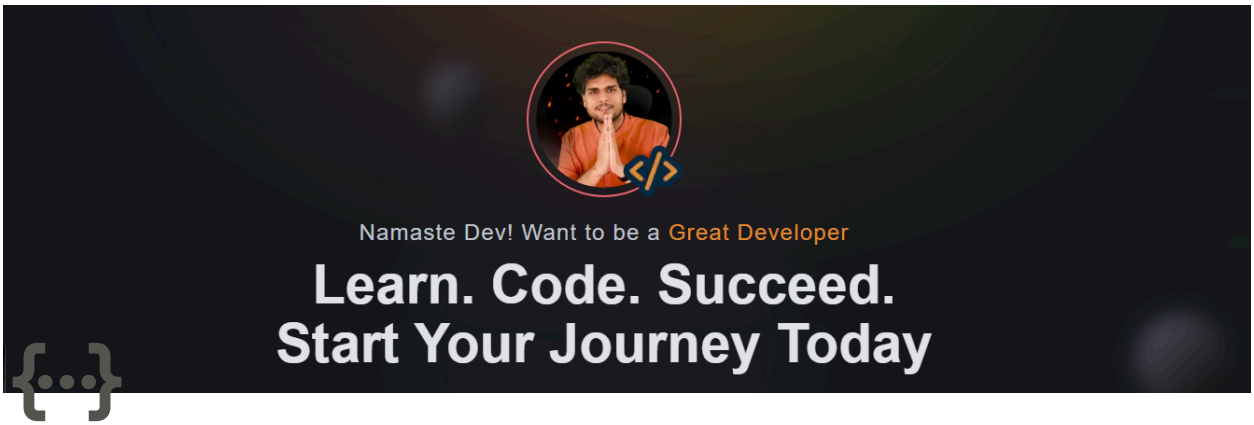
Route Handler : a function or block of code that executes when a specific URL (route) in a web application is accessed.



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**.

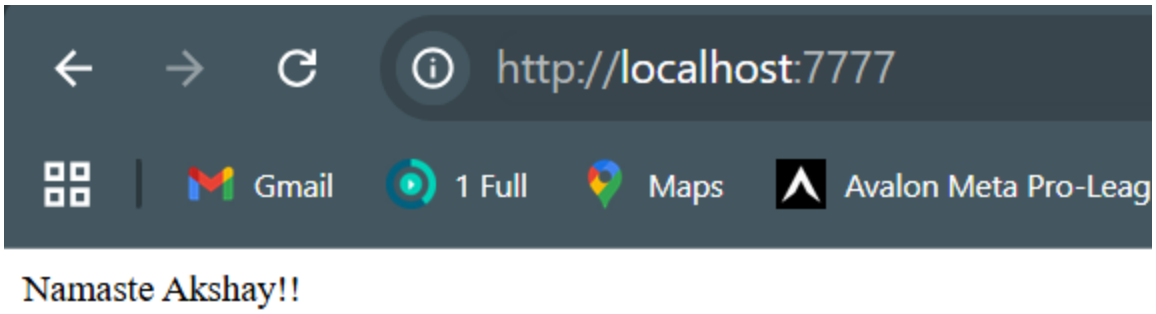
Next, I'll show you something a bit tricky { I want your 100% focus for this part }

Understanding Routes in Express

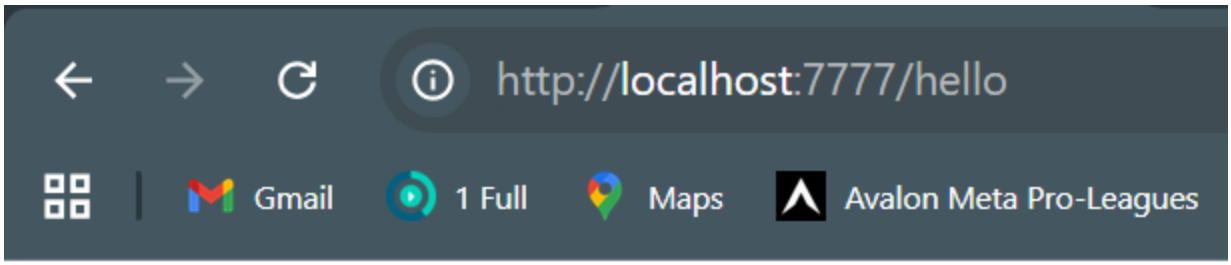
```
const express = require("express");

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");
});
```

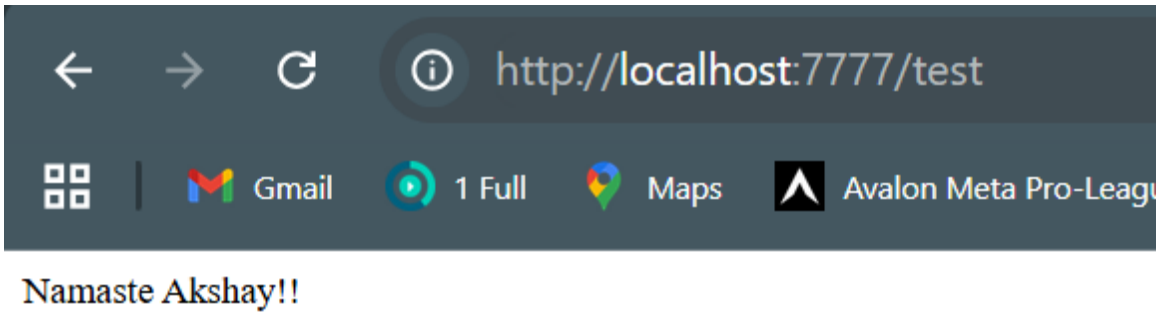
Let's check the `/` route.



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

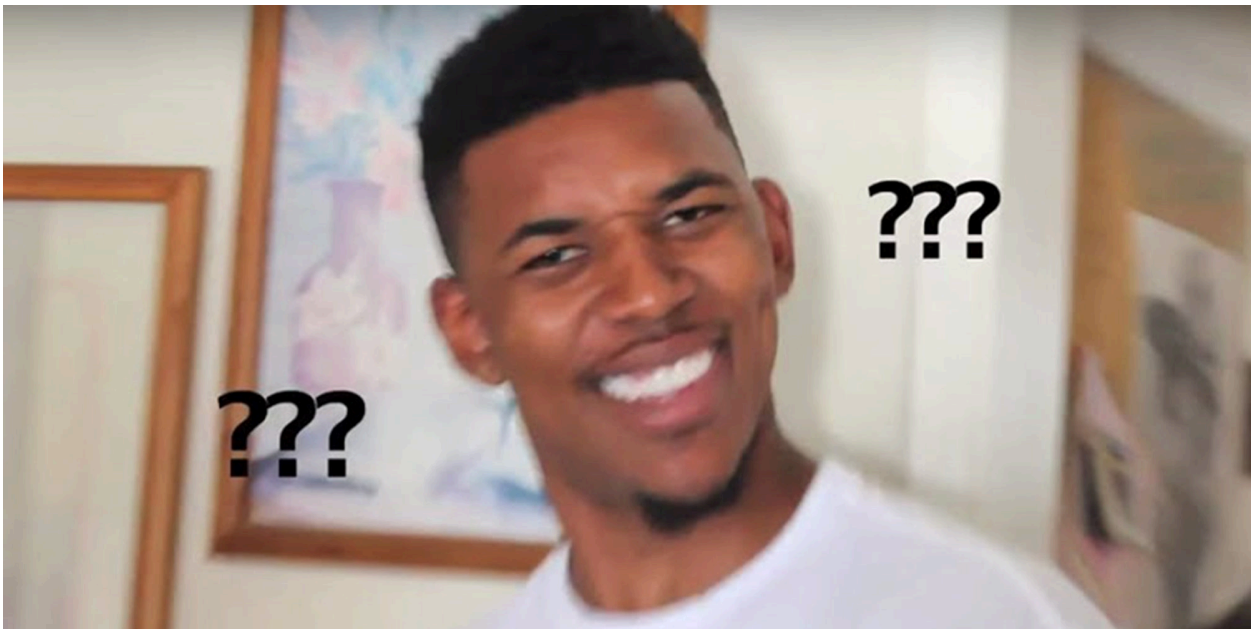


Hmm, something seems wrong. Let's check the `/test` route.



It's still showing the same issue😑

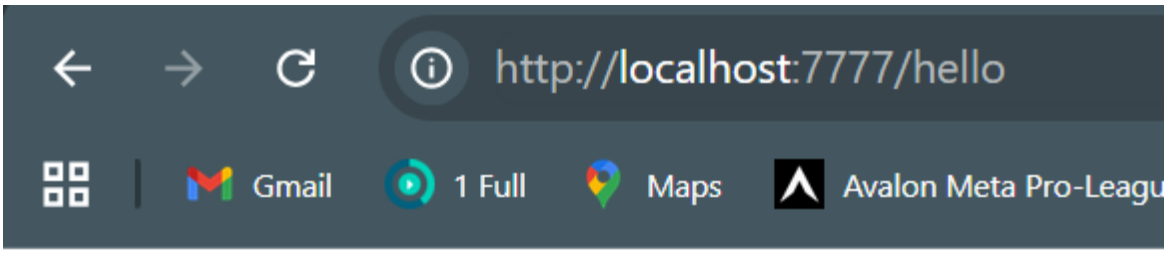
why isn't it displaying "Hello from the server" on the `/test` route?



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.

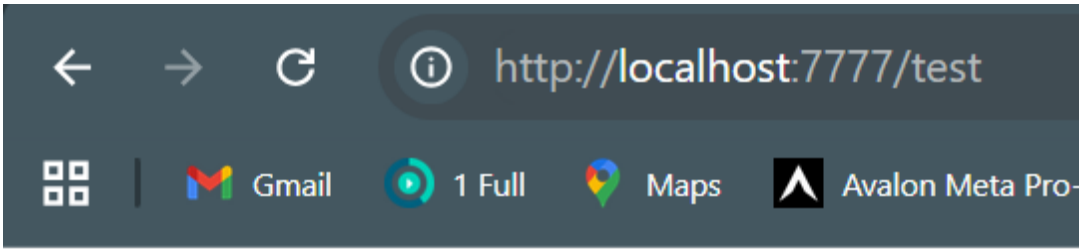
```
const express = require("express");

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");
});
```



hello hello hello!!

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

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



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:

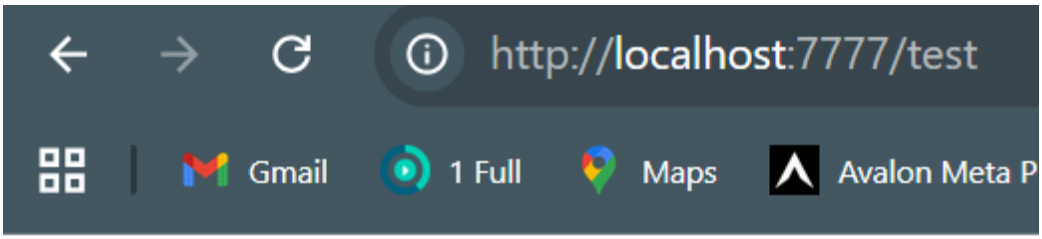
Does the sequence of your code matter?


```
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.

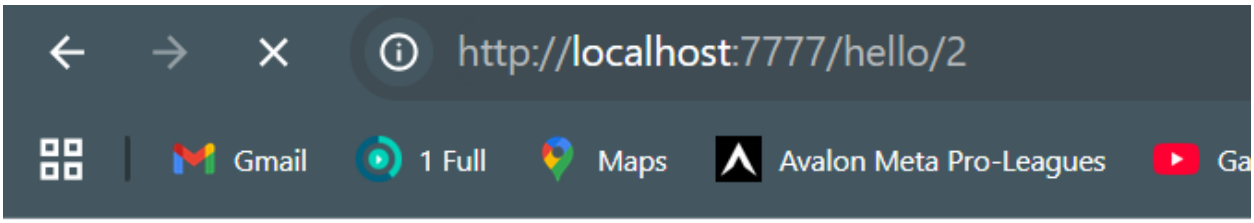


Hello from the server!!

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”.**



hello hello hello!!

Why?

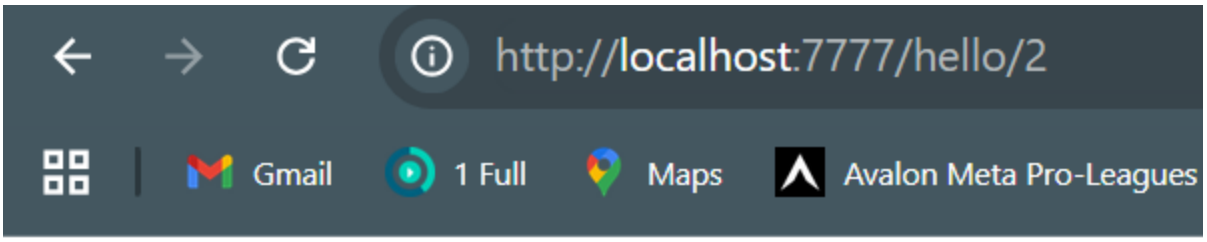
1) When a request comes in, Express checks routes **in order**.
2) It matches the first route that fits, then executes its handler.
3) So `/hello` matches before `/hello/2` in this order.

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!!");
});
```

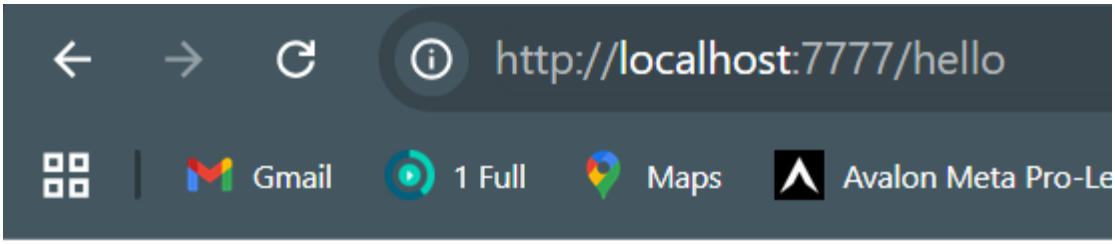
Now, everything works as expected.

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:

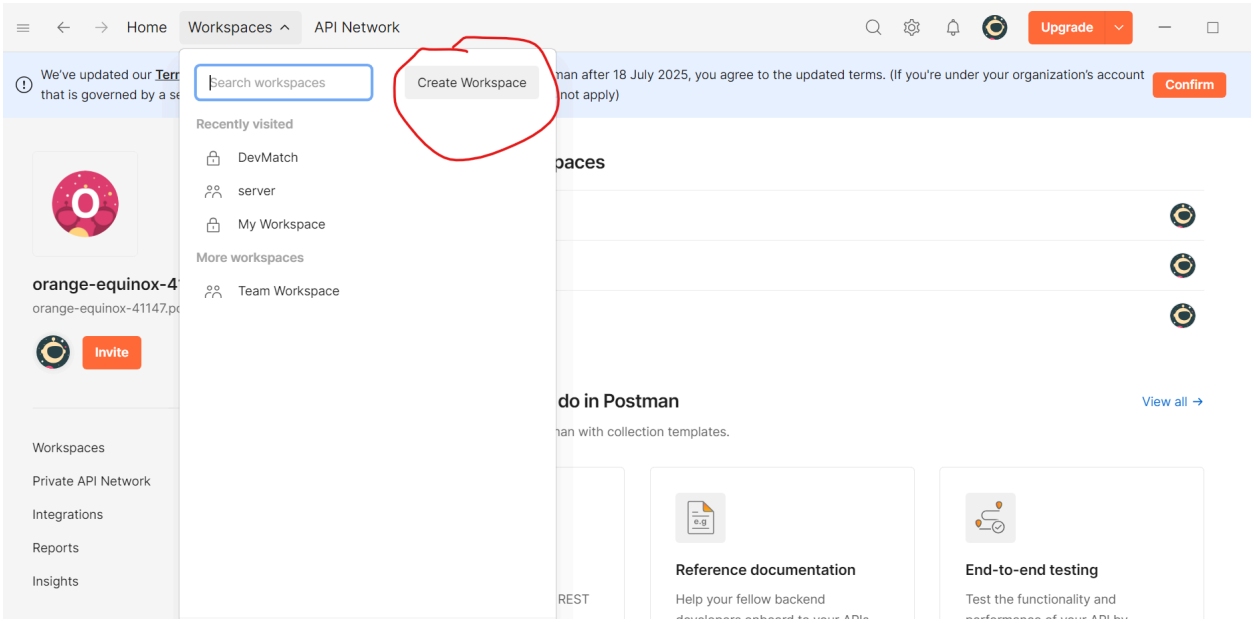
1. **GET:** Retrieves data from the server. It is considered "safe" (does not alter server state) and "idempotent" (multiple identical requests have the same effect as a single one).
2. **POST:** Sends data to the server to create a new resource or submit data for processing. It is neither safe nor idempotent, 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.

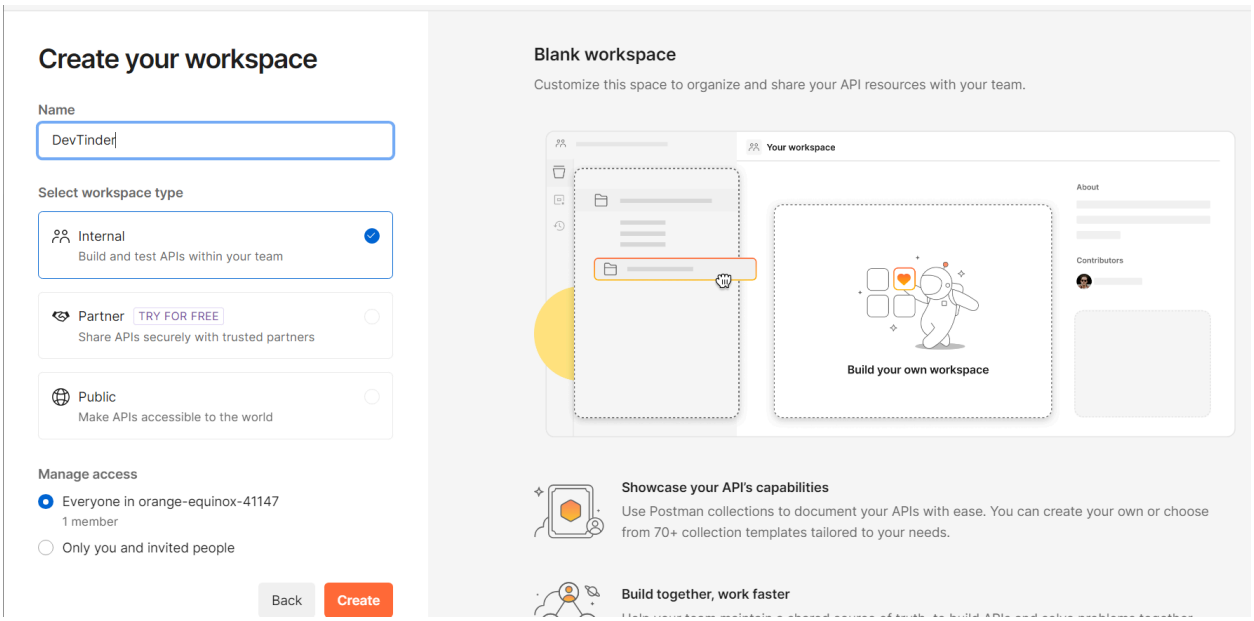
How do you make a **POST request**?

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

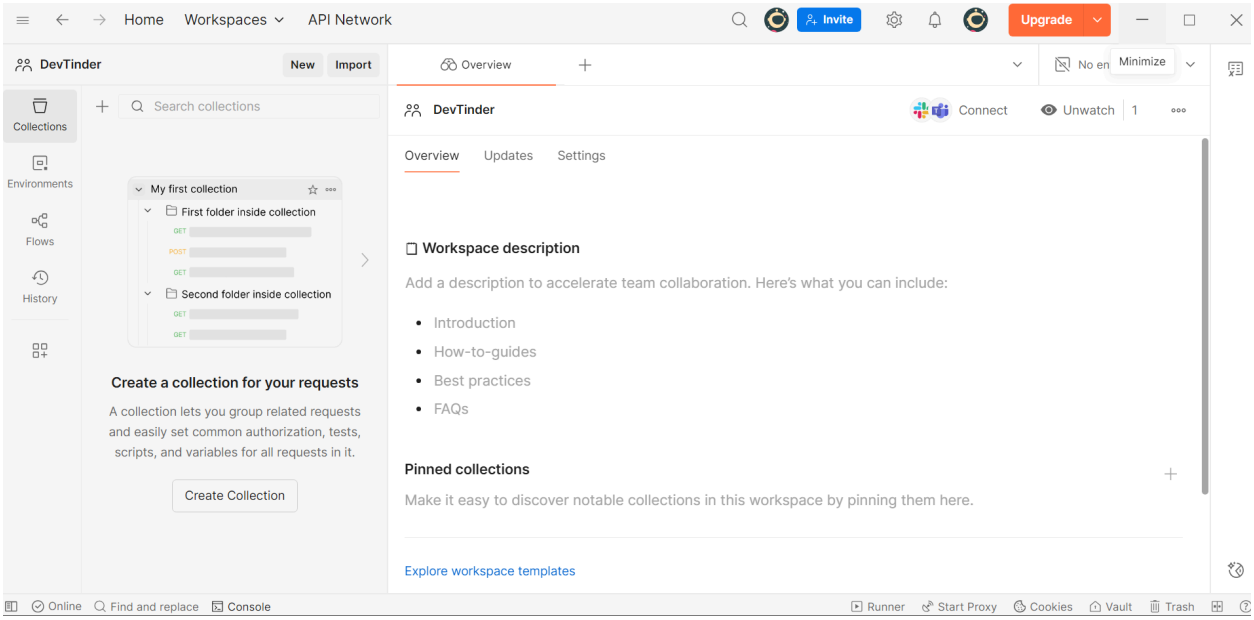
1. Install **Postman** on your system.
2. First, create a **workspace** for yourself in Postman.



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

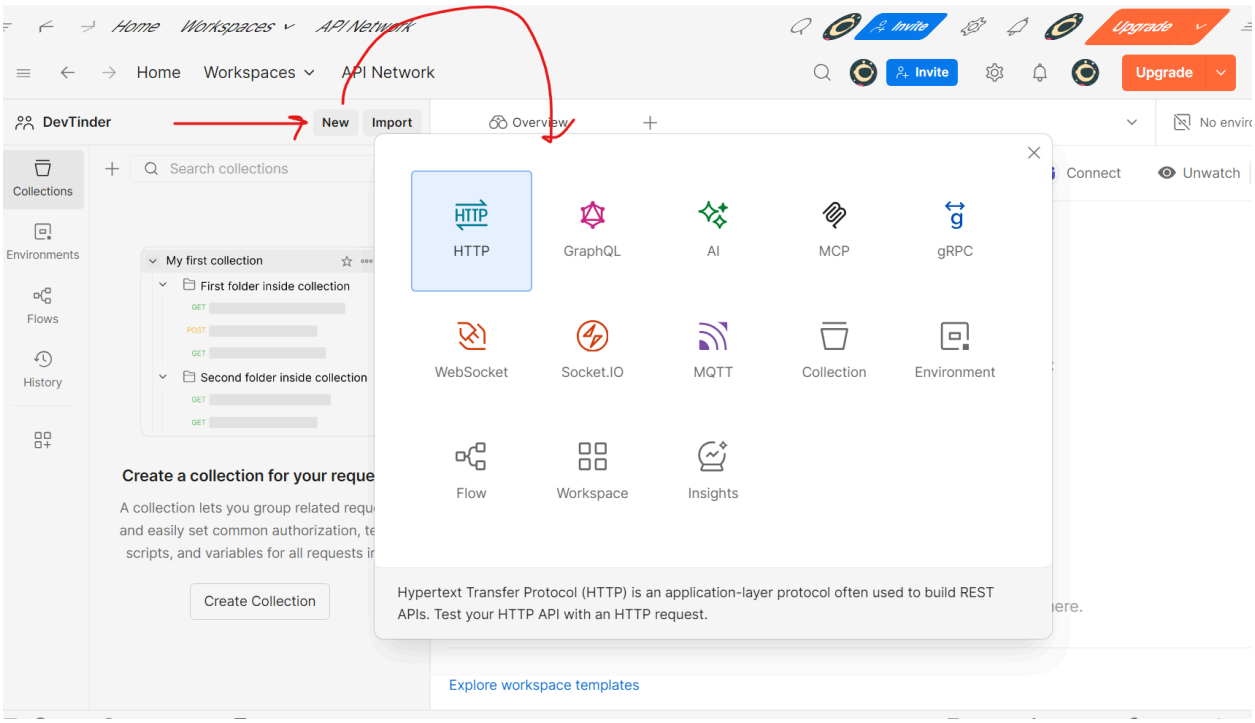


4. This is how the **workspace window** will look.

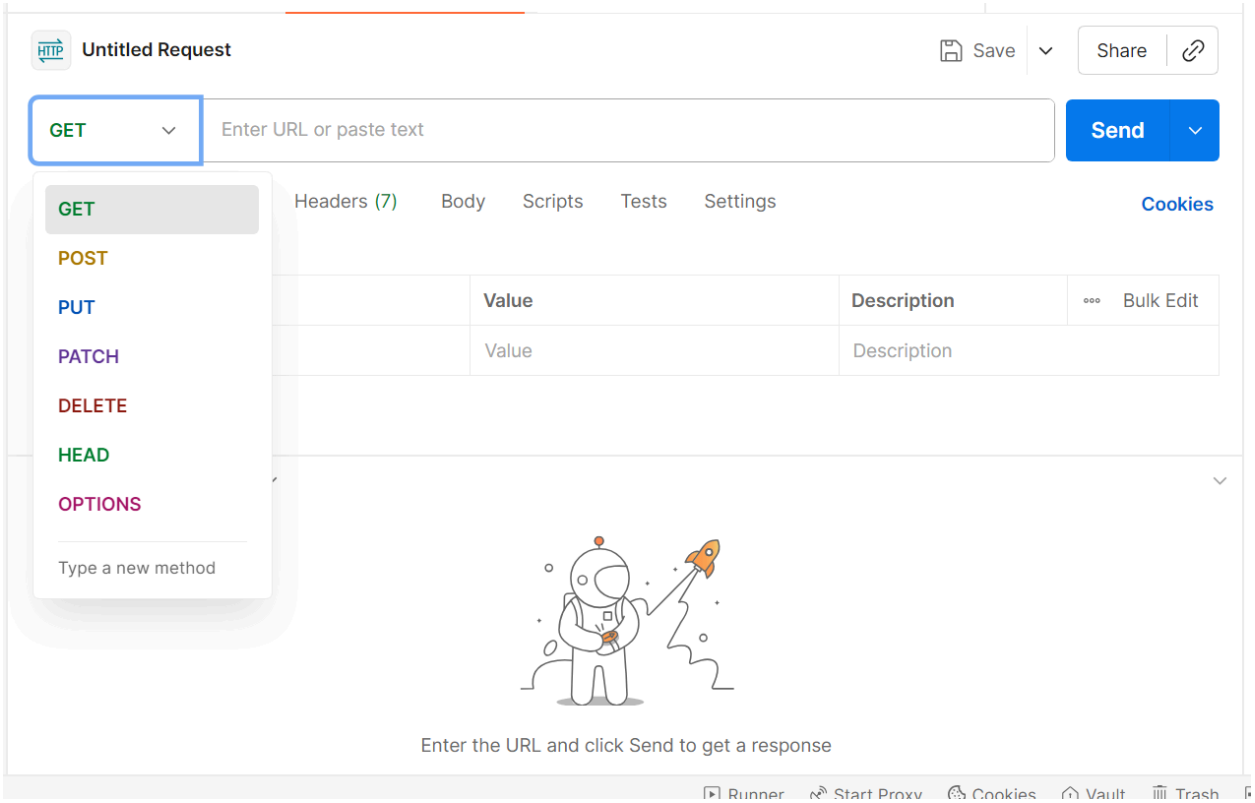


Testing an API in Postman

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



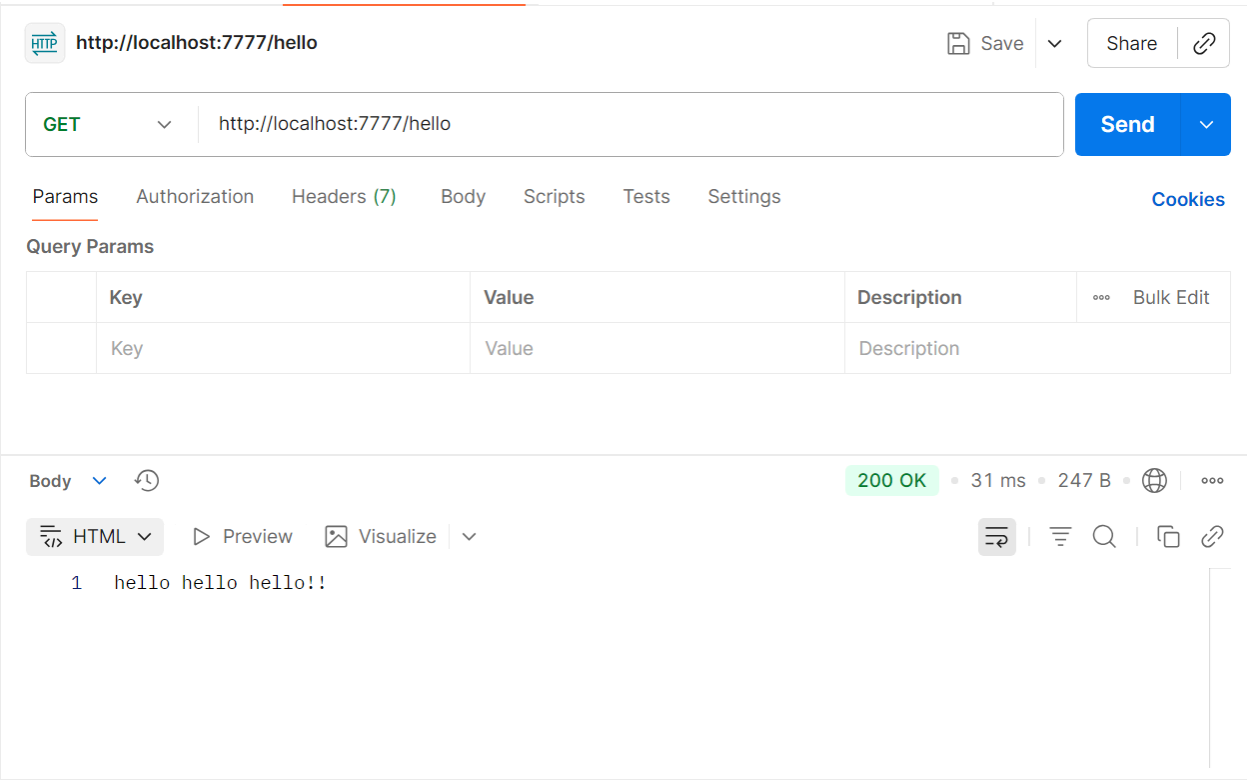
2. It will ask what kind of request you want to make.



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
```



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`

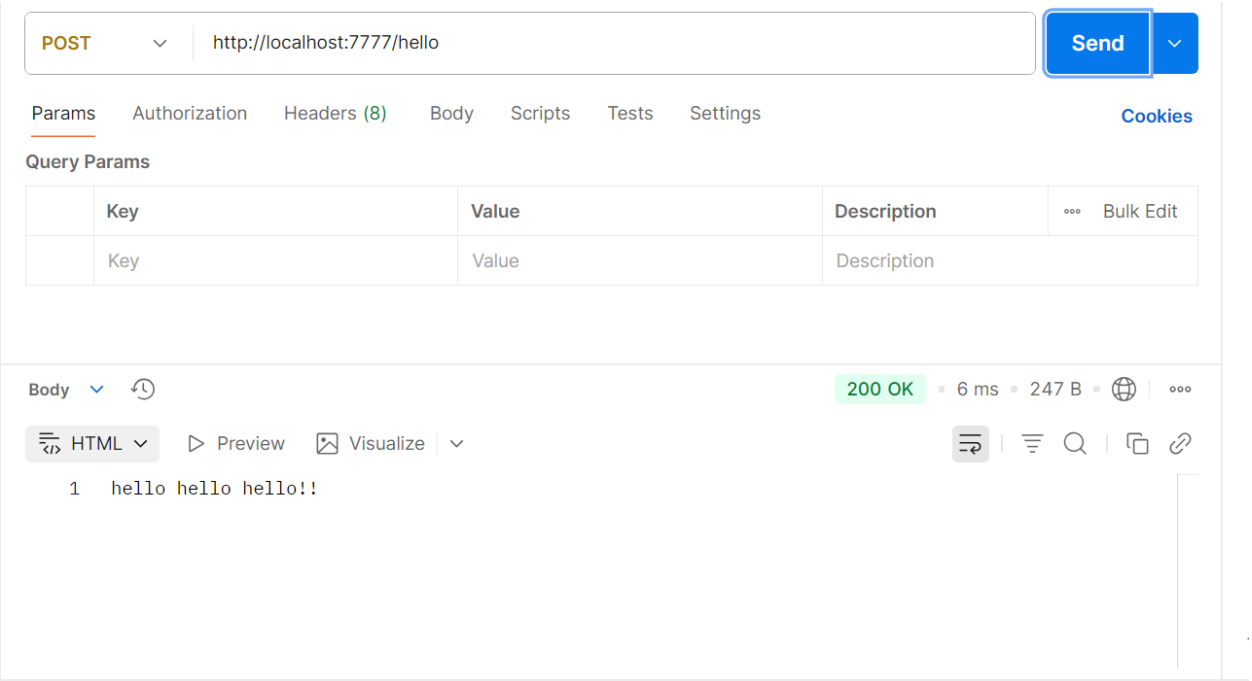
Welcome back! 👋

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



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?

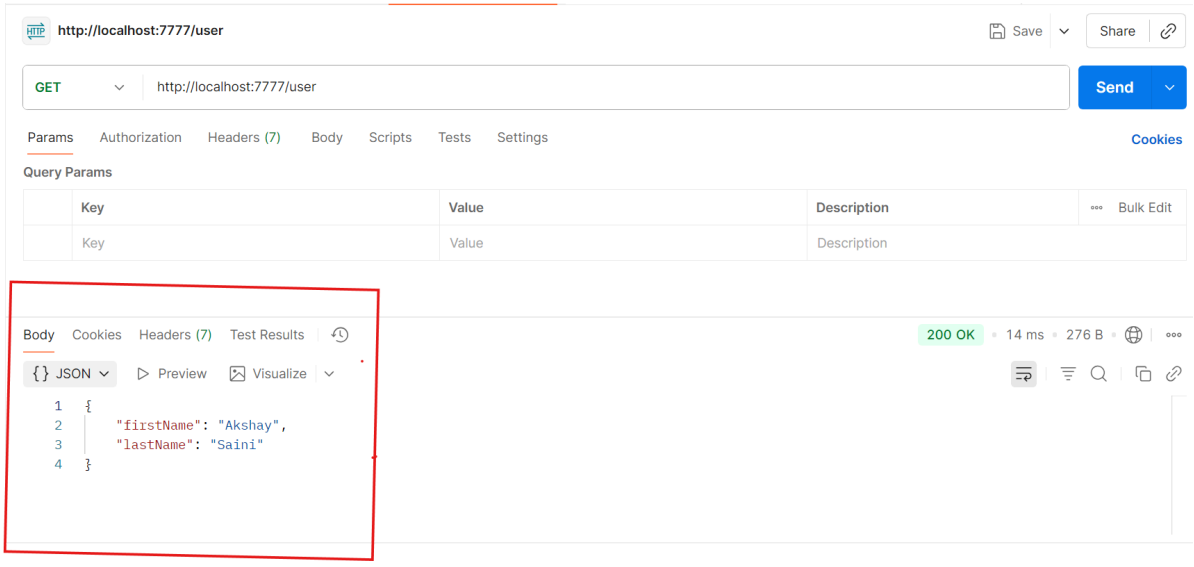
By using **method-specific handlers** like `app.get()` and `app.post()` instead of `app.use()`.

Here's an example 😀

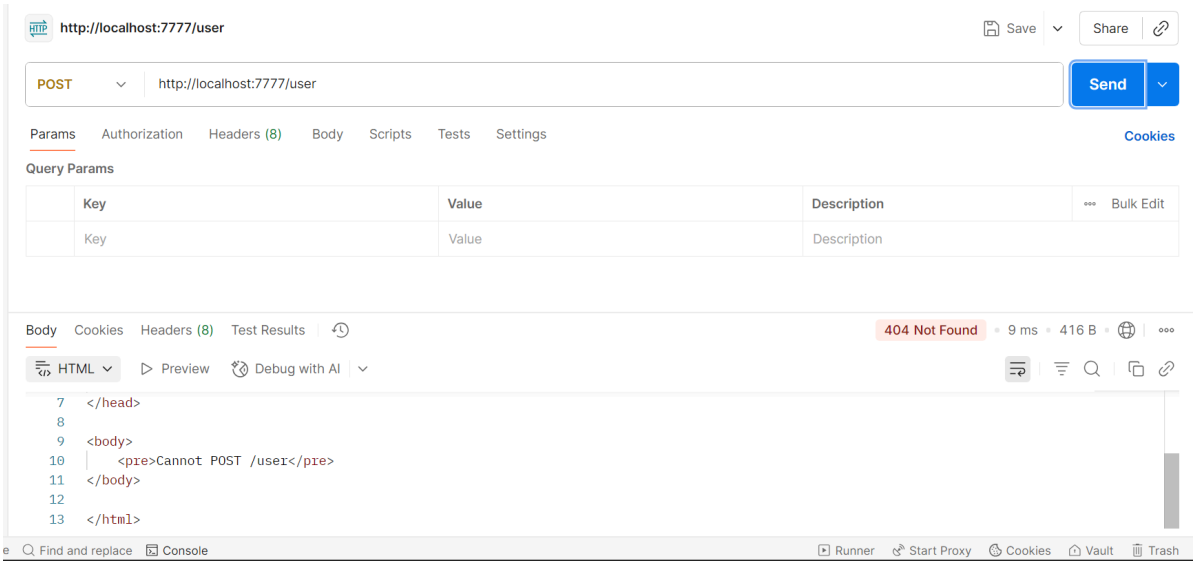
```
const express = require("express");
const app = express();

// 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" }`



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



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.**

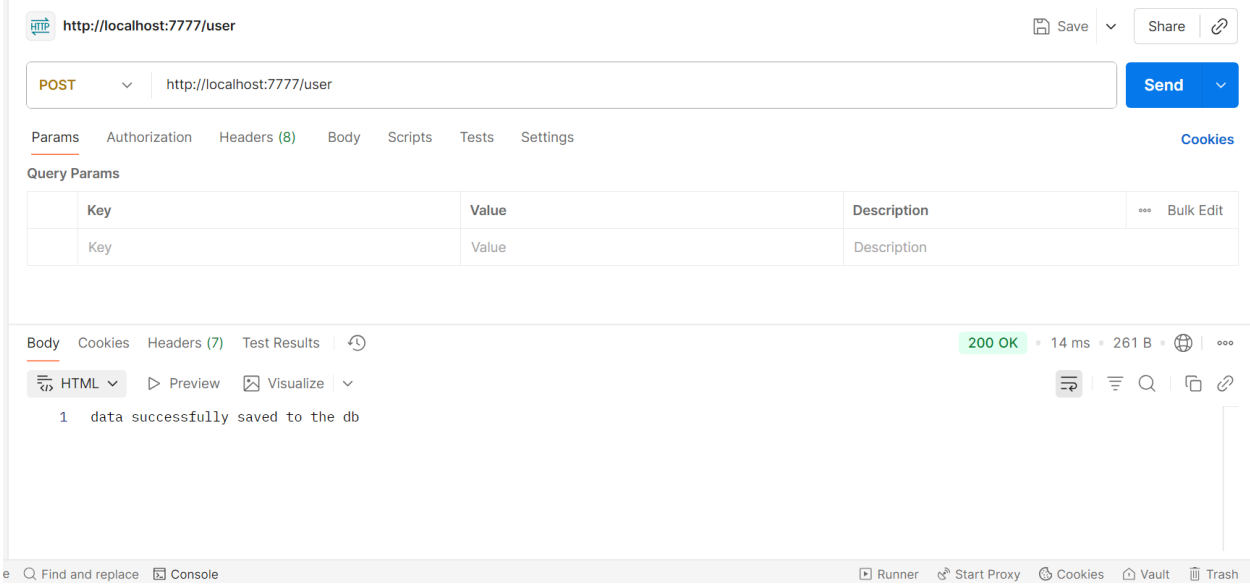
Now, if you add a **POST** and **DELETE** route to your code, each HTTP method will handle a different type of request → just like in real-world APIs.

```
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!!");
});
```

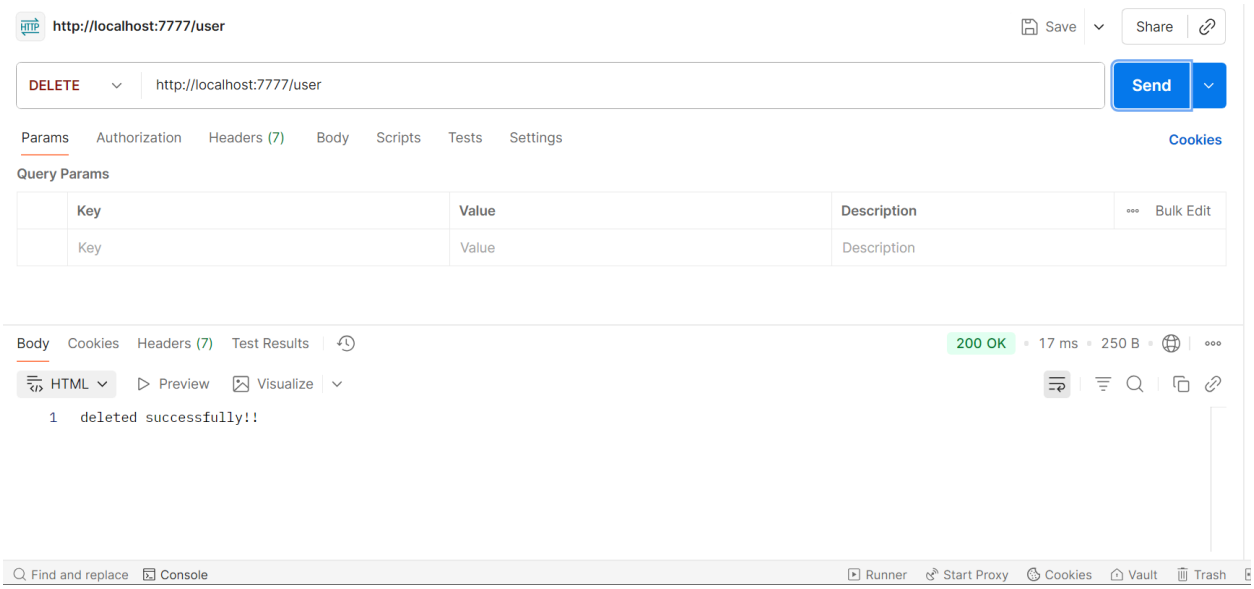
```
app.listen(7777, () => {
  console.log("Server is successfully listening on port 7777");
});
```

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

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



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



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("HAHAHAHAHAHA");
});
```

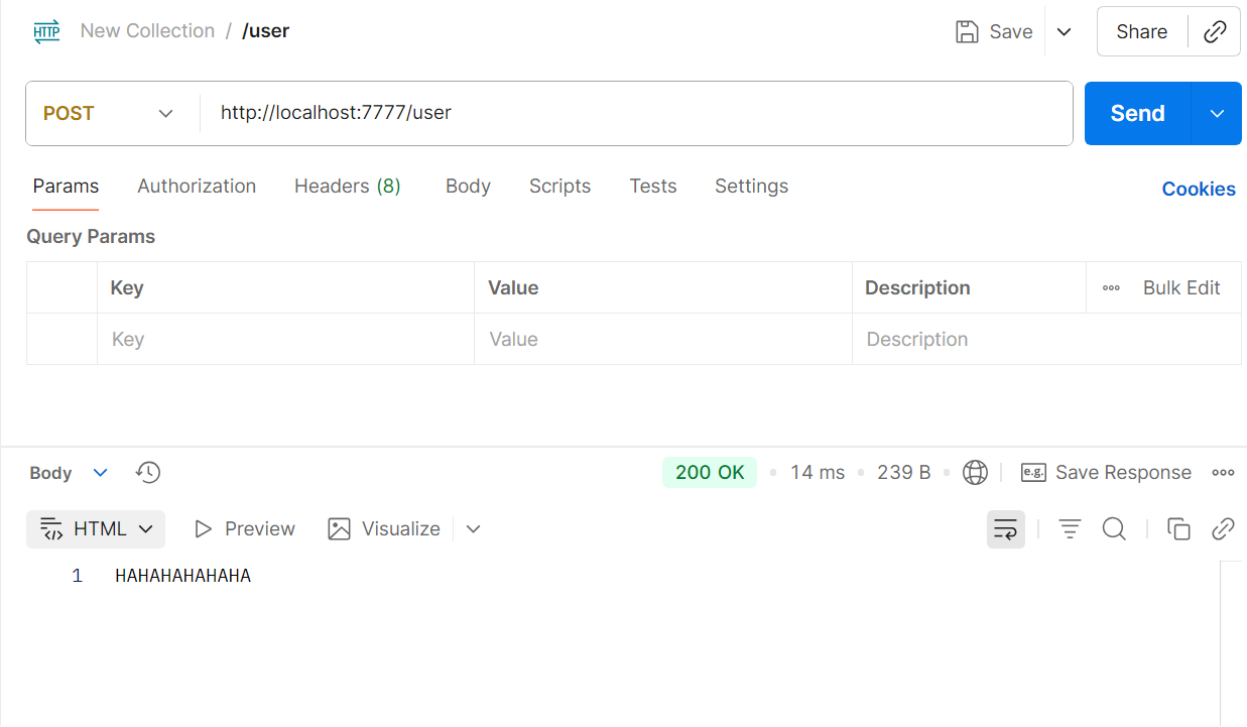
and place it **before** all your other routes (`app.get`, `app.pos t`, etc.),

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"** 😂



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("HAHAHAHAHAHA");
});
```

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:

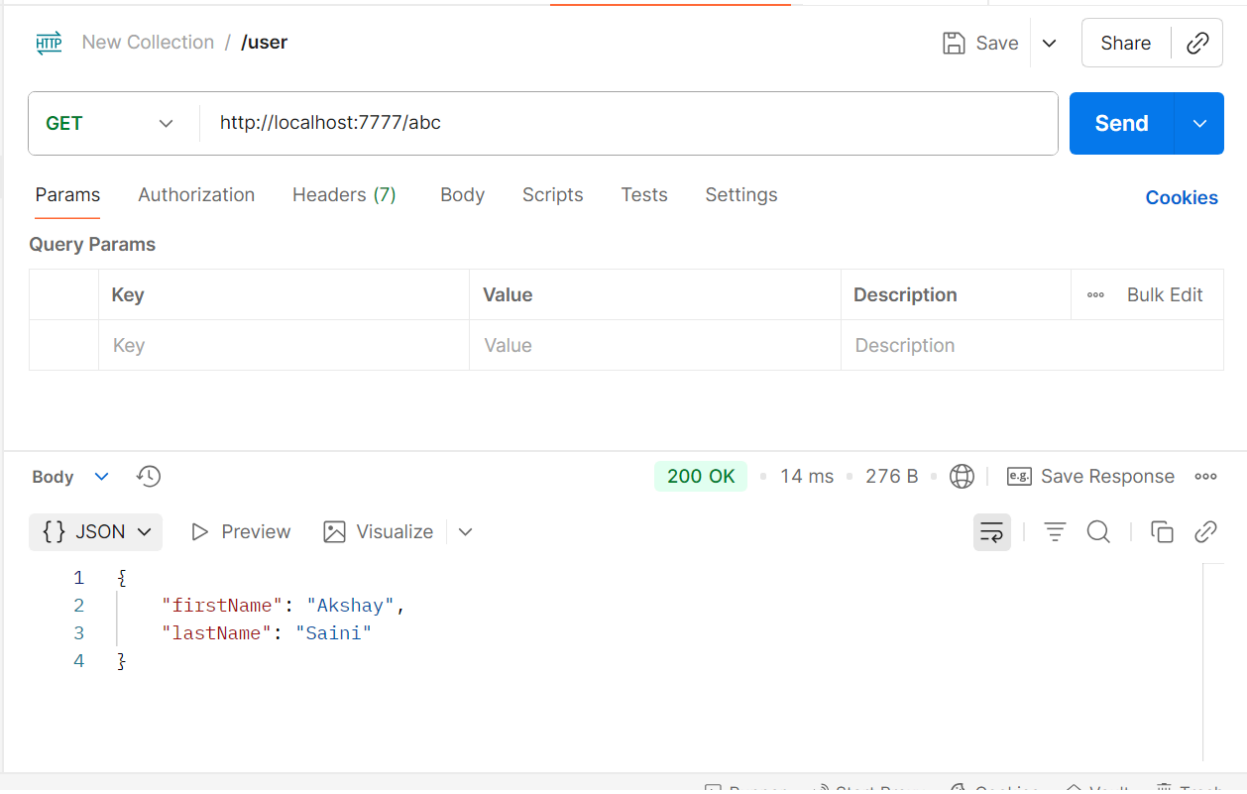
Write the logic to handle **GET, POST, PUT, and DELETE** API calls and test them using **Postman**.

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.



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/abcbcbcbcbcd
```

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:

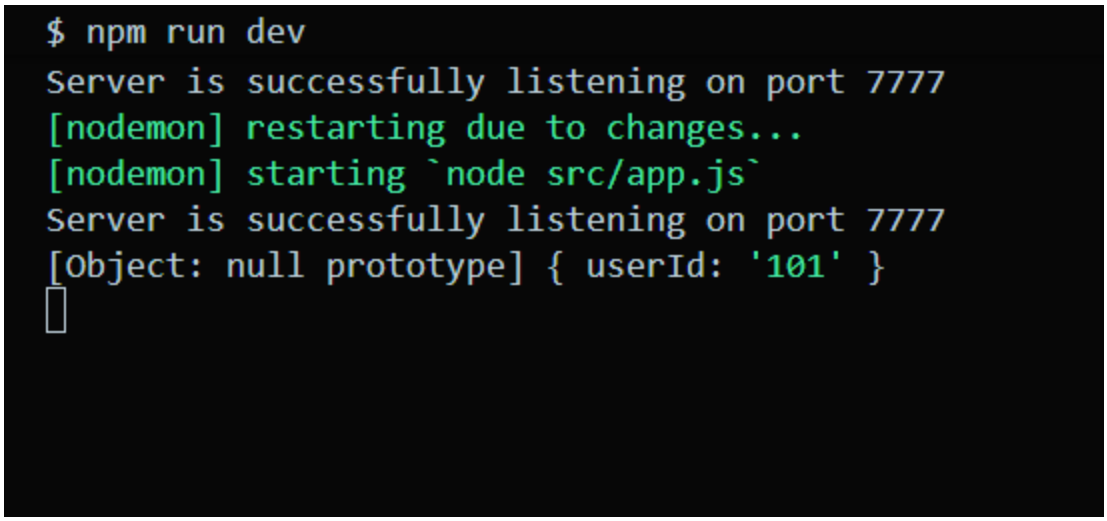
`http://localhost:7777/user?userId=101`

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.



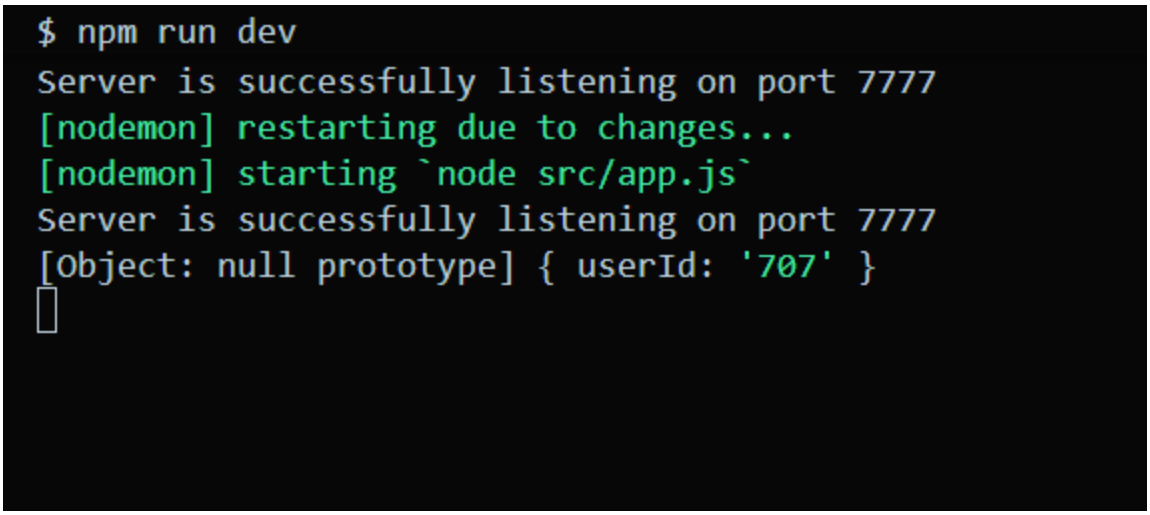
Dynamic Routes

If you have a URL like

`http://localhost:7777/user/707`

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

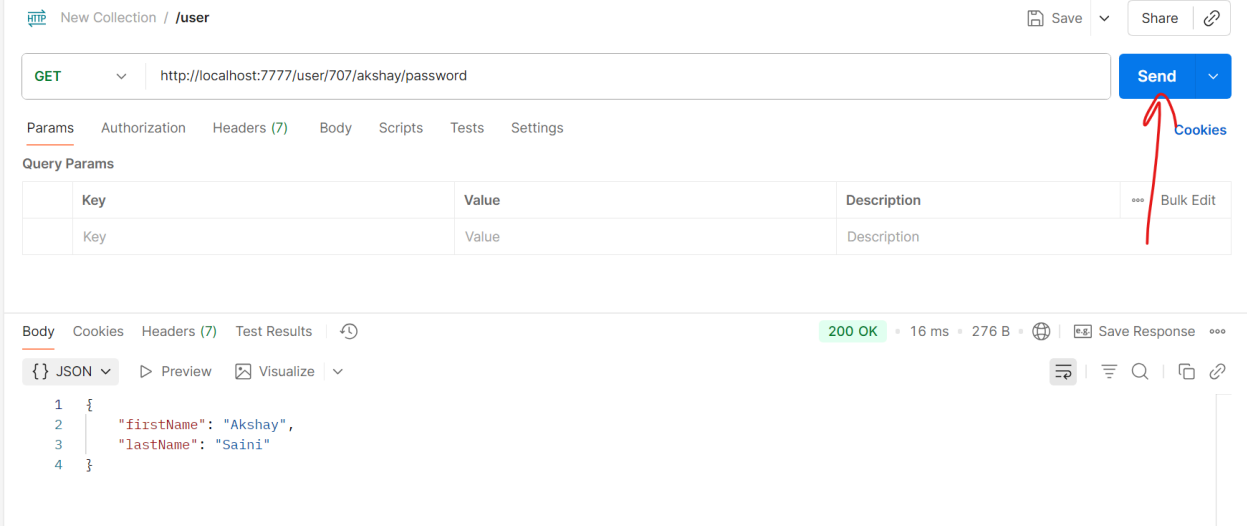
```
const express = require("express");
const app = express();

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");
});
```

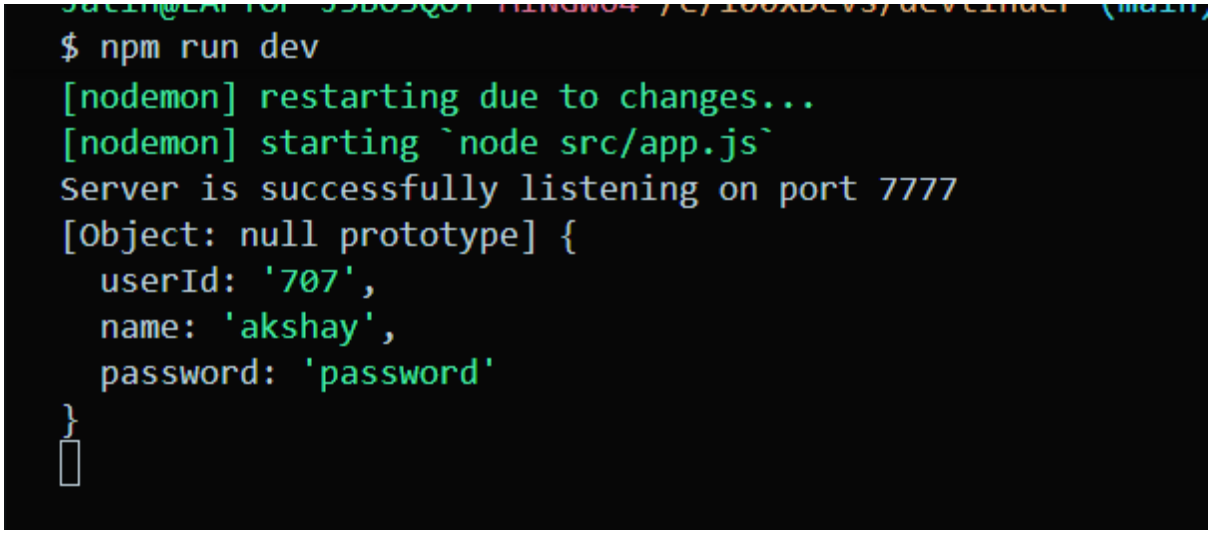


Now, when you make an API call and check the terminal, you'll see `{ userId: '707' }`.

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");
});
```





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**.