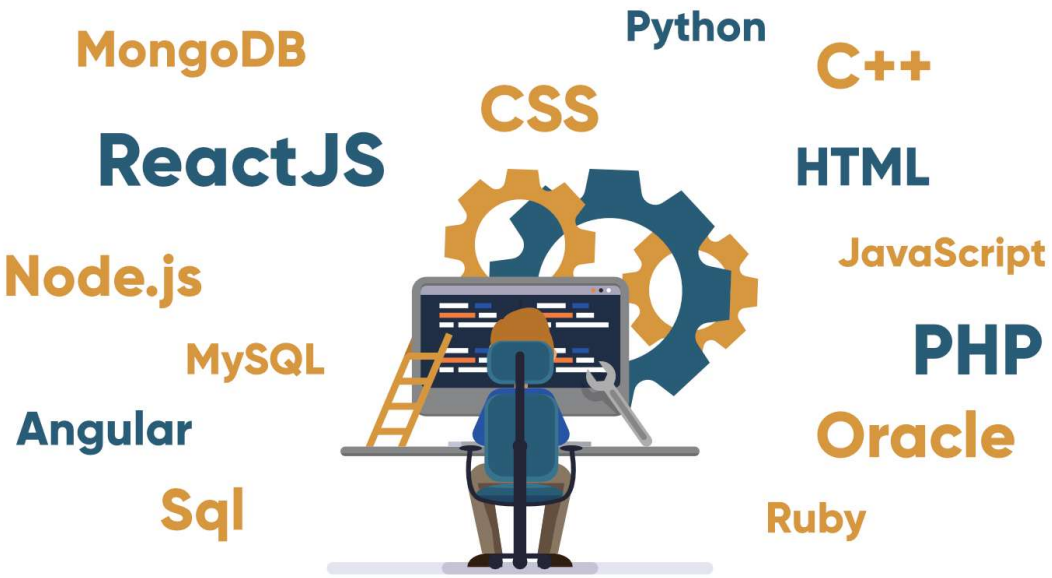
FULL STACK DEVELOPMENT

What is Full Stack Development ?
Full Stack Development refers to the development of both **front end** (client side) and **back end** (server side) portions of web applications.

Full Stack Web Developers
Full Stack web developers have the ability to design complete web applications and websites. They work on the frontend, backend, database, and debugging of web applications or websites.



Technology Related to Full Stack Development

Front-end Development
It is the visible part of website or web application which is responsible for user experience. The user directly interacts with the front-end portion of the web application or website.

Front End Libraries and Frameworks:
AngularJS, React.js, Bootstrap, jQuery

Back-end Technologies
It refers to the server-side development of web application or website with a primary focus on how the website works. It is responsible for managing the database through queries and APIs by client-side commands.

Backend technologies include programming languages like Python, Java, PHP, and Node.js; frameworks like Django, Spring Boot, and Ruby on Rails; databases such as MySQL, PostgreSQL, and MongoDB; and tools for API development, version control, and cloud platforms.

Popular Stacks

- **MERN Stack:** MongoDB, Express, ReactJS and Node.js
- **MEAN Stack:** MongoDB, Express, AngularJS and Node.js.

PROGRAM – 1

(NODE.JS)

Introduction:

Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to execute JavaScript code outside of a web browser. Traditionally, JavaScript was confined to client-side web development within browsers. Node.js extends the reach of JavaScript, enabling its use for server-side development, command-line tools, and various other applications.

Key characteristics and features of Node.js include:

- **JavaScript Runtime:**

It utilizes Google Chrome's V8 JavaScript engine, which is known for its high performance in executing JavaScript code.

- **Asynchronous and Event-Driven:**

Node.js operates on an event-driven, non-blocking I/O model. This allows it to handle many concurrent connections efficiently without creating a new thread for each request, making it highly scalable for I/O-bound operations.

- **Single-Threaded Event Loop:**

While Node.js itself is single-threaded, it uses an event loop to manage asynchronous operations, preventing the main thread from blocking while waiting for I/O operations to complete.

- **Versatility:**

Node.js is widely used for building various types of applications, including:

Web servers and APIs, Real-time applications (e.g., chat applications, online gaming), Command-line tools, Microservices, IoT applications

Note: Node.js empowers developers to use JavaScript for full-stack development, leveraging a single language for both client-side and server-side logic.

Node.js Installation procedure:

1. Download the installer:
Go to the official Node.js website: **Node.js**
Download the installer for your operating system (Windows, macOS, or Linux).
Choose either the LTS (Long Term Support) or Current version. LTS is generally recommended for most users.
2. Run the installer:
Locate the downloaded installer file (usually an .msi file for Windows or a .pkg file for macOS).
Double-click the file to start the installation process.
Follow the on-screen instructions, accepting the license agreement and choosing a destination folder if needed.
3. Verify the installation:
Open your terminal or command prompt.
Type node -v and press Enter. This should display the installed Node.js version.
Type npm -v and press Enter. This should display the installed npm (Node Package Manager) version.

a. AIM: Write a program to show the workflow of JavaScript code executable by creating web server in Node.js.

Introduction:

When you view a webpage in your browser, you are making a request to another computer on the internet, which then provides you the webpage as a response. That computer you are talking to via the internet is a *web server*. A web server receives HTTP requests from a client, like your browser, and provides an HTTP response, like an HTML page or JSON from an API.

Step 1: Create folder with any name like project
Step 2: Create new text file and rename with file_name.js like server.js and file open with note pad and paste script below

 # CODE

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


const hostname = '127.0.0.1';
const port = 3000;


const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});


server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Step 4: Open command prompt and type cd path copied like

cd C:\Users\fsd\project

Step 5: run the server by using node file_name.js like node server.js

It shows like this **Server running at http://127.0.0.1:3000/**

Step 6: Open any browser and paste **http://127.0.0.1:3000/** then it displays

Hello, World!

b. Write a program to transfer data over http protocol using http module.

Key Concepts:

- **HTTP module** is used to create a server that listens for incoming HTTP requests.
- **Data transfer** happens using res.end(data) where data is sent to the client.
- Content-Type is set to application/json for structured data.

Node.js program to **transfer data over the HTTP protocol** using the built-in http module.

#CODE

transferData.js

```
// Load the http module
const http = require('http');

// Define the port for the server
const PORT = 4000;

// Sample data to transfer
const data = {
    id: 101,
    name: "HTTP Data Transfer",
    status: "Success",
    description: "This data is sent over HTTP using Node.js http module"
};

// Create the server
const server = http.createServer((req, res) => {
    console.log(`Request received: ${req.method} ${req.url}`);

    // Set response headers
    res.writeHead(200, {
        'Content-Type': 'application/json',
        'Access-Control-Allow-Origin': '*'
    });
```

```
  // Send the data as JSON
   res.end(JSON.stringify(data));
});

// Start the server
server.listen(PORT, () => {
   console.log(`Server running at http://localhost:${PORT}/`);
});
```

How to Run:

1. Save the code in a file named transferData.js.
2. Open terminal and run: node transferData.js
3. Open your browser and go to:
   http://localhost:4000

OUTPUT:

```
{
 "id": 101,
 "name": "HTTP Data Transfer",
 "status": "Success",
 "description": "This data is sent over HTTP using Node.js http module"
}
```

c. Create a text file src.txt and add the following content to it. (HTML, CSS, Javascript, Typescript, MongoDB, Express.js, React.js, Node.js)

To create a text file named src.txt and add the specified content to it, follow these steps using **Node.js**:

createFile.js

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

// Define the file name and content
const fileName = 'src.txt';
const content = 'HTML, CSS, Javascript, Typescript, MongoDB, Express.js, React.js, Node.js';

// Create the file and write the content
fs.writeFile(path.join(__dirname, fileName), content, (err) => {
  if (err) {
    console.error('Error writing file:', err);
    return;
  }
  console.log(`${fileName} created successfully with content.`);
});
```

Steps to Run:

1. Save the above code in a file named createFile.js.
2. Open your terminal or command prompt.
3. Run the script using: **node createFile.js**
4. **It will create src.txt in the same directory with the following contents:**

src.txt (Output Content)

HTML

CSS

JavaScript

TypeScript

MongoDB

Express.js

React.js

Node.js

PROGRAM-2
(TypeScript)

- **TypeScript is a syntactic superset of JavaScript which adds static typing** developed by Microsoft.
- TypeScript is a programming language that's a superset of JavaScript, which means it understands all of JavaScript's syntax and capabilities, while adding additional features.
- JavaScript is a loosely typed language. It can be difficult to understand what types of data are being passed around in JavaScript.
- TypeScript allows specifying the types of data being passed around within the code, and has the ability to report errors when the types don't match.
- TypeScript uses compile time type checking. Which means it checks if the specified types match **before** running the code, not **while** running the code and if there are type errors it will not successfully compile, which prevents bad code from going out to be run.

How Does TypeScript Work?

- Browsers and most other platforms like node that run JavaScript do not have native support for TypeScript. TypeScript comes with a **compiler called tsc** which will convert them into JavaScript so that the browser (or whatever platform you're using) will be able to run your code as if it was JavaScript.

Install TypeScript Compiler (tsc):

✅ **Step 1: Install Node.js (if not already installed)**

1. Go to: https://nodejs.org
2. Download and install the **LTS version** (Long-Term Support)
3. After installation, open **Command Prompt** and check:
   node -v
   npm -v

✅ **Step 2: Install TypeScript globally**

Run this in Command Prompt:

npm install -g typescript

The -g flag installs it **globally** so you can run tsc from any folder.

✅ **Step 3: Verify the installation**

Run this to check if tsc is working:

tsc -v

✅ **Step 4: Compile and Run TypeScript File**

1. Save your code in a file like **types.ts**
2. Compile it**:** **tsc types.ts**
   It will generate a types.js file.
3. Run it with Node.js: **node types.js**

a. Write a Typescript program to understand simple and special types.

✅ **Simple Types:**
- number, string, boolean

✅ **Special Types:**
- any, unknown, void, null, undefined, never

#PROGRAM CODE

```
// Simple Types
let myName: string = "Alice";       // string type
let myAge: number = 30;             // number type
let isStudent: boolean = true;      // boolean type

// Output simple types
console.log("Name:", myName);
console.log("Age:", myAge);
console.log("Is Student:", isStudent);

// Special Types
let notKnown: any = "Hello";        // any type (can be anything)
notKnown = 100;                     // can change to number
console.log("Any type value:", notKnown);

let unknownValue: unknown = "I might be anything";
// console.log(unknownValue.toUpperCase()); // ❌ Error without type check
if (typeof unknownValue === "string") {
  console.log("Unknown as string:", unknownValue.toUpperCase());
}

// null and undefined
let nothingHere: null = null;
let notAssigned: undefined = undefined;
console.log("Null value:", nothingHere);
console.log("Undefined value:", notAssigned);
```

```
// void type - for functions that return nothing
function greet(): void {
  console.log("Hello from a void function!");
}
greet();

// never type - for functions that never return
function throwError(message: string): never {
  throw new Error(message);
}
// throwError("Something went wrong!"); // Uncomment to test
```

Steps to Run:

1. Save your code in a file like types.ts
2. Compile it**:** **tsc types.ts**
   It will generate a types.js file.
3. Run it with Node.js: **node types.js**

OUTPUT:

```
Name: Alice
Age: 30
Is Student: true
Any type value: 100
Unknown as string: I MIGHT BE ANYTHING
Null value: null
Undefined value: undefined
Hello from a void function!
```

b) Write a program to understand function parameter and return types.

Function Parameter Types

When you define a function in TypeScript, you can declare the type of each parameter the function accepts. This ensures that the function is used correctly by enforcing type rules at compile time.

Function Return Types

TypeScript lets you **explicitly specify the return type** of a function. This helps prevent bugs by ensuring that the function returns the expected type.

Optional Parameters

Sometimes, not all parameters are required. You can mark a parameter as **optional** using the ? symbol.

Default Parameters

You can also assign default values to parameters.

Arrow Functions with Types

Arrow functions also support type annotations.

Benefits of Using Function Types

- Prevents bugs by catching type mismatches
- Improves code readability and documentation
- Enables better autocomplete and error checking in IDEs

#PROGRAM CODE

```
// Function with number parameters and number return type
function add(a: number, b: number): number {
  return a + b;
}

// Function with string parameters and string return type
function greet(name: string): string {
  return `Hello, ${name}!`;
}
// Function with boolean parameter and void return type
function checkStatus(isActive: boolean): void {
  if (isActive) {
    console.log("Status: Active");
  } else {
    console.log("Status: Inactive");
  }
}
```

```
// Function with optional parameter
function multiply(a: number, b?: number): number {
  return b ? a * b : a * 1; // default to 1 if b is undefined
}

// Function with default parameter
function power(base: number, exponent: number = 2): number {
  return base ** exponent;
}

// Calling the functions
console.log("Add:", add(10, 20));
console.log(greet("Alice"));
checkStatus(true);
console.log("Multiply:", multiply(5));
console.log("Power:", power(3));
```

Steps to Run:

1. Save your code in a file like 2b.ts
2. Compile it**:** **tsc 2b.ts**
   It will generate a types.js file.
3. Run it with Node.js: **node 2b.js**

Output:

Add: 30

Hello, Alice!

Status: Active

Multiply: 5

C) Write a program to show the importance with Arrow function. Use optional, default and REST parameters.

ARROW function

An **arrow function** in JavaScript is a shorter way to write function expressions. It was introduced in ES6 (ECMAScript 2015) and is especially useful for writing concise callbacks and anonymous functions.

Basic Syntax:

```
const add = (a, b) => a + b;
```

- **Features of Arrow Functions:**
  - ✓ **Concise syntax** — cleaner and shorter.
  - ✓ **Does not have its own this** — it captures this from the surrounding context.
  - ✓ **Cannot be used as constructors** — new with arrow function throws an error.
  - ✓ **Does not have arguments object** — must use rest parameters instead.

Examples:

✅ **1. Simple Arrow Function**

```
const greet = () => console.log("Hello, world!");
greet();
```

✅ **2. Arrow Function with Parameters**

```
const multiply = (x, y) => x * y;
console.log(multiply(4, 5)); // Output: 20
```

✅ **3. With Default Parameter**

```
const welcome = (name = "Guest") => `Welcome, ${name}!`;
console.log(welcome());          // Output: Welcome, Guest!
console.log(welcome("AIML"));  // Output: Welcome, AIML!
```

✅ **4. With Optional Parameter (using undefined)**

```
const showMessage = (message, sender = "Admin") => {
  console.log(`${sender}: ${message}`);
};
showMessage("This is a test"); // Admin: This is a test
```

✅ **5. With Rest Parameters**

```
const sumAll = (...nums) => nums.reduce((acc, val) => acc + val, 0);
console.log(sumAll(1, 2, 3, 4)); // Output: 10
```

What are Rest Parameters in JavaScript?

Rest parameters allow a function to accept any number of arguments as an array. It's a way to represent "the rest" of the arguments using the ... syntax.

Syntax:

```
function myFunction(...args) {
  // args is an array
}
```

In arrow functions:

```
const myFunction = (...args) => {
  console.log(args);
};
```

 # PROGRAM CODE:

```
//  Arrow function with default parameters
const greet = (name: string = "Guest"): string => {
  return `Hello, ${name}!`;
};

// Arrow function with optional parameter
const fullName = (firstName: string, lastName?: string): string => {
  return lastName ? `${firstName} ${lastName}` : firstName;
};

//  Arrow function with rest parameters
const sum = (...numbers: number[]): number => {
  return numbers.reduce((total, num) => total + num, 0);
};

//  Arrow function assigned to a variable (shows reusability and compact syntax)
const square = (n: number): number => n * n;

//  Using all functions
console.log(greet()); // default name
console.log(greet("Alice"));

console.log(fullName("John"));            // optional last name
console.log(fullName("John", "Doe"));     // with last name

console.log("Sum:", sum(1, 2, 3, 4, 5));    // REST parameters
console.log("Square of 6:", square(6));
```

Steps to Run:

1. Save your code in a file like 2C.ts
2. Compile it**:** **tsc 2C.ts**
   It will generate a types.js file.

3. Run it with Node.js: **node 2C.js**

OUTPUT:

```
 Hello, Guest!
Hello, Alice!
John
John Doe
Sum: 15
Square of 6: 36
```

d. Write a program to understand the working of typescript with class, constructor, properties, methods and access specifiers.

TypeScript works with **class**, **constructor**, **properties**, **methods**, and **access specifiers** — all concepts from object-oriented programming.

Concept	Purpose
class	Defines a blueprint for objects
constructor	Initializes the object properties
property	Holds data related to the object
method	Performs actions (functions inside the class)
public	Can be accessed anywhere
private	Can only be accessed inside the class
protected	Can be accessed inside the class and subclasses

#PROGRAM CODE

```
class Person {
  // Properties with access specifiers
  public name: string;
  private age: number;
  protected gender: string;

  // Constructor to initialize properties
  constructor(name: string, age: number, gender: string) {
    this.name = name;
    this.age = age;
    this.gender = gender;
  }

  // Public method (accessible from outside)
  public greet(): void {
    console.log(`Hello, my name is ${this.name}.`);
  }
  // Private method (accessible only inside class)
  private showAge(): void {
    console.log(`I am ${this.age} years old.`);
  }
```

```
  // Protected method (accessible in subclass)
  protected displayGender(): void {
    console.log(`Gender: ${this.gender}`);
  }
  // Method to access private method
  public displayDetails(): void {
    this.showAge(); // private method
    this.displayGender(); // protected method
  }
}
// Subclass that inherits Person
class Student extends Person {
  private studentId: number;

  constructor(name: string, age: number, gender: string, studentId: number) {
    super(name, age, gender);
    this.studentId = studentId;
  }

  public showStudentInfo(): void {
    console.log(`Student ID: ${this.studentId}`);
    this.displayGender(); // Can access protected method
  }
}
// Using the class
const person1 = new Person("Alice", 30, "Female");
person1.greet();
person1.displayDetails();
// person1.showAge(); â Œ Error: private method
// person1.displayGender(); â Œ Error: protected method

const student1 = new Student("Bob", 20, "Male", 101);
student1.greet();
student1.displayDetails();
student1.showStudentInfo();
```

Steps to Run:

1. Save your code in a file like 2d.ts
2. Compile it**:** **tsc 2d.ts**
   It will generate a types.js file.

3. Run it with Node.js: **node 2d.js**

OUTPUT:

Hello, my name is Alice.

I am 30 years old.

Gender: Female

Hello, my name is Bob.

I am 20 years old.

Gender: Male

Student ID: 101

Gender: Male

e) Write a program to understand the working of namespaces and modules.

- **Namespaces** are good for organizing code in one file.
- **Modules** are file-based and preferred for modern projects, especially with tools like Webpack, ESBuild, etc.

 # PROGRAM CODE

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

  export function subtract(a: number, b: number): number {
    return a - b;
  }
}

// Using the namespace
console.log("Namespace Add:", MathUtils.add(10, 5));
console.log("Namespace Subtract:", MathUtils.subtract(10, 5));
```

Steps to Run:

1. Save your code in a file like 2e.ts
2. Compile it**:** **tsc 2e.ts**
   It will generate a types.js file.
3. Run it with Node.js: **node 2e.js**

OUTPUT:

```
Namespace Add: 15
Namespace Subtract: 5
```

f. Write a program to understand generics with variables, functions, and constraints.

What are Generics?

Generics in TypeScript — a powerful feature that allows you to write reusable, type-safe code.

Generics enable writing code that works with **any data type**, while still preserving **type safety**.

- **Syntax:**

```
function identity<T>(arg: T): T {
  return arg;
}
```

Here, T is a placeholder for a **type** that will be provided later.

Use Case	Example
Generic Function	function<T>(arg: T): T {}
Generic Variable	let val: <T>(x: T) => T;
Constraint	<T extends { length: number }>
Generic Interface	interface Box<T> { content: T; }
Generic Class	class Container<T> {}

#PROGRAM CODE

```
// 1. Generic Variable
let genericValue: <T>(value: T) => T = function<T>(value: T): T {
  return value;
};

console.log("Generic Variable (number):", genericValue<number>(42));
console.log("Generic Variable (string):", genericValue<string>("Hello"));


// 2. Generic Function
function identity<T>(arg: T): T {
  return arg;
}
console.log("Generic Function (boolean):", identity<boolean>(true));
console.log("Generic Function (array):", identity<number[]>([1, 2, 3]));


//  3. Generic Function with Multiple Types
function pair<A, B>(first: A, second: B): [A, B] {
  return [first, second];
}
console.log("Generic Pair:", pair<string, number>("Age", 25));
```

```
// 4. Generic with Constraint
interface HasLength {
  length: number;
}

function printLength<T extends HasLength>(item: T): void {
  console.log("Length is:", item.length);
}

printLength("TypeScript");            // string has length
printLength([1, 2, 3, 4]);            // array has length
// printLength(123); â□Œ Error: number doesn't have 'length'


// 5. Generic Class
class Box<T> {
  private _value: T;

  constructor(value: T) {
    this._value = value;
  }

  getValue(): T {
    return this._value;
  }
}

const stringBox = new Box<string>("Generic Box");
console.log("Box Value:", stringBox.getValue());
```

Steps to Run:

1. Save your code in a file like 2F.ts
2. Compile it**:** **tsc 2F.ts**
   It will generate a types.js file.

3. Run it with Node.js: **node 2F.js**

OUTPUT:

```
Generic Variable (number): 42
Generic Variable (string): Hello
Generic Function (boolean): true
Generic Function (array): [ 1, 2, 3 ]
Generic Pair: [ 'Age', 25 ]
Length is: 10
Length is: 4
```

PROGRAM – 3
ExpressJS – Routing, HTTP Methods, Middleware.

a) AIM: Write a program to define a route, Handling Routes, Route Parameters, Query Parameters and URL building.

Express.js is a **web application framework for Node.js** that helps you build:

- Web servers
- APIs (Application Programming Interfaces)
- Web apps

Express.js is a minimal and flexible Node.js web application framework that provides a list of features for building web and mobile applications easily. It simplifies the development of server-side applications by offering an easy-to-use API for routing, middleware, and HTTP utilities by providing a robust set of features for handling HTTP requests and responses.

Why Choose Express.js?

Express provides a thin layer of fundamental web application features without effecting Node.js features. It offers:

- A robust routing system
- It simplifies building web servers and APIs.
- Integrates seamlessly with Node.js.
- Offers extensive middleware support to respond to HTTP requests
- A templating engine for dynamic HTML rendering
- Error handling middleware
- Ideal for single-page applications and RESTful APIs.

Definitions & Explanations

Term	Explanation
Route	A route is a URL pattern (like /home, /about) defined in Express to respond to HTTP requests like GET, POST, etc.
Handling Routes	This means writing code that determines what happens when a user visits a specific URL (route).
Route Parameters	Parts of the URL defined using :paramName. Used to pass dynamic values in the URL. Example: /user/:id.
Query Parameters	Key-value pairs sent in the URL after a ?. Example: /search?keyword=nodejs. Used to pass extra data.
URL Building	Dynamically generating a URL by combining base paths, route/query parameters for linking or redirection.

 # PROGRAM CODE

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

// Define a basic route
app.get('/', (req, res) => {
  res.send('Hello! This is the home page.');
});

// Handling a route with a parameter
app.get('/user/:name', (req, res) => {
  const userName = req.params.name;
  res.send(`Hello, ${userName}!`);
});

// Handling a route with query parameters
app.get('/search', (req, res) => {
  const keyword = req.query.q;
  res.send(`You searched for: ${keyword}`);
});

// URL building example
app.get('/url', (req, res) => {
  const fullUrl = req.protocol + '://' + req.get('host') + req.originalUrl;
  res.send(`Full URL: ${fullUrl}`);
});

// Start server
app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

COMPILATION & EXECUTION:

To compile and execute the above **Express.js** program, follow these simple steps:

☑ **Step-by-Step Instructions**

1. Install Node.js

- Download and install from: https://nodejs.org
- Verify installation in terminal/command prompt:

```
node -v
npm -v
```

2. Create Your Project Folder
mkdir my-express-app
cd my-express-app

3. Initialize the Project
npm init -y
This creates a package.json file.

4. Install Express
npm install express

5. Create the File
Create a file named 3A.js:
notepad app.js # (Windows)

6. Run the Program
node app.js
☑ You'll see:
Server is running on http://localhost:3000

7. Test It in the Browser

Open your browser and try:

Try These URLs in Browser

URL	What It Does
http://localhost:3000/	Shows "Welcome to the Home Page"
http://localhost:3000/user/AIML	Shows "Hello, AIML"
http://localhost:3000/search?q=express	Shows "You searched for: express"
http://localhost:3000/url	It shows the full URL

b. AIM: Write a program to accept data, retrieve data and delete a specified resource using http methods.

When working with HTTP methods in web applications (like RESTful APIs), we use different methods to interact with resources (data).

Here's how accepting, retrieving, and deleting data works:

1. Accept Data → POST

- Method: POST
- Purpose: Used to send/accept data from the client to the server.
- Example: Submitting a new student record to the database.
- Flow:
    - Client → sends data (e.g., { "name": "Ravi", "rollno": 10 })
    - Server → accepts and stores the data, returns confirmation or the newly created resource.

☑ POST = Create new resource

2. Retrieve Data → GET

- Method: GET
- Purpose: Used to retrieve/fetch data from the server.
- Example: Fetch all student records or get details of a student by ID.
- Flow:
    - Client → sends GET /students
    - Server → responds with data (list of students in JSON, HTML, etc.).

☑ GET = Read resource

3. Delete a Specific Resource → DELETE

- Method: DELETE
- Purpose: Used to remove a specific resource from the server.
- Example: Delete student with ID 10.
- Flow:
    - Client → sends DELETE /students/10
    - Server → removes student with ID 10 and returns success message.

☑ DELETE = Remove resource

What is Postman?
Postman is a popular API (Application Programming Interface) testing tool used by developers, testers, and backend engineers to send, receive, and analyze HTTP requests and responses.

☑ **Why is Postman Used?**

Purpose	Explanation
1. Test APIs easily	You can send GET, POST, PUT, DELETE, etc., requests to any API and see the response instantly.
2. Debug backend issues	Helps find and fix issues in API response, parameters, headers, etc.
3. No code required	You don't need to write a program—just fill in the API URL, method, headers, and body.
4. Save & share requests	You can save your request collections and share with team members.
5. Automate testing	Write pre-request scripts and test cases using JavaScript inside Postman.

 # PROGRAM CODE

```
const express = require('express');
const app = express();
const port = 3000;

// Middleware to parse JSON
app.use(express.json());

// In-memory data storage
let resources = [];

// Route: GET all resources
app.get('/resources', (req, res) => {
  res.json(resources);
});

// Route: POST to add a new resource
app.post('/resources', (req, res) => {
  const resource = req.body;
if (!resource.id || !resource.name) {
    return res.status(400).json({ error: 'Resource must have id and name' });
  }
  // Check for duplicate ID
  const exists = resources.some(r => r.id === resource.id);
  if (exists) {
    return res.status(409).json({ error: 'Resource with this ID already exists' });
  }
  resources.push(resource);
  res.status(201).json({ message: 'Resource added successfully', resource });
});

// Route: DELETE a resource by ID
app.delete('/resources/:id', (req, res) => {
  const id = req.params.id;
```

```
  const index = resources.findIndex(r => r.id === id);

  if (index === -1) {
    return res.status(404).json({ error: 'Resource not found' });
  }

  const deleted = resources.splice(index, 1);
  res.json({ message: 'Resource deleted successfully', deleted });
});

// Start server
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```

Working with Postman

Using Postman (Graphical Tool)
🔧 Setup
Download & install Postman from https://www.postman.com/downloads

☑ A. POST – Add Data

➕ Add Resource (POST)
URL:http://localhost:3000/resources

Method: POST
Body:
Go to the "Body" tab
Select raw and choose JSON
Enter this JSON:

```
{
  "id": "1",
  "name": "Book"
}
```

Click"Send"
→ Should respond: Item added.

☑ B. GET – Retrieve Data
Method: GET
URL: http://localhost:3000/resources
Click"Send"
→ You'll get the full list of items.
http://localhost:3000/resources---
{copy this URL in browser & see the result}

☑ C. DELETE – Delete by Name
Delete Resource (DELETE)
URL:http://localhost:3000/resources/1
Method: DELETE
Click"Send"
→ Should respond: Item with name 'pen' deleted.

c. AIM: Write a program to show the working of middleware.

What is Middleware in Express?

Middleware functions are functions that have access to:

- req (request)
- res (response)
- next (function to move to the next middleware)

They run **before** the final route handler and are used for:

- Logging
- Request validation
- Authentication
- Error handling

```
const express = require('express');
const app = express();
const port = 3000;

// ----------------------
// ☑ Middleware Function
// ----------------------
const loggerMiddleware = (req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next(); // Go to next middleware or route handler
};

// -------------------------
// ☑ Apply Middleware
// -------------------------
app.use(loggerMiddleware); // Applies to all routes

// -------------------------
// ☑ Route Handlers
// -------------------------
app.get('/', (req, res) => {
  res.send('Hello, this is the home page!');
});

app.get('/about', (req, res) => {
  res.send('This is the about page.');
});

// -------------------------
// ☑ Start Server
// -------------------------
app.listen(port, () => {
  console.log(`Server is running at http://localhost:${port}`);
});
```

What Happens Here?

- app.use(loggerMiddleware) — applies the logger to all routes.
- Every time you hit a route (like / or /about), the middleware logs the request method and URL before responding.

Test in Browser

Open a browser

- **http://localhost:3000/** → Output: "Hello, this is the home page!"
- **http://localhost:3000/about** → Output: "This is the about page."

PROGRAM – 4
ExpressJS – Templating, Form Data

a. AIM: Write a program using templating engine.

What is a Template Engine?

A **Template Engine** is a tool that helps you **build HTML pages dynamically** using your data. Instead of writing static HTML files manually, a template engine lets you:

- Insert variables (like user names, messages)
- Use logic (like if, for loops)
- Separate your **HTML layout** from your **JavaScript code**

👉 This makes your web app easier to build and maintain.

What is EJS?

EJS stands for **Embedded JavaScript**.

It is a **popular template engine** for Node.js and Express that lets you:

- Embed JavaScript code **inside HTML**
- Generate dynamic web pages on the server

☑ **Features of EJS:**

- Easy to use
- Supports loops and conditionals
- Clean syntax like <%= %> for inserting data

How it Works in Express:

1. You set EJS as the view engine:

   app.set('view engine', 'ejs');

2. Place .ejs files in the /views folder.
3. Use res.render('filename', data) to render the template with dynamic data.

   In web development, **`render`** means:

   🟤 **To generate and send a final HTML page** to the client (browser), often using a template and dynamic data.

   Example: index.ejs

Step 1: Check Your Views Directory Structure

Make sure you have this folder and file:
C:\Users\ksbam\my-express-app\views\index.ejs

- ☑ views folder **must exist**
- ☑ index.ejs file **must be inside** the views folder

If you're using .ejs, the filename should be: **index.ejs**

```
<!DOCTYPE html>
<html>
<head>
  <title>Home</title>
</head>
<body>
  <h1>Hello from EJS!</h1>
</body>
</html>
```

Step 2: Check That EJS Is Installed
Run this command in your project directory:

```
npm init -y
npm install ejs
```

◇ **Step 3: Set the View Engine in 4A.js**
Make sure you have the correct configuration at the top of your 4A.js:

☑ **full code for the program --- 4a.js**

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

app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));

app.get('/', (req, res) => {
  res.render('index');
});

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

Run the app:
node 4a.js
Open in your browser :
http://localhost:3000

OUTPUT: "Hello from EJS"

B) AIM: Write a program to work with form data.

Step 1: Create a views folder

Make sure the directory C:\Users\ksbam\my-express-app\views exists.
If it doesn't:

1. Open your project folder.
2. Create a folder named views.

☑ **Step 2: Create the form view file**
Inside the views folder, add a file named:

- If you're using **EJS**: form.ejs

views/form.ejs:

```
<!DOCTYPE html>
<html>
<head>
  <title>Form</title>
</head>
<body>
  <h1>Sample Form</h1>
  <form method="post">
    <input type="text" name="username" placeholder="Enter name" />
    <button type="submit">Submit</button>
  </form>

  <% if (username) { %>
    <h2>You entered: <%= username %></h2>
  <% } %>
</body>
</html>
```

☑ **Step 3: Check your app settings**
Ensure your Express app sets the views and view engine correctly in 4b.js:

4b.js

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

app.use(bodyParser.urlencoded({ extended: true }));
app.set('view engine', 'ejs');

app.get('/', (req, res) => res.render('form', { username: null }));

app.post('/', (req, res) => res.render('form', { username: req.body.username }));

app.listen(3000, () => console.log('Server running at http://localhost:3000'));
```

☑ **Step 4: Install EJS if not installed**
Type the following commands in command prompt

npm init -y
npm install ejs
npm install express ejs body-parser

☑ **Step 5: Run the app**

Start your server:
node 4b.js
Then visit:
👉 **http://localhost:3000**

You should see your form.

OUTPUT:

Sample Form

Enter name Submit

Sample Form

AIML Submit

Sample Form

Enter name Submit

You entered: AIML

PROGRAM – 5

ExpressJS – Cookies, Sessions, Authentication

a. AIM: Write a program for session management using cookies and sessions.

What Is Session Management?

Session management is the process of **tracking user interactions** with a web app across multiple requests.

For example:
When a user logs in, the server remembers them for future visits during that session.

❁ Core Concepts

☑ 1. Cookies

- Cookies are small pieces of data stored in the **user's browser**.
- Sent automatically with every request to the server.
- Used to identify users or store small data (like username or session ID).

Example:

```
res.cookie('username', 'AIML');
```

☑ 2. Sessions

- A session is stored **on the server**.
- The client only stores the **session ID** in a cookie.
- Sessions are **safer** than storing everything in cookies.

Example:

```
req.session.username = 'AIML;
```

🔐 Authentication Flow Using Cookies & Sessions

1. User logs in

- Client sends username/password to the server.
- Server verifies the credentials.
- If correct:
  - Server creates a session.
  - Session ID is stored in a cookie and sent to the client.

2. User visits protected pages

- On every request, the cookie with session ID is sent.
- Server uses that session ID to retrieve session data.
- If session is valid, user is allowed access.

3. User logs out

- Server deletes the session.
- Cookie is cleared or expired.

How ExpressJS Handles This

Middleware used:

1. **express-session** → handles session creation and tracking
2. **cookie-parser** → parses cookies from incoming requests
3. **body-parser** → parses form data (for login forms)

Benefits of This Approach

- **Security**: Sensitive data is stored on the server.
- **Scalability**: Sessions can be stored in memory, Redis, or a database.
- **Convenience**: Automatically handles login state across multiple pages.

ExpressJS program with:

- ☑ Session creation on login
- ☑ Accessing session data
- ☑ Logout and session destroy

#PROGRAM CODE

☑5A .js

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

app.use(session({
  secret: 'abc',
  resave: false,
  saveUninitialized: true
}));
// Login and set session
app.get('/login', (req, res) => {
  req.session.username = 'Ambika';
  res.send('User logged in');
});

// Access protected page
app.get('/profile', (req, res) => {
  if (req.session.username) {
    res.send(`Welcome ${req.session.username}`);
  } else {
    res.send('Please login first');
  }
});
```

// Logout and destroy session

```
app.get('/logout', (req, res) => {
  req.session.destroy();
  res.send('User logged out');
});
app.listen(3000, () => {
  console.log('http://localhost:3000');
});
```

How to Use Execute :

Steps:

1. **node -v**
2. **npm -v**

Setup Instructions

First, install dependencies:

1. **npm list express**
2. **npm install express express-session**
3. **npm init -y**

Check if express-session is installed locally:

npm list express-session

1. Visit http://localhost:3000/**login → Session created**
2. Visit http://localhost:3000/**profile → Shows username**
3. Visit http://localhost:3000/**logout → Session destroyed**
4. Visit /profile again → Asks to login again

b. AIM: Write a program for user authentication.

ExpressJS Authentication Flow (using Sessions and Cookies)

☑ **Step-by-Step Flow:**

1. User opens login page
2. User enters username and password
3. Server checks credentials
4. **If correct:**
    - Server creates a **session**
    - Stores user info in session (on the server)
    - Sends a **cookie** with a session ID to the browser
5. **For future requests:**
    - Cookie is sent automatically
    - Server checks session ID
    - If valid, user is considered logged in

❁ **Key Tools in ExpressJS:**

Tool	**Purpose**
express-session	Creates and manages sessions
cookie-parser	Parses cookies from HTTP requests
body-parser	Parses form data (POST requests)

 # PROGRAM CODE

```
const express = require('express');
const bodyParser = require('body-parser');
const session = require('express-session');

const app = express();
const PORT = 3000;

// Middleware
app.use(bodyParser.urlencoded({ extended: true }));
app.use(session({
```

```
  secret: 'secret-key',
  resave: false,
  saveUninitialized: true
}));

// Dummy user credentials
const USER = {
  username: 'admin',
  password: '1234'
};

// Login page
app.get('/', (req, res) => {
  res.send(`
    <h2>Login</h2>
    <form method="POST" action="/login">
      <input type="text" name="username" placeholder="Username" required/><br/>
      <input type="password" name="password" placeholder="Password" required/><br/>
      <button type="submit">Login</button>
    </form>
  `);
});

// Login handler
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  if (username === USER.username && password === USER.password) {
    req.session.user = username;
    res.send(`<h2>Welcome, ${username}!</h2><a href="/logout">Logout</a>`);
  } else {
    res.send('<h3>Invalid credentials. <a href="/">Try again</a></h3>');
  }
});
```

```
// Logout handler
app.get('/logout', (req, res) => {
  req.session.destroy();
  res.redirect('/');
});

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

Execution Steps:

1. **node -v**
2. **npm -v**

Setup Instructions

First, install dependencies:

3. **npm list express** (if version number is shown as empty u have to install express) i.e, --→ npm install express
4. **npm install express ejs body-parser**
5. **npm install express express-session**
6. **npm init -y**

Test with Browser:

1. Visit http://localhost:3000 to see the login form.
2. Enter:
   - **Username**: admin
   - **Password**: 1234
3. You'll be logged in with a welcome message.

Note: If wrong user name & password is entered it shows invalid credentials.

PROGRAM-6

ExpressJS – Database, RESTful APIs

a .Write a program to connect MongoDB database using Mongoose and perform CRUD operations.

What is MongoDB?

MongoDB is a **NoSQL database** that stores data in a **document-oriented format** (JSON-like documents) instead of traditional **rows and tables** like SQL databases.
It's widely used in **modern web and mobile apps** because it is **fast, scalable, and flexible**

What is JSON?

The full form of JSON is:

☞ **JavaScript Object Notation**

- A **lightweight data-interchange format**.
- Used to **store and exchange data** between a server and a client.
- Easy to read and write for humans.
- Easy to parse and generate for machines.

◇ **Example of JSON**

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

Key Features of MongoDB

1. **Document-oriented**
2. **Collections instead of Tables**
    - Documents are grouped into **collections** (like tables in SQL).
    - Example: A users collection can have many documents.
3. **Schema-less (Flexible)**
    - Unlike SQL, you don't have to predefine a strict schema.
    - One user document may have { name, email, age }, another may have { name, phone }.

Mongoose is a popular **Object Data Modeling (ODM) library** for MongoDB and Node.js.

What Mongoose Does

- Acts as a **bridge** between your Express app (Node.js) and **MongoDB database**.
- Provides a **schema-based solution** to model your data (like defining structure, validation rules, and relationships).
- Makes working with MongoDB easier using **JavaScript objects**, instead of writing raw queries.

◇ **Why Use Mongoose with Express?**

1. **Schema definition** – You can define how your MongoDB documents should look.
2. **Validation** – Ensures data integrity before saving into MongoDB.
3. **Query helpers** – Easier to query, update, and delete data.
4. **Middleware (hooks)** – Functions that run before/after saving, updating, or deleting.
5. **Relationships & population** – Manage references between collections.

CRUD operations

CRUD operations are the **fundamental building blocks** of working with any database or backend system.

CRUD stands for the four basic operations you can perform on data in a database:

◇ **CRUD = Create, Read, Update, Delete**

- **Create** → Add a new contact.
- **Read** → View your contacts.
- **Update** → Edit a contact's phone number.
- **Delete** → Remove a contact.

Install MongoDB locally

download from: https://www.mongodb.com/try/download/community

Verify installation : **mongod --version**

After installing:

- Start MongoDB server:

This starts MongoDB on mongodb://127.0.0.1:27017 (default port).

Steps: Install Required Packages

Run in terminal:

1. node -v
2. npm -v
3. npm install express mongoose
4. npm list mongoose
5. **mongod –version**
6. npm init -y

 # PROGRAM CODE

```
const express = require('express');
const mongoose = require('mongoose');
const app = express();
app.use(express.json()); // for JSON data

// 1 Connect to MongoDB
mongoose.connect('mongodb://127.0.0.1:27017/mydb')
  .then(() => console.log('☑ MongoDB Connected'))
  .catch(err => console.log(err));

// 2 Create Schema & Model
const UserSchema = new mongoose.Schema({
  name: String,
  email: String
});
const User = mongoose.model('User', UserSchema);

// 3 CRUD Routes
// CREATE (POST)
app.post('/users', async (req, res) => {
  const user = new User(req.body);
  await user.save();
  res.send(user);
});
// READ (GET)
app.get('/users', async (req, res) => {
  const users = await User.find();
  res.send(users);
});

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

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

// 4 Start Server
app.listen(3000, () => console.log('🚀 Server running on port 3000'));
```

Step : Run the Project

node 6a.js

☑ You can test the API using **Postman**

Explanation:

Step 1: Create a User (POST)
Request: POST : http://localhost:3000/users

Body (JSON):

```
{
  "name": "Alice",
  "email": "alice@test.com"
}
```

Response Example:

```
{
  "_id": "66a90f4a8c1b3f18f92c1a23",
  "name": "Alice",
  "email": "alice@test.com",
  "__v": 0
}
```

Step 2: Update the User (PUT)
Use the _id from the response above in your **PUT request**:
PUT http://localhost:3000/users/66a90f4a8c1b3f18f92c1a23
Body (JSON):

```
{
  "name": "Alice Updated",
  "email": "alice.new@test.com"
}
```

Step 3: Check Update
Then you can GET all users to confirm:
GET http://localhost:3000/users

Step 4: For Delete

DELETE http://localhost:3000/users/66a90f4a8c1b3f18f92c1a23

b. AIM: Write a program to develop a single page application using RESTful APIs.

What is a RESTful API?

A **RESTful API** (Representational State Transfer API) is a way for two systems (like a client and server) to communicate over the internet using **HTTP methods**.

Example: RESTful API for Users

HTTP Method	Endpoint	Description
GET	/users	Get all users
GET	/users/1	Get user with ID = 1
POST	/users	Create a new user
PUT/PATCH	/users/1	Update user with ID = 1
DELETE	/users/1	Delete user with ID = 1

1. Install dependencies

- **npm init -y**
- **npm install express body-parser**

2. 6b.js

```
const express = require('express');
const app = express();
app.use(express.json());
app.use(express.static('public'));

let items = ["Apple", "Banana"];

app.get('/api/items', (req, res) => res.json(items));
app.post('/api/items', (req, res) => { items.push(req.body.name); res.json(items); });

app.listen(3000, () => console.log('http://localhost:3000'));
```

3. public/index.html (Single Page App)

Create a folder named **public** in your own created folder and save this file as index.html in public folder

```
<!DOCTYPE html>
<html>
<body>
  <h2>Items</h2>
```

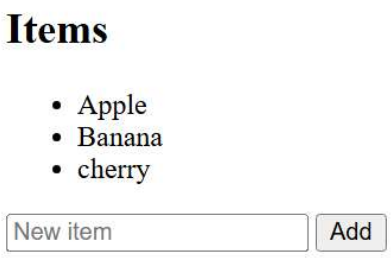
```
  <ul id="list"></ul>
  <input id="item" placeholder="New item">
  <button onclick="addItem()">Add</button>

  <script>
   async function load(){
    const res = await fetch('/api/items');
    document.getElementById('list').innerHTML = (await res.json()).map(i=>`<li>${i}</li>`).join('');
   }
   async function addItem(){
      await      fetch('/api/items',{method:'POST',headers:{'Content-
Type':'application/json'},body:JSON.stringify({name:document.getElementById('item').value})});
    document.getElementById('item').value='';
    load();
   }
   load();
  </script>
</body>
</html>
```

To Run:

node 6b.js

Output:



PROGRAM-7

ReactJS – Render HTML, JSX, Components – function & Class

What is React?

- React is a JavaScript library for building user interfaces.
- React is used to build single-page applications.
- React is a tool for building UI components.
- React allows us to create reusable UI components.
- React is a front-end JavaScript library.
- React was developed by the Facebook Software Engineer Jordan Walke.
- React is also known as React.js or ReactJS.

How does React Work?

- React creates a VIRTUAL DOM in memory.
- Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.
- React only changes what needs to be changed!
- React finds out what changes have been made, and changes **only** what needs to be changed.

Advantages of React

- ✓ Faster UI updates with Virtual DOM
- ✓ Reusable and maintainable code through components.
- ✓ Large community and ecosystem (React Router, Redux, Next.js, etc.).
- ✓ Easy to learn if you know JavaScript.
- ✓ Works for both **web** (React) and **mobile apps** (React Native).

React vs Framework

- React is a library (not a full framework).
- It only handles the view layer (UI).
- For routing, state management, and APIs, we use additional tools.

Real-World Usage

- Used by big companies → Facebook, Instagram, Netflix, Uber, Airbnb, etc.
- Suitable for:
  - Dashboards
  - Social media apps
  - E-commerce websites
  - Real-time applications (chat, video apps)

a) AIM: Write a program to render HTML to a web page.

React Render HTML

- React's goal is in many ways to render HTML in a web page.
- React renders HTML to the web page via a container, and a function called **createRoot().**

The createRoot Function

- The createRoot function is located in the main.jsx file in the src folder, and is a built-in function that is used to create a root node for a React application
- The **createRoot()** function takes one argument, an HTML element.
- The purpose of the function is to define the HTML element where a React component should be displayed.

The Container

- React uses a container to render HTML in a web page.
- Typically, this container is a element in the index.html file.

The render Method

- The render method defines what to render in the HTML container.
- The result is displayed in the element.

Setting up a React Environment

- First, make sure you have Node.js installed. You can check by running this in your terminal:
  node -v
- If Node.js is installed, you will get a result with the version number:
  v22.15.0
- If not, you will need to install Node.js.

Build Tool (Vite)

Vite is a modern frontend build tool that helps developers set up and run web projects very quickly.

The word **"Vite"** comes from **French**, meaning **"fast"**.
Its main focus is **speed**: faster startup and faster builds compared to older tools.

Vite = A **fast tool** to build and run modern web projects.

- **Dev server**: Instant updates when coding.
- **Build tool**: Optimizes code for production.
- **Supports** React, Vue, Svelte, etc.
- Name "Vite" = French for **fast**.

Install a Build Tool (Vite)

- When you have Node.js installed, you can start creating a React application by choosing a build tool.
- Run this command to install Vite:

  npm install -g create-vite

- If the installation was a success, you will get a result like this:

  added 1 package in 649ms

Create a React Application

Run this command to create a React application named my-react-app:

npm create vite@latest my-react-app -- --template react

If you get this message, just press y and press Enter to continue:

Need to install the following packages

create-vite@6.5.0
Ok to proceed? (y)

If the creation was a success, you will get a result like this:

> npx

> create-vite my-react-app --template react

o Scaffolding project in C:\Users\stale\my-react-app...

— Done.

Install Dependencies

- As the result above suggests, navigate to your new react application directory:

cd my-react-app

- And run this command to install dependencies:

npm install

Which will result in this:

added 154 packages, and audited 155 packages in 8s

33 packages are looking for funding

run `npm fund` for details

found 0 vulnerabilities

Run the React Application

- Now you are ready to run your first *real* React application!
- Run this command to run the React application my-react-app:

npm run dev

Which will result in this:

VITE v6.3.5 ready in 217 ms

➜ Local: http://localhost:5173/
➜ Network: use --host to expose
➜ press h + enter to show help

Rendering:

In **React**, **rendering** means **taking React elements (JSX or components) and displaying them inside the browser's DOM**.

Step-by-Step Explanation of the Example

1. HTML Setup

```
<div id="root"></div>
```

- This empty is where React will show all the content.
- Think of it as the **container** for React.

2. Import React and ReactDOM

- React → helps us create components using JSX.
- ReactDOM → is responsible for actually rendering (showing) those components inside the browser.

3. Create a Component

```
function Hello() {
  return <h1>Hello, React Rendering!</h1>;
}
```

- This is a **function component**.
- It returns **JSX** (<h1>...</h1>), which looks like HTML but is actually JavaScript.

4. Render to DOM

const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(<Hello />);

- **document.getElementById("root")** → finds the .
- **ReactDOM.createRoot(...)** → tells React where to put the UI.
- **.render(<Hello />)** → renders the Hello component inside that div.

✅ So what happens?

- React takes <Hello /> → replaces it with <h1>Hello, React Rendering!</h1>
- Then inserts it inside
- Finally, the user sees **Hello, React Rendering!** on the webpage.

👉 In short:

1. **Component** creates UI (JSX).
2. **ReactDOM** puts that UI inside the HTML page.

Look in the my-react-app directory, and you will find a src folder. Inside the src folder there is a file called **main.js**, open it and it will look like this:

Example:

The default content of the src/main.jsx file is replaced with the following content

main.jsx

```
import { createRoot } from 'react-dom/client'

createRoot(document.getElementById('root')).render(
  <h1>Hello React!</h1>
)
```

Output: Hello React!

Example

Display a paragraph inside the "root" element:

main.jsx

```
import { createRoot } from 'react-dom/client'

createRoot(document.getElementById('root')).render(
  <p>Welcome!</p>
)
```

Output:
Welcome

b. AIM: Write a program for writing markup with JSX

What is JSX?

- JSX stands for JavaScript XML.
- JSX allows us to write HTML in React.
- JSX makes it easier to write and add HTML in React.

Coding JSX

- JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods.
- JSX converts HTML tags into react elements.
- You are not required to use JSX, but JSX makes it easier to write React applications.

main.jsx

```
const myElement = <h1>I Love JSX!</h1>;

createRoot(document.getElementById('root')).render(
  myElement
);
```

JSX is an extension of the JavaScript language based on ES6, and is translated into regular JavaScript at runtime.

Writing markup with JSX

JSX is a syntax extension for JavaScript, used with React to describe what the UI should look like. This guide will explain why React mixes markup with rendering logic, how JSX differs from HTML, and how to display information using JSX.

How JSX is Different from HTML?

JSX may look similar to HTML, but there are key differences:

- **JavaScript Expressions:** JSX allows you to embed JavaScript expressions inside curly braces {}. This lets you dynamically render content based on your application state or props.
- **Attributes**: JSX attributes follow camelCase naming conventions instead of the kebab-case used in HTML. For example, class in HTML becomes className in JSX.
- **Self-Closing Tags**:Like XML, JSX requires that tags be properly closed. If an element doesn't have children, you must use a self-closing tag (e.g., <input /> instead of <input>).
- **Fragment Syntax**: JSX supports fragment syntax, allowing you to group multiple elements without adding extra nodes to the DOM. This can be done using <React.Fragment> or shorthand syntax <> </>.

Rules of JSX

1. Return a single root element

To return multiple elements from a component, enclose them within a single parent element.

```
<div>
  <h1>Geeks for Geeks</h1>
  <ul>
    .....
  </ul>
</div>
```

2. Close all the Tags

JSX mandates explicit closure of tags: self-closing tags such as `<img>` must be written as `<img />`, and wrapping tags like `<li>oranges` should be corrected to `<li>oranges</li>`.

```
<>
<img    src= "https://google.com/photos.jpg"
          alt =" google "
          class = "photo"
/>
   <ul>
         <li> Blue </li>
         <li> Green </li>
         <li>  Red </li>
  </ul>
</>
```

3. camelCase most of the things

JSX converts into JavaScript, where JSX attributes become keys in JavaScript objects. To handle this conversion seamlessly, React adopts camelCase for attribute names. For instance, attributes like stroke-width in JSX are transformed into strokeWidth in JavaScript. Similarly, to avoid conflicts with reserved words like class, React uses className instead, aligning with corresponding DOM properties for clarity and compatibility.

```
<img
       src= "https://imgur.com/"
       alt= "gfg"
       className = "photo"
/>
```

Step -by- step procedure:

Look in the my-react-app directory, and you will find a src folder. Inside the src folder there is a file called **main.js**, open it and replace with your required content.

❖ **Example 1**

Main.JSX:

```
const myElement = <h1>I Love JSX!</h1>;

createRoot(document.getElementById('root')).render(
  myElement
);
```

Output: I Love JSX!

Example 2: Expressions in JSX

With JSX you can write expressions inside curly braces { }.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Execute the expression 5 + 5:

main.jsx

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

Output: React is 10 times better with JSX

Example 3: Inserting a Large Block of HTML

To write HTML on multiple lines, put the HTML inside parentheses:

Create a list with three list items:

main.jsx

```
const myElement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);
```

Output:

- Apples
- Bananas
- Cherries

Example 3: One Top Level Element

The HTML code must be wrapped in *ONE* top level element.

So if you like to write *two* paragraphs, you must put them inside a parent element, like a div element.

Example

Wrap two paragraphs inside one DIV element

main.jsx

```
const myElement = (
  <div>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </div>
);
```

Output:

I am a paragraph.
I am a paragraph too.

- JSX will throw an error if the HTML is not correct, or if the HTML misses a parent element.
- Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.
- A fragment looks like an empty HTML tag: <></>.

Example
Wrap two paragraphs inside a fragment:

main.jsx

```
const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
);
```

Output:

I am a paragraph.
I am a paragraph too.

Elements Must be Closed

JSX follows XML rules, and therefore HTML elements must be properly closed.

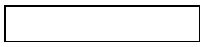
Example

Close empty elements with />

main.jsx

```
const myElement = <input type="text" />;
```

Output:



Note: JSX will throw an error if the HTML is not properly closed.

Attribute class = className

The class attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the class keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

Use attribute **className** instead.

JSX solved this by using className instead. When JSX is rendered, it translates className attributes into class attributes.

Example

Use attribute className instead of class in JSX:

main.jsx

```
const myElement = <h1 className="myclass">Hello World</h1>;
```

Comments in JSX

Comments in JSX are written with {/* */}

Example

main.jsx

```
const myElement = <h1>Hello {/* Wonderful */} World </h1>;
```

Output:

Hello World

c. AIM: Write a program for creating and nesting components (function and class).

React Components

What is a Component?

- A **component** is a reusable, independent piece of UI in React.
- Each component can represent part of a webpage (like a button, header, or form).
- Components make code **modular, reusable, and easier to maintain**.

Types of Components

React provides two main ways to create components:

a) Function Components

- A function that returns JSX.
- Simple and lightweight.
- Can accept **props** (input data) and display them.
- In modern React, hooks (useState, useEffect, etc.) make function components powerful.

Example:

```
function Greeting(props) {
  return <h2>Hello, {props.name}!</h2>;
}
```

b) Class Components

- Written as JavaScript **classes**.
- Must extend React.Component.
- Have a **render()** method that returns JSX.
- Can manage their own **state** and lifecycle methods.

Example:

```
class Welcome extends React.Component {
  render() {
    return <h1>Welcome, {this.props.name}!</h1>;
  }
}
```

3. Nesting Components

- Nesting means **using one component inside another**.
- This helps build complex UIs by combining small, reusable components.

Example:

```
function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Welcome name="Bob" />
    </div>
  );
}
```

Here:

- App is the parent component.
- Greeting and Welcome are **nested (child) components** inside App.

4. Props and State in Nesting

- **Props** (properties) → Passed from parent to child. They are read-only.
- **State** → Local to a component; used to manage changing data.
- Nested components usually **receive data via props** from parent components.

5. Why Use Nesting?

✅ Reusability → One component can be used in many places.
✅ Separation of Concerns → Each component handles its own logic/UI.
✅ Easy Maintenance → Small, independent units are easier to debug.
✅ Clear Hierarchy → Parent → Child → Sub-child structure makes UI structured.

📌 **Summary:**

- Function components → simple, use hooks.
- Class components → use render() and lifecycle methods.
- Nesting → placing one component inside another to build a hierarchy.

📌 **Example: Creating & Nesting Function and Class Components**

```
// Import React and ReactDOM
import React from "react";
import ReactDOM from "react-dom/client";
```

```
// ✅ Function Component

function Greeting(props) {
  return <h2>Hello, {props.name}!</h2>;
}

// ✅ Class Component

class Welcome extends React.Component {
  render() {
    return (
      <div>
        <h1>Welcome Component</h1>
        {/* Nesting function component inside class */}
        <Greeting name="Alice" />
        <Greeting name="Bob" />
      </div>
    );
  }
}

// ✅ Another Function Component nesting both
function App() {
  return (
    <div>
      <h1>App Component</h1>
      {/* Nesting class component */}
      <Welcome />
    </div>
  );
}

// Render into the root element
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<App />);
```

Output:

```
App Component
Welcome Component
Hello, Alice!
Hello, Bob!
```

🔍 **Explanation:**

1. **Greeting** → A simple **function component** with props.
2. **Welcome** → A **class component** that **nests Greeting** multiple times.
3. **App** → The main function component, which **nests Welcome**.
4. **root.render(<App />)** → Renders everything into the HTML page.

PROGRAM-8

ReactJS – Props and States, Styles, Respond to Events

a. AIM: Write a program to work with props and states.

In React, State allows components to manage and update internal data dynamically, while Props enables data to be passed from a parent component to a child component. Understanding their differences and use cases is essential for developing efficient React applications.

State in React

- React components has a built-in state object.
- The state object is where you store property values that belong to the component.
- When the state object changes, the component re-renders.

State is a built-in object in React components that holds data or information about the component. It is mutable, which means it can be updated within the component using the setState method in class components or the useState hook in functional components.

- State is local to the component and cannot be accessed by child components unless passed down as props.
- It is mutable, meaning it can change over time based on user interactions or API responses.
- When state updates, the component re-renders to reflect the changes.
- Managed using useState in functional components or this.setState in class components.

 # PROGRAM

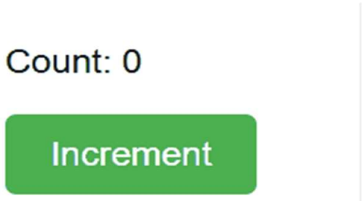
```
import React, { useState } from 'react';

function Counter() {
   const [count, setCount] = useState(0);

   return (
      <div>
         <p>Count: {count}</p>
         <button onClick={() => setCount(count + 1)}>Increment</button>
      </div>
   );
}

export default Counter;
```

Output



In this example

- useState(0) initializes the state variable count with the value 0.
- The setCount function is used to update the state whenever the button is clicked. This triggers a re-render, updating the displayed count.

Props in React

- Props are arguments passed into React components.
- Props are passed to components via HTML attributes.
- props stand for properties.
- React Props are like function arguments in JavaScript *and* attributes in HTML.

Props (short for Properties) are used to pass data from a parent component to a child component. Unlike state, props are immutable, meaning they cannot be modified within the receiving component.

- Props allow components to be reusable and dynamic.
- Props are read-only and cannot be changed by the child component.
- They help in data communication between components.
- Passed as attributes in JSX elements.
- import React from 'react';

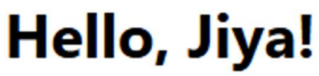
 # PROGRAM CODE

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

function App() {
   return <Greeting name="Jiya" />;
}

export default App;
```

Output



Props in React

Here, the Greeting component receives a prop named name and displays a personalized message. The App component passes the value "Jiya" as a prop.

b. AIM: Write a program to add styles (CSS & Sass Styling) and display data

CSS in React is used to style components, just like in normal HTML, but React lets you organize styles in multiple ways:

1. **Normal CSS** – Create a .css file and import it into a component.
2. **Inline Styling** – Add styles directly as a JavaScript object using the style attribute.
3. **CSS Modules** – Scoped CSS files (.module.css) that avoid class name conflicts.
4. **Sass (SCSS)** – Advanced styling with variables, nesting, and mixins using .scss files.

CSS makes React components look good, and React gives you flexible ways to apply it.

What is Sass?

- Sass stands for Syntactically Awesome Stylesheet
- Sass is an extension to CSS
- Sass is a CSS pre-processor
- Sass is completely compatible with all versions of CSS
- Sass reduces repetition of CSS and therefore saves time
- Sass was designed by Hampton Catlin and developed by Natalie Weizenbaum in 2006
- Sass is free to download and use

Why Use Sass?

Stylesheets are getting larger, more complex, and harder to maintain. This is where a CSS pre-processor can help.

Sass lets you use features that do not exist in CSS, like variables, nested rules, mixins, imports, inheritance, built-in functions, and other stuff.

Sass File Type

Sass files has the **".scss"** file extension.

Sass Comments

Sass supports standard CSS comments /* comment */, and in addition it supports inline comments // comment:

Sass Example

```
/* define primary colors */
$primary_1: #a2b9bc;
$primary_2: #b2ad7f;

/* use the variables */
.main-header {
  background-color: $primary_1; // here you can put an inline comment
}
```

Step -by -step Procedure:

1. Folder Setup

my-app/

├ src/

| ├ App.js

| ├ App.css

| ├ App.scss

| └ index.js

Step 2: Create a React Project

npx create-react-app my-app

- my-app is your project folder.
- It will take a few minutes to set up.

Step 3: Navigate to Your Project

cd my-app

Step 4: Install Sass

npm install sass

- This allows React to compile .scss files automatically.

Step 5: Create Your Files

5a. App.css : (Inside src folder)

```
container {
  text-align: center;
  background-color: #f9f9f9;
  padding: 20px;
```

```
}
title {
  color: #2c3e50;
  font-size: 2rem;
}
```

5b. App.scss : (Inside src folder)

```
$list-color: #27ae60;

.data-list {
  list-style: none;
  padding: 0;

  li {
    background: lighten($list-color, 30%);
    margin: 8px 0;
    padding: 10px;
    border-radius: 6px;
    font-weight: bold;

    &:hover {
      background: darken($list-color, 10%);
      color: #fff;
      cursor: pointer;
    }
  }
}
```

5c. App.js (Inside src folder)

```
import React from "react";
import "./App.css";   // CSS
import "./App.scss";  // Sass

function App() {
  const fruits = ["Apple", "Banana", "Cherry", "Mango", "Orange"];

  return (
    <div className="container">
      <h1 className="title">Fruit List</h1>
      <ul className="data-list">
        {fruits.map((fruit, index) => (
          <li key={index}>{fruit}</li>
        ))}
      </ul>
    </div>
  );
}
export default App;
```

Explanation

1. App.js imports CSS and SCSS for styling.
2. The fruits array is mapped to a list of <li> elements.
3. App.css contains basic styling like text alignment and title color.
4. App.scss uses Sass features like variables ($fruit-color) and nested styles for .fruit-list .fruit-item.
5. When you hover over a fruit, the color changes thanks to the &:hover in Sass.

Execution steps:

Open the terminal and follow the below commands

1. **npx create-vite my-react-app --template react**
2. **cd my-react-app**
3. **npm install**
4. **npm install sass**
5. **npm run dev**

-→

VITE v7.1.6 ready in 508 ms

➜ Local: http://localhost:5173/
➜ Network: use --host to expose
➜ press h + enter to show help

Copy the url: **http://localhost:5173/** in any web browser and it displays the below result.

Output:

```
---------------------------------------
|                                     |
|          Fruit List                 |
|                                     |
|   [Apple]                           |
|   [Banana]                          |
|   [Cherry]                          |
|   [Mango]                           |
|   [Orange]                          |
|                                     |
---------------------------------------
```

c. AIM: Write a program for responding to events.

In React, events are actions that occur within an application, such as clicking a button, typing in a text field, or moving the mouse. React provides an efficient way to handle these actions using its event system.

React event handlers are written inside curly braces:

Syntax

<button onClick={shoot}>Take the Shot!</button>

Event handlers like onClick, onChange, and onSubmit are used to capture and respond to these events.

Syntax

<element onEvent={handlerFunction} />

- element: The JSX element where the event is triggered (e.g., <button>, <input>, etc.).
- onEvent: The event name in camelCase (e.g., onClick, onChange).
- handlerFunction: The function that handles the event when it occurs.

Commonly Used React Event Handlers

React provides a variety of built-in event handlers that we can use to handle different user interactions:

React Event	Description
onClick	This event is used to detect mouse clicks in the user interface.
onChange	This event is used to detect a change in the input field in the user interface.
onSubmit	This event fires on the submission of a form in the user interface and is also used to prevent the default behavior of the form.
onKeyDown	This event occurs when the user press any key from the keyboard.
onKeyUp	This event occurs when the user releases any key from the keyboard.
onMouseEnter	This event occurs when the ouse enters the boundary of the element

 # PROGRAM CODE

// App.jsx

```
import React, { useState } from "react";

function App() {
  // State to store input value
  const [name, setName] = useState("");
  // State to store button click message
  const [message, setMessage] = useState("");

  // Event handler for button click
  const handleClick = () => {
    setMessage(`Hello, ${name || "Guest"}!`);
  };

  // Event handler for input change
  const handleChange = (event) => {
    setName(event.target.value);
  };

  return (
    <div style={{ textAlign: "center", marginTop: "50px" }}>
      <h1>React Event Handling Example</h1>

      <input
        type="text"
        placeholder="Enter your name"
        value={name}
        onChange={handleChange}
        style={{ padding: "10px", fontSize: "16px" }}
      />
      <br /><br />

      <button
        onClick={handleClick}
        style={{ padding: "10px 20px", fontSize: "16px", cursor: "pointer" }}
      >
        Greet Me
      </button>
      <br /><br />

      <h2>{message}</h2>
    </div>
  );
}

export default App;
```

Output:

- Type your name in the input box → click **Greet Me** → the message "Hello, [Your Name]!" appears

PROGRAM-9
ReactJS – Conditional Rendering, Rendering Lists, React Forms

a. AIM: Write a program for conditional rendering.

What is Conditional Rendering?

Conditional rendering in React allows you to render different components or elements based on a condition. It's like using an if-else statement in HTML—React decides what to display depending on the state or props.

Components of the Program

State Management (useState):

```
const [isLoggedIn, setIsLoggedIn] = useState(false);
```

- Here, **isLoggedIn** is a state variable that tracks whether the user is logged in (true) or not (false).
- **setIsLoggedIn** is a function to update the state.

Conditional Rendering with Ternary Operator:

```
{isLoggedIn ? (
  <p>Welcome, User! You are logged in.</p>
) : (
  <p>Please log in to continue.</p>
)}
```

- If isLoggedIn is true, it displays a welcome message.
- If isLoggedIn is false, it asks the user to log in.
- This is the core part of conditional rendering.

Event Handling (onClick):

```
<button onClick={handleLoginToggle}>
  {isLoggedIn ? "Logout" : "Login"}
</button>
```

- Clicking the button toggles the isLoggedIn state.
- The button text also changes based on the state.

Dynamic UI Update:
React automatically re-renders the component whenever the state changes. So, when the user logs in or out, the displayed message and button text update immediately.

Why It's Useful

- Conditional rendering allows dynamic user interfaces.
- You can show or hide elements, switch components, or change content depending on user actions, API responses, or application state.
- It keeps the UI interactive and responsive without reloading the page.

 # PROGRAM CODE:

A **simple ReactJS example** for **conditional rendering** using a functional component.

```
import React, { useState } from "react";

function App() {
  // State to track whether the user is logged in
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  // Function to toggle login status
  const handleLoginToggle = () => {
    setIsLoggedIn(!isLoggedIn);
  };

  return (
    <div style={{ textAlign: "center", marginTop: "50px" }}>
      <h1>Conditional Rendering Example</h1>

      {/* Conditional rendering using ternary operator */}
      {isLoggedIn ? (
        <p>Welcome, User! You are logged in.</p>
      ) : (
        <p>Please log in to continue.</p>
      )}

      {/* Button to toggle login status */}
      <button onClick={handleLoginToggle} style={{ padding: "10px 20px", marginTop: "20px" }}>
        {isLoggedIn ? "Logout" : "Login"}
      </button>
    </div>
  );
}

export default App;
```

Output Example

Initially:

Please log in to continue.

[Login]

After clicking Login:

Welcome, User! You are logged in.

[Logout]

b) AIM: Write a program for rendering lists.

What is List Rendering?

In React, list rendering is the process of displaying multiple items from an array or collection dynamically in the UI. Instead of manually writing each item, React allows you to loop through the data and render it efficiently.

Components of the Program

Array of Items:

```
const fruits = ["Apple", "Banana", "Cherry", "Mango", "Orange"];
```

- This is the data source.
- Each element in the array will be rendered as a list item in the UI.

Using map() to Render Lists:

```
{fruits.map((fruit, index) => (
  <li key={index}>{fruit}</li>
))}
```

- The map() function iterates over the array.
- For each element, it returns a React element (<li> in this case).
- key={index} is a unique identifier required by React to optimize rendering and efficiently update the UI when items change.

Dynamic UI Rendering:

When the array changes (items added, removed, or updated), React automatically updates the displayed list. This keeps the interface reactive and up-to-date.

Using <ul> for List Structure:

```
<ul> ... </ul>
```

The <ul> (unordered list) element is used to display the list items visually in a bullet format**.**

Why It's Useful

- Dynamically renders any number of items from arrays or objects.
- Efficiently updates the DOM when data changes without reloading the page.
- Essential for building dynamic UIs like menus, tables, and lists from APIs or databases.

 # PROGRAM CODE

A simple ReactJS example for rendering lists using a functional component.

```
import React from "react";

function App() {
  // Array of fruits
  const fruits = ["Apple", "Banana", "Cherry", "Mango", "Orange"];

  return (
    <div style={{ textAlign: "center", marginTop: "50px" }}>
      <h1>Rendering Lists Example</h1>

      <ul>
        {/* Rendering list items using map() */}
        {fruits.map((fruit, index) => (
          <li key={index}>{fruit}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Output

The program will render:

- Apple
- Banana
- Cherry
- Mango
- Orange

Each element of the array is displayed as a **list item** in the browser.

c) AIM: Write a program for working with different form fields using react forms.

What are React Forms?

In React, forms are used to **collect user input**. React forms can be either **controlled** or **uncontrolled**. Controlled forms are recommended because the form data is **stored in React state**, making it easier to manage, validate, and update dynamically.

Components of the Program

State Management (useState):

```
const [formData, setFormData] = useState({
  name: "",
  email: "",
  gender: "",
  hobbies: [],
  country: "",
});
```

- All form fields are stored in a single state object called formData.
- Each field corresponds to a key in the state (e.g., name, email, gender, hobbies, country).

Handling Input Changes:

- **const handleChange = (e) => { ... }**
  - handleChange updates the state whenever a user types in a text box, selects a radio button, checkbox, or dropdown.
  - Checkboxes are handled specially because multiple options can be selected, so we update the hobbies array in the state.

Form Fields:

1. **Text Input** (name)
2. **Email Input** (email)
3. **Radio Buttons** (gender) – allows selecting **one option**
4. **Checkboxes** (hobbies) – allows selecting **multiple options**
5. **Select Dropdown** (country) – allows choosing **one option**

Form Submission:

- **const handleSubmit = (e) => { ... }**
  - Prevents default HTML form submission using e.preventDefault().
  - Displays the collected form data in an alert (or can be sent to a backend API).

Why It's Useful

- React forms **collect and manage user input** dynamically.
- Using controlled components ensures the UI **stays in sync with state**.
- Useful for creating **registration forms, surveys, login forms**, and other interactive user inputs.

 # PROGRAM CODE

```
import React, { useState } from "react";

function App() {
 // State to store form data
 const [formData, setFormData] = useState({
  name: "",
  email: "",
  gender: "",
  hobbies: [],
  country: "",
 });

 // Options for hobbies
 const hobbyOptions = ["Reading", "Traveling", "Cooking", "Sports"];

 // Handle input changes
 const handleChange = (e) => {
  const { name, value, type, checked } = e.target;

  if (type === "checkbox") {
   let newHobbies = [...formData.hobbies];
   if (checked) {
    newHobbies.push(value);
   } else {
    newHobbies = newHobbies.filter((hobby) => hobby !== value);
   }
   setFormData({ ...formData, hobbies: newHobbies });
  } else {
```

```
    setFormData({ ...formData, [name]: value });
  }
};

// Handle form submission
const handleSubmit = (e) => {
  e.preventDefault();
  alert("Form Data Submitted:\n" + JSON.stringify(formData, null, 2));
};

return (
  <div style={{ maxWidth: "500px", margin: "50px auto" }}>
    <h2>React Form Example</h2>
    <form onSubmit={handleSubmit}>
      {/* Text Input */}
      <div>
        <label>Name:</label>
        <input
          type="text"
          name="name"
          value={formData.name}
          onChange={handleChange}
          required
        />
      </div>

      {/* Email Input */}
      <div>
        <label>Email:</label>
        <input
          type="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
          required
        />
```

```
    </div>

    {/* Radio Buttons */}
    <div>
      <label>Gender:</label>
      <input
        type="radio"
        name="gender"
        value="Male"
        checked={formData.gender === "Male"}
        onChange={handleChange}
      />{" "}
      Male
      <input
        type="radio"
        name="gender"
        value="Female"
        checked={formData.gender === "Female"}
        onChange={handleChange}
      />{" "}
      Female
    </div>

    {/* Checkboxes */}
    <div>
      <label>Hobbies:</label>
      {hobbyOptions.map((hobby) => (
        <label key={hobby} style={{ marginRight: "10px" }}>
          <input
            type="checkbox"
            name="hobbies"
            value={hobby}
            checked={formData.hobbies.includes(hobby)}
            onChange={handleChange}
          />
          {hobby}
```

```
          </label>
        ))}
      </div>

      {/* Select Dropdown */}
      <div>
        <label>Country:</label>
        <select
          name="country"
          value={formData.country}
          onChange={handleChange}
          required
        >
          <option value="">Select Country</option>
          <option value="India">India</option>
          <option value="USA">USA</option>
          <option value="UK">UK</option>
          <option value="Australia">Australia</option>
        </select>
      </div>

      {/* Submit Button */}
      <button type="submit" style={{ marginTop: "20px" }}>
        Submit
      </button>
    </form>
  </div>
 );
}

export default App;
```

Explanation

1. **Controlled Components**:
    - Each form field's value is controlled by React state (formData).
    - This allows React to keep track of form data dynamically.
2. **Text and Email Inputs**:
    - Standard <input> fields with type="text" and type="email".
3. **Radio Buttons (Gender)**:
    - Only one option can be selected at a time.
    - checked={formData.gender === "Male"} ensures the correct radio button is selected based on state.
4. **Checkboxes (Hobbies)**:
    - Multiple options can be selected.
    - The state array formData.hobbies stores all selected hobbies.
5. **Select Dropdown (Country)**:
    - <select> allows users to choose one option.
    - Controlled with value={formData.country}.
6. **Form Submission**:
    - handleSubmit prevents default form submission and displays an alert with form data.

Output

- User enters name and email, selects gender, hobbies, and country.
- On clicking **Submit**, the form displays:

Form Data Submitted:

{
"name": "John Doe",
"email": "john@example.com",
"gender": "Male",
"hobbies": ["Reading", "Sports"],
"country": "USA"
}

PROGRAM-10

ReactJS – React Router, Updating the Screen

a. AIM: Write a program for routing to different pages using react router

What is React Router?

React Router is a standard library in React used for routing—i.e., navigating between different components (pages) in a single-page application (SPA) without reloading the page.

- It allows React apps to update the URL and render different components on the same page.

Key components:

1. <BrowserRouter> – Wraps the app to enable routing.
2. <Routes> – Contains all <Route> elements.
3. <Route> – Maps a URL path to a component.
4. <Link> – Used for navigation between routes without page reload.

- React Router is used to navigate between different pages/components in a React application.
- Using BrowserRouter, Routes, and Route, we can define paths and the components to render for eachpath.
- This example will show a simple 3-page application: Home, About, and Contact.

 # PROGRAM CODE

```
import React from "react";
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";

// Home Component
function Home() {
  return <h2>Welcome to the Home Page</h2>;
}

// About Component
function About() {
  return <h2>About Us Page</h2>;
}

// Contact Component
function Contact() {
  return <h2>Contact Us Page</h2>;
}

// App Component with Routing
function App() {
  return (
    <Router>
      <div style={{ textAlign: "center", marginTop: "50px" }}>
        <h1>React Router Example</h1>
```

```
      {/* Navigation Links */}
      <nav style={{ marginBottom: "20px" }}>
        <Link to="/" style={{ margin: "10px" }}>Home</Link>
        <Link to="/about" style={{ margin: "10px" }}>About</Link>
        <Link to="/contact" style={{ margin: "10px" }}>Contact</Link>
      </nav>

      {/* Routes */}
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </div>
  </Router>
 );
}

export default App;
```

Output:

Initial Render (Home Page)
React Router Example

[Home] [About] [Contact]

Welcome to the Home Page
The navigation links are at the top: Home | About | Contact.
Since the default path is /, the Home component is displayed.

After Clicking "About" Link
React Router Example

[Home] [About] [Contact]

About Us Page
The URL changes to /about.
The About component is rendered without reloading the page.

After Clicking "Contact" Link
React Router Example

[Home] [About] [Contact]

Contact Us Page
The URL changes to /contact.
The Contact component is rendered dynamically.

b. AIM: Write a program for updating the screen.

Updating the Screen

In React, updating the screen is mainly achieved through state changes. When the state of a component changes, React re-renders the component or parts of the UI that depend on that state.

 # PROGRAM for <u>React Router and Screen Update</u>

```
import React, { useState } from "react";
function App() {
  // useState hook to manage the count
  const [count, setCount] = useState(0);
return (
    <div style={{ textAlign: "center", marginTop: "50px" }}>
      <h1>Screen Update Example</h1>
      <p>Current Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase Count</button>
      <button onClick={() => setCount(count - 1)} style={{ marginLeft: "10px" }}>
        Decrease Count
      </button>
    </div>
  );
}
export default App;
```

Output

Initial Render:
Screen Update Example
Current Count: 0

[Increase Count] [Decrease Count]
After Clicking "Increase Count" Button Twice:
Screen Update Example
Current Count: 2

[Increase Count] [Decrease Count]
After Clicking "Decrease Count" Button Once:
Screen Update Example
Current Count: 1

[Increase Count] [Decrease Count]

PROGRAM-11

ReactJS – Hooks, Sharing data between Components

a. AIM: Write a program to understand the importance of using hooks.

Importance of Using Hooks

Hooks are special functions in React that let you **use state and other React features in functional components**, without writing class components.

This program demonstrates two essential hooks:

1. **useState**
   - Allows a functional component to maintain and update **state**.
   - In the program, it manages a **counter value** (count) that can be incremented or decremented by buttons.
2. **useEffect**
   - Allows a functional component to perform **side effects**, such as updating the browser title, fetching data, or subscribing to events.
   - In the program, it updates the **document title** whenever the count changes.

Key Takeaways:

- Hooks make functional components **powerful and stateful**, which previously required class components.
- They simplify code by avoiding **complex lifecycle methods** like componentDidMount and componentDidUpdate.
- Hooks improve **code readability, reusability, and maintainability**.

 # PROGRAM CODE

```
import React, { useState, useEffect } from "react";

function Counter() {
  // useState hook to manage count state
  const [count, setCount] = useState(0);

  // useEffect hook to perform side effects
  useEffect(() => {
    document.title = `You clicked ${count} times`;
    console.log(`Document title updated: ${count}`);
  }, [count]); // Dependency array: runs effect whenever 'count' changes

  return (
```

```
    <div style={{ textAlign: "center", marginTop: "50px" }}>
      <h1>React Hooks Example</h1>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </div>
  );
}

export default Counter;
```

Explanation

1. **useState**:
   - Initializes count to 0.
   - setCount updates the count value.
2. **useEffect**:
   - Runs after every render where count changes.
   - Updates the browser tab title dynamically.
3. **Buttons**:
   - Clicking the buttons changes state, demonstrating how hooks manage state in functional components.

Importance of Hooks

- Eliminates the need for class components for state and lifecycle methods.
- Makes code **more readable** and **less boilerplate**.
- Encourages **reusable logic** via custom hooks.

Output:

- Initial render:
  - Count: 0
  - Browser tab: You clicked 0 times
- Click **Increment**:
  - Count: 1
  - Browser tab: You clicked 1 times
- Click **Decrement**:
  - Count: 0
  - Browser tab: You clicked 0 times

b. AIM: Write a program for sharing data between components.

- In React, data flows from parent to child using props.
- This program will show a Parent component passing data to a Child component.
- We'll also show how a child can notify the parent using a function passed as a prop.

Program: Sharing Data Between Components

```
import React, { useState } from "react";

// Child Component
function Child({ message, sendDataToParent }) {
  return (
    <div style={{ border: "1px solid blue", padding: "10px", margin: "10px" }}>
      <h3>Child Component</h3>
      <p>Message from Parent: {message}</p>
      <button onClick={() => sendDataToParent("Hello Parent! From Child")}>
        Send Data to Parent
      </button>
    </div>
  );
}

// Parent Component
function Parent() {
  const [parentMessage, setParentMessage] = useState("Hi Child! From Parent");
  const [childData, setChildData] = useState("");

  return (
    <div style={{ textAlign: "center", marginTop: "50px" }}>
      <h1>Data Sharing Between Components</h1>

      <Child message={parentMessage} sendDataToParent={setChildData} />

      {childData && (
        <p style={{ marginTop: "20px", fontWeight: "bold" }}>
          Data received from Child: {childData}
        </p>
      )}
    </div>
  );
}

export default Parent;
```

Explanation

1. **Parent Component**
   - Holds the state parentMessage to send to the child.
   - Holds childData to store data received from the child.
2. **Child Component**
   - Receives message and sendDataToParent as props.
   - Displays the message from the parent.
   - Calls sendDataToParent function when the button is clicked to send data back to the parent.
3. **Data Flow**
   - **Parent → Child**: via props.message
   - **Child → Parent**: via props.sendDataToParent (callback function)

Key Points

- Props are used for **sharing data from parent to child**.
- Functions passed as props allow **child-to-parent communication**.
- This is the **simplest and most common way** to share data between components in React.

OUTPUT:

Initial Render (Browser View)
Data Sharing Between Components

Child Component
Message from Parent: Hi Child! From Parent

[Send Data to Parent Button]
The child component displays the message received from the parent:
Hi Child! From Parent
There is a button labeled:
Send Data to Parent

After Clicking "Send Data to Parent" Button
Data Sharing Between Components

Child Component
Message from Parent: Hi Child! From Parent

[Send Data to Parent Button]
Data received from Child: Hello Parent! From Child
Clicking the button sends data from the child back to the parent.
The parent updates its state and displays:
Data received from Child: Hello Parent! From Child

PROGRAM – 12

MongoDB – Installation, Configuration, CRUD operations

a) AIM: Install MongoDB and configure ATLAS

MongoDB Installation

MongoDB is a **NoSQL, document-oriented database** that stores data in **JSON-like documents** instead of traditional rows and columns.

MongoDB can be used in two ways:

1. **Local Installation** – Installing MongoDB software on your system.
2. **Cloud-based Atlas** – Using MongoDB Atlas, a fully managed cloud database service that provides scalability, security, and easy access without manual setup.

Part 1: Install MongoDB (Locally)

On Windows

Download MongoDB

- ✓ Go to: **https://www.mongodb.com/try/download/community**
- ✓ Choose Windows, select the MSI installer.

Install MongoDB

- ✓ Run the installer.
- ✓ Select *Complete Installation*.
- ✓ Make sure *Install MongoDB as a Service* is checked.

Option 1: Use the Local System Account (Recommended) i.e, Install as a service (run MongoDB in background on your PC)

- During installation, when it asks **"Service Configuration"**, do the following:

- ✓ ☑ **Select**: Run service as Network Service or Local System
- ✓ ✗ **Do NOT select**: "Run service as a specific user"

This way, **you don't need to enter username/password**, and Windows will run the MongoDB service under a built-in system account.

a. Make sure that u have clicked the check-box below i.e;
   ☑ Install MongoDB Compass (GUI tool).

Verify the path of the folder where MongoDB is installing
Click on next→next->--to Finish.

Step-by-Step Fix: Add MongoDB to the PATH Environment Variable (Windows)

Step 1: Find MongoDB Installation Folder

1. Go to: C:\Program Files\MongoDB\Server\6.0\bin

The version (6.0) might be different on your system (e.g., 5.0, 4.4, etc.)
Make sure this folder has the files like mongo.exe, mongod.exe, etc.

Step 2: Add This Path to Environment Variables

1. Press Windows + S → Search for **Environment Variables**
2. Click **"Edit the system environment variables"**
3. In the System Properties window, click on **Environment Variables**

4. Under **System variables**, find and select the Path variable → Click **Edit**
5. Click **New** → Paste the MongoDB bin path:
6. C:\Program Files\MongoDB\Server\6.0\bin

Click **OK** on all windows to save.

Step 3: Verify Installation

Open *Command Prompt* and type:

mongod --version

- If installed correctly, it will show MongoDB version.

MongoDB Atlas

MongoDB **Atlas** is a **cloud-based, fully managed database service** for MongoDB. It runs on **AWS, Azure, and Google Cloud**, allowing you to create, run, and scale MongoDB databases without managing servers yourself.

☑ **Key points**

- Hosted MongoDB in the cloud.
- Fully managed (backup, scaling, monitoring, security).
- Runs on AWS, Azure, GCP.
- Provides free and paid clusters.
- Easy to connect with apps using a connection string.

Configure MongoDB Atlas (Cloud)

1. **Create Atlas Account**
   - Go to **https://www.mongodb.com/cloud/atlas**
   - Sign up (use Google/GitHub/email).
2. **Create a Cluster**
   - Click *Build a Database*.
   - Choose a free tier (M0 – free, 512MB storage).
   - Pick a cloud provider (AWS, Azure, or GCP).
   - Select a region (closest to you).
   - Click *Create Cluster*.
3. **Create a Database User**
   - Go to *Database Access*.
   - Add new user (set username & password).
   - Save credentials.
4. **Configure Network Access**
   - Go to *Network Access*.

- Add IP address → choose *Allow Access from Anywhere* (0.0.0.0/0).
- Or add your local machine IP for security.

5. **Connect to Cluster**
   - Click *Connect → Connect your application*.
   - Get the **Connection String** from the "Connect" option.

     Example:

     mongodb+srv://username:password@cluster0.abcd.mongodb.net/myDatabase
6. **Connect using MongoDB Shell**, Compass, or directly from applications.

   mongosh "mongodb+srv://cluster0.abcd.mongodb.net/myDatabase" --username myUser

Benefits of MongoDB Atlas

- No need to install or maintain servers.
- Provides automatic scaling and backups.
- Accessible from anywhere with an internet connection.
- Secure access control with authentication and IP whitelisting.

☑ Now you have both:

- Local MongoDB installed.
- Atlas configured and ready to connect from your app.

MongoDB Atlas Vs MongoDB Compass

MongoDB Atlas	**MongoDB Compass**
Hosts your database in the cloud	Lets you view & manage a database
Handles server maintenance	GUI tool for human interaction
Can run MongoDB without installing locally	Requires a MongoDB server to connect to
Cloud service with replication & backup	Desktop app for browsing/querying

☑ **Summary:**

- Use **Atlas** if you want a **managed MongoDB server in the cloud**.
- Use **Compass** if you want a **GUI to explore, query, and manage MongoDB databases** (Atlas or local).
- You can use them together: **Atlas hosts your database → Compass connects to it for easy management**.

b. AIM: Write MongoDB queries to perform CRUD operations on document using insert(), find(), update(), remove().

CRUD stands for **Create, Read, Update, and Delete**, which are the basic operations to manage data in a MongoDB collection. MongoDB provides built-in methods like insert(), find(), update(), and remove() to perform these operations.

Working with MongoDB queries in MongoDB compass (GUI Tool)

- Go to MongoDB compass Icon on the Desktop & open it.
- Click on "Add New Connection"
- Verify that it is displaying MongoDB URL with the port number connected.
- Give your preferred name & color and click on "Save & Connect".
- It will open the new connection window and click on any name of the database displayed
- It will show the option "open MongoDB command shell" click on it.
- The command shell for MongoDB is opened to work by opening with default database i.e. **test**

Example of working with Mongo DB queries:

Command: show databases **(**or shorthand **show dbs)**

-→

admin 40.00 KiB
config 108.00 KiB
local 72.00 KiB

- ✓ used to list all the databases present on the MongoDB server.
- ✓ When you run show databases (or shorthand show dbs) in the Mongo Shell, it displays a list of all the available databases on the server.
- ✓ Each database is shown along with its size on disk.
- ✓ A database will only appear in the list if it has at least one collection with some data. (Empty databases are not shown.)

Command: use school

---→ switched to db school

- ✓ The use command tells MongoDB shell which database you want to work with.
- ✓ If the database already exists, MongoDB switches to it.
- ✓ If the database does not exist yet, MongoDB creates it only after you insert data (like creating a collection or document).

A **collection in MongoDB is a container for documents, similar to a table in SQL.**

Key points:

- A **document** = one record (stored in JSON-like format, BSON).
- A **collection** = multiple related documents stored together.
- Unlike SQL tables, collections **don't require a fixed schema** → each document can have different fields.
- **Example:**

- { "name": "Raju", "age": 20 }
- { "name": "Sita", "class": "10th", "marks": 450 }

Both can exist in the same collection, even though fields differ

Use db.createCollection() → manually create an empty collection.

// Creating Collections

Command: db.createCollection('class10')

{ ok: 1 }

Command: db.createCollection('class9')

{ ok: 1 }

Command: show collections

used to list all the collections present in the current database.

→
class10
class9

1. Create (Insert Documents)

The insert() method is used to add new documents into a collection.

MongoDB provides different methods for inserting data:

1. **insertOne()** → Insert a single document
2. **insertMany()** → Insert multiple documents at once
3. **insert()** → Older version (can insert single/multiple, but now replaced by the above methods)

Command: db.class10.insertOne({'name':'Raju', 'rollno':1})

→
{
acknowledged: true,
insertedId: ObjectId('68b2b3c0f1189b6ebcbd4420')
}

Command : db.class10.insertOne({'name':'ramu','rollno':2,'phone':123456789})

→
{
acknowledged: true,
insertedId: ObjectId('68b2b48ef1189b6ebcbd4421')
}

Command: db.class10.insertOne({'name':'ramu','rollno':3})

→
{
acknowledged: true,
insertedId: ObjectId('68b2b722f1189b6ebcbd4422')
}

2. Read (Find Documents)

The find() method is used to retrieve documents from a collection.

What is find()?

- The **find()** method in MongoDB is used to **retrieve documents** (records) from a **collection**.
- By default, it returns **all documents** in the collection.
- You can also pass conditions (filters), projection (fields to show), sorting, and limits.

db.collection.find(query)
query → criteria/conditions to match documents (like WHERE in SQL).

// Find all documents

Command: db.class10.find()

→
```
{
  _id: ObjectId('68b2b3c0f1189b6ebcbd4420'),
  name: 'Raju',
  rollno: 1
}
{
  _id: ObjectId('68b2b48ef1189b6ebcbd4421'),
  name: 'ramu',
  rollno: 2,
  phone: 123456789
}

{
  _id: ObjectId('68b2b722f1189b6ebcbd4422'),
  name: 'ramu',
  rollno: 3
}
```

Command: show dbs

```
admin    40.00 KiB
config  108.00 KiB
local    72.00 KiB
mydb     72.00 KiB
school   80.00 KiB
```

// Find with a filter

- **db.class10.find({'rollno':2})**

// gives only one value

- **db.class10.findOne({'rollno':2})**

```
 {
  _id: ObjectId('68b2b48ef1189b6ebcbd4421'),
  name: 'ramu',
  rollno: 2,
  phone: 123456789
}
```

Command: db.class10.find()

```
 {
_id: ObjectId('68b2b48ef1189b6ebcbd4421'),
 name: 'ramu',
 rollno: 2,
 phone: 123456789
}
{
 _id: ObjectId('68b2b722f1189b6ebcbd4422'),
 name: 'ramu',
 rollno: 3
}
```

Command: db.class10.find({'name':'ramu'})

```
{
id: ObjectId('68b2b48ef1189b6ebcbd4421'),
 name: 'ramu',
 rollno: 2,
 phone: 123456789
}

{
 _id: ObjectId('68b2b722f1189b6ebcbd4422'),
 name: 'ramu',
 rollno: 3

}
```

3. Update (Modify Documents)

What is Update?

- The **update methods** in MongoDB are used to **modify existing documents** in a collection.
- Unlike insert (which adds new documents), update changes fields of documents that already exist.

◇ **Main Update Methods**

1. **updateOne()** → updates the **first matching document**.
2. **updateMany()** → updates **all matching documents**.
3. **replaceOne()** → replaces the entire document with a new one.
4. (Older method: update(), now mostly replaced by the above.)

Command: db.class10.updateOne({'rollno':2}, { $set: { 'phone': 1234567890 } })

Finds the document where **rollno = 2** and updates **phone to 1234567890**

4. Delete (Remove Documents)

The remove() method is used to delete documents from a collection.

Main Delete Methods

1. **deleteOne()** → removes the **first matching document**.
2. **deleteMany()** → removes **all documents** matching a condition.
3. (Old method: remove(), now replaced by the above two).

Command: db.class10.deleteOne({'name':'Raju'})

PROGRAM -13

MongoDB – Databases, Collections and Records

a) AIM: Write MongoDB queries to Create and drop databases and collections.

In MongoDB, databases and collections are the fundamental storage structures. Databases act as containers for collections, while collections store documents (records). To manage them, MongoDB provides specific commands:

1. Creating a Database

- MongoDB does not require an explicit CREATE DATABASE command.
- A database is created automatically when you switch to it using the use command and insert at least one document.
- MongoDB creates a database when you switch to it and insert at least one document.

// Switch to (or create) a database

use myDatabase

Output:

switched to db myDatabase

2. Drop a Database

To delete the currently active database:

// Drops the selected database

db.dropDatabase()

Output:

{ "dropped" : "myDatabase", "ok" : 1 }

3. Create a Collection

Collections are created either automatically when a document is inserted, or explicitly using createCollection().

// Explicitly create a collection

db.createCollection("students")

Output:

{ "ok" : 1 }

4. Drop a Collection

To remove a collection and all its documents:

// Drop the "students" collection

db.students.drop()

Output:

true

☑ **Summary:**

- Created a new database using use.
- **use <dbName>** → Creates/switches to a database.
- Created a collection with db.createCollection().
- **db.createCollection("name")** → Creates a collection.
- Verified collections with **show collections.**
- Dropped a collection using .drop().
- **db.collectionName.drop()** → Drops a collection.

Deleted the entire database using db.dropDatabase().

- **db.dropDatabase()** → Deletes a database.

❖ **Important Commands**

use mydb	-- switch to DB (doesn't create until write)
db.createCollection("c")	-- create empty collection (explicit)
db.c.insertOne({...})	-- insert → creates collection if needed
show dbs	-- list databases (only non-empty ones)
show collections	-- list collections in current DB
db.c.drop()	-- drop collection
db.dropDatabase()	-- drop current database
db.collection.deleteMany({})	-- remove all docs but keep collection

b)Write MongoDB queries to work with records using find(), limit(), sort(), createIndex(), aggregate().

In MongoDB, records are stored as documents inside collections. To query and manipulate these records, several methods are commonly used:

Step 1: Retrieve Records using find()

Task: Fetch all records from the students collection.

Query:

db.students.find()

Expected Output (sample):

{ "_id": 1, "name": "Ravi", "department": "CSE" }

{ "_id": 2, "name": "Anita", "department": "CSE" }

Step 2: Apply Conditions with find()

Task: Fetch only students from the CSE department.

Query:

db.students.find({ "department": "CSE" })

Expected Output (sample):

{ "_id": 3, "name": "Sita", "department": "CSE" }

{ "_id": 4, "name": "Kiran", "department": "CSE" }

Step 3: Limit Results using limit()

Task: Display only the first 3 records.

Query:

db.students.find().limit(3)

Expected Output:
(Only 3 student documents shown, even if more exist in collection)

Step 4: Sort Records using sort()

Task: Sort students by name in ascending order.

Query:

db.students.find().sort({ "name": 1 })

Expected Output (sample):

{ "_id": 7, "name": "Anita", "department": "ECE" }

{ "_id": 2, "name": "Kiran", "department": "CSE" }

{ "_id": 4, "name": "Ravi", "department": "CSE" }

Step 5: Create Index using createIndex()

Task: Create an index on the name field to speed up searches.

Query:

db.students.createIndex({ "name": 1 })

Expected Output:

"name_1"

Step 6: Aggregate Records using aggregate()

Task: Count the number of students in each department.

Query:

```
db.students.aggregate([
  { $group: { _id: "$department", total: { $sum: 1 } } }
])
```

Expected Output (sample):

{ "_id": "CSE", "total": 25 }

{ "_id": "ECE", "total": 18 }

☑ **Conclusion:**

☑ **Summary of Commands:**

- find() → Retrieve records.
- limit() → Restrict number of results.
- sort() → Order results.
- createIndex() → Improve query performance.
- aggregate() → Perform advanced computations like grouping, counting, summing.