UNIT - 6

NodeJs

Introduction to Node.js

What is Node.js?

•Node.js is an **open-source, cross-platform JavaScript runtime environment** that allows developers to execute JavaScript code outside the browser. It is built on **Google Chrome's V8 JavaScript engine** and is designed for building scalable and high-performance network applications.

Features of Node.js:

- **Asynchronous and Event-Driven**: Uses non-blocking I/O operations, making it fast and efficient.
- **Single-Threaded but Highly Scalable**: Uses an event loop to handle multiple requests efficiently.
- **Fast Execution**: Built on the V8 engine, which compiles JavaScript directly to machine code.
- **Cross-Platform Compatibility**: Runs on Windows, macOS, and Linux.
- **Rich Package Ecosystem**: Uses NPM (Node Package Manager) with thousands of open-source libraries.
- **Micro-services and RESTful APIs**: Ideal for developing backend APIs.

Applications of Node.js:

- Web applications (Express.js, Next.js)
- Real-time chat applications (Socket.io)
- Streaming services (Netflix, YouTube, Twitch)
- APIs and micro-services (RESTful and GraphQL APIs)
- IoT applications (Raspberry Pi, Arduino)
- Serverless computing (AWS Lambda, Azure Functions)

Environment Setup

To work with Node.js, you need to set up the environment on your system.

Step 1: Download and Install Node.js

1. Go to the Node.js official website.
2. Download the **LTS** (Long-Term Support) version for stability.
3. Install Node.js following the setup instructions.

Step 2: Verify Installation

•Open a terminal or command prompt and run:

```
node –v
npm –v
```

This will display the installed Node.js and npm (Node Package Manager) versions.

Contd.

Step 3: Setting Up a Project

1.Create a new project folder:

```
mkdir my-node-app
cd my-node-app
```

2. Initialize a Node.js project:

```
npm init –y
```

This generates a **package.json** file containing project details and dependencies.

First Node.js Application

Create a simple **Hello World** program in Node.js.

1. Create a file **app.js**:

```
console.log("Hello, Node.js!");
```

2. Run the application:

```
node app.js
```

Output:

Hello, Node.js!

Creating Node.js Application

To create a "Hello, World!" web application using Node.js, you need the following three important components −

- **Import required modules** − We use the require directive to load Node.js modules.
- **Create server** − A server which will listen to client's requests similar to Apache HTTP Server.
- **Read request and return response** − The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

Contd.

Step 1 - Import Required Module

•We use the require directive to load the http module and store the returned HTTP instance into an http variable as follows −

```
var http = require("http");
```

Step 2 - Create Server

•We use the created http instance and call http.createServer() method to create a server instance and then we bind it at port 3000 using the listen method associated with the server instance. Pass it a function with parameters request and response.

Contd.

- The **createserver()** method has the following syntax −

```
http.createServer(requestListener);
```

- The requestlistener parameter is a function that executes whenever the server receives a request from the client. This function processes the incoming request and forms a server reponse.
- The requestlistener function takes request HTTP request and response objects from Node.js runtime, and returns a ServerResponse object.

Contd.

```
listener = function (request, response) {
   // Send the HTTP header
   // HTTP Status: 200 : OK
   // Content Type: text/plain
   response.writeHead(200, {'Content-Type': 'text/html'});

   // Send the response body as "Hello World"
   response.end('<h2 style="text-align: center;">Hello World</h2>');
};
```

Contd.

- The above function adds the status code and content-type headers to the ServerResponse, and Hello World message.
- This function is used as a parameter to createserver() method. The server is made to listen for the incoming request at a particular port (let us assign 3000 as the port).

Step 3 - Testing Request & Response

- Write the sample implementation to always return "Hello World". Save the following script as **hello.js.**

Contd.

```
http = require('node:http');
listener = function (request, response) {
    // Send the HTTP header
    // HTTP Status: 200 : OK
    // Content Type: text/html
    response.writeHead(200, {'Content-Type': 'text/html'});

    // Send the response body as "Hello World"
    response.end('<h2 style="text-align: center;">Hello World</h2>');
};

server = http.createServer(listener);
server.listen(3000);

// Console will print the message

console.log('Server running at http://127.0.0.1:3000/');
```

Contd.

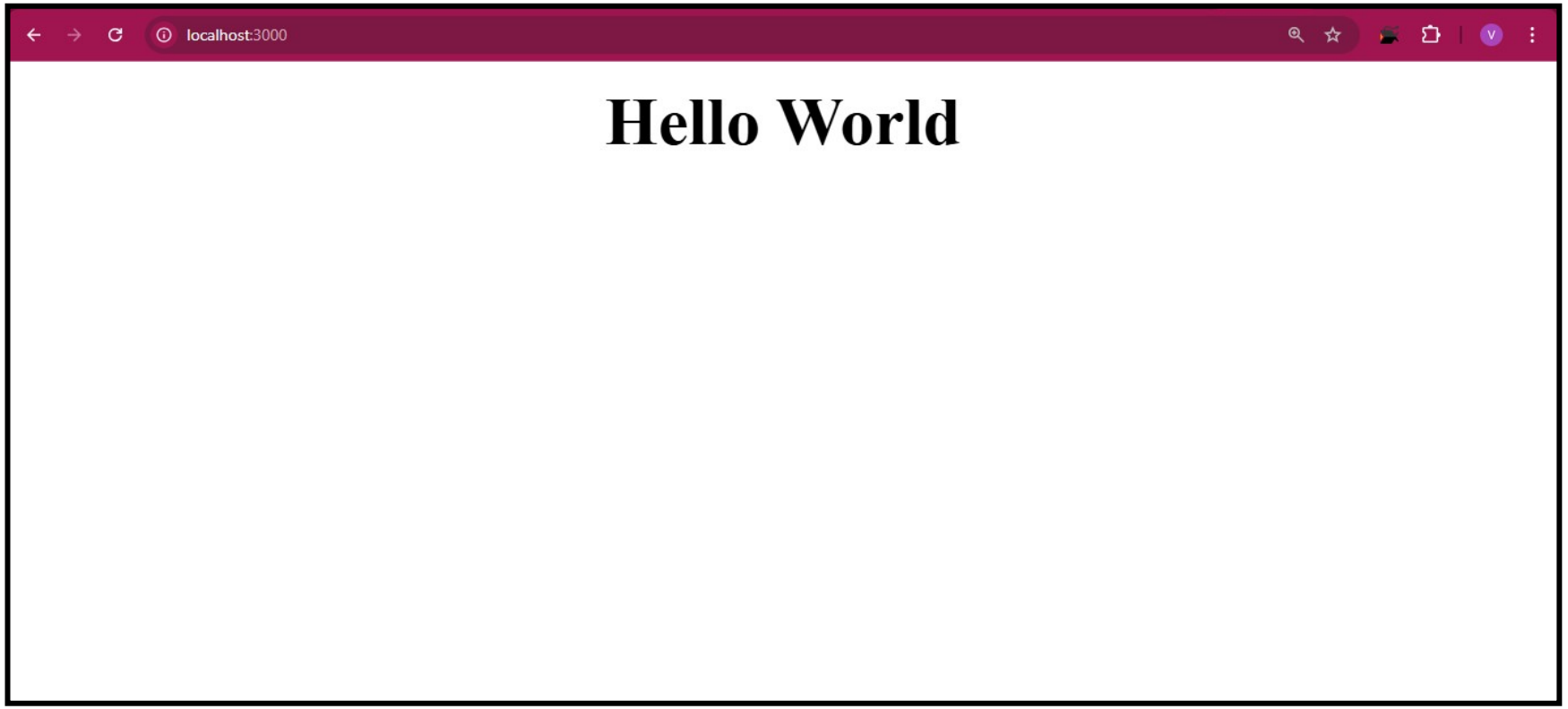
In the PowerShell terminal, enter the following command.

```
PS C:\Users\Jagadeesh\my-node-app> node index.js
Server running at http://127.0.0.1:3000/
```

•The program starts the Node.js server on the localhost, and goes in the listen mode at port 3000. Now open a browser, and enter http://127.0.0.1:3000/ as the URL. The browser displays the Hello World message as desired.

Output

After that we can open the browser type localhost:3000/ and press enter the output will shown:



NodeJS REPL (READ, EVAL, PRINT, LOOP)

NodeJS REPL (Read-Eval-Print Loop) is an interactive shell that allows you to execute JavaScript code line-by-line and see immediate results. This tool is extremely useful for quick testing, debugging, and learning, providing a sandbox where you can experiment with JavaScript code in a NodeJS environment.

Note: **sandbox** refers to a controlled environment where you can safely execute JavaScript code without affecting the main system or application.

What is REPL?

- REPL is like a JavaScript playground in your terminal. If you type the code, REPL runs it, shows you the result, and then waits for your next command. It’s a loop:
- **READ:** You type some JavaScript code into the terminal, and REPL reads what you typed.
- **EVAL:** REPL runs (evaluates) your code.
- **PRINT**: REPL shows you the result of your code.
- **LOOP**: REPL goes back to step 1, waiting for you to type more code. This loop continues until you quit REPL.

Getting Started with REPL

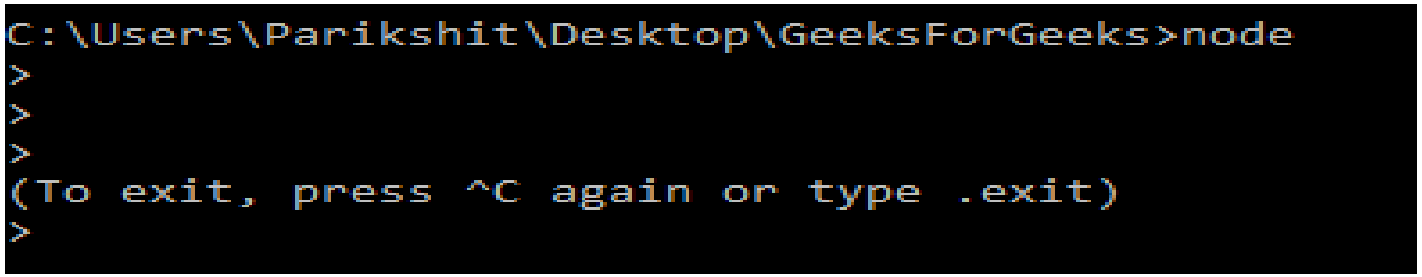
To start working with the REPL environment of NodeJS, follow one of these two methods:

Starting REPL in the Terminal or Command Prompt

•Open your terminal (for UNIX/Linux) or Command Prompt (for Windows).

•Type **node** and press '**Enter**' to start the REPL.

Note: Us **node** e the c



open node repl

Key Features of NodeJS REPL

Executing JavaScript Code

The REPL is a full-featured JavaScript environment, meaning you can run any valid **JavaScript** code inside it.

Example:

```
> const x = 10;
> const y = 20;
> x + y
30
```

•You can declare variables, create functions, and run any code that would work in a regular JavaScript runtime.

Contd.

Multi-Line Input

In case of complex logic (like loops or functions), the REPL supports multi-line input. When you enter a block of code, the REPL will continue reading input until the block is complete.

Example:

```
>function add(a, b) {
...   return a + b;... }
>add(50, 10)
60
```

•Here, the REPL waits for you to complete the function block before evaluating the code.

Contd.

Underscore (_) Variable

The REPL provides a special variable _ (**underscore**) that stores the result of the last evaluated expression.

Example:

```
>2 + 2
4
>_ * 5
20
```

•In this case, the result of 2 + 2 is stored in _, which is then used in the next line to calculate 20.

Built-in REPL Commands

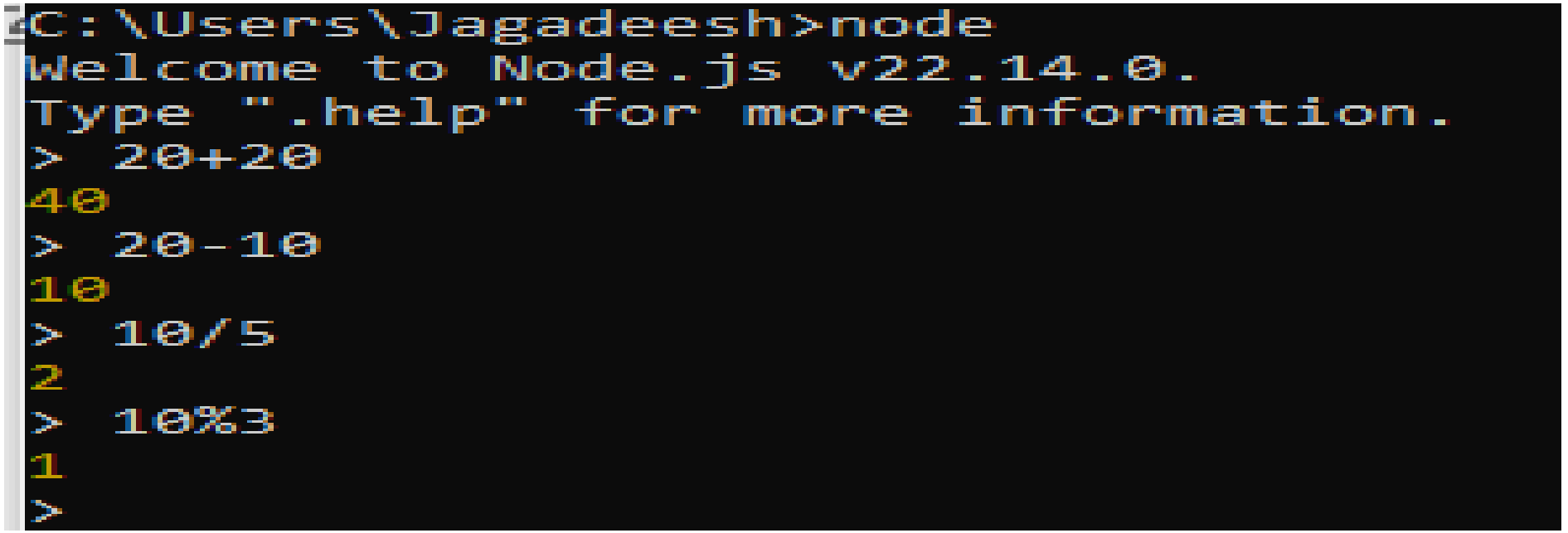
NodeJS REPL provides several built-in commands (REPL commands always start with a dot .).

- **.help:** Displays a list of all available REPL commands.
- **.break:** Breaks out of multi-line input or clears the current input.
- **.clear:** Resets the REPL context by clearing all declared variables.
- **.exit:** Exits the REPL session.

Contd.

Arithmetical operations in REPL:

The arithmetical operations in REPL are:



Contd.

Using variables in REPL

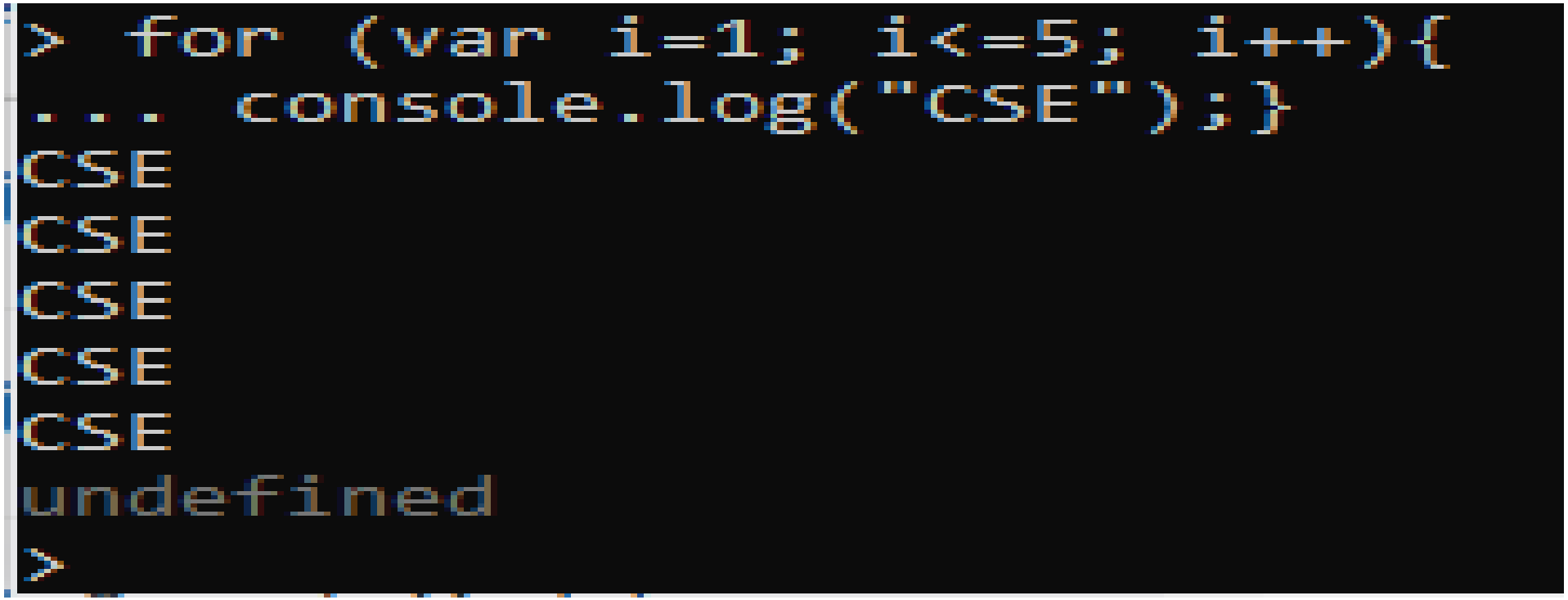
- The keyword **var** is used to assign values to variables.

```
> var a=10;
undefined
> a
10
> var c="Hello World";
undefined
> c
'Hello World'
>
```

Contd.

Using loops in REPL

- Loops can be used in REPL as in other editors.



Contd.

.help is used to list out all the commands.

```
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the REPL
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press Ctrl+C to abort current expression, Ctrl+D to exit the REPL
>
```

Node Package Manager (NPM)

A **Node Package Manager (NPM)** is a package manager specifically designed for **JavaScript** and **Node.js** applications. It helps manage libraries, frameworks, and dependencies in Node.js projects.

- It comes **automatically installed with Node.js.**
- It has over **1 million free packages** to speed up development.
- You can **install, update**, and **remove** packages easily.

Types of Node Package Managers

1. npm (Node Package Manager)

- Default package manager for Node.js.
- Comes pre-installed with Node.js.
- Example usage:

  npm install express

2. yarn (Yet Another Resource Negotiator)

- Developed by Facebook.
- Better performance and caching.
- Example usage:

  yarn add express

3. pnpm (Performant NPM)

- Stores packages in a global cache.
- Saves space and speeds up installations.
- **Example usage:**

  pnpm install express

Basic NPM Commands

- **Check installed version:**

```
npm -v
```

- Install a package locally:

```
npm install express
```

- Install a package globally:

```
npm install -g nodemon
```

- Uninstall a package:

```
npm uninstall express
```

- List installed packages:

```
npm list
```

NodeJS HTTP Module

The NodeJS HTTP module allows you to create web servers and handle HTTP requests and responses, making it a fundamental part of building web applications in NodeJS. HTTP is the built-in module in NodeJs through which the data is transferred.

- Uses the require() method to include the HTTP module.
- It provides utilities to create both client and server applications.
- Supports various HTTP methods like GET, POST, PUT, DELETE, etc.
- Allows ease in handling request headers, query parameters, and response bodies.

Modules in Node.js

- Node.js modules are reusable blocks of code that help in structuring applications efficiently. They enable encapsulation, code reuse, and maintainability. Each module in Node.js has its own scope, meaning variables and functions inside a module are private unless explicitly exported.
- In Node.js, modules can be categorized into different types based on their origin and functionality.

Types of Node.js Modules

1. Core Modules (Built-in Modules)
2. Local Modules (User-defined Modules)
3. Third-party Modules (npm Modules)

1. Core Modules (Built-in Modules)

- These are pre-installed with Node.js and provide essential functionalities such as file system operations, networking, and utilities.

- No need to install them separately.

Contd.

Example:

```
const fs = require('fs'); // File System module
const http = require('http'); // HTTP module
```

Some common core modules:

- fs (File System) - Handling file operations
- http - Creating HTTP servers
- path - Working with file paths
- os - Getting OS-related info
- crypto - Implementing cryptographic functions

Contd.

2. Local Modules (User-defined Modules)

- Created by developers for specific application requirements.
- These can be exported and used in other files.

Example:

Create math.js:

```
function add(a, b) {
    return a + b;
}
module.exports = add;
```

Import and use it in another file (app.js):

```
const add = require('./math');
console.log(add(5, 3)); // Output: 8
```

Contd.

3. Third-party Modules (npm Modules)

•Installed via npm (Node Package Manager).

•Useful for extending functionality (e.g., Express.js for web servers, Mongoose for MongoDB).

Example:

Install a module:

```
npm install express
```

Use it in a file:

```
const express = require('express');
const app = express();
app.get('/', (req, res) => res.send('Hello World!'));
app.listen(3000, () => console.log('Server running on port 3000'));
```

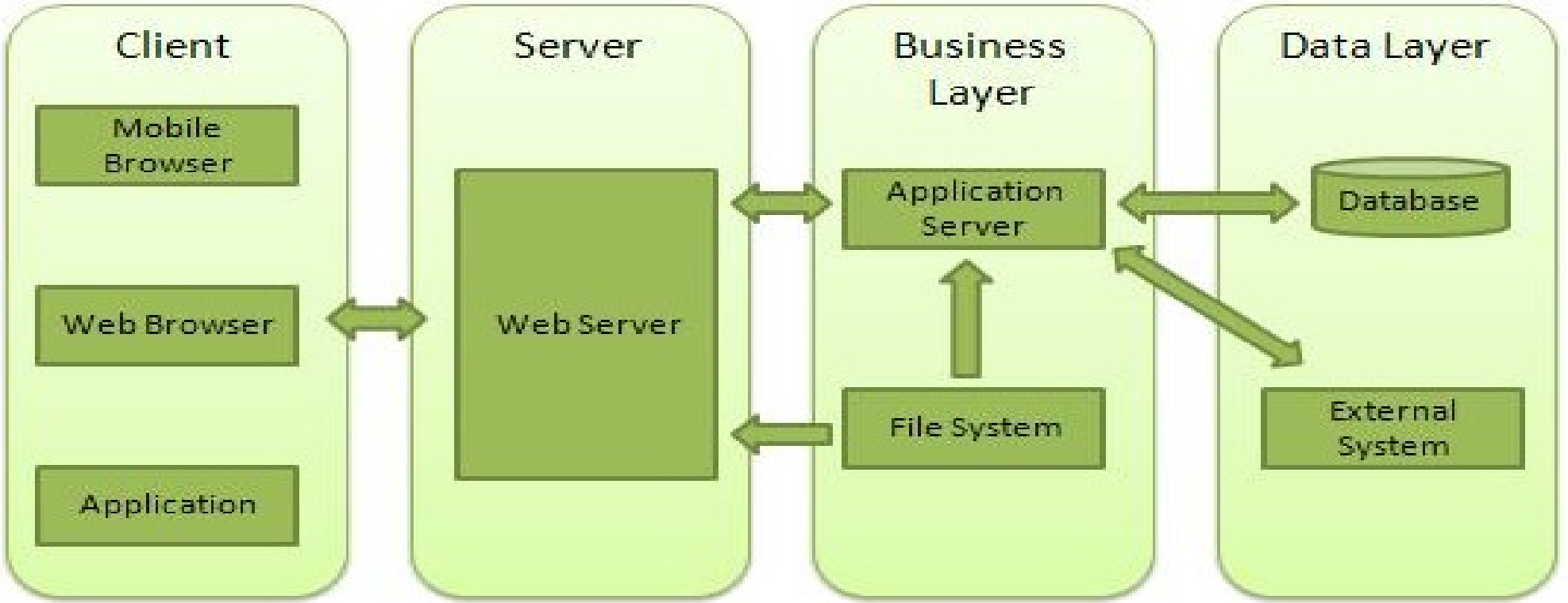
Node.js - Web Module

What is a Web Server?

- The http module in Node.js enables data transfer between the server and client over the Hyper Text Transfer Protocol (HTTP). The createServer() function in http module creates an instance of Node.js http server. It listens to the incoming requests from other http clients over the designated host and port.
- The Node.js server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver html documents along with images, style sheets, and scripts.

Web Application Architecture

- A Web application is usually divided into four layers –



Contd.

- **Client** − This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.
- **Server** − This layer has the Web server which can intercept the requests made by the clients and pass them the response.
- **Business** − This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.
- **Data** − This layer contains the databases or any other source of data.

Creating a Simple Web Server

```
const http = require('http'); // Import the HTTP module
// Create a server
const server = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/plain' }); //setting Headers
    res.end('Hello, Welcome to Node.js Web Module!');
});
const PORT = 3000;
server.listen(PORT, () => {
    console.log(`Server running at http://localhost:${PORT}/`);
});
```

Handling Multiple Routes

- We can serve **different pages** based on the requested URL.

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

const server = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html' });

    if (req.url === '/') {
        res.end('<h1>Welcome to the Home Page</h1>');
    } else if (req.url === '/about') {
        res.end('<h1>About Us</h1><p>This is a Node.js web module example.</p>');
    } else {
        res.writeHead(404);
        res.end('<h1>404 - Page Not Found</h1>');
    }
});
server.listen(3000, () => {
    console.log('Server running at http://localhost:3000/');
});
```

What is Express.js?

- Express.js is a minimal and fast web framework for Node.js.
- Simplifies server-side web application development.
- Provides routing, middleware, and API handling features.
- Used to create RESTful APIs and dynamic web applications.

Key Features of Express.js

- Middleware support for request processing.
- Routing system to handle different HTTP requests.
- Template engine support (EJS, Pug, Handlebars).
- Integration with databases like MongoDB and MySQL.
- Error handling and simplified request handling.

Simple Express.js Example

```
const express = require('express');
const app = express();
// Define a basic route
app.get('/', (req, res) => {
    res.send('Hello, World!');
});
// Start the server
app.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

Simple Express.js Routing Example

```
const express = require('express')
const app =express()
const port = 3002;
app.get('/fruit',(req,res)=>{
    res.end("Apple is a fruit")
})
app.get('/products',(req,res)=>{
    res.end("Product lists")
})
app.listen(port, ()=>{
    console.log("server is running:")
})
```

Middleware in Express.js

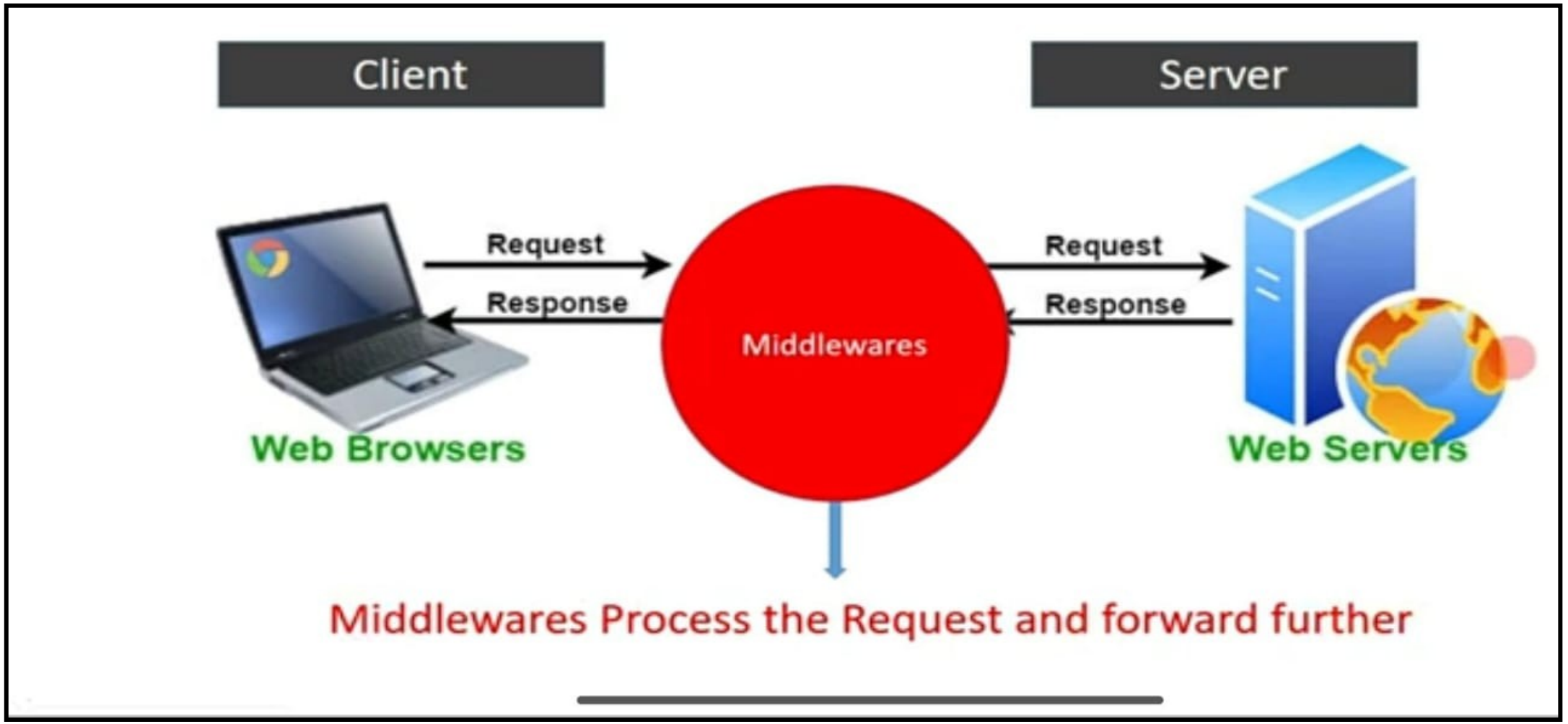
What is Middleware?

Middleware in Express.js is a function that executes between the request and response cycle. It helps process requests before reaching the final route handler.

Key Roles of Middleware:

- Modify Request & Response Objects
- Handle Authentication & Authorization
- Log Requests for Debugging
- Process Data (e.g., Parsing JSON, Form Data)
- Handle Errors Efficiently

Middlewares



Middleware Handling & Execution Flow in Express.js

```
const express = require('express')
const app =express()
const port = 3009;
const firstHandler = ((req, res, next) =>{
    if(10<20){
        next()
    }
})
```

Contd.

```
const secondtHandler = ((req, res, next) =>{
    if(10>20){
        next()
    } else {
        console.log("Sorry you are not allowed")
    }
})
const thirdtHandler = ((req, res, next) =>{
    if(30<40){
        next()
    }
})
```

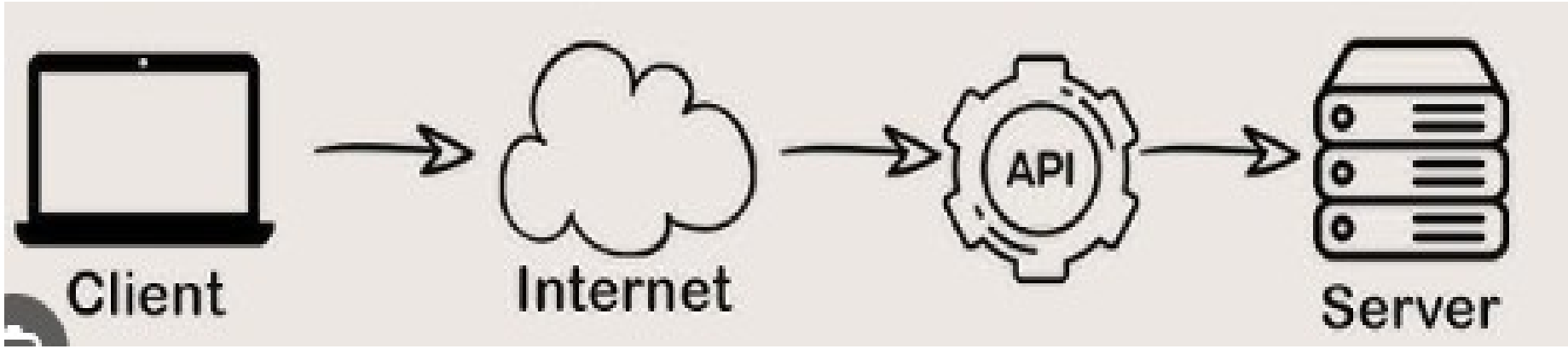
Contd.

```
app.get('/fruit', firstHandler, (req,res)=>{
    res.end("Apple is a fruit")
})
app.get('/products', secondtHandler, (req,res)=>{
    res.end("Product lists")
})
app.get('/user/:105', thirdtHandler, (req, res)=>{
    res.send("The user serached for 105")
})
app.listen(port, ()=>{
    console.log("server is running:")
})
```

API [Application Programming Interface]

An **API** (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate and interact with each other, enabling them to exchange data, features, and functionalities.

•An API is a messenger that delivers your request to a system and returns its response.



Types of Web APIs

There are different ways that APIs can work depending on when and why they were created.

SOAP APIs:

These APIs use simple Object Access Protocol. Client and Server exchange messages using XML.

RPC APIs:

These APIs are called Remote Procedure Calls. The client completes a function for procedural on the server and the server sends the output back to the client.

REST APIs:

These are the most popular and flexible APIs found on the web today. The client sends requests to the server as data. The server uses this client input to start internal functions and returns output data back to the client.

RESTful API

What is a RESTful API?

- A **RESTful API (REpresentational State Transfer API)** is a web service that follows REST principles, allowing clients to interact with a server using standard HTTP methods.
- REST dominates modern web APIs due to simplicity and scalability

Key Principles of RESTful APIs:

- **Stateless:** Each request from a client must contain all the information needed for processing.
- **Client-Server Architecture:** The client and server communicate independently.

Use of HTTP Methods:

GET → Retrieve data

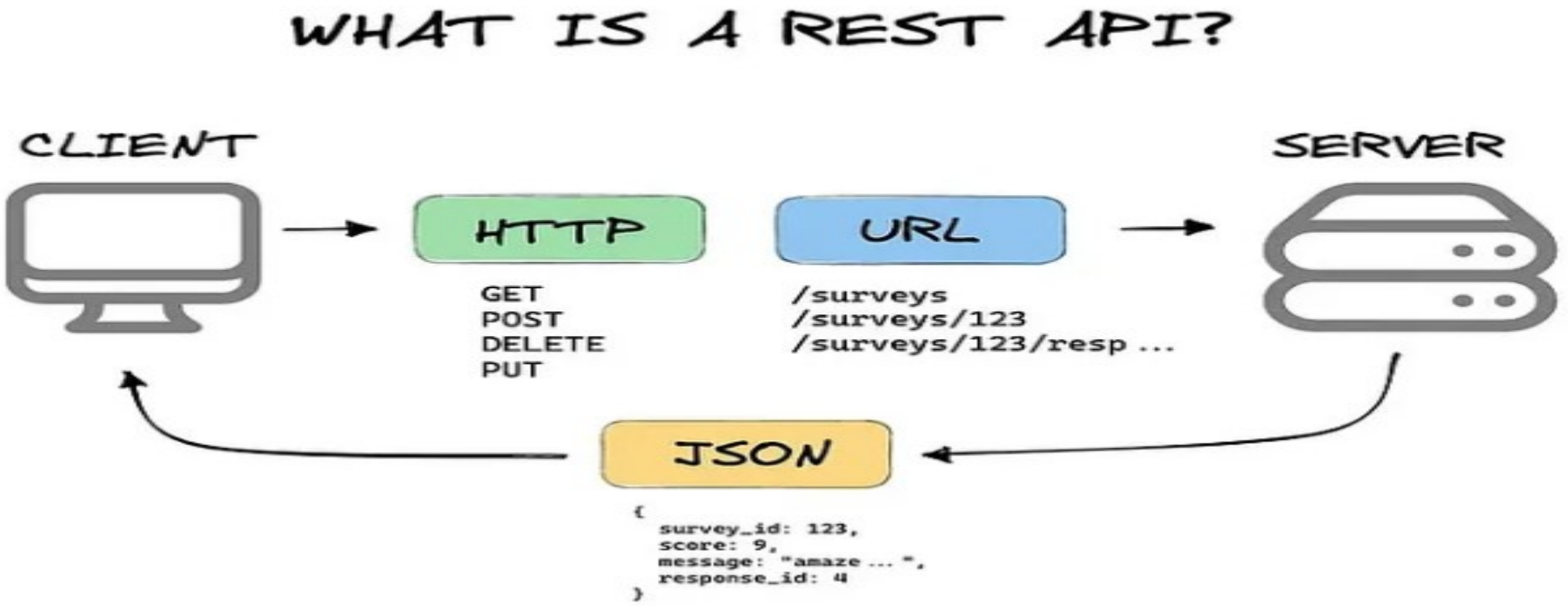
POST → Create new data

PUT → Update existing data

DELETE → Remove data

- **Resource-Based URLs:** Resources are accessed via meaningful URLs (e.g., /users, /products).
- **Use of JSON:** Responses are typically returned in JSON format.

Architecture of REST API



Contd.

- **REST Key Components**
- **Resources:** Key entities exposed by the API, each identified by a unique URl.
- **HTTP Methods:** Used to perform CRUD operations on resources (GET, POST, PUT, DELETE).
- **Representations:** Data format representing resource state (e.g., JSON, XML).
- **Hypermedia Links:** Embedded links in responses to enable dynamic navigation.
- **Status Codes:** Indicate the status of a request (e.g., 200 OK, 404 Not Found).

HTTP Methods = CRUD Operations

HTTP Method	CRUD	Example Endpoint	Action
`GET`	Read	`/users`	Fetch all users.
`POST`	Create	`/users`	Add a new user.
`PUT`	Update	`/users/1`	Replace user data.
`DELETE`	Delete	`/users/1`	Remove a user.

Status Codes

Key Codes:

200 OK → Success.

201 Created → Resource added.

400 Bad Request → Invalid input.

404 Not Found → Missing resource.

Testing your RESTful APIs

1. **Command Line (cURL)**
    - winget install curl.curl (Install the cURL)
    - curl –version (Check if cURL is Already Installed)
2. **Browser (GET Only)**
3. **Postman (GUI)**
4. **Automated Testing (JavaScript)**

Build a RESTful User API with Express.js

```
// Import dependencies
const express = require('express');
const bodyParser = require('body-parser');

// Initialize Express app
const app = express();
const PORT = process.env.PORT || 5000;

// Middleware (Telling Express to understand JSON data).
app.use(bodyParser.json());
```

Contd.

```
// Sample data
let users = [
    { id: 1, name: 'Vamsi', email: 'vamsi@example.com' },
    { id: 2, name: 'Chandra', email: 'chandra@example.com' }
];

// Debugging: Check if the app is initialized
console.log("Express app initialized successfully");
```

Contd.

```
// Routes
// Get all users
app.get('/users', (req, res) => {
    res.json(users);
});
// Get a single user by ID
app.get('/users/:id', (req, res) => {
    const user = users.find(u => u.id === parseInt(req.params.id));
    if (!user) return res.status(404).send('User not found');
    res.json(user);
});
```

Contd

```
// Create a new user
app.post('/users', (req, res) => {
    if (!req.body.name || !req.body.email) {
        return res.status(400).send('Name and email are required');
    }
    const newUser = {
        id: users.length + 1,
        name: req.body.name,
        email: req.body.email
    };
    users.push(newUser);
    res.status(201).json(newUser);
});
```

Contd.

```
// Update a user
app.put('/users/:id', (req, res) => {
    const user = users.find(u => u.id === parseInt(req.params.id));
    if (!user) return res.status(404).send('User not found');

    user.name = req.body.name || user.name;
    user.email = req.body.email || user.email;
    res.json(user);
});
```

Contd.

```
// Delete a user
app.delete('/users/:id', (req, res) => {
    const userIndex = users.findIndex(u => u.id === parseInt(req.params.id));
    if (userIndex === -1) return res.status(404).send('User not found');
    users.splice(userIndex, 1);
    res.json({ message: 'User deleted successfully' });
});

// Start server
app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

RESTful API Command Line Execution

1. Start Your Server

node api.js

✅ **Expected Output:**

Server running on port 5000

2. Open a New Terminal

Test these cURL commands in a separate terminal window:

GET Requests

Description	Command
Fetch all users	`curl http://localhost:5000/users`
Fetch single user (ID=1)	`curl http://localhost:5000/users/1`

Contd.

Sample Response (GET /users):


```
 {"id":1,"name":"Vamsi","email":"vamsi@example.com"},
 {"id":2,"name":"Chandra","email":"chandra@example.com"}
]
```


POST Request (Create User)

```
curl -X POST http://localhost:5000/users ^
    -H "Content-Type: application/json" ^
    -d "{\"name\":\"charan\",\"email\":\"charan@example.com\"}"
```

✅ **Success Response (201 Created):**


```
{"id":3,"name":"charan","email":"charan@example.com"}
```

Contd.

PUT Request (Update User)

```
curl -X PUT http://localhost:5000/users/1 ^
    -H "Content-Type: application/json" ^
    -d "{\"name\":\"Charan up\",\"email\":\"charan@example.com\"}"
```

✅ **Success Response (200 OK):**

{"id":1,"name":"Charan up","email":"charan@example.com"}

DELETE Request

```
curl -X DELETE http://localhost:5000/users/1
```

✅ **Success Response (200 OK):**

{"message":"User deleted successfully"}