**React :Exercise-1**

**1. Define SPA and its benefits**

A **Single Page Application (SPA)** is a web application that loads a single HTML page and dynamically updates the content without refreshing the page. It uses AJAX to load data and render content without making full page requests.

**Benefits of SPA:**

* **Fast Load Time**: Once the initial page is loaded, subsequent interactions are faster.
* **Seamless User Experience**: No page reloads or interruptions during interactions.
* **Reduced Server Load**: After the initial load, fewer requests are sent to the server.

**2. Define React and identify its working**

**React** is a JavaScript library used to build user interfaces, especially for Single Page Applications (SPAs). It follows a **component-based architecture**, meaning the UI is divided into reusable components. React efficiently updates the DOM using its **Virtual DOM** concept.

**How it works:**

* React creates a virtual representation of the real DOM.
* When a component’s state changes, React updates the virtual DOM first, compares it with the real DOM, and updates only the changed parts.

**3. Explain Pros & Cons of Single-Page Application**

**Pros:**

* **Fast and responsive** after the initial load.
* **Better user experience** with smooth transitions.
* **Efficient communication** with the server (only requesting new data).

**Cons:**

* **SEO Challenges**: SPAs are harder to index by search engines.
* **First Load Time**: Initial loading may take longer since everything is bundled.
* **Dependency on JavaScript**: If JavaScript is disabled, the app won’t work.

**4. Explain about React**

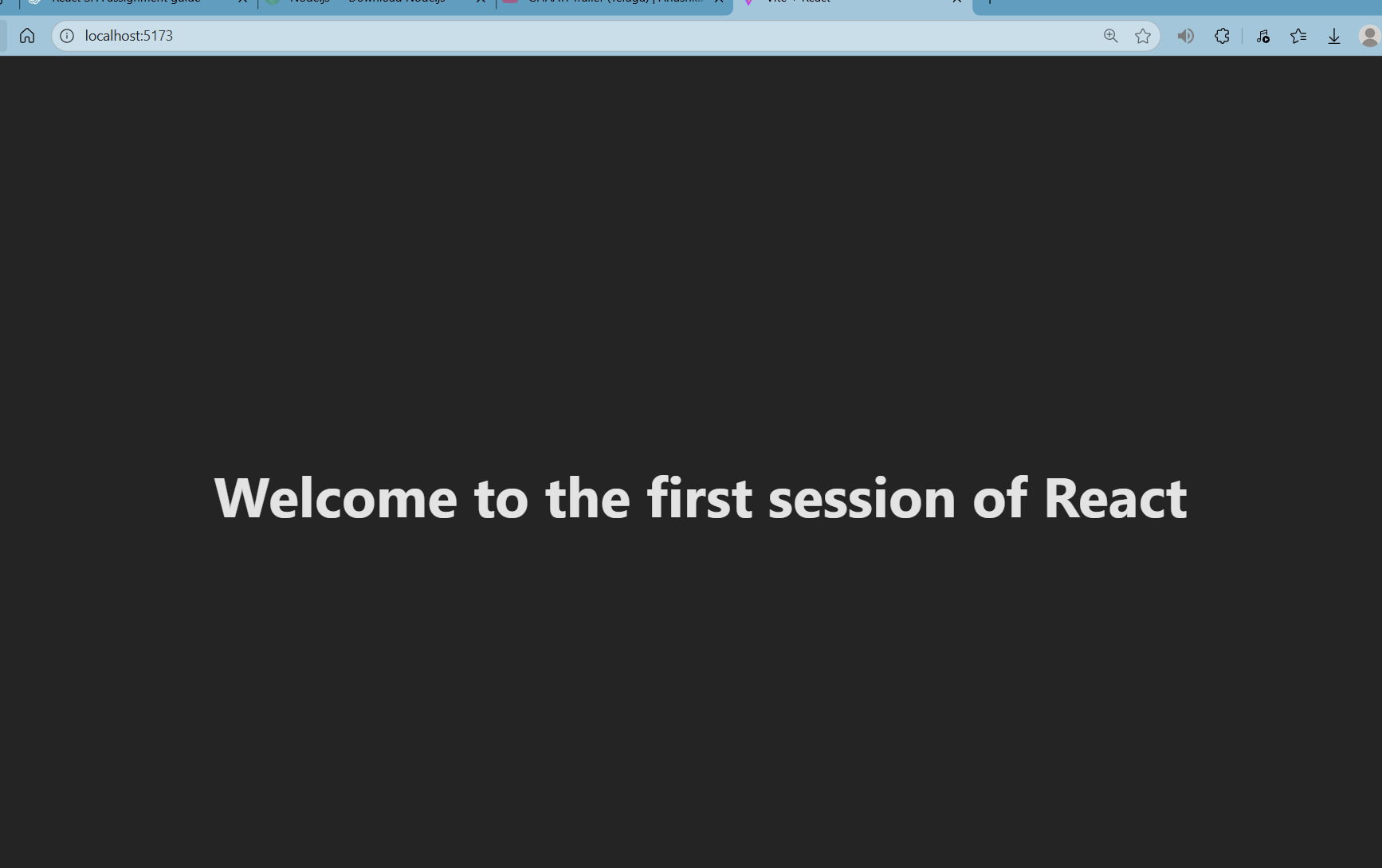
**React** is a declarative, efficient, and flexible JavaScript library for building user interfaces, particularly for SPAs. React allows you to design complex UIs by breaking them down into smaller, reusable components. It is maintained by Facebook and used by many companies worldwide.

**5. Define Virtual DOM**

The **Virtual DOM (VDOM)** is an in-memory representation of the actual DOM. It allows React to efficiently update the UI by comparing changes between the previous and current versions of the VDOM (called **diffing**) and applying only the necessary updates to the real DOM.

**6. Explain Features of React**

* **Component-Based**: React is built around the concept of components, making code more modular and reusable.
* **Declarative Syntax**: React uses JSX (a JavaScript extension) to describe UI components in a syntax similar to HTML.
* **Unidirectional Data Flow**: Data flows in one direction, making the app easier to manage and debug.
* **Virtual DOM**: React optimizes rendering by using the Virtual DOM, updating only the parts of the UI that have changed.
* **Reusable Components**: Components can be reused and composed to build complex UIs.
* **JSX (JavaScript Syntax Extension)**: React allows you to write HTML-like code within JavaScript, making it easy to build UI components.



**Exercise-2: React**

**Explain React Components:**React components are the building blocks of a React application. They are JavaScript functions or classes that accept inputs (called props) and return React elements describing how the UI should look.

Differences between Components and JavaScript Functions:

* React Components: Have the ability to manage internal state (if it's a class component), lifecycle methods, and can render UI based on inputs.
* JavaScript Functions: Are simply reusable blocks of code that return a value but do not manage UI or state on their own.

Types of Components:

* Class Components: Traditional React components that allow for more advanced features like state management and lifecycle methods.
* Function Components: Simpler, modern React components that use hooks for state and effects (if needed). They are often preferred for simpler logic.

Class Component:  
A class component extends the React.Component class and can hold state, handle events, and access lifecycle methods.

Example

class MyComponent extends React.Component {

render() {

return <h1>Hello from Class Component</h1>;

}

}

**Function Component**:  
A function component is a simpler type of component. It is just a JavaScript function that returns JSX.

function MyComponent() {

return <h1>Hello from Function Component</h1>;

}

**Component Constructor**:  
The **constructor** method is called when a class component is created. It is typically used to initialize state and bind methods.

constructor(props) {

super(props);

this.state = { message: 'Hello' };

}

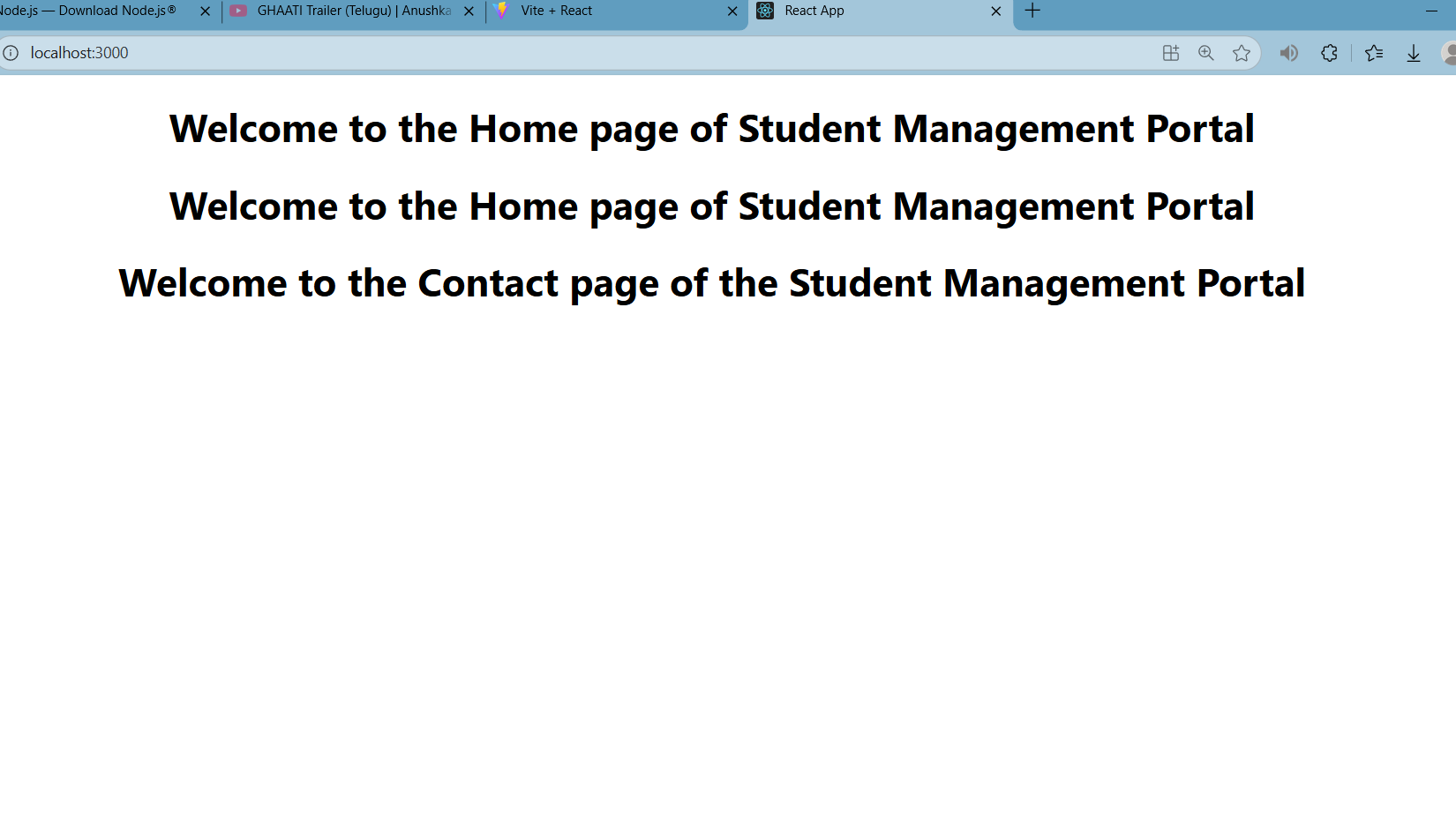
**Render() Function**:  
The **render()** function is required in class components. It returns JSX that determines what the UI should look like.

render() {

return <h1>{this.state.message}</h1>;

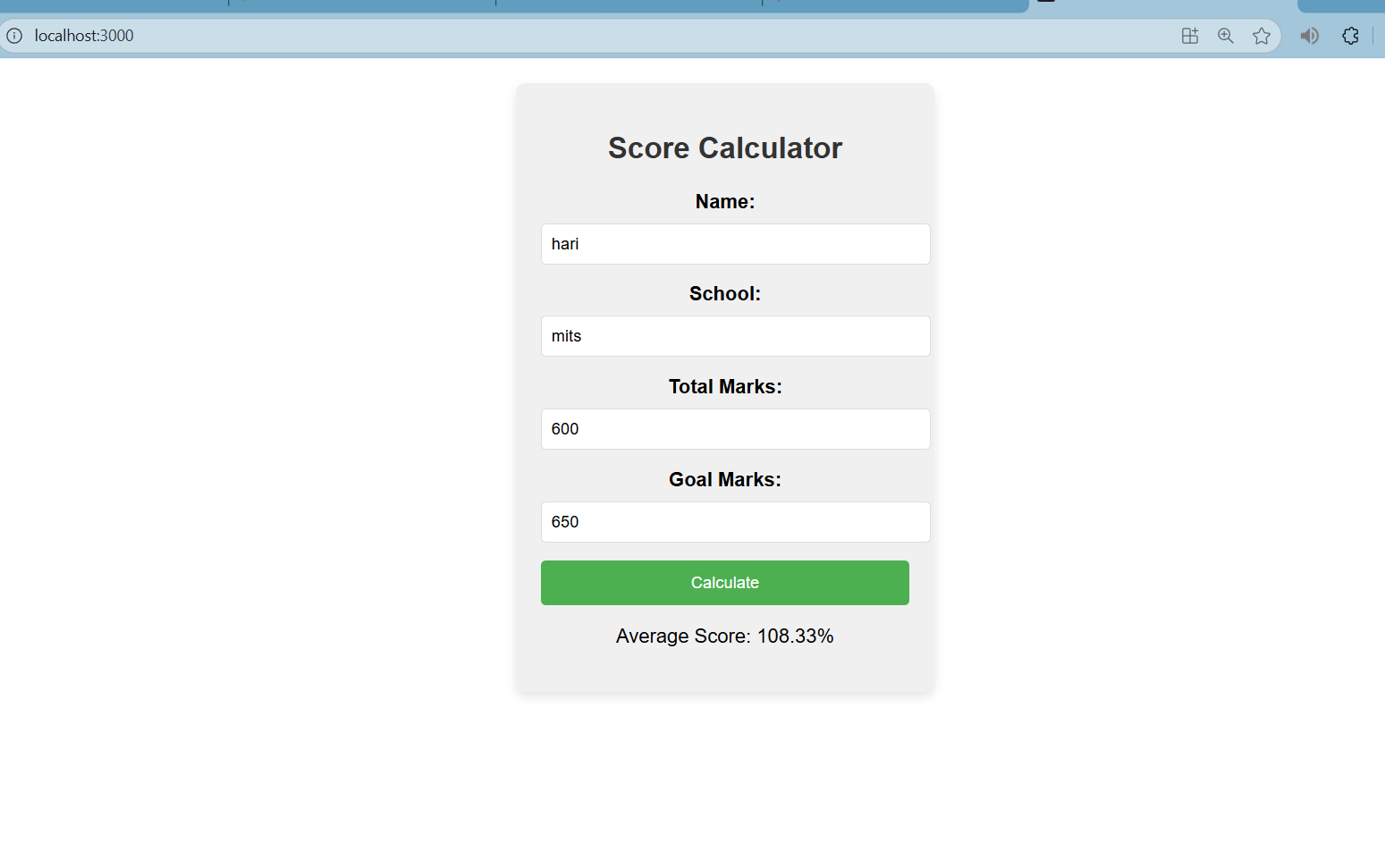
}

Output:



**React: Exercise 3**

Completed ScoreCalulator



**Exercise-4:**

**1. Explain the Need and Benefits of Component Life Cycle**

The **component life cycle** in React refers to the series of methods that are called at different stages of a component's existence (from creation to removal). The life cycle helps developers manage side effects, such as data fetching, and make the component behave as expected in different situations.

**Benefits:**

* **Efficient Data Fetching**: Fetch data when the component is ready to render.
* **Resource Management**: Clean up resources (e.g., timers, subscriptions) when the component is removed.
* **State Management**: Handle state changes and re-render components when needed.

**2. Identify Various Life Cycle Hook Methods**

For **Class Components**, common life cycle hook methods include:

* **constructor()**: Called before the component is mounted, used for initialization and state setup.
* **componentDidMount()**: Called after the component is mounted (useful for data fetching or setup).
* **shouldComponentUpdate()**: Decides if the component should re-render (optimization).
* **componentDidUpdate()**: Called after the component updates.
* **componentWillUnmount()**: Called just before the component is removed, used for cleanup.

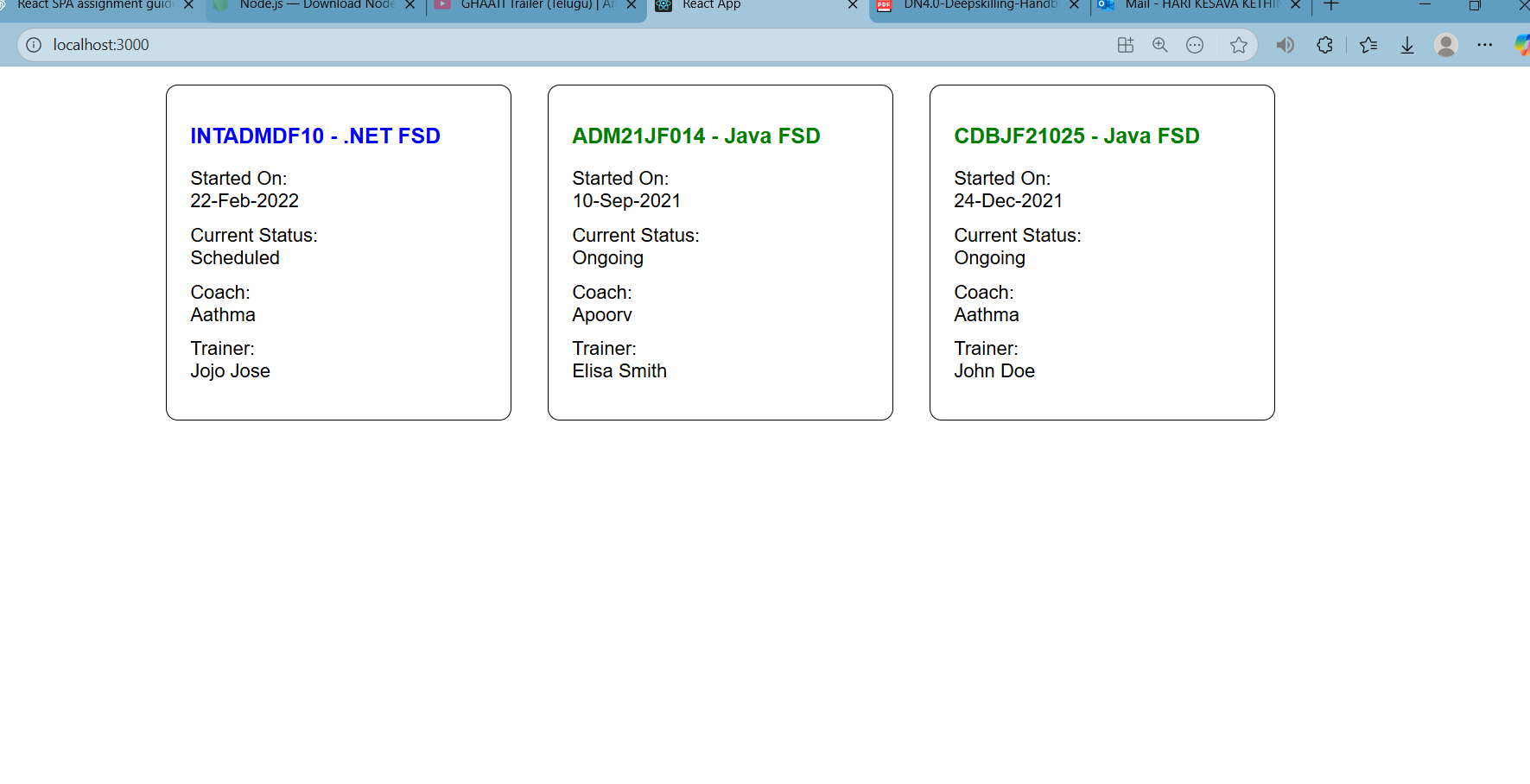
For **Function Components**, **Hooks** like useEffect are used to manage side effects.

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

1. **Initialization**: The component is created and initialized (i.e., constructor is called).
2. **Mounting**: The component is mounted to the DOM.
   * **componentDidMount()** is called (for class components).
3. **Rendering**: The component’s **render()** method is called, generating the UI.
4. **Updating**: If state or props change, the component will re-render.
   * **shouldComponentUpdate()** is called to decide if it should re-render.
   * **componentDidUpdate()** is called after an update.
5. **Unmounting**: The component is removed from the DOM.
   * **componentWillUnmount()** is called to clean up resources.



Week 5:



**Week 6:**

**1. Need and Benefits of React Router**

**React Router** is a standard library used for routing in **React** applications. It enables navigation between different components based on the **URL** in a **Single Page Application (SPA)** without refreshing the page.

**Benefits:**

* **Client-Side Navigation**: React Router allows for navigation without reloading the entire page. This provides a **smooth user experience** and is essential for building modern SPAs.
* **Dynamic Routing**: You can define routes that change based on user interactions or URL changes, making the app dynamic.
* **SEO-Friendly**: With proper configuration, React Router can improve SEO by allowing search engines to crawl different views of the app.
* **Modular Structure**: React Router allows developers to split the application into smaller, reusable components, leading to better organization and maintainability.

**2. Components in React Router**

React Router provides several components to manage routing:

* **BrowserRouter**: This is the top-level router component used to enable navigation within the app. It keeps the UI in sync with the URL.
* **Routes**: This component renders a set of **Route** components based on the current URL.
* **Route**: The Route component defines a **path** and which **component** should be rendered when the user navigates to that path.
* **Link**: A Link component is used to create clickable navigation items (like hyperlinks) that lead to different routes in the application without a page reload.
* **useParams**: A hook used to access the **parameters** passed through the URL.

**3. Types of Router Components**

React Router provides different types of routers based on how they handle the URL:

* **BrowserRouter**: The most common type, it uses the **HTML5 history API** to manipulate the URL and manage the browser’s history stack. It's used when building single-page applications where URL changes don’t require a full page reload.
* **HashRouter**: This type uses a **hash** in the URL (like example.com/#/home). It's useful for applications that don’t need a server-side setup and are hosted on static file servers.
* **MemoryRouter**: Stores the history of navigation in memory, without affecting the URL in the browser. It’s typically used for testing or in environments where the browser history is not important.

**4. Parameter Passing via URL**

One of the powerful features of React Router is the ability to pass parameters via the URL. These parameters can be retrieved and used within the components.

For example:

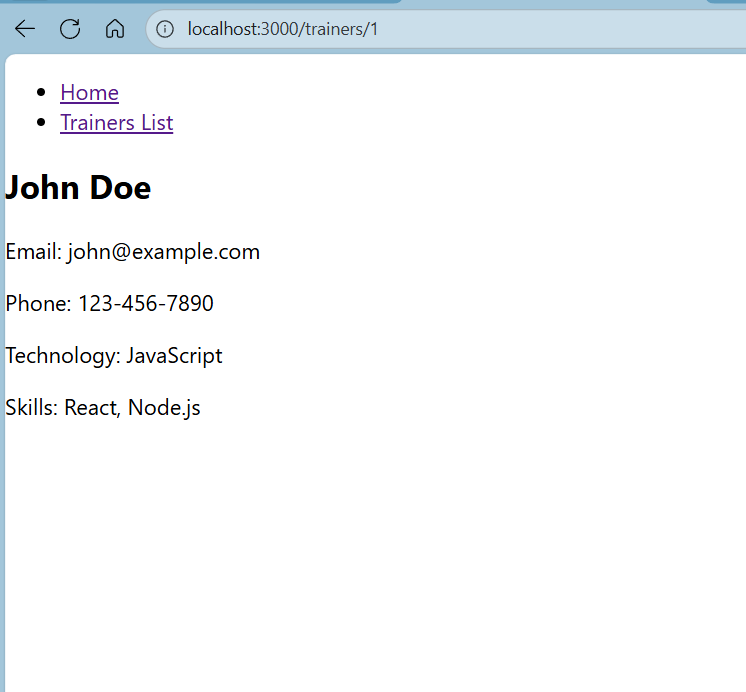
<Route path="/trainer/:id" element={<TrainerDetails />} />

In this case, :id is a **route parameter**, and the value for this parameter is accessible within the TrainerDetails component using the **useParams** hook:

const { id } = useParams();

This allows for dynamic rendering of content based on the values passed through the URL, such as fetching trainer data using the **id** parameter.

Output:



**Week 7:**

**1. Define Props**

In React, **props** (short for **properties**) are used to pass data from a **parent component** to a **child component**. They allow components to be dynamic and reusable by passing different values for each instance. **Props** are read-only, meaning the child component cannot modify them. They can be used to customize the behavior or content of a child component.

**Example:**

jsx

CopyEdit

function Welcome(props) {

return <h1>Hello, {props.name}!</h1>;

}

// Passing a prop (name)

<Welcome name="John" />

In this example, the **name** prop is passed to the Welcome component and is used to display a personalized greeting.

**2. Explain Default Props**

**Default props** are values that are automatically assigned to props if the parent component does not provide a value. This ensures that the component has a value for props, even if the parent component doesn’t explicitly pass them.

You can define default props by setting them on the component class or function.

**Example:**

function Welcome(props) {

return <h1>Hello, {props.name}!</h1>;

}

// Default prop for `name`

Welcome.defaultProps = {

name: 'Guest'

};

<Welcome /> // This will display: "Hello, Guest!"

In this example, if the **name** prop is not provided, the component will default to using "Guest".

**3. Identify the Differences Between State and Props**

* **Props**:
  + **Read-only**: Cannot be changed by the component that receives them.
  + **Passed from Parent to Child**: Used to pass data down the component tree.
  + **Used for Static Data**: Typically used for values that do not change, such as configuration settings or content.
* **State**:
  + **Mutable**: State can be updated by the component that owns it.
  + **Managed within the Component**: Used for data that can change over time, like user inputs, dynamic content, or UI states.
  + **Triggers Re-rendering**: When the state changes, React triggers a re-render of the component.

**Example of state:**

class Counter extends React.Component {

constructor() {

super();

this.state = {

count: 0

};

}

increment = () => {

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

};

render() {

return (

<div>

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

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

</div>

);

}

}

Here, **count** is part of the **state**, and **increment** updates it.

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

**reactDOM.render()** is a method used to render React components to the **DOM** (Document Object Model). It tells React where to place the component's output in the web page. This method is typically called in **index.js** to start rendering the root component of the React application.

**Syntax**:

ReactDOM.render(

<Component />, // React component

document.getElementById('root') // The DOM element to render the component

);

For example:

jsx

CopyEdit

ReactDOM.render(

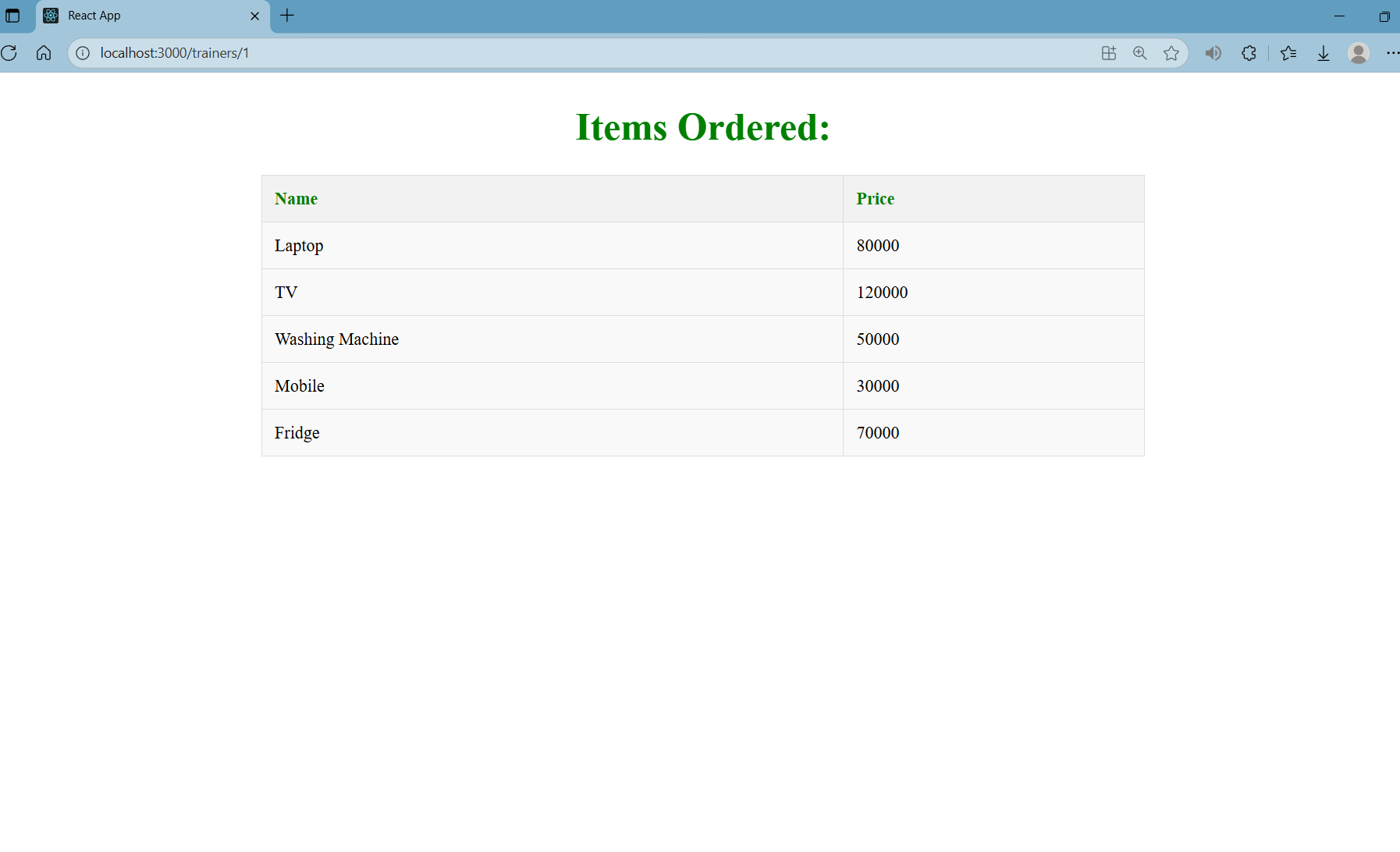
<App />,

document.getElementById('root')

);

In this case, **App** is the root component, and React will render it inside the **<div id="root"></div>** element in the **index.html** file.

Output :



**Week 8:**

React state is a special object that holds data that can change over time in a React component. This data is used to track information that affects the rendering of the component. When state changes, React re-renders the component to reflect the updated data.

State is local to the component, which means only the component that owns the state can change it. State is usually used for things like user input, form data, or anything that might change in the UI while the app is running.

For example, if you have a counter in your app, the state will store the current value of the counter. When the counter is clicked to increase, the state is updated, and the component will re-render with the new value.

****

**Week 9:**

**1. Features of ES6**

**ES6** (ECMAScript 2015) introduced several new features to JavaScript to make it more powerful and easier to use. Some of the key features are:

* **Let and Const**: Block-scoped variables.
* **Arrow Functions**: Shorter syntax for function expressions.
* **Template Literals**: Allow embedding expressions inside string literals using ${}.
* **Destructuring**: Easily unpack values from arrays or objects into variables.
* **Default Parameters**: Functions can have default parameter values.
* **Classes**: Object-oriented programming using classes and constructors.
* **Modules**: import and export to modularize code.
* **Promises**: Simplified handling of asynchronous code.

**2. JavaScript let**

**let** is a variable declaration keyword introduced in ES6. Unlike **var**, **let** is **block-scoped**, meaning it is only accessible within the block (enclosed by curly braces) where it is declared.

**Example:**

let name = "John";

if (true) {

let name = "Jane"; // This is a different variable scoped to the if block

console.log(name); // Output: Jane

}

console.log(name); // Output: John

**3. Differences Between var and let**

* **Scope**:
  + **var** is function-scoped, meaning it is accessible within the function where it is declared.
  + **let** is block-scoped, meaning it is accessible only within the block (like inside loops or conditionals) where it is declared.
* **Hoisting**:
  + **var** is hoisted to the top of its scope and initialized with undefined.
  + **let** is hoisted but is not initialized, leading to a **ReferenceError** if accessed before its declaration.
* **Re-declaration**:
  + **var** allows re-declaration within the same scope.
  + **let** does not allow re-declaration within the same scope.

**4. JavaScript const**

**const** is used to declare variables whose value should not be reassigned after initialization. It is also **block-scoped** like let, but its value is immutable after assignment.

**Example:**

const PI = 3.14;

PI = 3.1415; .

However, note that objects or arrays declared with const can still have their properties or elements modified.

**5. ES6 Class Fundamentals**

ES6 introduced **classes**, a more structured way to create objects and manage inheritance, compared to **prototypes** in earlier JavaScript versions.

**Example:**

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

greet() {

console.log(`Hello, my name is ${this.name}.`);

}

}

const john = new Person("John", 30);

john.greet();

**6. ES6 Class Inheritance**

ES6 also introduced **class inheritance**, allowing one class to inherit properties and methods from another using the **extends** keyword.

**Example:**

class Employee extends Person {

constructor(name, age, jobTitle) {

super(name, age);

this.jobTitle = jobTitle;

}

introduce() {

console.log(`I am ${this.name} and I work as a ${this.jobTitle}.`);

}

}

const jane = new Employee("Jane", 28, "Engineer");

jane.introduce(); .

**7. ES6 Arrow Functions**

Arrow functions provide a shorter syntax for writing functions and automatically bind the value of this to the surrounding context.

**Example:**

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

console.log(add(2, 3));

* **No binding of this**: In traditional functions, this refers to the function's own context, but in arrow functions, this refers to the surrounding context, making it more predictable.

**8. Set() and Map()**

* **Set()**: A collection of unique values. It stores values without duplicates.

**Example:**

const mySet = new Set();

mySet.add(1);

mySet.add(2);

mySet.add(1);

console.log(mySet);

* **Map()**: A collection of key-value pairs. Unlike objects, keys in a **Map** can be of any data type.

**Example:**

const myMap = new Map();

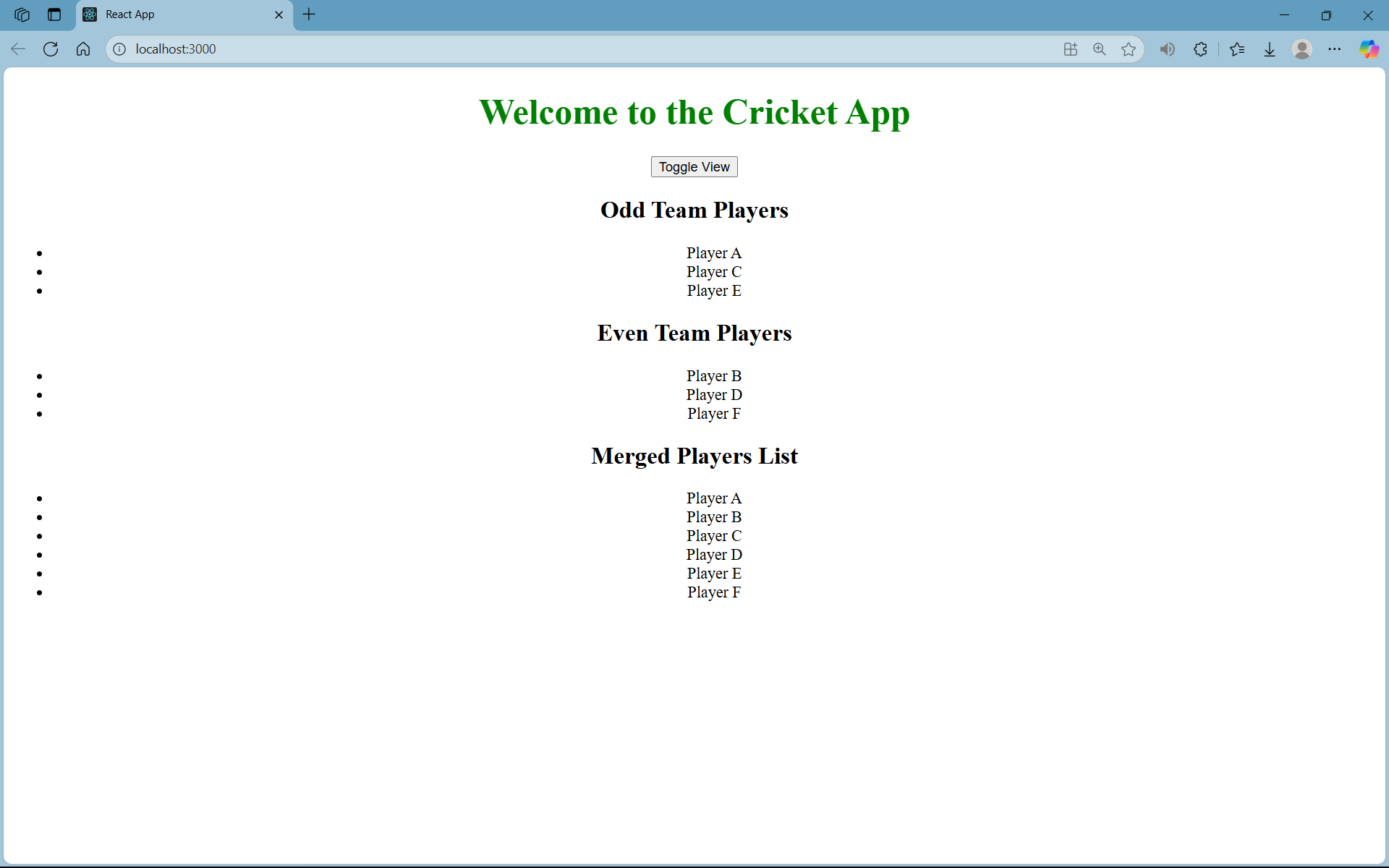
myMap.set('name', 'Alice');

myMap.set(1, 'One');

console.log(myMap.get('name'));

console.log(myMap.get(1));

Output:



**Week 10:**

**1. Define JSX**

**JSX** (JavaScript XML) is a syntax extension for JavaScript used in **React**. It allows you to write HTML-like code within JavaScript, which React can convert into regular JavaScript objects. JSX makes it easier to create and manage UI elements by combining HTML and JavaScript in one place.

Example:

const element = <h1>Hello, World!</h1>;

JSX simplifies the process of defining UI components, but under the hood, it's compiled into React.createElement() calls.

**2. Explain About ECMA Script**

**ECMAScript** is the standard that defines the scripting language used in JavaScript. It is maintained by **ECMA International** and sets guidelines for how JavaScript should behave, including features and syntax.

The most popular version is **ECMAScript 6** (ES6), which introduced many new features to JavaScript such as:

* **Let and const** for block-scoped variables.
* **Arrow functions** for shorter function syntax.
* **Classes** for object-oriented programming.
* **Modules** with import and export.

**3. Explain React.createElement()**

React.createElement() is a low-level API used by React to create React elements. When you write JSX, React compiles it into React.createElement() calls. This function takes three arguments:

1. **Type** of the element (like 'div' or 'h1').
2. **Props**: An object of properties for the element (like className or style).
3. **Children**: Any child elements or content inside the element.

For example:

const element = React.createElement('h1', { className: 'header' }, 'Hello, World!');

This would create an <h1> tag with the text "Hello, World!" and the class header.

**4. Explain How to Create React Nodes with JSX**

In React, a **node** represents a component or element in the UI. You can create React nodes using JSX, which gets transpiled into React elements.

Example of creating a node with JSX:

const node = <h1>Hello, World!</h1>;

Here, <h1> is a JSX element, and React will convert it into a node that can be rendered to the DOM.

**5. Define How to Render JSX to DOM**

To render JSX to the DOM, you use **ReactDOM.render()** (or **ReactDOM.createRoot()** in React 18). This method takes two arguments:

1. The **React element** (JSX).
2. The **DOM element** where the React element should be inserted.

Example:

ReactDOM.render(<h1>Hello, World!</h1>, document.getElementById('root'));

This will render the JSX <h1> tag inside the HTML element with the ID root.

**6. Explain How to Use JavaScript Expressions in JSX**

You can embed **JavaScript expressions** inside JSX by wrapping them in curly braces {}. These expressions can be variables, functions, or any valid JavaScript code that returns a value.

Example:

const name = "John";

const element = <h1>Hello, {name}!</h1>;

In this example, the value of name is inserted into the JSX. You can also use expressions like math operations:

const sum = 5 + 3;

const element = <p>{sum}</p>;

**7. Explain How to Use Inline CSS in JSX**

In JSX, you can use **inline styles** by passing a JavaScript object to the **style** attribute. The keys in the object are written in camelCase, not kebab-case, and the values are usually strings.

Example:

const divStyle = {

color: 'blue',

backgroundColor: 'lightgrey'

};

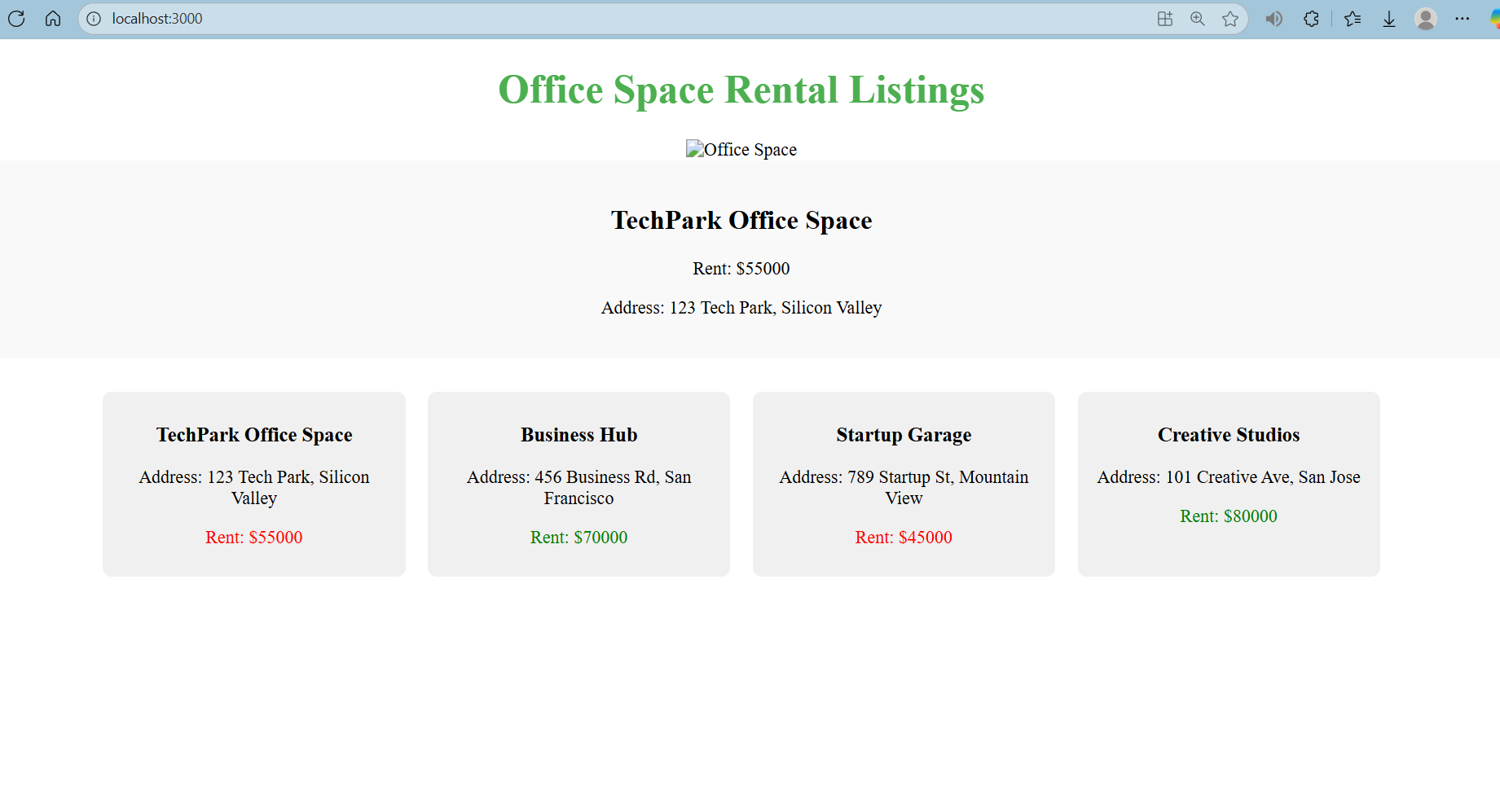
const element = <div style={divStyle}>Hello, World!</div>;

In this example, the div will have blue text and a light grey background.

**Conclusion**

* **JSX** allows you to write HTML-like code in JavaScript to create React elements.
* **ECMAScript** is the standard specification that defines how JavaScript works, with ES6 introducing many powerful features.
* **React.createElement()** is used to create React elements, which JSX is transpiled into.
* **JSX** makes it easier to work with React nodes and render them to the DOM.
* JavaScript expressions can be used inside JSX with curly braces {}.
* **Inline CSS** can be added in JSX by passing a JavaScript object to the style attribute.

Output:



**Week 11:**

**1. React Events**

React events are the way to handle user interactions such as clicks, key presses, or form submissions in React applications. They are synthetically handled by React, which normalizes the events across different browsers. React’s event system is based on the W3C specification for DOM events.

React events are attached to elements using camelCase syntax, and they trigger JavaScript functions when an action occurs (e.g., button click, form submission).

Example:

<button onClick={handleClick}>Click Me</button>

**2. Event Handlers**

Event handlers are functions that handle specific user actions or events. They are usually defined as methods or functions in your components. In React, event handlers are assigned to JSX elements as props, and the event handler function is called when the event occurs.

* In **functional components**, event handlers are typically defined as functions.
* In **class components**, event handlers are often defined as methods.

Example in a class component:

class MyComponent extends React.Component {

handleClick() {

console.log('Button clicked!');

}

render() {

return <button onClick={this.handleClick}>Click Me</button>;

}

}

Example in a functional component:

const MyComponent = () => {

const handleClick = () => {

console.log('Button clicked!');

};

return <button onClick={handleClick}>Click Me</button>;

};

**3. Synthetic Event**

A **Synthetic Event** is React's cross-browser wrapper around the browser's native event system. React creates a synthetic event for each native event (e.g., click, keydown, submit). This ensures that events behave the same way across all browsers.

Synthetic events in React normalize event properties like target, preventDefault(), and stopPropagation(), making them consistent and easier to manage.

Example:

const handleClick = (event) => {

console.log(event.target); // Synthetic event, has the same API across browsers

};

**4. React Event Naming Convention**

React event names follow **camelCase** naming conventions instead of the typical lowercase used in HTML. This means that event names are written in camelCase style, where the first word is lowercase, and each subsequent word starts with an uppercase letter.

For example:

* **onClick** instead of onclick
* **onChange** instead of onchange
* **onSubmit** instead of onsubmit

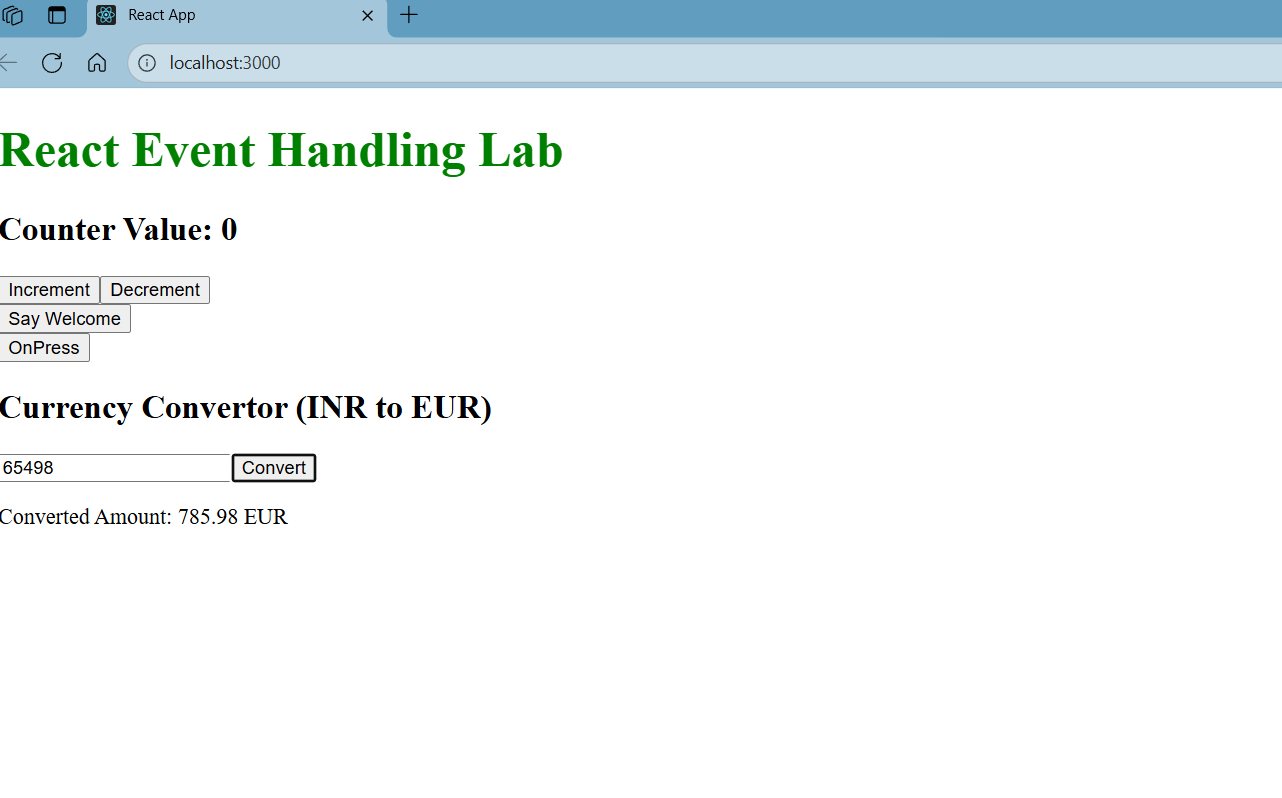
This consistency helps React to distinguish its events from other attributes in JSX.

**Conclusion**

* **React events** are user interactions managed using React’s synthetic event system.
* **Event handlers** are functions triggered by events that control what happens when the event occurs.
* **Synthetic events** ensure that React handles events consistently across different browsers.
* **React event naming convention** uses **camelCase** to name event handlers, making them distinct from HTML attributes.

These concepts are fundamental to handling user interactions and event-driven behaviors in React applications.

**Output:**

****

**Week 12:**

**1. Conditional Rendering in React**

In React, **conditional rendering** is like telling your app to show different things based on certain conditions. It's similar to using **if/else** statements in JavaScript. For example, if a user is logged in, you might show them a dashboard, but if they’re not logged in, you might show them a login page.

You can use:

* **Ternary operators** to make decisions right inside your JSX.
* **Logical AND (&&)** to show something only when a condition is true.
* **if/else** in functions or methods to decide what to render.

For example, using a **ternary operator**:

{isLoggedIn ? <UserPage /> : <GuestPage />}

Here, React will show the **UserPage** if the user is logged in, or the **GuestPage** if not. It’s a simple way to dynamically change what’s on the screen.

**2. Element Variables**

In React, **element variables** are just variables that hold JSX code, which is the structure of your page. These elements can be returned or displayed just like regular HTML, but they’re inside JavaScript. It’s a way to store a UI component that you can use elsewhere in your app.

Here’s an example:

const element = <h1>Hello, World!</h1>;

You can store JSX like this in a variable, and then use it wherever you need it. For instance, in a functional component, you might do something like this:

function App() {

const greeting = <h1>Welcome to the app!</h1>;

return <div>{greeting}</div>;

}

So, **element variables** let you save and reuse your JSX in a neat and organized way.

**3. How to Prevent Components from Rendering**

Sometimes, you don’t want a component to show up on the screen. In React, you can easily prevent rendering by checking a condition. If the condition isn’t met, you just **return null** to stop the component from rendering.

For example, if a user isn’t logged in, you might want to prevent the rendering of sensitive content:

if (!isLoggedIn) {

return null; // This skips rendering if not logged in

}

This is useful if you need to stop a component from rendering anything at all.

Or, in some cases, you might use **conditional rendering** to decide whether a component should display based on a state or a prop.

Example:

function SecretPage({ isLoggedIn }) {

if (!isLoggedIn) {

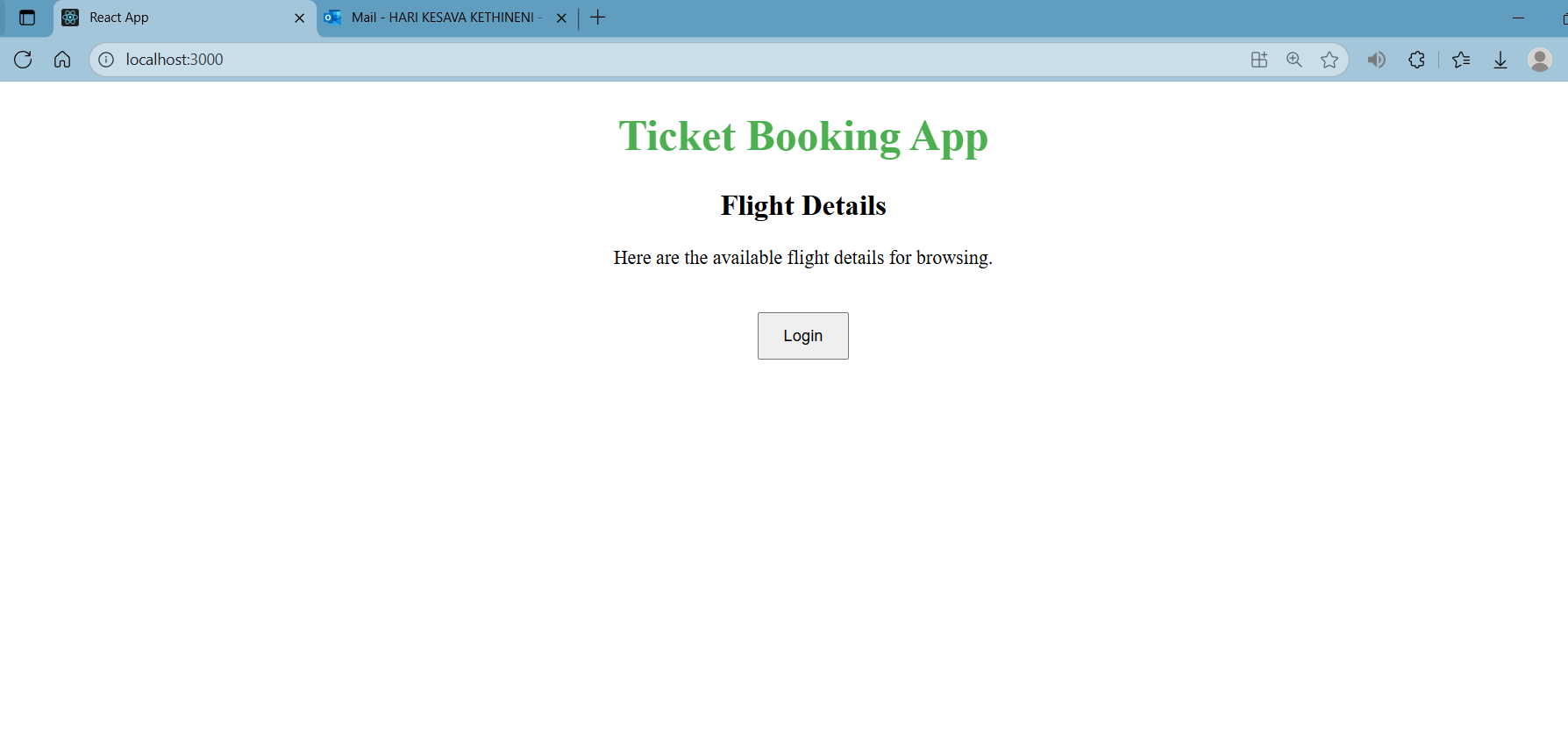
return null; // Don't render anything if not logged in

}

return <h1>Secret Content</h1>;

}

Here, **return null** just means don’t render anything, and that can be helpful if you want to hide parts of your app based on the user’s status.

****

**Week 13:**

**1. Various Ways of Conditional Rendering in React**

In React, **conditional rendering** allows you to display different content based on certain conditions. React doesn't have a special syntax for conditionals; instead, you use regular JavaScript expressions to decide what to display.

Here are a few common ways to handle **conditional rendering** in React:

* **Ternary Operator (? :)**: It’s a quick way to choose between two options. You can use it to render one component or another based on a condition.
* {isLoggedIn ? <UserProfile /> : <Login />}

This will render **UserProfile** if the user is logged in, and **Login** if they are not.

* **Logical AND (&&)**: This is useful when you only want to render something when a condition is true, and nothing when it’s false.
* {isLoggedIn && <UserProfile />}

This will render **UserProfile** only if the user is logged in. If not, it renders nothing.

* **if/else**: In more complex cases, you may need to use traditional if/else statements, especially in event handlers or functions.
* if (isLoggedIn) {
* return <UserProfile />;
* } else {
* return <Login />;
* }

Each of these methods helps React determine what to render based on the conditions you define.

**2. Rendering Multiple Components in React**

In React, you can render **multiple components** at once, either inside each other or within the same parent component. You can simply include several components within the return statement of another component.

For example:

function App() {

return (

<div>

<Header />

<MainContent />

<Footer />

</div>

);

}

Here, **Header**, **MainContent**, and **Footer** are all rendered inside a **div**. This is how you can render several components together.

**3. Defining List Components in React**

A **list component** in React is a component that displays an array or list of items. You typically use the .map() function to iterate over an array of data and render each item in the list.

Here’s an example of a list component:

function ItemList() {

const items = ['Apple', 'Banana', 'Orange'];

return (

<ul>

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

<li key={index}>{item}</li>

))}

</ul>

);

}

In this example, **ItemList** renders a list of fruits. The .map() function goes through each item in the **items** array and renders it inside a **<li>** element.

**4. Keys in React Applications**

In React, **keys** help React identify which items in the list are changed, added, or removed. When rendering lists, **key** is a special string attribute that you need to add to each element in a list.

Without a **key**, React can’t optimize the updates, potentially leading to inefficient re-renders.

Here’s an example where **key** helps React identify each list item uniquely:

const fruits = ['Apple', 'Banana', 'Orange'];

const fruitList = fruits.map((fruit, index) => (

<li key={index}>{fruit}</li>

));

The **key** prop should be a **unique identifier** for each item, ideally something like an ID from the data, but using the index can also work if no unique ID is available.

**5. Extracting Components with Keys**

When you need to extract or split a list of items into separate components, each component should have a **key** to ensure React can track changes correctly. You can create a separate component and pass the list item data as props.

Example:

function Fruit({ name }) {

return <li>{name}</li>;

}

function FruitList() {

const fruits = ['Apple', 'Banana', 'Orange'];

return (

<ul>

{fruits.map((fruit, index) => (

<Fruit key={index} name={fruit} />

))}

</ul>

);

}

In this example, **Fruit** is a child component that displays each fruit. The **key** prop is passed from the **FruitList** component to each **Fruit** component.

**6. React Map and the map() Function**

In React, the **map()** function is used to iterate over an array and return a new array of elements. It’s often used for rendering lists in JSX.

The **map()** function works like this:

const numbers = [1, 2, 3, 4, 5];

const doubled = numbers.map(num => num \* 2);

console.log(doubled); // Output: [2, 4, 6, 8, 10]

When rendering a list of items in React, you usually combine **map()** with **key** to render each element with a unique identifier:

function NumberList() {

const numbers = [1, 2, 3, 4, 5];

return (

<ul>

{numbers.map((number, index) => (

<li key={index}>{number}</li>

))}

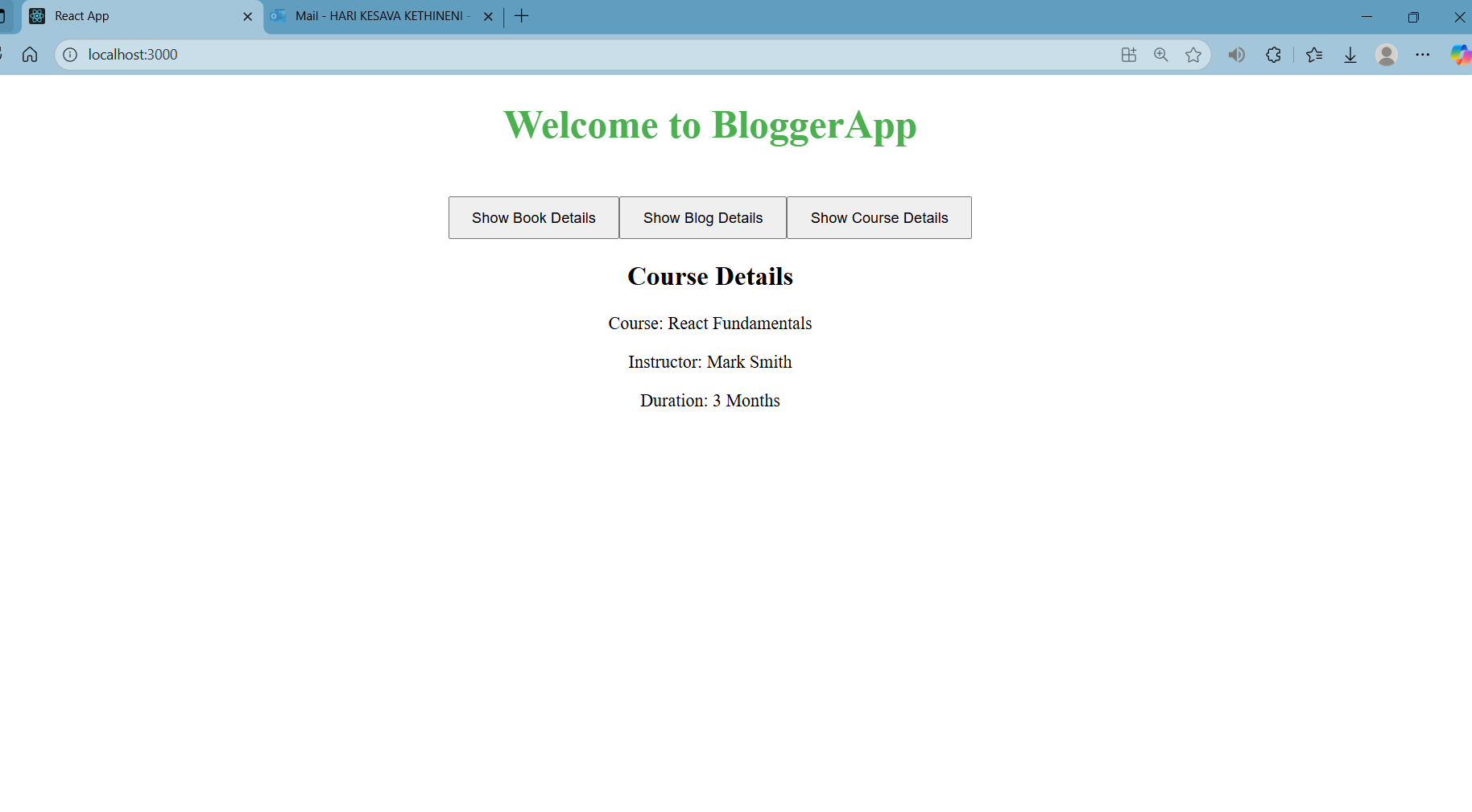
</ul>

);

}

In this example, **map()** loops through the **numbers** array, and each item is rendered inside a <li> element. The **key** is used to help React track each list item efficiently.

**Output:**



**Week 14:**

**1. React Context API: Need and Benefits**

The **React Context API** helps manage state across your app without passing props through every component. It solves the problem of **prop drilling**, where data is passed down through multiple layers of components.

**Benefits**:

* Simplifies state management.
* Avoids prop drilling.
* Makes global data (like themes, authentication) accessible to any component.
* Reduces boilerplate code and improves maintainability.

**2. Working with createContext()**

createContext() creates a **Context object** to share data across components. It provides a **Provider** to pass the data and a **Consumer** to access it.

Example:

const ThemeContext = createContext('light');

<ThemeContext.Provider value={theme}>

<ChildComponent />

</ThemeContext.Provider>

**3. Types of Router Components**

* **<BrowserRouter>**: Handles URL navigation using the HTML5 History API (most common router).
* **<Route>**: Defines a path and the component to render when the path matches.
* **<Link>**: A clickable element that updates the URL without reloading the page.
* **<Switch>**: Renders only the first matching route from a list of routes.

Example:

<BrowserRouter>

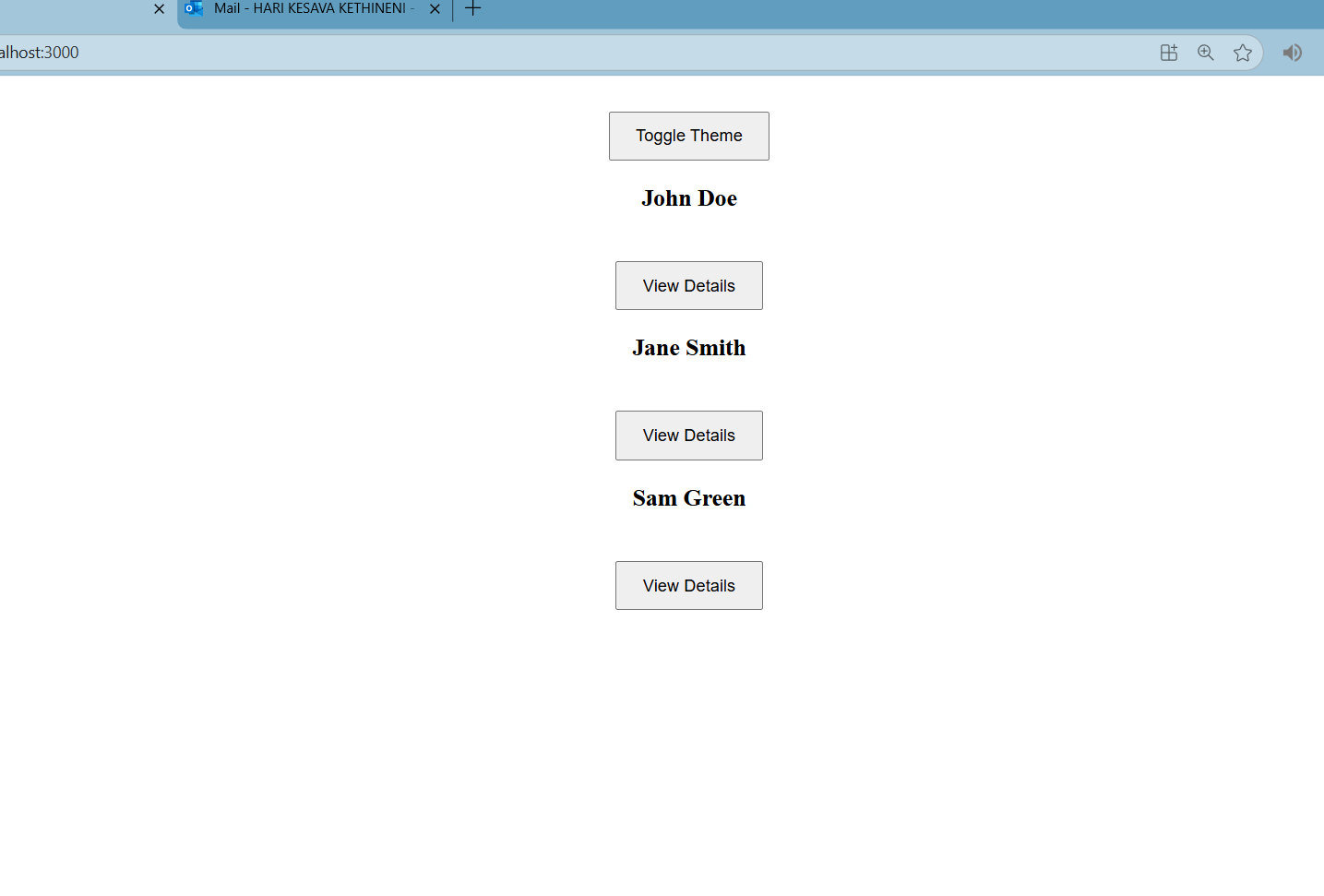
<Switch>

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

</Switch>

</BrowserRouter>

**Output:**



**Week 15:**

**1. React Forms**

**React forms** allow users to input data, which is then handled by the application. In React, form elements like **input fields, buttons, text areas,** etc., are controlled by the component state. This provides full control over the form and ensures React is always aware of the form's state.

**2. Controlled Components**

A **controlled component** is a form element where React manages its value through the component’s **state**. The state drives the form element's behavior, making React the "source of truth" for input values.

For example, in a controlled input:

const [name, setName] = useState('');

const handleChange = (e) => {

setName(e.target.value);

};

return <input type="text" value={name} onChange={handleChange} />;

Here, **name** is the state variable that controls the **input field**, and **setName** updates it based on user input.

**3. Various Input Controls**

React provides several types of **input controls** to handle different kinds of user inputs:

* **Text input**: For simple text entry.
* **Textarea**: For multiline text input (e.g., complaints or comments).
* **Checkbox**: For true/false or yes/no options.
* **Radio buttons**: For selecting one option from a group.
* **Select**: For dropdown menus.

These input controls are used with **state** to manage their values, making it easier to handle form data.

**4. Handling Forms**

Handling forms in React involves managing the **state** of input fields and reacting to events like **onChange** or **onSubmit**. React provides an efficient way to handle complex forms by binding each input element’s value to a piece of state.

We can use **controlled components** to manage form inputs:

function MyForm() {

const [value, setValue] = useState('');

const handleChange = (e) => setValue(e.target.value);

return (

<form>

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

</form>

);

}

**5. Submitting Forms**

Form submission is handled by using the **onSubmit** event. React provides a way to handle form submissions without reloading the page by preventing the default form behavior using **event.preventDefault()**.

const handleSubmit = (event) => {

event.preventDefault();

alert('Form submitted');

};

return (

<form onSubmit={handleSubmit}>

<input type="text" />

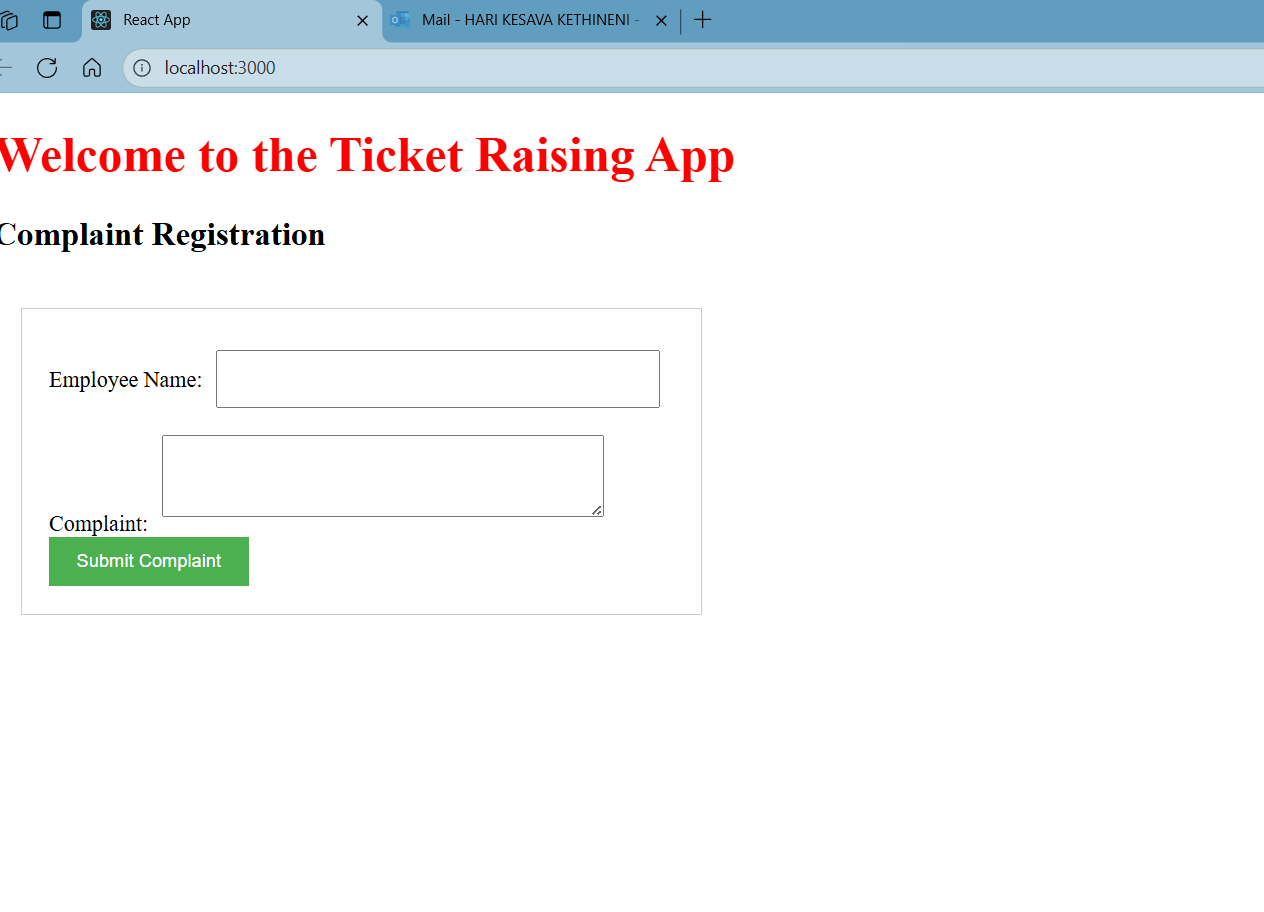
<button type="submit">Submit</button>

</form>

);

In this example, the **handleSubmit** function handles the form submission without the page reload.

**Output:**



**Week 16:**

**1. React Forms Validation**

**React form validation** ensures that the data entered by a user in a form meets certain criteria before it's submitted. It’s essential for improving user experience and data accuracy. You can validate forms by checking conditions like:

* Field length (e.g., name must be at least 5 characters).
* Proper formats (e.g., email should contain "@" and ".").
* Password strength (e.g., at least 8 characters).

React handles form validation with **state** and **event handlers** (e.g., onSubmit), providing immediate feedback if a user enters invalid data.

**2. Differences Between React Form and HTML Form**

* **HTML Forms**: In a standard HTML form, the input data is handled by the browser and can trigger page reloads on form submission. Input elements are controlled by the browser’s native behavior.
* **React Forms**: In React, forms are **controlled** by the **component state**. React handles form inputs directly via state, which means you have full control over the data, and there's no page reload when submitting. React forms use **controlled components** to manage form data.

**3. Controlled Components**

A **controlled component** in React refers to form elements whose values are controlled by the component’s **state**. Instead of relying on the browser's default behavior, React manages the value of the input field through state, providing full control over the form data.

Example:

const [name, setName] = useState('');

const handleChange = (e) => setName(e.target.value);

Here, the name value is managed by React’s state.

**4. Various React Form Input Controls**

React supports a variety of input controls that are used to collect data from users:

* **Text Input** (<input type="text" />): For simple text entries.
* **Textarea** (<textarea />): For multiline text inputs (e.g., feedback).
* **Checkbox** (<input type="checkbox" />): For binary options (yes/no).
* **Radio Buttons** (<input type="radio" />): For selecting one option from a group.
* **Select** (<select />): For dropdown selections.

These input controls work in conjunction with **state** to store and manage user inputs.

**5. Handling React Forms**

Handling forms in React involves:

* Using **state** to manage input values.
* Setting up **event handlers** like onChange for form inputs to update state.
* Using the **onSubmit** event to handle form submissions.

React gives you full control over the form behavior, enabling validation, conditional rendering, and dynamic form adjustments based on the data.

**6. Submitting Forms in React**

In React, submitting a form typically involves:

* Handling the **onSubmit** event, preventing the default form submission behavior using **event.preventDefault()**.
* Validating the form data before submission.
* Optionally, sending the data to an API or server.

For example:

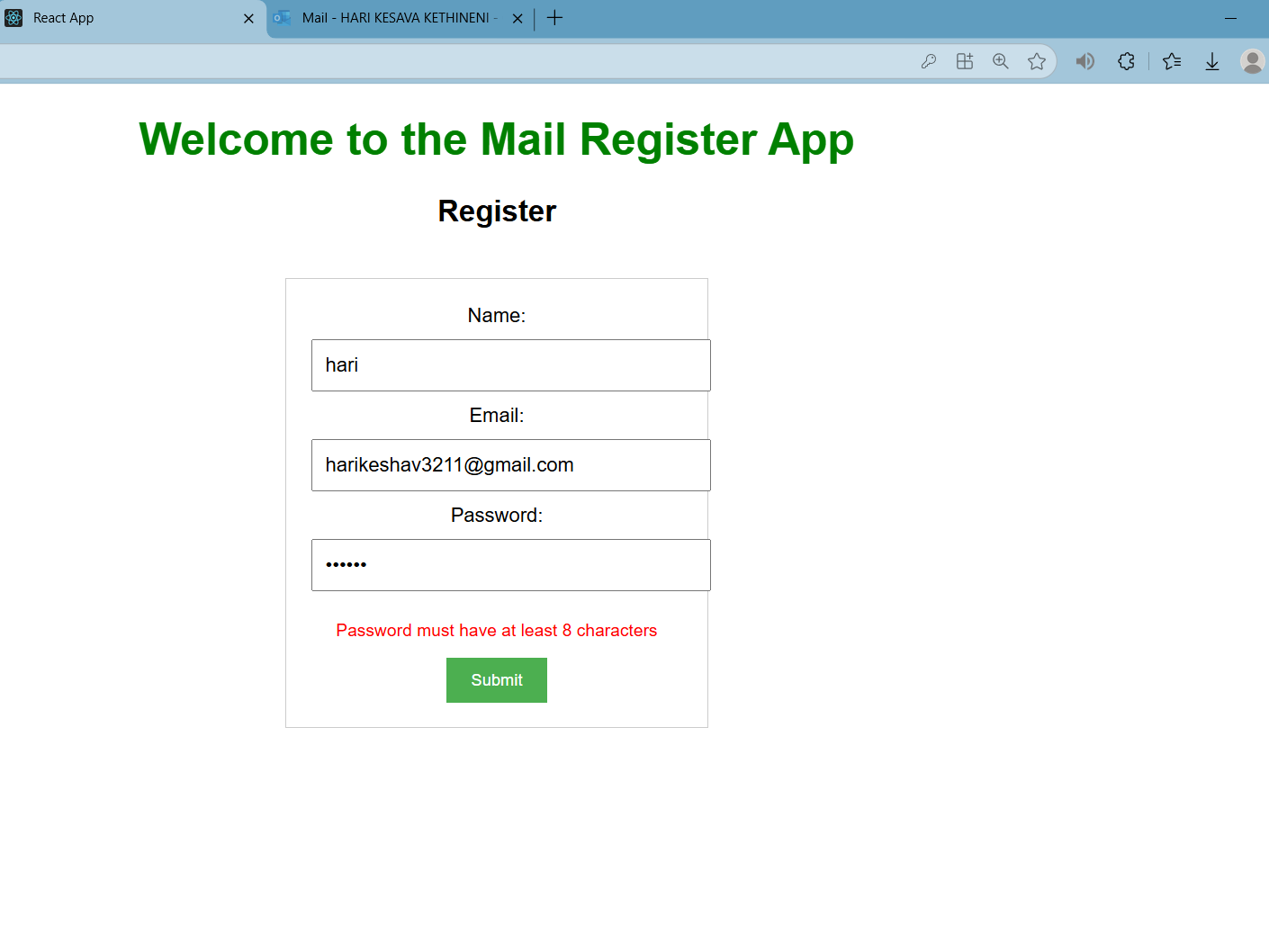
const handleSubmit = (e) => {

e.preventDefault();

};

React does not reload the page on form submission; instead, it handles the process without a full page refresh, improving the user experience.

**Ouput:**

****

**Week 17:**

Consuming **REST APIs** in React applications involves making HTTP requests to fetch or send data to external servers. This is typically done using **fetch()**, **axios**, or similar libraries. Here's how it works:

1. **Make a request**: You initiate a request to a REST API using methods like GET (to fetch data) or POST (to send data).
2. **Handle the response**: The server responds with data, usually in **JSON** format. You handle this response and update your app’s **state**.
3. **Display the data**: Once the data is fetched, you can display it on your app, update the UI, or trigger other actions.

**Example using fetch()**:

fetch('https://api.example.com/data')

.then(response => response.json()) // Parse the response

.then(data => {

// Use the data in your React component state

this.setState({ data });

})

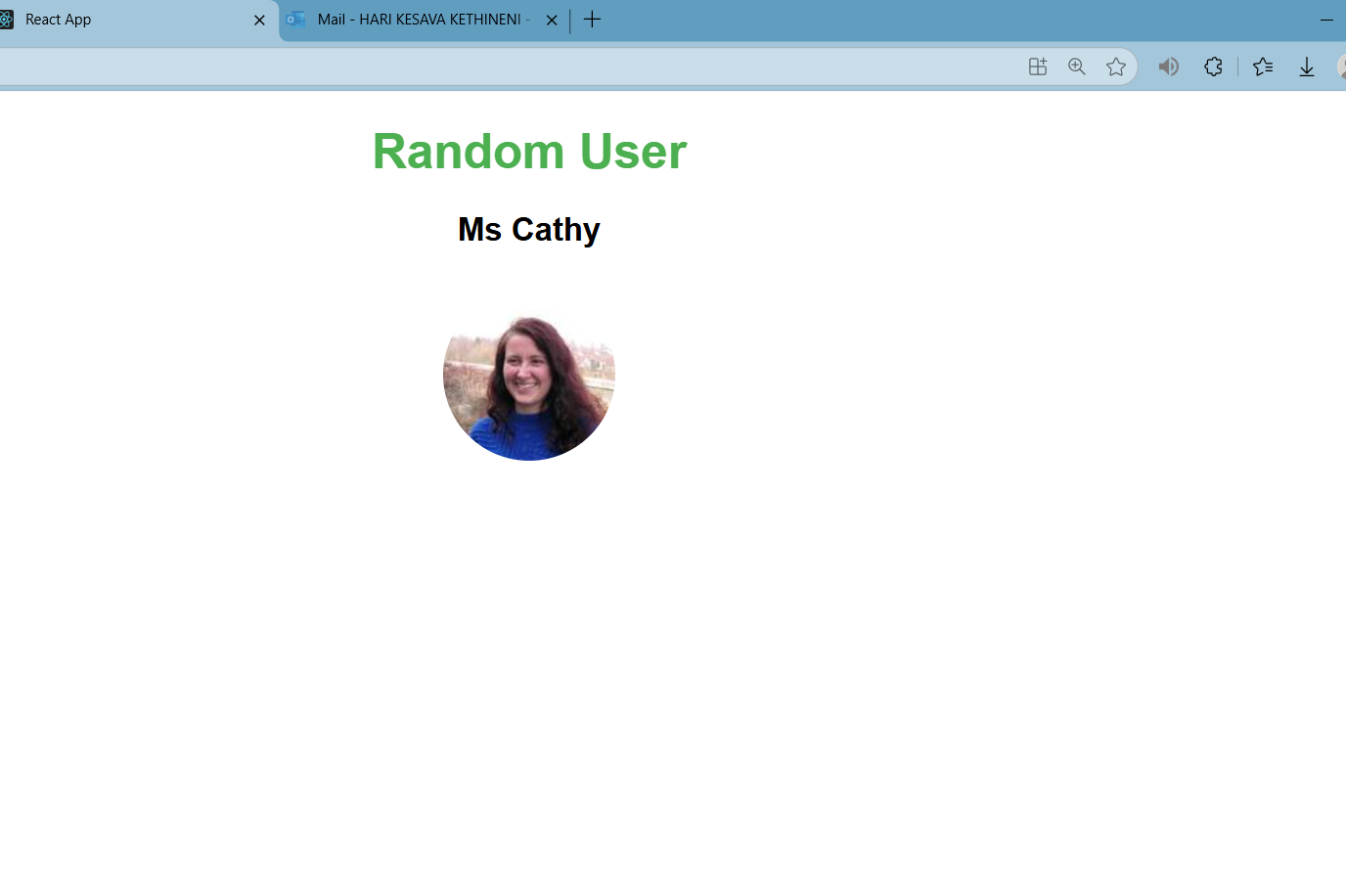
.catch(error => {

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

});

In React, you usually make API calls in **lifecycle methods** (like componentDidMount() in class components) or **hooks** (like useEffect() in functional components). This allows you to fetch data when the component mounts and update the state accordingly.

**Ouput:**

****