Node JS

Node.js is a runtime environment that allows you to run JavaScript outside the browser, typically on the server side.

📌 **In Simple Terms**:
Normally, JavaScript runs in browsers (like Chrome or Firefox) for things like animations, forms, and interactivity.

But Node.js lets you run JavaScript on your computer/server, so you can build backend applications like:

- Servers (e.g., APIs)
- Command-line tools
- Real-time apps (like chat apps)
- File systems or databases interaction

Node.js is a JavaScript Runtime Environment.

- Runs JavaScript code outside the browser.
- Built on Chrome's V8 Engine. (Made with C++)

🏗️ Who Created Node.js and Why?

- **Created by**: Ryan Dahl
- **Released in**: 2009 (initial work started in 2007)
- **Reason**:
  - Traditional servers like **Apache** handled concurrent requests inefficiently.
  - Node.js was designed for **non-blocking**, event-driven, real-time applications.

Installing Node.js

- Download from https://nodejs.org
- Choose:
  - **LTS (Long Term Support)**: Stable version recommended for most users.
  - **Current**: Latest features but less stable.

Running JavaScript Files with Node

- Use the terminal/command prompt:

```
node <filename> .js
```

- Node provides its own runtime environment with built-in **APIs** like:
  - `fs` (file system)
  - `http` (server creation)

- `path`, etc.

📦 Packages in Node.js

- **Packages** are reusable libraries or tools.
- Installed using **npm (Node Package Manager)**.
- Example:

```
npm install cat-me
```

☯️ Packages vs Modules

Feature	Package	Module
Definition	Third-party tools/libraries	Built-in features provided by Node.js
Source	Installed via `npm`	Comes with Node.js
Examples	`express`, `cat-me`	`http`, `fs`, `path`

💻 Server Create Through HTTP Module

- Make a file named `server.js`

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

- While installing `cat-me` we used `npm install cat-me` but we're not using any npm packages while running `http`

Reason: `http` is a module, not a package.

Server Creation:

```
http.createServer()
```

Server Start:

```
const server = http.createServer()

server.listen(3000,()=>{
  console.log("Server is running on port 3000")
})
```

- The callback will get executed when the server is ready to take requests & handle it.

Request & Response

```
const http = require('http')
const server = http.createServer((req, res)=>{
  res.end("hello World From The Server")
})

server.listen(3000,()=>{
  console.log("Server is running on port 3000")
})
```

- programming the server - if any request comes this will be the consistent response.

🟤 Why We don't Use HTTP Server Directly?

Node.js comes with a built-in `http` module, which lets you create a web server.

☑ It Works fine for very basic servers.
❌ But quickly becomes messy as you add more features like routes, middleware, JSON parsing, authentication, etc.

Express is a framework built on top of Node's http module.

- It simplifies tasks that are cumbersome with raw `http`

🟤 Routing made easy

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

app.get('/', (req, res) => res.send('Hello World'));
app.get('/about', (req, res) => res.send('About Page'));

app.listen(3000, () => console.log('Server running'));
```

☑ Cleaner and scalable than multiple if conditions.

🟤 **Middleware Support**: Express lets you use middleware for tasks like logging, parsing JSON, authentication.

```
app.use(express.json()); // automatically parses JSON requests
```

🟢 **Error Handling**: Express has built-in ways to handle errors globally, rather than manually checking in every callback.

🟢 **Easier to integrate with templates, APIs, and databases**

Express works seamlessly with EJS, Pug, or Handlebars and APIs like MongoDB or MySQL.

🟢 **Large Ecosystem**: Many npm packages are designed to work with Express directly.

Installation

```
npm init -y
npm i express
```

Express Server Running

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

app.listen(3000,()=>{
    console.log("Server is running on port 3000");
})
```

This will show us an error on the screen - `Cannot GET \`
Hence we'll do another step

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

app.get('/home',(req,res)=>{
    res.end("Home Page");
})
app.get('/about',(req,res)=>{
    res.end("About Page");
})

app.listen(3000,()=>{
    console.log("Server is running on port 3000");
})
```

Now in Terminal -`node server.js`
will show us Home Page written in Webpage `http://localhost:3000/home`

🙏 Request (req)

The Incoming Data from the client in a web server context. Object containing details of client requests.

- Whenever a client (like a browser, app, or API consumer) sends a request to your server, all the details about that request are contained inside the req object.

"The data of whatever client has requested" = req object in backend.

```
app.get('/user', (req, res) => {
  console.log(req.query);   // Data sent in URL query ?name=pratik
  console.log(req.params);  // Data from route parameters /user/:id
  console.log(req.body);    // Data sent in request body (POST/PUT)
  console.log(req.headers); // Request headers like Content-Type, Auth tokens
  res.send('Request received');
});
```

Part	Description	Example
`req.body`	Data sent in POST/PUT requests	`{ username: "pratik" }`
`req.query`	Data from URL query string	`/user?age=22` → `{ age: "22" }`
`req.params`	Data from route parameters	`/user/10` → `{ id: "10" }`
`req.headers`, `req.cookies`	Metadata (Credentials) about the request	Authorization, Content-Type

Response (res)

Object your server uses to send data back to the client after processing their request.

```
app.get('/hello', (req, res) => {
  res.send('Hello, Client!');         // Sends a plain text response
});
```

Method	Purpose	Example
`res.send()`	Sends text, HTML, or JSON automatically	`res.send('Welcome!')`
`res.json()`	Sends a JSON response	`res.json({ success: true })`
`res.status()`	Sets HTTP status code	`res.status(404).send('Not Found')`
`res.redirect()`	Redirects client to another URL	`res.redirect('/login')`
`res.render()`	Renders a template (used with view engines)	`res.render('index', { user })`

📢 What is an API?

API (Application Programming Interface) is a set of rules and definitions that allows two software applications to communicate or interact with each other.

An API acts like a messenger - it takes a request from one application, tells another application what it needs to do, and then returns the response back.

- It allows one software to request data or services from another.
- How the communication happens doesn't matter - no strict rules or structure needed.

🧩 Example:

1. When your weather app fetches current temperature — it's calling a `Weather API`.
2. When you log in with Google on another site — that site uses `Google's API` to verify your account.

🌐 What is a REST API?

REST (Representational State Transfer) is a set of architectural principles for designing web APIs that allow communication between client and server using standard HTTP methods.

- A type of API that follows specific rules and guidelines for communication.

So, a REST API is an API that follows REST principles to handle requests and responses in a consistent, predictable way.

⚙️ Key Characteristics:

- Uses HTTP Methods:
    - GET → Retrieve data
    - POST → Create new data
    - PUT → Update existing data
    - DELETE → Remove data
- Each request is independent - the server doesn't remember previous requests.
- Uses URLs to represent resources:
    - Example:

`/users` → all users

`/users/1` → user with ID 1

- Structured Data Format: Usually exchanges data in JSON (sometimes XML).

💡 Example:

- Client → "GET `/users/1`"
- Server → { "id": 1, "name": "Pratik" }