FULL STACK DEVELOPMENT - 2

(23SO8554)

Laboratory Manual

B. Tech (PVP23)

(III Year- I semester)

Department of Computer Science and Engineering

PRASAD V POTLURI SIDDHARTHA INSTITUTE OF TECHNOLOGY

(Permanently affiliated to JNTU :: Kakinada, Approved by AICTE)

(An NBA & NAAC A+ accredited and ISO certified institution)

Kanuru, Vijayawada - 520007

Contents

ExP. No.	Experiment Title	Page No.
	Introduction	1
	Initial set up of instructions for MERN Stack Application Development	2
1	**Express JS – Routing, HTTP Methods, Middleware.** a. Write a program to define a route, Handling Routes, Route Parameters, Query Parameters and URL building. b. Write a program to accept data, retrieve data and delete a specified resource using http methods. c. Write a program to show the working of middleware.	7 11 13 16
2	**Express JS – Templating, Form Data** a. Write a program using templating engine. b. Write a program to work with form data.	18 20
3	**Express JS – Cookies, Sessions, Authentication** a. Write a program for session management using cookies and sessions. b. Aa program for user authentication.	22 25
4	**ReactJS – Render HTML, JSX, Components – function & Class** a. Write a program to render HTML to a web page. b. Write a program for writing markup with JSX. c. Write a program for creating and nesting components (function and class).	29 31 32
5	**ReactJS – Props and States, Styles, Respond to Events** a. Write a program to work with props and states. b. Write a program to add styles (CSS & Sass Styling) and display data. c. Write a program for responding to events.	34 36 38
6	**ReactJS – Conditional Rendering, Rendering Lists, React Forms** a. Write a program for conditional rendering. b. Write a program for rendering lists. c. Write a program for working with different form fields using react forms.	39 41 42

7	**ReactJS – React Router, Updating the Screen** a. Write a program for routing to different pages using react router. b. Write a program for updating the screen.	44 47
8	**ReactJS – Hooks, Sharing data between Components** a. Write a program to understand the importance of using hooks. b. Write a program for sharing data between components.	48 50
9	**MongoDB – Installation, Configuration, CRUD operations** a. Install MongoDB and configure ATLAS b. Write MongoDB queries to perform CRUD operations on document using insert(), find(), update(), remove()	52 54
10	**MongoDB – Databases, Collections and Records** a. Write MongoDB queries to Create and drop databases and collections. b. Write MongoDB queries to work with records using find(), limit(), sort(), createIndex(), aggregate().	56 57
11	**Express JS – Database, RESTful APIs** a. Write a program to connect MongoDB database using Mangoose and perform CRUD operations. b. Write a program to develop a single page application using RESTful APIs.	58 62
12	**CAPSTONE PROJECT**:Develop a project (student/faculty choice) using full stack technologies like react, node.js, express.js and mongoDB. Example (for reference): E-commerce application.	---

Introduction

The Full Stack Development with MERN technology, is a collection of JavaScript-based technologies used to develop web applications.

MERN is an acronym that stands for:

1. **MongoDB**: A NoSQL database that stores data in JSON-like documents. It is known

for its scalability and flexibility. stores the application's data.

2. **Express.js**: A web application framework for Node.js, designed for building web

applications and APIs. handles HTTP requests and responses, providing a robust set of features for web and mobile applications.

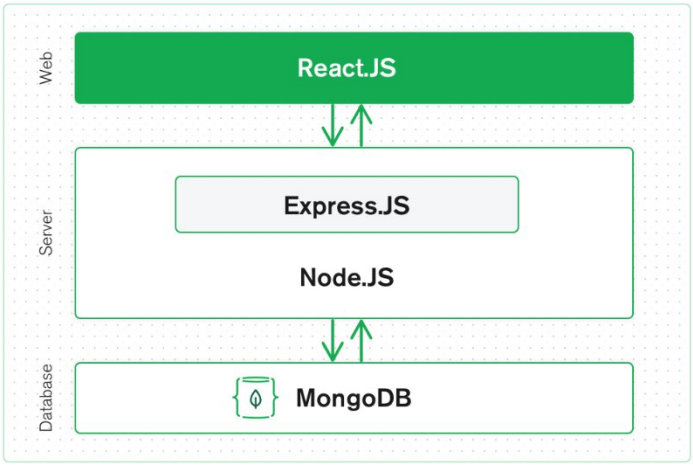
3. **React**: A front-end framework developed by Google for building dynamic, single-

page web applications (SPAs) with HTML, CSS, and Java Script. provides the client-side user interface, making the application dynamic and responsive.

4. **Node.js**: A JavaScript runtime built on Chrome's V8 JavaScript engine that allows

developers to execute JavaScript on the server side.

The MERN Architecture:



Initial set up of instructions for MERN Stack Application Development

1.Node.js Initial Setup Instructions

Step 1: Install Node.js and npm

On Windows / macOS / Linux:

1. **Download Installer**:
   - Go to the official Node.js website: https://nodejs.org
   - Download the **LTS (Long Term Support)** version.
2. **Run the Installer**:
   - Follow the installation steps.
   - Ensure that the box **"Automatically install npm"** is checked.
3. **Verify Installation**:
   Open your terminal (Command Prompt / Terminal / PowerShell) and run:

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

Step 2: Set Up Your First Node.js Project

1. **Create a Project Folder**:

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

2. **Initialize the Project**:

```
npm init -y
```

This creates a package.json file with default values.

Step 3: Create an Entry File

Create a file named index.js:

touch index.js # On macOS/Linux# orecho > index.js # On Windows

Add this simple code to index.js:

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

Run the file:

```
node index.js
```

2. Express installation setup

If you're building a web server:

```
npm install express
```

Create a simple Express app in index.js:

```
const express = require('express');
const app = express();
    app.get('/', (req, res) => {
    res.send('Hello from Express!');
});app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

Run the app:

```
node index.js
```

Open your browser and visit http://localhost:3000.

3.React.js Initial Setup Instructions

Prerequisites:

- Node.js and npm must be installed.

  node -v

  npm -v

Step 1: Create a React App

Use the official create-react-app tool:

npx create-react-app my-react-app

- npx comes with npm (5.2+), so no need to install create-react-app globally.
- my-react-app is the name of your folder/project.

This command:

- Creates a new directory my-react-app
- Sets up the project structure
- Installs React and dependencies
- Adds a development server and build scripts

Step 2: Navigate to the Project Folder

cd my-react-app

Step 3: Start the Development Server

npm start

This will:

- Open http://localhost:3000 in your default browser
- Show the default React welcome page

Step 4: Explore the Project Structure

Inside my-react-app, you'll see:

```
my-react-app/
├── node_modules/
├── public/
│   └── index.html
├── src/
│   ├── App.js
│   ├── index.js
│   └── ...
├── package.json
```

- **public/index.html** – Entry HTML file
- **src/index.js** – React entry point
- **src/App.js** – Main component

You can start editing App.js:

```
function App() {
  return (
    <div className="App">
      <h1>Hello, React!</h1>
    </div>
  );
}
```

Save the file — the browser reloads automatically.

Step 5: Install Additional Packages (Optional)

Examples:

```
npm install axios          # For HTTP requests
npm install react-router-dom # For routing
npm install bootstrap      # For styling
```

4.MongoDB Initial Setup Instructions

Node.js and npm installed (for backend integration with Express or Mongoose)

Step 1: Download and Install MongoDB

Go to the Official MongoDB Website:

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

1. Choose **MongoDB Community Server**
2. Select your OS (Windows/macOS/Linux)
3. Click **Download**
4. Run the installer with the **default settings**
5. Ensure the box **"Install MongoDB as a Service"** is checked

Step 2: Install MongoDB Compass (Optional GUI)

MongoDB Compass is a graphical interface to view and manage databases.

- Download from: https://www.mongodb.com/try/download/compass
- Install it separately if it isn't included with the main installer.

Step 3: Verify Installation

Open a terminal or command prompt and run:

```
mongod --version
mongo --version
```

Step 4: Start MongoDB Server

By default, MongoDB uses the path:

- **Data directory**: C:\data\db (Windows) or /data/db (Linux/macOS)

Create it manually if it doesn't exist:

```
# Windows
mkdir C:\data\db

# macOS/Linux
```

```
sudo mkdir -p /data/db
sudo chown -R `id -un` /data/db
```

Then start the MongoDB server:

```
mongod
```

In another terminal window, you can connect using:

```
mongo
```

This opens the MongoDB shell (CLI).

Step 5: Use MongoDB with Node.js (Optional for Backend)

1. Navigate to your Node project folder:

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

2. Install **Mongoose** (MongoDB ODM):

```
npm install mongoose
```

3. Example connection (index.js or db.js):

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

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

MongoDB Setup is Complete!

Experiment- 1: Express JS – Routing, HTTP Methods, Middleware

Aim:

To demonstrate the use of a Routing, HTTP methods and Middleware.

Concept	Purpose
Express.js	Framework to build web apps and APIs in Node.js
Routing	Define application behavior at different URLs
HTTP Methods	Perform CRUD operations
Middleware	Process requests before final route handlers

1. Express.js – Introduction

Express.js is a minimal and flexible Node.js web application framework used to build **RESTful APIs** and **web applications** easily.

Key Features:

- Lightweight and fast
- Simplifies server-side coding in Node.js
- Built-in support for routing, middleware, and templates

We can use Express.js :

- Handles HTTP requests efficiently
- Easily connects with databases like MongoDB
- Large ecosystem and community support

2. Routing in Express.js

Routing refers to defining how an application responds to client requests to specific endpoints (URLs) using HTTP methods.

- **Syntax:**

  app.METHOD(PATH, HANDLER)

  - **METHOD**: HTTP method (GET, POST, etc.)
  - **PATH**: URL endpoint
  - **HANDLER**: Function that handles the request

Example:

```
app.get('/home', (req, res) => {
  res.send('Welcome Home');
});
```

- **Types of Parameters:**
  - **Route Parameters**: /user/:id → accessed via req.params.id
  - **Query Parameters**: /search?q=book → accessed via req.query.q

3. HTTP Methods

These are standard verbs used in client-server communication, especially in RESTful APIs:

Method	Description
GET	Retrieve data
POST	Submit data (create)
PUT	Update an existing resource
PATCH	Partially update a resource
DELETE	Delete a resource

- **Example:**

```
// POST method
app.post('/books', (req, res) => {
  res.send('Book added');
});
```

4. Middleware in Express.js

Middleware is a function that has access to the request (req), response (res), and the next middleware function (next) in the request-response cycle.

Use Cases:

- Logging requests
- Authentication
- Parsing JSON
- Error handling

Example:

```
const logger = (req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // Pass control to next function
};
app.use(logger);
```

Program Code:

(a) Define Routes, Handle Route & Query Parameters, URL Building

Code: routes.js

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

// Route: Basic GET
app.get('/', (req, res) => {
  res.send('Welcome to the Home Page!');     // http://localhost:3000/
});

// Route with Route Parameters
app.get('/user/:username', (req, res) => {
  const username = req.params.username;
  res.send(`Hello, ${username}`);    // http://localhost:3000/user/Jane
});

// Route with Query Parameters
app.get('/search', (req, res) => {
  const keyword = req.query.q;
  res.send(`You searched for: ${keyword}`);   //  http://localhost:3000/search?q=nodejs
});

// URL Building Example
app.get('/build-url', (req, res) => {
  const name = 'Alex';                                  // Demonstrates building a URL dynamically.
const url = `/user/${name}?ref=lab`;    // Example response: "Built URL: /user/Alex?ref=lab"
  res.send(`Built URL: ${url}`);
});
```

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

Use a browser or Postman to test endpoints like:

- **http://localhost:3000/**
- **http://localhost:3000/user/John**
- **http://localhost:3000/search?q=express**
- **http://localhost:3000/build-url**

(b) Accept Data, Retrieve Data, Delete Resource (HTTP Methods)

Express.js code implements a simple in-memory Book API with the ability to:

- Add a book (POST)
- View all books (GET)
- Delete a book by ID (DELETE)

Code: crud.js

```
const express = require('express');
const app = express();
const port = 3000;
app.use(express.json()); // Middleware to parse JSON
let books = [
  { id: 1, title: "Book One" },
  { id: 2, title: "Book Two" }
];

// POST: Add a new book
app.post('/books', (req, res) => {
  const newBook = req.body;
  books.push(newBook);
  res.status(201).send('Book added successfully.');
});
// GET: Retrieve all books
app.get('/books', (req, res) => {
  res.json(books);
});
// DELETE: Delete book by ID
app.delete('/books/:id', (req, res) => {
  const bookId = parseInt(req.params.id);
  books = books.filter(book => book.id !== bookId);
```

```
  res.send(`Book with ID ${bookId} deleted.`);
});
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Test the application:

Use **Postman**, **curl**, or **any REST client**.

1.Start the server

Server running at http://localhost:3000

2.GET /books

URL: http://localhost:3000/books
Method: GET
Output (initially):

```
[
    { "id": 1, "title": "Book One" },
    { "id": 2, "title": "Book Two" }
]
```

3. POST /books

URL: http://localhost:3000/books
Method: POST
Body (JSON):

```
{
    "id": 3,
    "title": "Book Three"
}
```

OUTOUT

Book added successfully.

Resulting /books GET output:

```
[
    { "id": 1, "title": "Book One" },
    { "id": 2, "title": "Book Two" },
    { "id": 3, "title": "Book Three" }
]
```

DELETE /books/:id

URL: http://localhost:3000/books/2
Method: DELETE
Output:

Book with ID 2 deleted.

(c) Demonstrate Middleware Functionality

Express.js code demonstrates the use of custom middleware (logger) to log all incoming HTTP requests.

Code: middleware.js

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

// Custom Middleware
const logger = (req, res, next) => {
  console.log(`${new Date().toISOString()} - ${req.method} ${req.url}`);
next(); // Pass control to the next handler. next() is called to continue to the next middleware or
        route.
};

// Apply middleware to all routes
app.use(logger);

app.get('/', (req, res) => {
  res.send('Welcome with logging middleware!');
});

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

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

Output in Browser:

1. GET /

URL: http://localhost:3000/

Welcome with logging middleware!

GET /about

URL: http://localhost:3000/about
Browser Output:

About page

Expected Output:

- Route params: http://localhost:3000/user/Alex
- Query params: http://localhost:3000/search?q=express
- URL building shows dynamic path
- CRUD works for adding, viewing, deleting books
- Middleware logs all requests with timestamps

Experiment- 2: Express JS – Templating, Form Data

Aim:

To demonstrate the use of a **templating engine (EJS)** and handle **form data** in an Express.js application.

Task	Description
a. Templating	Used **EJS** to embed JavaScript in HTML
b. Form Data	Collected and processed form input using POST method

(a) Program Using Templating Engine (EJS)

Step 1: Install Required Packages

npm init -y

npm install express ejs

Step 2: Directory Structure

project-folder/

├── views/

│ └── index.ejs

├── app.js

Program Code: app.js

code sets up a simple Express.js server using EJS (Embedded JavaScript Templates) to render dynamic HTML pages

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

const app = express();

const port = 3000;

// Set EJS as templating engine

app.set('view engine', 'ejs'); // Tells Express to use EJS as the default view engine for rendering .ejs files from the views/ folder.

// Route to render template

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

  const name = 'Student';
```

```
  res.render('index', { username: name });});
// When you visit http://localhost:3000/, it renders an EJS template called index.ejs and passes { username: 'Student' } as data.
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Code: views/index.ejs

```
<!DOCTYPE html>
<html>
<head>
  <title>Welcome</title>
</head>
<body>
  <h1>Hello, <%= username %>!</h1>
</body>
</html>
```

Visit: http://localhost:3000 → Output: "Hello, Student!"

(b) Program to Work with Form Data

Step 1: Install Express + Body Parser Middleware (built-in)

npm install express

Code: formApp.js

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

// Middleware to parse URL-encoded form data
app.use(express.urlencoded({ extended: true }));

// Serve form
app.get('/', (req, res) => {
  res.send(`
    <form action="/submit" method="POST">
      Name: <input type="text" name="name"><br>
      Age: <input type="number" name="age"><br>
      <button type="submit">Submit</button>
    </form>
  `);
});

// Handle form submission
app.post('/submit', (req, res) => {
  const { name, age } = req.body;
  res.send(`Received submission: Name = ${name}, Age = ${age}`);
});
```

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

Expected Output

- GET / → Displays a form
- POST /submit → Displays the submitted name and age

In Browser Accessing /

URL: http://localhost:3000/
Output (form in browser):

Name: [_____]

Age: [_____]

[Submit]

Example Input:

Name: Alice

Age: 23

Response after submission:

Received submission: Name = Alice, Age = 23

Experiment- 3: Express JS – Cookies, Sessions, Authentication

Aim:

To demonstrate session management using **cookies and sessions**, and to implement basic **user authentication** in an Express.js application.

Concept	Description
Cookies	Store small data on client side
Sessions	Server-side storage linked with session ID
Authentication	Validates and restricts user access

(a) Session Management using Cookies and Sessions

Step 1: Install Dependencies

npm init -y

npm install express express-session cookie-parser

Code: sessionApp.js

Example of using sessions and cookies in an Express.js application to track how many times a user has visited the site during a session.

```
const express = require('express');
const session = require('express-session');
const cookieParser = require('cookie-parser');
const app = express();
const port = 3000;

// Middleware
app.use(cookieParser());       //  cookieParser() is required for reading cookies.
app.use(session({                  //  express-session stores session data server-side but sets a session ID cookie on the client.
  secret: 'mySecretKey',
  resave: false,
  saveUninitialized: true,
```

```
  cookie: { maxAge: 60000 } // 1 minute       // Session expires in 1 minute (maxAge: 60000 milliseconds).
}));

// Home route
app.get('/', (req, res) => {
  if (req.session.views) {
    req.session.views++;
    res.send(`<h1>Welcome back! Views: ${req.session.views}</h1>`);
  } else {
    req.session.views = 1;
    res.send(`<h1>Hello! First visit.</h1>`);
  }
});

// Clear session
app.get('/logout', (req, res) => {
  req.session.destroy();
  res.send('Session ended. <a href="/">Start Again</a>');
});

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

Output:

- Accessing / increases the session view count.
- Visiting /logout clears the session.

Routes and Expected Output

1. GET / – Track Views

First visit:

<h1>Hello! First visit.</h1>

Subsequent visits (within 1 minute):

<h1>Welcome back! Views: 2</h1>

<h1>Welcome back! Views: 3</h1>

...

2.GET /logout – Destroy Session

Output:

Session ended. Start Again

Test

- **Start server: node app.js**
- **Visit http://localhost:3000/**
- **Refresh the page several times within a minute to see Views increase.**
- **Visit http://localhost:3000/logout to clear the session.**
- **Return to / to see it reset to first visit.**

(b) User Authentication

Step 1: Install Dependencies

npm install express express-session body-parser

Express.js app demonstrates a simple and effective session-based authentication system. Here's a complete explanation of what the code does and what output to expect.

Code: authApp.js

```
const express = require('express');
const session = require('express-session');
const bodyParser = require('body-parser');
const app = express();
const port = 3000;

// Middleware
app.use(bodyParser.urlencoded({ extended: true }));
app.use(session({
  secret: 'authSecret',
  resave: false,
  saveUninitialized: true,
}));

// Dummy user (for demo)
const USER = { username: "admin", password: "1234" };

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

```
      <button type="submit">Login</button>
    </form>
  `);
});

// Authenticate user
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  if (username === USER.username && password === USER.password) {
    req.session.user = username;
    res.redirect('/dashboard');
  } else {
    res.send('Invalid credentials. <a href="/">Try again</a>');
  }
});

// Protected route
app.get('/dashboard', (req, res) => {
  if (req.session.user) {
    res.send(`<h2>Welcome ${req.session.user}!</h2><a href="/logout">Logout</a>`);
  } else {
    res.send('Access denied! Please <a href="/">login</a>.');
  }
});

// Logout route
app.get('/logout', (req, res) => {
  req.session.destroy();
  res.send('Logged out. <a href="/">Login again</a>');
});
```

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

This Does:

- / – Login form
- /login – Authenticates user
- /dashboard – Accessible only if logged in
- /logout – Ends session

1. GET / – Login Form

Browser Output:

```
<h2>Login</h2>
<form method="POST" action="/login">
  Username: <input type="text" name="username"><br>
  Password: <input type="password" name="password"><br>
  <button type="submit">Login</button>
</form>
```

2. POST /login – Authenticate

- Correct credentials (admin / 1234):
  Redirects to /dashboard
- Wrong credentials:
  Output:

    Invalid credentials. Try again

GET /dashboard – Protected Page

- **If logged in:**

    Welcome admin!

    Logout

If not logged in:

Access denied! Please login.

4.GET /logout – End Session

- Output:

Logged out. Login again

Destroys the session and resets the login state.

Testing Instructions:

1.Start the app:

node app.js

2.Open browser and go to:
http://localhost:3000/

3.Log in using:

Username: admin

Password: 1234

Visit /dashboard to see the protected content.

Click "Logout" or go to /logout.

Experiment – 4 ReactJS – Render HTML, JSX, Components (Function & Class)

Aim:

To demonstrate rendering HTML, using JSX, and creating React components using both function and class syntax.

Task Concept Demonstrated

a. Rendering static HTML in React

b. Writing and embedding logic in JSX

c. Creating and nesting function & class components

Prerequisite Setup (Run Once):

```
npx create-react-app react-lab
cd react-lab
npm start
```

You can now place all code inside the /src folder, typically in App.js.

(a) Render HTML to a Web Page

Code (inside App.js):

```
import React from 'react';
function App() {
  return (
    <div>
      <h1>Welcome to React!</h1>
      <p>This HTML is rendered using React.</p>
    </div>
  );
}
export default App;
```

Output in the Browser:

When this component is rendered (e.g., inside index.js with ReactDOM.render()), the browser will display:

Welcome to React!

This HTML is rendered using React.

Rendered HTML:

In the browser's "View Source" or DevTools, you'll see:

```
<div>
  <h1>Welcome to React!</h1>
  <p>This HTML is rendered using React.</p>
</div>
```

(b) Write Markup with JSX

JSX (JavaScript XML) allows writing HTML-like syntax in JavaScript, which React compiles to React.createElement() calls.

Code (inside App.js):

React component will render dynamic content based on variables and conditions.

```
import React from 'react';
function App() {
  const name = "React Learner";
  const isStudent = true;

  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>{isStudent ? "You are a student." : "You are not a student."}</p>
    </div>
  );
}

export default App;
```

Output:

Dynamically rendered JSX based on JavaScript values.

You'll See in the Browser:

If isStudent is true (as in your code):

Hello, React Learner!

You are a student.

If you change isStudent to false, it would show:

Hello, React Learner!

You are not a student.

(c) Create and Nest Components (Function & Class)

Tdemonstrate component composition in React — where one component includes and renders other

App.js:

```
import React from 'react';
import Welcome from './Welcome';
import Message from './Message';
function App() {
  return (
    <div>
      <h1>Main Component</h1>
      <Welcome name="Alice" />
      <Message />
    </div>
  );
}

export default App;
```

Welcome.js – Function Component

```
import React from 'react';

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

export default Welcome;
```

Message.js – Class Component

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

class Message extends Component {
  render() {
    return <p>This is a message from a class component.</p>;
  }
}

export default Message;
```

Output:

The browser displays nested components from both functional and class components.

Main Component

Welcome, Alice!

This is a message from another component.

Experiment – 5 ReactJS – Props, State, Styles, Event Handling

Aim:

To demonstrate the use of **props**, **state**, **CSS/SASS styling**, and **event handling** in a React application.

Task Concept Demonstrated

a. Props and State interaction

b. CSS & SASS styling, dynamic data rendering

c. Handling button click events in React

(a) Program to Work with Props and States

App.js

React example demonstrates how useState and props work together across components.

```
import React, { useState } from 'react';
import Greeting from './Greeting';

function App() {
  const [name, setName] = useState("Alice");

  return (
    <div>
      <h1>React Props & State Example</h1>
      <Greeting userName={name} />
      <button onClick={() => setName("Bob")}>Change Name</button>
    </div>
  );
}
```

```
export default App;
```

Greeting.js

```
import React from 'react';

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

export default Greeting;
```

Output:

Displays Hello, Alice!
On clicking the button → changes to Hello, Bob!

Output in the Browser

Initially:

React Props & State Example

Hello, Alice!

[Change Name]

After clicking the **Change Name** button:

React Props & State Example

Hello, Bob!

[Change Name]

Concept Example Used

useState const [name, setName] = useState("Alice")

Props <Greeting userName={name} />

State change onClick={() => setName("Bob")}

Component reuse <Greeting /> in separate file

(b) Program to Add Styles (CSS & Sass) and Display Data

Step 1: Add CSS file

App.css

```
.container {
  background-color: #f0f0f0;
  padding: 20px;
  border-radius: 10px;
  text-align: center;
}

.title {
  color: #007acc;
  font-family: Arial, sans-serif;
}
```

App.js

```
import React from 'react';
import './App.css';

function App() {
  const student = {
    name: "John",
    age: 20,
    course: "ReactJS"
  };
```

```
  return (
    <div className="container">
      <h1 className="title">Student Info</h1>
      <p>Name: {student.name}</p>
      <p>Age: {student.age}</p>
      <p>Course: {student.course}</p>
    </div>
  );
}

export default App;
```

(Optional) Sass Setup:

Install SASS:

npm install sass

Then rename your CSS file as App.scss and import it:

import './App.scss';

(c) Program to Respond to Events

React code demonstrates a simple and clear example of event handling with state management using hooks. Here's what it does, step-by-step:

App.js

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

function App() {
  const [count, setCount] = useState(0);
  const handleClick = () => {
    setCount(count + 1);
  };
  return (
    <div style={{ textAlign: "center", padding: "40px" }}>
      <h1>Event Handling Example</h1>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>Click Me</button>
    </div>
  );
}

export default App;
```

Experiment – 6 ReactJS – Conditional Rendering, Rendering Lists, React Forms

Aim:

To demonstrate:

1. **Conditional Rendering**
2. **Rendering Lists**
3. **Handling Forms** with various input types in ReactJS

Concept	Description
Conditional Rendering	Display UI based on state/conditions
List Rendering	Use .map() to display multiple items
React Forms	Handle input, dropdowns, checkboxes with state

(a) Program for Conditional Rendering

App.js

```
import React, { useState } from 'react';
function App() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);
  return (
    <div style={{ textAlign: "center", padding: "30px" }}>
      <h1>Conditional Rendering</h1>
      {isLoggedIn ? (
        <h2>Welcome, User!</h2>
      ) : (
        <h2>Please log in.</h2>
      )}
```

```
    <button onClick={() => setIsLoggedIn(!isLoggedIn)}>
      {isLoggedIn ? "Logout" : "Login"}
    </button>
  </div>  );} export default App;
```

Output:

- Displays different messages based on login status.
- Button toggles between login and logout view.

(b) Program for Rendering Lists

App.js

```
import React from 'react';

function App() {
  const students = ["Alice", "Bob", "Charlie", "Daisy"];

  return (
    <div style={{ padding: "20px" }}>
      <h1>Student List</h1>
      <ul>
        {students.map((student, index) => (
          <li key={index}>{student}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Output:

- Displays a list of student names using .map()

(c) Program for Working with React Forms

App.js

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

function App() {
  const [formData, setFormData] = useState({
    name: "",
    age: "",
    gender: "",
    course: "",
    agree: false
  });

  const handleChange = (e) => {
    const { name, value, type, checked } = e.target;
    setFormData(prev => ({
      ...prev,
      [name]: type === 'checkbox' ? checked : value
    }));
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Submitted: ${JSON.stringify(formData, null, 2)}`);
  };
```

```
  return (
    <div style={{ maxWidth: "400px", margin: "auto", padding: "20px" }}>
      <h1>React Form Example</h1>
      <form onSubmit={handleSubmit}>
        <label>Name: <input type="text" name="name" value={formData.name}
onChange={handleChange} /></label><br /><br />
        <label>Age: <input type="number" name="age" value={formData.age}
onChange={handleChange} /></label><br /><br />
        <label>Gender:
          <select name="gender" value={formData.gender} onChange={handleChange}>
            <option value="">Select</option>
            <option value="Male">Male</option>
            <option value="Female">Female</option>
          </select>
        </label><br /><br />
        <label>Course: <input type="text" name="course" value={formData.course}
onChange={handleChange} /></label><br /><br />
        <label>
          <input type="checkbox" name="agree" checked={formData.agree}
onChange={handleChange} />
          I agree to the terms
        </label><br /><br />
        <button type="submit">Submit</button>
      </form>
    </div>
  );
}

export default App;
```

Output:

- Form with text, number, dropdown, checkbox
- Form state is managed via useState

- Displays submitted data via alert()

Experiment- 7 ReactJS – React Router & Updating the Screen

Aim:

To demonstrate:

1. Page navigation using **React Router**
2. **Dynamic screen updates** using component state

Concept	Description
React Router	Enables navigation between components/pages
Updating Screen	Uses state to re-render parts of the UI dynamically

(a) Program for Routing to Different Pages Using React Router

Step 1: Install React Router

npm install react-router-dom

Directory Structure

```
src/
├── pages/
│   ├── Home.js
│   ├── About.js
│   └── Contact.js
├── App.js
```

✔ App.js

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

```
import Home from './pages/Home';
import About from './pages/About';
import Contact from './pages/Contact';
function App() {
  return (
    <BrowserRouter>
      <nav style={{ margin: "20px" }}>
        <Link to="/">Home</Link> |{" "}
        <Link to="/about">About</Link> |{" "}
        <Link to="/contact">Contact</Link>
      </nav>

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

export default App;
```

pages/Home.js

```
import React from 'react';
function Home() {
  return <h2>🏠 Welcome to the Home Page</h2>;
}
export default Home;
```

pages/About.js

```
import React from 'react';
function About() {
  return <h2>i About Us Page</h2>;
}
export default About;
```

pages/Contact.js

```
import React from 'react';
function Contact() {
  return <h2> Contact Page</h2>;
}
export default Contact;
```

Output:

- Navigates between pages without refreshing the browser
- URL updates according to the route

(b) Program for Updating the Screen

React updates the UI automatically based on **state changes**.

App.js

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

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

  return (
    <div style={{ textAlign: "center", padding: "50px" }}>
      <h1>Update Screen Example</h1>
      <p>Button clicked: <strong>{count}</strong> times</p>
      <button onClick={() => setCount(count + 1)}>Click Me</button>
    </div>
  );
}

export default App;
```

Output:

- Each click updates the count on the screen **without refreshing**

Experiment - 8 ReactJS – Hooks & Sharing Data Between Components

Aim:

1. To demonstrate how **Hooks** like useState and useEffect manage component state and side effects.
2. To demonstrate how to **share data** between components using **props** and **lifting state up**.

Concept	Description
useState	Hook to manage local state
useEffect	Hook to handle side effects (like DOM updates)
Sharing Data via Props	Pass data from parent to child
Lifting State Up	Move shared state to the closest common ancestor

(a) Program to Understand the Importance of Hooks

App.js

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

function App() {
  const [count, setCount] = useState(0); // useState hook

  // useEffect hook - runs after component renders
  useEffect(() => {
    document.title = `Clicked ${count} times`;
  }, [count]);

  return (
```

```
    <div style={{ textAlign: "center", padding: "40px" }}>
      <h1>React Hooks Demo</h1>
      <p>You clicked <strong>{count}</strong> times</p>
      <button onClick={() => setCount(count + 1)}>Click Me</button>
    </div>
  );
}

export default App;
```

Explanation:

- useState() allows function components to hold and update state.
- useEffect() lets you perform side effects (like updating document title) after render.
- Before hooks, these features were only possible in **class components**.

(b) Program for Sharing Data Between Components

App.js

```
import React, { useState } from 'react';
import Child from './Child';

function App() {
  const [message, setMessage] = useState("Hello from Parent!");

  const updateMessage = () => {
    setMessage("Message updated by Parent");
  };

  return (
    <div style={{ padding: "20px" }}>
      <h2>Parent Component</h2>
      <button onClick={updateMessage}>Update Message</button>
      <Child message={message} />
    </div>
  );
}

export default App;
```

Child.js

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

```
function Child({ message }) {
  return (
    <div style={{ border: "1px solid gray", padding: "15px", marginTop: "20px" }}>
      <h3>Child Component</h3>
      <p>Received Message: <strong>{message}</strong></p>
    </div>
  );
}

export default Child;
```

Explanation:

- App (Parent) holds the state.
- Child receives data as **props**.
- This is an example of **lifting state up** to the parent to share it with children.

Experiment – 9 MongoDB – Installation, Configuration, CRUD Operations

Aim:

To:

1. Install and configure **MongoDB (Atlas)**
2. Perform **CRUD operations** using MongoDB shell queries

Operation	Command Example
Insert	insert(), insertMany()
Read	find(), find().pretty()
Update	update(), updateMany()
Delete	remove()

(a) Install MongoDB and Configure Atlas

Option 1: Install MongoDB Locally

1. Go to: https://www.mongodb.com/try/download/community
2. Choose your OS and install **MongoDB Community Edition**
3. Start MongoDB server:
4. mongod
5. Open MongoDB shell in another terminal:
6. mongosh

Option 2: Setup MongoDB Atlas (Cloud)

1. Visit: https://cloud.mongodb.com
2. Sign up or log in

3. Create a **Free Shared Cluster**
4. Choose:
    - Provider: AWS/GCP/Azure
    - Region: Closest to your location
5. **Create a database user** (username & password)
6. Add your IP Address (0.0.0.0 to allow from anywhere for testing)
7. Connect to your cluster:
    - Choose "Connect your application"
    - Copy the **MongoDB connection URI** (e.g. mongodb+srv://<username>:<password>@cluster0.mongodb.net/test)

(b) CRUD Operations using MongoDB Shell

1. Insert Documents

```
use studentdb

db.students.insert({
  name: "John Doe",
  age: 21,
  course: "ReactJS"
})

db.students.insertMany([
  { name: "Alice", age: 20, course: "MongoDB" },
  { name: "Bob", age: 22, course: "NodeJS" }
])
```

2. Read / Retrieve Documents

```
// Find all documents
db.students.find()

// Find with filter
db.students.find({ course: "ReactJS" })

// Pretty print
db.students.find().pretty()
```

3. Update Documents

```
// Update one document
db.students.update(
  { name: "John Doe" },
  { $set: { course: "ExpressJS" } }
)

// Update many documents
db.students.updateMany(
  { course: "ReactJS" },
  { $set: { course: "Full Stack" } }
)
```

4. Delete Documents

```
// Remove one document
db.students.remove({ name: "Alice" })

// Remove all documents
db.students.remove({})
```

Sample Output

After running:

```
db.students.find().pretty()
```

You may get:

```
{
  "_id" : ObjectId("..."),
  "name" : "Bob",
  "age" : 22,
  "course" : "NodeJS"
```

}

Experiment- 10 MongoDB – Databases, Collections, and Records

Aim:

1. To create and drop databases and collections in MongoDB.
2. To perform record operations using find(), limit(), sort(), createIndex(), and aggregate().

Operation	Command Example
Create DB	use dbname
Drop DB	db.dropDatabase()
Create Collection	db.createCollection("name")
Drop Collection	db.collection.drop()
Insert Record	insert(), insertMany()
Read	find(), limit(), sort()
Index	createIndex({ field: 1 })
Aggregation	$group, $avg, $sum using aggregate()

(a) Create & Drop Databases and Collections

1. Create a Database

use librarydb

This creates librarydb if it doesn't already exist and switches to it.

2. Create a Collection

db.createCollection("books")

This creates a collection named books.

3. Drop a Collection

```
db.books.drop()
```

4. Drop a Database

```
use librarydb
db.dropDatabase()
```

(b) Working with Records in a Collection

Let's re-create the books collection with some data:

```
use librarydb

db.books.insertMany([
  { title: "MongoDB Basics", author: "John", year: 2022, price: 500 },
  { title: "Learning React", author: "Alice", year: 2021, price: 650 },
  { title: "Node.js Guide", author: "Bob", year: 2023, price: 400 },
  { title: "Express Handbook", author: "Jane", year: 2020, price: 300 }
])
```

1. find() – Retrieve Records

```
db.books.find()
db.books.find({ author: "Alice" })
```

2. limit() – Limit the Number of Records

```
db.books.find().limit(2)
```

🔁 3. sort() – Sort Records

```
// Ascending by price
db.books.find().sort({ price: 1 })

// Descending by year
db.books.find().sort({ year: -1 })
```

4. createIndex() – Create Index on Fields

db.books.createIndex({ title: 1 })

Indexing improves query performance on frequently accessed fields.

5. aggregate() – Perform Aggregation Operations

Example: Group books by year and count them:

```
db.books.aggregate([
  { $group: { _id: "$year", count: { $sum: 1 } } }
])
```

Example: Get average price of all books:

```
db.books.aggregate([
  { $group: { _id: null, avgPrice: { $avg: "$price" } } }
])
```

Experiment – 11 Express JS – Database, RESTful APIs

Aim:

1. Connect Express.js with MongoDB using **Mongoose** and perform **CRUD operations**.
2. Build a **single-page application (SPA)** that uses **RESTful APIs** to manage data.

Task	Technology Used
Backend (CRUD)	Express.js, MongoDB, Mongoose
Frontend (SPA)	ReactJS
Communication	RESTful API (Axios, Fetch)

(a) Connect MongoDB using Mongoose and Perform CRUD Operations

Step 1: Initial Setup

```
mkdir express-mongo-crud
cd express-mongo-crud
npm init -y
npm install express mongoose body-parser
```

✔ Step 2: Create server.js

```
// server.js
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');

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

```
// MongoDB connection
mongoose.connect('mongodb://localhost:27017/studentdb', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

const studentSchema = new mongoose.Schema({
  name: String,
  course: String,
  marks: Number,
});

const Student = mongoose.model('Student', studentSchema);

// Create - POST
app.post('/students', async (req, res) => {
  const student = new Student(req.body);
  await student.save();
  res.send(student);
});

// Read all - GET
app.get('/students', async (req, res) => {
  const students = await Student.find();
  res.send(students);
});

// Read by ID - GET
app.get('/students/:id', async (req, res) => {
  const student = await Student.findById(req.params.id);
```

```
  res.send(student);
});

// Update - PUT
app.put('/students/:id', async (req, res) => {
  const student = await Student.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.send(student);
});

// Delete - DELETE
app.delete('/students/:id', async (req, res) => {
  await Student.findByIdAndDelete(req.params.id);
  res.send({ message: 'Student deleted' });
});

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

Output:

- Use **Postman** or any REST client:
    - POST /students to add
    - GET /students to view all
    - GET /students/:id to view by ID
    - PUT /students/:id to update
    - DELETE /students/:id to delete

(b) Single Page Application using RESTful APIs (with React)

You can use the API from (a) in a React frontend like:

Sample React Code (App.jsx)

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

function App() {
  const [students, setStudents] = useState([]);
  const [form, setForm] = useState({ name: '', course: '', marks: '' });

  useEffect(() => {
    axios.get('http://localhost:3000/students')
      .then(res => setStudents(res.data));
  }, []);

  const handleSubmit = (e) => {
    e.preventDefault();
    axios.post('http://localhost:3000/students', form)
      .then(res => setStudents([...students, res.data]));
  };

  return (
    <div>
      <h2>Student Management</h2>
      <form onSubmit={handleSubmit}>
        <input placeholder="Name" onChange={e => setForm({...form, name: e.target.value})}
/>
```

```
      <input placeholder="Course" onChange={e => setForm({...form, course:
e.target.value})} />
      <input placeholder="Marks" type="number" onChange={e => setForm({...form, marks:
e.target.value})} />
      <button type="submit">Add</button>
    </form>

    <ul>
      {students.map(s => (
        <li key={s._id}>{s.name} - {s.course} - {s.marks}</li>
      ))}
    </ul>
  </div>
 );
}

export default App;
```

Don't forget to allow **CORS** in server.js if testing with React:

npm install cors

Then add to server.js:

```
const cors = require('cors');
app.use(cors());
```