Node.js

Node.js is an open-source, cross-platform runtime environment that allows JavaScript to run outside the browser. It is built on **Chrome's V8 JavaScript engine** and is designed for building scalable and high-performance applications, particularly for backend development.

Why Use Node.js?

1. **Asynchronous and Event-Driven**: Handles multiple operations at once using non-blocking I/O.
2. **Single Programming Language**: Uses JavaScript for both frontend and backend.
3. **High Performance**: Uses the V8 engine, which compiles JavaScript into machine code for faster execution.
4. **Scalability**: Uses an event loop to handle concurrent requests efficiently.
5. **Large Ecosystem**: Has a rich package ecosystem via **npm** (Node Package Manager).

Core Concepts of Node.js

Node.js is a runtime environment that allows JavaScript to be executed outside the browser. It is built on Chrome’s V8 JavaScript engine and is widely used for building scalable and high-performance web applications. Below are the core concepts of Node.js:

1. Asynchronous and Event-Driven Architecture: Node.js operates on a non-blocking, event-driven architecture. This means it does not wait for operations (like reading a file or querying a database) to complete before executing the next task.

2. Single-Threaded Event Loop: Unlike traditional multi-threaded servers, Node.js uses a single-threaded event loop that efficiently handles multiple client requests without creating a new thread for each one.

3. Non-Blocking I/O: Input/output operations (such as reading from a database, network calls, or file system operations) do not block other processes from executing.

4. Node.js Modules: Node.js follows a modular architecture where functionalities are separated into reusable modules.

- **Types of Modules**:
    1. **Core Modules**: Built-in modules like fs, http, path, etc.
    2. **User-Defined Modules**: Custom modules created by developers.
    3. **Third-Party Modules**: Installed using npm (Node Package Manager), e.g., express, mongoose, etc.

5. Package Management (npm): npm (Node Package Manager) is used to install, manage, and update Node.js packages.

6. Express.js Framework: Express.js is a minimal web framework for Node.js used to create RESTful APIs.

7. File System Module (fs): The fs module allows interaction with the file system.

8. Buffers & Streams

- **Buffers**: Used to handle binary data directly in memory.
- **Stream**: Used to handle large files without loading them into memory all at once.

9. Event-Driven Programming: Node.js uses an EventEmitter class to handle events.

10. Middleware in Express.js: Middleware functions handle requests before passing them to the next function.

11. Database Integration: Node.js supports databases like MongoDB (NoSQL) and MySQL (SQL).

12. JSON and REST APIs: Node.js is commonly used for creating RESTful APIs that exchange data in JSON format.

13. Authentication & Security: Authentication in Node.js is done using JWT (JSON Web Token), OAuth, or sessions.

14. WebSockets for Real-Time Communication: WebSockets enable real-time bidirectional communication between the client and server.

15. Deployment of Node.js Applications: Applications can be deployed using tools like **PM2**, **Docker**, and cloud platforms like **AWS, Heroku, and Vercel**.

Non-Blocking I/O & Event-Driven Architecture

Node.js uses **asynchronous programming**, meaning it does not wait for a task to complete before moving on to the next one. It relies on **event-driven programming** and a **single-threaded event loop** to handle multiple requests.
Example of Non-Blocking I/O:
javascript
CopyEdit

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

fs.readFile('file.txt', 'utf8', (err, data) => {
```

```
    if (err) {
        console.error(err);
        return;
    }
    console.log(data);
});

console.log("This will execute before file reading completes");
```

Here, readFile() does not block the execution of other code while reading the file.

Single-Threaded Event Loop in Node.js

Node.js operates on a **single-threaded** event loop model, meaning it uses a single main thread to handle multiple client requests **asynchronously**. Instead of creating a separate thread for each request (like in multi-threaded servers), Node.js relies on **non-blocking I/O operations** and an **event loop** to efficiently process multiple requests concurrently.

The **event loop** is the heart of Node.js. It continuously listens for incoming requests and delegates tasks efficiently.

Step-by-step process:

1. **Incoming request** → The event loop receives a request (I/O operation, database call, file system read, etc.).
2. **Check if blocking or non-blocking**:
    - If it's a **non-blocking task** (e.g., API call, file read, database query), Node.js delegates it to the appropriate system kernel thread (using the **libuv** library).
    - If it's a **blocking task** (like heavy computation), it can slow down the event loop.
3. **Execution of callbacks**: Once an asynchronous operation completes, its callback function is added to the event loop's queue.
4. **Event loop processes tasks continuously** until all queued tasks are completed.

Event Loop Phases

The event loop consists of multiple phases:

1. **Timers Phase**: Executes setTimeout() and setInterval() callbacks.
2. **I/O Callbacks Phase**: Executes pending I/O operations (file system, network, etc.).
3. **Idle & Prepare Phase**: Internal use (not relevant for most developers).
4. **Poll Phase**: Retrieves new I/O events and executes them if available.
5. **Check Phase**: Executes setImmediate() callbacks.
6. **Close Callbacks Phase**: Handles close events (e.g., socket.on('close', callback)).

```
console.log("1. Start");
setTimeout(() => {
    console.log("2. Timeout callback");
}, 0);
```

```
setImmediate(() => {
   console.log("3. Immediate callback");
});
console.log("4. End");
```

Modules in Node.js

A module in Node.js is a JavaScript file that stores reusable code. It helps organize an application by dividing it into smaller, manageable sections.

Types of Modules in Node.js

Node.js has three main types of modules:

1. **Core Modules** – Built-in modules like fs, http, path, etc.
2. **User-Defined Modules** – Custom modules created by developers.
3. **Third-Party Modules** – Installed via npm (e.g., express, mongoose).

Importing and Exporting Modules in Node.js

Node.js provides **CommonJS** module syntax (require and module.exports) and **ES6 Modules** (import and export).

CommonJS (Default in Node.js)

Exporting a module (utils.js):

javascript

CopyEdit

```
module.exports = {
   greet: function(name) {
      return `Hello, ${name}!`;
   }
};
```

Importing the module (index.js):

javascript

CopyEdit

```
const utils = require('./utils');
console.log(utils.greet('Prashant'));
```

ES6 Modules (Requires "type": "module" in package.json)

Exporting (utils.mjs):

javascript

CopyEdit

```
export function greet(name) {
   return `Hello, ${name}!`;
}
```

Importing (index.mjs):

javascript

CopyEdit

```
import { greet } from './utils.mjs';
console.log(greet('Prashant'));
```

Package Management with npm in Node.js

npm (Node Package Manager) is a tool used to install, manage, and share JavaScript packages (libraries) in Node.js. It is the **default package manager** for Node.js and comes bundled with Node.js installation.

Key Features of npm:

- Installs, updates, and removes packages (libraries).
- Manages dependencies in a project.
- Allows publishing and sharing custom packages.
- Provides access to a vast repository of open-source libraries.

2. Installing npm

npm is automatically installed when you install Node.js. To check if it's installed, run:

```
node -v   # Check Node.js version
npm -v    # Check npm version
```

3. Package Management with npm

a) Initializing a Node.js Project (package.json)

Every Node.js project with dependencies requires a **package.json** file, which stores project details and installed dependencies.

To create a package.json file, run: npm init

npm init -y # Creates package.json with default values

b) Installing Packages

You can install packages globally or locally.

1. Install a package locally (for the project)

npm install package-name

Example: Installing Express

npm install express

2. Install a package globally (for all projects)

npm install -g package-name

Example: Installing Nodemon globally

npm install -g nodemon

Global packages are stored in a system-wide location and can be used anywhere.

3. Install a package as a development dependency

- Some packages are needed only during development, such as testing tools. Use --save-dev:
- npm install eslint --save-dev

c) Removing Packages

To remove an installed package: npm uninstall package-name

Example: npm uninstall express

To remove a global package: npm uninstall -g package-name

d) Updating Packages

To update all packages to their latest versions: npm update

To update a specific package: npm update package-name

To update a global package: npm update -g package-name

To check outdated packages: npm outdated

e) Installing Packages from package.json

If you clone a project from GitHub, it will have a package.json file but no node_modules. To install all required packages, run: npm install

4. Using package-lock.json

When you install packages, npm creates a package-lock.json file.

- It locks dependency versions to ensure consistency across installations.
- It speeds up future installs.
- It helps teams maintain the same package versions.

5. Running Scripts with npm

npm run start

npm run dev

6. Publishing a Package on npm: If you want to share your own package:

1. **Login to npm**: npm login
2. **Publish your package**: npm publish

7.Clearing npm Cache

If you face installation issues, clear the cache: npm cache clean --force

Express.js

Express.js is a **fast, minimal, and flexible web framework** for Node.js used to build web applications and APIs. It simplifies handling HTTP requests, routing, middleware, and more.

Why Use Express.js?

- **Lightweight** – It provides essential web functionalities without adding extra complexity.

- **Middleware Support** – Enables request processing, authentication, logging, and more.
- **Routing System** – Helps define clean, structured routes for web applications.
- **Supports Templating Engines** – Works with **EJS, Pug, and Handlebars** for dynamic HTML pages.
- **Built-in Error Handling** – Catches and manages errors efficiently.
- **Compatible with Databases** – Works with **MongoDB (Mongoose), MySQL, PostgreSQL**, etc.

Installing Express.js

Before using Express, ensure **Node.js** and **npm** are installed. Then, install Express in your project: npm install express

Basic Express.js Server

This simple server responds with "Hello, Express!" when accessed at http://localhost:3000/.

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

const app = express();

const PORT = 3000;


// Define a route

app.get('/', (req, res) => {

   res.send('Hello, Express!');

});


// Start the server

app.listen(PORT, () => {

   console.log(`Server running on http://localhost:${PORT}`);

});
```

Run the server: node server.js

Routing in Express.js

Express allows defining multiple routes.

```
const express = require('express');
const app = express();
const PORT = 3000;
app.get('/', (req, res) => res.send('Home Page'));
app.get('/about', (req, res) => res.send('About Page'));
app.post('/submit', (req, res) => res.send('Data Submitted'));
app.put('/update', (req, res) => res.send('Data Updated'));
app.delete('/delete', (req, res) => res.send('Data Deleted'));
app.listen(PORT, () => console.log(`Server running on http://localhost:${PORT}`));
```

Middleware in Express.js

```
const express = require('express');
const app = express();
const PORT = 3000;
// Custom middleware function
const logger = (req, res, next) => {
    console.log(`${req.method} request for ${req.url}`);
    next(); // Pass control to the next handler
};
app.use(logger);
app.get('/', (req, res) => res.send('Hello, Middleware!'));
app.listen(PORT, () => console.log(`Server running on http://localhost:${PORT}`));
```

Handling JSON and URL-Encoded Data

```
const express = require('express');
const app = express();
const PORT = 3000;
// Middleware to parse JSON and form data
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.post('/user', (req, res) => {
   console.log(req.body); // Logs received data
   res.send(`User: ${req.body.name}`);
});
app.listen(PORT, () => console.log(`Server running on http://localhost:${PORT}`));
```

What is a REST API?

A **REST API (Representational State Transfer Application Programming Interface)** is a set of rules and conventions that allow systems to communicate over the internet using **HTTP methods**. It enables clients (e.g., web applications, mobile apps) to interact with servers by sending **requests** and receiving **responses** in a standardized format, typically **JSON** or **XML**.

Key Features of REST API

1. **Stateless**: Each request from the client contains all the necessary information; the server does not store the client's state.
2. **Client-Server Architecture**: The client and server are independent of each other.
3. **Uniform Interface**: Resources are accessed via standard HTTP methods.
4. **Resource-Based**: Everything is treated as a resource (e.g., users, products).
5. **Cacheable**: Responses can be cached to improve performance.
6. **Layered System**: The API architecture allows multiple layers (e.g., security, caching, load balancing).

HTTP Methods in REST API

HTTP Method	Description	Example
GET	Retrieve data	`GET /users` (Fetch all users)
POST	Create a new resource	`POST /users` (Create a new user)
PUT	Update an existing resource	`PUT /users/1` (Update user with ID 1)
PATCH	Partially update a resource	`PATCH /users/1` (Modify user with ID 1)
DELETE	Remove a resource	`DELETE /users/1` (Delete user with ID 1)

REST API Using Node.js and Express

Step 1: Install Node.js and Express

```
npm init -y
npm install express
```

Step 2: Create an API Server

```
const express = require('express');
const app = express();
app.use(express.json()); // Middleware to parse JSON
let users = [
    { id: 1, name: 'xyz', email: 'xyz@example.com' },
    { id: 2, name: 'xyz', email: 'xyz@example.com' }
];
// GET - Fetch all users
app.get('/users', (req, res) => {
    res.json(users);
});
// GET - Fetch 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).json({ message: 'User not found' });
```

```
    res.json(user);
});
// POST - Create a new user
app.post('/users', (req, res) => {
    const newUser = { id: users.length + 1, ...req.body };
    users.push(newUser);
    res.status(201).json(newUser);
});
// PUT - Update a user
app.put('/users/:id', (req, res) => {
    let user = users.find(u => u.id === parseInt(req.params.id));
    if (!user) return res.status(404).json({ message: 'User not found' });
    user.name = req.body.name;
    user.email = req.body.email;
    res.json(user);
});
// DELETE - Remove a user
app.delete('/users/:id', (req, res) => {
    users = users.filter(u => u.id !== parseInt(req.params.id));
    res.json({ message: 'User deleted successfully' });
});
// Start the server
app.listen(3000, () => {
    console.log('Server is running on http://localhost:3000');
});
```

Testing the REST API

You can test the API using:

1. **Postman** (GUI tool for API testing)
2. **cURL** (Command-line tool)
3. **Browser** (For GET requests)

Introduction to TypeScript

What is TypeScript?

TypeScript is a **superset of JavaScript** developed by Microsoft that adds **static typing** to JavaScript. It helps developers write more maintainable and error-free code by catching errors during development rather than runtime.

Key Features of TypeScript:

1. **Static Typing** – Variables, functions, and objects can have specific types.
2. **Object-Oriented Programming (OOP) Support** – Includes classes, interfaces, and inheritance.
3. **Enhanced JavaScript Features** – Supports ES6+ features like arrow functions, destructuring, and modules.
4. **Compilation to JavaScript** – TypeScript code is compiled into standard JavaScript, making it compatible with all browsers.
5. **Tooling & IDE Support** – Works well with Visual Studio Code, offering autocompletion, error checking, and refactoring tools.

Why Use TypeScript?

- **Error Detection:** Detects errors at compile time, reducing runtime bugs.
- **Better Readability:** Code is easier to understand and maintain.
- **Scalability:** Useful for large-scale applications with multiple developers.
- **Improved Productivity:** Features like autocompletion and refactoring speed up development.

Basic TypeScript Syntax

1. Type Annotations

```
let username: string = "John";

let age: number = 25;

let isStudent: boolean = true;
```

2. Functions with Types

```
function add(a: number, b: number): number {
   return a + b;
}

console.log(add(5, 10));
```

3. Interfaces

```
interface Person {
    name: string;
    age: number;
}
let user: Person = { name: "Alice", age: 30 };
console.log(user);
```

4. Classes

```
class Car {
    brand: string;
    constructor(brand: string) {
        this.brand = brand;
    }
    display(): void {
        console.log(`Car brand: ${this.brand}`);
    }
}
let myCar = new Car("Toyota");
myCar.display();
```

Introduction to Angular

Angular is a **TypeScript-based open-source framework** developed by Google for building **dynamic, modern, and scalable** web applications. It follows the **Component-Based Architecture** and provides tools to create **single-page applications (SPAs)** with a seamless user experience.

Why Use Angular?

Angular is preferred for **enterprise-level** applications due to the following benefits:

1. **Component-Based Architecture** → Code is divided into reusable components.
2. **Two-Way Data Binding** → Synchronizes the model and the view automatically.
3. **Dependency Injection (DI)** → Enhances code modularity and reusability.

4. **Directives** → Extend HTML functionality using custom elements.
5. **Built-in Routing** → Create Single Page Applications (SPA) with client-side navigation.
6. **TypeScript Support** → Offers strong typing and better maintainability.
7. **Cross-Platform** → Works for **web, mobile, and desktop applications**.
8. **Faster Rendering** → Uses **Ahead-of-Time (AOT) compilation** for better performance.

Core Building Blocks of Angular

Angular applications are made up of the following key components:

Angular Concept	Description
Modules (NgModules)	Organizes the application into logical units.
Components	UI building blocks of an application.
Templates & Directives	Define how the HTML should be structured and modified.
Services & Dependency Injection	Used to fetch, manipulate, and share data between components.
Routing	Manages navigation between different views.
Pipes	Transforms displayed data (e.g., formatting dates, numbers).

1. Angular Modules (NgModules)

Modules are containers for different application parts, making it modular and maintainable.

- Every Angular app has at least one **root module**, i.e., AppModule.
- Defined using @NgModule decorator.

Example: Creating a Module

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
```

```
@NgModule({
  declarations: [AppComponent], // Components
  imports: [BrowserModule],     // Angular built-in modules
  providers: [],                // Services
  bootstrap: [AppComponent]     // Root component
})
export class AppModule { }
```

2. Components in Angular

A **component** is the fundamental building block of an Angular application. It consists of:

1. **Template (HTML)**
2. **Class (TypeScript)**
3. **Styles (CSS/SCSS)**

Component Structure

- Defined using @Component decorator.
- Components contain **logic and UI**.

Example: Creating a Component

typescript

CopyEdit

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-hello',
  template: `<h1>Hello, {{name}}!</h1>`,
  styles: [`h1 { color: blue; }`]
})
export class HelloComponent {
  name = 'Angular';
}
```

Adding Component to a Module

typescript

CopyEdit

```
@NgModule({
  declarations: [HelloComponent], // Registering component
  bootstrap: [HelloComponent]
})
```

3. Templates & Directives

Templates: Templates define the **HTML structure** of a component.

Example:

```
<div>
  <h1>Welcome {{ userName }}</h1>
</div>
```

Directives: Directives modify **HTML elements, attributes, and components**.

Types of Directives

Type	**Directive**	**Description**
Structural	*ngIf, *ngFor, *ngSwitch	Controls the DOM structure.
Attribute	[ngClass], [ngStyle]	Changes the appearance of elements.
Custom	@Directive	Allows defining custom behavior.

Example: Using *ngIf

```
  <p *ngIf="isLoggedIn">Welcome back!</p>
```

Example: Using *ngFor

```
<ul>
  <li *ngFor="let item of items">{{ item }}</li>
</ul>
```

4. Data Binding in Angular: Data Binding is used to **synchronize** data between the model and view.

Type	Syntax	Example
Interpolation	{{ data }}	{{ title }}
Property Binding	[property]="data"	<img [src]="imageUrl">
Event Binding	(event)="function()"	<button (click)="showMessage()">Click</button>
Two-Way Binding	[(ngModel)]="data"	<input [(ngModel)]="name">

Example of Two-Way Binding

```
<input [(ngModel)]="username">
<p>Hello, {{ username }}!</p>
```

5. Services & Dependency Injection

Services allow **sharing data** between components.

Creating a Service

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class DataService {
  getData() {
    return "Data from Service";
  }
}
```

Injecting a Service in a Component

```
import { Component } from '@angular/core';
import { DataService } from './data.service';
```

```
@Component({
  selector: 'app-data',
  template: `<p>{{ data }}</p>`
})
export class DataComponent {
  data: string;
  constructor(private dataService: DataService) {
    this.data = this.dataService.getData();
  }
}
```

6. Routing in Angular

Routing allows navigation between different views in **Single Page Applications (SPA)**.

Step 1: Configure Routes

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Step 2: Add Router Outlet in Template

```
<nav>
  <a routerLink="/">Home</a>
  <a routerLink="/about">About</a>
</nav>
<router-outlet></router-outlet>
```

7. Pipes in Angular: Pipes **transform** data in templates.

Example of Built-in Pipes

```
<p> {{ today | date:'short' }}</p>
<p> {{ price | currency:'USD' }}</p>
```

Pipe	**Purpose**
date	Formats date values
uppercase	Converts text to uppercase
lowercase	Converts text to lowercase
currency	Formats numbers as currency

Angular CLI (Command Line Interface)

Angular CLI (Command Line Interface) is a powerful tool provided by Angular to automate development tasks such as creating components, services, modules, running a development server, and building production-ready code.

Why Use Angular CLI?

- Automates Project Setup – Quickly generates the folder structure and configurations.
- Generates Components, Modules, Services, etc. – Reduces manual work.
- Hot Reloading & Live Development Server – Speeds up development.
- Efficient Production Builds – Optimized performance.
- Testing & Linting Support – In-built unit testing and code quality checks.

Installing Angular CLI

To install Angular CLI globally on your system, use: npm install -g @angular/cli

Check if it's installed correctly: ng version

2. Creating a New Angular Project

To create a new Angular application, run:

ng new my-angular-app

You will be asked:

- **Would you like to add Angular routing?** → Type Y or N
- **Which stylesheet format?** → Choose CSS, SCSS, LESS, or Stylus

After installation, navigate to your project folder:

cd my-angular-app

Start the development server: ng serve

By default, the app runs at: http://localhost:4200/

Services in Angular

A **service** in Angular is a **reusable business logic unit** used to **fetch, store, and share data across components**. It **improves code modularity** by separating data-related logic from the UI.

Why Use Services in Angular?

✓ **Code Reusability** – Shared logic across multiple components
✓ **Separation of Concerns** – Keeps components lightweight
✓ **Dependency Injection (DI)** – Efficient data management
✓ **Data Sharing** – Components can communicate using services
✓ **API Handling** – Fetch data from backend servers

Creating a Service in Angular

To create a new service, use the Angular CLI: ng generate service my-service

Defining an Angular Service

A service is a **class decorated with @Injectable()**, meaning it can be injected into other parts of the application.

import { Injectable } from '@angular/core';

```
@Injectable({
  providedIn: 'root' // Automatically provides service at root level
})
export class MyService {
  getMessage() {
    return "Hello from MyService!";
  }
}
```

React.js: Overview, Uses, Advantages, and Disadvantages

What is React.js?

React.js is an open-source **JavaScript library** developed by **Meta (Facebook)** for building **interactive user interfaces (UIs)** and **single-page applications (SPAs)**. It follows a **component-based architecture**, allowing developers to create reusable UI components.

📌 **Key Features of React.js:**
✓ **Component-Based Architecture** – Breaks UI into reusable components
✓ **Virtual DOM** – Improves performance by minimizing real DOM updates
✓ **One-Way Data Binding** – Enhances predictability and debugging
✓ **JSX (JavaScript XML)** – Allows writing HTML inside JavaScript
✓ **Declarative UI** – Updates UI automatically when data changes
✓ **Fast Rendering** – Uses efficient diffing algorithms

Why Use React.js?

React is widely used for **web applications, dashboards, mobile apps (React Native), and enterprise-level applications** because of the following reasons:

1**Performance Optimization**

- Uses **Virtual DOM** for efficient UI rendering.
- Updates only necessary parts of the DOM instead of the entire page.

2**Reusable Components**

- Encourages the development of **modular and maintainable** applications.

- UI elements are independent and reusable.

3 **Strong Community Support**

- Backed by **Meta (Facebook)** and an **active open-source community**.
- Plenty of third-party libraries and tools available.

4 **SEO-Friendly**

- Can improve SEO when used with **server-side rendering (SSR)** like **Next.js**.

5 **Flexible & Scalable**

- Works well with other libraries like **Redux** for state management.
- Can be used in **small** and **large-scale applications**.

Advantages of React.js

1. **Fast and Efficient** – Uses **Virtual DOM** to enhance performance.
2. **Component-Based** – Allows reusable UI components.
3. **Unidirectional Data Flow** – Makes debugging easier and ensures data consistency.
4. **Easy to Learn** – Uses JSX, which is similar to HTML & JavaScript.
5. **Strong Ecosystem** – Supports tools like **Redux, React Router, and Next.js**.
6. **Cross-Platform** – Can be used to build **mobile apps (React Native)**.
7. **Active Community** – Large support base and frequent updates.

Disadvantages of React.js

1. **Steep Learning Curve** – Requires knowledge of **JSX, props, state, hooks, and lifecycle methods**.
2. **Complex State Management** – Large applications require **Redux, Context API, or MobX**.
3. **Frequent Updates** – Fast-changing ecosystem can make previous versions obsolete.
4. **SEO Limitations** – Client-side rendering (CSR) may impact SEO performance.
5. **Boilerplate Code** – Requires additional configurations compared to simpler libraries like **jQuery**.

Feature	React.js	Angular
Type	Library	Framework
Performance	High (Virtual DOM)	Medium
Learning Curve	Medium	Steep
Data Binding	One-way	Two-way
Best Use Case	SPAs, UI Components	Large Apps
Community Support	High	High

JSX in React.js

JSX (**JavaScript XML**) is a **syntax extension for JavaScript** used in **React.js** that allows developers to write **HTML-like code inside JavaScript**. It helps create **UI components in a more readable and declarative way**.

JSX is **not valid JavaScript**, but React **compiles JSX into regular JavaScript** using **Babel** before execution.

Why Use JSX?

1. **Easier to Read & Write** – Combines HTML and JavaScript in a single file.
2. **Prevents XSS Attacks** – React **escapes** JSX expressions to prevent injection attacks.
3. **Optimized for Performance** – JSX is converted into **React.createElement()**, making it efficient.
4. **Component-Based** – JSX simplifies the creation of reusable UI components.

JSX Syntax and Examples

1. Basic JSX Example

```
const element = <h1>Hello, JSX!</h1>;

ReactDOM.render(element, document.getElementById('root'));
```

This creates an **h1** element and renders it inside a div with the ID **root**.

2. JSX with JavaScript Expressions ({})

JSX allows embedding **JavaScript expressions** inside {}.

```
const name = "John";
```

```
const element = <h1>Hello, {name}!</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

{name} gets replaced with **John**.

3. JSX with Functions

```
function greet(user) {
  return <h1>Hello, {user}!</h1>;
}
const element = greet("Alice");
ReactDOM.render(element, document.getElementById('root'));
```

JSX can be returned from functions, making UI dynamic.

4. JSX with Conditional Rendering (? and &&)

JSX supports **if-else conditions** and logical operators.

Ternary Operator (? :)

```
const isLoggedIn = true;
const message = <h1>{isLoggedIn ? "Welcome Back!" : "Please Sign In"}</h1>;
```

If isLoggedIn is true, it shows **"Welcome Back!"**, otherwise **"Please Sign In"**.

Logical AND (&&)

```
const isAdmin = true;
const adminMessage = isAdmin && <h1>Admin Panel</h1>;
```

If isAdmin is true, it renders **"Admin Panel"**.

5. JSX with Inline Styles

JSX **uses objects** for inline styles instead of regular CSS.

```
const style = {
  color: "blue",
  fontSize: "20px",
  backgroundColor: "lightgray"
};
```

const element = <h1 style={style}>Styled JSX</h1>;

Keys are in camelCase (backgroundColor instead of background-color).

6. JSX with Class Attributes (className instead of class)

JSX uses className instead of class, as class is a reserved JavaScript keyword.

const element = <h1 className="title">JSX Class Example</h1>;

This applies the **"title"** CSS class.

7. JSX with Fragments (<>...</>)

React requires a **single parent element** in JSX. Use <React.Fragment> or <>...</> to avoid extra s.

Without Fragment (Incorrect)

```
return (
  <h1>Title</h1>
  <p>Paragraph</p>  // ❌ Error: Adjacent JSX elements must be wrapped
);
```

With Fragment (Correct)

```
return (
  <>
    <h1>Title</h1>
    <p>Paragraph</p>
  </>
);
```

Fragments prevent unnecessary DOM elements.

Feature	JSX	HTML
Attributes	Uses camelCase (`className` , `tabIndex`)	Uses lowercase (`class` , `tabindex`)
JavaScript Expressions	`{variable}` allowed	No JS inside HTML
Self-Closing Tags	Required (`<img />` , ` `)	Optional (`<img>` , ` `)
Inline Styles	Uses objects (`style={{color: "red"}}`)	Uses strings (`style="color: red;"`)
Fragments	`<></>` or `<React.Fragment>`	No equivalent

Props and State in React.js

In React, **props** and **state** are two important concepts that help in managing data and rendering UI components efficiently.

What are Props in React?

Props (short for "properties") are used to pass data **from a parent component to a child component**. Props are **read-only** and cannot be modified by the child component.

Key Characteristics of Props:

- **Immutable:** Cannot be changed inside the child component.
- **Passed from Parent to Child:** Helps in communication between components.
- **Read-only:** Cannot be modified by the component that receives them.

Example: Using Props in React

Parent Component (App.js)

```
import React from 'react';
import Greeting from './Greeting';
function App() {
  return <Greeting name="XYZ" />;
}
export default App;
```

Child Component (Greeting.js)

```
import React from 'react';
```

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
export default Greeting;
```

Output:
Hello, XYZ!

What is State in React?

State is a built-in React object that holds data that can change over time and affects the component's rendering.

Key Characteristics of State:

- **Mutable:** Can be changed using the setState function in class components or useState in functional components.
- **Local to the Component:** Unlike props, state is managed inside a component and not passed from parent to child.
- **Triggers Re-render:** When state changes, React re-renders the component to reflect the updated data.

Example: Using State in React

Using useState in a Functional Component

```
import React, { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default Counter;
```

Feature	Props	State
Mutability	Immutable (read-only)	Mutable (can be updated)
Scope	Passed from parent to child	Local to the component
Modifiable?	Cannot be modified by the child	Can be modified using `setstate` or `useState`
Re-render Trigger?	Does not trigger re-render	Triggers re-render when updated
Usage	For passing static/fixed data	For managing dynamic/changing data

Using Props and State Together

```
import React, { useState } from 'react';
function UserProfile({ name }) {
  const [age, setAge] = useState(25);
  return (
    <div>
      <h1>Name: {name}</h1>  {/* Using Props */}
      <h2>Age: {age}</h2>  {/* Using State */}
      <button onClick={() => setAge(age + 1)}>Increase Age</button>
    </div>
  );
}
export default UserProfile;
```

Lifecycle of a React Component

React component lifecycle consists of **three main phases**:

1. **Mounting (Component is Created & Inserted into DOM)**
2. **Updating (Component is Re-rendered due to State/Props Changes)**
3. **Unmounting (Component is Removed from DOM)**

Each phase has **lifecycle methods** (for class components) or **React Hooks** (for functional components).

React Component Lifecycle - Complete Notes

Introduction

React components follow a lifecycle that consists of three main phases:

1. **Mounting** – The component is created and inserted into the DOM.
2. **Updating** – The component re-renders when state or props change.
3. **Unmounting** – The component is removed from the DOM.

Each phase has specific lifecycle methods (for class components) and equivalent hooks (for functional components).

1. Mounting Phase (Component is Created & Inserted into DOM)

The mounting phase occurs when the component is first created and added to the DOM.

Lifecycle Methods in Class Components

- constructor(props): Initializes state and binds event handlers.
- static getDerivedStateFromProps(props, state): Updates state before rendering the component.
- render(): Returns JSX to define the UI.
- componentDidMount(): Executes after the component is mounted. Used for API calls, event listeners, etc.

2. Updating Phase (Component Re-renders due to State/Props Changes)

This phase occurs when state or props change, causing the component to re-render.

Lifecycle Methods in Class Components

- static getDerivedStateFromProps(props, state): Updates state when new props are received.
- shouldComponentUpdate(nextProps, nextState): Determines if the component should re-render.
- render(): Re-renders the component.
- getSnapshotBeforeUpdate(prevProps, prevState): Captures values (e.g., scroll position) before update.
- componentDidUpdate(prevProps, prevState, snapshot): Runs after re-rendering, used for API calls.

3. Unmounting Phase (Component is Removed from DOM)

This phase occurs when the component is about to be removed from the DOM.

Lifecycle Method in Class Components

- componentWillUnmount(): Runs just before the component is removed, used for cleanup (timers, event listeners).

```
import React, { Component, useState, useEffect } from 'react';

// CLASS COMPONENT LIFECYCLE
class LifecycleExample extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
    console.log("1. Constructor called");
  }

  static getDerivedStateFromProps(props, state) {
    console.log("2. getDerivedStateFromProps called");
    return null;
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log("3. shouldComponentUpdate called");
    return true;
  }

  getSnapshotBeforeUpdate(prevProps, prevState) {
```

```
    console.log("4. getSnapshotBeforeUpdate called");
    return null;
  }

  componentDidUpdate(prevProps, prevState) {
    console.log("5. componentDidUpdate called");
  }

  componentDidMount() {
    console.log("6. componentDidMount called");
  }
  componentWillUnmount() {
    console.log("7. componentWillUnmount called");
  }
  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };
  render() {
    console.log("8. Render called");
    return (
      <div>
        <h1>Class Component Count: {this.state.count}</h1>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

export LifecycleExample ;
```

MongoDB

What is MongoDB?

MongoDB is a **NoSQL database** that stores data in a **document-oriented format** instead of traditional tables. It uses **BSON (Binary JSON)** for storing data, making it highly flexible and scalable.

Key Features of MongoDB

1. Schema-less Database (Flexible Data Model)

- Unlike SQL databases, MongoDB does not require a predefined schema.
- Each document can have a different structure, making it easy to modify data.

2. Document-Oriented Storage

- Stores data in **JSON-like BSON format** instead of tables and rows.
- Allows nested structures, arrays, and complex objects.

3. Scalability (Horizontal Scaling with Sharding)

- Supports **sharding**, meaning large datasets are distributed across multiple servers.
- Ensures fast read/write operations and prevents server overload.

4. High Performance & Speed

- Uses **indexing and in-memory processing** to enhance performance.
- Faster than traditional databases, especially for big data applications.

5. Replication & High Availability

- **Replica sets** allow automatic data duplication across multiple servers.
- Ensures system reliability even if a server fails.

6. Indexing for Fast Queries

- Supports indexing on any field to improve query performance.
- Without indexing, queries may take longer on large datasets.

7. Aggregation Framework

- Provides a powerful query mechanism to perform **filtering, sorting, and grouping** of data efficiently.

8. Load Balancing & Auto-Sharding

- Distributes read/write loads across multiple nodes for efficient processing.
- Supports automatic **load balancing** for large-scale applications.

Why Use MongoDB?

MongoDB is best for applications requiring **high scalability, flexible schema, and rapid data retrieval**. It is widely used in:

- **Big Data & Analytics** → Handles massive datasets efficiently.
- **Real-time Applications** → Ideal for social media, IoT, and live tracking.
- **E-commerce & Content Management** → Stores dynamic and unstructured data.
- **Cloud-Based Applications** → Easily deployable on cloud platforms like AWS and Azure.

Advantages of MongoDB

- **Flexible Schema** → No need for predefined tables and columns.
- **High Scalability** → Easily handles large amounts of data across multiple servers.
- **Fast Read/Write Operations** → Indexing boosts performance.
- **Easy Integration with JavaScript** → Uses JSON-like BSON format.
- **Replication & Fault Tolerance** → Ensures high availability.

Disadvantages of MongoDB

- **Higher Memory Usage** → BSON format consumes more memory than traditional SQL databases.
- **No ACID Transactions for Multiple Documents** → Not ideal for complex financial applications.
- **No Support for SQL Joins** → Managing relational data is more difficult.
- **Indexing Overhead** → Poorly indexed collections can slow down queries.

Feature	SQL (Relational)	NoSQL (Non-Relational)
Data Structure	Structured (Tables with Rows & Columns)	Unstructured/Semi-structured (JSON, BSON, Key-Value, Graph, Column-Family)
Schema	Fixed Schema (Predefined)	Flexible Schema (Dynamic)
Scalability	Vertically Scalable (Add More CPU/RAM)	Horizontally Scalable (Add More Servers)
Transactions	Supports ACID (Atomicity, Consistency, Isolation, Durability)	Supports BASE (Basically Available, Soft State, Eventual Consistency)
Query Language	SQL (Structured Query Language)	NoSQL Query Languages (e.g., MongoDB Query Language)
Best Use Cases	Banking, ERP, CRM, E-commerce	Real-time analytics, Big Data, IoT, Social Networks

CRUD Operations in MongoDB

MongoDB is a **NoSQL database** that stores data in **JSON-like BSON (Binary JSON) format**. CRUD (Create, Read, Update, Delete) are the four basic operations used to manage data in MongoDB.

---

1. Create (Insert Documents)

MongoDB allows inserting data into collections using:

- **insertOne()** – Inserts a **single document**.
- **insertMany()** – Inserts **multiple documents**.

Example: Insert One Document

js

CopyEdit

```
db.users.insertOne({
  name: "John Doe",
  age: 30,
  email: "john@example.com"
});
```

📌 This inserts a user document into the users collection.

Example: Insert Multiple Documents

```
js

CopyEdit

db.users.insertMany([

  { name: "Alice", age: 25, email: "alice@example.com" },

  { name: "Bob", age: 28, email: "bob@example.com" }

]);
```

📌 This inserts **multiple users** into the collection.

---

2. Read (Query Documents)

MongoDB provides methods to **retrieve data** from collections.

- **findOne()** – Returns **one document** that matches the query.
- **find()** – Returns **multiple documents**.

Example: Find One Document

```
js

CopyEdit

db.users.findOne({ name: "Alice" });
```

📌 Finds **the first user** named **Alice**.

Example: Find All Documents

```
js

CopyEdit

db.users.find({});
```

📌 Returns **all documents** in the users collection.

Example: Find Documents with Filters

```
js

CopyEdit

db.users.find({ age: { $gt: 25 } });
```

📌 Finds users **older than 25** ($gt = greater than).

Example: Find with Projection

js

CopyEdit

```
db.users.find({ age: { $gt: 25 } }, { name: 1, _id: 0 });
```

📌 Returns only **name** field (excluding _id).

---

3. Update (Modify Documents)

MongoDB allows updating documents using:

- **updateOne()** – Updates **the first matching document**.
- **updateMany()** – Updates **multiple documents**.
- **replaceOne()** – Replaces a document.

Example: Update One Document

js

CopyEdit

```
db.users.updateOne(
  { name: "Alice" },
  { $set: { age: 26 } }
);
```

📌 Updates **Alice's age** to **26**.

Example: Update Multiple Documents

js

CopyEdit

```
db.users.updateMany(
  { age: { $lt: 30 } },
  { $set: { status: "young" } }
);
```

📌 Adds a new field **"status": "young"** for users **younger than 30**.

Example: Replace a Document

js

CopyEdit

```
db.users.replaceOne(
  { name: "Alice" },
  { name: "Alice Johnson", age: 27, email: "alicej@example.com" }
);
```

📌 **Replaces** Alice's document with a new structure.

---

4. Delete (Remove Documents)

MongoDB allows deleting documents using:

- **deleteOne()** – Removes **one matching document**.
- **deleteMany()** – Removes **multiple documents**.

Example: Delete One Document

js

CopyEdit

```
db.users.deleteOne({ name: "Bob" });
```

📌 Deletes **the first user** named **Bob**.

Example: Delete Multiple Documents

js

CopyEdit

```
db.users.deleteMany({ age: { $lt: 25 } });
```

📌 Deletes **all users** younger than **25**.

CRUD Operations Using Mongoose (Node.js ORM)

Mongoose is an **ODM (Object Data Modeling) library** for MongoDB in **Node.js**.

1. Install Mongoose

```
npm install mongoose
```

2. Connect to MongoDB

```
const mongoose = require("mongoose");
mongoose.connect("mongodb://localhost:27017/mydatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
});
```

3. Define a Schema and Model

```
const userSchema = new mongoose.Schema({
  name: String,
  age: Number,
  email: String
});
const User = mongoose.model("User", userSchema);
```

4. Perform CRUD Operations

Create (Insert)

```
const newUser = new User({ name: "John", age: 30, email: "john@example.com" });
newUser.save().then(() => console.log("User Created"));
```

Read (Find)

```
User.find({ age: { $gt: 25 } }).then(users => console.log(users));
```

Update

```
User.updateOne({ name: "J" }, { $set: { age: 31 } }).then(() => console.log("User Updated"));
```

Delete

```
User.deleteOne({ name: "J" }).then(() => console.log("User Deleted"));
```

Mongoose for MongoDB Operations

Mongoose is an **ODM (Object Data Modeling) library** for MongoDB in **Node.js**. It provides **schema-based** solutions for organizing and validating data.

1. Installing Mongoose

To use Mongoose, install it via npm: npm install mongoose

2. Connecting to MongoDB

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

mongoose.connect("mongodb://localhost:27017/mydatabase", {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("MongoDB Connected"))
.catch(err => console.log(err));
```

3. Defining a Schema and Model

In Mongoose, data is structured using **Schemas**, which define **the shape and validation** of documents.

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  age: { type: Number, required: true },
  email: { type: String, required: true, unique: true }
});


const User = mongoose.model("User", userSchema);
```

4. CRUD Operations Using Mongoose

4.1. Create (Insert)

To add a new user:

```
 const newUser = new User({
```

```
  name: "John Doe",
  age: 30,
  email: "john@example.com"
});

newUser.save()
  .then(() => console.log("User Created"))
  .catch(err => console.log(err));
```

save() writes the document to MongoDB.

Alternative: create()

```
User.create({
  name: "Alice",
  age: 25,
  email: "alice@example.com"
})
.then(() => console.log("User Created"))
.catch(err => console.log(err));
```

create() is shorthand for new Model().save().

4.2. Read (Find)

Find All Users

```
User.find({})
  .then(users => console.log(users))
  .catch(err => console.log(err));
```

Find One User

```
User.findOne({ name: "Alice" })
  .then(user => console.log(user))
  .catch(err => console.log(err));
```

Find by ID

```
User.findById("65d3b2f5c37c2f9a9b6c98d2")
  .then(user => console.log(user))
  .catch(err => console.log(err));
```

4.3. Update (Modify)

Update One Document

```
User.updateOne({ name: "Alice" }, { $set: { age: 26 } })
  .then(() => console.log("User Updated"))
  .catch(err => console.log(err));
```

Update Multiple Documents

```
User.updateMany({ age: { $lt: 30 } }, { $set: { status: "young" } })
  .then(() => console.log("Users Updated"))
  .catch(err => console.log(err));
```

Find and Update

```
User.findOneAndUpdate(
  { name: "Alice" },
  { age: 27 },
  { new: true } // Returns updated document
)
  .then(user => console.log(user))
  .catch(err => console.log(err));
```

{ new: true } ensures the updated document is returned.

4.4. Delete (Remove)

Delete One User

```
User.deleteOne({ name: "Alice" })
  .then(() => console.log("User Deleted"))
  .catch(err => console.log(err));
```

Delete by ID

```
User.findByIdAndDelete("65d3b2f5c37c2f9a9b6c98d2")
  .then(() => console.log("User Deleted"))
  .catch(err => console.log(err));
```

Delete Multiple Users

```
User.deleteMany({ age: { $lt: 25 } })
  .then(() => console.log("Users Deleted"))
  .catch(err => console.log(err));
```

5. Mongoose Schema Features

5.1. Default Values

```
const userSchema = new mongoose.Schema({
  name: String,
  age: Number,
  email: String,
  createdAt: { type: Date, default: Date.now }
});
```

default: Date.now sets the creation time automatically.

5.2. Schema Methods

Custom methods can be added to the schema.

```
userSchema.methods.getUserInfo = function() {
  return `${this.name} is ${this.age} years old.`;
};
const User = mongoose.model("User", userSchema);
User.findOne({ name: "John" })
  .then(user => console.log(user.getUserInfo()));
```

5.3. Virtual Fields

Virtual fields don’t get stored in the database but are computed on-the-fly.

```
userSchema.virtual("isAdult").get(function() {
```

```
  return this.age >= 18;
});
const User = mongoose.model("User", userSchema);
User.findOne({ name: "Alice" })
  .then(user => console.log(user.isAdult)); // true or false
```

6. Advanced Mongoose Features

6.1. Population (Joining Collections)

You can **reference another collection** using populate().

```
const postSchema = new mongoose.Schema({
  title: String,
  content: String,
  author: { type: mongoose.Schema.Types.ObjectId, ref: "User" }
});
const Post = mongoose.model("Post", postSchema);
// Find post and populate author details
Post.find().populate("author").then(posts => console.log(posts));
```

populate("author") fetches user details instead of just author ID.

6.2. Indexing for Performance

Indexes **speed up queries**.

```
userSchema.index({ email: 1 }, { unique: true });
```

Ensures **fast searches on email**.

7. Using Mongoose with Express.js

```
const express = require("express");
const mongoose = require("mongoose");
const app = express();
app.use(express.json());
```

```
mongoose.connect("mongodb://localhost:27017/mydatabase", { useNewUrlParser: true, useUnifiedTopology: true });

const userSchema = new mongoose.Schema({
  name: String,
  age: Number,
  email: String
});

const User = mongoose.model("User", userSchema);

// Create User
app.post("/users", async (req, res) => {
  const user = new User(req.body);
  await user.save();
  res.send(user);
});

app.get("/users", async (req, res) => {
  const users = await User.find();
  res.send(users);
});

app.put("/users/:id", async (req, res) => {
  const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.send(user);
});

app.delete("/users/:id", async (req, res) => {
  await User.findByIdAndDelete(req.params.id);
  res.send("User deleted");
});

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

What is CORS (Cross-Origin Resource Sharing)

CORS (Cross-Origin Resource Sharing) is a security feature implemented in web browsers that **controls which resources can be requested from another domain**.

Why is CORS Needed?

By default, browsers follow the **Same-Origin Policy (SOP)**, which prevents web pages from making requests to a different domain than the one that served them.

- **Example:** A website http://example.com cannot directly fetch data from http://api.example2.com unless CORS is enabled.

How CORS Works?

1. When a browser makes a cross-origin request, the **server** must send specific headers allowing the request.
2. If the server does not include the correct CORS headers, the browser blocks the request.

Handling CORS in Node.js (Express Example)

To enable CORS in an Express server, install and use the cors package:

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

const cors = require('cors');

const app = express();

app.use(cors()); // Enables CORS for all routes

app.get('/data', (req, res) => {

  res.json({ message: 'CORS is enabled!' });

});

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

What is Axios?

Axios is a **JavaScript library** used to make HTTP requests from the browser or Node.js. It provides a simple and easy-to-use API for fetching data from external sources.

Why Use Axios?

1. Supports **Promises & Async/Await**
2. Automatically transforms JSON responses

3. Handles request **timeouts and errors**
4. Allows **interceptors** for request/response manipulation

Axios GET Request Example

```
import axios from 'axios';

axios.get('https://jsonplaceholder.typicode.com/posts')
  .then(response => console.log(response.data))
  .catch(error => console.error('Error fetching data:', error));
```

Axios POST Request Example

```
axios.post('https://example.com/api', {
    name: 'John',
    email: 'john@example.com'
  })
  .then(response => console.log(response.data))
  .catch(error => console.error(error));
```

3. CORS with Axios

If a request is blocked due to CORS issues, the server must be configured to allow the required origin.

Handling CORS in Axios (Frontend Solution)

You can use a **proxy server** in development to bypass CORS restrictions.

React (package.json) Proxy Setup:

```
"proxy": "http://localhost:5000"
```

- This allows Axios to send requests without triggering CORS errors.

Angular & React Applications with a Node.js Backend

Angular and React are popular **front-end frameworks** used for building **dynamic web applications**, while Node.js is a powerful **JavaScript runtime** commonly used for back-end development.

By combining **Angular/React with a Node.js backend**, you can build **full-stack applications** where:

- The **frontend (React/Angular)** handles UI & user interactions.
- The **backend (Node.js + Express)** processes requests & interacts with the database (MongoDB, MySQL, etc.).

2. Why Use Node.js as a Backend for Angular & React?

1. **Same Language (JavaScript)** – Makes development easier since both frontend & backend use JS.
2. **Asynchronous & Fast** – Handles multiple requests efficiently.
3. **RESTful APIs & Microservices** – Ideal for API-driven applications.
4. **Scalability** – Easily handles real-time applications.
5. **Rich Ecosystem** – Uses npm packages for authentication, security, and performance enhancements.

3. Setting Up a Node.js Backend (Express.js)

Step 1: Install Node.js & Express

mkdir backend

cd backend

npm init -y

npm install express cors body-parser mongoose

- **express** → Web framework for handling API requests
- **cors** → Handles cross-origin requests
- **body-parser** → Parses JSON requests
- **mongoose** → Connects with MongoDB

Step 2: Create a Simple Node.js Server (server.js)

javascript

CopyEdit

const express = require('express');

const cors = require('cors');

const mongoose = require('mongoose');

```
const app = express();
app.use(cors());
app.use(express.json());

// Sample Route
app.get('/', (req, res) => {
  res.send('Node.js Backend Connected!');
});

// Start Server
const PORT = 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

4. Connecting Angular/React with Node.js

(A) React with Node.js Backend

Step 1: Install Axios in React

```
npm install axios
```

Step 2: Fetch Data from Node.js Backend (App.js)

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';

function App() {
  const [message, setMessage] = useState('');
  useEffect(() => {
    axios.get('http://localhost:5000')
      .then(response => setMessage(response.data))
      .catch(error => console.error(error));
  }, []);
  return <h1>{message}</h1>;
```

```
}
export default App;
```

(B) Angular with Node.js Backend

Step 1: Create an Angular Service to Call Node.js API

```
ng generate service api
```

Step 2: Modify api.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private apiUrl = 'http://localhost:5000';
  constructor(private http: HttpClient) {}
      getMessage(): Observable<string> {
    return this.http.get<string>(this.apiUrl);
  }
}
```

Step 3: Use the Service in a Component (app.component.ts)

```
import { Component, OnInit } from '@angular/core';
import { ApiService } from './api.service';
@Component({
  selector: 'app-root',
  template: `<h1>{{ message }}</h1>`,
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
```

```
  message = '';
  constructor(private apiService: ApiService) {}
  ngOnInit(): void {
    this.apiService.getMessage().subscribe(data => {
      this.message = data;
    });
  }
}
```

5. Connecting Node.js to MongoDB

Install MongoDB & Mongoose

```
npm install mongoose
```

Create MongoDB Connection in server.js

```
mongoose.connect('mongodb://localhost:27017/mydatabase', {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log('MongoDB Connected'))
  .catch(err => console.log(err));
```

Define a Model (User.js)

```
const mongoose = require('mongoose');
const UserSchema = new mongoose.Schema({
  name: String,
  email: String
});
module.exports = mongoose.model('User', UserSchema);
```

Create API Endpoint to Fetch Data

```
const User = require('./User');
app.get('/users', async (req, res) => {
  const users = await User.find();
```

```
  res.json(users);
});
```