ADVANCED WEB TECHNOLOGIES LAB MANUAL

FOR

FOR B. TECH III YEAR II SEMESTER

IT



2024-2025

CVR COLLEGE OF ENGINEERING

(AUTONOMOUS)

Vastunagar, Mangalpalli(V), Ibrahimpatan (M),

R.R. District. Pin : 501510

Email : [info@cvr.ac.in](mailto:info@cvr.ac.in)

Web: [http://cvr.ac.in](http://cvr.ac.in/)

**CVR COLLEGE OF ENGINEERING**

**VISION**

➢ To be a state-of-the-art institution of engineering in pursuit of excellence, in the service of society.

**MISSION**

➢ To excel in providing quality education at under graduate and graduate levels.

➢ To encourage research and innovation.

➢ To provide infrastructure and facilities to meet the latest technological needs.

➢ To establish Centers of Excellence through active interaction with industry.

➢ To nurture students towards holistic development with human values and ethics.

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**Vision**

➢ The Department aims to produce quality Software Engineers and Data Scientists, with an attitude to adapt to ever changing IT needs of local, national and international arena, through teaching, interactions with alumni and industry.

➢**Mission**

**M1**: To provide a holistic learning environment for students and to create awareness in legal and ethical practices.

**M2**: To provide quality infrastructure through labs, other resources and to continuously upgrade to the latest technology requirements.

**M3**: To train the students in soft skills to excel in placements and competitive exams at higher levels.

**M4**: To develop into a Centre of excellence for application development using open-source technologies and contribute to the open-source community.

**M5**: To have a healthy industry - institute interaction through faculty development programs, student internships, guest lectures and so on.

**M6**: To provide a research-oriented environment for the faculty to meet the ever-changing societal needs.

**M7**: To train the PG students to be able to carry out research and development work to meet the societal needs.

**M8**: To provide effective consultancy to meet the industrial requirement.

**Program Educational Objectives (PEOs)**

**PEO 1**: Graduates will acquire capability to apply their knowledge and skills to

solve various kinds of computational engineering problems.

**PEO 2**: Graduates will be in a position to recognize and incorporate societal

needs and practice their profession with high regard to legal and

ethical practices.

**PEO 3**: Graduates will be ready to work in projects related to complex

problems involving multi-disciplinary areas.

**PEO 4:** To evolve as resourceful engineers catering to global changes and engage in life-long learning.

**PEO 5**: To enable the students with required soft skills, that can be used in a pragmatic manner and excel in diverse environments in the competitive world.

**Program Outcomes (POs)**

**Engineering Graduates will be able to:**

**1.** **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

**2. Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

**3. Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

**4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

**5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

**6. The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

**7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

**8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

**9. Individual and teamwork:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

**10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

**11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

**12. Life-long learning:** Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

**Program Specific Outcomes (PSOs)**

**PSO1:** An ability to analyze a problem, design algorithm, identify and define the computing requirements within realistic constraints and implement through analytical, logical and problem-solving skills.

**PSO2:** Ability to design and develop prototype innovatively leading to product development and contributions to open-source community meeting ever changing societal needs.

**PSO3:** Effectively integrate IT-based solutions into the user environment.

**PSO4:** Ability to pursue higher studies and research in inter-disciplinary areas.

**Course Outcomes**

At the end of the course, the student should be able to

**CO 1:** Proficient in leveraging ES6 capabilities to streamline development tasks, improving code efficiency, and solving programming challenges with a modern JavaScript mindset and managing various versions of the product using git.

**CO 2 :** Able to develop the skills to construct scalable and maintainable web applications by harnessing the asynchronous, event-driven nature of Node.js and the modular architecture provided by Express.

**CO 3 :** Designing and implementing efficient NoSQL database solutions, understanding when and how to use document-oriented databases like MongoDB to address specific application requirements.

**CO 4 :** Developing the dynamic and single-page web applications using the MEAN stack and understand how to manage client-side routing, and UI components to deliver a seamless user experience.

**CO 5 :** Excelling in both server-side and client-side aspects of web application development using MERN stack.

**Course to PO, PSO Mapping**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Course | PO1 | PO2 | PO3 | PO4 | PO5 | PO6 | PO7 | PO8 | PO9 | PO10 | PO11 | PO12 | PSO1 | PSO2 | PSO3 | PSO4 |
| 22IT382 | 1 | 2 | 1 | 3 | 3 |  |  |  |  | 2 |  | 1 | 2 | 3 | 2 | 3 |

**Course Outcomes to PO and PSO Mapping:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Course Outcome | P01 | P02 | P03 | P04 | P05 | P06 | P07 | P08 | P09 | P010 | P011 | P012 | PSO1 | PSO2 | PSO3 | PSO4 |
| CO 1 | 1 | 2 |  | 3 | 2 |  |  |  |  |  |  | 1 | 1 | 3 | 1 |  |
| CO 2 | 2 | 1 | 1 | 3 | 3 |  |  |  |  |  |  | 1 | 1 | 2 | 2 | 2 |
| CO3 | 1 | 2 | 1 | 2 | 2 |  |  |  |  |  |  | 2 | 2 | 1 | 2 |  |
| CO4 |  | 2 |  | 3 | 3 |  |  |  |  | 2 |  | 1 | 2 | 3 | 2 | 3 |
| CO5 | 1 | 2 | 2 | 3 | 3 |  |  |  |  | 2 |  | 1 | 2 | 2 | 2 | 3 |

**Note: 1- Slight 2- Moderate 3- Substantial**

Course Code: 22IT382

**ADVANCED WEB TECHNOLOGIES LAB (IT)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Instruction | : | 3 Periods / week | Sessional Marks | : | 40 |
| Tutorial | : | - | End Examination Marks | : | 60 |
| Credits | : | 1 | End Exam Duration | : | 3 Hours |

**Prerequisites:** A course on “Web Technologies”

**Course Objectives:**

1. To enhance code readability, maintainability, and developer productivity by adopting ES6 features and explore version control system.
2. To gain proficiency and expertise in utilizing NPM (Node Package Manager) and developing RESTful APIs using Express framework.
3. To explore how NoSQL databases, particularly MongoDB, can be leveraged to design scalable and flexible data solutions.
4. To Gain proficiency in front-end development using Angular.
5. To provide the knowledge of building modular and reusable components and create interactive user interfaces for MERN stack applications.

**List of Experiments:**

**TASK 1: Managing the versions of the product using Version Control System (GIT)**

a. Create a git repository local add and commit a simple web application consisting of 5

pages.

b. Create a remote repository in github.com. Push the above local repository to the

github.com. Explore the push, pull and fetch options with remote repository.

c. Clone a remote repository into local directory, modify the implementation and push

the updated changes back to remote repository.

d. Create branch and manage the work distributions. Merge all the branches and commit

the changes.

e. Publish the application using Github pages.

**TASK 2: Implementing the advanced features of JavaScript**

a. Working with Prototypal Inheritance and Classes.

b. Working with Object and Array Destructuring.

c. Working with Modules.

d. Working with Function Generators and Symbols.

e. Working with Closure.

**TASK 3: Asynchronous Execution of JavaScript**

a. Working with higher order function in JavaScript.

b. Using Callback and creating a Callback Hell situation to understand the drawbacks.

c. Working with XHR: response.

d. Dealing with the Callback Hell situation using Promise. Exploring the different

ways of creating and using promise in executing the asynchronous task.

e. Dealing with Promise chaining and async / await.

**TASK 4: Fetching server state using JavaScript (fetch)**

a. Use fetch function to access remote data using the given api and display the data in the

form of a table.

b. Use fetch function to read the weather details from openweathermap.org and display

the details like city, min-temp, max-temp, humidity on the webpage for a given city.

c. From the same website read the weather forecast details for a given city and display

the details like date – temperature in a table.

d. Plot a bar chart for the above implementation using date and temperature along X and

Y axis respectively. Use ChartJS library.

**TASK 5: Node JS**

a. Create custom / local modules and export them using various module patterns.

b. Explore the functionality of os, path, util and events modules.

c. Use the fs module for creating directories and files of different formats.

d. Write script to read and write the streaming data using readable and writable streams.

**TASK 6: Working with http**

a. Create a http server listening request at port 3000. Process the request to provide

different type of resources as response. (HTML, TEXT, JSON, etc.).

b. Create express server listening request at port 3000. Add different endpoints to

provide access to the resources.

**TASK 7: Working with Express**

a. Create a custom API for Users data and add different endpoints in express server to

perform CRUD operations on the API. Test the endpoints using POSTMAN.

b. Use EJS view-engine to display the dynamic response. Display the data read from REST

API in the form of a table in EJS.

**TASK 8: Working with model (MongoDB & Firebase)**

a. Create a database for Users in Mongo DB. Create User collection and documents to the

User collection. Perform all DB operations (CREATE, READ, UPDATE and DELETE) on

the User collection.

b. Create a real time database in firebase for the student management system and explore

the features of Firebase Real Time Database. Perform CRUD operations on the Real

Time Database.

**TASK 9: Working with Express & Mongo DB**

Create express server that has endpoints connecting to Users collection present in

Mongo DB database using mongoose library and perform CRUD operation on that.

**TASK 10: Authentication and Authorization**

a. Create express server that has authorized endpoint using JWT (JSON Web Token)

library.

b. Create express server that connects to Mongo DB database to authenticate the user

and generate the authorized token to access the protected endpoints.

**TASK 11: MEAN Stack Development**

a. Create a user profile management system where users can update their profiles,

including details like name, email, phone. Angular's data binding ensures that changes

made by users are instantly reflected in the UI.

b. Develop an angular application that interacts with the backend API and executes CRUD

operations on it.

**TASK 12: Angular Routing**

a. Develop angular application consisting of App, Home, About, Contact, Profile, Login

and Register Components. Add a Navigation bar and navigate to the respective

component using angular routing.

b. Develop a Single Page Application in Angular for User Management System that

interacts with the backend database created in Task 8. Use Services and HttpClient

to access the express endpoints of Task 9.

**TASK 13: MERN Stack Development**

a. Create react functional and class components. Implement the lifecycle methods of

react component.

b. Develop react application with App, Home, About and Contact components. Implement

the use of react props and state in these components.

**TASK 14: React Routing**

a. Develop a react application that demonstrates the routing feature to navigate across

different components of react and pass the data in between the components.

b. Develop a SPA in react for User Management System that interacts with the backend

API using axios and perform CRUD operations on that.

**TASK 15: Single Page Application in Angular / React**

A TODO application serves as a simple yet powerful tool to help individuals and teams organize their tasks, manage priorities, and enhance productivity. TODO applications provide a structured and efficient way for individuals and teams to manage tasks, prioritize work, and achieve their goals. Develop a Single Page TODO Application in Angular / React to manage the daily tasks with the following features:

**a. Task Creation:** Allow users to create new tasks with a title, description, due date,

and priority level. Provide a straightforward interface for entering task details.

**b. Task Listing:** Display a list of all tasks with essential details. Tasks can be organized

based on different criteria such as due date, priority, or completion status.

**c. Task Editing and Updating:** Enable users to edit task details, including the ability to

modify the title, description, due date, and priority. Changes should be reflected in

real-time.

**d. Task Deletion:** Provide the option to delete tasks that are no longer relevant or

completed. Include a confirmation prompt to prevent accidental deletions.

**e. Task Completion:** Allow users to mark tasks as completed or mark them with a

specific status. Completed tasks may be moved to a separate section or visually

differentiated.

**f. User Authentication and Authorization:** Implement user accounts with

authentication to ensure data privacy. Differentiate between users and provide

appropriate authorization levels.

**g. Data Persistence:** Ensure that tasks are persistently stored, so users can access their

TODO lists even after closing and reopening the application.

**(NOTE: TASK 1 – TASK 14 are mandatory to complete in the labs)**

**Course Outcomes:** At the end of the course, the student will be able to

CO 1 : Proficient in leveraging ES6 capabilities to streamline development tasks, improving

code efficiency, and solving programming challenges with a modern JavaScript

mindset and managing various versions of the product using git.

CO 2 : Able to develop the skills to construct scalable and maintainable web applications

by harnessing the asynchronous, event-driven nature of Node.js and the modular

architecture provided by Express.

CO 3 : Designing and implementing efficient NoSQL database solutions, understanding

when and how to use document-oriented databases like MongoDB to address

specific application requirements.

CO 4 : Developing the dynamic and single-page web applications using the MEAN stack

and understand how to manage client-side routing, and UI components to deliver a

seamless user experience.

CO 5 : Excelling in both server-side and client-side aspects of web application development

using MERN stack.

**TEXT BOOKS:**

1. 1. MEAN Web Development, Amos Q. Haviv, Second Edition, Packt Publications,

November 2016.

1. “Pro MERN Stack, Full Stack Web App Development with Mongo, Express, React,

and Node”, Vasan Subramanian, 2nd Edition, APress, 2017.

**REFERENCE BOOKS:**

1. Learning Node: Moving to the Server-Side, Shelly Powers, 2nd Edition, O’REILLY, 2016.
2. Getting MEAN with Mongo, Express, Angular, and Node, Simon D. Holmes and Clive Harber, Second Edition, Manning Publications, 2019.
3. Node.js, MongoDB and Angular Web Development, Brad Dayley, “2nd Edition, Addison- Wesley Professional, 2017.
4. https://angular-2-training-book.rangle.io.
5. https://www.atlassian.com/git
6. https://www.typescriptlang.org/docs/handbook/basic-types.html
7. [https://firebase.google.com](https://firebase.google.com/)

**INDEX**

|  |  |  |
| --- | --- | --- |
| **S. No** | **Name of Experiment** | **Page No** |
| **1** | **TASK 1: Managing the versions of the product using Version Control System (GIT)**  a. Create a git repository local add and commit a simple web application consisting of 5 pages.  b. Create a remote repository in github.com. Push the above local repository to the github.com. Explore the push, pull and fetch options with remote repository.  c. Clone a remote repository into local directory, modify the implementation and push the updated changes back to remote repository.  d. Create branch and manage the work distributions. Merge all the branches and commit the changes.  e. Publish the application using Github pages. | **1-3** |
| **2** | **TASK 2: Implementing the advanced features of JavaScript**  a. Working with Prototypal Inheritance and Classes.  b. Working with Object and Array Destructuring.  c. Working with Modules.  d. Working with Function Generators and Symbols.  e. Working with Closure | **4-6** |
| **3** | **TASK 3: Asynchronous Execution of JavaScript**  a. Working with higher order function in JavaScript.  b. Using Callback and creating a Callback Hell situation to understand the drawbacks.  c. Working with XHR: response.  d. Dealing with the Callback Hell situation using Promise. Exploring the different  ways of creating and using promise in executing the asynchronous task.  e. Dealing with Promise chaining and async / await. | **7-9** |
| **4** | **TASK 4: Fetching server state using JavaScript (fetch)**  a. Use fetch function to access remote data using the given api and display the data in the form of a table.  b. Use fetch function to read the weather details from openweathermap.org and display the details like city, min-temp, max-temp, humidity on the webpage for a given city.  c. From the same website read the weather forecast details for a given city and display the details like date – temperature in a table.  d. Plot a bar chart for the above implementation using date and temperature along X and Y axis respectively. Use ChartJS library. | **10-14** |
| **5** | **TASK 5: Node JS**  a. Create custom / local modules and export them using various module patterns.  b. Explore the functionality of os, path, util and events modules.  c. Use the fs module for creating directories and files of different formats.  d. Write script to read and write the streaming data using readable and writable streams. | **15-18** |
| **6** | **TASK 6: Working with http**  a. Create a http server listening request at port 3000. Process the request to provide different type of resources as response. (HTML, TEXT, JSON, etc.).  b. Create express server listening request at port 3000. Add different endpoints to provide access to the resources. | **19-22** |
| **7** | **TASK 7: Working with Express**  a. Create a custom API for Users data and add different endpoints in express server to perform CRUD operations on the API. Test the endpoints using POSTMAN.  b. Use EJS view-engine to display the dynamic response. Display the data read from REST API in the form of a table in EJS. | **23-26** |
| **8** | **TASK 8: Working with model (MongoDB & Firebase)**  a. Create a database for Users in Mongo DB. Create User collection and documents to the User collection. Perform all DB operations (CREATE, READ, UPDATE and DELETE) on the User collection.  b. Create a real time database in firebase for the student management system and explorethe features of Firebase Real Time Database. Perform CRUD operations on the Real Time Database. | **27-29** |
| **9** | **TASK 9: Working with Express & Mongo DB**  Create express server that has endpoints connecting to Users collection present in Mongo DB database using mongoose library and perform CRUD operation on that. | **30-32** |
| **10** | **TASK 10: Authentication and Authorization**  a. Create express server that has authorized endpoint using JWT (JSON Web Token) library.  b. Create express server that connects to Mongo DB database to authenticate the user and generate the authorized token to access the protected endpoints. | **33-40** |
| **11** | **TASK 11: MEAN Stack Development**  a. Create a user profile management system where users can update their profiles, including details like name, email, phone. Angular's data binding ensures that changes made by users are instantly reflected in the UI.  b. Develop an angular application that interacts with the backend API and executes CRUD operations on it. | **41-49** |
| **12** | **TASK 12: Angular Routing**  a. Develop angular application consisting of App, Home, About, Contact, Profile, Login and Register Components. Add a Navigation bar and navigate to the respective component using angular routing.  b. Develop a Single Page Application in Angular for User Management System that interacts with the backend database created in Task 8. Use Services and HttpClient to access the express endpoints of Task 9. | **50-55** |
| **13** | **TASK 13: MERN Stack Development**  a. Create react functional and class components. Implement the lifecycle methods of react component.  b. Develop react application with App, Home, About and Contact components. Implement the use of react props and state in these components | **56-60** |
| **14** | **TASK 14: React Routing**  a. Develop a react application that demonstrates the routing feature to navigate across different components of react and pass the data in between the components.  b. Develop a SPA in react for User Management System that interacts with the backend API using axios and perform CRUD operations on that. | **61-67** |
| **15** | **TASK 15: Single Page Application in Angular / React**  A TODO application serves as a simple yet powerful tool to help individuals and teams organize their tasks, manage priorities, and enhance productivity. TODO applications provide a structured and efficient way for individuals and teams to manage tasks, prioritize work, and achieve their goals. Develop a Single Page TODO Application in Angular / React to manage the daily tasks with the following features:  a. Task Creation: Allow users to create new tasks with a title, description, due date,  and priority level. Provide a straightforward interface for entering task details.  b. Task Listing: Display a list of all tasks with essential details. Tasks can be organized  based on different criteria such as due date, priority, or completion status.  c. Task Editing and Updating: Enable users to edit task details, including the ability to  modify the title, description, due date, and priority. Changes should be reflected in  real-time.  d. Task Deletion: Provide the option to delete tasks that are no longer relevant or  completed. Include a confirmation prompt to prevent accidental deletions.  e. Task Completion: Allow users to mark tasks as completed or mark them with a  specific status. Completed tasks may be moved to a separate section or visually  differentiated.  f. User Authentication and Authorization: Implement user accounts with  authentication to ensure data privacy. Differentiate between users and provide  appropriate authorization levels.  g. Data Persistence: Ensure that tasks are persistently stored, so users can access their  TODO lists even after closing and reopening the application. | **68-78** |

**Experiment 1: Managing the versions of the product using Version Control System (GIT)**

**Objective**

a. Create a git repository local add and commit a simple web application consisting of 5

pages.

b. Create a remote repository in github.com. Push the above local repository to the

github.com. Explore the push, pull and fetch options with remote repository.

c. Clone a remote repository into local directory, modify the implementation and push

the updated changes back to remote repository.

d. Create branch and manage the work distributions. Merge all the branches and commit

the changes.

e. Publish the application using Github pages.

**a. Create a Git Repository Locally**

# Step 1: Create a directory and navigate into it

mkdir SimpleWebApp

cd SimpleWebApp

# Step 2: Initialize a Git repository

git init

# Step 3: Add simple HTML files

echo "<!DOCTYPE html><html><head><title>Home</title></head><body><h1>Home Page</h1></body></html>" > index.html

echo "<!DOCTYPE html><html><head><title>About</title></head><body><h1>About Page</h1></body></html>" > about.html

echo "<!DOCTYPE html><html><head><title>Services</title></head><body><h1>Services Page</h1></body></html>" > services.html

echo "<!DOCTYPE html><html><head><title>Contact</title></head><body><h1>Contact Page</h1></body></html>" > contact.html

echo "<!DOCTYPE html><html><head><title>Gallery</title></head><body><h1>Gallery Page</h1></body></html>" > gallery.html

# Step 4: Stage and commit the files

git add .

git commit -m "Initial commit with 5 pages"

**Output**

Initialized empty Git repository in /SimpleWebApp/.git/

[master (root-commit) b123456] Initial commit with 5 pages

5 files changed, 25 insertions(+)

create mode 100644 about.html

create mode 100644 contact.html

create mode 100644 gallery.html

create mode 100644 index.html

create mode 100644 services.html

**b. Create a Remote Repository and Push Local Repository**

# Step 1: Add the remote repository

git remote add origin https://github.com/<your-username>/SimpleWebApp.git

# Step 2: Push the local repository to GitHub

git branch -M main

git push -u origin main

**Exploring Push, Pull, Fetch**

# Push changes

git push

# Fetch latest updates

git fetch

# Pull changes and merge

git pull

**c. Clone a Remote Repository and Push Updates**

# Clone the repository

git clone https://github.com/<your-username>/SimpleWebApp.git

cd SimpleWebApp

# Modify index.html

echo "<p>Updated content for the Home Page</p>" >> index.html

# Stage, commit, and push changes

git add index.html

git commit -m "Updated Home Page content"

git push

**d. Branching and Merging**

# Create branches

git checkout -b feature/about-page

echo "<p>Improved content on About Page</p>" >> about.html

git add about.html

git commit -m "Improved About Page"

git checkout main

git checkout -b feature/contact-page

echo "<p>Added contact form</p>" >> contact.html

git add contact.html

git commit -m "Added Contact Form"

# Merge branches into main

git checkout main

git merge feature/about-page

git merge feature/contact-page

# Push changes

git push

**e. Publish Using GitHub Pages**

# No specific Git commands. Go to GitHub repository > Settings > Pages > Source > Select 'main' branch.

**Experiment 2: Implementing the advanced features of JavaScript**

**Objectives**

a. Working with Prototypal Inheritance and Classes.

b. Working with Object and Array Destructuring.

c. Working with Modules.

d. Working with Function Generators and Symbols.

e. Working with Closure

**a. Working with Prototypal Inheritance and Classes**

// Parent Object

const Person = {

greet() {

return `Hello, my name is ${this.name}`;

},

};

// Child Object inheriting from Parent

const john = Object.create(Person);

john.name = "John";

console.log(john.greet());

**Classes**

class Animal {

constructor(name) {

this.name = name;

}

makeSound() {

return `${this.name} makes a sound.`;

}

}

class Dog extends Animal {

makeSound() {

return `${this.name} barks.`;

}

}

const dog = new Dog("Buddy");

console.log(dog.makeSound());

**Output:**

Hello, my name is John

Buddy barks.

**b. Working with Object and Array Destructuring**

**Object Destructuring**

const user = { name: "Alice", age: 30, city: "New York" };

const { name, age } = user;

console.log(`Name: ${name}, Age: ${age}`);

**Array Destructuring**

const numbers = [1, 2, 3, 4];

const [first, second, ...rest] = numbers;

console.log(`First: ${first}, Second: ${second}, Rest: ${rest}`);

Output:

Name: Alice, Age: 30

First: 1, Second: 2, Rest: 3,4

**c. Working with Modules**

**Module 1: math.js**

export function add(a, b) {

return a + b;

}

export function multiply(a, b) {

return a \* b;

}

**Module 2: app.js**

import { add, multiply } from './math.js';

console.log(`Addition: ${add(5, 3)}`);

console.log(`Multiplication: ${multiply(5, 3)}`);

**Output:**

Addition: 8

Multiplication: 15

**d. Working with Function Generators and Symbols**

**Function Generators**

function\* numberGenerator() {

yield 1;

yield 2;

yield 3;

}

const gen = numberGenerator();

console.log(gen.next().value);

console.log(gen.next().value);

console.log(gen.next().value);

**Symbols**

const sym = Symbol("unique");

const obj = {

[sym]: "This is a unique property",

};

console.log(obj[sym]);

**Output:**

1

2

3

This is a unique property

**e. Working with Closures**

function counter() {

let count = 0;

return function () {

count++;

return count;

};

}

const increment = counter();

console.log(increment());

console.log(increment());

console.log(increment());

**Output:**

1

2

3

**Experiment 3: Asynchronous Execution of JavaScript**

**Objective**

a. Working with higher order function in JavaScript.

b. Using Callback and creating a Callback Hell situation to understand the drawbacks.

c. Working with XHR: response.

d. Dealing with the Callback Hell situation using Promise. Exploring the different

ways of creating and using promise in executing the asynchronous task.

e. Dealing with Promise chaining and async / await.

**Program**

1. **Working with Higher-Order Functions in JavaScript**

function applyOperation(arr, operation) {

return arr.map(operation);

}

const numbers = [1, 2, 3, 4];

const square = (num) => num \* num;

console.log(applyOperation(numbers, square));

**Output:**

[1, 4, 9, 16]

1. **Using Callback and Creating a Callback Hell**

function fetchData(step, callback) {

setTimeout(() => {

console.log(`Step ${step} complete`);

callback();

}, 1000);

}

// Creating Callback Hell

fetchData(1, () => {

fetchData(2, () => {

fetchData(3, () => {

fetchData(4, () => {

console.log("All steps complete!");

});

});

});

});

**Output:**

Step 1 complete

Step 2 complete

Step 3 complete

Step 4 complete

All steps complete!

1. **Working with XHR: Response**

const xhr = new XMLHttpRequest();

xhr.open("GET", "https://jsonplaceholder.typicode.com/posts/1");

xhr.onload = function () {

if (xhr.status === 200) {

console.log("Response:", JSON.parse(xhr.responseText));

} else {

console.error("Error:", xhr.status);

}

};

xhr.send();

**Output:**

Response: { id: 1, title: "Some Title", body: "Some Content" }

1. **Dealing with Callback Hell Using Promises**

function fetchData(step) {

return new Promise((resolve) => {

setTimeout(() => {

console.log(`Step ${step} complete`);

resolve();

}, 1000);

});

}

// Resolving Callback Hell with Promises

fetchData(1)

.then(() => fetchData(2))

.then(() => fetchData(3))

.then(() => fetchData(4))

.then(() => console.log("All steps complete!"));

**Output:**

Step 1 complete

Step 2 complete

Step 3 complete

Step 4 complete

All steps complete!

1. **Dealing with Promise Chaining and Async/Await**

**Promise Chaining**

function fetchData(step) {

return new Promise((resolve) => {

setTimeout(() => {

console.log(`Step ${step} complete`);

resolve();

}, 1000);

});

}

fetchData(1)

.then(() => fetchData(2))

.then(() => fetchData(3))

.then(() => fetchData(4))

.then(() => console.log("All steps complete!"));

**Using Async/Await**

async function executeSteps() {

await fetchData(1);

await fetchData(2);

await fetchData(3);

await fetchData(4);

console.log("All steps complete!");

}

executeSteps();

**Output for Both:**

Step 1 complete

Step 2 complete

Step 3 complete

Step 4 complete

All steps complete!

**Experiment 4: Fetching server state using JavaScript (fetch)**

**Objective**

a. Use fetch function to access remote data using the given api and display the data in the

form of a table.

b. Use fetch function to read the weather details from openweathermap.org and display

the details like city, min-temp, max-temp, humidity on the webpage for a given city.

c. From the same website read the weather forecast details for a given city and display

the details like date – temperature in a table.

d. Plot a bar chart for the above implementation using date and temperature along X and

Y axis respectively. Use ChartJS library.

**Program**

1. **Use fetch to Access Remote Data and Display as a Table**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Fetch Data Table</title>

</head>

<body>

<table border="1" id="data-table">

<thead>

<tr>

<th>ID</th>

<th>Title</th>

<th>Body</th>

</tr>

</thead>

<tbody></tbody>

</table>

<script>

fetch('https://jsonplaceholder.typicode.com/posts')

.then(response => response.json())

.then(data => {

const tableBody = document.querySelector("#data-table tbody");

data.slice(0, 10).forEach(post => {

const row = `

<tr>

<td>${post.id}</td>

<td>${post.title}</td>

<td>${post.body}</td>

</tr>

`;

tableBody.insertAdjacentHTML('beforeend', row);

});

})

.catch(error => console.error("Error fetching data:", error));

</script>

</body>

</html>

1. **Fetch Weather Details from OpenWeatherMap**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Weather Details</title>

</head>

<body>

<h1>Weather Details</h1>

<input type="text" id="city" placeholder="Enter City Name">

<button onclick="getWeather()">Get Weather</button>

<div id="weather-details"></div>

<script>

function getWeather() {

const city = document.getElementById('city').value;

fetch(`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=YOUR\_API\_KEY&units=metric`)

.then(response => response.json())

.then(data => {

const details = `

<p>City: ${data.name}</p>

<p>Min Temp: ${data.main.temp\_min}°C</p>

<p>Max Temp: ${data.main.temp\_max}°C</p>

<p>Humidity: ${data.main.humidity}%</p>

`;

document.getElementById('weather-details').innerHTML = details;

})

.catch(error => console.error("Error fetching weather:", error));

}

</script>

</body>

</html>

1. **Fetch Weather Forecast and Display in a Table**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Weather Forecast</title>

</head>

<body>

<h1>Weather Forecast</h1>

<input type="text" id="forecast-city" placeholder="Enter City Name">

<button onclick="getForecast()">Get Forecast</button>

<table border="1" id="forecast-table">

<thead>

<tr>

<th>Date</th>

<th>Temperature (°C)</th>

</tr>

</thead>

<tbody></tbody>

</table>

<script>

function getForecast() {

const city = document.getElementById('forecast-city').value;

fetch(`https://api.openweathermap.org/data/2.5/forecast?q=${city}&appid=YOUR\_API\_KEY&units=metric`)

.then(response => response.json())

.then(data => {

const tableBody = document.querySelector("#forecast-table tbody");

tableBody.innerHTML = "";

data.list.forEach(item => {

const row = `

<tr>

<td>${new Date(item.dt\_txt).toLocaleDateString()}</td>

<td>${item.main.temp}</td>

</tr>

`;

tableBody.insertAdjacentHTML('beforeend', row);

});

})

.catch(error => console.error("Error fetching forecast:", error));

}

</script>

</body>

</html>

1. **Plot a Bar Chart Using Chart.js**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Bar Chart - Weather Forecast</title>

<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

</head>

<body>

<h1>Weather Forecast Bar Chart</h1>

<input type="text" id="chart-city" placeholder="Enter City Name">

<button onclick="plotChart()">Plot Chart</button>

<canvas id="weather-chart" width="400" height="200"></canvas>

<script>

function plotChart() {

const city = document.getElementById('chart-city').value;

fetch(`https://api.openweathermap.org/data/2.5/forecast?q=${city}&appid=YOUR\_API\_KEY&units=metric`)

.then(response => response.json())

.then(data => {

const dates = data.list.map(item => new Date(item.dt\_txt).toLocaleDateString());

const temperatures = data.list.map(item => item.main.temp);

const ctx = document.getElementById('weather-chart').getContext('2d');

new Chart(ctx, {

type: 'bar',

data: {

labels: dates,

datasets: [{

label: 'Temperature (°C)',

data: temperatures,

backgroundColor: 'rgba(75, 192, 192, 0.2)',

borderColor: 'rgba(75, 192, 192, 1)',

borderWidth: 1

}]

},

options: {

scales: {

x: { beginAtZero: true },

y: { beginAtZero: true }

}

}

});

})

.catch(error => console.error("Error fetching forecast:", error));

}

</script>

</body>

</html>

**Experiment 5:** **Node JS**

**Objective**

a. Create custom / local modules and export them using various module patterns.

b. Explore the functionality of os, path, util and events modules.

c. Use the fs module for creating directories and files of different formats.

d. Write script to read and write the streaming data using readable and writable streams

**a. Create Custom/Local Modules and Export Using Various Module Patterns**

**math.js (Custom Module):**

// CommonJS pattern

function add(a, b) {

return a + b;

}

function multiply(a, b) {

return a \* b;

}

module.exports = { add, multiply };

**app.js (Main File):**

const math = require('./math');

console.log("Addition:", math.add(5, 3));

console.log("Multiplication:", math.multiply(5, 3));

**ES6 Module Pattern**

**math.js (Custom Module):**

export function add(a, b) {

return a + b;

}

export function multiply(a, b) {

return a \* b;

}

**app.js (Main File):**

import { add, multiply } from './math.js';

console.log("Addition:", add(5, 3));

console.log("Multiplication:", multiply(5, 3));

**Output for Both Patterns:**

Addition: 8

Multiplication: 15

**b. Explore the Functionality of os, path, util, and events Modules**

**Using os Module**

const os = require('os');

console.log("OS Platform:", os.platform());

console.log("Free Memory:", os.freemem());

console.log("Total Memory:", os.totalmem());

**Using path Module**

const path = require('path');

console.log("File Extension:", path.extname(\_\_filename));

console.log("Directory Name:", path.dirname(\_\_filename));

**Using util Module**

const util = require('util');

const asyncFunction = util.promisify(setTimeout);

asyncFunction(1000).then(() => console.log("Completed after 1 second"));

**Using events Module**

const EventEmitter = require('events');

const emitter = new EventEmitter();

emitter.on('greet', () => {

console.log("Hello, World!");

});

emitter.emit('greet');

**Output:**

OS Platform: linux

Free Memory: 1548570624

Total Memory: 8589934592

File Extension: .js

Directory Name: /path/to/directory

Completed after 1 second

Hello, World!

**c. Use the fs Module for Creating Directories and Files**

const fs = require('fs');

// Create a directory

if (!fs.existsSync('exampleDir')) {

fs.mkdirSync('exampleDir');

}

// Create a text file

fs.writeFileSync('exampleDir/example.txt', 'Hello, this is a text file.');

// Create a JSON file

const jsonData = { name: "John", age: 30 };

fs.writeFileSync('exampleDir/example.json', JSON.stringify(jsonData, null, 2));

**Output Files:**

* example.txt containing: Hello, this is a text file.
* example.json containing: { "name": "John", "age": 30 }
  1. **Script to Read and Write Streaming Data**

**data.txt (Sample Data):**

This is a sample data file for streaming operations.

**stream.js:**

const fs = require('fs');

// Create a readable stream

const readableStream = fs.createReadStream('data.txt', { encoding: 'utf8' });

// Create a writable stream

const writableStream = fs.createWriteStream('output.txt');

// Pipe data from readable to writable

readableStream.pipe(writableStream);

readableStream.on('end', () => {

console.log("Streaming completed. Check output.txt for the result.");

});

**Output in output.txt:**

This is a sample data file for streaming operations.

**Experiment 6: Working with http**

**Objective**

a. Create a http server listening request at port 3000. Process the request to provide different type of resources as response. (HTML, TEXT, JSON, etc.).

b. Create express server listening request at port 3000. Add different endpoints to provide access to the resources.

**a. Create an HTTP Server**

const http = require('http');

const server = http.createServer((req, res) => {

if (req.url === '/' && req.method === 'GET') {

// Respond with HTML

res.writeHead(200, { 'Content-Type': 'text/html' });

res.end('<h1>Welcome to the HTTP Server!</h1>');

} else if (req.url === '/text' && req.method === 'GET') {

// Respond with Plain Text

res.writeHead(200, { 'Content-Type': 'text/plain' });

res.end('This is a plain text response.');

} else if (req.url === '/json' && req.method === 'GET') {

// Respond with JSON

res.writeHead(200, { 'Content-Type': 'application/json' });

res.end(JSON.stringify({ message: 'This is a JSON response.', status: 200 }));

} else {

// Handle 404

res.writeHead(404, { 'Content-Type': 'text/plain' });

res.end('404 Not Found');

}

});

server.listen(3000, () => {

console.log('HTTP server is listening on port 3000');

});

**Run the Code:**

1. Save it as server.js.
2. Start the server using node server.js.
3. Access the endpoints:
   * http://localhost:3000/ (HTML response)
   * http://localhost:3000/text (Text response)
   * http://localhost:3000/json (JSON response)

**b. Create an Express Server**

const express = require('express');

const app = express();

const port = 3000;

// Middleware for parsing JSON

app.use(express.json());

// Endpoint for HTML response

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

res.send('<h1>Welcome to the Express Server!</h1>');

});

// Endpoint for Text response

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

res.type('text').send('This is a plain text response.');

});

// Endpoint for JSON response

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

res.json({ message: 'This is a JSON response.', status: 200 });

});

// Endpoint for POST request

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

const data = req.body;

res.json({ message: 'Data received', receivedData: data });

});

// 404 Error handler

app.use((req, res) => {

res.status(404).send('404 Not Found');

});

// Start the server

app.listen(port, () => {

console.log(`Express server is running on http://localhost:${port}`);

});

**Run the Code:**

1. Save it as app.js.
2. Install Express using npm install express.
3. Start the server using node app.js.
4. Access the endpoints:
   * http://localhost:3000/ (HTML response)
   * http://localhost:3000/text (Text response)
   * http://localhost:3000/json (JSON response)
   * Test POST endpoint using tools like Postman:
     + URL: http://localhost:3000/post
     + Method: POST
     + Body (JSON): { "name": "John", "age": 30 }

**Experiment 7: Working with Express**

**Objective**

a. Create a custom API for Users data and add different endpoints in express server to perform CRUD operations on the API. Test the endpoints using POSTMAN.

b. Use EJS view-engine to display the dynamic response. Display the data read from REST API in the form of a table in EJS.

**Program:**

1. **Create a Custom API for Users Data with CRUD Operations**

const express = require('express');

const bodyParser = require('body-parser');

const app = express();

const port = 3000;

// Middleware

app.use(bodyParser.json());

// Sample Users Data

let users = [

{ id: 1, name: 'John Doe', age: 25 },

{ id: 2, name: 'Jane Smith', age: 30 }

];

// CREATE: Add a new user

app.post('/api/users', (req, res) => {

const newUser = { id: users.length + 1, ...req.body };

users.push(newUser);

res.status(201).json({ message: 'User created', user: newUser });

});

// READ: Get all users

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

res.json(users);

});

// READ: Get a specific user by ID

app.get('/api/users/:id', (req, res) => {

const user = users.find(u => u.id === parseInt(req.params.id));

if (!user) {

return res.status(404).json({ message: 'User not found' });

}

res.json(user);

});

// UPDATE: Update a user's data

app.put('/api/users/:id', (req, res) => {

const user = users.find(u => u.id === parseInt(req.params.id));

if (!user) {

return res.status(404).json({ message: 'User not found' });

}

Object.assign(user, req.body);

res.json({ message: 'User updated', user });

});

// DELETE: Remove a user

app.delete('/api/users/:id', (req, res) => {

const userIndex = users.findIndex(u => u.id === parseInt(req.params.id));

if (userIndex === -1) {

return res.status(404).json({ message: 'User not found' });

}

users.splice(userIndex, 1);

res.json({ message: 'User deleted' });

});

// Start server

app.listen(port, () => {

console.log(`API server is running on http://localhost:${port}`);

});

**Steps to Test:**

1. Start the server: node app.js.
2. Use **Postman** to test the endpoints:
   * POST /api/users to create a new user (body: { "name": "Alice", "age": 28 }).
   * GET /api/users to fetch all users.
   * GET /api/users/:id to fetch a specific user.
   * PUT /api/users/:id to update user data.
   * DELETE /api/users/:id to remove a user.
3. **Use EJS to Display Data in a Table**
4. **Install EJS:**

npm install ejs\

1. **Update Express Configuration: Modify your server file to include EJS as the view engine.**

const express = require('express');

const bodyParser = require('body-parser');

const app = express();

const port = 3000;

// Middleware

app.use(bodyParser.json());

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

// Sample Users Data

let users = [

{ id: 1, name: 'John Doe', age: 25 },

{ id: 2, name: 'Jane Smith', age: 30 }

];

// Endpoint to Fetch Users and Render Table

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

res.render('users', { users });

});

// Start server

app.listen(port, () => {

console.log(`EJS server is running on http://localhost:${port}`);

});

**EJS Template (views/users.ejs):**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Users Data</title>

<style>

table {

width: 50%;

margin: auto;

border-collapse: collapse;

}

th, td {

border: 1px solid #ddd;

padding: 8px;

text-align: left;

}

th {

background-color: #f4f4f4;

}

</style>

</head>

<body>

<h1 style="text-align: center;">Users Data</h1>

<table>

<thead>

<tr>

<th>ID</th>

<th>Name</th>

<th>Age</th>

</tr>

</thead>

<tbody>

<% users.forEach(user => { %>

<tr>

<td><%= user.id %></td>

<td><%= user.name %></td>

<td><%= user.age %></td>

</tr>

<% }); %>

</tbody>

</table>

</body>

</html>

**Steps to Test:**

1. Start the server: node app.js.
2. Access the EJS page in your browser at http://localhost:3000/users.

**Experiment 8: Working with model (MongoDB & Firebase)**

**Objective**

a. Create a database for Users in Mongo DB. Create User collection and documents to the User collection. Perform all DB operations (CREATE, READ, UPDATE and DELETE) on the User collection.

b. Create a real time database in firebase for the student management system and explore the features of Firebase Real Time Database. Perform CRUD operations on the Real Time Database.

**a. Working with MongoDB for User Data**

**Step 1: Install and Set Up MongoDB**

Install MongoDB on your system or use a cloud service like MongoDB Atlas.

Install the **MongoDB Node.js Driver** in your project:

npm install mongodb

**Step 2: Create MongoDB Database and Perform CRUD Operations**

const { MongoClient } = require('mongodb');

// MongoDB Connection URI

const uri = 'mongodb://127.0.0.1:27017';

const client = new MongoClient(uri);

// Database and Collection Names

const dbName = 'userDB';

const collectionName = 'users';

async function main() {

try {

await client.connect();

console.log('Connected to MongoDB');

const db = client.db(dbName);

const collection = db.collection(collectionName);

// CREATE: Insert Documents

const newUser = { name: 'Alice', age: 25, email: 'alice@example.com' };

const result = await collection.insertOne(newUser);

console.log('User Inserted:', result.insertedId);

// READ: Find Documents

const users = await collection.find().toArray();

console.log('Users:', users);

// UPDATE: Update a Document

const updateResult = await collection.updateOne(

{ name: 'Alice' },

{ $set: { age: 26 } }

);

console.log('User Updated:', updateResult.modifiedCount);

// DELETE: Delete a Document

const deleteResult = await collection.deleteOne({ name: 'Alice' });

console.log('User Deleted:', deleteResult.deletedCount);

} catch (error) {

console.error('Error:', error);

} finally {

await client.close();

console.log('MongoDB connection closed');

}

}

main();

**Run the Code:**

1. Start your MongoDB server.
2. Save the code as mongoCRUD.js.
3. Run the script

node mongoCRUD.js

* 1. **Firebase Real-Time Database for Student Management**

**Step 1: Set Up Firebase**

1. Create a Firebase project on Firebase Console.
2. Add a Realtime Database to your project.
3. Install the Firebase Node.js SDK:

npm install firebase-admin

**Firebase Realtime Database CRUD**

**Setup Firebase (firebaseCRUD.js):**

const admin = require('firebase-admin');

const serviceAccount = require('./serviceAccountKey.json');

// Initialize Firebase Admin SDK

admin.initializeApp({

credential: admin.credential.cert(serviceAccount),

databaseURL: 'https://<your-database-name>.firebaseio.com', // Replace with your database URL

});

const db = admin.database();

const ref = db.ref('students');

// CREATE: Add a Student

ref.push({

name: 'John Doe',

age: 20,

course: 'Computer Science',

}, (error) => {

if (error) {

console.error('Error adding student:', error);

} else {

console.log('Student added successfully');

}

});

// READ: Retrieve All Students

ref.once('value', (snapshot) => {

console.log('Students:', snapshot.val());

});

// UPDATE: Update a Student

ref.child('<student-id>').update({

age: 21,

}, (error) => {

if (error) {

console.error('Error updating student:', error);

} else {

console.log('Student updated successfully');

}

});

// DELETE: Remove a Student

ref.child('<student-id>').remove((error) => {

if (error) {

console.error('Error deleting student:', error);

} else {

console.log('Student removed successfully');

}

});

**Experiment 9: Working with Express & Mongo DB**

**Objective**

**Create express server that has endpoints connecting to Users collection present in Mongo DB database using mongoose library and perform CRUD operation on that.**

**Program:**

**Step 1: Install Required Packages**

npm install express mongoose body-parser

**Step 2: Set Up Mongoose and Express**

const express = require('express');

const bodyParser = require('body-parser');

const mongoose = require('mongoose');

const app = express();

const port = 3000;

// Middleware

app.use(bodyParser.json());

// Connect to MongoDB

mongoose

.connect('mongodb://127.0.0.1:27017/userDB', {

useNewUrlParser: true,

useUnifiedTopology: true,

})

.then(() => console.log('Connected to MongoDB'))

.catch((err) => console.error('MongoDB connection error:', err));

// Define User Schema

const userSchema = new mongoose.Schema({

name: { type: String, required: true },

age: { type: Number, required: true },

email: { type: String, required: true, unique: true },

});

// Create User Model

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

// CREATE: Add a New User

app.post('/api/users', async (req, res) => {

try {

const newUser = new User(req.body);

const savedUser = await newUser.save();

res.status(201).json({ message: 'User created', user: savedUser });

} catch (err) {

res.status(400).json({ error: err.message });

}

});

// READ: Get All Users

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

try {

const users = await User.find();

res.json(users);

} catch (err) {

res.status(500).json({ error: err.message });

}

});

// READ: Get a Specific User by ID

app.get('/api/users/:id', async (req, res) => {

try {

const user = await User.findById(req.params.id);

if (!user) return res.status(404).json({ message: 'User not found' });

res.json(user);

} catch (err) {

res.status(500).json({ error: err.message });

}

});

// UPDATE: Update a User by ID

app.put('/api/users/:id', async (req, res) => {

try {

const updatedUser = await User.findByIdAndUpdate(req.params.id, req.body, {

new: true,

runValidators: true,

});

if (!updatedUser) return res.status(404).json({ message: 'User not found' });

res.json({ message: 'User updated', user: updatedUser });

} catch (err) {

res.status(400).json({ error: err.message });

}

});

// DELETE: Remove a User by ID

app.delete('/api/users/:id', async (req, res) => {

try {

const deletedUser = await User.findByIdAndDelete(req.params.id);

if (!deletedUser) return res.status(404).json({ message: 'User not found' });

res.json({ message: 'User deleted' });

} catch (err) {

res.status(500).json({ error: err.message });

}

});

// Start the Server

app.listen(port, () => {

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

});

**Step 3: Start the Server**

**Save the file as server.js.**

**Run the server**

node server.js

**Endpoints to Test (Using Postman or Curl)**

| **HTTP Method** | **Endpoint** | **Description** |
| --- | --- | --- |
| POST | /api/users | Create a new user (send JSON body). |
| GET | /api/users | Get all users. |
| GET | /api/users/:id | Get a specific user by ID. |
| PUT | /api/users/:id | Update a user by ID (send JSON body). |
| DELETE | /api/users/:id | Delete a user by ID. |

**Example JSON Payload for Testing**

**POST /api/users:**

{

"name": "John Doe",

"age": 30,

"email": "john.doe@example.com"

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Experiment 10: Authentication and Authorization**

**Objective**

**a. Create express server that has authorized endpoint using JWT (JSON Web Token) library.**

**b. Create express server that connects to Mongo DB database to authenticate the user and generate the authorized token to access the protected endpoints**

**Program:**

* 1. **Express Server with Authorized Endpoint using JWT**

**Step 1: Install Required Packages**

npm install express jsonwebtoken body-parser bcrypt

**Express Server with JWT Authentication**

const express = require('express');

const bodyParser = require('body-parser');

const jwt = require('jsonwebtoken');

const bcrypt = require('bcrypt');

const app = express();

const port = 3000;

// Secret key for JWT

const JWT\_SECRET = 'your\_jwt\_secret\_key';

// Middleware

app.use(bodyParser.json());

// Mock user database

const users = [

{ id: 1, username: 'john', password: bcrypt.hashSync('password123', 10) }

];

// Login Endpoint: Authenticate User and Generate Token

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

const { username, password } = req.body;

const user = users.find(u => u.username === username);

if (!user || !bcrypt.compareSync(password, user.password)) {

return res.status(401).json({ message: 'Invalid credentials' });

}

const token = jwt.sign({ id: user.id, username: user.username }, JWT\_SECRET, { expiresIn: '1h' });

res.json({ token });

});

// Middleware to Verify JWT

const authenticateJWT = (req, res, next) => {

const token = req.headers.authorization?.split(' ')[1];

if (!token) {

return res.status(401).json({ message: 'Access token missing' });

}

try {

const decoded = jwt.verify(token, JWT\_SECRET);

req.user = decoded;

next();

} catch (err) {

res.status(403).json({ message: 'Invalid or expired token' });

}

};

// Protected Endpoint

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

res.json({ message: `Hello ${req.user.username}, you are authorized!` });

});

// Start Server

app.listen(port, () => {

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

});

Testing the Endpoints

**Login:**

* Endpoint: POST /login
* Request Body

{

"username": "john",

"password": "password123"

}

Response:

{

"token": "your\_jwt\_token"

}

**Access Protected Endpoint:**

* Endpoint: GET /protected
* Add Authorization Header: Bearer your\_jwt\_token

Response:

{

"message": "Hello john, you are authorized!"

}

* 1. **Express Server with MongoDB for Authentication and JWT**

Step 1: Install Required Packages

npm install mongoose

Express Server with MongoDB and JWT

const express = require('express');

const bodyParser = require('body-parser');

const jwt = require('jsonwebtoken');

const bcrypt = require('bcrypt');

const mongoose = require('mongoose');

const app = express();

const port = 3000;

// Secret key for JWT

const JWT\_SECRET = 'your\_jwt\_secret\_key';

// Middleware

app.use(bodyParser.json());

// MongoDB Connection

mongoose

.connect('mongodb://127.0.0.1:27017/authDB', { useNewUrlParser: true, useUnifiedTopology: true })

.then(() => console.log('Connected to MongoDB'))

.catch(err => console.error('MongoDB connection error:', err));

// User Schema and Model

const userSchema = new mongoose.Schema({

username: { type: String, required: true, unique: true },

password: { type: String, required: true },

});

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

// Register Endpoint: Create New User

app.post('/register', async (req, res) => {

try {

const { username, password } = req.body;

const hashedPassword = await bcrypt.hash(password, 10);

const user = new User({ username, password: hashedPassword });

await user.save();

res.status(201).json({ message: 'User registered successfully' });

} catch (err) {

res.status(400).json({ error: err.message });

}

});

// Login Endpoint: Authenticate User and Generate Token

app.post('/login', async (req, res) => {

const { username, password } = req.body;

const user = await User.findOne({ username });

if (!user || !(await bcrypt.compare(password, user.password))) {

return res.status(401).json({ message: 'Invalid credentials' });

}

const token = jwt.sign({ id: user.\_id, username: user.username }, JWT\_SECRET, { expiresIn: '1h' });

res.json({ token });

});

// Middleware to Verify JWT

const authenticateJWT = (req, res, next) => {

const token = req.headers.authorization?.split(' ')[1];

if (!token) {

return res.status(401).json({ message: 'Access token missing' });

}

try {

const decoded = jwt.verify(token, JWT\_SECRET);

req.user = decoded;

next();

} catch (err) {

res.status(403).json({ message: 'Invalid or expired token' });

}

};

// Protected Endpoint

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

res.json({ message: `Hello ${req.user.username}, you are authorized!` });

});

// Start Server

app.listen(port, () => {

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

});

**Testing the MongoDB Integration**

**Register a User:**

* Endpoint: POST /register
* Request Body:

**{**

"username": "john",

"password": "password123"

}

**Login to Generate JWT Token:**

* Endpoint: POST /login
* Request Body:

{

"username": "john",

"password": "password123"

}

Response:

{

"token": "your\_jwt\_token"

}

Access Protected Endpoint:

Endpoint: GET /protected

Add Authorization Header: Bearer your\_jwt\_token

Response:

{

"message": "Hello john, you are authorized!"

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Experiment 11: MEAN Stack Development**

**Objective**

**a.** **Create a user profile management system where users can update their profiles, including details like name, email, phone. Angular's data binding ensures that changes made by users are instantly reflected in the UI.**

**b. Develop an angular application that interacts with the backend API and executes CRUD operations on it.**

**Program**

**1. Backend (Node.js with Express and MongoDB) to handle the data storage and provide the API for CRUD operations.**

**2. Frontend (Angular) to interact with the backend and perform CRUD operations.**

**Step 1: Backend Development (Node.js + Express + MongoDB)**

npm init -y

npm install express mongoose body-parser cors

Create a Simple Backend API

File structure:

/server

├── server.js

├── models

└── user.js

**server.js** (Node.js Backend):

const express = require('express');

const mongoose = require('mongoose');

const bodyParser = require('body-parser');

const cors = require('cors');

const User = require('./models/user');

const app = express();

const port = 3000;

app.use(bodyParser.json());

app.use(cors()); // Allow cross-origin requests

// Connect to MongoDB

mongoose.connect('mongodb://127.0.0.1:27017/meanstack', {

useNewUrlParser: true,

useUnifiedTopology: true,

})

.then(() => console.log('Connected to MongoDB'))

.catch((err) => console.log('MongoDB connection error:', err));

// User Schema and Model

const userSchema = new mongoose.Schema({

name: String,

email: String,

phone: String,

});

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

// CRUD operations

// Create a new user profile

app.post('/api/users', async (req, res) => {

try {

const newUser = new User(req.body);

await newUser.save();

res.status(201).send(newUser);

} catch (err) {

res.status(400).send(err);

}

});

// Get all users

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

try {

const users = await User.find();

res.status(200).send(users);

} catch (err) {

res.status(400).send(err);

}

});

// Get a specific user profile

app.get('/api/users/:id', async (req, res) => {

try {

const user = await User.findById(req.params.id);

if (!user) return res.status(404).send('User not found');

res.status(200).send(user);

} catch (err) {

res.status(400).send(err);

}

});

// Update a user profile

app.put('/api/users/:id', async (req, res) => {

try {

const updatedUser = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });

if (!updatedUser) return res.status(404).send('User not found');

res.status(200).send(updatedUser);

} catch (err) {

res.status(400).send(err);

}

});

// Delete a user profile

app.delete('/api/users/:id', async (req, res) => {

try {

const deletedUser = await User.findByIdAndDelete(req.params.id);

if (!deletedUser) return res.status(404).send('User not found');

res.status(200).send('User deleted');

} catch (err) {

res.status(400).send(err);

}

});

// Start the server

app.listen(port, () => {

console.log(`Server is running at http://localhost:${port}`);

});

**models/user.js** (Mongoose Model for User Profile):

const mongoose = require('mongoose');

// User Schema

const userSchema = new mongoose.Schema({

name: String,

email: String,

phone: String,

});

module.exports = mongoose.model('User', userSchema);

**Step 2: Frontend Development (Angular)**

* **Install Angular CLI**

npm install -g @angular/cli

* Create a new Angular Application

ng new user-profile-app

cd user-profile-app

* Create a User Profile Component:

ng generate component user-profile

**user-profile.component.ts**:

import { Component, OnInit } from '@angular/core';

import { HttpClient } from '@angular/common/http';

@Component({

selector: 'app-user-profile',

templateUrl: './user-profile.component.html',

styleUrls: ['./user-profile.component.css']

})

export class UserProfileComponent implements OnInit {

users = [];

currentUser = { name: '', email: '', phone: '' };

apiUrl = 'http://localhost:3000/api/users'; // Backend API URL

constructor(private http: HttpClient) { }

ngOnInit(): void {

this.getUsers();

}

// Get all users

getUsers() {

this.http.get<any[]>(this.apiUrl).subscribe(data => {

this.users = data;

});

}

// Add or update user

saveUser() {

if (this.currentUser['\_id']) {

// Update existing user

this.http.put(`${this.apiUrl}/${this.currentUser['\_id']}`, this.currentUser)

.subscribe(() => this.getUsers());

} else {

// Create new user

this.http.post(this.apiUrl, this.currentUser)

.subscribe(() => this.getUsers());

}

this.currentUser = { name: '', email: '', phone: '' };

}

// Edit user

editUser(user: any) {

this.currentUser = { ...user };

}

// Delete user

deleteUser(id: string) {

this.http.delete(`${this.apiUrl}/${id}`).subscribe(() => this.getUsers());

}

}

**user-profile.component.html**:

<div class="container">

<h2>User Profile Management</h2>

<form (ngSubmit)="saveUser()">

<div class="form-group">

<label for="name">Name</label>

<input type="text" id="name" class="form-control" [(ngModel)]="currentUser.name" name="name" required>

</div>

<div class="form-group">

<label for="email">Email</label>

<input type="email" id="email" class="form-control" [(ngModel)]="currentUser.email" name="email" required>

</div>

<div class="form-group">

<label for="phone">Phone</label>

<input type="text" id="phone" class="form-control" [(ngModel)]="currentUser.phone" name="phone" required>

</div>

<button type="submit" class="btn btn-primary">Save</button>

</form>

<h3>Users List</h3>

<ul class="list-group">

<li class="list-group-item" \*ngFor="let user of users">

{{ user.name }} - {{ user.email }} - {{ user.phone }}

<button (click)="editUser(user)" class="btn btn-warning btn-sm ml-2">Edit</button>

<button (click)="deleteUser(user.\_id)" class="btn btn-danger btn-sm ml-2">Delete</button>

</li>

</ul>

</div>

**2.Add HTTP Client Module to Angular Module**

Open app.module.ts and import HttpClientModule.

**app.module.ts**:

import { BrowserModule } from '@angular/platform-browser';

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

import { HttpClientModule } from '@angular/common/http';

import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

import { UserProfileComponent } from './user-profile/user-profile.component';

@NgModule({

declarations: [

AppComponent,

UserProfileComponent

],

imports: [

BrowserModule,

HttpClientModule,

FormsModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**Step 3: Run the Application**

1. **Run the Backend:** Start your backend server using node server.js from the backend folder.
2. **Run the Angular Application:** Start your Angular frontend using

ng serve

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Experiment 12: Angular Routing**

**Objective**

**a. Develop angular application consisting of App, Home, About, Contact, Profile, Login and Register Components. Add a Navigation bar and navigate to the respective component using angular routing.**

**b. Develop a Single Page Application in Angular for User Management System that interacts with the backend database created in Task 8. Use Services and HttpClient to access the express endpoints of Task 9**

**Program**

**Angular Routing - Develop an Angular Application with Components**

1. **Set Up the Angular Application**

ng new angular-routing-app

cd angular-routing-app

ng generate component home

ng generate component about

ng generate component contact

ng generate component profile

ng generate component login

ng generate component register

1. **Install Angular Router**

Routing is already included in Angular by default, but you need to configure it by importing RouterModule in your AppModule.

1. **Configure Routing**

**app-routing.module.ts**: This file will handle routing for the entire application

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

import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';

import { ContactComponent } from './contact/contact.component';

import { ProfileComponent } from './profile/profile.component';

import { LoginComponent } from './login/login.component';

import { RegisterComponent } from './register/register.component';

const routes: Routes = [

{ path: '', component: HomeComponent }, // Default route

{ path: 'home', component: HomeComponent },

{ path: 'about', component: AboutComponent },

{ path: 'contact', component: ContactComponent },

{ path: 'profile', component: ProfileComponent },

{ path: 'login', component: LoginComponent },

{ path: 'register', component: RegisterComponent },

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

1. **Update app.module.ts to Import Routing**

In **app.module.ts**, import the AppRoutingModule:

import { BrowserModule } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';

import { ContactComponent } from './contact/contact.component';

import { ProfileComponent } from './profile/profile.component';

import { LoginComponent } from './login/login.component';

import { RegisterComponent } from './register/register.component';

import { AppRoutingModule } from './app-routing.module'; // Importing routing module

@NgModule({

declarations: [

AppComponent,

HomeComponent,

AboutComponent,

ContactComponent,

ProfileComponent,

LoginComponent,

RegisterComponent

],

imports: [

BrowserModule,

AppRoutingModule // Add AppRoutingModule here

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

1. **Create a Navigation Bar**

In **app.component.html**, add the following code to create a navigation bar that will allow the user to navigate between different components:

<div>

<nav>

<ul>

<li><a routerLink="/home" routerLinkActive="active">Home</a></li>

<li><a routerLink="/about" routerLinkActive="active">About</a></li>

<li><a routerLink="/contact" routerLinkActive="active">Contact</a></li>

<li><a routerLink="/profile" routerLinkActive="active">Profile</a></li>

<li><a routerLink="/login" routerLinkActive="active">Login</a></li>

<li><a routerLink="/register" routerLinkActive="active">Register</a></li>

</ul>

</nav>

<router-outlet></router-outlet>

</div>

1. **Style the Navigation Bar**

Add some basic styles in **app.component.css**:

nav ul {

display: flex;

list-style-type: none;

padding: 0;

}

nav ul li {

margin-right: 15px;

}

nav ul li a {

text-decoration: none;

}

nav ul li a.active {

font-weight: bold;

color: blue;

}

router-outlet {

margin-top: 20px;

}

1. **Create Component Templates**

[**home.component.html**](http://home.component.html/)

<h2>Welcome to Home Page</h2>

**about.component.html**

<h2>About Us</h2>

**contact.component.html:**

<h2>Contact Us</h2>

**profile.component.html**:

<h2>Your Profile</h2>

**login.component.html**:

<h2>Login</h2>

**register.component.html**:

<h2>Register</h2>

1. **Test the Application**

ng serve

**b.User Management System with Angular, Services, and HttpClient**

**1. Set Up Backend**

Ensure that your **Node.js backend (from Task 8)** is running, as it will handle the CRUD operations for user management.

Make sure the server provides endpoints like:

* **POST** /api/users for creating a user
* **GET** /api/users for fetching all users
* **GET** /api/users/:id for fetching a specific user by ID
* **PUT** /api/users/:id for updating a user
* **DELETE** /api/users/:id for deleting a user

**2.Set Up Angular Service for HTTP Requests**

ng generate service user

**user.service.ts**:

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

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class UserService {

private apiUrl = 'http://localhost:3000/api/users'; // Backend URL

constructor(private http: HttpClient) { }

// Get all users

getUsers(): Observable<any[]> {

return this.http.get<any[]>(this.apiUrl);

}

// Get a single user by ID

getUser(id: string): Observable<any> {

return this.http.get<any>(`${this.apiUrl}/${id}`);

}

// Create a new user

createUser(user: any): Observable<any> {

return this.http.post<any>(this.apiUrl, user);

}

// Update an existing user

updateUser(id: string, user: any): Observable<any> {

return this.http.put<any>(`${this.apiUrl}/${id}`, user);

}

// Delete a user

deleteUser(id: string): Observable<any> {

return this.http.delete<any>(`${this.apiUrl}/${id}`);

}

}

1. **Use the Service in Components**

user-profile.component.ts

import { Component, OnInit } from '@angular/core';

import { UserService } from './user.service';

@Component({

selector: 'app-user-profile',

templateUrl: './user-profile.component.html',

styleUrls: ['./user-profile.component.css']

})

export class UserProfileComponent implements OnInit {

users = [];

currentUser = { name: '', email: '', phone: '' };

constructor(private userService: UserService) { }

ngOnInit(): void {

this.getUsers();

}

getUsers(): void {

this.userService.getUsers().subscribe(data => {

this.users = data;

});

}

createUser(): void {

this.userService.createUser(this.currentUser).subscribe(() => {

this.getUsers();

this.resetForm();

});

}

updateUser(id: string): void {

this.userService.updateUser(id, this.currentUser).subscribe(() => {

this.getUsers();

this.resetForm();

});

}

deleteUser(id: string): void {

this.userService.deleteUser(id).subscribe(() => {

this.getUsers();

});

}

resetForm(): void {

this.currentUser = { name: '', email: '', phone: '' };

}

}

**4. Test the Application**

Run both the backend (Node.js) and the frontend (Angular). Ensure that the Angular frontend is able to interact with the backend API, perform CRUD operations, and update the UI accordingly.

* To test, add users, view the list, edit profiles, and delete users using the Angular frontend.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Experiment 13: MERN Stack Development**

**Objective**

**a. Create react functional and class components. Implement the lifecycle methods of react component.**

**b. Develop react application with App, Home, About and Contact components. Implement the use of react props and state in these components**

**Program**

**A: React Functional and Class Components with Lifecycle Methods**

1. **Create React Application**

npx create-react-app mern-react-app

cd mern-react-app

1. **Class Component Example (with Lifecycle Methods)**

ClassComponent.js

import React, { Component } from 'react';

class ClassComponent extends Component {

constructor(props) {

super(props);

this.state = { count: 0 };

console.log("Constructor called");

}

// Lifecycle method: componentDidMount

componentDidMount() {

console.log("Component did mount");

}

// Lifecycle method: componentDidUpdate

componentDidUpdate(prevProps, prevState) {

console.log("Component did update");

if (prevState.count !== this.state.count) {

console.log("Count updated");

}

}

// Lifecycle method: componentWillUnmount

componentWillUnmount() {

console.log("Component will unmount");

}

increment = () => {

this.setState({ count: this.state.count + 1 });

};

render() {

console.log("Render called");

return (

<div>

<h2>Class Component</h2>

<p>Count: {this.state.count}</p>

<button onClick={this.increment}>Increment</button>

</div>

);

}

}

export default ClassComponent;

1. **Functional Component Example (with Hooks)**

FunctionalComponent.js

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

function FunctionalComponent() {

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

// useEffect simulates componentDidMount, componentDidUpdate, componentWillUnmount

useEffect(() => {

console.log("Component mounted or updated");

// Cleanup function (acts like componentWillUnmount)

return () => {

console.log("Component will unmount");

};

}, [count]); // The effect runs when the count changes

const increment = () => {

setCount(count + 1);

};

return (

<div>

<h2>Functional Component</h2>

<p>Count: {count}</p>

<button onClick={increment}>Increment</button>

</div>

);

}

export default FunctionalComponent;

**B: Develop React Application with App, Home, About, and Contact Components**

**In this part, we will create the following components:**

* **App Component (Root component)**
* **Home Component**
* **About Component**
* **Contact Component**

1. **App Component (App.js)**

import React, { useState } from 'react';

import Home from './Home';

import About from './About';

import Contact from './Contact';

function App() {

const [user, setUser] = useState({ name: 'John Doe', email: 'john.doe@example.com' });

return (

<div>

<h1>Welcome to the MERN React App</h1>

<Home user={user} />

<About />

<Contact />

</div>

);

}

export default App;

1. **Home Component (Home.js)**

import React from 'react';

function Home(props) {

const { name, email } = props.user;

return (

<div>

<h2>Home</h2>

<p>User Name: {name}</p>

<p>User Email: {email}</p>

</div>

);

}

export default Home;

1. **About Component (About.js)**

import React from 'react';

function About() {

return (

<div>

<h2>About</h2>

<p>This is a React app that uses props and state for dynamic content.</p>

</div>

);

}

export default About;

1. **Contact Component (Contact.js)**

import React, { useState } from 'react';

function Contact() {

const [email, setEmail] = useState('');

const handleChange = (e) => {

setEmail(e.target.value);

};

const handleSubmit = (e) => {

e.preventDefault();

alert(`Contact info submitted: ${email}`);

};

return (

<div>

<h2>Contact</h2>

<form onSubmit={handleSubmit}>

<label>

Email:

<input type="email" value={email} onChange={handleChange} />

</label>

<button type="submit">Submit</button>

</form>

</div>

);

}

export default Contact;

**Running the Application**

After creating the components, the folder structure would look something like this:

src/

├── App.js

├── About.js

├── Contact.js

├── Home.js

├── index.js

To run the application, use the following command:

npm start

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Experiment 14: React Routing**

**Objective**

**a. Develop a react application that demonstrates the routing feature to navigate across different components of react and pass the data in between the components.**

**b. Develop a SPA in react for User Management System that interacts with the backend API using axios and perform CRUD operations on that.**

**Program**

**A: React Routing - Navigate Across Components and Pass Data**

* 1. **Setting Up the React Application**

npx create-react-app react-routing-app

cd react-routing-app

npm install react-router-dom

* 1. **Setting Up Routing**

# Generate components

npx generate-react-component home

npx generate-react-component about

npx generate-react-component profile

* 1. **Home Component** (Home.js)

import React from 'react';

import { Link } from 'react-router-dom';

function Home() {

const userData = { name: 'John Doe', email: 'john@example.com' };

return (

<div>

<h2>Home</h2>

<p>Welcome to the Home Page!</p>

<Link to={{

pathname: "/profile",

state: { user: userData } // Passing data through state

}}>

Go to Profile

</Link>

</div>

);

}

export default Home;

**About Component** (About.js)

import React from 'react';

function About() {

return (

<div>

<h2>About</h2>

<p>This is the About Page.</p>

</div>

);

}

export default About;

**Profile Component** (Profile.js)

import React from 'react';

import { useLocation } from 'react-router-dom';

function Profile() {

const location = useLocation();

const user = location.state?.user; // Retrieve passed data

return (

<div>

<h2>Profile</h2>

{user ? (

<div>

<p>Name: {user.name}</p>

<p>Email: {user.email}</p>

</div>

) : (

<p>No user data available.</p>

)}

</div>

);

}

export default Profile;

* 1. **App Component with Routing Setup**

import React from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import Home from './Home';

import About from './About';

import Profile from './Profile';

function App() {

return (

<Router>

<div>

<h1>React Routing Example</h1>

<nav>

<ul>

<li><a href="/home">Home</a></li>

<li><a href="/about">About</a></li>

</ul>

</nav>

<Switch>

<Route path="/home" component={Home} />

<Route path="/about" component={About} />

<Route path="/profile" component={Profile} />

</Switch>

</div>

</Router>

);

}

export default App;

**5.** **Test the Application**

npm start

**B: SPA for User Management System with CRUD Operations**

1. **Install Axios.**

npm install axios

1. **Setting Up the Backend API**

For this part, assume the following backend API endpoints:

* **POST** /api/users - Create a new user.
* **GET** /api/users - Get the list of users.
* **GET** /api/users/:id - Get a user by ID.
* **PUT** /api/users/:id - Update a user by ID.
* **DELETE** /api/users/:id - Delete a user by ID.

The backend should be implemented in **Node.js** with **Express** and **MongoDB**.

1. **Creating the Components**

**UserList Component (UserList.js)**

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

import axios from 'axios';

function UserList() {

const [users, setUsers] = useState([]);

useEffect(() => {

axios.get('http://localhost:5000/api/users')

.then(response => {

setUsers(response.data);

})

.catch(error => {

console.error('There was an error fetching users!', error);

});

}, []);

const handleDelete = (id) => {

axios.delete(`http://localhost:5000/api/users/${id}`)

.then(response => {

setUsers(users.filter(user => user.\_id !== id));

})

.catch(error => {

console.error('There was an error deleting the user!', error);

});

};

return (

<div>

<h2>User List</h2>

<ul>

{users.map(user => (

<li key={user.\_id}>

{user.name} ({user.email})

<button onClick={() => handleDelete(user.\_id)}>Delete</button>

</li>

))}

</ul>

</div>

);

}

export default UserList;

**UserForm Component (UserForm.js)**

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

import axios from 'axios';

function UserForm({ match, history }) {

const [user, setUser] = useState({ name: '', email: '' });

const userId = match.params.id; // Extract user ID from URL

useEffect(() => {

if (userId) {

axios.get(`http://localhost:5000/api/users/${userId}`)

.then(response => {

setUser(response.data);

})

.catch(error => {

console.error('There was an error fetching the user data!', error);

});

}

}, [userId]);

const handleChange = (e) => {

setUser({ ...user, [e.target.name]: e.target.value });

};

const handleSubmit = (e) => {

e.preventDefault();

if (userId) {

axios.put(`http://localhost:5000/api/users/${userId}`, user)

.then(() => {

history.push('/users');

})

.catch(error => {

console.error('There was an error updating the user!', error);

});

} else {

axios.post('http://localhost:5000/api/users', user)

.then(() => {

history.push('/users');

})

.catch(error => {

console.error('There was an error creating the user!', error);

});

}

};

return (

<div>

<h2>{userId ? 'Edit User' : 'Create User'}</h2>

<form onSubmit={handleSubmit}>

<label>

Name:

<input

type="text"

name="name"

value={user.name}

onChange={handleChange}

/>

</label>

<br />

<label>

Email:

<input

type="email"

name="email"

value={user.email}

onChange={handleChange}

/>

</label>

<br />

<button type="submit">{userId ? 'Update' : 'Create'}</button>

</form>

</div>

);

}

export default UserForm;

1. **App Component with Routing**

import React from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import UserList from './UserList';

import UserForm from './UserForm';

function App() {

return (

<Router>

<div>

<h1>User Management</h1>

<Switch>

<Route path="/users" component={UserList} />

<Route path="/user/:id" component={UserForm} />

<Route path="/create" component={UserForm} />

</Switch>

</div>

</Router>

);

}

export default App;

1. **Test the Application**

npm start

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Experiment 15: Single Page Application in Angular / React**

**Objective**

A TODO application serves as a simple yet powerful tool to help individuals and teams organize their tasks, manage priorities, and enhance productivity. TODO applications provide a structured and efficient way for individuals and teams to manage tasks, prioritize work, and achieve their goals. Develop a Single Page TODO Application in Angular / React to manage the daily tasks with the following features:

a. Task Creation: Allow users to create new tasks with a title, description, due date, and priority level. Provide a straightforward interface for entering task details.

b. Task Listing: Display a list of all tasks with essential details. Tasks can be organized based on different criteria such as due date, priority, or completion status.

c. Task Editing and Updating: Enable users to edit task details, including the ability to modify the title, description, due date, and priority. Changes should be reflected in real-time.

d. Task Deletion: Provide the option to delete tasks that are no longer relevant or completed. Include a confirmation prompt to prevent accidental deletions.

e. Task Completion: Allow users to mark tasks as completed or mark them with a specific status. Completed tasks may be moved to a separate section or visually differentiated.

f. User Authentication and Authorization: Implement user accounts with authentication to ensure data privacy. Differentiate between users and provide appropriate authorization levels.

g. Data Persistence: Ensure that tasks are persistently stored, so users can access their TODO lists even after closing and reopening the application.

1: Setting Up the Project

npx create-react-app todo-app

cd todo-app

npm install firebase react-router-dom

2: Firebase Configuration for Data Persistence and Authentication

1. **Set up Firebase Project:**
   * Go to Firebase Console.
   * Create a new Firebase project.
   * Enable **Firebase Authentication** (Email/Password).
   * Enable **Firestore Database** for storing tasks.
2. **Configure Firebase in React:**

In src/firebase.js, configure Firebase SDK.

import firebase from 'firebase/app';

import 'firebase/auth';

import 'firebase/firestore';

// Firebase configuration

const firebaseConfig = {

apiKey: "YOUR\_API\_KEY",

authDomain: "YOUR\_AUTH\_DOMAIN",

projectId: "YOUR\_PROJECT\_ID",

storageBucket: "YOUR\_STORAGE\_BUCKET",

messagingSenderId: "YOUR\_MESSAGING\_SENDER\_ID",

appId: "YOUR\_APP\_ID",

};

// Initialize Firebase

firebase.initializeApp(firebaseConfig);

const db = firebase.firestore();

const auth = firebase.auth();

export { db, auth };

**3: User Authentication**

We will implement **sign-up**, **login**, and **logout** functionalities.

1. **Signup Component (Signup.js):**

import React, { useState } from 'react';

import { auth } from './firebase';

import { useHistory } from 'react-router-dom';

function Signup() {

const [email, setEmail] = useState('');

const [password, setPassword] = useState('');

const history = useHistory();

const handleSubmit = (e) => {

e.preventDefault();

auth.createUserWithEmailAndPassword(email, password)

.then(() => history.push('/'))

.catch((error) => alert(error.message));

};

return (

<div>

<h2>Sign Up</h2>

<form onSubmit={handleSubmit}>

<input

type="email"

placeholder="Email"

value={email}

onChange={(e) => setEmail(e.target.value)}

/>

<input

type="password"

placeholder="Password"

value={password}

onChange={(e) => setPassword(e.target.value)}

/>

<button type="submit">Sign Up</button>

</form>

</div>

);

}

export default Signup;

Login Component (Login.js):

import React, { useState } from 'react';

import { auth } from './firebase';

import { useHistory } from 'react-router-dom';

function Login() {

const [email, setEmail] = useState('');

const [password, setPassword] = useState('');

const history = useHistory();

const handleSubmit = (e) => {

e.preventDefault();

auth.signInWithEmailAndPassword(email, password)

.then(() => history.push('/'))

.catch((error) => alert(error.message));

};

return (

<div>

<h2>Login</h2>

<form onSubmit={handleSubmit}>

<input

type="email"

placeholder="Email"

value={email}

onChange={(e) => setEmail(e.target.value)}

/>

<input

type="password"

placeholder="Password"

value={password}

onChange={(e) => setPassword(e.target.value)}

/>

<button type="submit">Login</button>

</form>

</div>

);

}

export default Login;

Logout Functionality (Navbar.js):

import React from 'react';

import { auth } from './firebase';

import { Link } from 'react-router-dom';

function Navbar() {

const handleLogout = () => {

auth.signOut();

};

return (

<nav>

<Link to="/">Home</Link>

<Link to="/login">Login</Link>

<Link to="/signup">Sign Up</Link>

<button onClick={handleLogout}>Logout</button>

</nav>

);

}

export default Navbar;

4: TODO App Core Functionality

* 1. **Creating a Task (TaskForm.js)**

import React, { useState } from 'react';

import { db, auth } from './firebase';

function TaskForm({ refreshTasks }) {

const [title, setTitle] = useState('');

const [description, setDescription] = useState('');

const [dueDate, setDueDate] = useState('');

const [priority, setPriority] = useState('');

const handleSubmit = (e) => {

e.preventDefault();

const userId = auth.currentUser.uid;

db.collection('tasks').add({

userId,

title,

description,

dueDate,

priority,

completed: false,

})

.then(() => {

setTitle('');

setDescription('');

setDueDate('');

setPriority('');

refreshTasks();

});

};

return (

<form onSubmit={handleSubmit}>

<input

type="text"

placeholder="Task Title"

value={title}

onChange={(e) => setTitle(e.target.value)}

required

/>

<textarea

placeholder="Description"

value={description}

onChange={(e) => setDescription(e.target.value)}

/>

<input

type="date"

value={dueDate}

onChange={(e) => setDueDate(e.target.value)}

/>

<select

value={priority}

onChange={(e) => setPriority(e.target.value)}

>

<option value="Low">Low</option>

<option value="Medium">Medium</option>

<option value="High">High</option>

</select>

<button type="submit">Add Task</button>

</form>

);

}

export default TaskForm;

* 1. **Displaying Tasks (TaskList.js)**

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

import { db, auth } from './firebase';

function TaskList() {

const [tasks, setTasks] = useState([]);

useEffect(() => {

const userId = auth.currentUser.uid;

const unsubscribe = db.collection('tasks')

.where('userId', '==', userId)

.onSnapshot(snapshot => {

const taskList = snapshot.docs.map(doc => ({

id: doc.id,

...doc.data(),

}));

setTasks(taskList);

});

return unsubscribe;

}, []);

const handleDelete = (id) => {

db.collection('tasks').doc(id).delete();

};

const handleComplete = (id, completed) => {

db.collection('tasks').doc(id).update({ completed: !completed });

};

return (

<div>

<h2>Your Tasks</h2>

{tasks.map((task) => (

<div key={task.id}>

<h3>{task.title}</h3>

<p>{task.description}</p>

<p>Due: {task.dueDate} | Priority: {task.priority}</p>

<button onClick={() => handleComplete(task.id, task.completed)}>

{task.completed ? 'Mark as Incomplete' : 'Mark as Completed'}

</button>

<button onClick={() => handleDelete(task.id)}>Delete</button>

</div>

))}

</div>

);

}

export default TaskList;

5: Main App and Routing

App Component with Routing

import React from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import TaskForm from './TaskForm';

import TaskList from './TaskList';

import Navbar from './Navbar';

import Signup from './Signup';

import Login from './Login';

function App() {

return (

<Router>

<Navbar />

<Switch>

<Route path="/signup" component={Signup} />

<Route path="/login" component={Login} />

<Route path="/" exact>

<TaskForm />

<TaskList />

</Route>

</Switch>

</Router>

);

}

export default App;

**Conclusion**

In this **TODO application**, we:

* Implemented **user authentication** using Firebase (signup, login, logout).
* Created tasks with fields such as **title**, **description**, **due date**, and **priority**.
* Implemented **CRUD** operations for tasks: create, read, update, and delete.
* Used **Firestore** for persistent storage of tasks and **Firebase Authentication** for user privacy.
* Implemented **task completion** feature to mark tasks as done.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_