**SUPERSET ID - 6364309**

**HANDSON DOC1**

**Theories:**

## **Single-Page Application (SPA)**

**Definition**  
A Single-Page Application (SPA) is a web application that loads a single HTML page and dynamically updates content as the user interacts with the app, without requiring a full page reload. All necessary resources (HTML, CSS, JavaScript) are either fetched once at startup or loaded on demand, and routing within the app is handled client‑side.

**Benefits: Seamless User Experience:** Transitions between “pages” feel instantaneous since only parts of the page update.

**Reduced Server Load:** After the initial load, only data (usually via APIs) is requested from the server, not full HTML pages.

**Easier State Management:** Client‑side frameworks can maintain application state (e.g., user authentication, UI state) more consistently.

**Reusability of Components:** UI components can be built once and reused across different views.

**Rich Interactivity:** Enables highly interactive, desktop‑like experiences in the browser.

## **Multiple-Page Application (MPA) vs. SPA**

| **Feature** | **SPA** | **MPA** |
| --- | --- | --- |
| **Page Reloads** | **No full reload; content swaps dynamically** | **Full page reload on each navigation** |
| **Routing** | **Client-side routing (e.g., via History API)** | **Server-side routing; each URL corresponds to a server endpoint** |
| **Speed (after load)** | **Very fast interactions post-initial load** | **Slower; each navigation fetches entire page** |
| **SEO** | **Requires extra setup (e.g., server-side rendering)** | **Works out of the box with server-rendered pages** |
| **Initial Load Time** | **Longer (downloads JS framework & bundle)** | **Shorter; only required assets for that page** |
| **Development Model** | **Often decoupled frontend/backend** | **Tighter coupling; backend renders views/templates** |

## **Pros & Cons of Single‑Page Applications**

**Pros: Smooth UX:** No flicker or blank pages during navigation.

**Efficient Data Usage:** Only data is transferred; avoids redundant HTML/CSS.

**Modular Frontend Architecture:** Encourages component‑based design.

**Offline Support & Caching:** Easier to implement service workers for offline mode.

**Cons : SEO Challenges:** Search engines may struggle without server‑side rendering or prerendering.

**Initial Bundle Size:** Can be large, leading to slower first‑load times.

**JavaScript Dependency:** If JS fails or is disabled, the app may not function.

**Memory Leaks:** Long‑running single page may accumulate unused resources.

## **React**

**Definition**  
React is an open‑source JavaScript library for building user interfaces, maintained by Facebook. It focuses on the view layer (UI) of an application, enabling developers to create reusable, declarative components.

**How React Works**

**Component-Based Architecture:** UI is broken into self‑contained components (functions or classes) that manage their own state and render UI.

**Unidirectional Data Flow:** Data flows down from parent to child components via props, making state changes predictable.

**Reconciliation:** When state or props change, React computes the minimal set of updates needed to sync the UI with the data model.

**Virtual DOM Diffing:** React maintains an in‑memory Virtual DOM representation; it diffs this against the previous snapshot to batch and apply only necessary real DOM updates.

## **Virtual DOM**

**Definition**  
The Virtual DOM is a lightweight, in‑memory representation of the actual DOM. Rather than manipulating the real DOM directly (which is slow), React first updates the Virtual DOM, computes the differences (“diff”), and then efficiently applies the minimal changes to the real DOM.

## **Key Features of React**

**JSX (JavaScript XML):** A syntax extension that allows writing HTML‑like code within JavaScript, improving readability and component structure.

**Reusable Components:** Encapsulate markup, styles, and behavior to promote DRY (Don’t Repeat Yourself) principles.

**Lifecycle Methods / Hooks:**

**Class Components:** componentDidMount, shouldComponentUpdate, etc.

**Functional Components:** Hooks like useState, useEffect, useContext, enabling state and side‑effects.

**Context API:** Allows sharing global data (e.g., theme, locale) without prop drilling.

**High Performance:** Virtual DOM diffing, fiber architecture, and selective update batching.

**Rich Ecosystem:** Tools like React Router for routing, Redux or Context for state management, and a vast library of third‑party components.

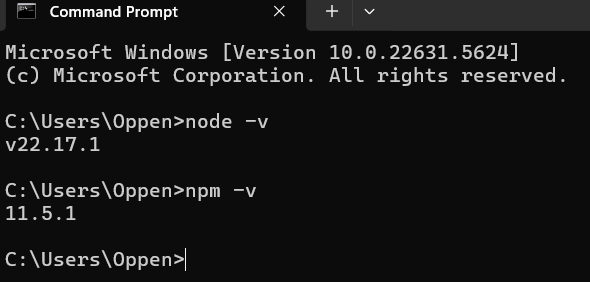
**Server-Side Rendering (SSR) & Hydration:** Frameworks like Next.js enable pre‑rendering pages on the server for SEO and faster initial loads.

**Work Done**

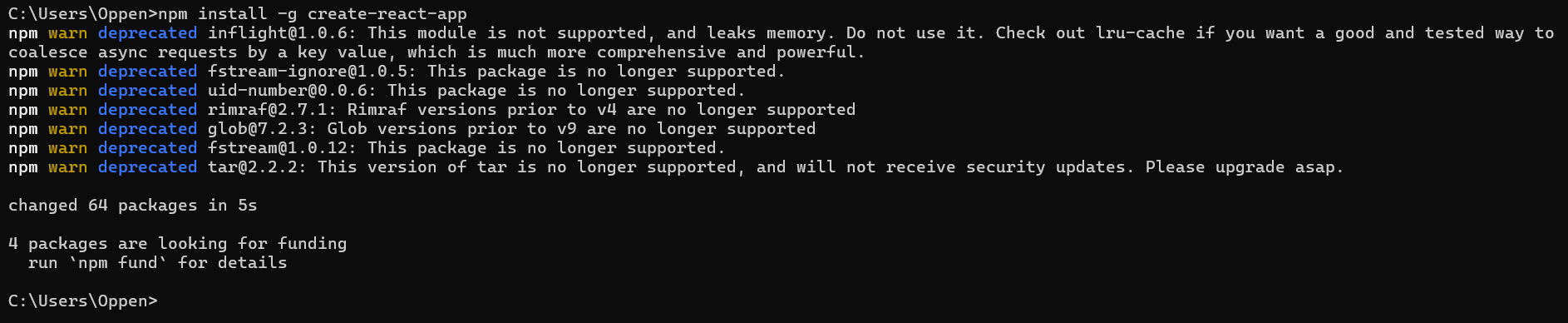
Create a new React Application with the name “myfirstreact”, Run the application to print “welcome to the first session of React” as heading of that page.

1. Nodejs and Npm was installed from the following link:

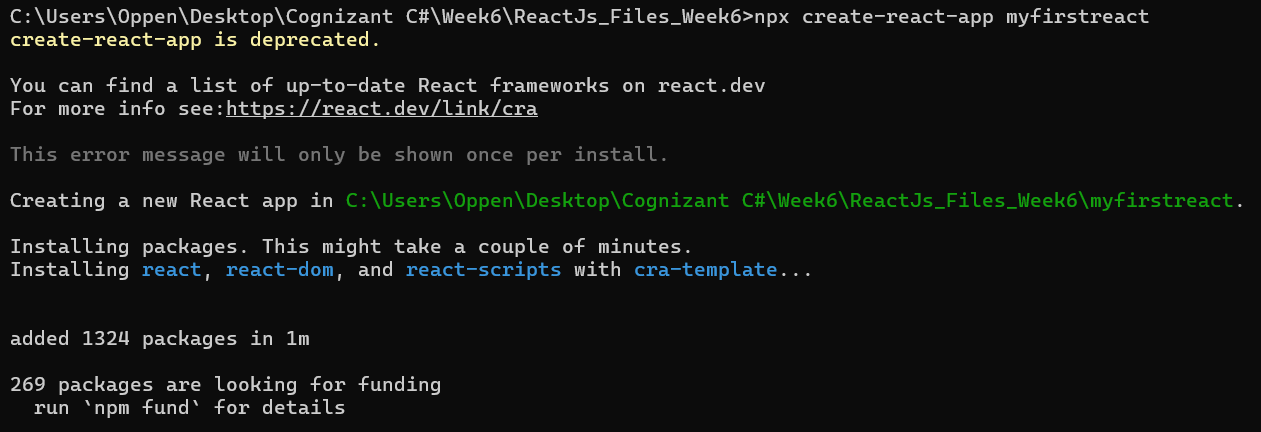
<https://nodejs.org/en/download/>



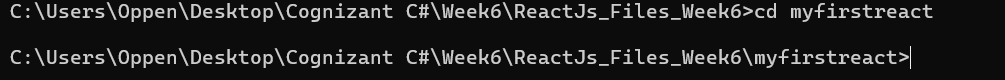
1. Install Create-react-app by running the following command in the command prompt:



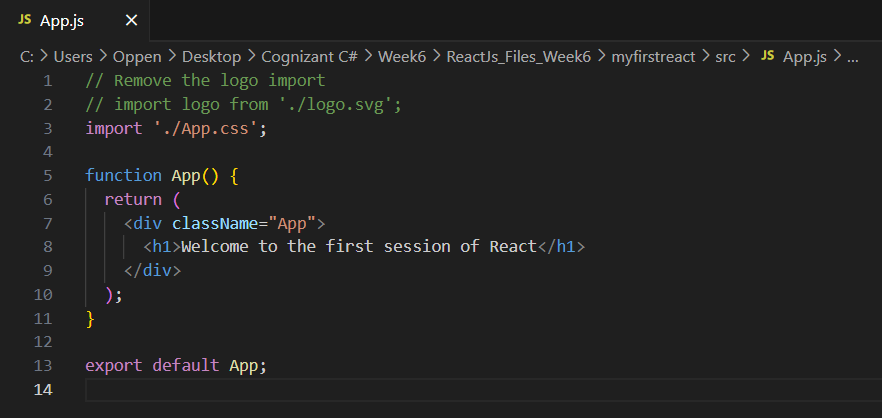
1. To create a React Application with the name of “myfirstreact”, type the following command:



1. Once the App is created, navigate into the folder of myfirstreact by typing the following command:



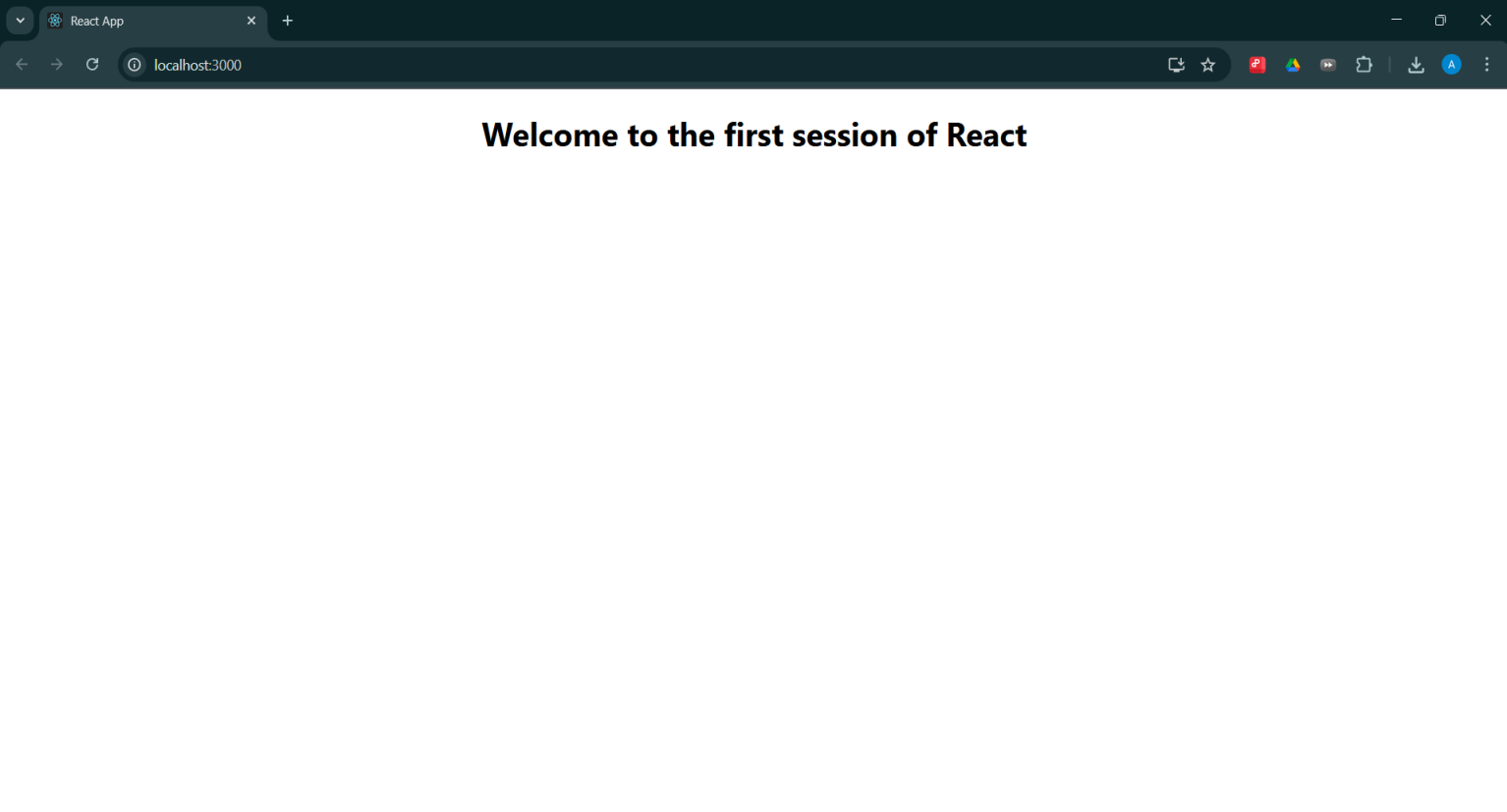
1. Open the folder of myfirstreact in Visual Studio Code
2. Open the App.js file in Src Folder of myfirstreact
3. Remove the current content of “App.js”
4. Replace it with the following:



1. Run the following command to execute the React application:



1. Open a new browser window and type “localhost:3000” in the address bar



**HANDSON DOC 2**

**1. React Components**

**Definition:** React components are self‑contained, reusable building blocks that encapsulate markup, logic, and styling for a part of the user interface.

**Key Characteristics:**

**Encapsulation:** Each component manages its own internal data (state) and exposes configuration via properties (props).

**Reusability:** Components can be composed and reused throughout an application.

**Isolation:** Changes in one component’s state or logic do not directly affect others.

**2. Components vs. Plain JavaScript Functions**

| **Aspect** | **JavaScript Function** | **React Component** |
| --- | --- | --- |
| ****Invocation**** | **Called explicitly in code** | **Rendered by React via JSX tags** |
| ****Return Value**** | **Any JavaScript value** | **A description of UI (JSX / Virtual DOM)** |
| ****Side‑Effect Handling**** | **No built‑in lifecycle hooks** | **Lifecycle methods (classes) or Hooks (functions)** |
| ****Purpose**** | **Encapsulate logic** | **Encapsulate UI structure + behavior** |

**3. Types of React Components:**

**Class Components:**

Implemented as ES6 classes extending React.Component.

Provide built‑in lifecycle methods and a constructor.

**Function Components:**

Plain JavaScript functions (or arrow functions).

Use React Hooks (e.g., useState, useEffect) for state and side‑effects.

**4. Class Components:**

**Definition:** Components defined by creating a class that extends React.Component.

**Lifecycle Overview:**

**Mounting:** constructor() → render() → componentDidMount()

**Updating:** shouldComponentUpdate() → render() → componentDidUpdate()

**Unmounting:** componentWillUnmount()

**State Management:**

Initialized in the constructor.

Updated via this.setState(), triggering re‑renders.

**5. Function Components**

**Definition:** Components implemented as functions that accept props and return JSX.

**Hooks:** useState for local state. , useEffect for side‑effects tied to render cycles.

**Advantages:** Simpler syntax, no need for constructors or explicit lifecycle methods.

Encourages cleaner, more declarative code.

**6. Component Constructor**

**Context:** Applies only to class components.

**Purpose:** Initialize internal state.

Bind event‑handler methods to the component instance.

**Signature:**

constructor(props) {

super(props);

// this.state = { … };

// this.method = this.method.bind(this);

}

**Note:** Must call super(props) before accessing this.

**7. render() Function:**

**Role:** Required in class components; defines what UI the component displays.

**Characteristics:**

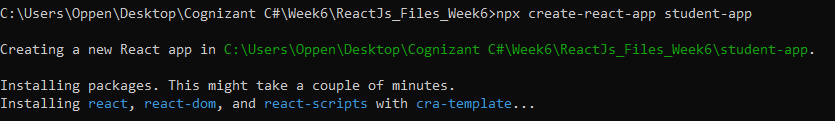
**Pure:** Should not produce side‑effects or depend on external mutable variables.

**Returns:** JSX or null to describe the component’s UI.

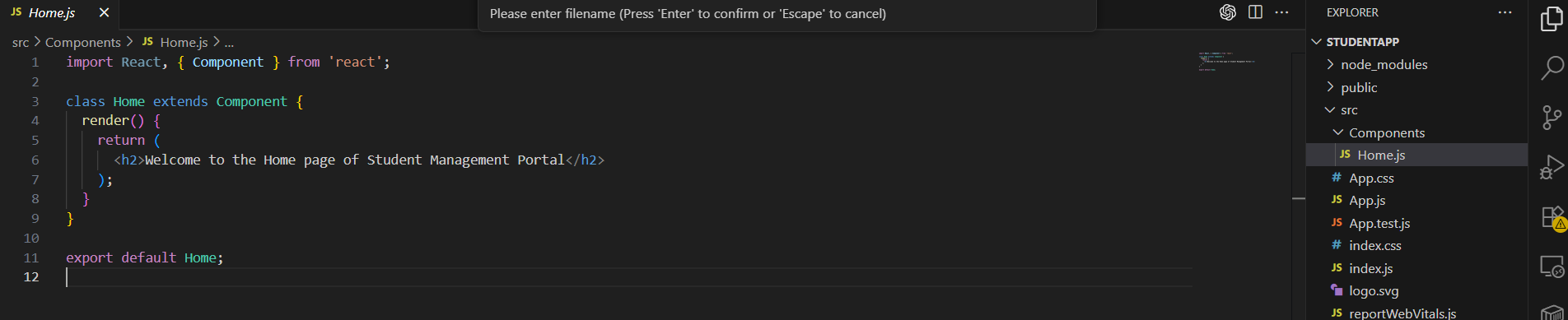
**Execution:** Invoked by React during mounting and upon state or prop changes, driving the reconciliation process that updates the actual DOM.

Create a react app for Student Management Portal named StudentApp and create a component named Home which will display the Message “Welcome to the Home page of Student Management Portal”. Create another component named About and display the Message “Welcome to the About page of the Student Management Portal”. Create a third component named Contact and display the Message “Welcome to the Contact page of the Student Management Portal”. Call all the three components.

1. Create a React project named “StudentApp” type the following command in terminal of Visual studio:



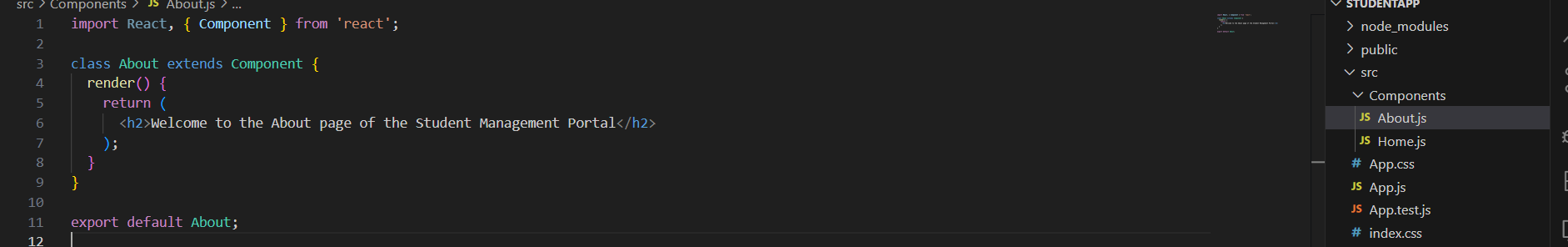
1. Create a new folder under Src folder with the name “Components”. Add a new file named “Home.js”



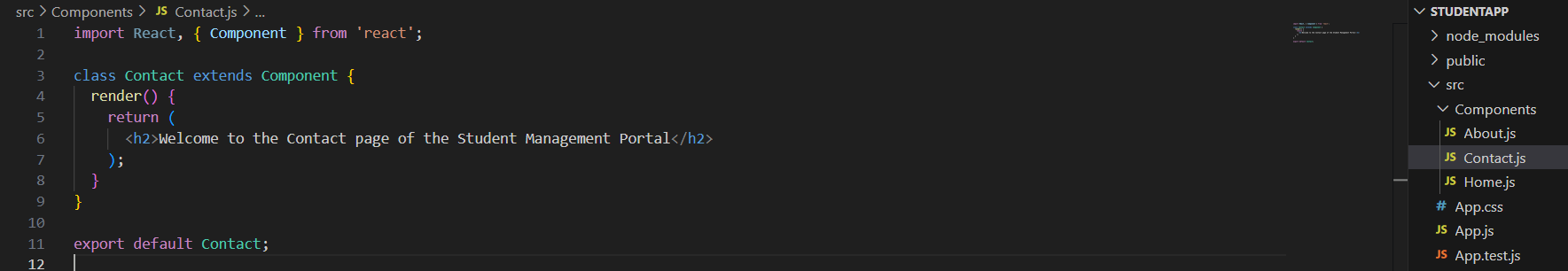
1. Type the following code in Home.js



1. Under Src folder add another file named “About.js”

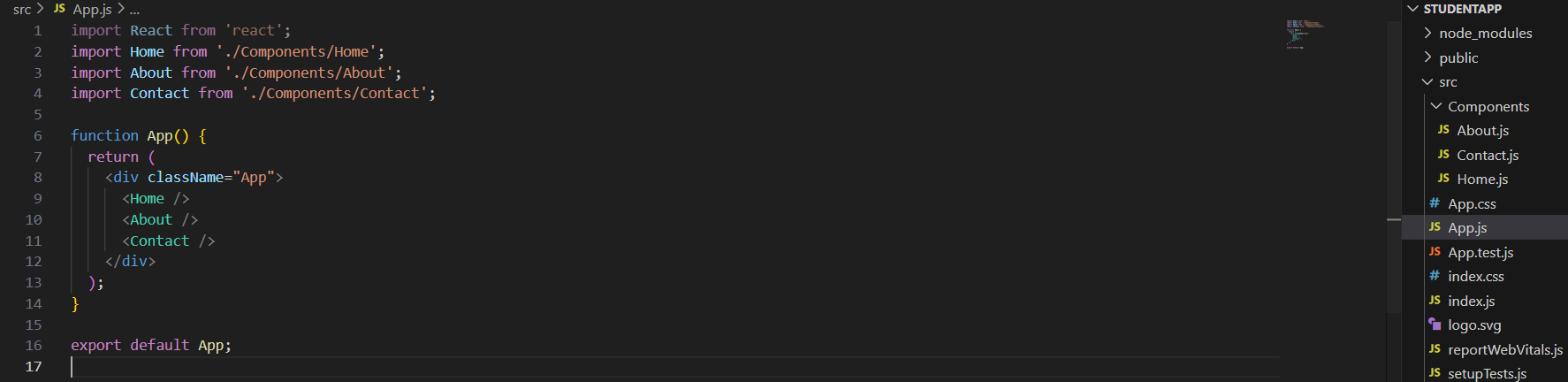


1. Repeat the same steps for Creating “About” and “Contact” component by adding a new file as ”About.js”, “Contact.js” under “Src” folder and edit the code as mentioned for “Home” Component.



1. Edit the App.js to invoke the Home, About and Contact component as follows:



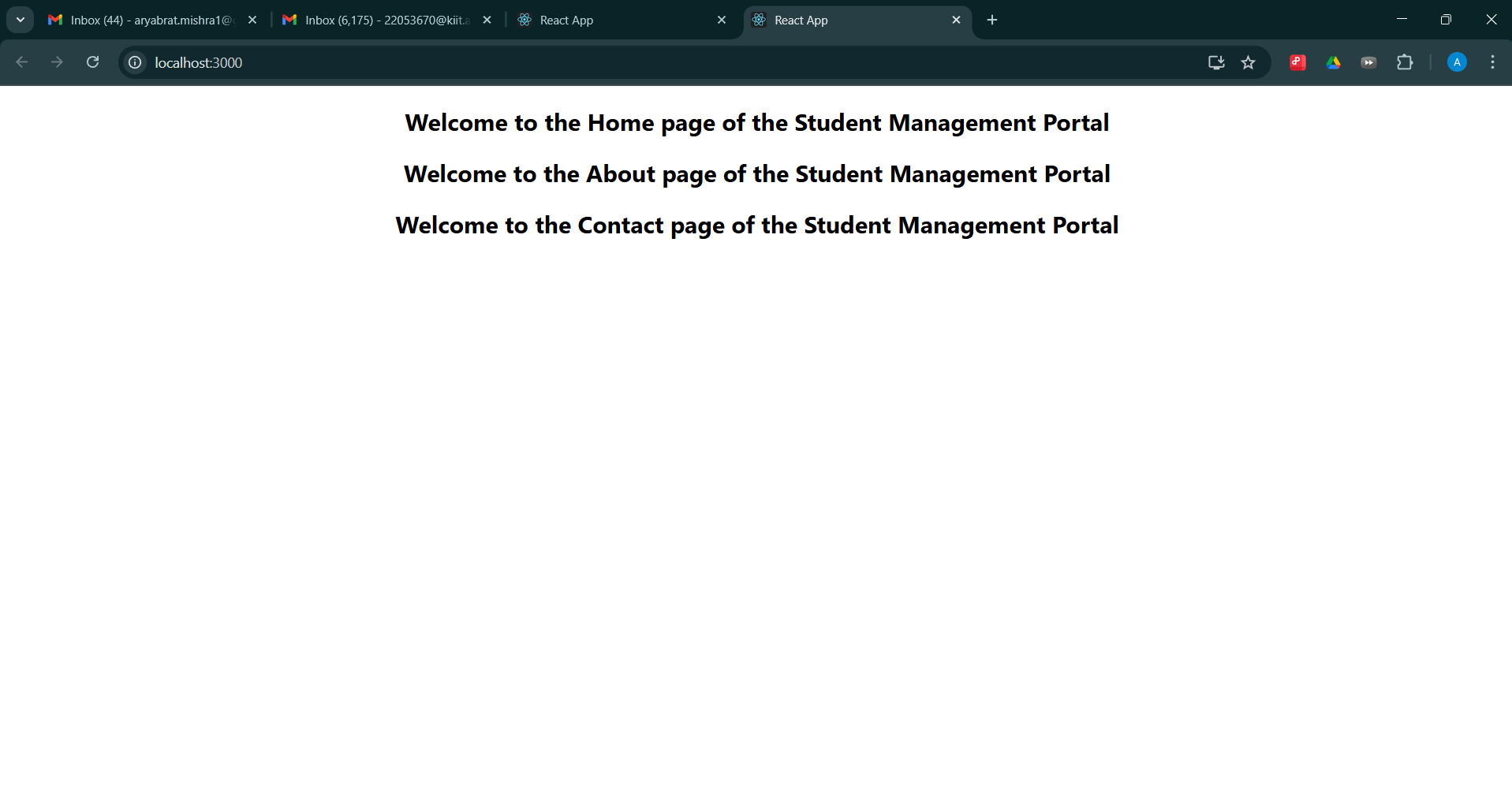


1. In command Prompt, navigate into StudentApp and execute the code by typing the following command:



1. Open browser and type “localhost:3000” in the address bar:





**HANDSON DOC - 3**

**Theories**

**1. React Components**  
**Definition:**  
A React component is a self‑contained, reusable unit of UI that encapsulates its own structure (markup), behavior (logic), and styling. Components allow you to break complex interfaces into smaller, manageable pieces.

**Key Characteristics:**

**Encapsulation:** Manages its own state and accepts inputs (“props”) from its parent.

**Reusability:** Can be composed together and reused throughout the application.

**Isolation:** Changes in one component do not directly affect others.

**2. Differences Between Components and JavaScript Functions**

| **Aspect** | **Plain JavaScript Function** | **React Component** |
| --- | --- | --- |
| **Invocation** | Called explicitly (e.g. fn()) | Instantiated by React via JSX tags (<MyComp/>) |
| **Return Value** | Any JS value | JSX (UI description) or null |
| **Side‑Effect Handling** | No built‑in lifecycle | Lifecycle methods (classes) or Hooks (functions) |
| **Purpose** | Encapsulate logic | Encapsulate both UI & logic |

**3. Types of React Components:**

**Class Components**

Defined as ES6 classes extending React.Component.

Provide built‑in lifecycle methods and a constructor.

**Function Components**

Plain JavaScript functions (or arrow functions).

Use React Hooks (e.g., useState, useEffect) for state and side‑effects.

**4. Class Component**  
**Definition:**  
A component defined by creating a class that extends React.Component.

**Lifecycle Phases:**

**Mounting :** constructor() → render() → componentDidMount()

**Updating :** shouldComponentUpdate() → render() → componentDidUpdate()

**Unmounting :**componentWillUnmount()

**State Management:**

Initialized in the constructor.

Updated via this.setState(), which triggers a re‑render.

1. **Function Component**  
   **Definition:** A component implemented as a plain function that accepts props and returns JSX.

**Hooks for State & Side‑Effects:**

useState — manages local state.

useEffect — handles side‑effects tied to render cycles.

**Advantages:** Simpler syntax (no classes, constructors, or explicit lifecycle methods).

Encourages a more declarative style of coding.

1. **Component Constructor**  
   **Context:** Only in class components.

**Purpose:**

Initialize internal state.

Bind event‑handler methods to the component instance if necessary.

**Signature & Requirements:**

constructor(props) {

super(props);

// initialize this.state

// bind methods, if needed

}

**Note:** Always call super(props) before accessing this

**7. render() Function**  
**Role:**  
The required method in class components (and the function body in function components) that returns the UI’s description.

**Characteristics:**

**Pure:** Must not produce side‑effects; only reads from this.props and this.state.

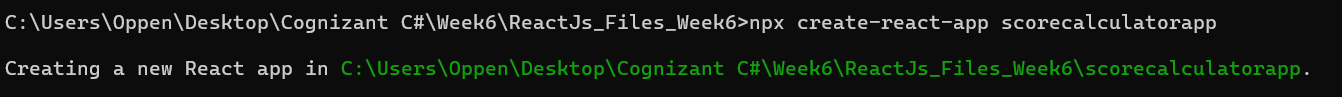
**Return Value:** JSX (or null) that describes the component’s UI.

**Invocation:** Called on mounting and whenever props or state change, driving React’s reconciliation of the real DOM.

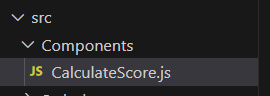
**Work Done**

1. Create a React project named “scorecalculatorapp” type the following command in terminal of Visual studio:





1. Create a new folder under Src folder with the name “Components”. Add a new file named “CalculateScore.js”



1. Type the following code in CalculateScore.js

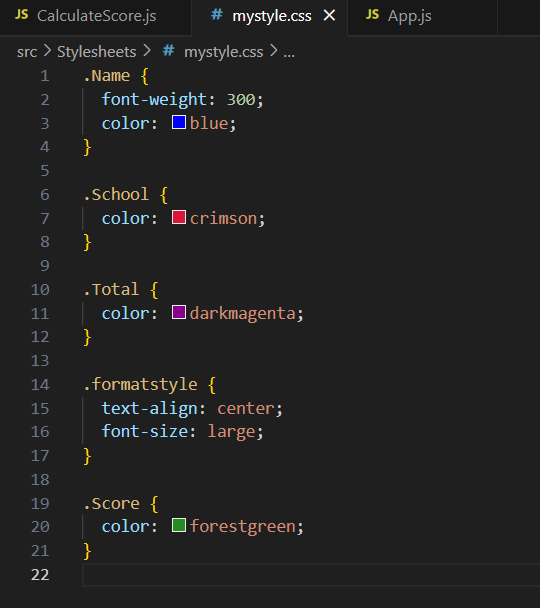






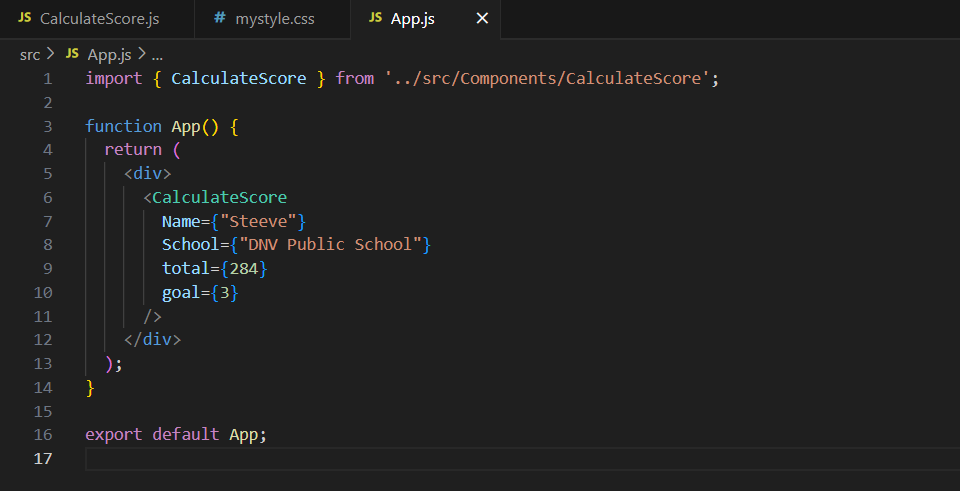
1. Create a Folder named Stylesheets and add a file named “mystyle.css” in order to add some styles to the components:





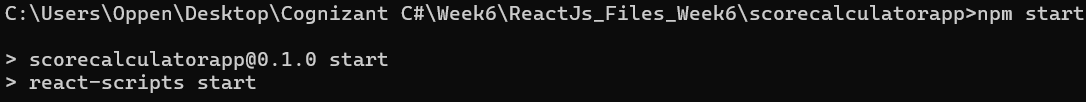
1. Edit the App.js to invoke the CalculateScore functional component as follows:





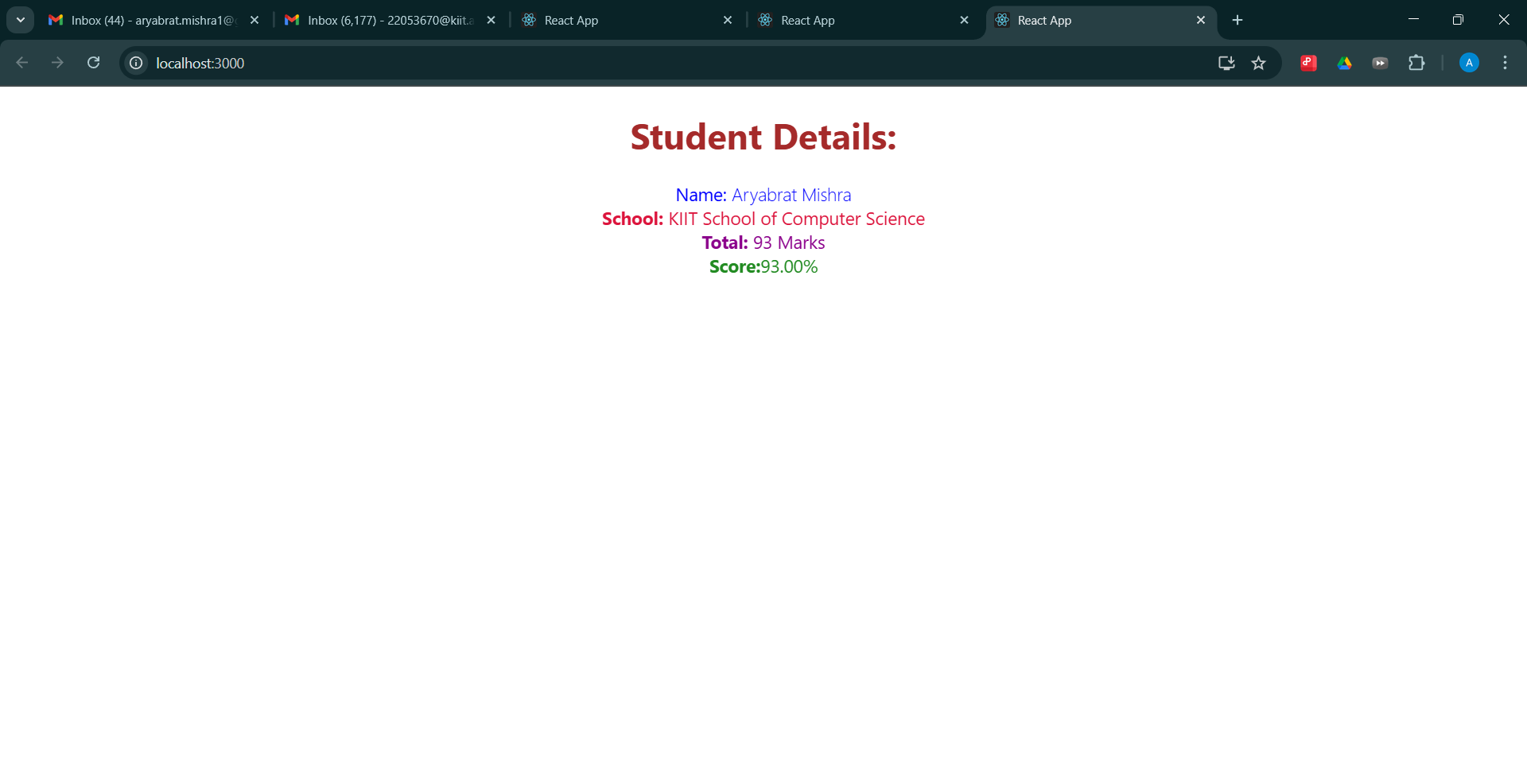
1. In command Prompt, navigate into scorecalculatorapp and execute the code by typing the following command:





1. Open browser and type “localhost:3000” in the address bar:





**HANDSON DOC 4**

**THEORIES :**

## **React Component Lifecycle**

### 1. Need and Benefits of Component Lifecycle

React components pass through distinct phases—mounting, updating, unmounting, and error handling—which expose “lifecycle” hooks. Leveraging these hooks delivers several advantages:

**Controlled Side‑Effects**

Perform API calls, subscriptions, or timers at precise moments (e.g. only after the component is rendered).

**Resource Management**

Allocate resources (e.g. event listeners) when needed and clean them up to avoid memory leaks.

**Predictable Initialization**

Set up initial state or derive state from props in a single, centralized place.

**Error Resilience**

Catch rendering errors in child components and display fallback UI without breaking the entire app.

**Performance Optimization**

Prevent unnecessary re‑renders via decision hooks (e.g. shouldComponentUpdate).

### 2. Lifecycle Hook Methods

React divides lifecycle hooks into four categories:

**A. Mounting (initial insertion into the DOM)**

constructor(props)

static getDerivedStateFromProps(props, state)

render()

componentDidMount()

**B. Updating (on props or state change)**

static getDerivedStateFromProps(props, state)

shouldComponentUpdate(nextProps, nextState)

render()

getSnapshotBeforeUpdate(prevProps, prevState)

componentDidUpdate(prevProps, prevState, snapshot)

**C. Unmounting (removal from the DOM)**

componentWillUnmount()

**D. Error Handling**

static getDerivedStateFromError(error)

componentDidCatch(error, info)

### 3. Sequence of Steps in Rendering a Component

When a class component is first rendered in an SPA, React executes these steps in order:

**Constructor :**Initialize internal state and bind methods.

**getDerivedStateFromProps** (if defined)

Sync state with incoming props.

**Render :** Produce a Virtual DOM tree based on this.props and this.state.

**Commit Phase :** React applies the Virtual DOM diff to the real DOM.

**componentDidMount :** Invoke post‑render logic (e.g., data fetching, subscriptions).

Subsequent updates (due to new props or setState) follow the “Updating” hooks in their prescribed order, then re‑commit via the real DOM.

### Hands‑On Lab Tasks

In this lab, you will deepen your understanding of React’s lifecycle by implementing two key hooks in a class component:

**Implement** componentDidMount()

**Purpose:** Execute one‑time setup tasks immediately after the component is inserted into the DOM (e.g., fetch remote data, start timers).

**Key Steps (Theory):**

Define componentDidMount within your class.

Within it, invoke side‑effect logic (API calls, subscriptions).

Update component state as needed to trigger a re‑render with fresh data.

**Implement** componentDidCatch()

**Purpose:** Act as an error boundary to catch JavaScript errors anywhere in the component tree below, log them, and render a fallback UI.

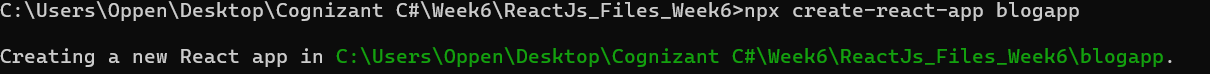
**Key Steps (Theory):**

Define static getDerivedStateFromError to update state when an error occurs.

Define componentDidCatch to perform error logging (e.g., send to an external monitoring service).

In render(), check for error state and return fallback UI if present.

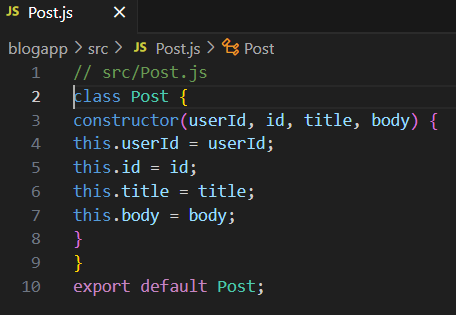
1. Create a new react application using *create-react-app* tool with the name as “blogapp”



1. Open the application using VS Code
2. Create a new file named as **Post.js** in **src folder** with following properties



Figure 2: Post class



1. Create a new class based component named as **Posts** inside **Posts.js** file



Figure 3: Posts Component

1. Initialize the component with a list of Post in state of the component using the constructor
2. Create a new method in component with the name as **loadPosts()** which will be responsible for using Fetch API and assign it to the component state created earlier. To get the posts use the url (<https://jsonplaceholder.typicode.com/posts>)



Figure 4: loadPosts() method

1. Implement the **componentDidMount()** hook to make calls to **loadPosts()** which will fetch the posts



Figure 5: componentDidMount() hook

1. Implement the **render()** which will display the title and post of posts in html page using heading and paragraphs respectively.

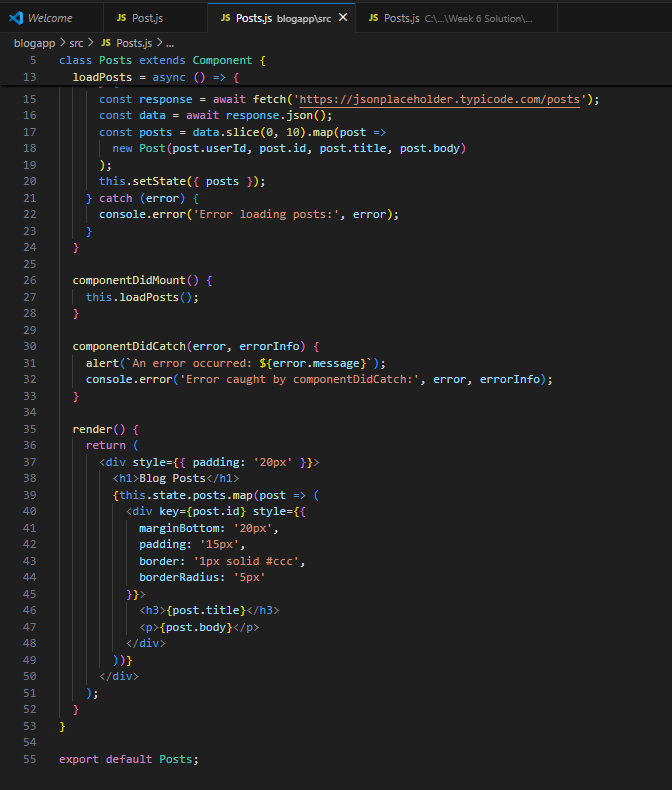


Figure 6: render() method

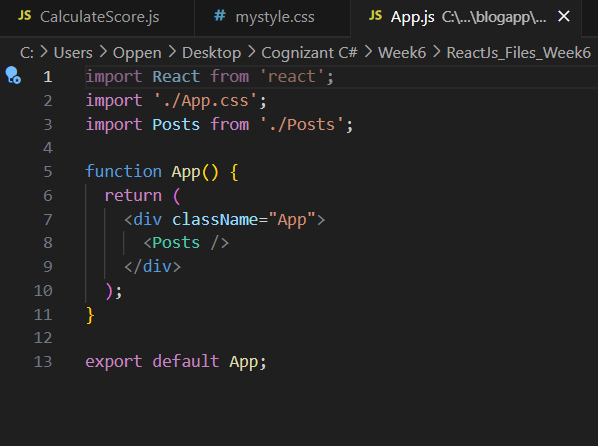
1. Define a **componentDidCatch()** method which will be responsible for displaying any error happing in the component as alert messages.



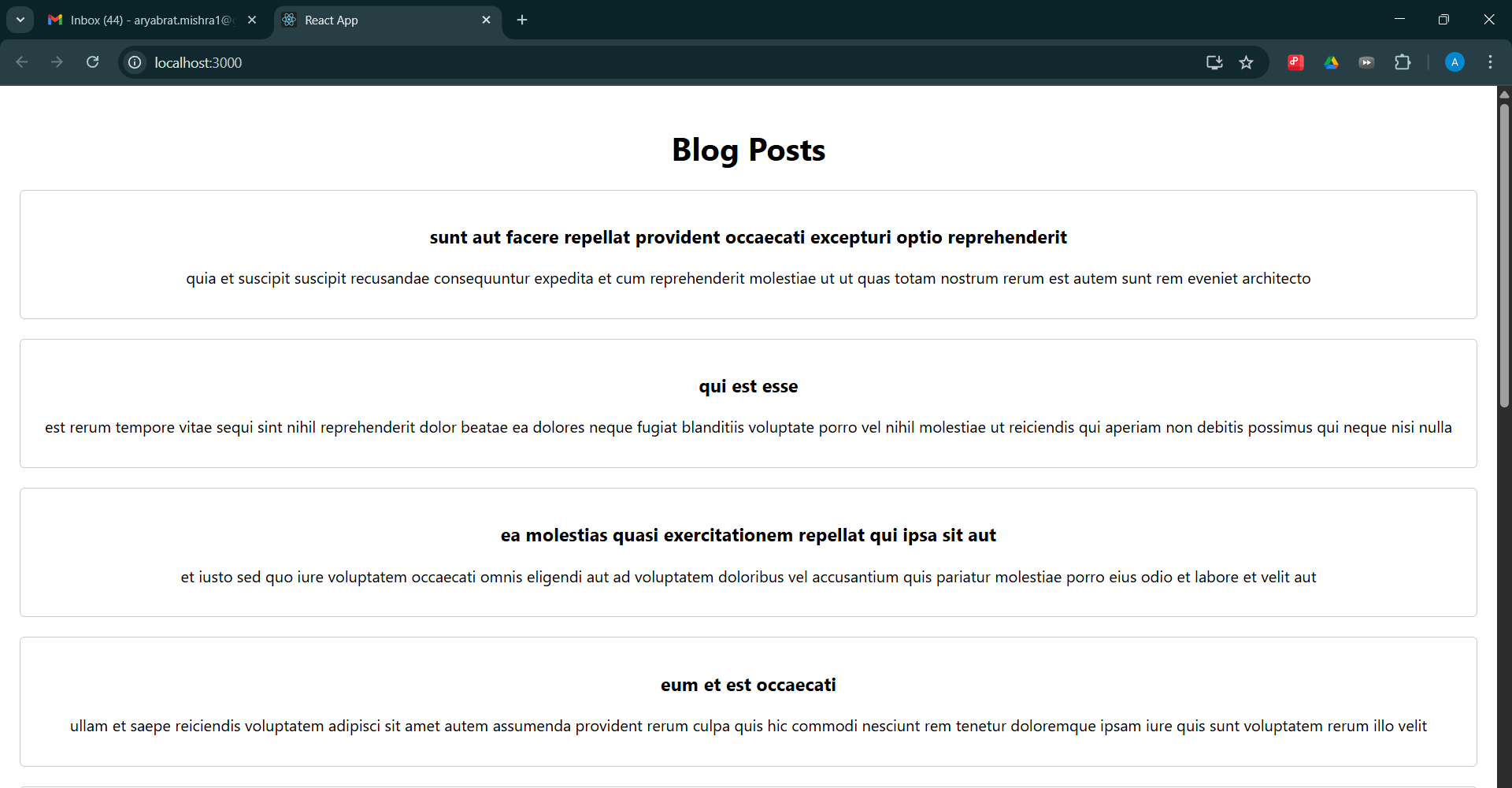
Figure 7: componentDidCatch() hook



1. Add the Posts component to App component.



1. Build and Run the application using *npm start* command.



**HANDSON DOC 5**

Theories :

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

**Why Style at the Component Level?**

**Encapsulation of Presentation**

Keeps styles co‑located with markup and behavior, making each component a self‑contained unit.

**Avoid Global Namespace Collisions**

Traditional global CSS can lead to conflicting selectors; component‑scoped styling prevents unintended overrides.

**Enhanced Maintainability**

When styles live alongside component logic, it’s easier to trace and update the look‑and‑feel of individual UI pieces.

**Dynamic & Themed UIs**

Components often need to adapt their styling based on props or application theme; local styles enable fine‑grained control.

**Performance Optimizations**

Bundlers can split out only the CSS actually used by each component, reducing overall stylesheet size and improving load times.

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

#### A. CSS Modules

**Definition:**

A build‑time technique that transforms .css files into scoped, locally‑named class maps, ensuring style rules apply only to the importing component.

**Key Characteristics:**

**Automatic Scoping:** Class names are hashed (e.g. Button\_module\_\_primary\_\_a1b2c) to avoid clashes.

**Familiar Syntax:** Write standard CSS (or preprocessors) without modifications.

**Explicit Imports:** Components import only the classes they use, making dependencies clear.

**Benefits:**

Strong encapsulation without moving away from plain CSS.

Easier code‑splitting and tree‑shaking of unused styles.

**Considerations:**

Requires build‑step support (Webpack, Vite, etc.).

Global theming via CSS variables or shared module files.

#### B. Inline Styles

**Definition:**

Styling via the React style prop, passing a JavaScript object of CSS properties.

**Key Characteristics:**

**Dynamic Values:** Compute styles at render time based on props or state.

**JavaScript-Centric:** CSS properties in camelCase (backgroundColor, fontSize).

**Benefits:**

No external stylesheet or build configuration required.

Ideal for ad‑hoc or highly dynamic styling (e.g., animations, conditional layouts).

**Limitations:**

Cannot leverage pseudo‑classes (:hover) or media queries directly.

May lead to verbose markup when overused.

Less efficient for large, static style blocks.

**Choosing Between CSS Modules and Inline Styles**

Use **CSS Modules** for component‑specific, static styles that benefit from full CSS feature set (selectors, media queries, pseudo‑classes).

Use **Inline Styles** for small, dynamic adjustments driven by JavaScript logic (e.g., toggling visibility, computing dimensions at runtime).

**Work Done**

My Academy team at Cognizant want to create a dashboard containing the details of ongoing and completed cohorts. A react application is created which displays the detail of the cohorts using react component. You are assigned the task of styling these react components.

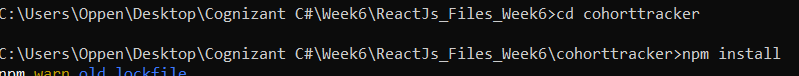
Download and build the attached react application.



1. Unzip the react application in a folder
2. Open command prompt and switch to the react application folder
3. Restore the node packages using the following commands



Figure 1: Restore packages



1. Open the application using VS Code
2. Create a new CSS Module in a file called “CohortDetails.module.css”

1. Define a css class with the name as “box” with following properties

*Width = 300px;*

*Display = inline block;*

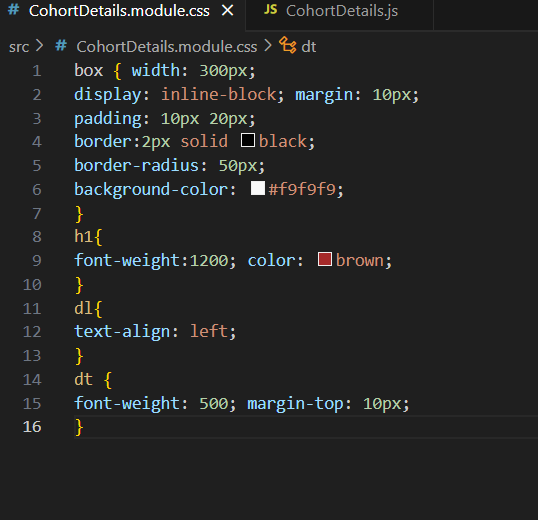
*Overall 10px margin*

*Top and bottom padding as 10px*

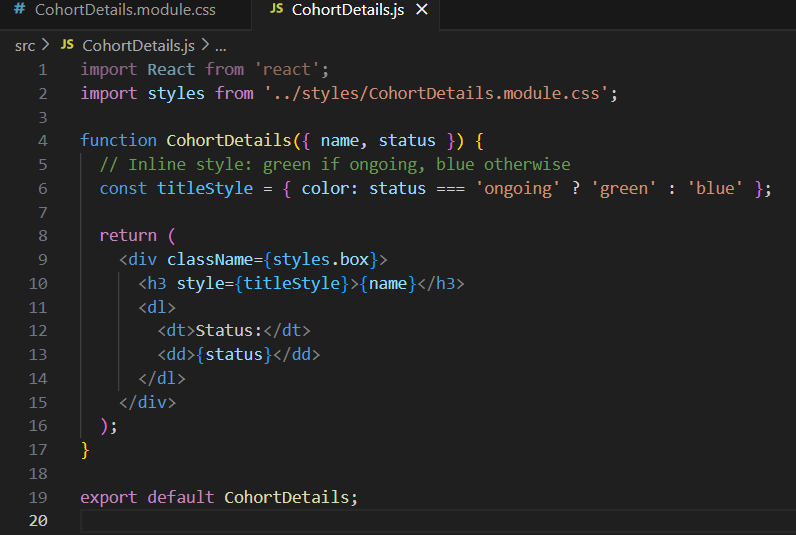
*Left and right padding as 20px*

*1 px border in black color*

*A border radius of 10px*



1. Define a css style for html <dt> element using tag selector. Set the font weight to 500.
2. Open the cohort details component and import the CSS Module
3. Apply the box class to the container div
4. Define the style for <h3> element to use “green” color font when cohort status is “ongoing” and “blue” color in all other scenarios.



1. Final result should look similar to the below image



Figure 2: Final Result

