**WEEK-06**

**REACT HANDS ON SOLUTIONS**

**ReactJS-HOL**

**EXERCISE-01- ReactJS-HOL**

IMPLEMENTATION

As part of this lab, I explored the foundational concepts of **Single Page Applications (SPA)** and implemented a basic React application.

Through this hands-on, I not only learned how to set up a React environment but also gained a better understanding of how modern JavaScript libraries work behind the scenes to create smooth and responsive user interfaces.

I began by understanding what a Single Page Application (SPA) is. A SPA is a web application that loads a single HTML page and dynamically updates content without reloading the entire page. This approach results in faster performance and a smoother user experience, especially after the initial load. Unlike traditional Multi-Page Applications (MPA), which reload the full page on each user interaction, SPAs only fetch and render the parts that need updating. This makes SPAs ideal for dynamic platforms like Gmail, Facebook, and Twitter.

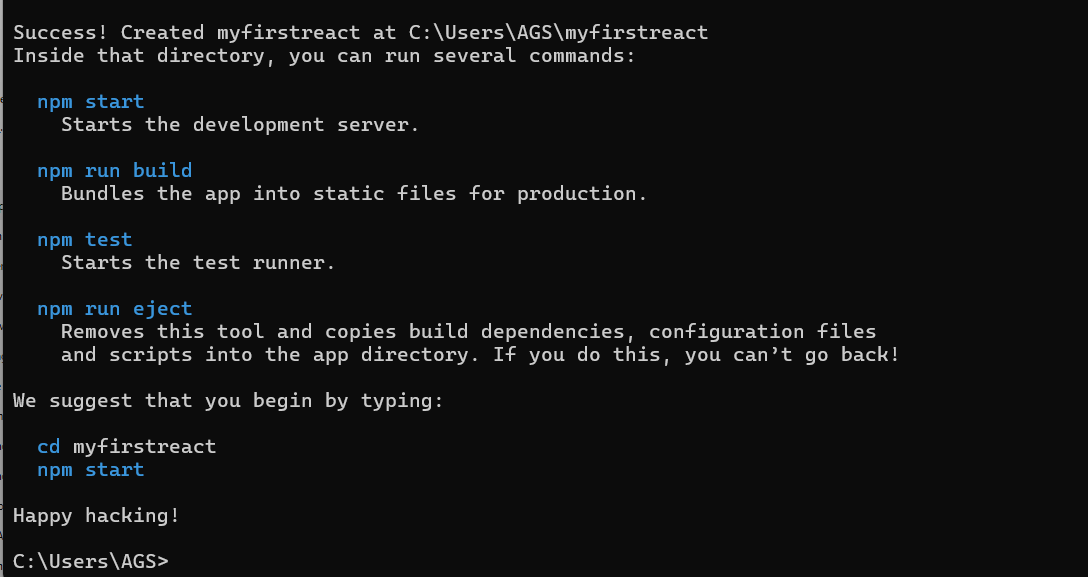
Next, I learned about **React**, a JavaScript library developed by Facebook that is widely used for building SPAs. React works by breaking the user interface into reusable components. These components are written using **JSX**, a syntax extension that allows HTML to be written inside JavaScript. React also uses something called the **Virtual DOM**—a lightweight copy of the actual DOM that enables efficient updating of the UI whenever data changes. React detects differences between the current and previous Virtual DOM, and only updates the changed parts in the real DOM, making it highly performant.

During the lab, I also explored the key differences between SPA and MPA. SPAs are faster and more interactive but can be challenging when it comes to SEO. On the other hand, MPAs are better for SEO but can feel slower due to full-page reloads. Understanding these trade-offs helped me appreciate why React is often used for SPAs and how it aligns with the needs of modern web applications.

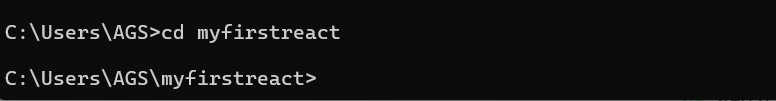
I also reflected on the **pros and cons of SPAs**. The biggest advantages are speed and responsiveness, but they come with challenges like initial load time and SEO limitations. These are important considerations when choosing the right architecture for a web application.

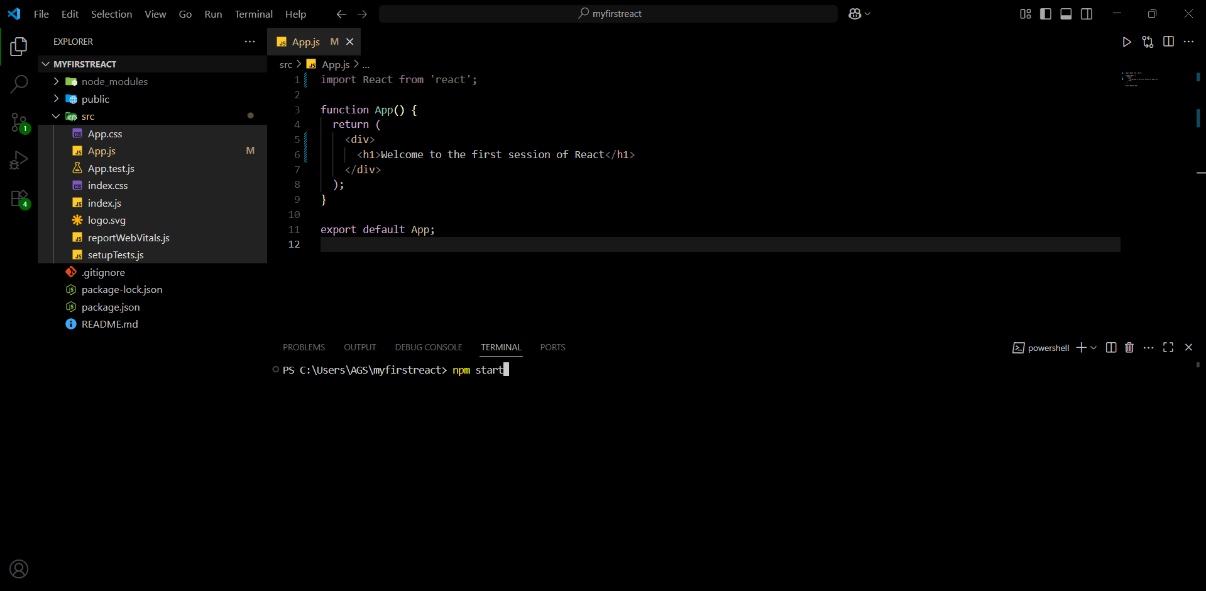
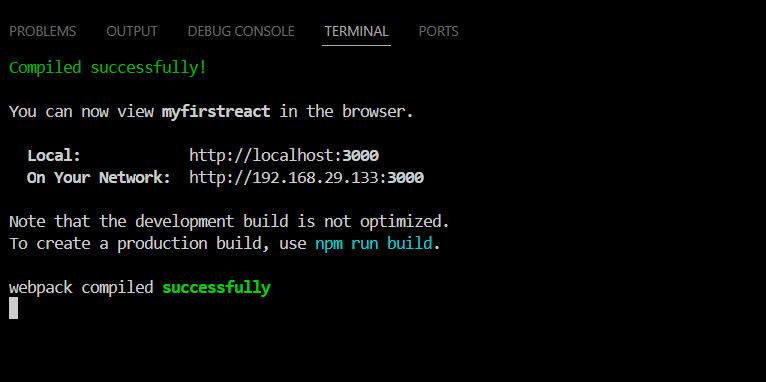
React itself offers many powerful features. Besides being component-based and using the Virtual DOM, it promotes **one-way data binding**, which makes data flow predictable and easier to manage. The introduction of **React Hooks** has also made it easier to manage state and side effects in functional components.

After going through the theoretical part, I proceeded with the practical implementation. I installed **Node.js** and **npm**, and then used the command npx create-react-app myfirstreact to scaffold a new React application. Once the project was created, I opened it in **Visual Studio Code**, navigated to the App.js file inside the src folder, and replaced its contents with a simple React component that displays a heading. The updated code showed the message “**Welcome to the first session of React**” as a <h1> heading.



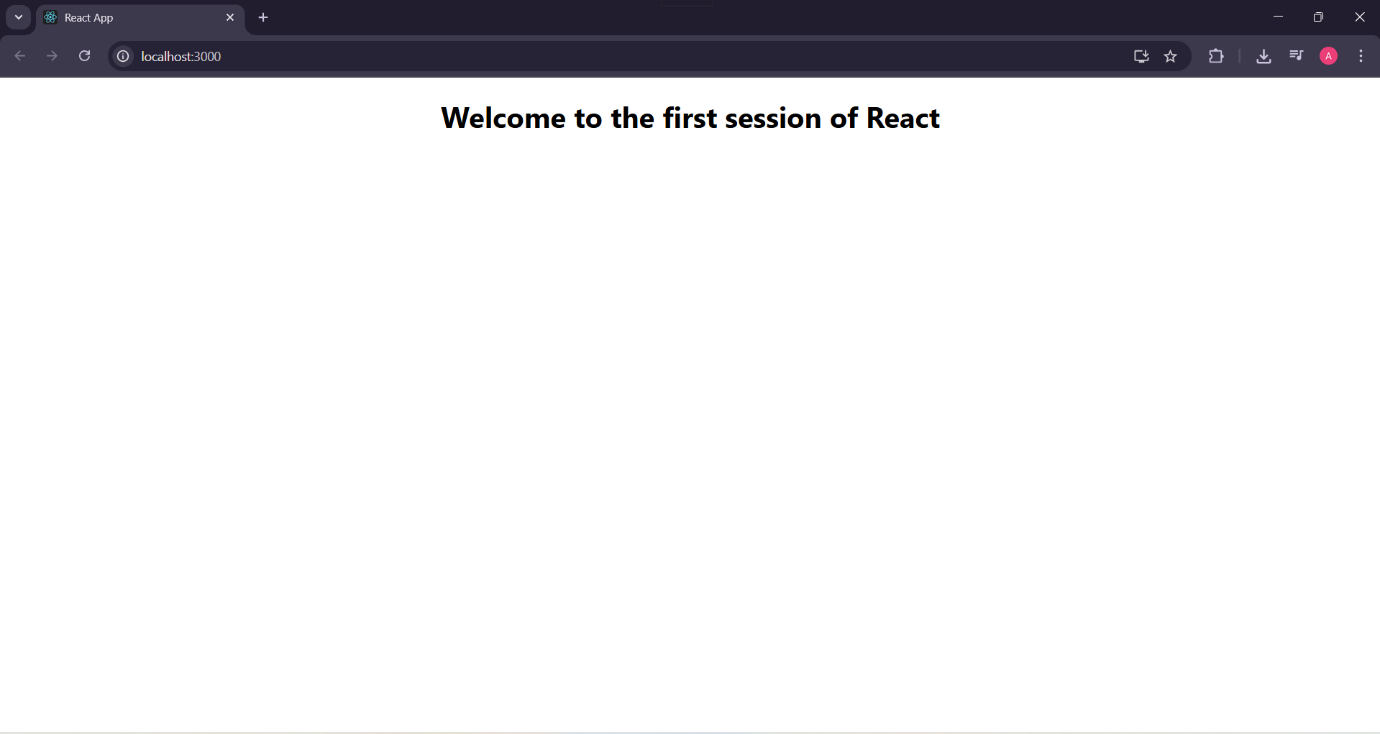
To run the application, I used the npm start command in the terminal, which launched the app in my browser at http://localhost:3000. Seeing my message displayed as a heading in the browser was a satisfying confirmation that everything worked as expected.



Overall, this hands-on helped me build confidence in setting up a React development environment, understanding how SPAs work, and getting comfortable with JSX and component-based development. It also reinforced the importance of tools like the Virtual DOM and the role of React in simplifying UI building. This was a valuable and practical first step into the world of React development.

**OUTPUT:**



**EXERCISE-02-2. ReactJS-HOL**

To implement the Student Management Portal using React, the development process began by setting up the project environment using the create-react-app command-line utility.

This utility simplifies the bootstrapping of modern React applications by generating a structured project with pre-configured tools such as Webpack, Babel, and ESLint.

The project was named studentapp—intentionally in lowercase—to comply with npm package naming conventions, which disallow capital letters.

After the project was created successfully, I navigated into the project directory using the cd studentapp command to begin development.

Following the setup, the next step was to organize the source files for modular development.

A dedicated folder named Components was created inside the src directory.

This folder acts as a logical container for React components, supporting code reusability and maintainability, which is a fundamental principle in component-based architecture. Inside this folder, three functional components—Home.js, About.js, and Contact.js—were developed.

Each of these components was defined as a pure JavaScript function that returns JSX, the XML-like syntax extension for JavaScript that enables declarative UI creation in React.

In each component file, the React module was imported at the top to gain access to core React functionalities.

The Home component, for example, was created to return a simple JSX structure containing a heading that says “Welcome to the Home page of Student Management Portal.”

This structure was replicated for the About and Contact components with respective message variations.

These components were designed to be stateless and without lifecycle methods, thus making functional components the optimal choice due to their simplicity and performance advantages.

Once the individual components were created, they were imported into the App.js file—the root component of the application.

This step involved using JavaScript import statements to bring the Home, About, and Contact components into scope.

Inside the App component’s JSX return block, each of these components was invoked using their respective tags (<Home />, <About />, <Contact />).

This illustrates the core concept of component composition in React, where smaller components are assembled together to form a complete user interface.

After the components were successfully integrated into the main application component, the development server was initiated by executing npm start in the terminal.

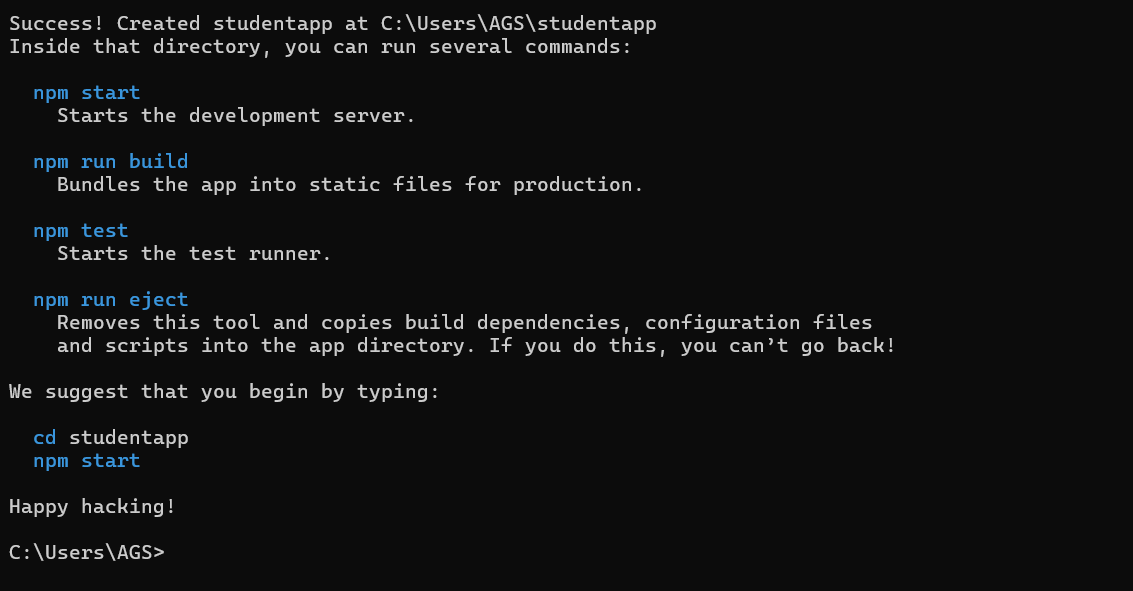
This command starts the React development server, compiles the code, and automatically launches the application in a web browser at <http://localhost:3000>.

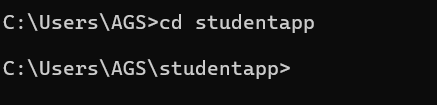
Upon visiting this URL, the expected output of all three components—Home, About, and Contact—was displayed in sequence, confirming that the components were rendered correctly.

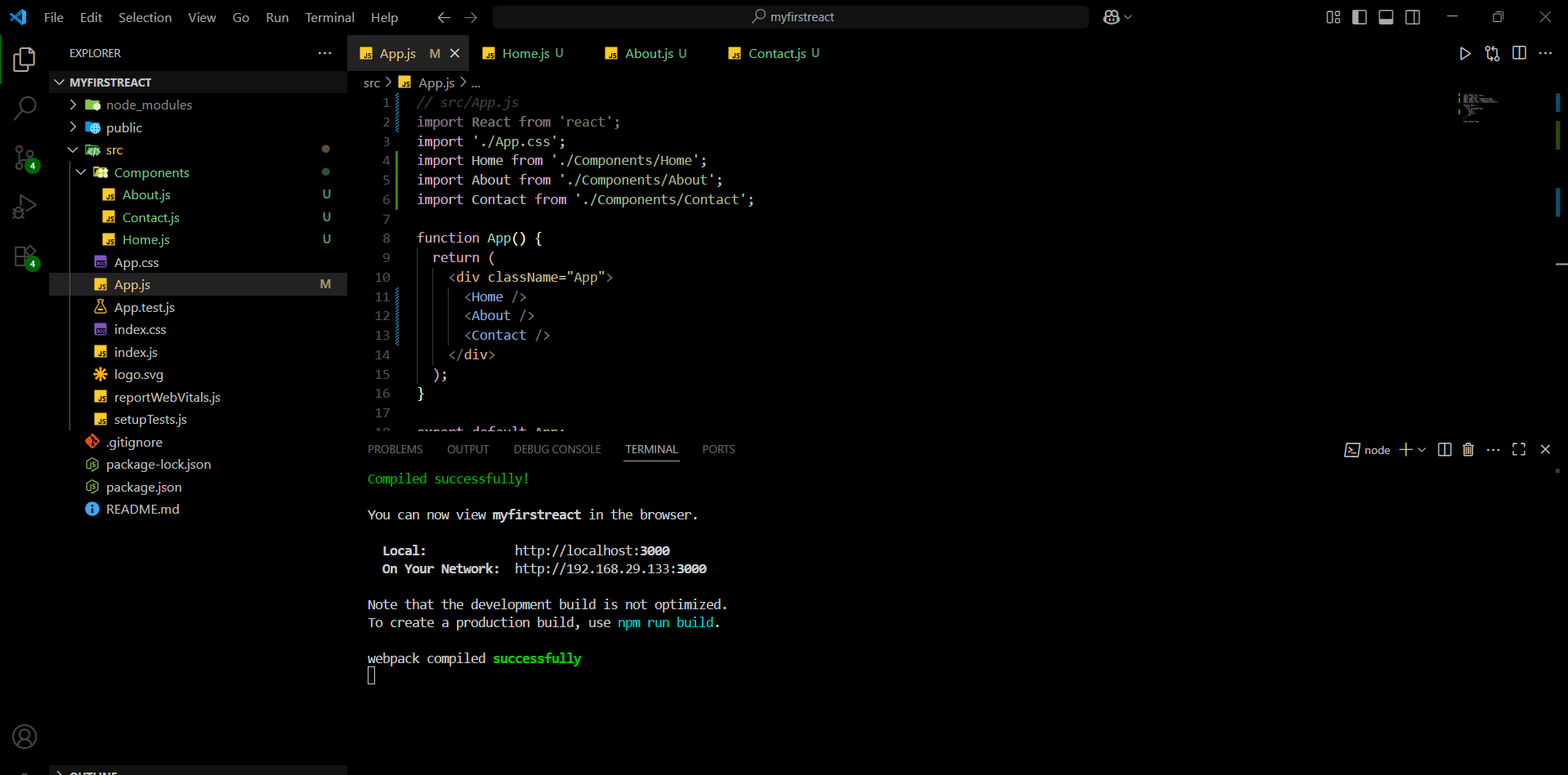
Overall, this hands-on exercise demonstrated foundational React principles such as component creation, JSX syntax, modular file organization, and rendering logic.

It reinforced the importance of reusable UI elements, declarative programming practices, and clean component-based architecture.

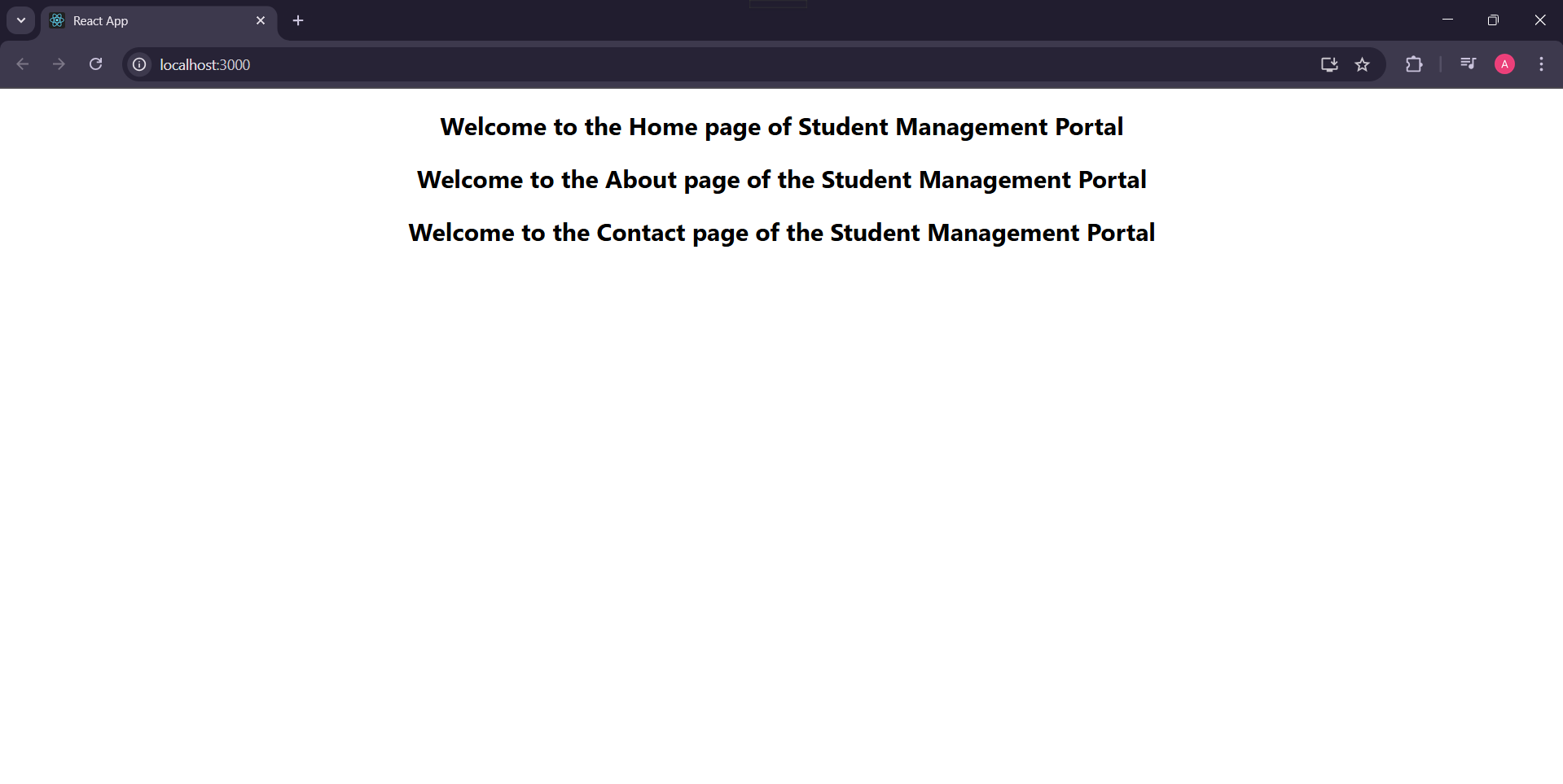
The structure laid down in this lab serves as a base for extending the application further with advanced concepts such as props, state management, routing, and API integration.







**OUTPUT:**



**EXERCISE-03 -** **3. ReactJS-HOL**

IMPLEMENTATION:

The Score Calculator App is a modular front-end application developed using the React library, which is known for building efficient and component-driven user interfaces.

The development process began by initializing the project environment with the command npx create-react-app scorecalculatorapp, which scaffolded a new React project with all the necessary dependencies and boilerplate configurations.

Due to React's naming conventions, the application folder name was kept in lowercase to avoid compilation issues.

Once the project structure was successfully created, the working directory was changed using the cd scorecalculatorapp command, marking the entry into the project’s development phase.

Within the src directory of the project, a subdirectory named Components was manually created to house custom reusable components, in line with React’s component-centric design principles.

Inside this folder, the primary functional component CalculateScore.js was implemented.

This component was developed using modern JavaScript ES6 syntax, specifically utilizing arrow functions and destructuring to enhance readability and maintainability.

Additionally, a CSS file named mystyle.css was placed in the same directory to apply consistent and scoped styling across the component.

The component makes use of two helper functions: percentToDecimal, which converts percentage goals into decimal format, and calcScore, which calculates the final score by multiplying the total marks with the goal percentage.

These utility functions are defined within the same component to preserve cohesion and encapsulate the logic relevant to score computation.

The CalculateScore functional component receives several props, namely the student’s name, the name of the school, the total marks obtained, and the goal percentage.

These props are accessed using object destructuring directly within the component signature.

The computation logic is handled internally and the results are rendered using JSX syntax, which closely resembles HTML but is fully compatible with JavaScript expressions.

The output layout includes labeled display elements for each of the prop values along with the calculated score.

This layout is styled using the .formatstyle class defined in mystyle.css, enabling the text and structure to appear organized and visually appealing in the browser.

To render this component in the browser, it was imported into the root App.js file and instantiated using JSX with appropriate sample values for each prop.

For example, the component call <CalculateScore Name="Aaisha Sultana" School="VPS Public School" total={500} goal={5} /> displays the calculated result for a student with a total of 100 marks and a goal of 100%.

The application was subsequently launched using the npm start command, which triggered the React development server and rendered the component in the browser window, displaying the computed output dynamically based on the given input props.

From an architectural standpoint, the application strictly follows a stateless, functional paradigm.

The choice of functional components ensures predictability, ease of testing, and enhanced performance due to the absence of internal state.

Props are used as the primary means of passing data, ensuring a unidirectional data flow which aligns with React’s core principles.

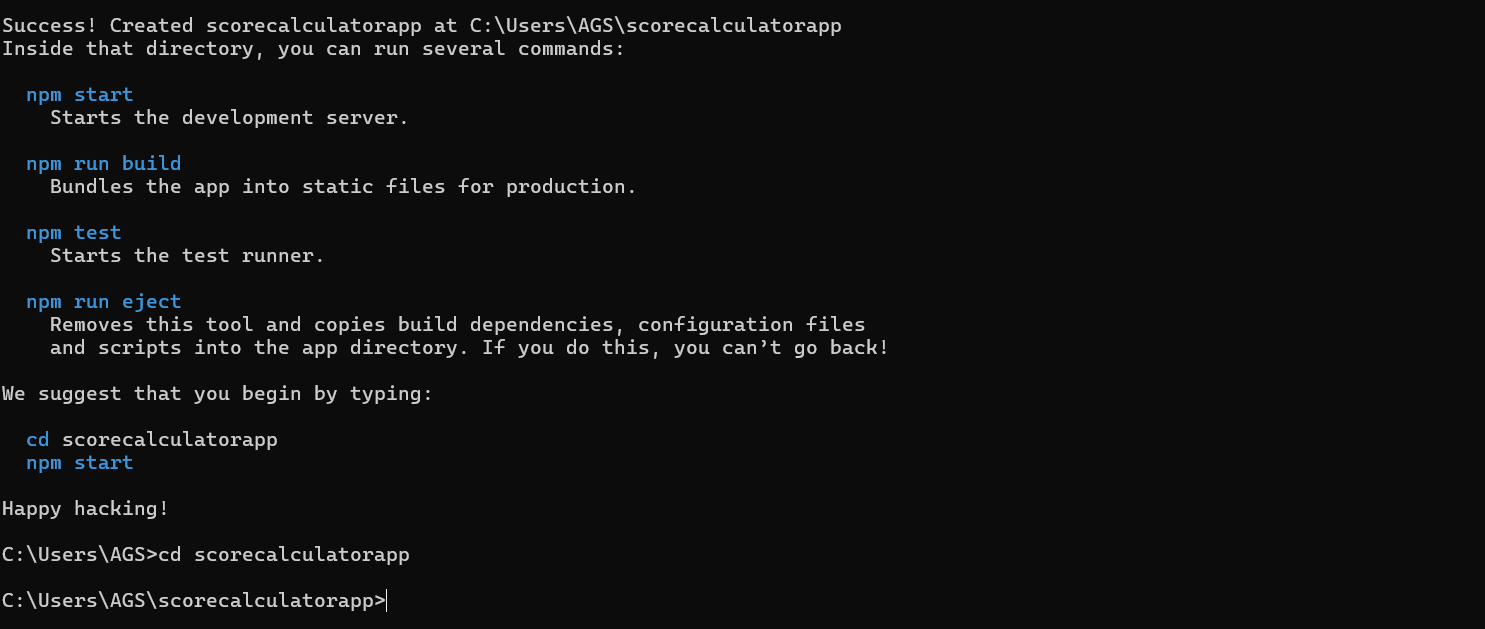
The application emphasizes separation of concerns by isolating business logic within utility functions and styling logic within a separate CSS file.

Furthermore, by keeping the computation logic pure and side-effect-free, the application maintains high readability and debuggability.

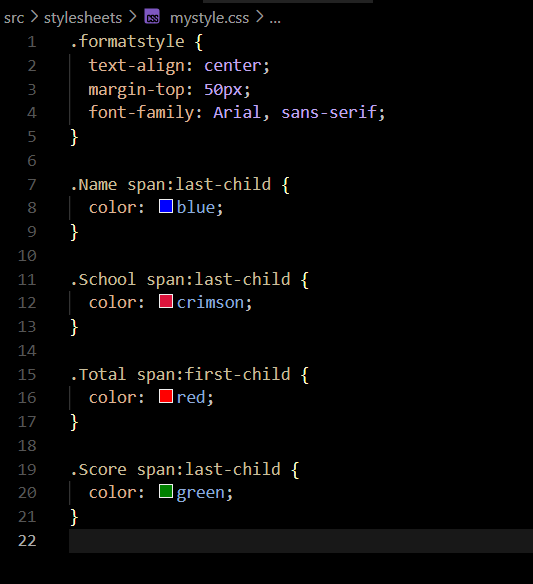
Overall, the Score Calculator App served as a foundational example of how to build lightweight, scalable, and maintainable components in React.

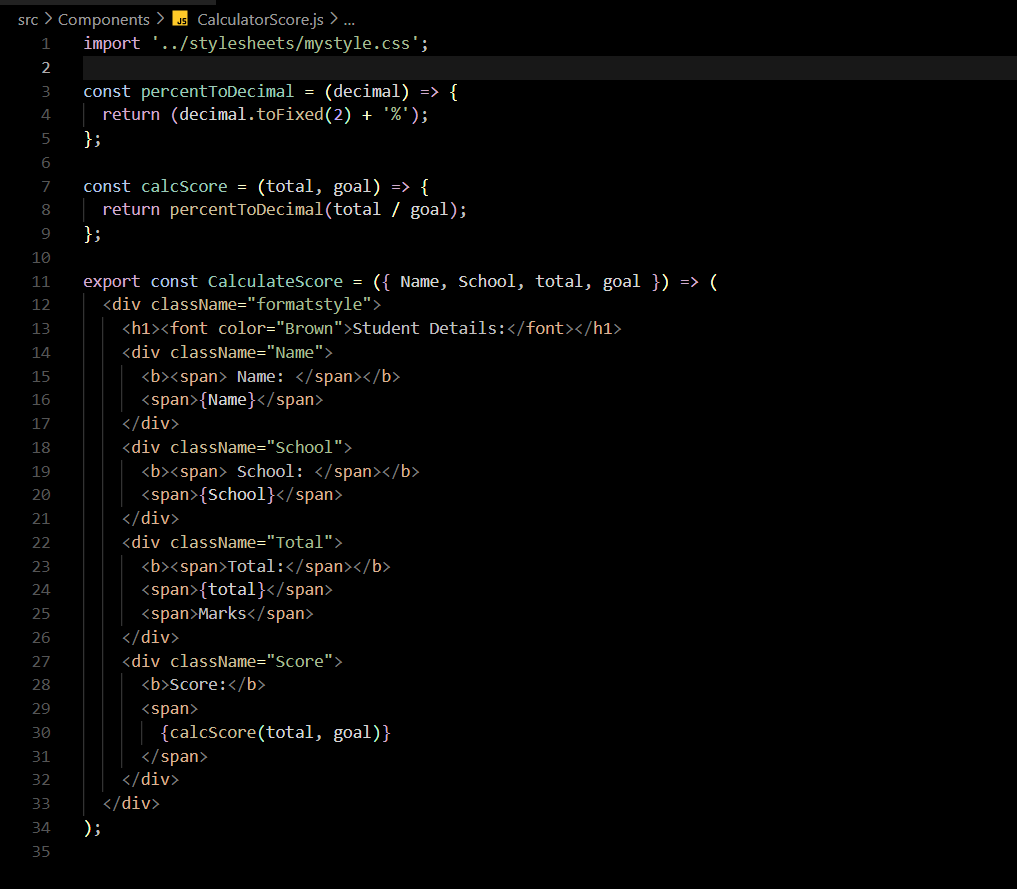
It adheres to best practices in component-based design, functional programming, and front-end architecture.

The design choices made during implementation support code modularity and reuse, making it an ideal educational tool for understanding core React concepts while delivering a simple, yet functional user-facing interface.

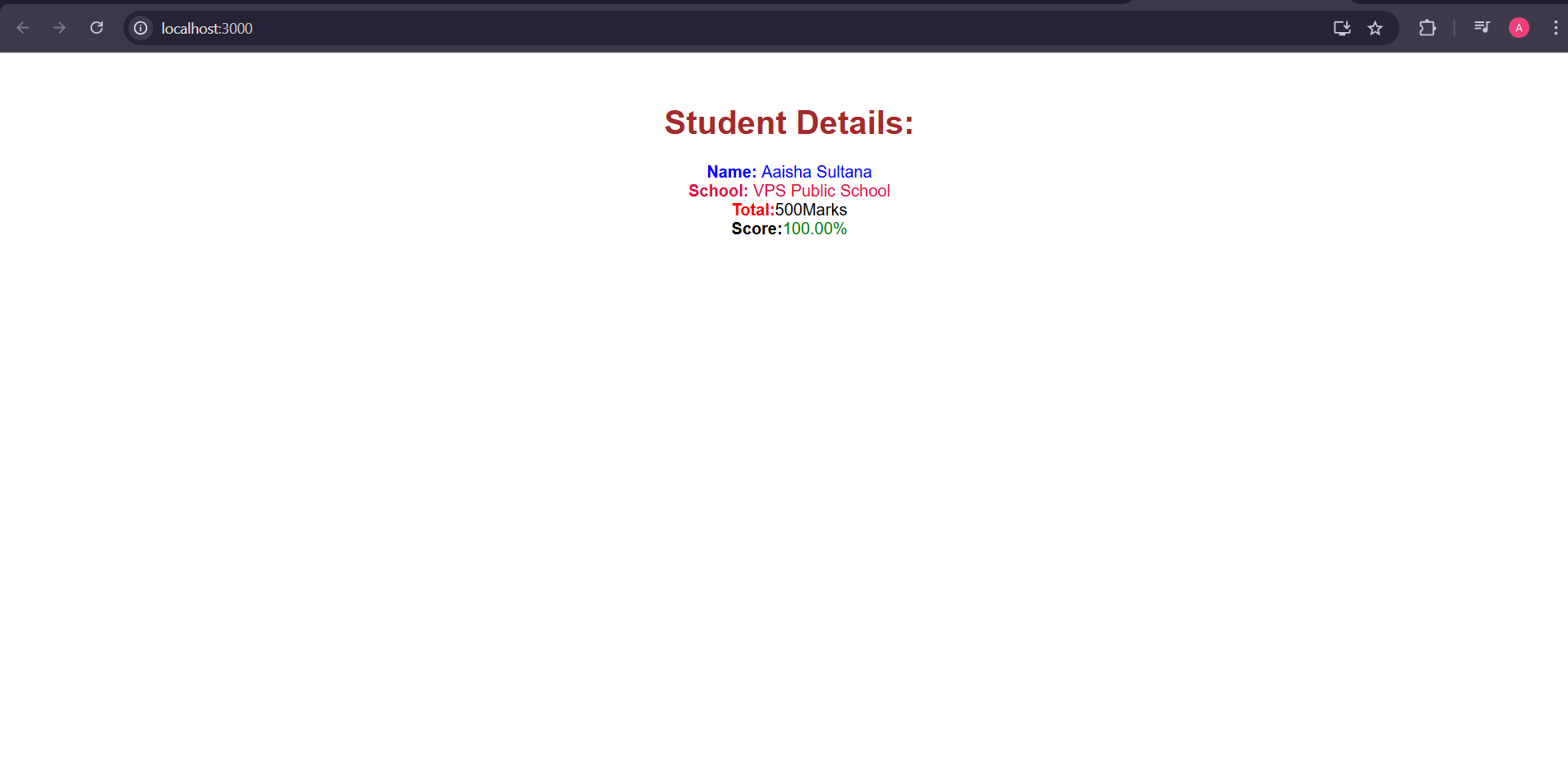
****

****

****

****

**OUTPUT:**



**EXERCISE-04**

Project Setup

A new React application was created using create-react-app with the name blogapp.

The project was opened in Visual Studio Code for development.

**Post Component**

A class component named Post was created in a new file Post.js.

Although not used further in rendering, it was a part of the initial setup for the structure.

**Posts Component**

Another class component named Posts was created in Posts.js.

This component was responsible for fetching and rendering blog posts.

**State Initialization**

The component was initialized with an empty posts array in its state using the constructor.

**loadPosts() Method**

A method loadPosts() was implemented using the Fetch API to retrieve blog posts from https://jsonplaceholder.typicode.com/posts. The fetched data was stored in the component's state. **componentDidMount() Hook**

This lifecycle hook was implemented to automatically call loadPosts() when the component was mounted to the DOM.

This allowed us to fetch and display the posts as soon as the component loaded.

**Rendering Posts**

The render() method was implemented to display the title and body of each post using HTML tags (<h3> for titles and <p> for body).

The posts were dynamically displayed by mapping over the array in state.

**Error Handling using componentDidCatch()**

The componentDidCatch() lifecycle hook was implemented to catch and display any runtime errors that might occur within the Posts component or its child components.

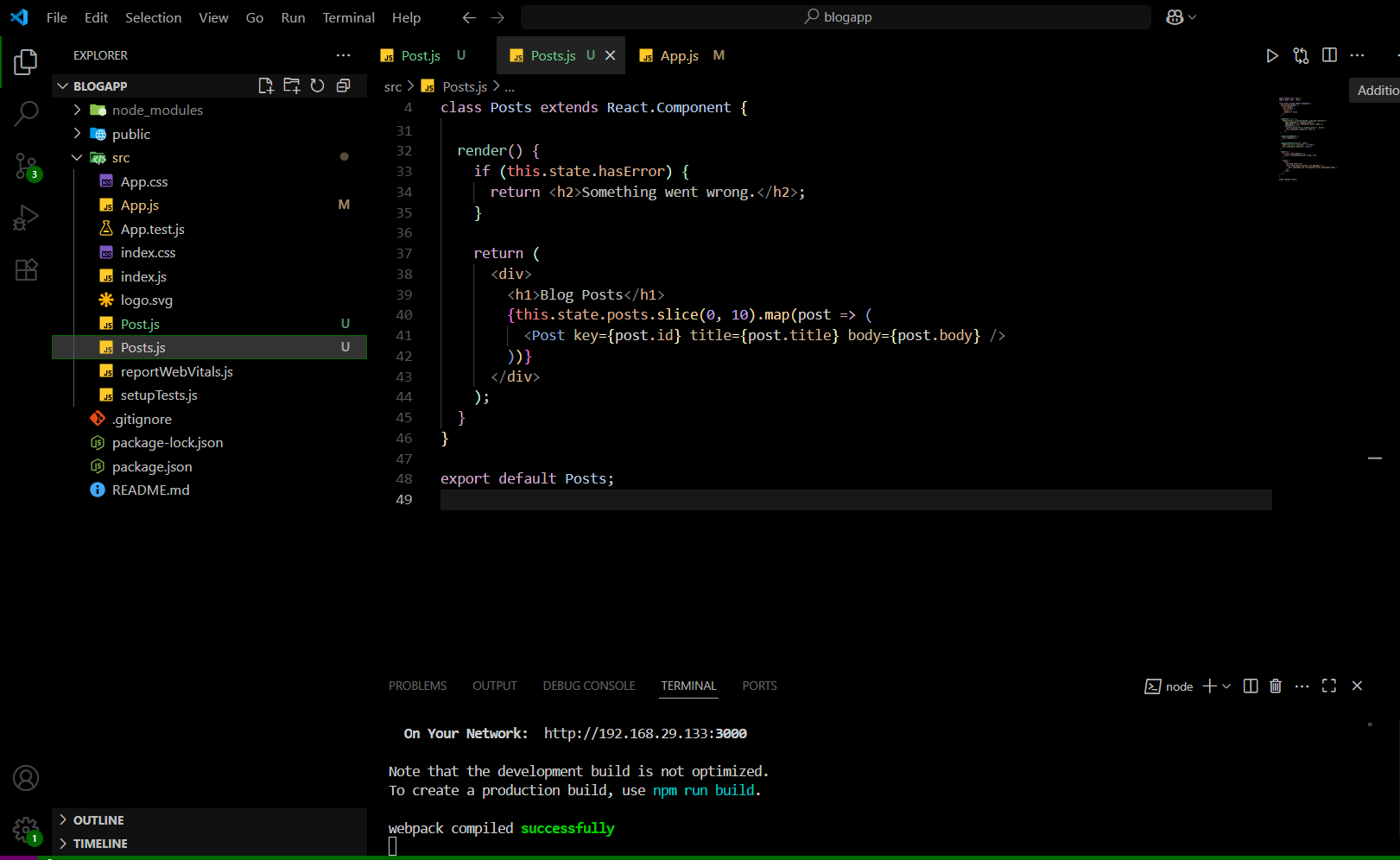
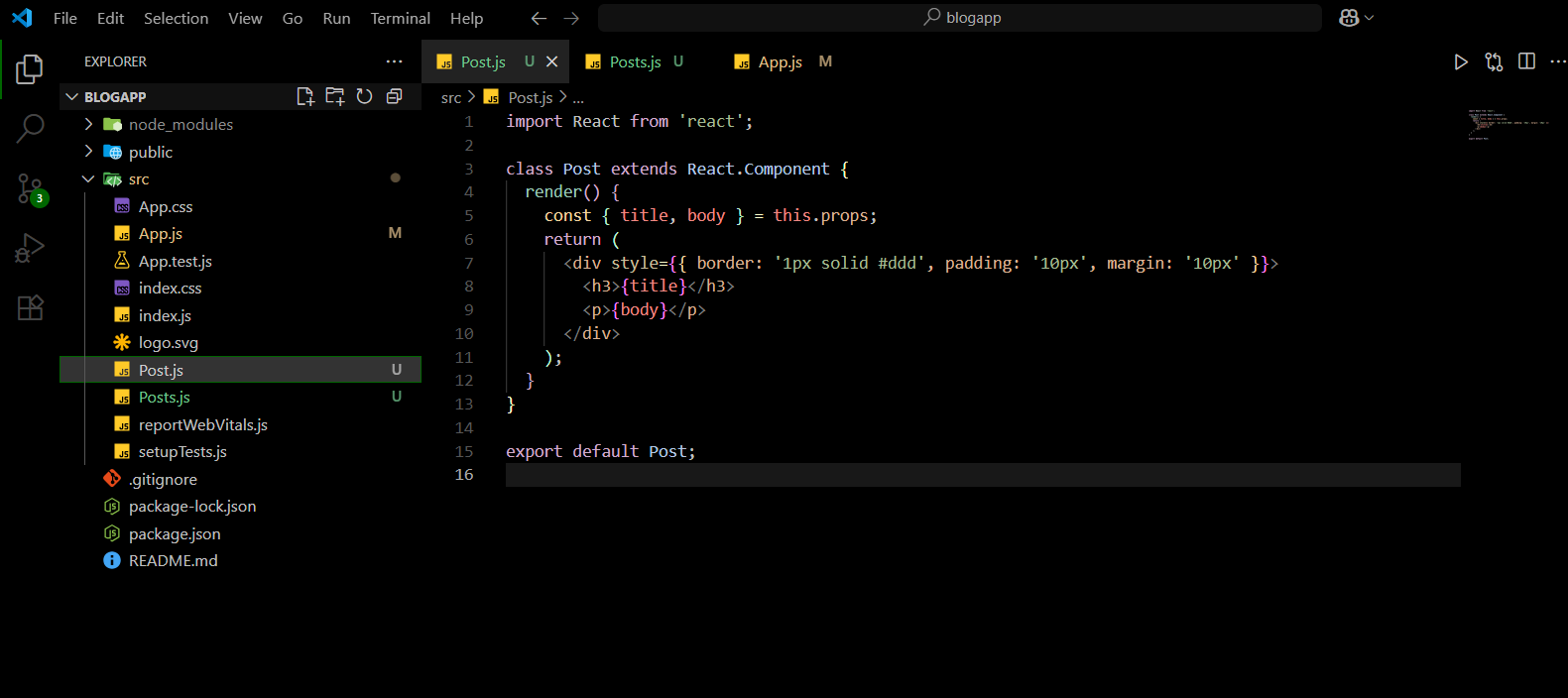
If an error occurs, an alert is shown.

**Integration with App Component**

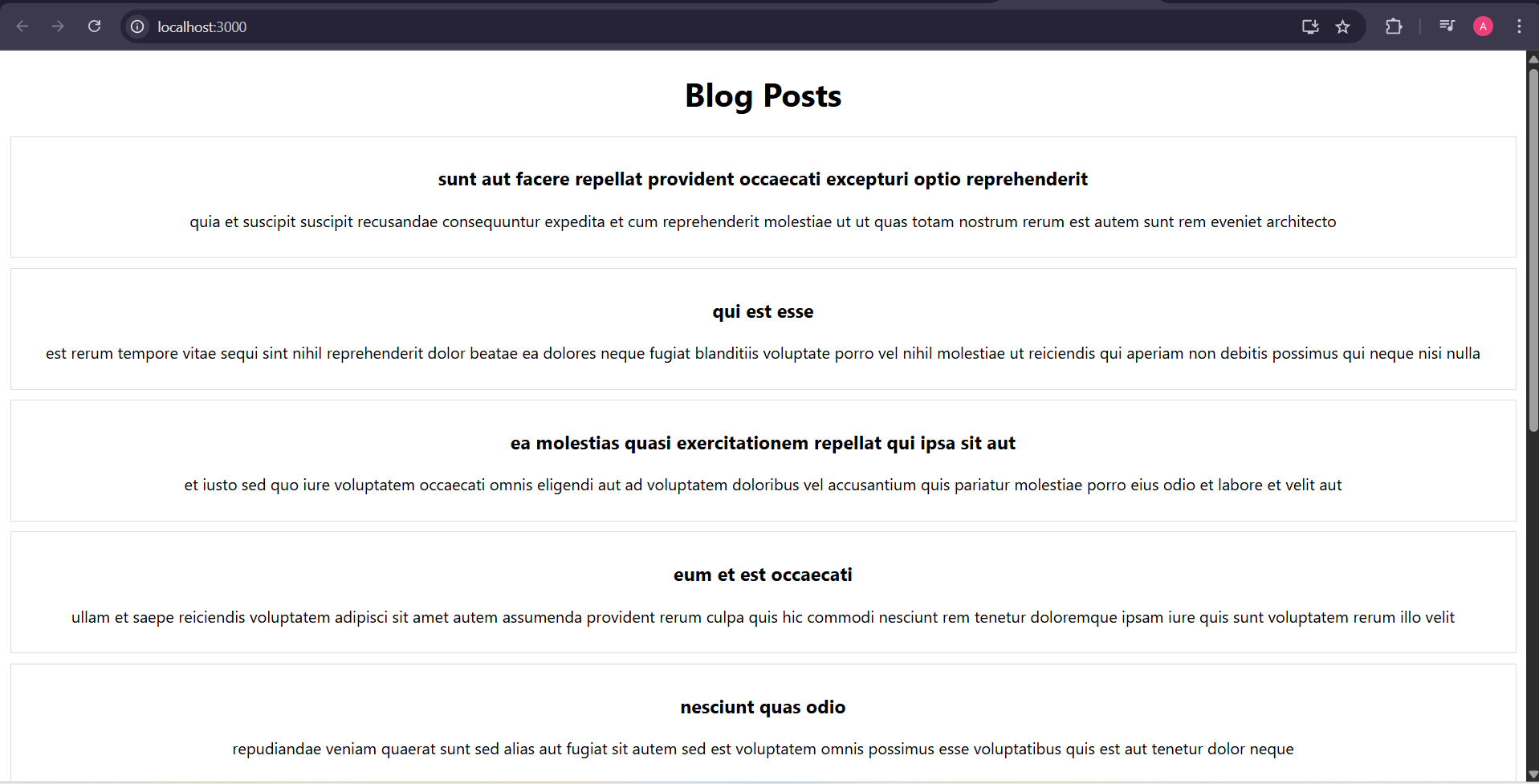
Finally, the Posts component was imported and used within App.js, completing the application setup.

Running the Application The application was successfully built and run using npm start.

The browser correctly displayed the list of blog posts fetched from the API.



**OUTPUT:**



**EXERCISE-05- 5-ReactJS-HOL**

**IMPLEMENTATION:**

As part of this hands-on exercise, I developed a modular and semantically rich React component named CohortDetails that displays detailed information for a given cohort in a structured and visually appealing card layout.

This component was designed using **React functional programming principles** and was built to accept a single cohort object as a prop, ensuring that the component remains reusable and adheres to the concept of **unidirectional data flow** in React.

The data was imported from a separate module (Cohort.js), allowing the parent component (App.js) to dynamically render multiple cohort cards through the .map() function.

To maintain clean, maintainable, and accessible markup, I replaced traditional <div>-based key-value rendering with semantic HTML using the <dl>, <dt>, and <dd> tags.

This choice greatly enhanced the **accessibility (a11y)** of the component by structuring data in a way that is recognized by screen readers and assistive technologies.

Each <dt> represents the label (definition term), and each <dd> displays the associated value (definition description), thereby offering a clearer and more meaningful structure to the cohort information.

This semantic improvement also supports better SEO and aligns with modern front-end best practices.

For styling, I implemented **CSS Modules**, a methodology that scopes CSS class definitions locally to each component rather than globally across the app.

This helps prevent style collisions in larger applications and improves overall maintainability.

I created a dedicated style file named CohortDetails.module.css, where I defined styles for the card container (.cohortCard), semantic list structure (.box), and individual label/value fields (.detailLabel and .detailValue).

The .box class applied display: flex with a flex-direction: column and gap: 10px to ensure proper spacing and alignment between data fields.

Additionally, the card styling included padding, border-radius, and a light box-shadow to enhance visual hierarchy and user readability.

A significant technical decision was the use of **conditional inline styling** for the cohort title.

I implemented a logic block that checks the technology property of the cohort and assigns the title color accordingly—for example, blue for .NET FSD and green for others.

This runtime styling approach allows dynamic visual cues based on the underlying data and is useful for distinguishing between different cohort streams at a glance.

The inline style was injected directly into the JSX for the heading using React’s style attribute, allowing the UI to reflect logical decisions tied to the data model.

From a layout perspective, the component architecture was designed to scale.

In the App.js file, the array of cohort data was iterated using the map() function, and each object was passed as a prop to an instance of the CohortDetails component.

I also wrapped the cards within a .cohort-container class, styled using Flexbox, to ensure the cards align properly and respond well across different screen sizes.

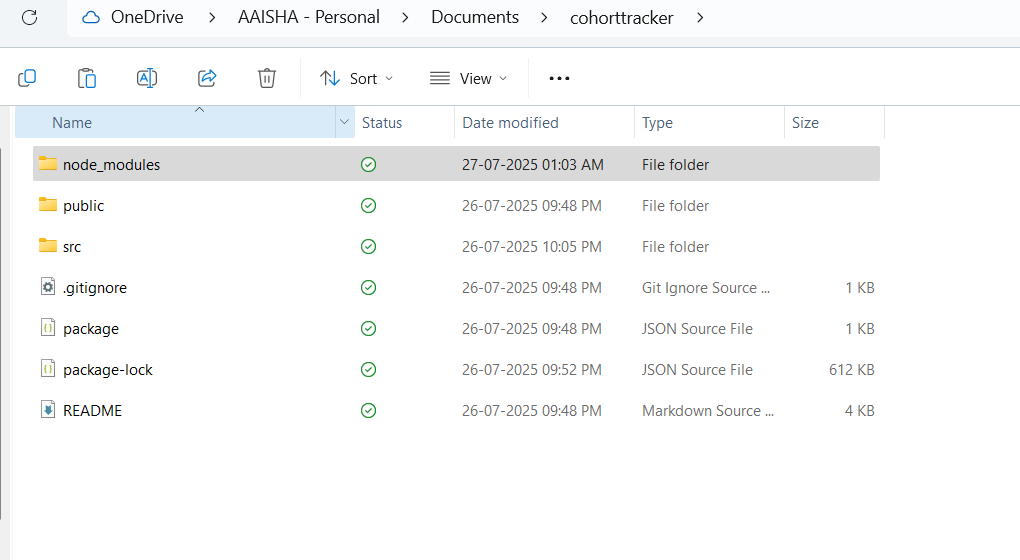
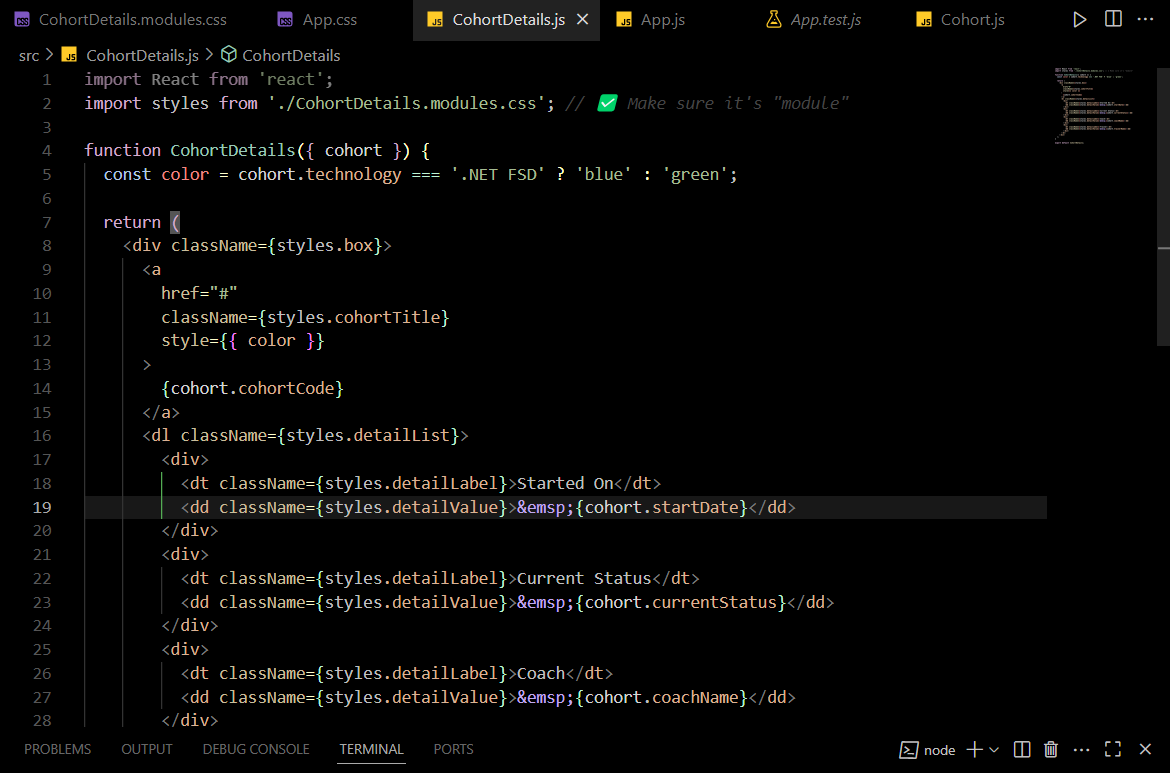
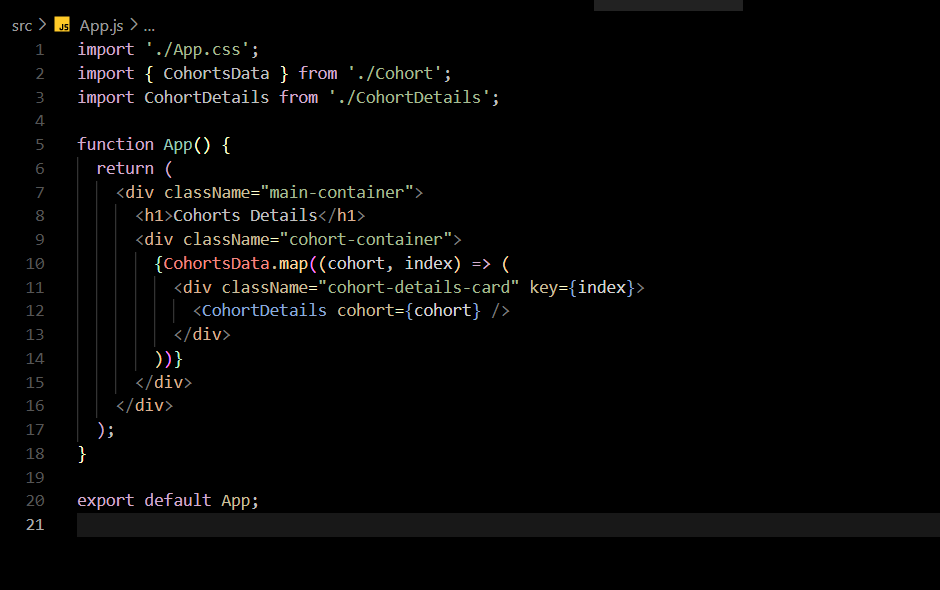
This approach reinforces responsive design principles and supports future extensibility.

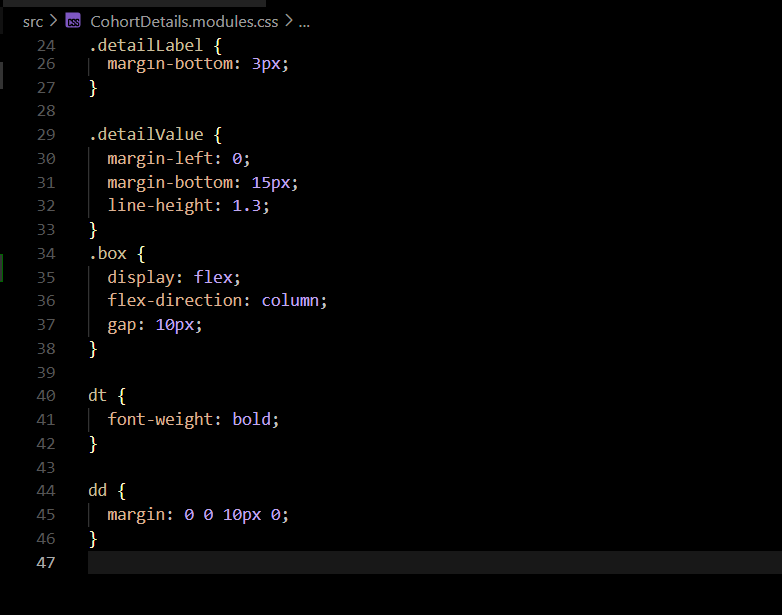
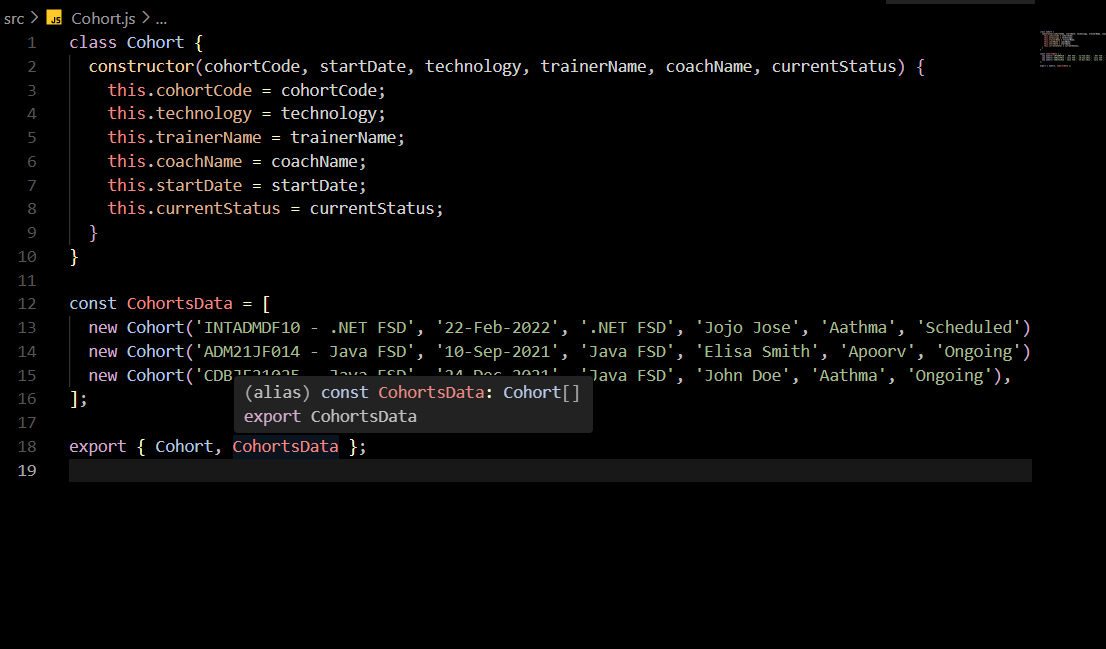
Overall, this implementation reflects a well-structured, component-driven approach to UI development in React.

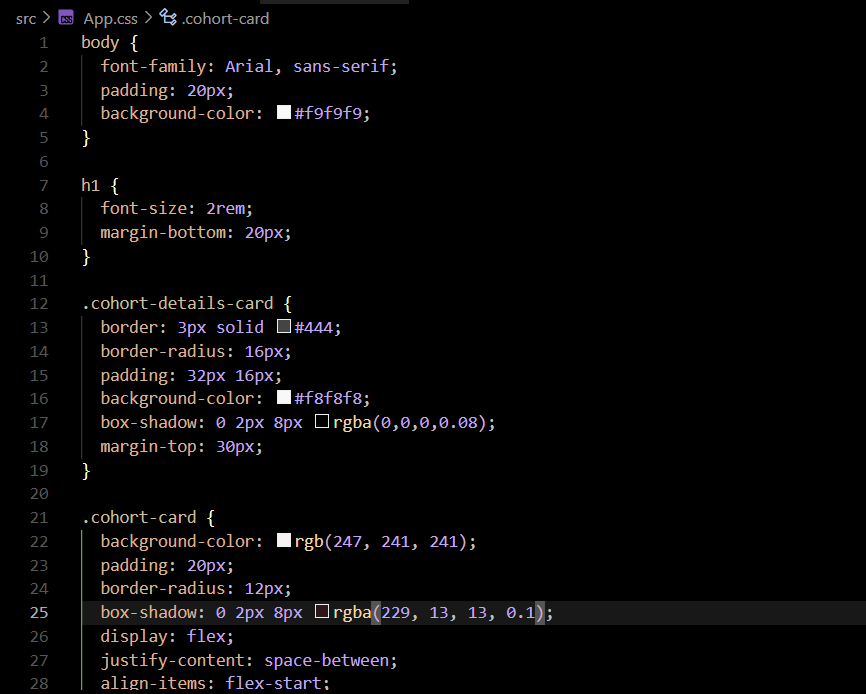
It demonstrates not only technical accuracy in terms of data handling, component communication, and styling scope but also thoughtful choices around semantic HTML and accessibility.

This hands-on experience has strengthened my understanding of **React component lifecycle, presentational vs. logical separation of concerns, semantic UI design, and modular CSS architecture**—all of which are essential skills for modern front-end development.

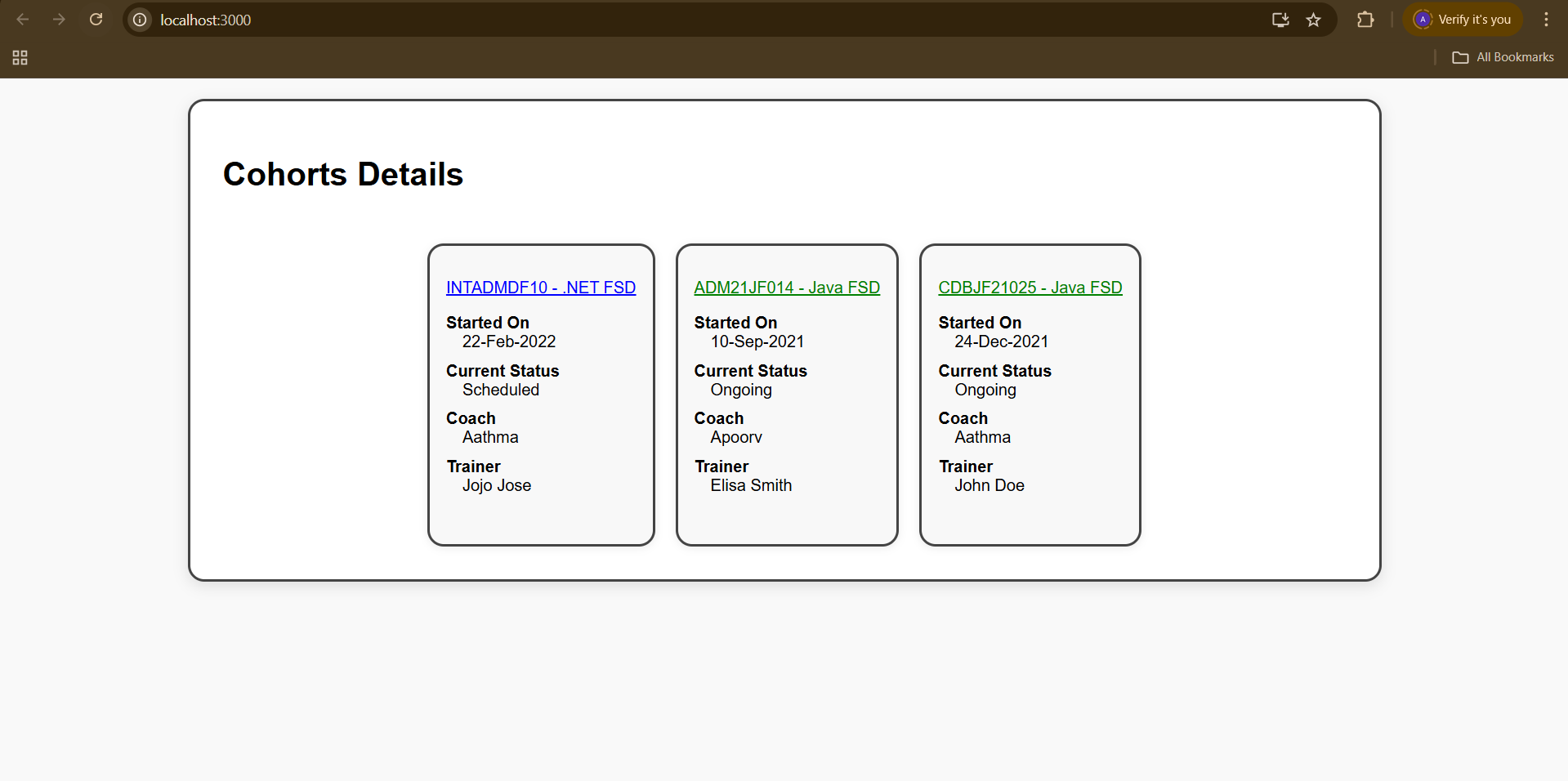
I tried that my output closely matches the final result this handson expects.Overall,The Output matches the requirements successfully.



**OUTPUT:**

****