**Experiment 02**

**AIM :**

To develop a React application that demonstrates the working of React hooks such as useEffect, useContext, and custom hooks, in order to understand state management, side-effect handling, and code reusability in functional components.

**THEORY :**

React Hooks as a Paradigm Shift

* Hooks were introduced in React 16.8 to overcome the limitations of class components, such as complicated lifecycle methods and difficulty in reusing logic.
* They offer a simpler and more functional approach to managing component behavior, making components easier to read, maintain, and reuse.

Rules of Hooks

* Hooks must always start with the keyword use, so React can properly identify and track them.
* They must be called at the top level of a functional component — not inside loops, conditions, or nested functions.
* Hooks can also be used inside custom hooks, allowing developers to create complex reusable logic.

useEffect Hook (Side Effects Management)

* React’s rendering process is pure, meaning it should only describe what the UI looks like, not perform side effects.
* Side effects include tasks such as:
  + Fetching data from an API
  + Subscribing to a WebSocket
  + Updating the document title
  + Setting or clearing timers
* The cleanup function inside useEffect ensures proper resource management by:
  + Removing event listeners
  + Canceling subscriptions
  + Clearing timers when a component unmounts
* The dependency array [ ] controls when the effect runs:
  + [] → Runs only once (like componentDidMount)
  + No array → Runs after every render
  + [values] → Runs only when the specified values change

useContext Hook (Global State Sharing)

* Traditionally, React components had to pass data through props, even to deeply nested children (known as prop drilling).
* The useContext hook solves this by allowing components to directly access shared data from a context provider.
* It’s commonly used for:
  + Theme management
  + Language or localization settings
  + Authentication data
  + Application-wide configurations
* Unlike external libraries like Redux, useContext is lightweight and built directly into React.

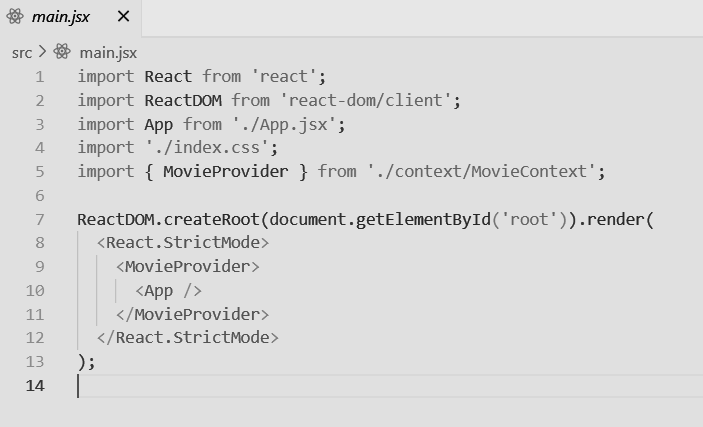
Custom Hooks (Reusable Stateful Logic)

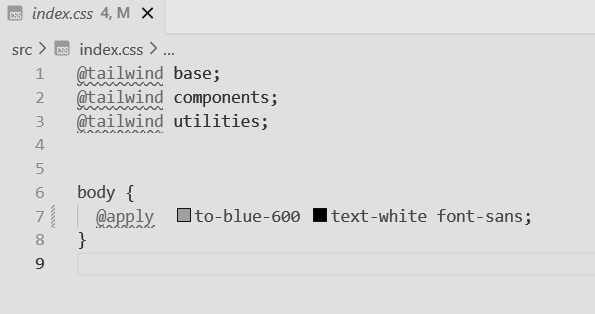
* A custom hook is simply a function that uses other hooks to encapsulate reusable logic.
* Example: Instead of writing API-fetching logic in multiple components, you can create a reusable useFetch hook.
* Custom hooks promote separation of concerns:
  + UI logic remains in the component
  + Stateful logic is abstracted into the hook
* They can also return multiple values, making them flexible and powerful for various use cases.

Advantages of Hooks in Modern React Development

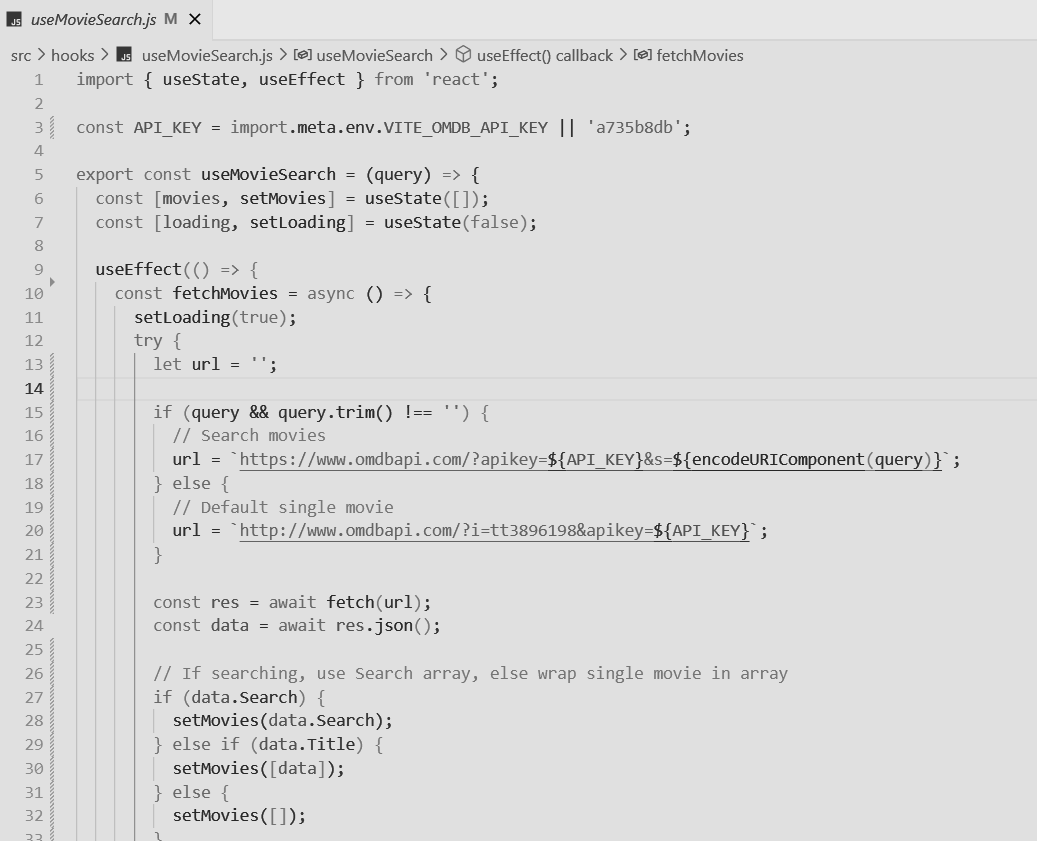
* Simplify code by eliminating unnecessary boilerplate from class components.
* Enhance readability and make large codebases easier to maintain.
* Encourage functional programming and promote modular code design.
* Improve testability — since hooks are just functions, they can be tested independently.
* Unify state and side-effect management, removing the need for class lifecycle methods like:
  + componentDidMount
  + componentDidUpdate
  + componentWillUnmount

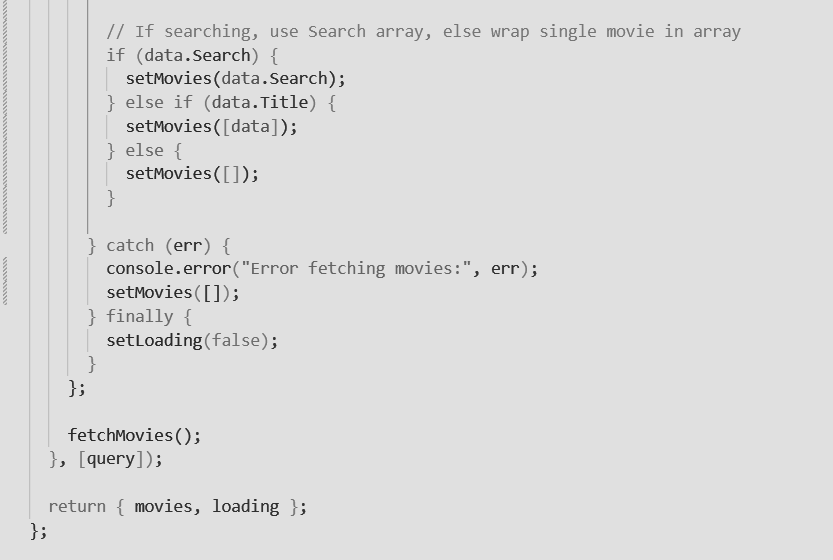
**SOURCE CODE :**





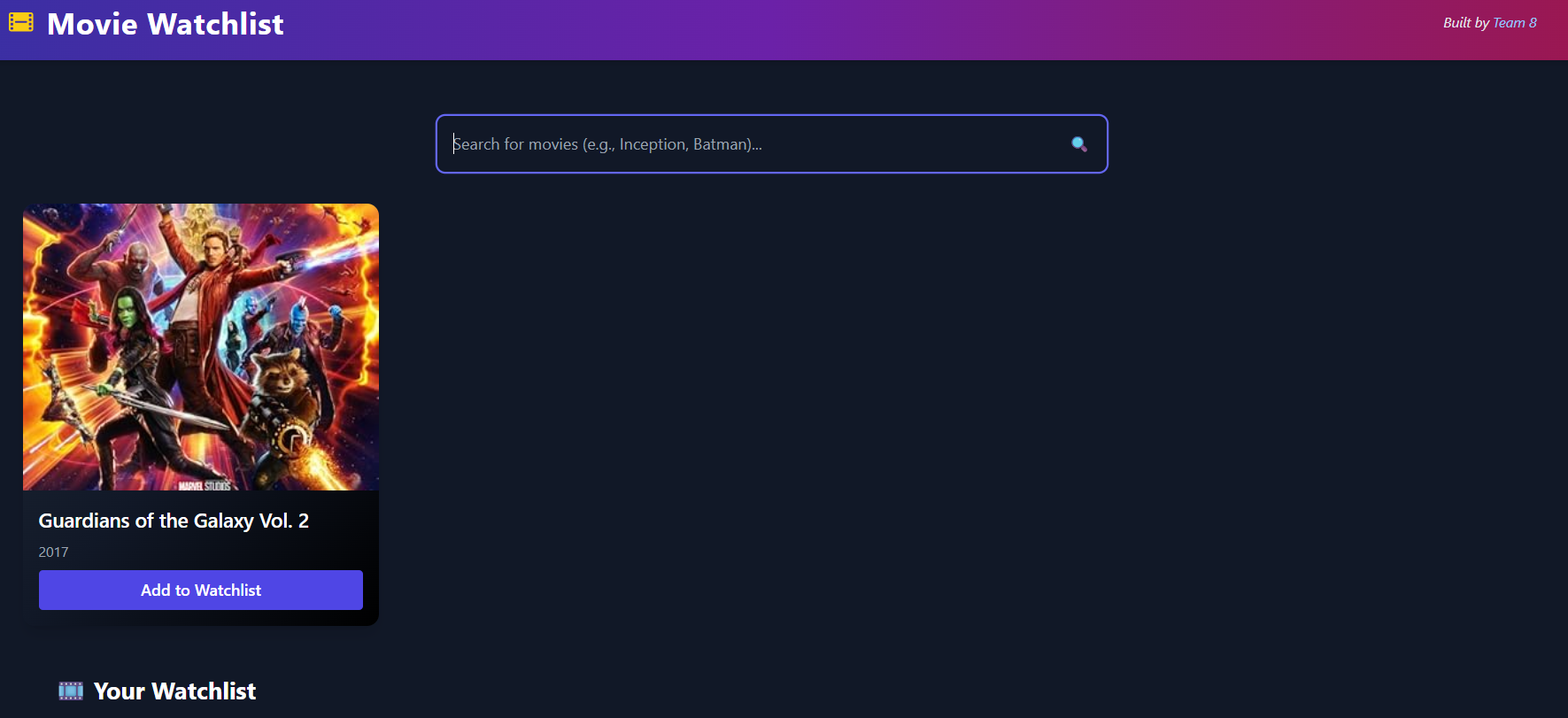


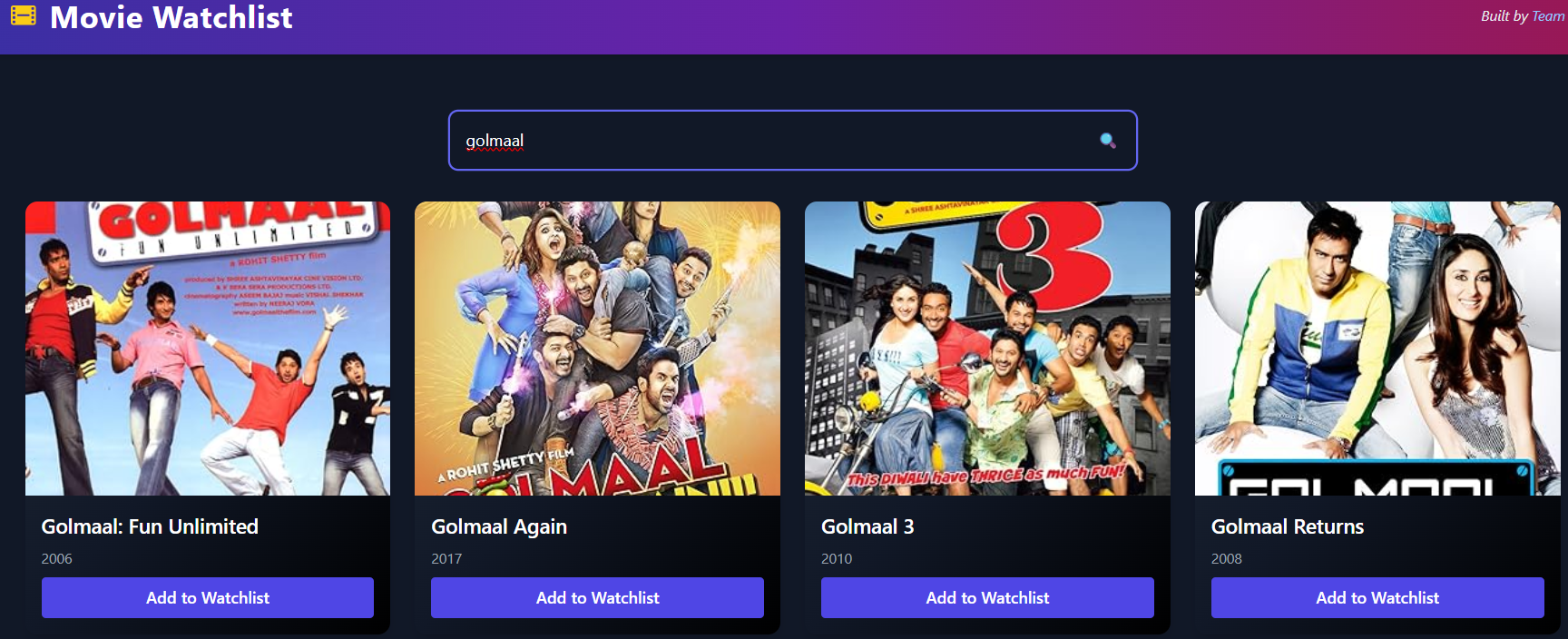


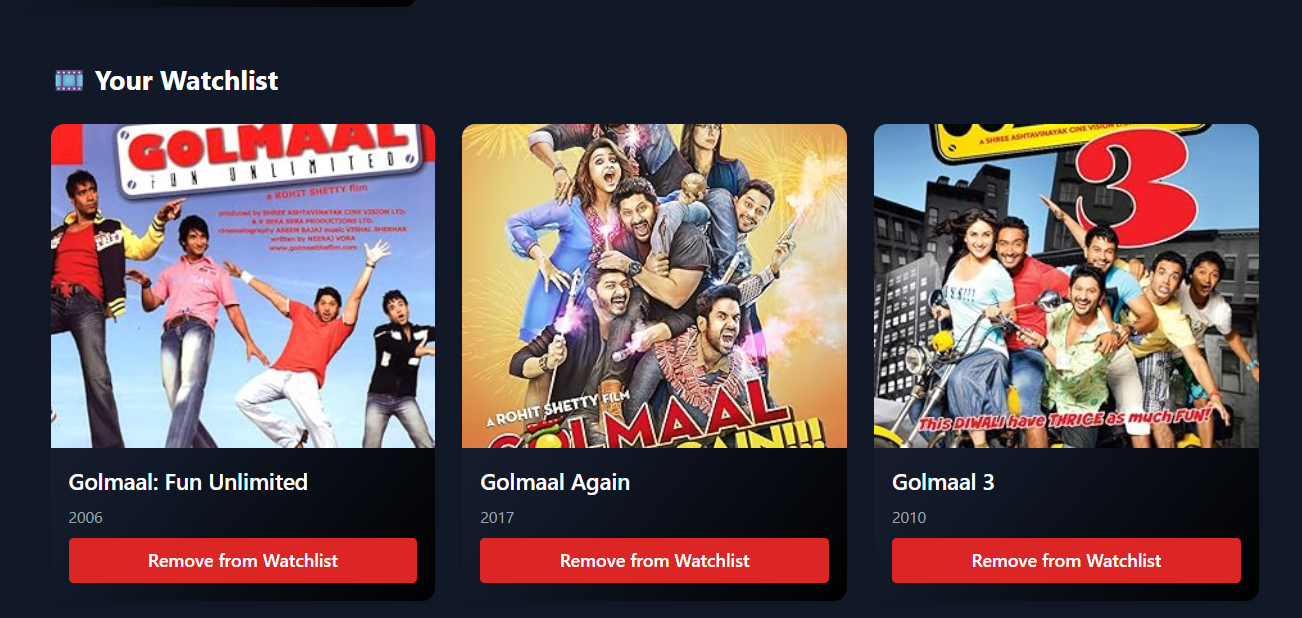




**OUTPUT :**

****

****

****

**CONCLUSION:**

In this experiment, we successfully implemented and analyzed the working of three important React Hooks: useEffect, useContext, and custom hooks. The use of useEffect helped in handling side effects such as fetching external data, useContext simplified global state sharing without the complexity of prop drilling, and custom hooks demonstrated the ability to encapsulate and reuse logic efficiently.

Thus, React Hooks not only reduce code complexity but also make applications more modular, maintainable, and scalable. Their usage reflects modern React development practices and provides a foundation for building real-world applications with better performance and cleaner architecture.