# Experiment 2

## Aim

Experiment based on React Hooks (useEffect, useContext, custom hooks)

## Theory

### What & Why

- **Hooks** let function components manage **state**, **side effects**, and **shared logic** without classes. Standard practice: keep components “pure UI” and push effects/shared logic into hooks.

### `useEffect`

- **Purpose**: Run **side effects** after render (data fetch, subscriptions, DOM APIs), with optional **cleanup**.
- **Timing**: Runs after paint; cleanup runs before the next effect or on unmount.
- **Dependencies**: Array controls when it re-runs—`[]` (mount/cleanup only), `[a,b]` (when `a` or `b` change). Missing deps = bugs.
- **Best practice**:
  - Keep effects **idempotent** and **minimal**; avoid doing rendering logic in effects.
  - Move fetches/subscriptions into **custom hooks**; return status + data.
  - Prefer event handlers/derived values over effects when possible (effects are a last resort).
    Always **return a cleanup** for subscriptions/timers.

### `useContext`

- **Purpose**: Provide **global-ish, read-mostly** values down the tree (theme, auth, i18n) without prop drilling.
- **Mechanics**: `const value = useContext(MyContext)` reads from nearest `<MyContext.Provider value=...>`.
- **Best practice**:
  - Keep context **stable** (memoize provider value) to avoid re-renders.
  - **Split contexts** by concern (state vs dispatch) or use **selectors** to reduce updates.
  - Don’t use context as a general store for frequently changing granular state—consider **local state** or a dedicated state library.

### Custom Hooks

- **Purpose**: **Encapsulate reusable logic** (state + effects + context usage) behind a simple API. Naming: `useSomething`.
- **Design**:
    - Inputs are **parameters**, outputs are a **stable object** or tuple.
    - Hide implementation details; expose the minimum surface (data, booleans, callbacks).
    - Keep hooks **pure** (no conditional hook calls), and **testable** (isolate side effects).
- **Examples**: `useFetch`, `useToggle`, `useLocalStorage`, `useBreakpoint`, `useAuth`.

**Common Pitfalls (and fixes)**

- **Stale closures** in effects → list all deps or use functional updates.
- **Effect doing too much** → split by concern; one effect per side effect.
- **Context value recreated each render** → wrap in `useMemo`/`useCallback`.
- **Custom hook leaks** (intervals/listeners) → always provide **cleanup**.
- **Overusing effects** for derived state → derive during render or with `useMemo` instead.

**Standard way**: Prefer local state + render logic first; reach for `useEffect` only for true side effects, `useContext` for stable cross-cutting values, and **custom hooks** to package reusable behavior cleanly with proper dependencies and cleanups.

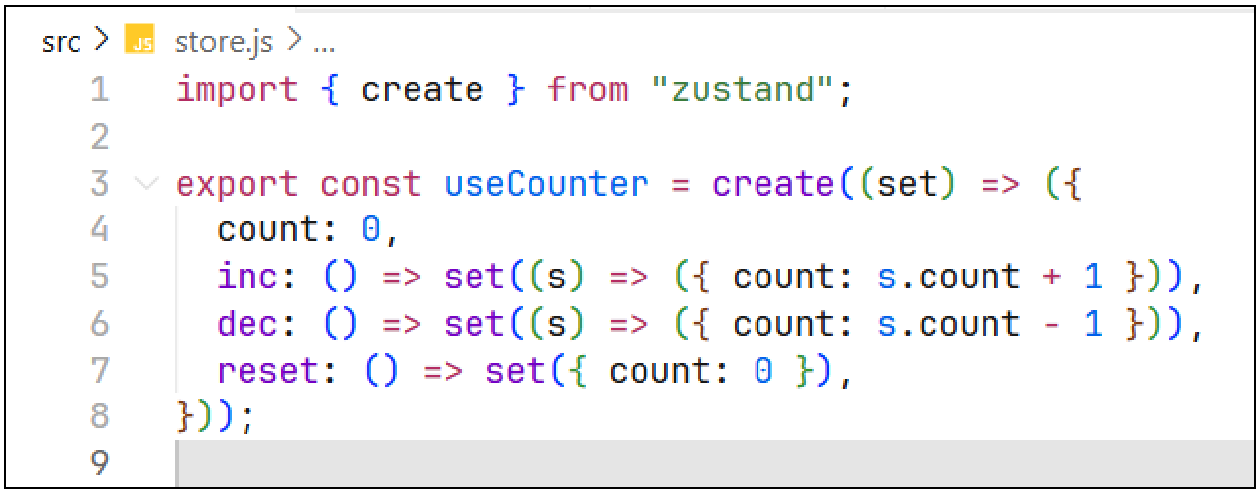
## 30% extra part

**Zustand**: lightweight state for React (vs Redux)

- **What**: Tiny state manager using hooks (`create`), no Provider or reducers needed. Components subscribe to **slices** via selectors.
- **Why better than Redux (for most apps)**:
    - **Near-zero boilerplate** (no actions/types/reducers).
    - **Selector-based subscriptions** → fewer re-renders by default.
    - **Mutable updates allowed** (or use `immer` middleware) with clean TypeScript inference.
    - **No context needed**; works across trees, portals, and outside React.
    - **Composability**: middlewares (`persist`, `immer`, `devtools`) added per store.

- **`shallow` / `useShallow`**: Optimize multiple-field selects. `shallow` compares object/array fields to avoid re-renders when values don’t change. (If your version exports `useShallow`, it’s a convenience wrapper for the same pattern.)
- **Server & async friendly**: Zustand stores can be used **outside React components**, which means you can update/read state in async functions, event handlers, or even on the server side — something Redux and Context patterns handle less cleanly.

**Source code**



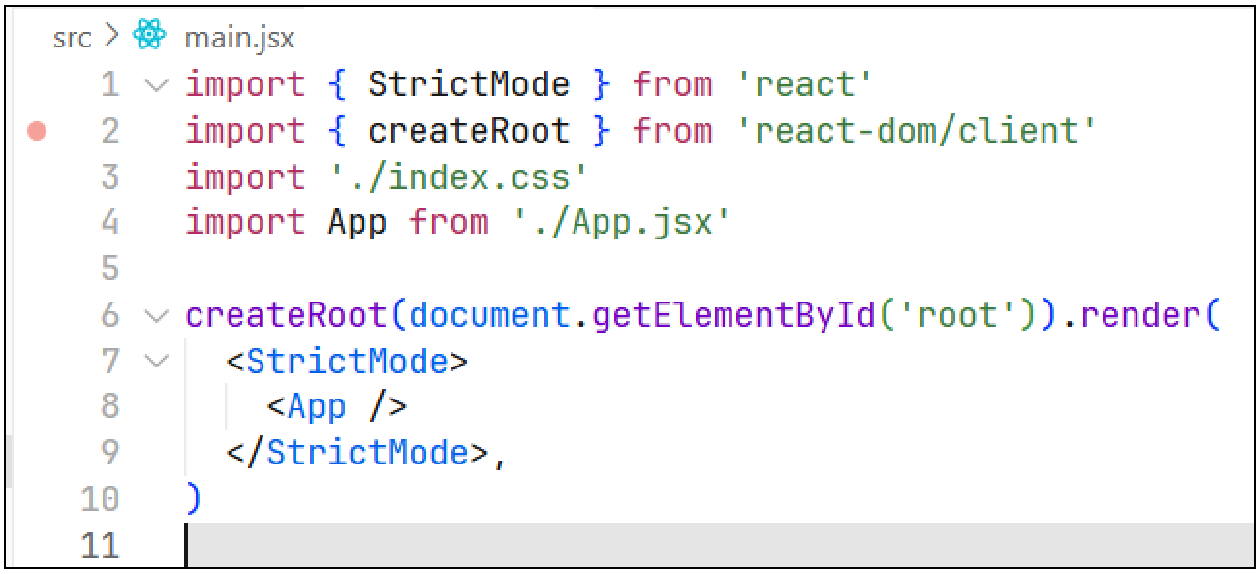
src > store.js > ...

```
import { create } from "zustand";

export const useCounter = create((set) => ({
  count: 0,
  inc: () => set((s) => ({ count: s.count + 1 })),
  dec: () => set((s) => ({ count: s.count - 1 })),
  reset: () => set({ count: 0 }),
}));

```

Fig 1.1



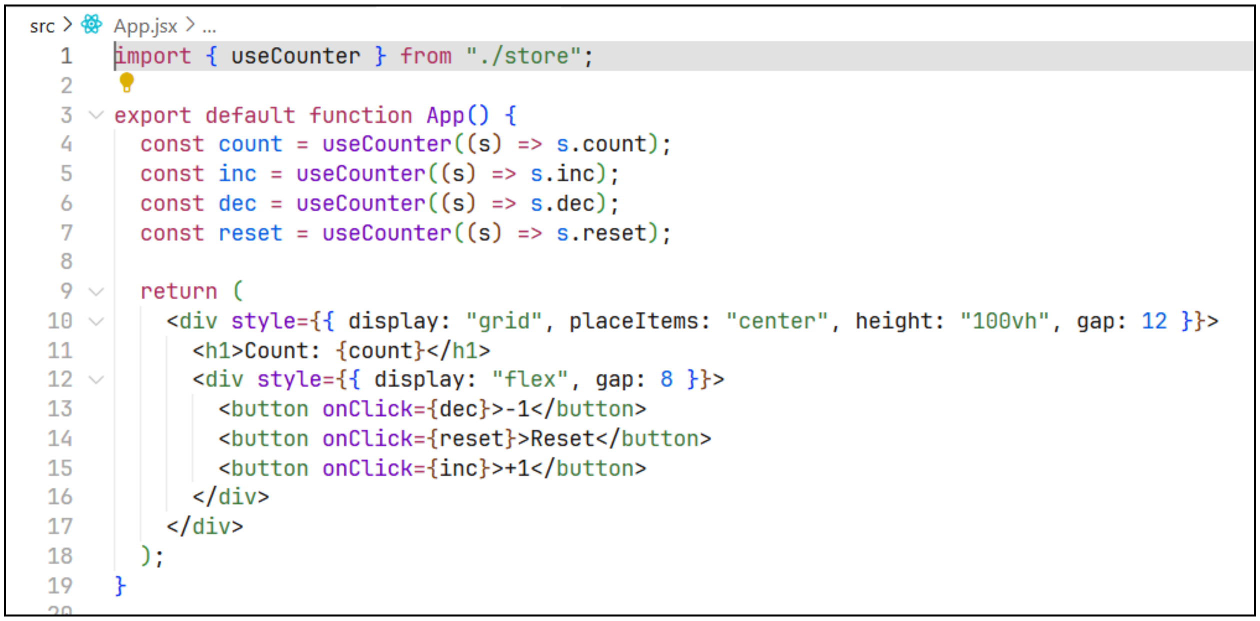
src > main.jsx

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)

```

Fig 1.2



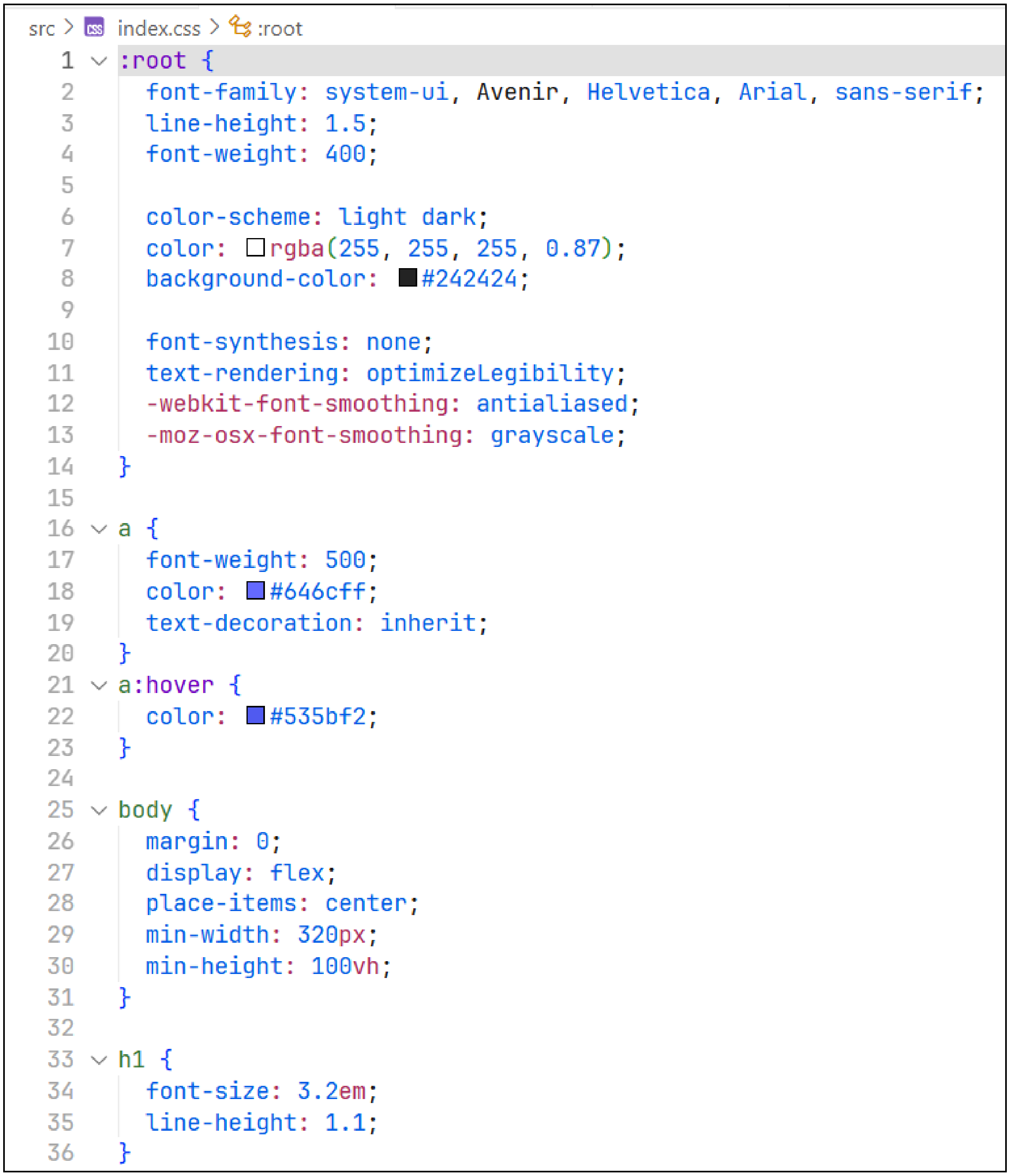
src > App.jsx > ...

```jsx
import { useCounter } from "./store";

export default function App() {
  const count = useCounter((s) => s.count);
  const inc = useCounter((s) => s.inc);
  const dec = useCounter((s) => s.dec);
  const reset = useCounter((s) => s.reset);

  return (
    <div style={{ display: "grid", placeItems: "center", height: "100vh", gap: 12 }}>
      <h1>Count: {count}</h1>
      <div style={{ display: "flex", gap: 8 }}>
        <button onClick={dec}>-1</button>
        <button onClick={reset}>Reset</button>
        <button onClick={inc}>+1</button>
      </div>
    </div>
  );
}
```

Fig 1.3


src > index.css > :root

```css
:root {
  font-family: system-ui, Avenir, Helvetica, Arial, sans-serif;
  line-height: 1.5;
  font-weight: 400;

  color-scheme: light dark;
  color: rgba(255, 255, 255, 0.87);
  background-color: #242424;

  font-synthesis: none;
  text-rendering: optimizeLegibility;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

a {
  font-weight: 500;
  color: #646cff;
  text-decoration: inherit;
}
a:hover {
  color: #535bf2;
}

body {
  margin: 0;
  display: flex;
  place-items: center;
  min-width: 320px;
  min-height: 100vh;
}

h1 {
  font-size: 3.2em;
  line-height: 1.1;
}
```

Fig 1.4

**Output**

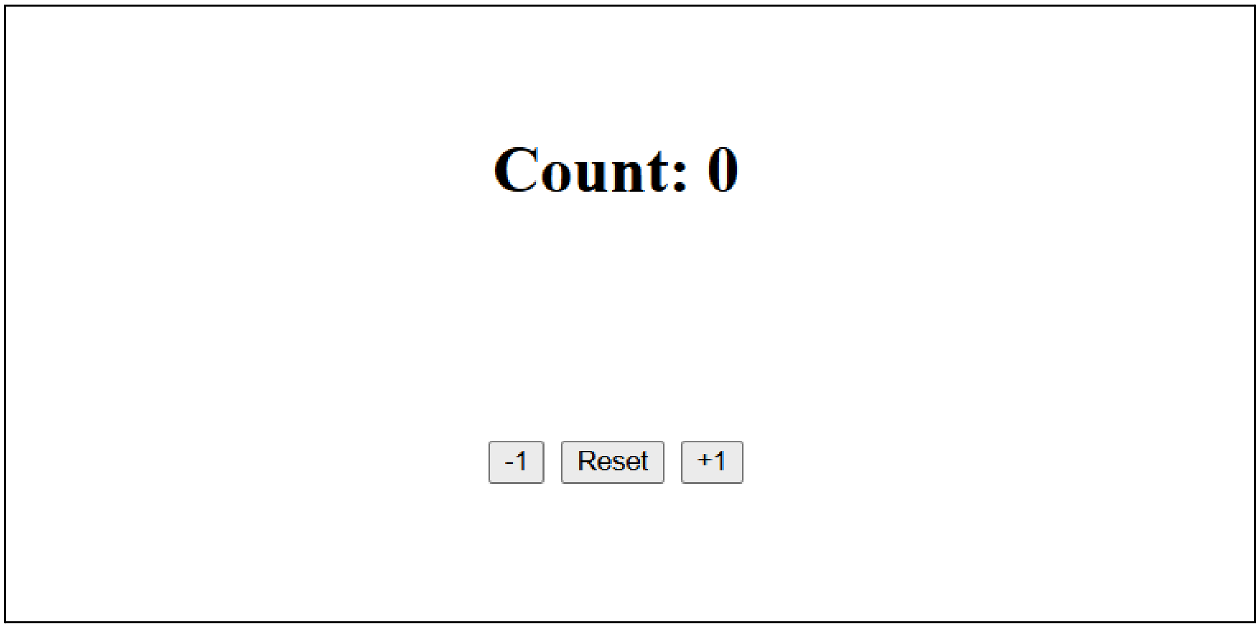

Count: 0

Fig 2.1

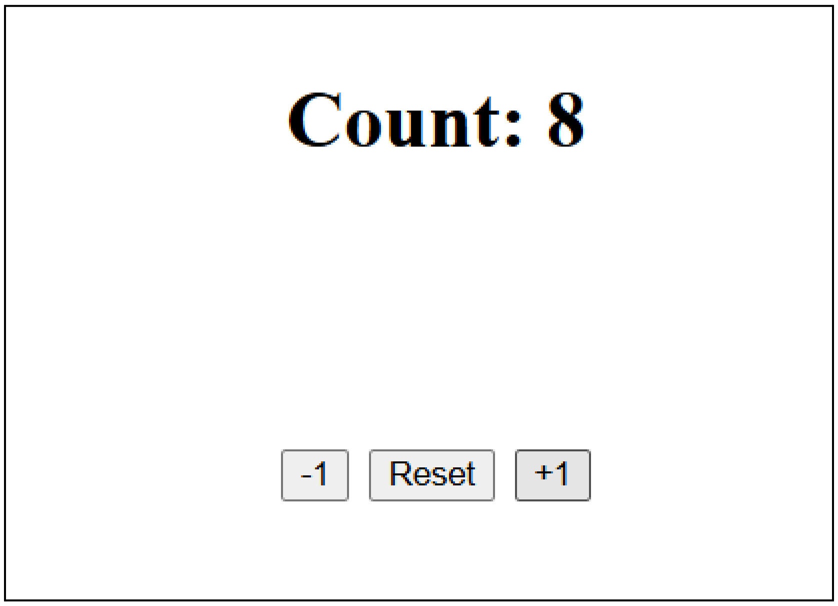

Count: 8

Fig 2.2

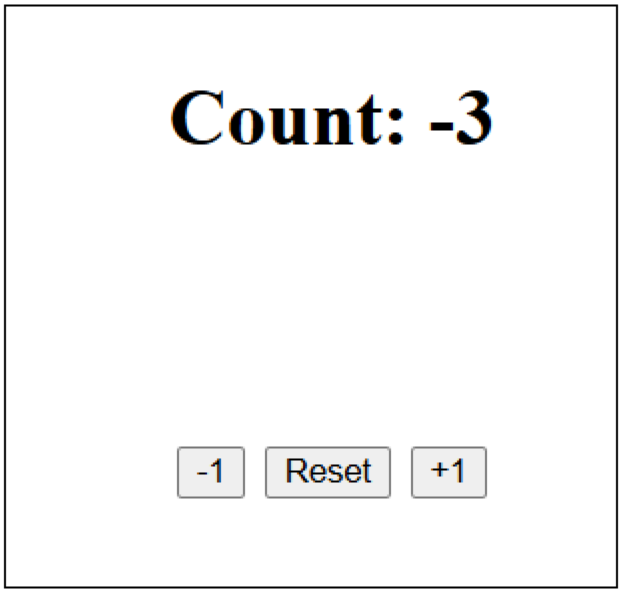


Fig 2.3

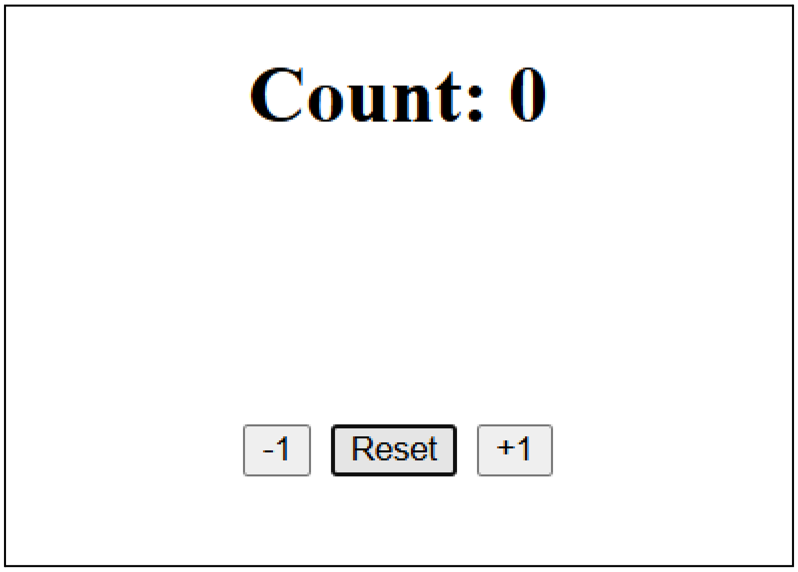


Fig 2.4

**Conclusion**

This experiment shows how React's built-in hooks (`useEffect`, `useContext`, and custom hooks) help manage side effects, share values across components, and encapsulate reusable logic in a clean, declarative way. Zustand complements these hooks by providing a lightweight state manager that reduces boilerplate, avoids unnecessary re-renders with tools like `shallow`/`useShallow`, and simplifies global state compared to Redux. Together, hooks and Zustand enable building scalable React apps with minimal code and maximum clarity.