**WEEK 6-MANDATORY HANDS ON EXERCISES**

**Module 10 – Single Page Application framework - React**

**Lab Documentation: Introduction to React & Single Page Applications**

**Objectives**

By the end of this lab, learners will be able to:

1. Define a Single Page Application (SPA) and explain its advantages.
2. Define React and describe how it works internally.
3. Differentiate between Single Page Applications (SPA) and Multi Page Applications (MPA).
4. Evaluate the pros and cons of using SPA architecture.
5. Describe the concept and role of the Virtual DOM in React.
6. Explain key features of React as a JavaScript library.
7. Set up a React development environment using Node.js and create-react-app.
8. Create and run a basic React project that displays a message on the browser.

**Theory Concepts**

**1. Single Page Application (SPA)**

A Single Page Application (SPA) is a type of web application that loads a single HTML page and dynamically updates the content as the user interacts with the app, without reloading the entire page.

**Benefits of SPA:**

* Faster page transitions and improved user experience.
* Less server bandwidth usage as only necessary data is exchanged.
* Enables building rich and responsive applications.

**2. What is React?**

React is a JavaScript library developed by Meta (formerly Facebook) used for building user interfaces, especially for SPAs. It allows developers to build reusable components that manage their own state.

**How React Works:**

* React uses a declarative approach to build UIs.
* It manages a virtual representation of the DOM.
* React automatically updates the actual DOM when the data (state/props) changes.

**3. SPA vs MPA**

| **Aspect** | **SPA** | **MPA** |
| --- | --- | --- |
| Page Reload | No | Yes |
| Speed | Fast after initial load | Slower due to frequent server calls |
| SEO Optimization | Difficult | Better suited |
| Backend Dependency | Less | More |
| User Experience | Smooth and dynamic | Traditional with full page reloads |

**4. Pros and Cons of SPA**

**Pros:**

* Fast and responsive user interactions.
* Reduced server load and network traffic.
* Enables mobile-app-like experience in web apps.

**Cons:**

* Poor SEO performance without additional setup.
* First-time load can be slower due to JavaScript bundles.
* More complex frontend development and routing logic.

**5. Virtual DOM**

The Virtual DOM is a lightweight copy of the actual DOM. React uses the Virtual DOM to efficiently update the user interface. It compares the current version with the previous one (diffing) and applies only the necessary changes to the actual DOM.

This process significantly improves rendering performance and user experience.

**6. Features of React**

* **Component-Based Architecture:** UI is broken into reusable components.
* **Declarative Syntax:** Developers describe what they want to see, and React handles the UI updates.
* **Virtual DOM:** Ensures faster and efficient UI rendering.
* **Unidirectional Data Flow:** Data flows in a single direction, making debugging easier.
* **JSX Support:** Allows writing HTML-like code inside JavaScript.
* **Rich Ecosystem:** Includes tools like React Router, Redux, and React Developer Tools.

**Hands-on Lab: Create a Basic React Application**

**Estimated Time: 30 Minutes**

**Prerequisites:**

* Node.js (with npm)
* Visual Studio Code
* Internet connection

**Step-by-Step Instructions**

**Step 1:**  
Download and install Node.js and npm from the official website:  
<https://nodejs.org/en/download/>

**Step 2:**  
Install the React development environment globally by installing the create-react-app CLI tool.

**Step 3:**  
Create a new React application with the name myfirstreact.

**Step 4:**  
Navigate into the created app folder named myfirstreact.

**Step 5:**  
Open the myfirstreact project folder in Visual Studio Code.

**Step 6:**  
In the src folder, locate and open the file named App.js.

**Step 7:**  
Delete the existing content of the App.js file.

**Step 8:**  
Replace the content with a new heading displaying the message:  
"Welcome to the first session of React"

**Step 9:**  
Use the terminal to run the application.

**Step 10:**  
Open a web browser and enter http://localhost:3000 in the address bar.  
The heading should appear on the webpage.

**Expected Output**

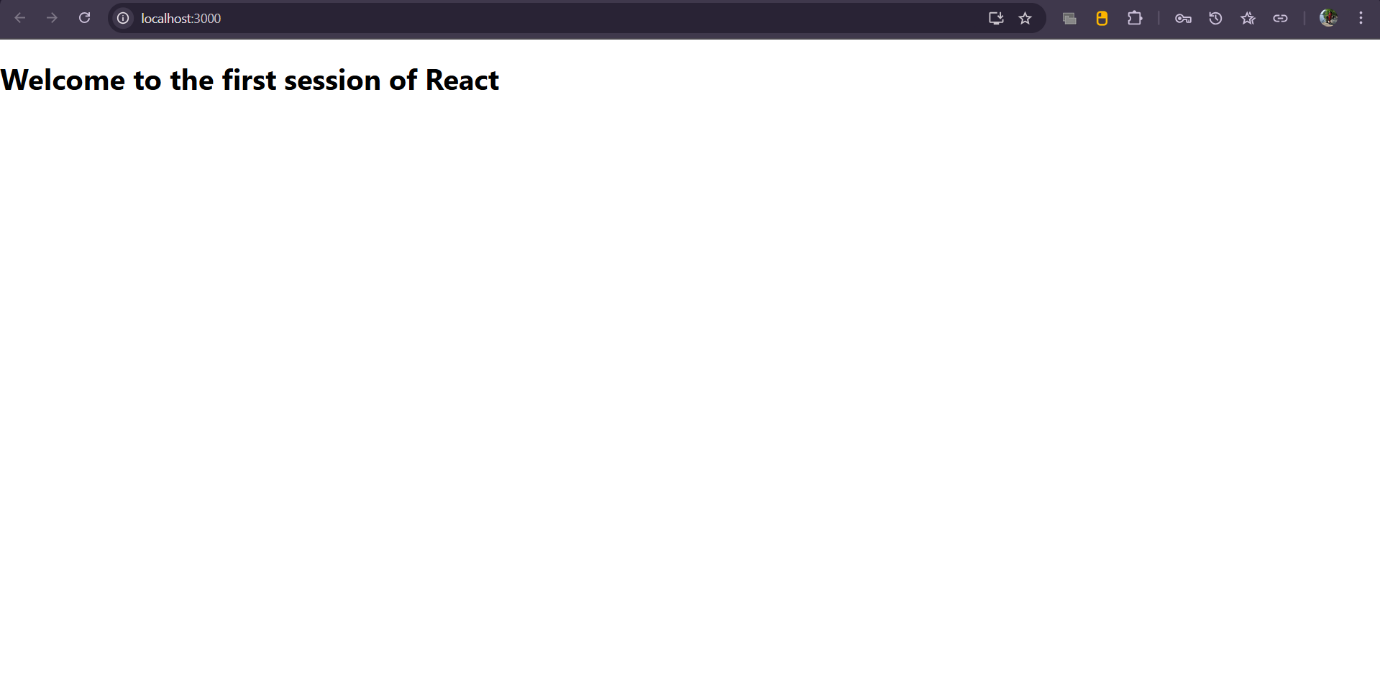
A heading should be displayed on the screen with the text:  
**"Welcome to the first session of React"**

This confirms that the React environment is correctly set up and running.

**Conclusion**

This lab introduced the concept of Single Page Applications and the fundamentals of React. Learners successfully set up their React environment and rendered content using React’s component system. Understanding the core differences between SPA and MPA, along with React’s internal working using the Virtual DOM, provides a strong foundation for building modern web applications.

Output:



**Lab Documentation: React Components and Rendering**

**Objectives**

By the end of this lab, learners will be able to:

* Explain the concept of React components.
* Identify the differences between React components and JavaScript functions.
* Describe the different types of React components.
* Explain the structure and purpose of class components.
* Explain the structure and purpose of function components.
* Define the component constructor in class components.
* Define the render() method and describe its role in rendering content.
* Create and render multiple components within a React application.

**Prerequisites**

To complete this hands-on lab, the following are required:

* Node.js installed on the system
* NPM (Node Package Manager)
* Visual Studio Code or any modern code editor

**Estimated Time**

30 minutes

**Theory Concepts**

**React Components**

React components are the building blocks of a React application. A component in React represents a reusable piece of the UI. Each component can maintain its own logic and rendering and may be composed of other components. Components allow applications to be built in a modular and manageable structure.

**Components vs JavaScript Functions**

While JavaScript functions can execute logic and return values, React components return elements that describe how a part of the UI should appear. Unlike regular functions, React components may maintain internal state and respond to user interaction by re-rendering the UI accordingly.

**Types of Components**

React supports two main types of components: class components and function components.

1. **Class Components**  
   These are ES6 classes that extend from React.Component. Class components must include a render() method that returns the UI to be displayed. They are capable of holding state and have access to lifecycle methods.
2. **Function Components**  
   These are simple JavaScript functions that return JSX. Originally, function components were stateless, but with the introduction of React Hooks, they can now use state and lifecycle-like features.

**Component Constructor**

The constructor in a class component is a special function used to initialize state and bind methods. It is called before the component is mounted and is defined using the constructor(props) syntax.

**Render Method**

The render() method is mandatory in class components. It returns the JSX that defines what should appear on the screen. Every time the component’s state or props change, the render() method is invoked to reflect those changes in the UI.

**Hands-on Lab: Student Management Portal with Multiple Components**

**Objective**

In this lab, learners will create a React project named **StudentApp** and build three components: **Home**, **About**, and **Contact**. Each component will display a different message related to a Student Management Portal. All three components will be rendered within the main application.

**Step-by-Step Instructions**

**Step 1**  
Create a new React application named **StudentApp** using the command-line interface. This initializes the project structure and installs the necessary dependencies.

**Step 2**  
Open the project folder in Visual Studio Code. Navigate to the src directory.

**Step 3**  
Inside the src folder, create a new folder named **Components**. This folder will contain all the custom components for the application.

**Step 4**  
Create a new file inside the Components folder named Home.js. This file will contain a class component that displays the message:  
*“Welcome to the Home page of Student Management Portal.”*

**Step 5**  
Create two additional files inside the Components folder: About.js and Contact.js.

* The About component should display the message:  
  *“Welcome to the About page of the Student Management Portal.”*
* The Contact component should display the message:  
  *“Welcome to the Contact page of the Student Management Portal.”*

**Step 6**  
Open the App.js file inside the src folder. Import all three components (Home, About, and Contact). Modify the App component to render the three components sequentially.

**Step 7**  
Open a terminal, navigate into the StudentApp project directory, and start the development server. This will compile the application and launch it in the browser.

**Step 8**  
Open a browser and enter the address http://localhost:3000. The application should load and display the messages from all three components on the screen.

**Expected Output**

Upon successful execution, the browser should display the following three messages:

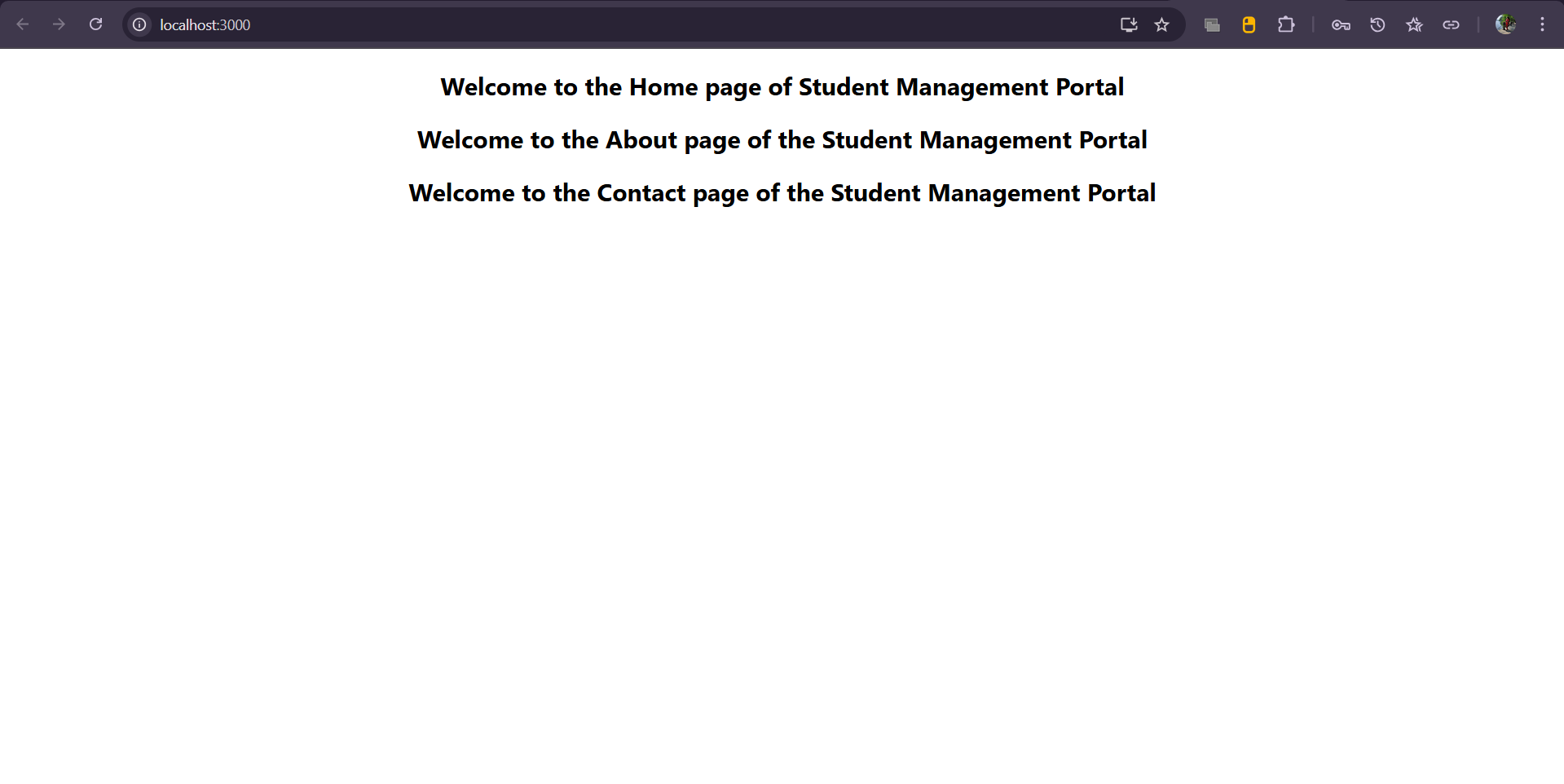
* Welcome to the Home page of Student Management Portal
* Welcome to the About page of the Student Management Portal
* Welcome to the Contact page of the Student Management Portal

Each message should appear as defined in the respective component files.

**Conclusion**

This lab guided learners through the creation and rendering of multiple components in a React application. By working with both the application structure and component logic, learners reinforced the foundational concepts of class components, function components, and the rendering lifecycle. The use of a modular component structure also demonstrated how large-scale applications can be effectively organized using React’s component-based architecture.

Output:



**Lab Documentation: Function Components and Styling in React**

**Objectives**

By the end of this lab, learners will be able to:

* Explain the structure and behavior of React components.
* Identify the distinctions between components and regular JavaScript functions.
* Recognize the different types of components available in React.
* Describe the functionality and purpose of class components.
* Describe the functionality and purpose of function components.
* Define the role of constructors in component initialization.
* Understand the use of the render() method in class components.
* Create a functional component in React.
* Apply external styles to React components.
* Render a functional component within a React application.

**Prerequisites**

To complete this hands-on lab, the following software and tools are required:

* Node.js installed on the system
* NPM (Node Package Manager)
* Visual Studio Code

**Estimated Time**

30 minutes

**Theory Concepts**

**React Components**

React components are independent, reusable building blocks that collectively form the user interface of a React application. These components can accept inputs (known as props), manage internal state (in case of class components or using hooks in function components), and render output dynamically in response to user interactions or data updates.

**Components vs JavaScript Functions**

While both React components and JavaScript functions are reusable blocks of logic, React components return JSX (JavaScript XML) which describes the structure and behavior of UI elements. JavaScript functions execute logic and return data or values, whereas React components are specifically designed to render elements within the DOM.

**Types of Components**

React supports two primary types of components:

1. **Class Components**  
   These are ES6 classes that inherit from the React.Component class. They include a render() method and can maintain state and utilize lifecycle methods.
2. **Function Components**  
   These are simpler components defined as JavaScript functions. They accept props and return JSX. With the advent of React Hooks, function components can now manage state and side effects as well.

**Component Constructor**

In class components, the constructor method is used to initialize component state and bind methods. It is called when the component is first created and sets up the initial configuration before rendering.

**Render Method**

The render() method is used in class components to return the JSX that should be displayed on the screen. It is called whenever the component’s state or props change, allowing the UI to update accordingly.

**Hands-on Lab: Building a Score Calculator Using a Function Component**

**Objective**

In this hands-on lab, learners will create a React application named **scorecalculatorapp**. A functional component named **CalculateScore** will be created. This component will accept four inputs—Name, School, Total, and Goal—to calculate and display the average score of a student. Additionally, external styles will be applied to the component using a separate stylesheet.

**Step-by-Step Instructions**

**Step 1**  
Create a new React application named **scorecalculatorapp** using the command-line interface. This will generate the required folder structure and install all dependencies.

**Step 2**  
Open the project folder in Visual Studio Code and navigate to the src directory.

**Step 3**  
Inside the src folder, create a new directory named **Components**. Within this folder, create a new file named CalculateScore.js. This file will contain the functional component responsible for accepting the required inputs and calculating the average score.

**Step 4**  
Create another directory inside the src folder named **Stylesheets**. Inside this folder, create a new CSS file named mystyle.css. This file will contain the styles to be applied to the **CalculateScore** component to enhance its appearance.

**Step 5**  
Open the App.js file in the src directory. Import the CalculateScore component from the Components folder. Modify the main application component to render the CalculateScore component, ensuring it appears on the homepage when the application loads.

**Step 6**  
In the terminal, navigate into the **scorecalculatorapp** directory and start the development server. This compiles the code and launches the app in the default web browser.

**Step 7**  
Open a browser and enter the address http://localhost:3000 to view the application. The user should see the interface for the score calculator along with the applied styles.

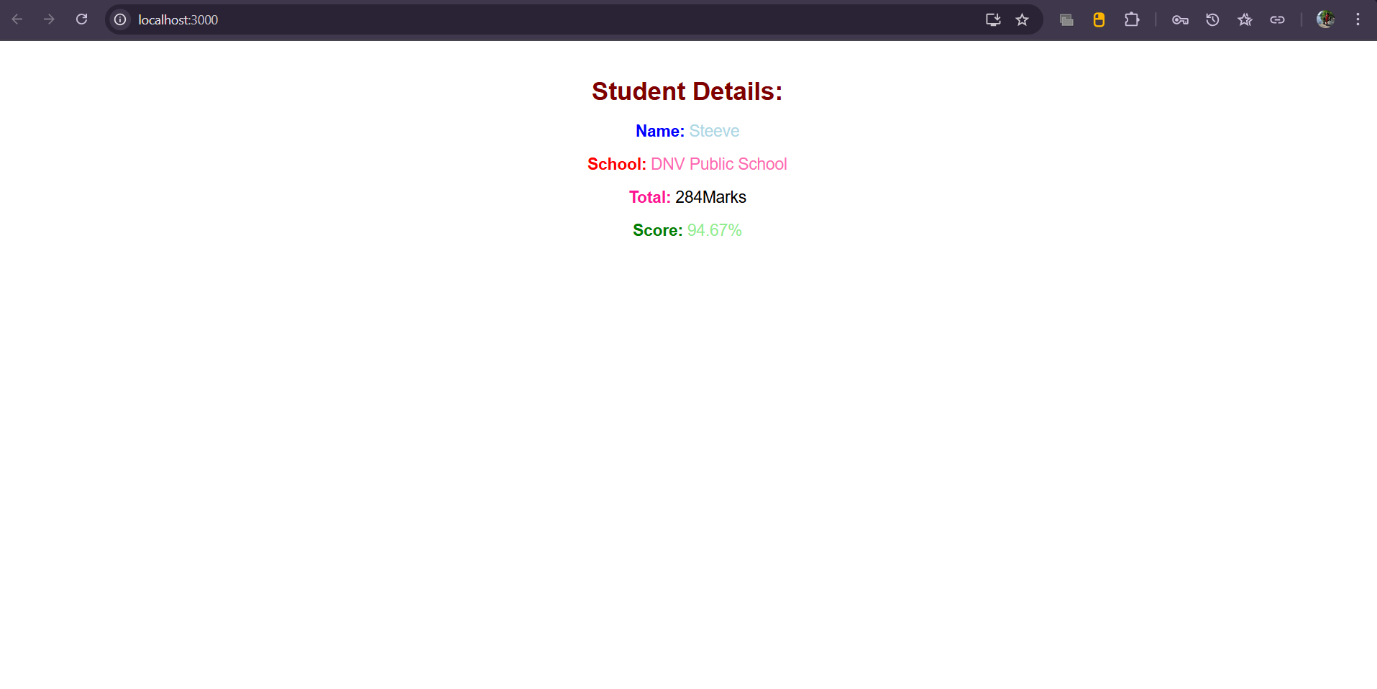
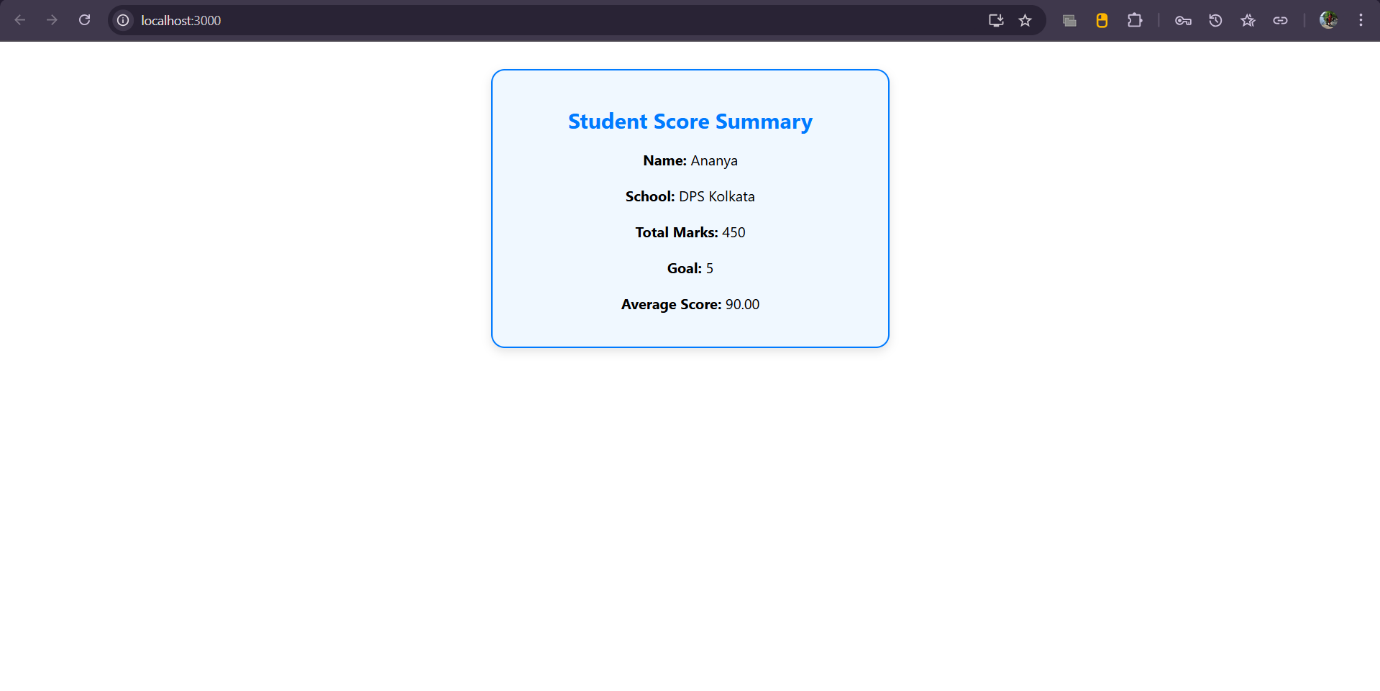
**Expected Output**

The browser should display the output of the CalculateScore functional component. The interface should show the student's Name, School, Total, and Goal, along with the calculated average score. The appearance of the component should reflect the styling defined in the external CSS file.

**Conclusion**

This lab demonstrated how to build and render a functional component in a React application. Learners applied fundamental concepts related to component creation, prop usage, and JSX rendering. Additionally, external styles were integrated using a custom stylesheet to separate structure from design. This hands-on exercise reinforced the practice of clean code separation and effective component-based architecture in React.

Output:



**React Component Lifecycle — Hands-on Lab Documentation**

**Objectives**

**1. Explain the Need and Benefits of Component Lifecycle**

React components go through a series of phases during their existence: mounting, updating, and unmounting. Lifecycle methods are special class functions that allow developers to control behavior at these different stages.

**Why are lifecycle methods needed?**

* To perform actions at specific times (e.g., fetching data after the component is inserted into the DOM).
* To clean up operations (like event listeners or timers) before the component is removed.
* To catch and handle errors during rendering or in lifecycle methods of child components.
* To optimize performance by controlling unnecessary re-rendering.

Lifecycle methods provide greater control, help organize code, and are essential for building robust, maintainable applications.

**2. Identify Various Lifecycle Hook Methods**

React class components come with a set of lifecycle hook methods, grouped by the phase in which they are invoked:

**Mounting Phase (when the component is being created and inserted into the DOM):**

* constructor()
* static getDerivedStateFromProps()
* render()
* componentDidMount()

**Updating Phase (when props or state changes):**

* static getDerivedStateFromProps()
* shouldComponentUpdate()
* render()
* getSnapshotBeforeUpdate()
* componentDidUpdate()

**Unmounting Phase (when the component is being removed):**

* componentWillUnmount()

**Error Handling:**

* componentDidCatch()
* static getDerivedStateFromError()

**3. List the Sequence of Steps in Rendering a Component**

When a class-based React component is rendered, the sequence of steps is as follows:

1. constructor()
2. static getDerivedStateFromProps()
3. render()
4. componentDidMount() (after the component is rendered into the DOM)

When the component is updated:

1. static getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

When an error occurs:

1. static getDerivedStateFromError()
2. componentDidCatch()

**Hands-on Lab Instructions**

**Estimated Time**: 60 minutes  
**Project Name**: blogapp

**Prerequisites**

Ensure the following are installed:

* Node.js
* NPM (comes with Node.js)
* Visual Studio Code

**Steps**

1. **Create a New React Application**
   * Use the create-react-app CLI tool to create a project named blogapp.
2. **Open the Project in Visual Studio Code**
   * Navigate to the project directory and open it in VS Code.
3. **Create Post.js Component**
   * Inside the src folder, create a new file named Post.js.
   * Define a class-based component representing a single blog post.
   * Include properties such as title and body.
4. **Create Posts.js Component**
   * In src, create another file named Posts.js.
   * Define a class-based component named Posts.
5. **Initialize Component State**
   * In the constructor of Posts, initialize the state with an empty array for storing posts.
6. **Create loadPosts() Method**
   * Define a method loadPosts() that fetches posts from the API https://jsonplaceholder.typicode.com/posts.
   * Use the Fetch API and update the component state with the retrieved posts.
7. **Implement componentDidMount()**
   * In the componentDidMount() lifecycle method, call loadPosts() to fetch data after the component has mounted.
8. **Render the Posts**
   * In the render() method, loop through the posts in the state and display the title and body using appropriate HTML tags like <h3> and <p>.
9. **Implement componentDidCatch()**
   * Add componentDidCatch() to handle any runtime errors within the component.
   * Display an alert message or log the error for debugging.
10. **Use the Posts Component in App**
    * Import the Posts component inside App.js.
    * Include <Posts /> within the main component’s JSX.
11. **Run the Application**
    * Use the npm start command to compile and run the React app.
    * The application should display the list of posts fetched from the API.

Output:



**Styling React Components — Hands-on Lab Documentation**

**Objectives**

**1. Understanding the Need for Styling React Components**

Styling is essential in any web application to create a visually appealing and user-friendly interface. In React, component-based architecture allows developers to build reusable UI units. Styling helps in:

* Defining how each component appears visually.
* Maintaining consistent design across different parts of the application.
* Improving user experience by enhancing readability and usability.
* Highlighting dynamic changes such as status or progress through visual cues.

React allows multiple styling approaches — inline styles, regular CSS, CSS Modules, styled-components, and more — offering flexibility depending on the project’s complexity and team preferences.

**2. Working with CSS Modules and Inline Styles**

**CSS Modules** provide scoped and reusable styling specific to a component, ensuring styles do not accidentally override others elsewhere in the app. This is especially helpful in large applications where global styles can create conflicts.

**Inline styles** are directly defined within the component using JavaScript objects. They are useful for dynamic styling, such as applying styles conditionally based on props or state.

**Benefits of using both:**

* CSS Modules offer organization and encapsulation.
* Inline styles allow dynamic, logic-driven visual changes.

Together, they help maintain modular and maintainable styling in a React application.

**In this hands-on lab, you will learn how to:**

* Style a React component using both CSS Modules and inline styles.
* Define reusable styles using a .module.css file.
* Apply styling using className (for CSS Module classes) and style (for inline styles).

**Prerequisites**

Before starting the lab, ensure the following tools are installed:

* Node.js
* NPM (comes with Node.js)
* Visual Studio Code

**Notes**

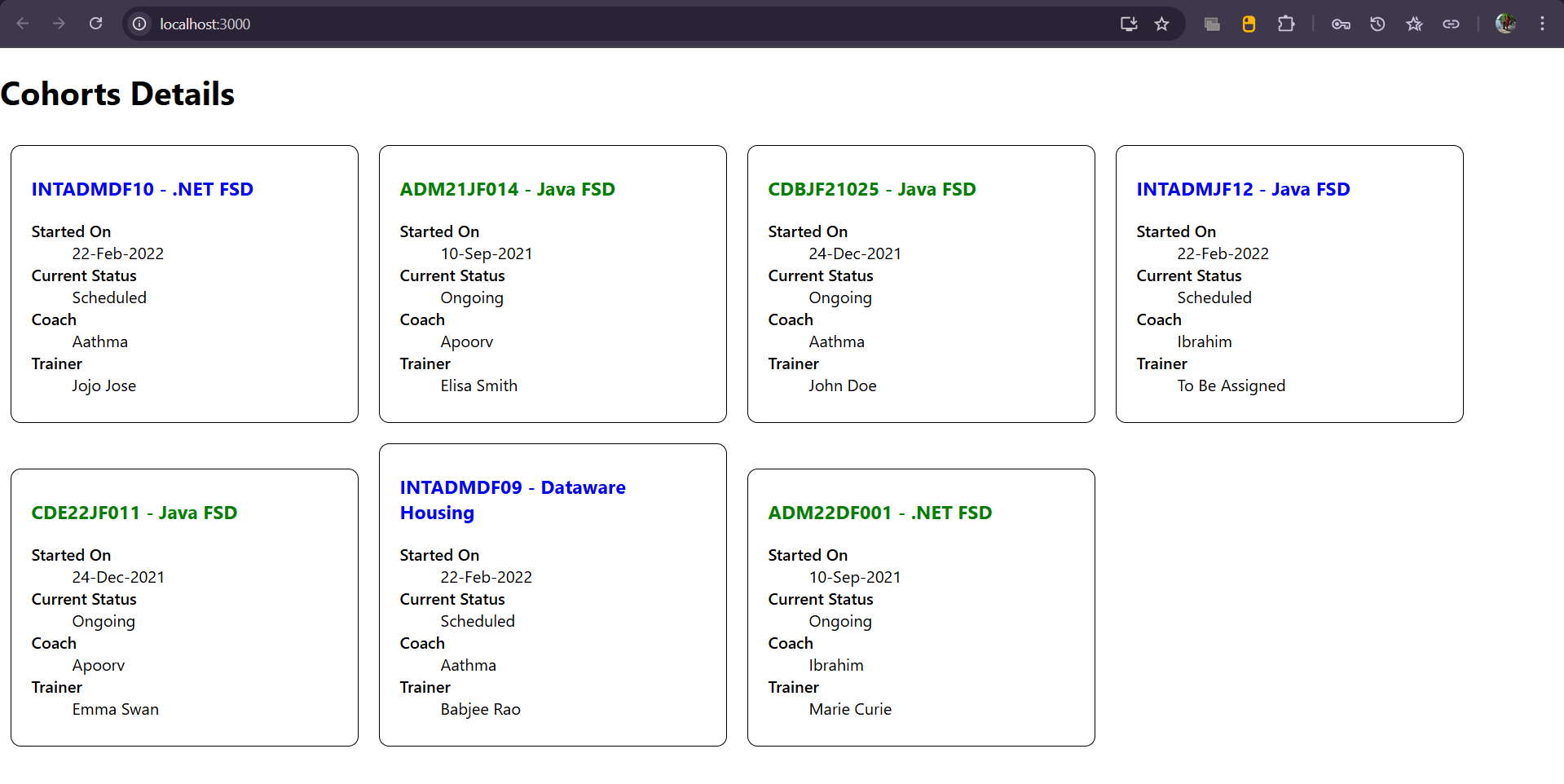
**Estimated Time to Complete**: 30 minutes  
**Context**: Styling a React dashboard that shows cohort details for Cognizant's internal My Academy team.

**Step-by-Step Instructions**

1. **Unzip the Project Folder**
   * Download and unzip the provided React application into a dedicated folder on your system.
2. **Install Dependencies**
   * Open Command Prompt or terminal.
   * Navigate to the unzipped project directory.
   * Run the appropriate command (npm install) to restore all required node packages.
3. **Open the Project in VS Code**
   * Use Visual Studio Code to open the project directory for editing.
4. **Create a CSS Module**
   * Inside the src folder (or where the CohortDetails component resides), create a new CSS Module file named CohortDetails.module.css.
5. **Define the 'box' Class**
   * Inside the CSS Module, define a class named box with the following styles:
     + Width: 300px
     + Display: inline-block
     + Margin: 10px on all sides
     + Padding: 10px (top and bottom), 20px (left and right)
     + Border: 1px solid black
     + Border radius: 10px
6. **Style the <dt> Tag**
   * Use a tag selector in the same CSS Module to style the <dt> HTML tag.
   * Set its font-weight to 500.
7. **Import CSS Module into Component**
   * Open the CohortDetails component file.
   * Import the CohortDetails.module.css file using ES module syntax.
8. **Apply the Box Class**
   * Assign the box class to the outermost container div of the component using the className attribute.
9. **Define Inline Styles for <h3>**
   * In the render or return block, apply an inline style to the <h3> element.
   * Use a conditional expression to set the font color:
     + Green when the cohort status is "ongoing"
     + Blue for all other statuses
10. **Verify the Final Output**

* The rendered output should visually resemble the reference design provided in the assignment (Figure 2).
* Each cohort block should be styled with the box class, and the heading should dynamically reflect the cohort status via color.

Output:



**React Router — Hands-on Lab Documentation**

**Objectives I Tackled**

**1. Why React Router?**

Okay, so React by default is a Single Page Application. Which means even though we feel like we're switching pages, it's all happening in one HTML file. But what if I want to have a proper navigation — like a homepage, a trainers list page, and individual trainer detail pages — all without reloading the site? That’s where React Router comes in.

It lets me:

* Switch between different components based on the URL
* Keep the app fast and smooth (no full-page reloads)
* Pass data through the URL itself (like passing an ID in /trainer/1)
* Handle dynamic routes easily

Basically, it gives me the power to make my React app feel like a full website.

**2. The Components I Used from React Router**

I used a few specific tools from the react-router-dom library:

* BrowserRouter: It’s the wrapper that enables routing in the whole app.
* Routes: Think of it like a container that holds all my individual routes.
* Route: Used to say "when this path is hit, show this component".
* Link: Replaces anchor tags (<a>) for React-style navigation.
* useParams: A hook that helped me grab the ID from the URL.

**3. Different Types of Routers**

I learned there are different kinds of routers, but I mainly focused on BrowserRouter because it works best for normal web apps. Just for reference though:

* BrowserRouter – uses clean URLs like /home
* HashRouter – URLs look like /#/home (useful when deploying static sites)
* MemoryRouter – doesn't mess with the browser URL, good for testing

**4. Passing Parameters Through the URL**

This is where it got interesting. I had to show trainer details based on who was clicked from the list. So I passed the trainer’s ID through the URL like this:  
/trainer/3

Then inside the TrainerDetail component, I used useParams() to grab that ID, searched my mock data, and showed the trainer’s full info. Super clean and practical.

**What I Did in the Hands-On Lab**

**Tools I Needed**

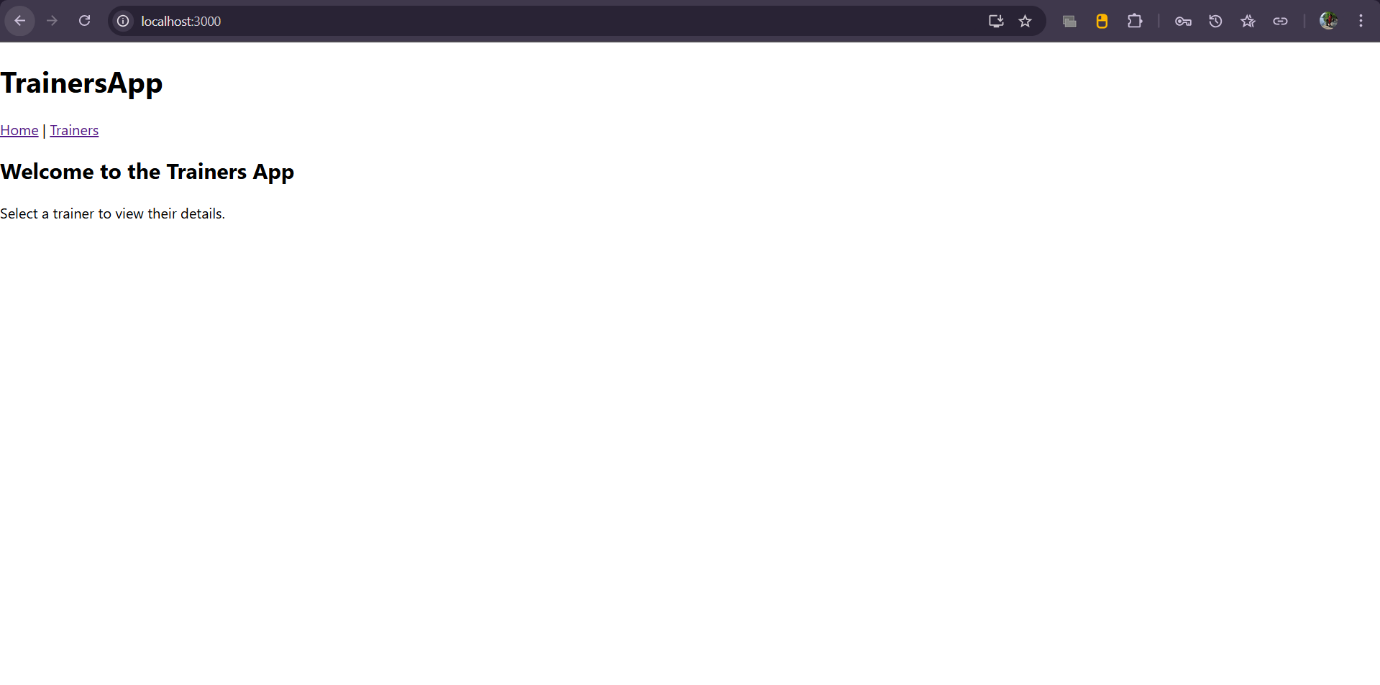
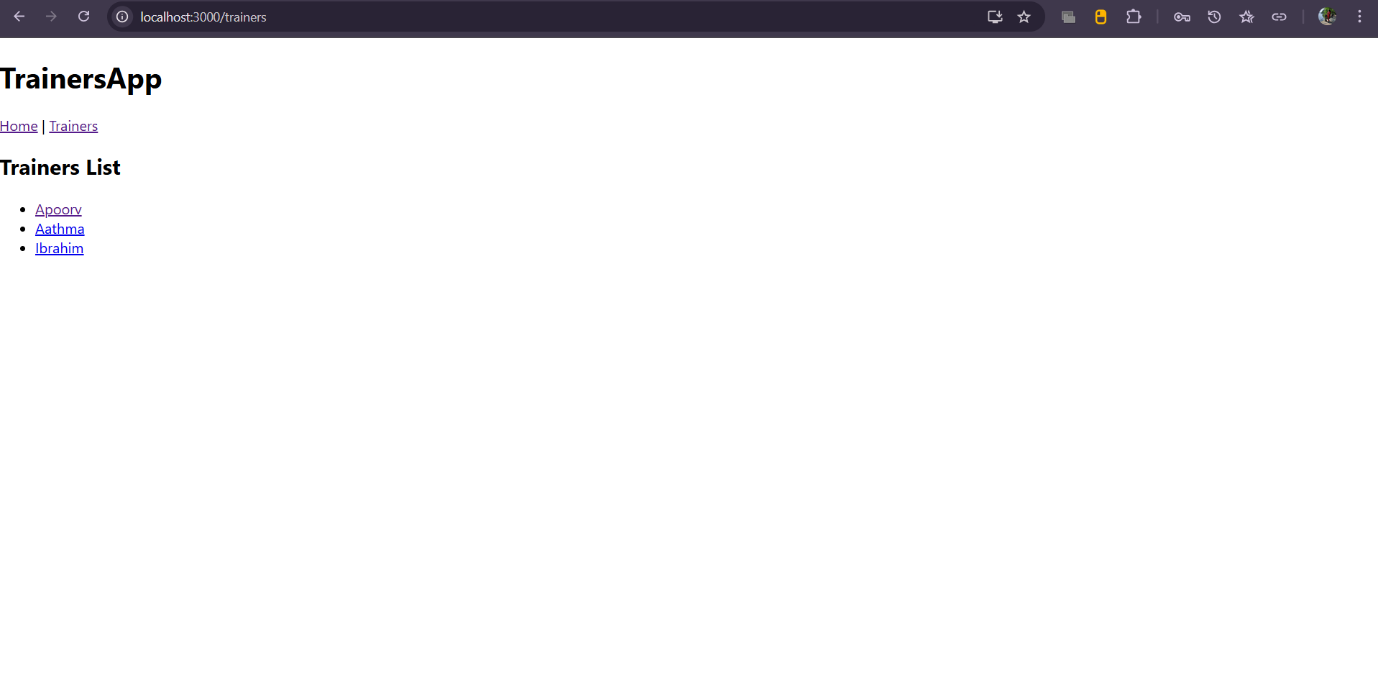
* Node.js
* NPM
* Visual Studio Code

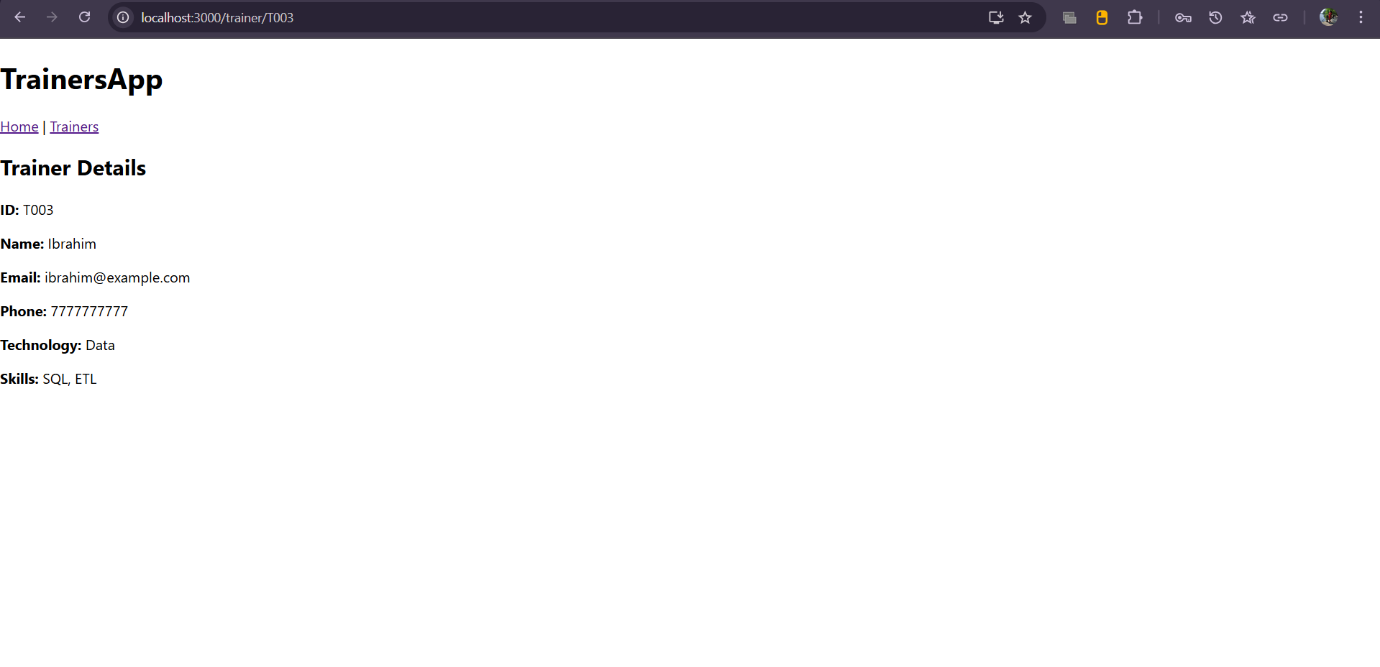
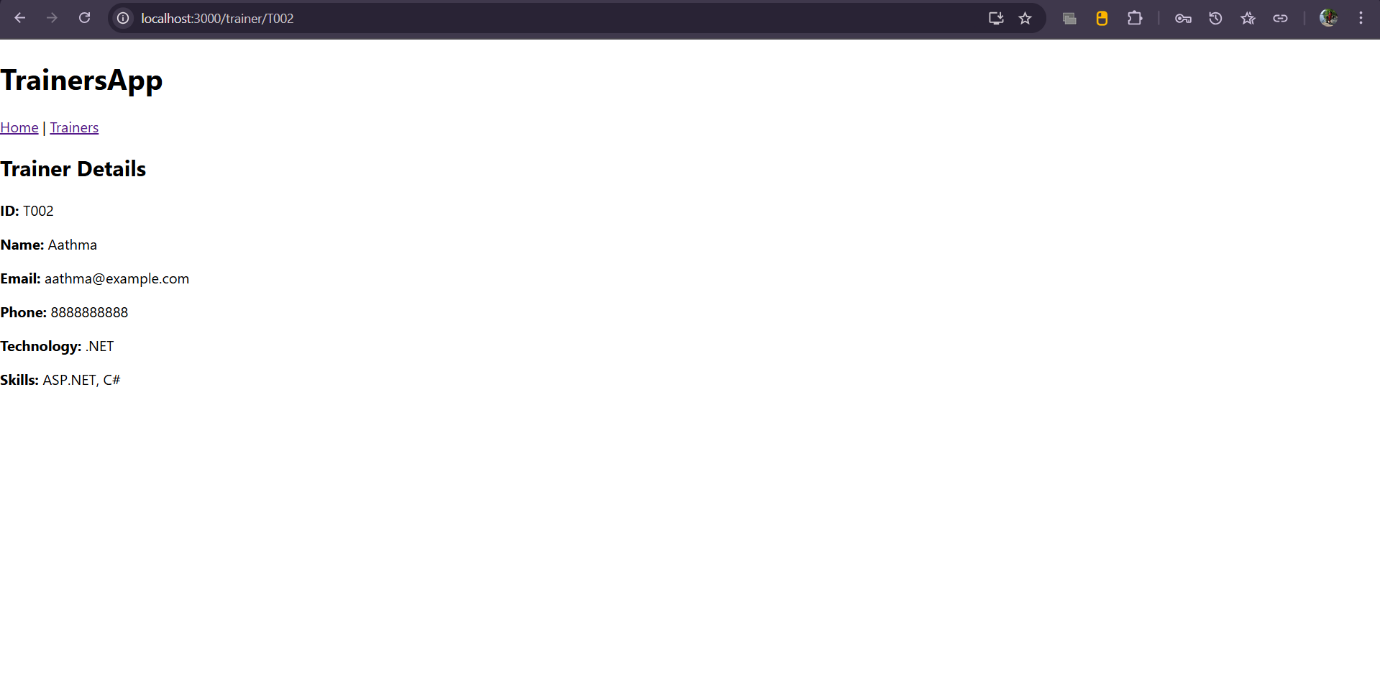
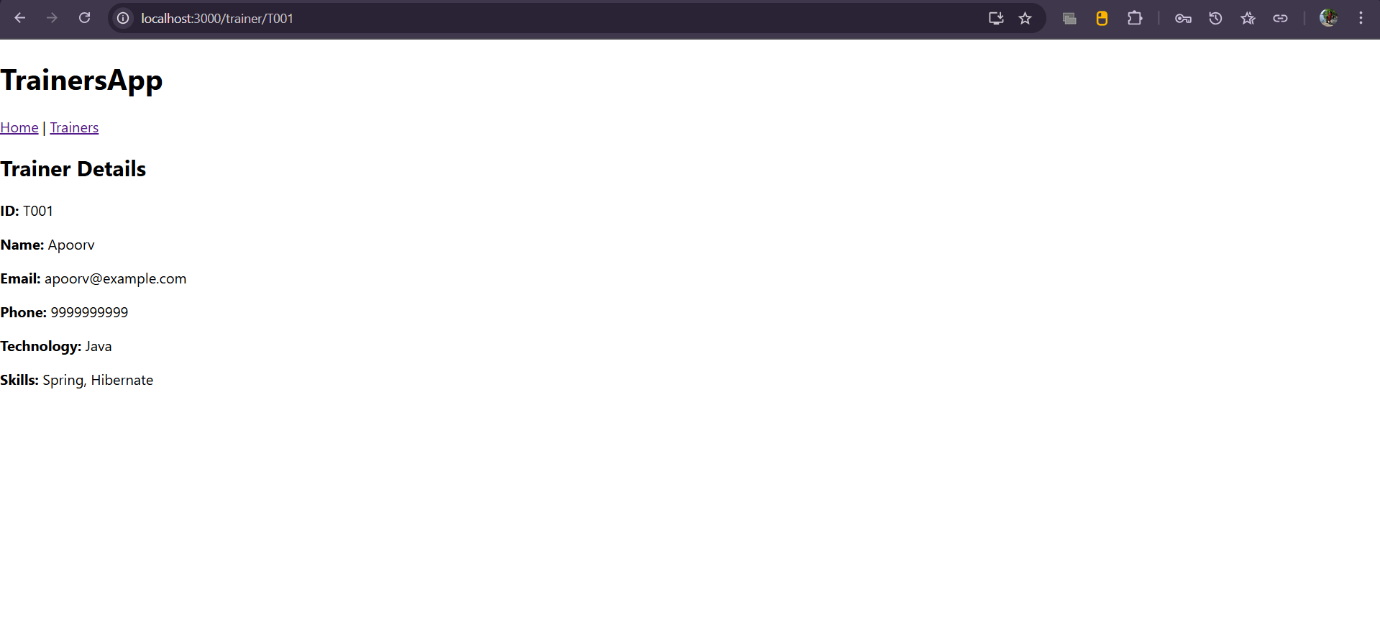
**Estimated Time: 60 mins**

(Felt like 40 tbh, if you’re not new to React)

**Here’s how I built the app (aka my checklist)**

1. **Created the React App**  
   I ran npx create-react-app TrainersApp and opened it up in VS Code.
2. **Created trainer.js**  
   Made a class called Trainer with all the needed fields: ID, name, phone, email, stream, skills.
3. **Mock Data Time (TrainersMock.js)**  
   I created an array of trainer objects in a new file so I had data to play with.
4. **Installed React Router**  
   Ran npm install react-router-dom to set everything up.
5. **Built TrainersList.js**  
   This component took trainer data and displayed a list of clickable names.  
   Clicking a name navigated to that trainer’s detail view.
6. **Made a Chill Home.js Page**  
   Just a simple welcome screen explaining the purpose of the app.
7. **Wired Up Routing in App.js**
   * Used BrowserRouter to wrap the app
   * Defined routes:
     + / → Home
     + /trainers → List of trainers
     + /trainer/:id → Detail view of a trainer
   * Used Link tags for navigation instead of anchor tags.
8. **Built TrainerDetails.js**  
   Used useParams to grab the ID from the URL, searched my mock data, and displayed all info for that trainer.
9. **Linked It All Together**  
   Updated TrainersList so each name linked to /trainer/<id>, and the TrainerDetails route handled it perfectly.
10. **Tested Everything**  
    Ran npm start, and the app worked like a charm. Navigation felt smooth, no reloads, data showed up exactly as expected.

Output:



**Props and Rendering in React — My POV Documentation**

**Objectives I Covered**

**1. Define Props**

Props (short for “properties”) are like the input parameters for React components. They’re used to pass data **from parent to child** components.

For example, if I have a Cart component, and I want to pass in the name and price of an item from OnlineShopping, I’ll send those as props.

They are **read-only** and cannot be changed by the child component — kind of like receiving a gift, but not being allowed to repack it.

**2. Explain Default Props**

Default props are fallback values. If the parent doesn’t pass a specific prop, React will use the default value instead.

For instance, if no item name is passed to Cart, I can set it to show "No item selected" by default. Super useful for avoiding empty displays.

**3. Difference Between State and Props**

| **Feature** | **Props** | **State** |
| --- | --- | --- |
| Mutability | Immutable (read-only) | Mutable (can be updated) |
| Ownership | Passed from parent | Managed within the component |
| Usage | Used to display data | Used to manage interactive behavior |

I think of **props** as inputs to a component and **state** as the internal memory of that component.

**4. Explain reactDOM.render()**

ReactDOM.render() is the method that tells React *what* to show and *where* to show it.

Syntax:

ReactDOM.render(<ComponentName />, document.getElementById('root'));

It takes a component (usually your root or App component) and renders it inside the HTML page, usually in the element with id="root".

**In This Lab, I Learned to:**

* Use props to pass item names and prices from one component to another
* Use ReactDOM.render() to render my root component
* Loop through an array in the parent component and pass each item as a prop to the child

**Prerequisites I Used**

* Node.js
* NPM
* Visual Studio Code

**Time Taken**

Estimated: 60 mins  
Felt like: 40 mins once I got the hang of props and .map().

**What I Did — Step by Step**

**1. Created the App**

Ran the command to create a new React app:

npx create-react-app shoppingapp

Opened it in VS Code and cleaned up the default files.

**2. Made Two Class Components**

* **OnlineShopping.js** — the main parent component
* **Cart.js** — the reusable child component for each item

**3. Created Properties in Cart**

Inside Cart, I created two props:

* itemname
* price

These were passed from the parent and displayed inside JSX using something like:

<h3>{this.props.itemname}</h3>

<p>{this.props.price}</p>

**4. Initialized Array in OnlineShopping**

Inside the OnlineShopping class, I made an array of 5 items

**5. Looped Through the Items**

Used .map() to loop through the array and rendered a Cart component for each item:

{items.map((item, index) => (

<Cart key={index} itemname={item.itemname} price={item.price} />

))}

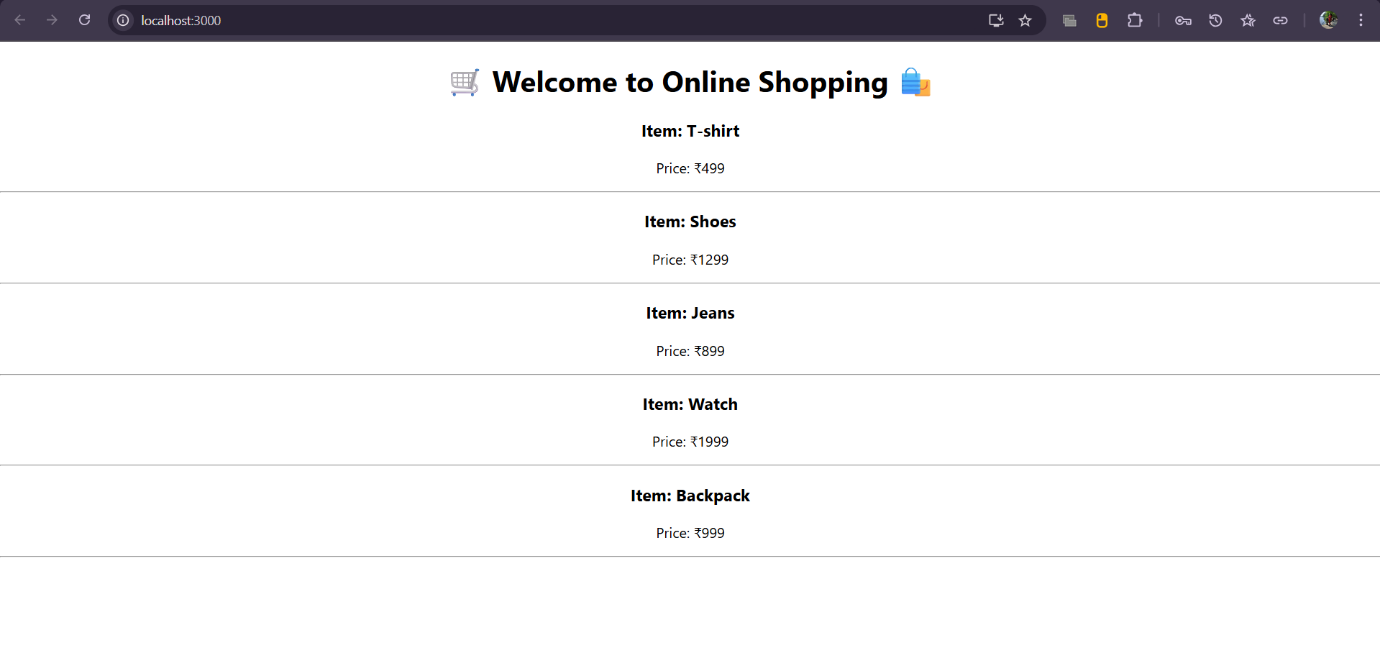
**6. Used reactDOM.render()**

In index.js, I used:

ReactDOM.render(<OnlineShopping />, document.getElementById('root'));

And— it displayed all five items with their name and price, just like a product list.

Output:



**React State – CountPeople Component (counterapp)**

**Objectives Answered in My POV**

1. **Explain React State:**

React State is like a component’s memory. It stores values that can change over time. When state changes, React re-renders the component automatically. In class components, this.state is used to declare state, and this.setState() is used to update it.

**Hands-On Lab Summary**

* **React App Name:** counterapp
* **Component Name:** CountPeople

The component manages the count of:

* People entering the mall.
* People exiting the mall.

**What I Did in This Lab**

* Used **Constructor** to define the state with two properties:
  + entrycount
  + exitcount
* Defined two methods:
  + UpdateEntry() → Increments entrycount
  + UpdateExit() → Increments exitcount
* Two buttons in the component:
  + **Login** → Triggers UpdateEntry()
  + **Exit** → Triggers UpdateExit()

