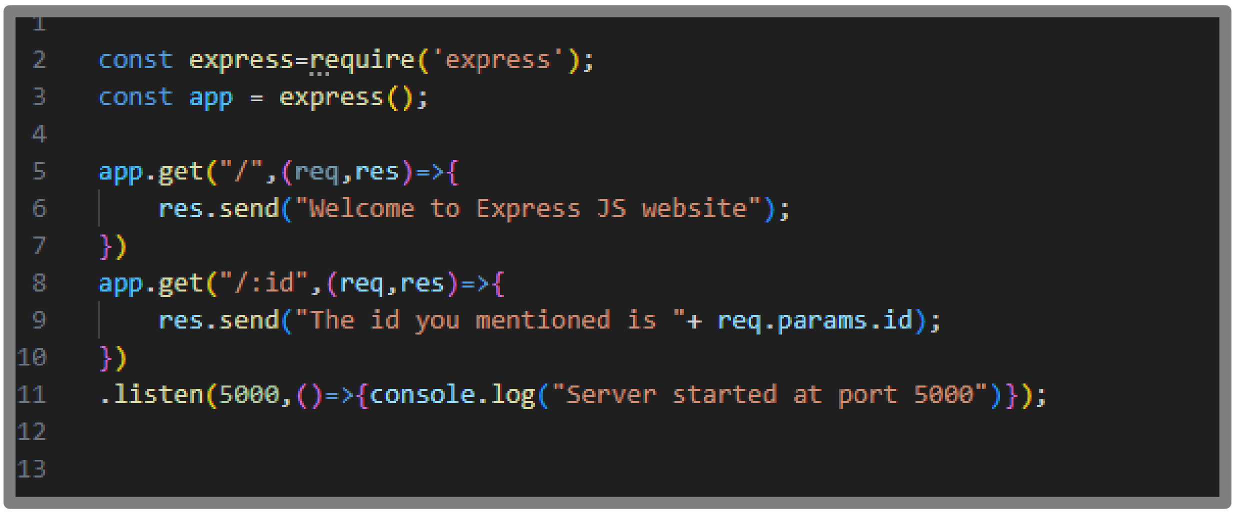
1)Express JS URL Building
2)Create REST API
3)Testing REST API in Postman

Express JS URL Building

URL building in Express.js refers to the process of constructing URLs dynamically within your application code.

This is important for generating links to different routes or resources within your application, especially when those links depend on dynamic data or parameters.

Route Parameters: Express allows you to define routes with parameters, which can then be used to generate dynamic URLs.
For example:



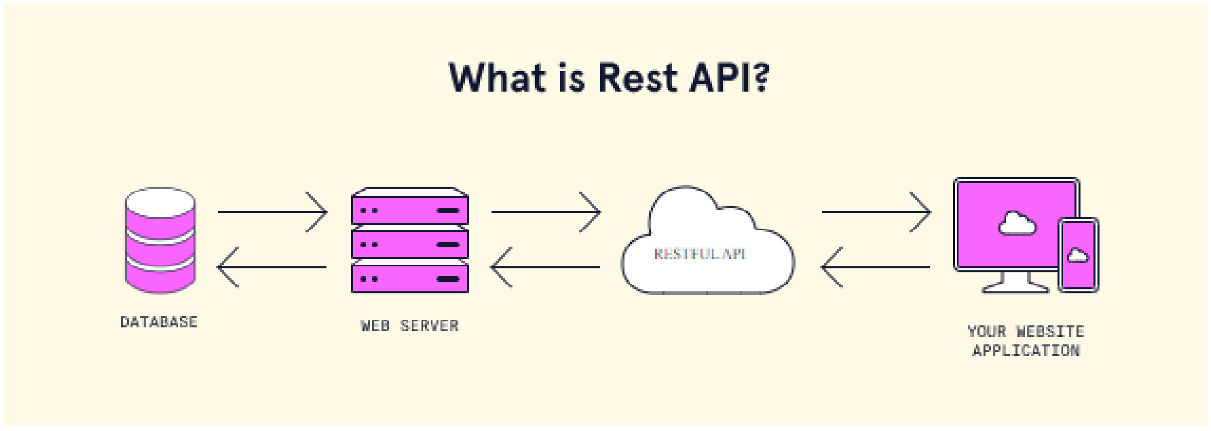
```

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

app.get("/",(req,res)=>{
    res.send("Welcome to Express JS website");
})
app.get("/:id",(req,res)=>{
    res.send("The id you mentioned is "+ req.params.id);
})
.listen(5000,()=>{console.log("Server started at port 5000")});
```

Create REST API

An **API** stands for **Application Programming Interface**. It acts as a messenger between different software programs, allowing them to talk to each other and exchange data.

A **REST API**, short for **Representational State Transfer Application Programming Interface**, is a standardized way for applications to communicate with each other over the internet. Imagine it like a waiter taking your order in a restaurant:

- **You (the client):** Use the API to send requests to the server (the restaurant).
- **The API (the menu):** Defines the types of data (resources) available (like food items) and the actions you can take (like ordering, modifying, or canceling).
- **The server (the restaurant):** Processes your requests and sends back responses (like your food or an error message).



In simple words, A RESTful API is an application programming interface (API) that uses HTTP requests to access and use data and was created by computer scientist Roy Fielding as a guideline to manage communication on a complex network like the internet

Principles that serve as guidelines for designing well-structured and user-friendly REST APIs.

Separation of Client and Server: In the REST architectural style, the implementation of the client and the implementation of the server can be done independently without each knowing about the other. This means that the code on the client side can be changed at any time without affecting the operation of the server, and the code on the server side can be changed without affecting the operation of the client.

State representations: The state of resources is typically represented in JSON or XML format, making data exchange interoperable across various platforms.

Standardized methods: Common HTTP methods like **GET, POST, PUT, PATCH, and DELETE** are used for CRUD (Create, Read, Update, Delete) operations on resources. This consistency allows clients to intuitively interact with the API.

```
const express=require('express');
const app = express();
const data = require("./MOCK_DATA.json")
const bodyparser = require("body-parser");
const PORT = 8000;
app.use(bodyparser.json())

//get data
app.get('/users',(req,res)=>{
  return res.json(data);
})
//get data with particular id
app.get('/users/:id',(req,res)=>{
    const id = Number(req.params.id);
    const user = data.find((user)=>user.id === id);
    return res.json(user);
  })

  //add new user to the list of users
  app.post('/users',(req,res)=>{
    const newUser = req.body;

    //validation - required properties are present and valid
    if( !newUser.first_name || !newUser.last_name || !newUser.email ||
!newUser.gender || !newUser.username){
        return res.status(400).json({message:"Missing data"})
    }
    //generate new  unique id for this user
    newUser.id = data.length+1;
    //push new user into array
    data.push(newUser);
    res.status(201).json(newUser);
});
app.patch("/users/:id",(req,res)=>{
    const id = Number(req.params.id);
    const update = req.body;
    const userIndex = data.findIndex((user)=>user.id === id);
    //check if the users exits
    if (userIndex === -1){
        return res.status(404).json({message:"Invalid ID"});
    }
    Object.assign(data[userIndex],update);
    res.status(200).json(data[userIndex]);

});
app.delete('/users/:id',(req,res) =>{
    const id = Number(req.params.id);
    const index = data.findIndex((item)=> item.id===id);
    if(index !==-1){
      data.splice(index,1);
      res.status(200).json("Deleted Successfully");
    }else {
      res.status(400).json('ID not found');
    }
});

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

Testing REST API in Postman

Postman is a popular collaboration platform for API development. It allows developers to design, test, and document APIs quickly and efficiently.

It supports various HTTP methods, authentication mechanisms, and request/response formats.

Install Postman: Download and install Postman from https://www.postman.com/.

Building Requests:

1. **Create a New Request:** Click the "New Request" button and enter the base URL of the API you want to test.
2. **Choose the HTTP Method:** Select the appropriate HTTP method (GET, POST, PUT, DELETE) based on the desired operation.
3. **Define the Resource Path:** Specify the specific resource you want to access, following the API's endpoint structure. For example, `/users` or `/products/123`.
4. **Set Headers (Optional):** Add any required headers for authentication, content type, or other specific functionalities.
5. **Compose Request Body (Optional):** For methods like POST or PUT, provide the data you want to send in the request body. You can use formats like JSON, form-data, or raw text.

6. **Send the Request:** Click the "Send" button to execute your request and receive a response from the server.

THANK YOU

Presented by –

Vidushi Garg