Batch: A1	**Roll No.:** 16010123012
Experiment / assignment / tutorial No.: 3.2	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Implementation of React Hooks.

AIM: To Implement the React Hooks

Problem Definition:

To demonstrate the working of react hooks based on the following points and Apply this on assigned programming task

- State in React
- useState
- useEffect
- useContext
- useReducer
- useCallback
- useMemo

(List is based commonly used hooks beyond this you can explore and add atleast 2 Hooks)

*(Students have to perform the task assigned within group/Individual and demonstrate the same).

Resources used:

https://react.dev/

Expected OUTCOME of Experiment:

CO 1: Build full stack applications in JavaScript using the MERN technologies.

CO5: Deploy MERN applications to cloud platforms like Heroku or AWS and collaborate using GitHub version control.

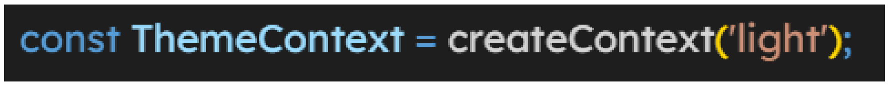
Books/ Journals/ Websites referred:

1. Shelly Powers Learning Node O' Reilly 2 nd Edition, 2016.

Pre Lab/ Prior Concepts:

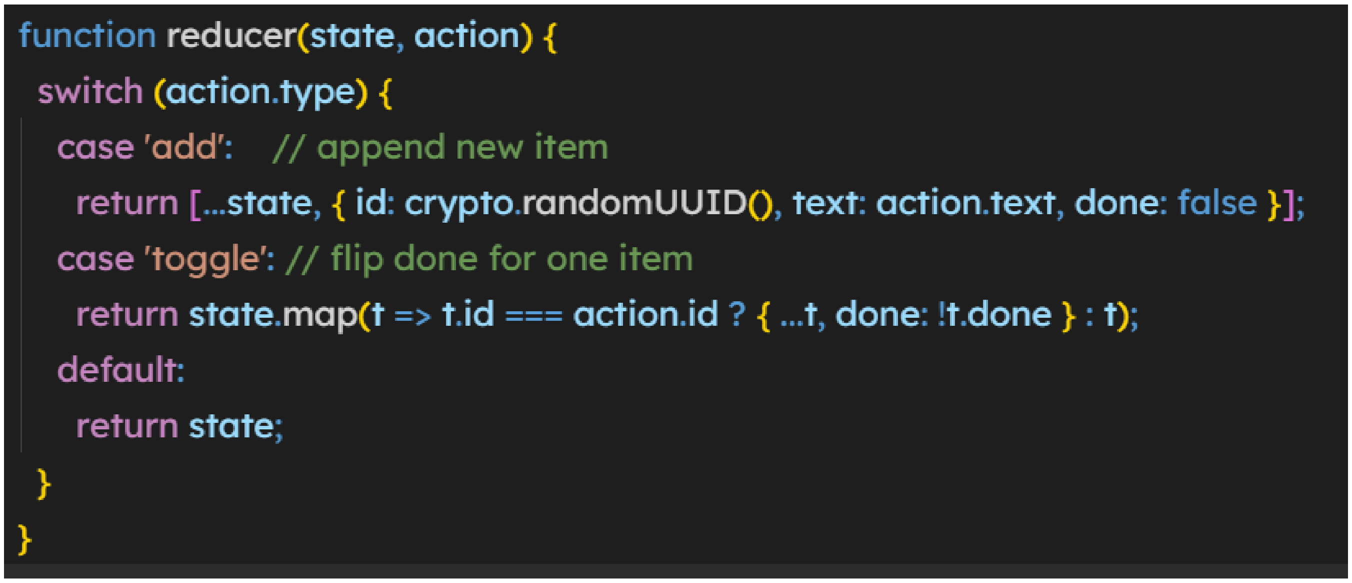
Write details about the following content

- **useState:** Local component state. Triggers re-render on set.



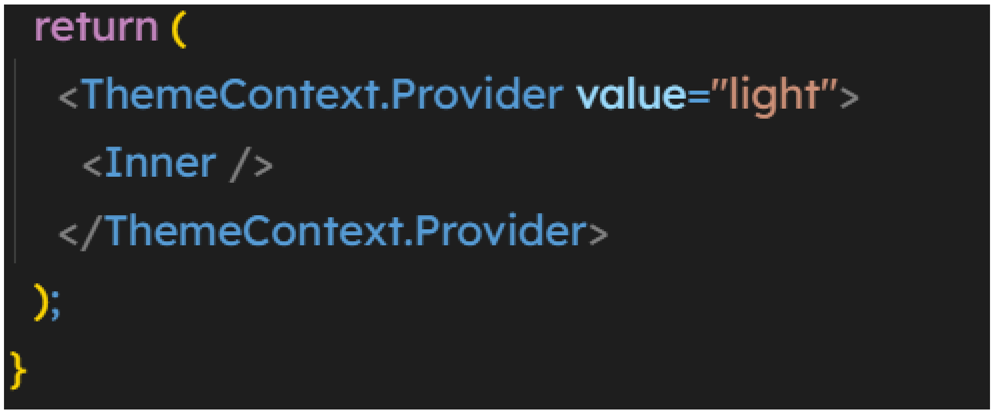
```
const ThemeContext = createContext('light');
```

- **useEffect:** Runs side-effects after render. Can clean up.



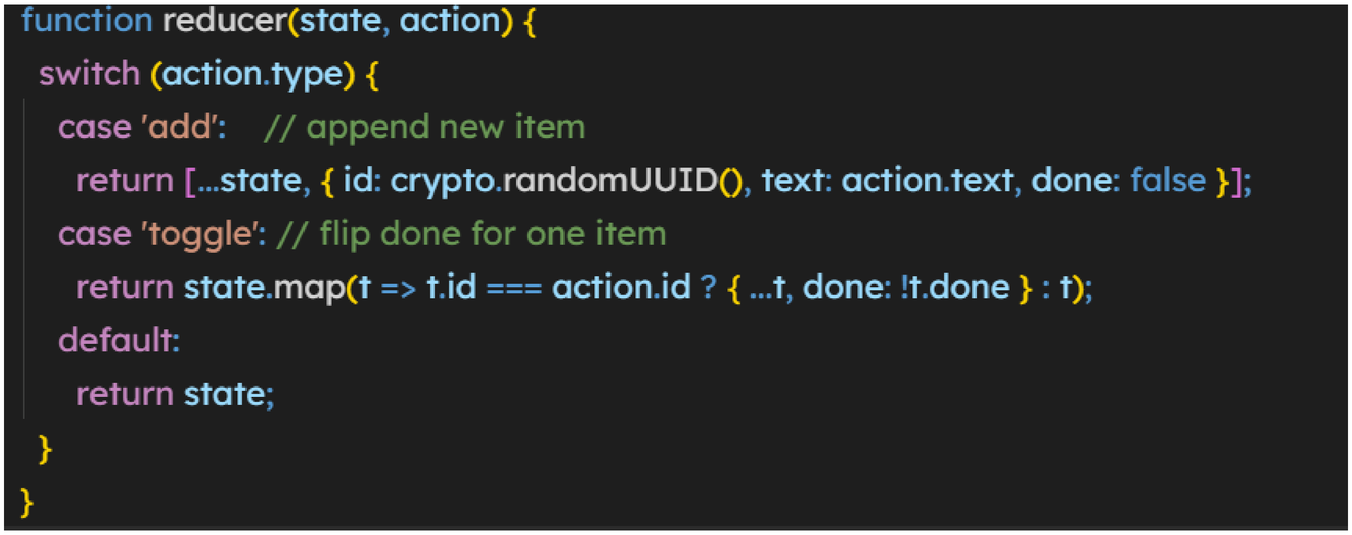
```
function reducer(state, action) {
  switch (action.type) {
    case 'add':    // append new item
      return [...state, { id: crypto.randomUUID(), text: action.text, done: false }];
    case 'toggle': // flip done for one item
      return state.map(t => t.id === action.id ? { ...t, done: !t.done } : t);
    default:
      return state;
  }
}
```

- **useContext:** Read shared value without prop drilling.



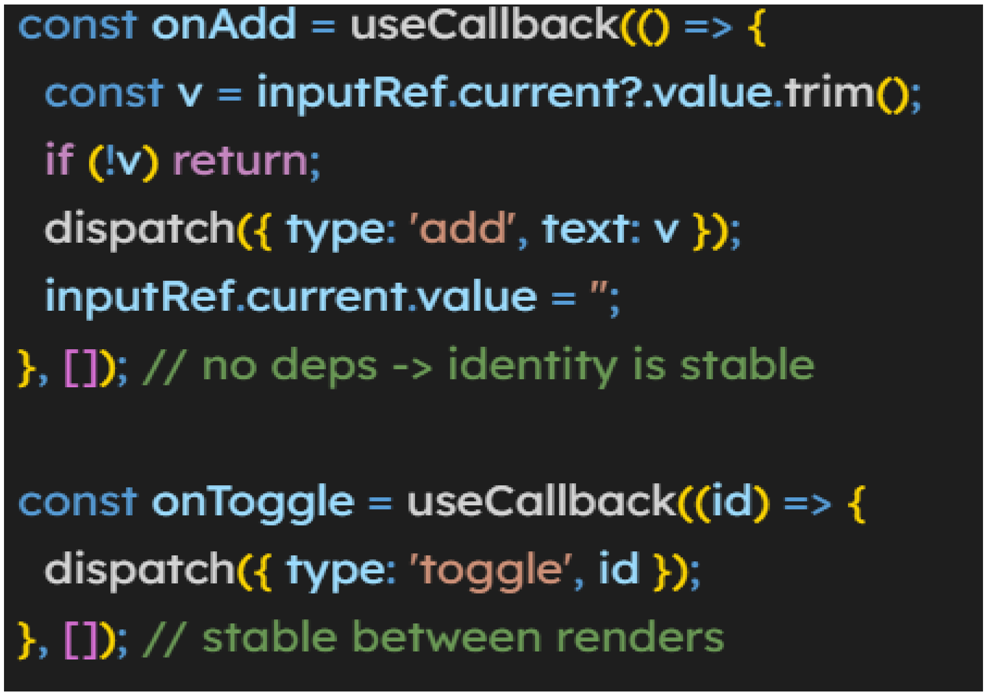
```
  return (
    <ThemeContext.Provider value="light">
      <Inner />
    </ThemeContext.Provider>
  );
}
```

- **useReducer:** State transitions via actions. Good for collections.



```
function reducer(state, action) {
  switch (action.type) {
    case 'add':    // append new item
      return [...state, { id: crypto.randomUUID(), text: action.text, done: false }];
    case 'toggle': // flip done for one item
      return state.map(t => t.id === action.id ? { ...t, done: !t.done } : t);
    default:
      return state;
  }
}
```

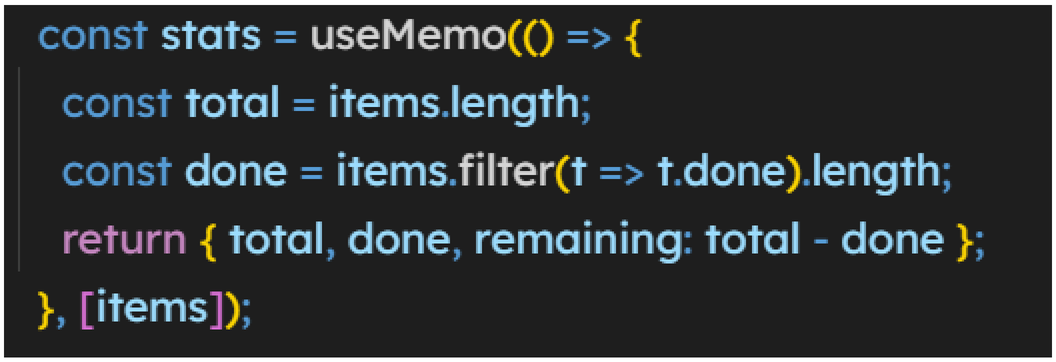
- **useCallback:** Memoize function identity to avoid re-renders of memoized children.



```
const onAdd = useCallback(() => {
  const v = inputRef.current?.value.trim();
  if (!v) return;
  dispatch({ type: 'add', text: v });
  inputRef.current.value = '';
}, []); // no deps -> identity is stable

const onToggle = useCallback((id) => {
  dispatch({ type: 'toggle', id });
}, []); // stable between renders
```

- **useMemo:** Memoize derived values until dependencies change.



```
const stats = useMemo(() => {
  const total = items.length;
  const done = items.filter(t => t.done).length;
  return { total, done, remaining: total - done };
}, [items]);
```

Implementation Details:

1. **Scaffold**

- npm create vite@latest hooks-demo -- --template react
- cd hooks-demo && npm i

2. **Context**

- Create ThemeContext = createContext('light').
- Wrap <Inner /> with <ThemeContext.Provider value="light"> in App.

3. **Reducer**

- Write reducer(state, action) with add and toggle.
- const [items, dispatch] = useReducer(reducer, []).

4. **Local state and effects**

- const [text, setText] = useState('').
- useEffect(() => setText('hello'), []) to show one-time update.
- useEffect(() => { const id = setInterval(()=>console.log('tick'),1000); return ()=>clearInterval(id); }, []) to prove cleanup.

5. **Refs and a11y**

- const inputRef = useRef(null); const inputId = useId();

- useEffect(() => inputRef.current?.focus(), []) for autofocus.

6. **Callbacks**

- onAdd = useCallback(() => { /* read inputRef, dispatch add */ }, []).
- onToggle = useCallback((id) => dispatch({ type:'toggle', id }), []).

7. **Memoized derivations**

- stats = useMemo(() => ({ total, done, remaining }), [items]).

8. **Memoized child**

- const TodoItem = React.memo(function TodoItem({ t, onToggle }) { /* ... */ }).
- Render with {items.map(t => <TodoItem key={t.id} t={t} onToggle={onToggle} />)}.

9. **UI**

- Show text and theme.
- Input + Add button.
- List with checkbox per item.
- Counters from stats.

Steps for execution:

1. Run dev
   npm run dev
   Open the shown URL.
2. Verify outputs

- Initial: text: hello | theme: light. Input focused. Console prints tick every second.
- Add “Buy milk” then “Pay bills”. Counters: Total: 2 | Done: 0 | Remaining: 2.
- Toggle first item. Strikethrough shows. Counters: Done: 1 | Remaining: 1.

- If instrumentation is on, toggling re-renders only that item. compute stats increments only on add/toggle.

3. Build
   npm run build
   npm run preview

Conclusion:
In this experiment we successfully implemented and demonstrated the use of core React Hooks within a sample application. The UI behavior and console outputs matched the expected results, confirming that React Hooks enable predictable, testable, and efficient component logic without using class components.

Postlab questions:
1) Differential state and Props
In React, state refers to data that a component owns and manages internally. It is mutable, meaning it can be updated within the component using hooks like useState or useReducer, and any change in state automatically triggers a re-render of that component. State is ideal for handling dynamic data that evolves due to user interactions or API responses.
On the other hand, props (short for properties) are read-only inputs passed from a parent component to a child. A child component cannot directly modify its props; if changes are needed, the parent must supply new values. While state is local and private to a component, props are external and allow data and configurations to flow from parent to child components. In short, state is internal and mutable, whereas props are external and immutable, but both work together to make components interactive and reusable.