Assignment-1

Name - Aaryan Gill
UID - 23BAI70473
Section - 23AML-2(B)

1) Benefits of Using Design Patterns in Frontend Development

Design patterns are reusable solutions to commonly occurring problems in software design. In the context of frontend development, they provide a structured approach to managing complexity as applications scale.

- **Code Reusability and DRY (Don't Repeat Yourself):** Patterns like **Factory** or **Singleton** allow developers to reuse logic across different parts of the application, reducing redundancy.
- **Maintainability:** By following established patterns, the codebase becomes more predictable. A new developer can understand the flow of data or component hierarchy more quickly if it follows standard conventions.
- **Decoupling and Modularization:** Patterns such as **Observer** or **Mediator** help in decoupling components. This ensures that changes in one module (e.g., the UI) do not necessitates drastic changes in another (e.g., the data layer).
- **Improved Testing:** When logic is separated through patterns like **Dependency Injection**, units of code become isolated, making it significantly easier to write meaningful unit tests.
- **Standardized Communication:** Design patterns provide a common vocabulary for developers. Saying "we should use a Provider pattern here" immediately conveys a specific architectural intent.

---

2) Classification: Global State vs. Local State in React

Managing state is the core of React development. The distinction between local and global state is defined by the **scope** of data access.

Feature	Local State	Global State
Definition	State managed within a single component.	State accessible by multiple, often unrelated, components.
Tools	useState, useReducer.	Redux, Context API, Zustand, Recoil.
Scope	Restricted to the component and its children (via props).	Entire application or large sub-trees.

Feature	Local State	Global State
Use Case	Form inputs, toggle switches, local loading spinners.	User authentication, theme settings, shopping cart data.
Complexity	Low; easy to implement and reason about.	High; requires boilerplate and architectural planning.

---

3) Comparison of Routing Strategies in SPAs

Routing determines how the application navigates between different views and how URLs are synchronized with the UI.

Client-Side Routing (CSR)

Mechanism: The entire application is loaded once. JavaScript intercepts URL changes and updates the DOM without a page refresh.

Trade-offs: Fast transitions after initial load; however, it has a heavy "Initial Bundle Size" and can face SEO challenges without pre-rendering.

Use Case: Highly interactive dashboards and private web apps.

Server-Side Routing (SSR)

Mechanism: Every route change triggers a request to the server, which returns a fully rendered HTML page.

Trade-offs: Excellent SEO and fast "First Contentful Paint." The downside is a "flicker" during navigation and higher server load.

Use Case: Content-heavy sites like blogs, news portals, and e-commerce.

Hybrid Routing (Isomorphic/Universal)

Mechanism: The first load is handled by the server (SSR), but subsequent navigations are handled by the client (CSR). Frameworks like Next.js utilize this.

Trade-offs: Best of both worlds (SEO + Speed). However, it increases architectural complexity and deployment requirements.

Use Case: Modern enterprise-grade SaaS and high-traffic consumer platforms.

---

4) Common Component Design Patterns

Container–Presentational Pattern

Concept: Separates **logic** (Container) from **UI** (Presentational).

Use Case: When a component needs to fetch data and also display it. Separation allows the UI part to be reused with different data sources.

Higher-Order Components (HOC)

Concept: A function that takes a component and returns a new component with injected functionality.

Use Case: Cross-cutting concerns like withAuthentication or withLogging where you want to wrap multiple components with the same logic.

Render Props

Concept: A component whose prop is a function that returns a React element.

Use Case: Sharing stateful logic (like mouse tracking or window resizing) where the consumer decides how the UI should look.

---

5) Responsive Navigation Bar (Material UI)

Below is a conceptual implementation using Material UI (@mui/material).

JavaScript

```
import React, { useState } from 'react';import { AppBar, Toolbar, Typography, Button, IconButton, Drawer, List, ListItem, ListItemText, Box } from '@mui/material';import MenuIcon from '@mui/icons-material/Menu';
const Navbar = () => {
 const [open, setOpen] = useState(false);
 const navItems = ['Dashboard', 'Projects', 'Teams', 'Settings'];

 return (
  <AppBar position="static" sx={{ backgroundColor: '#2C3E50' }}>
    <Toolbar>
      <Typography variant="h6" sx={{ flexGrow: 1 }}>CollabTool</Typography>

      {/* Desktop View */}
```

```
      <Box sx={{ display: { xs: 'none', md: 'block' } }}>
        {navItems.map((item) => <Button key={item} color="inherit">{item}</Button>)}
      </Box>

      {/* Mobile View */}
      <IconButton color="inherit" sx={{ display: { xs: 'block', md: 'none' } }} onClick={() =>
setOpen(true)}>
        <MenuIcon />
      </IconButton>
    </Toolbar>

    <Drawer anchor="right" open={open} onClose={() => setOpen(false)}>
      <List sx={{ width: 250 }}>
        {navItems.map((item) => (
          <ListItem button key={item} onClick={() => setOpen(false)}>
            <ListItemText primary={item} />
          </ListItem>
        ))}
      </List>
    </Drawer>
  </AppBar>
 );
};
```

---

6) Architecture Design: Collaborative Project Management Tool

a) SPA Structure & Routing

Framework: React with react-router-dom.

Nested Routing: Use <Outlet /> to render sub-views (e.g., /project/:id/tasks or /project/:id/settings).

Protected Routes: A wrapper component that checks the Redux auth state. If not authenticated, it redirects to /login using the Maps component.

b) Global State Management (Redux Toolkit)

Slices: authSlice, projectSlice, and taskSlice.

Middleware: Use **Redux-Saga** or **RTK Query** to handle asynchronous API calls.

Real-time Integration: Use a custom middleware to listen to **WebSockets (Socket.io)**. When a server event arrives, the middleware dispatches a Redux action to update the UI instantly for all users.

c) Responsive UI & Custom Theming

Material UI Theme: Define a custom primary/secondary palette in createTheme.

Responsive Grids: Use <Grid container spacing={2}> to ensure the project board (Kanban) shifts from 3 columns on desktop to 1 column on mobile.

d) Performance Optimization

Virtualization: Use react-window or react-virtuoso for rendering large task lists to ensure only visible items are in the DOM.

Memoization: Implement React.memo and useMemo for heavy calculations to prevent unnecessary re-renders during real-time updates.

Code Splitting: Use React.lazy() for route-based splitting to reduce the initial bundle size.

e) Scalability & Multi-user Concurrency

Optimistic Updates: Update the UI immediately when a user moves a task, then sync with the server. If the server fails, roll back the state.

Conflict Resolution: Implement **Operational Transformation (OT)** or **CRDTs** (Conflict-free Replicated Data Types) for simultaneous text editing.

Throttling: Throttle cursor movements or "user typing" events sent via WebSockets to prevent overwhelming the server.