St Francis Institute of Technology

(Autonomous Institute)

Department of Artificial Intelligence and Machine Learning

Second Year AIML Engineering (SEM-IV A.Y. 2025-26)

Web Programming Lab. Experiment Report

Experiment 8: Implement state management in React applications.

1. AIM

To implement React State Management using the useState Hook by developing:

- **Part A:** A Login Form with controlled components and dynamic data display
- **Part B:** A Counter Application using useState along with a controlled input form

2. LAB OBJECTIVE

- To understand React state and useState
- To implement controlled components
- To manage multiple inputs using a single state object
- To validate user input dynamically
- To build an interactive counter application
- To observe automatic re-rendering when state updates

3. LAB OUTCOME

After completing this experiment, students will be able to:

- Implement controlled forms in React
- Manage form data using a single state object
- Perform client-side validation
- Build a counter using useState
- Understand how state updates trigger re-rendering

4. PREREQUISITE

- HTML, CSS basics
- JavaScript (functions, objects, events)
- ES6 syntax
- Basic React components
- Props concept

5. THEORY

5.1 React State

State is a built-in React object used to store dynamic data in a component.

- When state changes → Component re-renders
- State makes applications interactive
- Managed using Hooks in functional components

• Password strength validation

5.2 useState Hook

Syntax: const [state, setState] = useState(initialValue);

- state → current value
- setState → function to update state
- initialValue → default value

5.3 Controlled Components

A controlled component is an input field controlled by React state.

```
<input value={stateValue} onChange={handleChange} />
```

Benefits:

- Predictable data flow
- Easier validation
- Single source of truth

5.4 Single State Object

Instead of multiple state variables:

```
const [username, setUsername] = useState("");
const [email, setEmail] = useState("");
```

Use:

```
const [formData, setFormData] = useState({
  username: "",
  email: "",
  password: ""
});
```

Advantages:

- Cleaner code
- Scalable structure
- Easier management

5.5 Form Validation

Validation ensures:

- Fields are not empty
- Email format is correct

- Password length is sufficient

Errors are stored in a separate state object and displayed dynamically.

6. PROCEDURE

Part A: Login Form with Controlled Components & Validation

Step 1: Create and Set Up React Application

1. Create a new React application using the appropriate React setup command.
2. Navigate into the project folder.
3. Start the development server.
4. Open the project in a code editor.
5. Clean unnecessary default files (logo, extra CSS, etc.) if required.

Step 2: Create a Login Form Component

1. Create a new functional component named **LoginForm**.
2. Ensure the component is exported properly.
3. Import and render the LoginForm component inside the main App component.

Step 3: Initialize State Using useState

1. Import the useState Hook from React.
2. Create a **single state object** to store:
   - Username
   - Email
   - Password
3. Initialize all fields with empty strings.
4. Create another state object to store validation errors.

Purpose:

- One state object → form data
- One state object → error messages

Step 4: Design the Form Structure

1. Create a form element.
2. Inside the form, create:
   - A text input for Username
   - A text input for Email
   - A password input for Password

    - A Submit button
3. Assign a unique name attribute to each input field.
4. Display validation error messages below each field.

Step 5: Implement Controlled Components

1. Connect each input field’s value to the corresponding property in the state object.
2. Create a single change handler function.
3. In the change handler:
    - Capture the input field name and value.
    - Update only the specific field inside the state object.
    - Use the spread operator logic to avoid overwriting other fields.

Outcome:

- Inputs become controlled by React state.
- Any change updates state immediately.

Step 6: Implement Validation Logic

1. Create a separate validation function.
2. Inside the function:
    - Check if username is empty.
    - Check if email is empty.
    - Verify email contains proper format (example: contains "@").
    - Check if password is empty.
    - Ensure password has minimum length (e.g., 6 characters).
3. Store all validation messages inside the error state object.
4. Return the errors from the validation function.

Step 7: Handle Form Submission

1. Create a submit handler function.
2. Prevent default form submission behavior.
3. Call the validation function.
4. If validation errors exist:
    - Update the error state.
    - Stop submission.
5. If no errors:
    - Clear error messages.
    - Display success message.
    - Optionally reset form fields.

Step 8: Display Dynamic User Input

1. Below the form, display:
   - Entered Username
   - Entered Email
2. Bind displayed values directly from the state object.
3. Observe how the data updates automatically when user types.

Step 9: Test the Application

verify:

- Empty form shows required field errors.
- Invalid email shows format error.
- Short password shows length error.
- Valid form displays success message.
- Entered data appears dynamically below the form.
- No page reload occurs during submission.

Part B: Counter App using useState (with Controlled Input)

Step 1: Create the Counter Component

1. Create a new functional component named **Counter**.
2. Export the component properly.
3. Import and render the Counter component inside the main App component.
4. Confirm that the application runs without errors.

Step 2: Import useState Hook

1. Import the useState Hook from React.
2. Understand that useState will allow the component to store and update dynamic values.

Step 3: Initialize State Variables

Create two separate state variables:

1. **Counter State**
   - Initialize with value 0.
   - This will store the current counter value.
2. **Step State**

    - Initialize with value 1.
    - This will control how much the counter increases or decreases.

Purpose:

- count → Stores the current number displayed.
- step → Stores the increment/decrement value entered by user.

Step 4: Design the User Interface

Inside the component:

1. Display a heading such as **“Counter App”**.
2. Create an input field for entering the step value.
3. Display the current counter value.
4. Create three buttons:
    - Increase
    - Decrease
    - Reset

Ensure the layout is clear and readable.

Step 5: Implement Controlled Input for Step Value

1. Connect the step input field to the step state.
2. Make the input field controlled by:
    - Setting its value from state.
    - Updating state whenever user changes the input.
3. Convert input value to a number before storing it in state.
4. Ensure invalid or empty inputs are handled properly.

Outcome:

- Step value updates dynamically.
- Input is fully controlled by React state.

Step 6: Implement Increase Functionality

1. Attach a click event to the **Increase button**.
2. When clicked:
    - Add the current step value to the current counter value.
    - Update the counter state.
3. Observe that the displayed number updates automatically.

Step 7: Implement Decrease Functionality

1. Attach a click event to the **Decrease button**.
2. When clicked:
   - Subtract the step value from the counter value.
   - Update the counter state.
3. Verify that the UI updates immediately.

Step 8: Implement Reset Functionality

1. Attach a click event to the **Reset button**.
2. When clicked:
   - Set the counter value back to 0.
3. Confirm that the display resets correctly.

Step 9: Observe State Re-rendering

Students should observe:

- When counter state changes → Component re-renders.
- When step value changes → Future increments use updated step.
- No page reload occurs.
- UI updates instantly.

Step 10: Test Different Scenarios

Students should test:

- Step value = 1
- Step value = 5
- Step value = negative number
- Step value = 0
- Reset after multiple operations

Verify that:

- Counter behaves correctly.
- Controlled input works as expected.
- State updates are consistent.

---

7. RESULTS / OUTCOME EXPECTED

- How does the counter increment and decrement functionality work in this experiment?
- How is the numeric step input implemented as a controlled component in the counter application?
- What happens in the React component when the counter state changes? Explain the rendering behavior.
- How does the reset button restore the counter to its initial value?

- Explain how multiple state variables are managed efficiently using the useState Hook in this experiment.

8. CONCLUSION

- What is the role of useState in this experiment?
- How do controlled components improve form handling?
- Why is a single state object used in the login form?
- How does React know when to re-render the counter?
- What happens if state is updated incorrectly?

9. POST-EXPERIMENT QUESTIONS

- Differentiate between props and state.
- What is the advantage of controlled components?
- Why do we use the spread operator while updating state?
- Can multiple useState Hooks be used in one component?
- How can this experiment be extended further?

10. REFERENCES

- React Official Documentation – https://react.dev
- MDN JavaScript Documentation
- W3Schools React Tutorial
- *Learning React* – Alex Banks