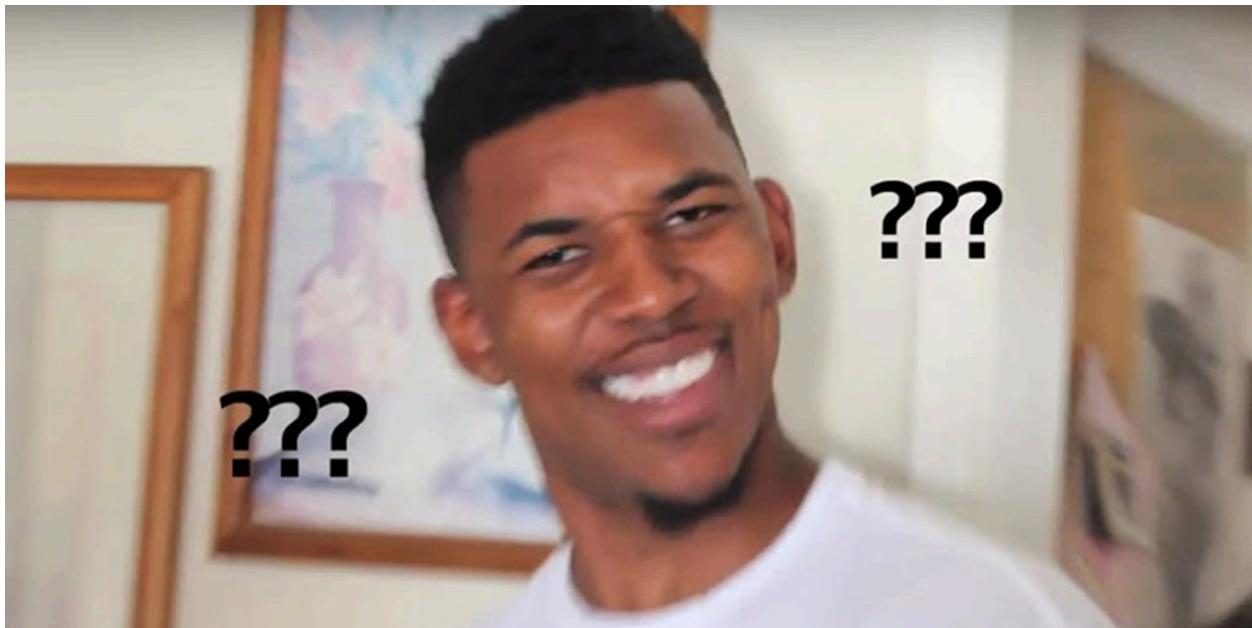


Hmm, something seems wrong. Let's check the `/test` route.



It's still showing the same issue 😞

why isn't it displaying "**Hello from the server**" on the `/test` route?



What happens if I go to a random URL? It doesn't work → why?

Now, let's try **removing** the `/` route from our code and run it again.

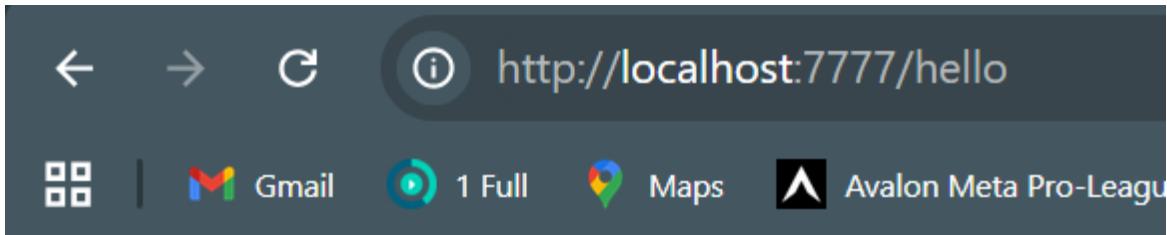
```
const express = require("express");

const app = express();

app.use("/hello", (req, res) => {
  res.send("hello hello hello!!");
});

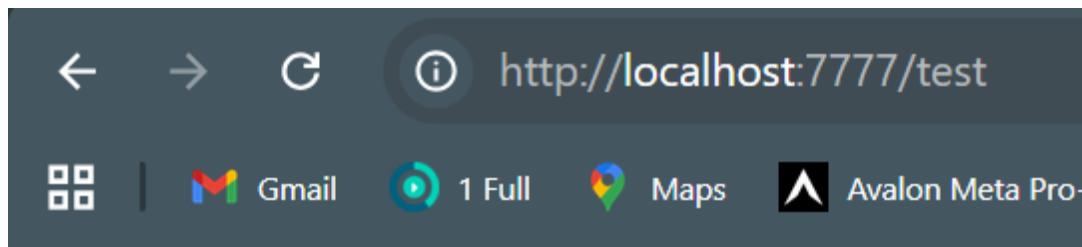
app.use("/test", (req, res) => {
  res.send("Hello from the server!!");
}

app.listen(7777, () => {
  console.log("Server is successfully listening on port 7777");
});
```



Now, visiting `/hello` will display “hello hello hello”.

And if I go to `/test`, it will show “Hello from the server”.



But if we introduce this code again:

```
app.use("/", (req, res) => {
  res.send("Namaste Akshay!!");
});
```

- This route **overrides everything else**.
- Any route that **matches / or starts with /** will be handled by this route — it acts like a **wildcard**.

- So it doesn't just cover `/hello` ; it also handles anything like `/hello/q21/312` .

### ⚠ Common confusion:

- If you write `/test` and then try `/test123`, it **will not Work**.
- `/test123` is a **different string**, so it won't be handled by the `/test` route.

Now, let me show you **one more interesting thing**, and I'll ask you a question:

### Does the sequence of your code matter?



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

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

```
app.use("/hello", (req, res) => {
  res.send("hello hello hello!!");
});
```

```
app.use("/test", (req, res) => {
```

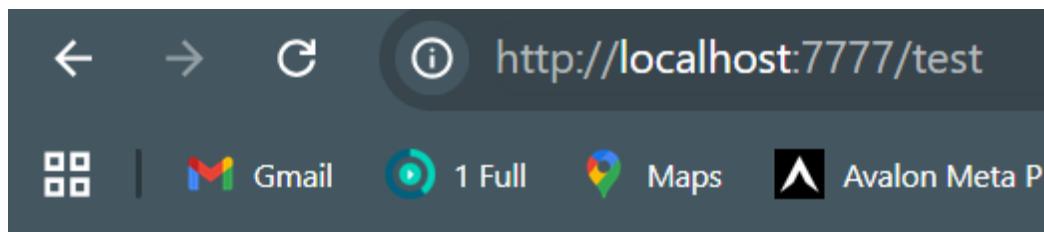
```

res.send("Hello from the server!!");
});
app.use("/", (req, res) => {
  res.send("Namaste Akshay!!!");
});
app.listen(7777, () => {
  console.log("Server is successfully listening on port 7777");
});

```

Now, if I go to `/test`, it gives me **"Hello from the server"**.

Just by **changing the order of the code**, a lot of things can change.



## Example:

```

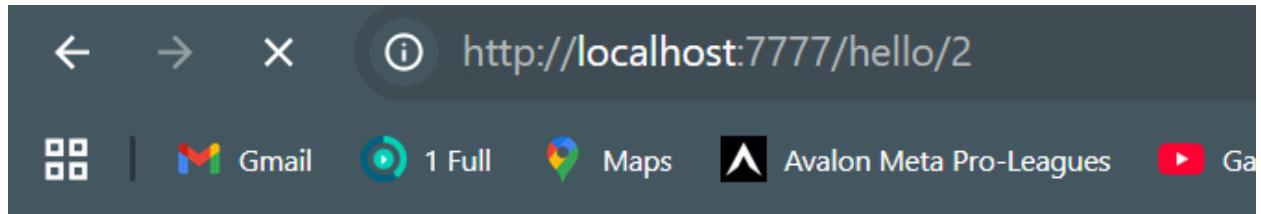
app.use("/hello", (req, res) => {
  res.send("hello hello hello!!!");
});

app.use("/hello/2", (req, res) => {
  res.send("Abra ka Dabra!!!");
});

```

Now, if I go to `/hello/2`, **what will it print ?** `"hello hello hello"` or `"Abra ka Dabra"`?

→ It will print `"hello hello hello"`.



## Why?

- 1) When a request comes in, Express checks routes **in order**.
- 2) It matches the first route that fits, then executes its handler.
- 3) So `/hello` matches before `/hello/2` in this order.

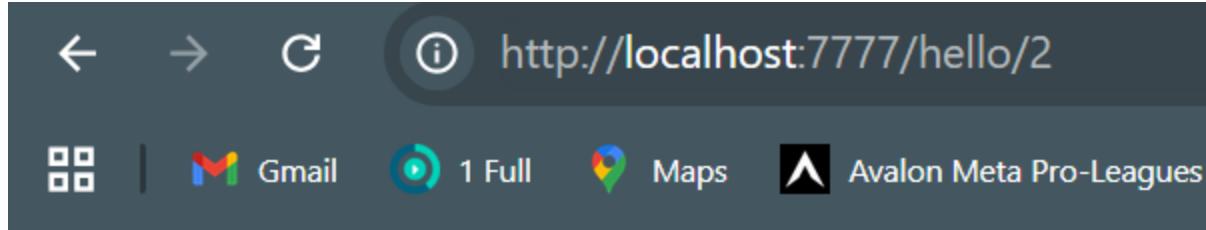
But if I **reverse the order**:

```
app.use("/hello/2", (req, res) => {
  res.send("Abra ka Dabra!!");
});

app.use("/hello", (req, res) => {
  res.send("hello hello hello!!");
});
```

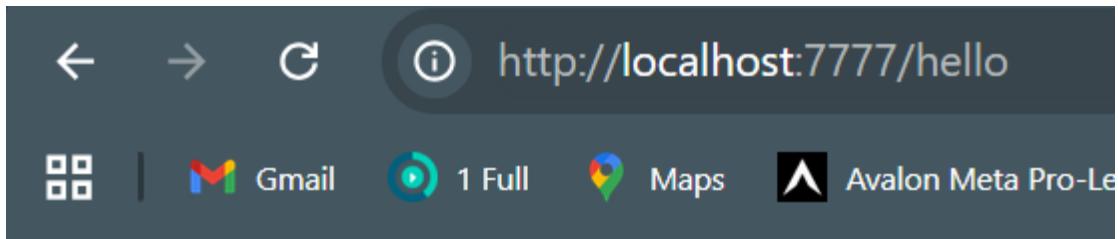
Now, everything works as expected.

1. `/hello/2` → prints **"Abra ka Dabra"**



Abra ka Dabra!!

2. `/hello` → prints “**hello hello hello**”



hello hello hello!!

## So, Order of Routes Matters

1. The **order in which you write routes** is very important in Express.
2. Routes are matched **top to bottom**, and the first match is executed.

## Homework

1. Play with routes and route extensions, e.g., `/hello`, `/test`, `/hello/2`, etc.
2. Observe how changing the order **affects which handler runs**.
3. Important lesson: **the order of routes matters a lot**.

## HTTP Methods

HTTP methods, also known as HTTP verbs, define the type of action a client wants to perform on a resource identified by a URL. These methods are fundamental to how web browsers and applications interact with web servers and APIs. The most commonly used methods include:

1. **GET**: Retrieves data from the server. It is considered "safe" (does not alter server state) and "idempotent" (multiple identical requests have the same effect as a single one).
2. **POST**: Sends data to the server to create a new resource or submit data for processing. It is neither safe nor idempotent, as repeated requests can create multiple resources or trigger multiple actions.
3. **PUT**: Replaces an existing resource or creates a new one if it doesn't exist, at a specified URL. It is idempotent because repeatedly sending the same PUT request will result in the same resource state.
4. **PATCH**: Applies partial modifications to a resource. Unlike PUT, which replaces the entire resource, PATCH only updates specific fields. It is not necessarily idempotent.
5. **DELETE**: Removes a specified resource from the server. It is idempotent as sending multiple DELETE requests for the same resource will result in the resource being deleted only once.

---

When you visit a server via a browser, it sends a **GET request** by default.

How do you make a **POST request**?

Browsers are not ideal for testing APIs, so we'll use **Postman**.



1. Install **Postman** on your system.
2. First, create a **workspace** for yourself in Postman.

A screenshot of the Postman web interface. The top navigation bar includes Home, Workspaces, API Network, and an Upgrade button. A red circle highlights the "Create Workspace" button in the top right corner of the workspace list area. The sidebar on the left shows "orange-equinox-4" and "orange-equinox-41147" workspaces, along with links for Workspaces, Private API Network, Integrations, Reports, and Insights. The main content area displays a "Create Workspace" dialog with a search bar and a "Create" button. Below it, sections for "do in Postman" (REST, Reference documentation, End-to-end testing) are shown.

3. Name your workspace → I'm naming mine **devtinder**.

**Create your workspace**

Name: DevTinder

Select workspace type:

- Internal** Build and test APIs within your team
- Partner** TRY FOR FREE Share APIs securely with trusted partners
- Public** Make APIs accessible to the world

Manage access:

- Everyone in orange-equinox-41147 1 member
- Only you and invited people

**Create**

**Blank workspace**

Customize this space to organize and share your API resources with your team.

**Your workspace**

**Showcase your API's capabilities**  
Use Postman collections to document your APIs with ease. You can create your own or choose from 70+ collection templates tailored to your needs.

**Build together, work faster**

#### 4. This is how the **workspace window** will look.

**DevTinder**

New Import

Collections

Environments

Flows

History

My first collection

First folder inside collection

GET  
POST  
GET

Second folder inside collection

GET  
GET

Create a collection for your requests

A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.

Create Collection

Overview

DevTinder

Overview Updates Settings

Connect Unwatch 1

Workshop description

Add a description to accelerate team collaboration. Here's what you can include:

- Introduction
- How-to-guides
- Best practices
- FAQs

Pinned collections

Make it easy to discover notable collections in this workspace by pinning them here.

Explore workspace templates

Online Find and replace Console

Runner Start Proxy Cookies Vault Trash

## Testing an API in Postman

1. Click on the **New** button on the left.

The screenshot shows the API Network interface with the 'DevTinder' workspace selected. The 'Overview' tab is active. On the left, there's a sidebar with 'Collections', 'Environments', 'Flows', and 'History'. The main area displays 'My first collection' with 'First folder inside collection' and 'Second folder inside collection', each containing several requests. Below this, there's a section for creating a collection and a detailed description of the HTTP protocol. Various icons for different API types like GraphQL, AI, MCP, and gRPC are shown on the right.

## 2. It will ask what kind of request you want to make.

The screenshot shows the Postman interface with an 'Untitled Request' titled 'HTTP'. The 'GET' method is selected. The main panel contains fields for 'Headers (7)', 'Body', 'Scripts', 'Tests', and 'Settings'. To the right, there's a 'Cookies' section. On the left, a sidebar lists other methods: POST, PUT, PATCH, DELETE, HEAD, and OPTIONS. Below the main panel is a cartoon illustration of an astronaut with a rocket pack. At the bottom, there's a placeholder text 'Enter the URL and click Send to get a response'.

### 3. Select **HTTP Request**.

4. Then choose the type of request: **GET, POST, PATCH**, etc.

Now, let's make a **GET request** to this URL:

http://localhost:7777/hello

The screenshot shows the Postman interface with a successful API call. At the top, there's a header bar with the URL "http://localhost:7777/hello". Below it, the request method is set to "GET". The "Params" tab is selected, showing a single query parameter "Key" with the value "Value". In the "Body" section, the response status is "200 OK" with a response time of "31 ms" and a size of "247 B". The response body is displayed as "hello hello hello!!".

You got the "**hello hello hello**" output → perfect! 🎉

So, we've successfully made an API call using **Postman**. It's simple, right?

## Your Task

Create a workspace in Postman and make API calls for the following routes:

- /test
- /hello

Welcome back! 🙌

I hope you've installed Postman and tested your GET APIs.

Now, let's make a **POST** call to the same URL:

```
http://localhost:7777/hello
```

so we have successfully make an api call using postman its simple right

The screenshot shows the Postman interface with a POST request to `http://localhost:7777/hello`. The 'Params' tab is selected. In the 'Query Params' section, there is a table with one row containing 'Key' and 'Value'. The 'Body' tab shows the response: a status of `200 OK`, a duration of `6 ms`, a size of `247 B`, and a preview of the response body which contains the text `hello hello hello!!`.

What? It's giving the same output? 🤯

Yes ,

that's because `app.use()` matches **all HTTP methods** (GET, POST, PUT, DELETE, etc.).

But ideally, if you make a **GET** call to `/profile` , it should return the user's profile data.

And if you make a **POST** call to `/profile` , it should **add or update** the profile data.

## So how do we handle that?

By using **method-specific handlers** like `app.get()` and `app.post()` instead of `app.use()` .

Here's an example 😊

```

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

// This will only handle GET requests to /user
app.get("/user", (req, res) => {
  res.send({ firstName: "Akshay", lastName: "Saini" });
});

// This will handle all HTTP method requests to /test
app.use("/test", (req, res) => {
  res.send("Hello from the server!!!");
});

app.listen(7777, () => {
  console.log("Server is successfully listening on port 7777");
});

```

## Now, when you test this API in Postman:

1. **GET /user** → You'll get { firstName: "Akshay", lastName: "Saini" }

The screenshot shows the Postman interface with a red box highlighting the response body. The URL in the header is `http://localhost:7777/user`. The method dropdown shows `GET`. The response status is `200 OK` with a `14 ms` latency and `276 B` size. The response body is displayed as JSON:

```

1  {
2   "firstName": "Akshay",
3   "lastName": "Saini"
4 }

```

2. **POST /user** → Won't work, because only GET is handled.

The screenshot shows the Postman interface. At the top, it says "POST" and "http://localhost:7777/user". Below that is a table for "Query Params" with one row: "Key" and "Value". The main area shows the "Body" tab selected, displaying the following HTML code:

```

7 </head>
8
9 <body>
10 | <pre>Cannot POST /user</pre>
11 </body>
12
13 </html>

```

At the bottom, there are buttons for "Find and replace" and "Console". To the right, there are links for "Runner", "Start Proxy", "Cookies", "Vault", and "Trash".

### 3. GET /test or POST /test → Both will return "Hello from the server!!"

That's the difference between `app.use()` and `app.get()` :

- `app.use()` matches **every HTTP method**,
- while `app.get()` handles **only GET requests**.

Now, if you add a **POST** and **DELETE** route to your code, each HTTP method will handle a different type of request → just like in real-world APIs.

```

const express = require("express");

const app = express();

// Handle POST request → Save data to the database
app.post("/user", (req, res) => {
  // saving data to DB
  res.send("Data successfully saved to the DB!");
});

// Handle DELETE request → Delete user data
app.delete("/user", (req, res) => {
  res.send("Deleted successfully!!!");
});

```

```
app.listen(7777, () => {
  console.log("Server is successfully listening on port 7777");
});
```

Now, when you test this in **Postman**

1. **POST /user** → returns "Data successfully saved to the DB!"

The screenshot shows the Postman interface with the following details:

- Request URL:** http://localhost:7777/user
- Method:** POST
- Headers:** (8)
- Body:** (Empty)
- Tests:** (Empty)
- Settings:** (Empty)
- Query Params:** (Empty)
- Params:** (Empty)
- Cookies:** (Empty)
- Body:** (Empty)
- Cookies:** (Empty)
- Headers:** (7)
- Test Results:** (Empty)
- Preview:** (Empty)
- Visualize:** (Empty)
- Response Status:** 200 OK
- Response Time:** 14 ms
- Response Size:** 261 B
- Response Body:** 1 data successfully saved to the db
- Bottom Navigation:** Find and replace, Console, Runner, Start Proxy, Cookies, Vault, Trash

2. **DELETE /user** → returns "Deleted successfully!!"

The screenshot shows the Postman interface. At the top, it says "HTTP" and "http://localhost:7777/user". Below that, there's a "DELETE" button and a "Send" button. The "Params" tab is selected. In the "Query Params" table, there is one row with "Key" and "Value" both set to "Value". On the right side, there are tabs for "Cookies", "Body", "Cookies", "Headers (7)", and "Test Results". The "Test Results" tab is active, showing a green "200 OK" status with a response time of "17 ms" and a response size of "250 B". The response body contains the text "1 deleted successfully!". At the bottom, there are buttons for "Find and replace", "Console", "Runner", "Start Proxy", "Cookies", "Vault", and "Trash".

Each route responds differently based on the **HTTP method** and that's how real APIs separate **fetching**, **creating**, and **deleting** operations.

## ADVANCE CONCEPTS

When you write this code:

```
app.use("/user", (req, res) => {
  res.send("HAHAHAHAHAHAHA");
});
```

and place it **before** all your other routes (`app.get`, `app.post`, etc.),

Express will **match it first** → because `app.use()` handles **all HTTP methods** (GET, POST, DELETE, etc.) and any route that **starts with** `/user`.

So when a request like `/user`, `/user/1`, or `/user/profile` comes in,

it gets caught by this `app.use()` handler before Express ever reaches your `app.get("/user")`, `app.post("/user")`, or `app.delete("/user")` routes.

That's why every request returns "**HAHAHAHAHAHA**" 😂

The screenshot shows the Postman interface. At the top, it says "HTTP New Collection / /user". Below that, a "POST" button is selected, and the URL "http://localhost:7777/user" is entered. To the right, there are "Save", "Share", and "Send" buttons. Under the "Params" tab, there is a table for "Query Params" with two rows: one for "Key" and one for "Value". In the main area, under "Body", the response is shown as "1 HAHAHAHAHHA". Above the response, status information is displayed: "200 OK" with a green background, "14 ms", "239 B", and a globe icon. To the right of the status, there are icons for "Save Response", "e.g.", and a copy link.

To fix this, you should always place your **specific routes before the generic ones**.

For example:

```
app.get("/user", (req, res) => {
  res.send({ firstName: "Akshay", lastName: "Saini" });
});

app.post("/user", (req, res) => {
  console.log("save data to the db");
  res.send("Data successfully saved to the DB!");
});

app.delete("/user", (req, res) => {
  res.send("Deleted successfully!!!");
});

// Keep this generic one last
app.use("/user", (req, res) => {
```

```
res.send("HAHAHAHAHAHAH");  
});
```

The order in which you define routes in Express **directly affects** which one gets executed → Express matches routes **from top to bottom** and stops at the first match.

Now that we've learned so much, it's time to pause and do a quick revision or homework.

### Your task:

Write the logic to handle **GET, POST, PUT, and DELETE** API calls and test them using **Postman**.



## Advanced Routing in Express

If I create a route `/abc` and make an API call to it, it works fine:

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

app.get("/abc", (req, res) => {
  res.send({ firstName: "Akshay", lastName: "Saini" });
});

app.listen(7777, () => {
  console.log("Server is successfully listening on port 7777");
});
```

But suppose I put a `?` before `c` → this means `b` is optional.

The screenshot shows the Postman application interface. At the top, there's a header bar with 'HTTP' and 'New Collection / user'. On the right, there are 'Save', 'Share', and a 'Send' button. Below the header, the request details are shown: 'GET' method, URL 'http://localhost:7777/abc', and a 'Send' button. Underneath, there are tabs for 'Params', 'Authorization', 'Headers (7)', 'Body', 'Scripts', 'Tests', 'Settings', and 'Cookies'. The 'Params' tab is selected, showing a table for 'Query Params' with one row: 'Key' and 'Value'. In the main body area, the 'Body' tab is selected, showing a JSON response: { "firstName": "Akshay", "lastName": "Saini" }. Above the body, the status is '200 OK' with metrics: 14 ms, 276 B, and a globe icon. To the right of the body, there are icons for 'Save Response', 'Copy', 'Find', and 'Edit'.

So if I make a call to `/abc` it will work, and even if I call `/ac`, it will still work fine.

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

app.get("/ab?c", (req, res) => {
  res.send({ firstName: "Akshay", lastName: "Saini" });
});

app.listen(7777, () => {
  console.log("Server is successfully listening on port 7777");
});
```

You can also write patterns like this:

```
app.get("/a(bc)+d", (req, res) => {
  res.send({ firstName: "Akshay", lastName: "Saini" });
});
```

If you open

```
http://localhost:7777/abcabcabcd
```

it will work perfectly!

### Note

The syntax

```
app.get('/ab?c', (req, res) => { ... });
```

was valid in **Express versions using path-to-regexp version 6 or below**,

which includes roughly **Express 4.x** and **early 5.x alpha** releases.

 Works on:

```
"express": "^4.18.2"
```

✖ Breaks on:

```
"express": "^5.0.0-beta.1" or newer
```

These complex patterns are rarely used in real projects, but it's good to know about them.

## Query Parameters

Suppose you have a URL like this:

```
http://localhost:7777/user?userId=101
```

If you make a call to it, how can you get the `userId` inside your route handler?

Use `req.query` → it gives you all the query parameters.

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

app.get("/user", (req, res) => {
  console.log(req.query);
  res.send({ firstName: "Akshay", lastName: "Saini" });
});

app.listen(7777, () => {
  console.log("Server is successfully listening on port 7777");
});
```

Now, go to Postman, make an API call, and check your terminal , you'll see the query params printed there.

```
$ npm run dev
Server is successfully listening on port 7777
[nodemon] restarting due to changes...
[nodemon] starting `node src/app.js`
Server is successfully listening on port 7777
[Object: null prototype] { userId: '101' }
```

## Dynamic Routes

If you have a URL like

<http://localhost:7777/user/707>

you can handle it using **route parameters**.

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

app.get("/user/:userId", (req, res) => {
  console.log(req.params);
  res.send({ firstName: "Akshay", lastName: "Saini" });
});

app.listen(7777, () => {
  console.log("Server is successfully listening on port 7777");
});
```

```
$ npm run dev
Server is successfully listening on port 7777
[nodemon] restarting due to changes...
[nodemon] starting `node src/app.js`
Server is successfully listening on port 7777
[Object: null prototype] { userId: '707' }
```

Now, when you make an API call and check the terminal, you'll see `{ userId: '707' }`.

You can also make more complex routes using multiple parameters:

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

app.get("/user/:userId/:name/:password", (req, res) => {
  console.log(req.params);
  res.send({ firstName: "Akshay", lastName: "Saini" });
});

app.listen(7777, () => {
  console.log("Server is successfully listening on port 7777");
});
```

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:7777/user/707/akshay/password`. The response status is 200 OK, and the response body is:

```

1  {
2   "firstName": "Akshay",
3   "lastName": "Saini"
4 }

```

The terminal output shows the command `$ npm run dev` being run. The output indicates the server is restarting due to changes and starting on port 7777. The response object contains user data:

```

[nodemon] restarting due to changes...
[nodemon] starting `node src/app.js`
Server is successfully listening on port 7777
[Object: null prototype] {
  userId: '707',
  name: 'akshay',
  password: 'password'
}

```

## Final Homework

- Explore **routing** and the use of special symbols like `?`, `.`, and `()` in routes.
- Try using **regex** in routes, e.g. `/a/` or `/.*fly$/`.
- Practice **reading query parameters** and **handling dynamic routes**.