ExpressJS

ExpressJS is a minimal and flexible web application framework for Node.js that provides a robust set of features for building dynamic web applications and APIs. It acts as a layer on top of Node.js to make backend development easier, allowing you to handle HTTP requests, routing, middleware, and more in a structured and efficient way

Why should you use ExpressJS?

- Simplifies Server Development: Express streamlines building web servers and APIs, providing a simplified interface over Node.js's core HTTP module. This means you write much less code for tasks like routing, handling different HTTP methods, and sending responses.
- Middleware Support: It has a powerful middleware system. Middleware functions can handle authentication, logging, parsing request bodies, error handling, and other repetitive tasks, keeping your core code clean.
- Powerful Routing: Express offers a flexible and efficient routing mechanism, crucial for designing RESTful APIs and dynamic websites.
- Unopinionated & Flexible: Express does not impose any fixed structure or patterns, so you are free to organize your app as needed.
- Template Engine Support: It integrates easily with template engines like Pug, EJS, and Handlebars, useful for generating dynamic content on web pages.
- Large Ecosystem: Express has a vast collection of third-party plugins (middleware) and strong community support, which helps with extending its functionality and quick problem-solving.
- Used in Modern Stacks: It is a critical component in popular JavaScript technology stacks such as MEAN and MERN (MongoDB, Express, Angular/React, Node.js), used by companies like Netflix, IBM, eBay, and Uber

Why we are saying Express is Framework and React is Library. What is the difference between Framework and Library?

Express is called a framework and React a library due to the level of control and architectural support they provide. The core difference between a *framework* and a *library* is about control flow and inversion of control.

- In a library, you call its functions to achieve specific tasks—you are in control of how and when the library is used in your code.

- In a framework, you write parts of your application that are then called by the framework itself. Here, the framework is in control, dictating the flow and invoking your code as needed—this is called *inversion of control*.

Feature	**Library (React)**	**Framework (Express)**
Control	You control the flow	Framework controls the flow
Usage	You call it	It calls your code
Structure enforced?	No (flexible)	Yes (has conventions)
Purpose	Specific functionality	Full application skeleton
Example	React (UI), Lodash, Axios	Express (web server), Angular (front-end)

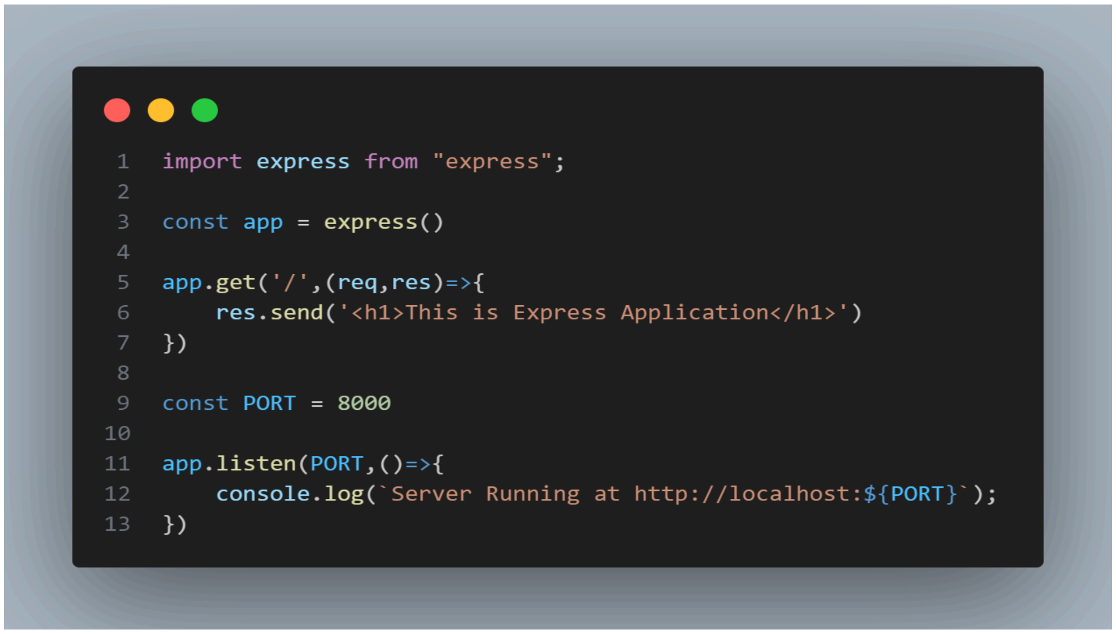
How to use or create Express applications?

-> Select or Create one Folder
-> Run command “npm init -y”
-> then run command “npm install express”
-> Go to package.json and edit “type: commonJS” to “type: module”
-> install nodemon using command “npm install nodemon - -save-dev”
-> then go to package.json in scripts instead of test give “ “start”:”node file_name.js” ” and “ “dev”:”nodemon file_name.js” ”
-> now we can use npm run dev to start our application

What is Nodemon?

Nodemon is a command-line tool that automatically restarts your Node.js application whenever it detects changes in your project files, making your development process more efficient by eliminating the need to manually stop and restart the server after every code modification

- It monitors your project directory for file changes and restarts the server automatically.
- It improves development workflow for Node.js and Express apps, allowing you to see changes instantly.
- It works without requiring changes to your application code—just run your app using nodemon instead of node

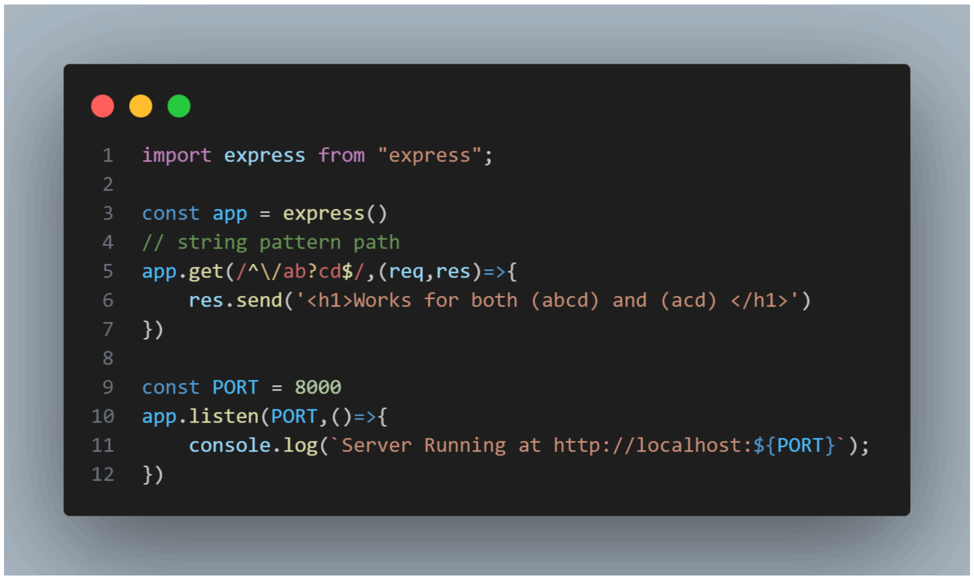


```javascript
import express from "express";

const app = express()

app.get('/',(req,res)=>{
    res.send('<h1>This is Express Application</h1>')
})

const PORT = 8000

app.listen(PORT,()=>{
    console.log(`Server Running at http://localhost:${PORT}`);
})
```

Above is a basic express code to create a Server.

<u>String Pattern Path</u>

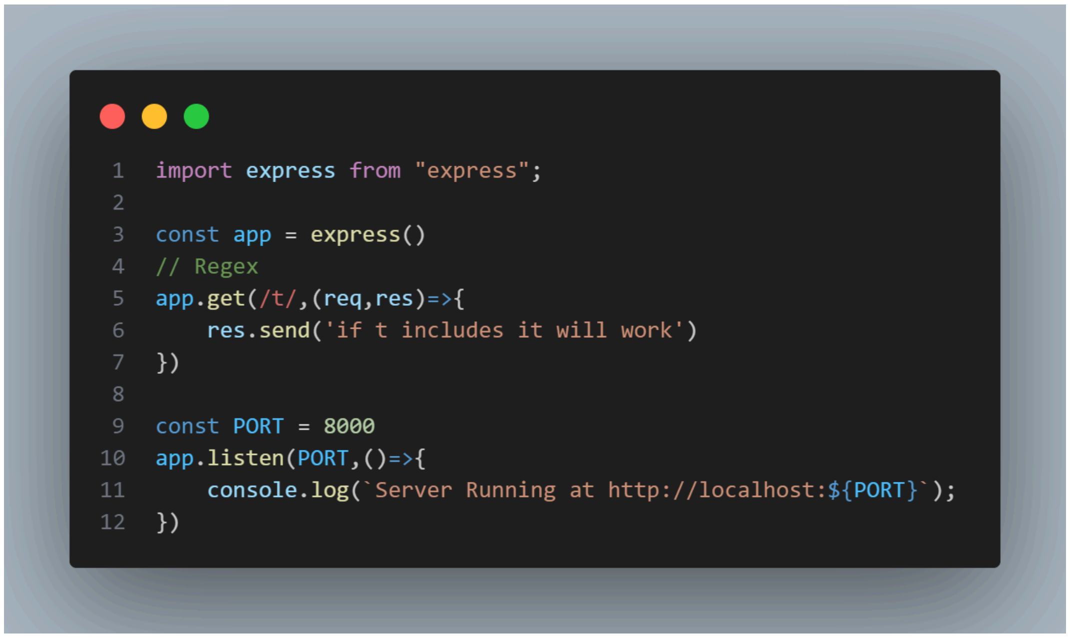
Optional Parameters



```javascript
import express from "express";

const app = express()
// string pattern path
app.get(/^\/ab?cd$/,(req,res)=>{
    res.send('<h1>Works for both (abcd) and (acd) </h1>')
})

const PORT = 8000
app.listen(PORT,()=>{
    console.log(`Server Running at http://localhost:${PORT}`);
})
```

`b?` means "the character `b` is optional"
Matches `/acd` and `/abcd`

Regex Route

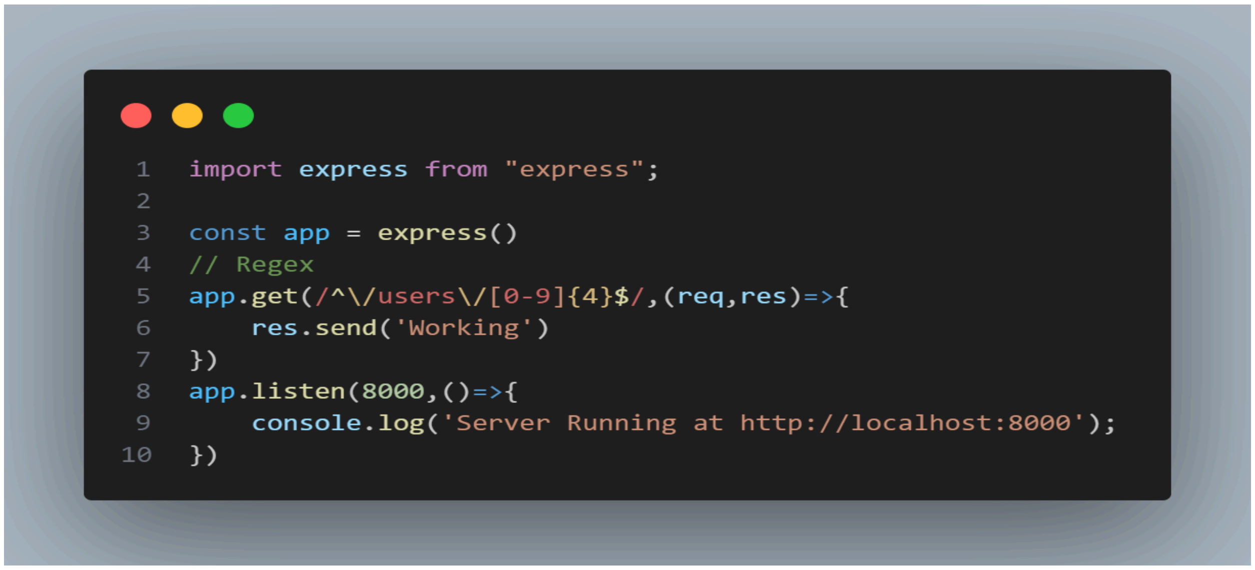


```
import express from "express";

const app = express()
// Regex
app.get(/t/,(req,res)=>{
    res.send('if t includes it will work')
})

const PORT = 8000
app.listen(PORT,()=>{
    console.log(`Server Running at http://localhost:${PORT}`);
})
```

This is a **regular expression**, not a string.
It matches any URL path that **includes the letter `t`**, anywhere in the path.
Examples that will match:
`/test, /task, /notebook, /team, /tea, /t`

Examples that **won't** match:
`/hello, /api, /123`

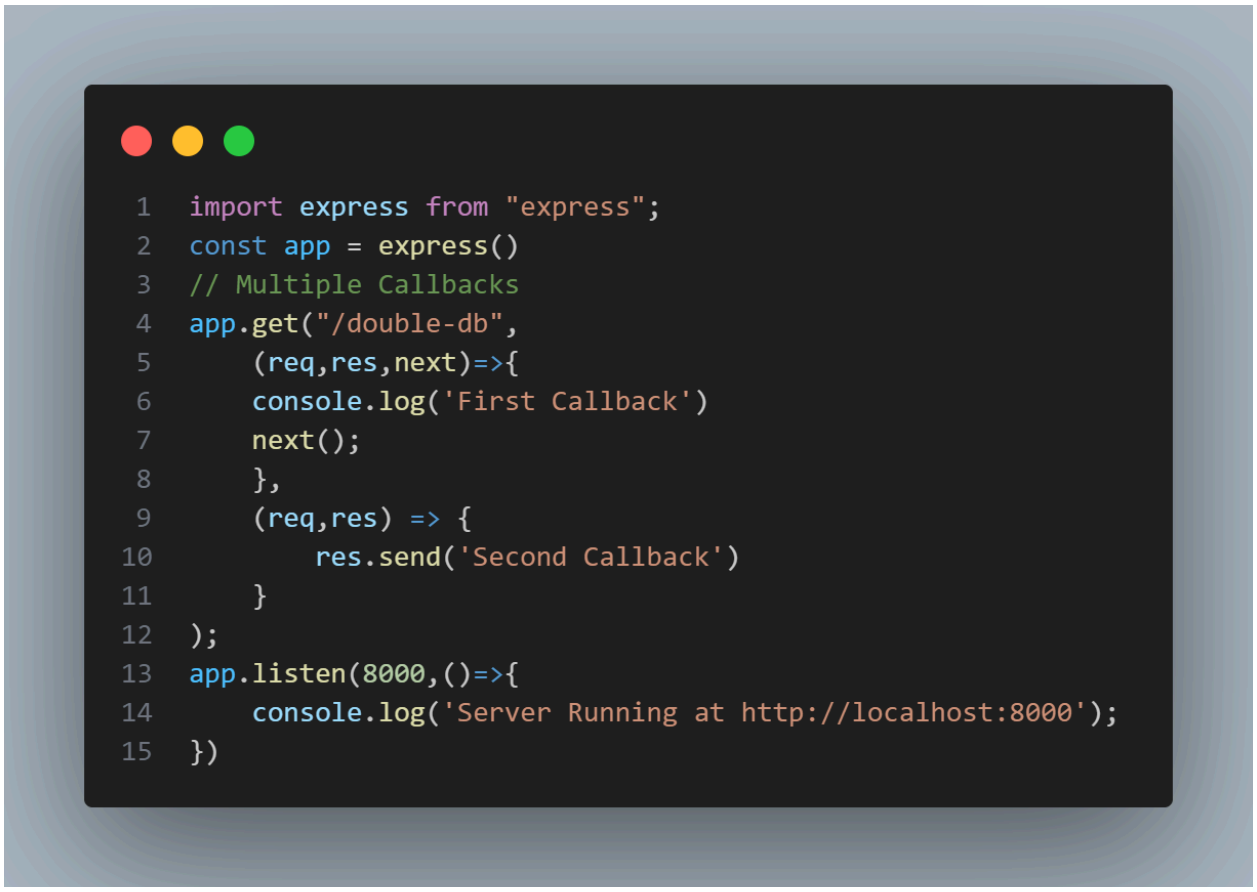
RegExp with Precise Control



```
import express from "express";

const app = express()
// Regex
app.get(/^\/users\/[0-9]{4}$/,(req,res)=>{
    res.send('Working')
})
app.listen(8000,()=>{
    console.log('Server Running at http://localhost:8000');
})
```

Part	Meaning
`^`	Start of the path
`/users/`	Literal match — path must start with `/users/`
`[0-9]{4}`	Exactly **4 digits** after `/users/`
`$`	End of the path — nothing allowed after the 4 digits

More than One Callback Function



```
import express from "express";
const app = express()
// Multiple Callbacks
app.get("/double-db",
    (req,res,next)=>{
    console.log('First Callback')
    next();
    },
    (req,res) => {
        res.send('Second Callback')
    }
);
app.listen(8000,()=>{
    console.log('Server Running at http://localhost:8000');
})
```

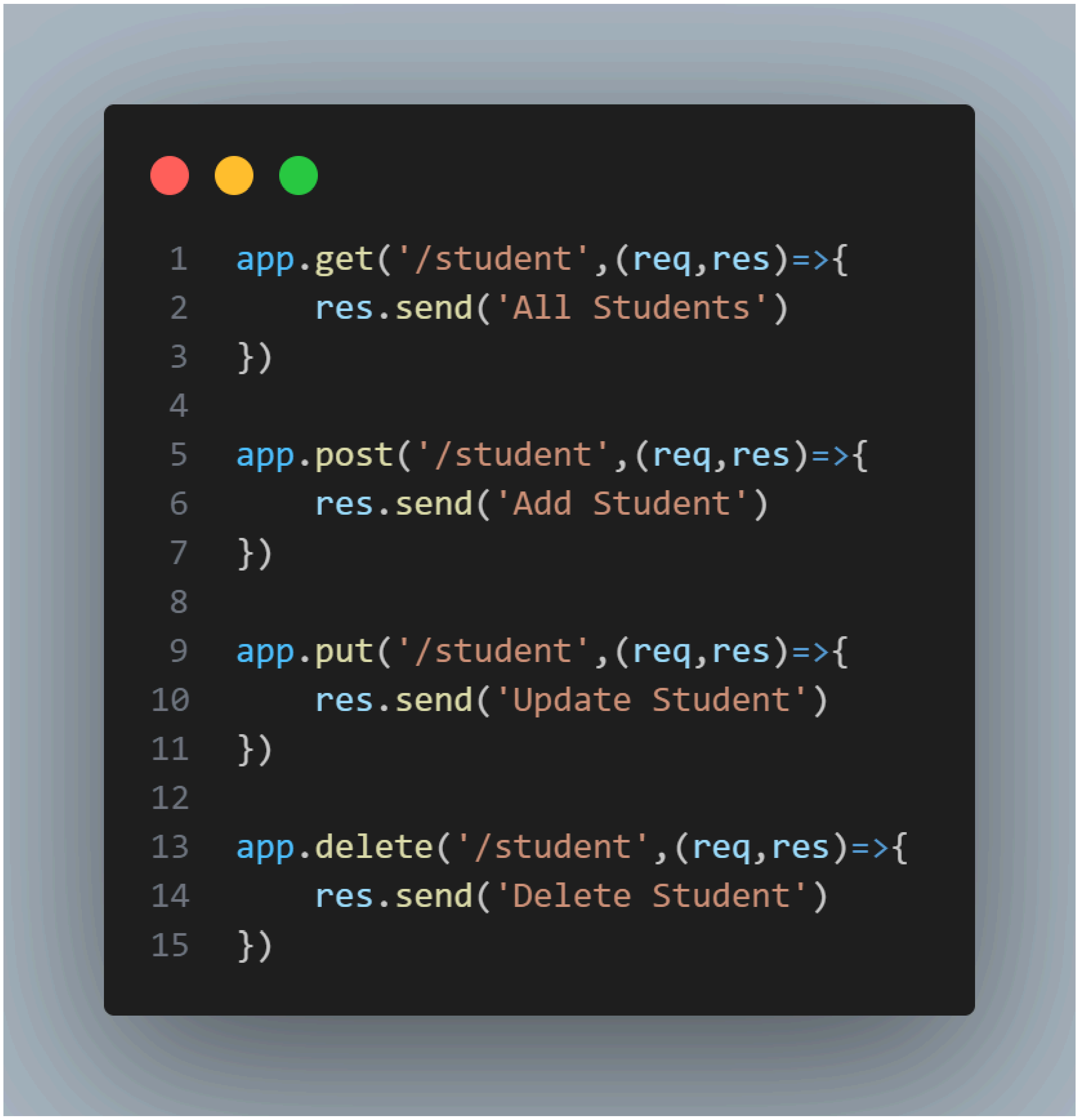
Why Use Multiple Callbacks?

This is the essence of **middleware in Express**.

Useful for: Logging, Authentication, Validation, Modularizing logic

App.Route() :-

Ugly Code:



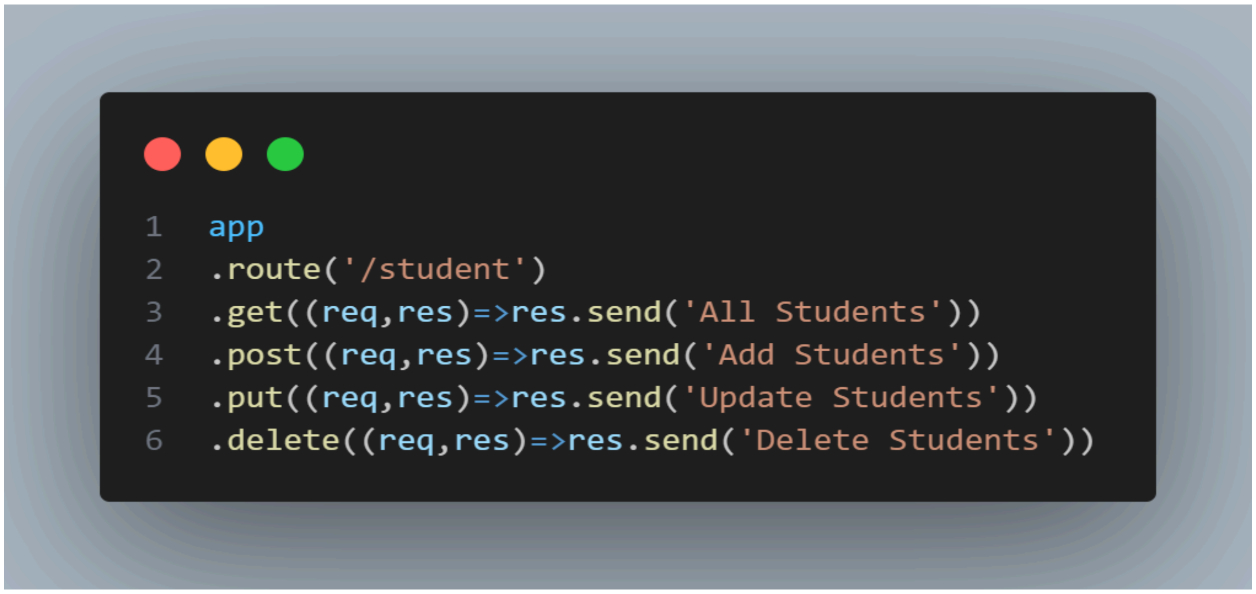
```
app.get('/student',(req,res)=>{
    res.send('All Students')
})

app.post('/student',(req,res)=>{
    res.send('Add Student')
})

app.put('/student',(req,res)=>{
    res.send('Update Student')
})

app.delete('/student',(req,res)=>{
    res.send('Delete Student')
})
```

Each HTTP method is handled **separately**.

Easy to read and beginner-friendly.

Good for small applications.

Good Code:



```
app
.route('/student')
.get((req,res)=>res.send('All Students'))
.post((req,res)=>res.send('Add Students'))
.put((req,res)=>res.send('Update Students'))
.delete((req,res)=>res.send('Delete Students'))
```

Uses `app.route()`, which allows chaining multiple methods for the **same route** (`/student`).
Cleaner and more organized, especially if you're handling many methods for a single endpoint.
Better for grouping related logic together.

Feature	First Code	Second Code (`.route()`)
Style	Individual route handlers	Chained route handlers
Readability	Simple for beginners	Cleaner for grouped logic
Use Case	Small apps or scattered logic	Grouped logic on same route

Router:-

In Express.js, the `router` method, specifically `express.Router()`, is a powerful tool for organizing and structuring your application's routing logic. It's a "mini-application" with its own set of routes and middleware, designed to be modular and reusable.

The `express.Router()` method creates a new `Router` object. This object acts like a complete middleware and routing system on its own. You define routes on this `Router` object using the same HTTP methods you would on the main `app` object (e.g., `router.get()`, `router.post()`, `router.put()`).

Once you've defined your routes and any specific middleware on the `Router` object, you can then "mount" it to a specific base path in your main Express application using `app.use()`.

Why should we use it?
1. Modularity and Code Organization

2. Encapsulation of Related Logic

3. Code Reusability

4.Scalability

Code:

```
import express from "express"
const router = express.Router()
router.get('/all',(req,res)=>{
      res.send('Get all student details')
})
router.post('/create',(req,res)=>{
    res.send('Create new Student ID')
})
router.put('/update',(req,res)=>{
    res.send('Update Student ID')
})
router.delete('/delete',(req,res)=>{
    res.send('Delete Student id from db')
})
export default router;
```

```
import express from "express";
import students from "./routes/student.js"

const app = express()

app.use('/students',students)
```

Route Parameters

Route parameters are named URL segments used to capture dynamic values from the URL. In Express.js, you define a route parameter by prefixing a colon (`:`) to a segment in the route path.

In your code: `app.get("/product/:category/:id", ...)`

- `:`category and `:`id are the route parameters.

When a user makes a request to a matching URL, Express parses the URL and extracts the values for these parameters. These values are then made available in the `req.params` object.

For example, if a user requests the URL `/product/electronics/123`, Express will do the following:

- It matches `electronics` to the `category` parameter.
- It matches `123` to the `id` parameter.

The `req.params` object will then look like this: `req.params = { category: 'electronics', id: '123' }`

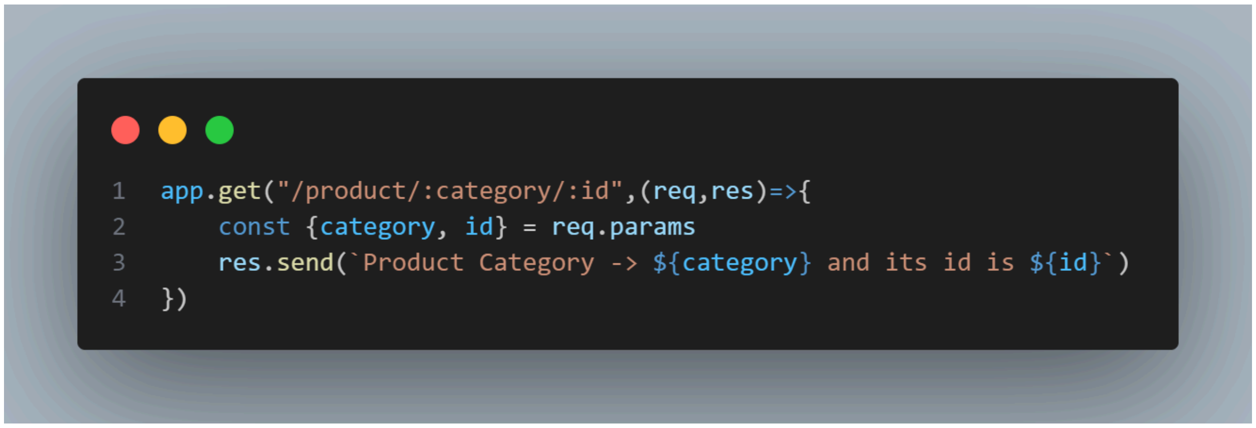
Your code then uses destructuring to get these values: `const {category, id} = req.params`

And sends a response using these dynamic values.

Advantages:-

1. Creating Dynamic and Flexible Routes
2. Building RESTful APIs
3. Encapsulation of Resource Identifiers
4. Simplicity and Readability
5. Passing Multiple Values Easily

Code:



```
app.get("/product/:category/:id",(req,res)=>{
    const {category, id} = req.params
    res.send(`Product Category -> ${category} and its id is ${id}`)
})
```

Route Parameters in Depth:

```
app.param("id",(req,res,next,id)=>{
    console.log(`id: ${id}`)
    next();
})

app.get("/user/:id",(req,res)=>{
    console.log("This is user id Path");
    res.send("Response Ok")
})
```

Query Parameters

In **Express.js**, **query parameters** are a way to pass data to the server through the URL. They appear after a question mark (`?`) in the URL and are typically used for filtering, sorting, or providing optional data in a GET request.

🔍 Example of Query Parameters in a URL:

`https://example.com/products?category=books&sort=price`

- `category=books` and `sort=price` are **query parameters**.
- These are **not part of the route path**, but extra information sent with the request.

🧠 Accessing Query Parameters in Express

In Express.js, you can access query parameters using:

```
req.query
```

It returns an object containing all query parameters.

```
import express from "express";
const app = express()

app.get('/login',(req,res)=>{
    const {username, password} = req.query
    res.send(`Username: ${username} && Password: ${password}`)
})

export default app;
```

Sending JSON

```
// Send JSON in Response
import express from "express";

const app = express()

app.get('/user/product/:category',(req,res)=>{
    const {category} = req.params
    const {quantity} = req.query
    const user = {
        name: "Ravi Shankar",
        id: 21,
        city: "Bhuvaneswar",
        category: category,
        quantity: quantity
    }
    res.json(user)
})

export default app;
```

This code sets up a small web server using Express. When someone visits the route `/user/product/:category`, it grabs two pieces of data:

1. `category` – from the URL path (like `/user/product/books`)

2. `quantity` – from the query string (like `?quantity=10`)

It then creates a user object with some fixed info (name, id, city), and adds the `category` and `quantity` values from the request. Finally, it sends that whole user object back as a **JSON response**.

So if someone goes to:

`/user/product/laptop?quantity=3`

They'll get a response like:

```
{
  "name": "Ravi Shankar",
  "id": 21,
  "city": "Bhuvaneswar",
  "category": "laptop",
  "quantity": "3"
}
```

At the end, `app` is exported so it can be used elsewhere, like in a main server file.

Here Sending JSON means
Sending JSON means sending data in a format that looks like a JavaScript object.
It's commonly used to send structured data between a server and a client.
In Express.js, `res.json()` is used to send this kind of data in the response.
Clients (like browsers or apps) can easily read and use the JSON data.