LAB 04

NAME: Trương Mạnh Nguyên

Student CODE: 23521065

Link GitHUB: https://github.com/AdeptCodee/UIT-WebProject/tree/main/LAB4

Section 1: The useEffect Hook

1.1. Conceptual Questions: The Effect Lifecycle

1. useEffect is used to run logic outside of React's rendering process, such as calling API, manipulating DOM, using timer, connecting WebSocket... React doesn't do these things by itself, so useEffect is like an "escape hatch" for components to synchronize with "external systems".

2. Effect has two main phases:
    a. **Setup** → React runs the logic in the body of useEffect.
    b. **Cleanup** → React runs the function that we return inside useEffect to stop synchronization (remove event, cancel timer...).

    React will run setup → cleanup → setup… every time dependencies change.

    Different from class component:

    - Class uses separate functions: componentDidMount, componentDidUpdate, componentWillUnmount.
    - useEffect combines everything in one place → easier to read because setup and cleanup are next to each other.

3. React's Strict Mode

    Strict Mode intentionally runs the effect twice to check if the component has unsafe side effects.

    Common types of bugs found:

    Effects based on stale values.

    Incomplete cleanup → causing memory leaks.

    Non-idempotent logic (running twice will cause errors… e.g. calling the API twice).

So, React runs twice to make sure its code is safe and does not depend on “running only once”.

1.2. Practical Exercise: Dependency Arrays

1. Scenario A: You want to fetch user data from an API, but only once when the component first mounts.

   ➔ useEffect(() => {

   fetchUser();

   }, []);

   Explain: Leaving the array empty helps the effect run only once when the component mounts.

2. Scenario B: You have a Timer component that needs to re-run its logic every time the component re-renders. (Note: This is rarely correct, but a valid scenario to understand).

   ➔ useEffect(() => {

   startTimer();

   });

   Explain: Not passing a dependency array means the effect will run after every render.

3. Scenario C: You have a UserProfile component that needs to fetch new data only when the userId prop changes.

   ➔ useEffect(() => {

   fetchUser(userId);

   }, [userId]);

   Explain: Effect only needs to run again if userId is different → fetch new data.

1.3. gitHUB

Section 2: The useRef Hook

2.1. gitHUB

2.2. Conceptual Questions: useRef vs. useState

1. What are the two primary differences between useRef and useState?

The two main differences are:

1. **Re-render behavior:**
    - useState triggers a re-render when its value changes.
    - useRef does *not* cause a re-render when its .current value changes.
2. **Update type (sync vs async):**
    - useState updates are **asynchronous**.
    - useRef updates are **synchronous** because it just changes an object property.

2. Why is useRef the correct tool for storing a setInterval ID?

A setInterval ID does not affect the UI. It only needs to be stored so we can clear it later.
If we use useState, every update would cause an unnecessary re-render, which is wasteful.
With useRef, we can store the interval ID without triggering re-renders, making it more efficient.

3. What is the "cardinal rule" for choosing between useState and useRef?

Use **useState** when the value is part of the rendering output (it affects what the UI shows).
Use **useRef** when the value does *not* impact the UI and you only need to store it between renders.

Section 3: Data Fetching Strategies

3.1. Conceptual Questions: fetch vs. axios

1. Describe two key differences between the native fetch API and the axios library (focus on JSON parsing and error handling).

Difference 1 — JSON Parsing

- fetch: does *not* automatically parse JSON. You must manually call response.json(), which returns a Promise.
- axios: automatically parses JSON. The parsed result is available directly in response.data.

Difference 2 — Error Handling

- fetch: does *not* throw an error for HTTP status codes like 404 or 500. It only throws errors for network failures.
- axios: automatically throws an error for any HTTP status outside the 200–299 range, so catch() will run on 4xx and 5xx responses.

2. With the fetch API, a 404 "Not Found" error does not trigger .catch(). How must you manually check for HTTP errors when using fetch?

You must manually check response.ok, and if it is false, you must throw your own error.

Example:

```
fetch(url)
  .then(response => {
    if (!response.ok) {
      throw new Error("HTTP error: " + response.status);
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

Explanation:

- response.ok is true for status codes 200–299.
- For 404, 500, etc., response.ok is false, so you must throw the error yourself for .catch() to run.

3.2. Practical Exercise: Loading and Error States

gitHUB

Section 4: Architecting Forms

4.1. gitHUB

4.2. Conceptual Questions: Controlled vs. Uncontrolled

1. What is the "single source of truth" for an input's value in a Controlled Component?

-> is React state.

2. What is the "single source of truth" for an input's value in an Uncontrolled Component?

-> is the DOM input element itself (its internal value).

3. What is the modern, preferred way to get the values from an Uncontrolled form upon submission?

-> is using the FormData API during form submission.

Example:

const formData = new FormData(event.target);

4. List one pro and one con for using Controlled components.

Pro:
You get full control over the input, allowing validation, instant updates, conditional UI, and tighter form logic.

Con:
They cause more re-renders and require more code, which can become heavy for large forms.

Section 5: React Router v6

5.1 gitHUB

5.2 gitHUB

5.3. Conceptual Questions: Protected Routes

1. Describe the simple, common pattern for creating a <ProtectedRoute /> wrapper component using <Outlet /> and <Navigate />.

A common pattern is:

- Check if the user is authenticated (for example: const isLoggedIn = …).
- If the user **is logged in**, return <Outlet /> so the child route can render.
- If the user **is not logged in**, return <Navigate to="/login" /> to redirect them.

So the <ProtectedRoute /> component basically works like:

return isLoggedIn ? <Outlet /> : <Navigate to="/login" />;

2. What is the "race condition" problem this simple pattern creates when used with modern data loaders in React Router?

The problem is:

- **Loaders run before the element renders.**
- The simple <ProtectedRoute /> check happens **too late**.
- This means the loader for a protected page may still run **even when the user is not authenticated**, because the redirect happens only after React tries to render the route.

So the “race condition” is that:
→ **The loader can fetch protected data before the redirect happens**, causing a security and data-leak issue

Section 6: The Context API

6.1. Conceptual Questions: Prop Drilling

1. What is “prop drilling” and why is it considered a problem in React development?

Prop drilling is when you pass props through several layers of components just so a deeply nested component can use them.

It’s considered a problem because:

- It makes the code harder to maintain.
- Components in the middle receive props they don’t actually need.
- Changing data or structure becomes messy.

2. Context is not a “silver bullet.” For what type of state is Context a poor choice, and why? (Hint: performance)

Context is a poor choice for rapidly changing or high-frequency state (for example, typing state, animations, or data that updates very often).

Reason:
→ When context value changes, every component that uses that context re-renders, which can hurt performance.

6.2. gitHUB

Section 7: Custom Hooks

7.1. Conceptual Questions: Custom Hook Rules

1. What are the two mandatory naming conventions for a function to be considered a custom hook?

1. The function must start with the word use (e.g., useSomething).
2. It must follow camelCase naming (e.g., useMyHook, not UseMyHook or use_my_hook).

2. What is the key difference between sharing logic with a custom hook and sharing state? If two components call the same useCustomHook, do they share the same state variables?

The key difference is:

- A custom hook shares logic, not state.

If two components call the same useCustomHook:

→ No, they do NOT share the same state.
Each call creates its own separate state variables, so each component gets its own independent state instance.

7.2. gitHUB

Part 8: Capstone Project - "BlogDash"