Name: Rishit Sagar

UID: 23BAI70462

Sec: 23aml-2 A

Subject: Full Stack Assignment 1

Q1) Summarize the benefits of using design patterns in frontend development.

Answer:

Design patterns are proven solutions to recurring problems in software design. In frontend development, they provide a standard terminology and are essential for writing code that is easy to understand and maintain. The key benefits include:

1. **Reusability:** Patterns encourage writing modular code. Instead of reinventing the wheel for every feature (like a modal or a data fetcher), we can reuse established structures, reducing redundancy.
2. **Maintainability:** By separating concerns (e.g., separating business logic from UI), code becomes easier to debug and update. If a bug arises in the logic, we know exactly where to look without breaking the UI.
3. **Scalability:** As applications grow, "spaghetti code" becomes a major issue. Patterns like the Observer or Singleton pattern help manage complex state and component interactions, allowing the app to scale without becoming unmanageable.
4. **Communication:** Design patterns create a shared vocabulary. When I tell a teammate I used a "Singleton" for the WebSocket connection or a "Provider" for the theme, they immediately understand the architectural decision.

Q2) Classify the difference between global state and local state in React.

Answer:

Feature	Local State	Global State
Definition	State that is managed within a single component and only affects that specific component (or its immediate children via props).	State that is shared across the entire application or multiple distinct component trees.
Implementation	useState, useReducer.	Redux, Context API, Zustand, Recoil.
Data Persistence	Lost when the component unmounts.	Persists as long as the app is running (or until page refresh); easier to persist in local storage.
Use Cases	Form input values, toggling a modal open/close, active tab selection.	User authentication status, theme settings (Dark/Light mode), shopping cart data, cached API responses.
Prop Drilling	Requires passing props down manually if children need it (which can get messy).	Avoids prop drilling; data is accessible by any component subscribed to the store.

Q3) Compare different routing strategies in Single Page Applications (client-side, server-side, and hybrid) and analyze the trade-offs and suitable use cases for each.

Answer:

1. Client-Side Routing (CSR):

- **Mechanism:** The browser loads a single HTML file. JavaScript handles the URL changes and DOM updates without refreshing the page.
- **Trade-offs:** Fast transitions after initial load, but slower initial load (large JS bundle) and historically poorer SEO.
- **Use Case:** Dashboards, SaaS platforms, or apps behind a login where SEO is not a priority (e.g., Trello, Jira).

2. Server-Side Routing (SSR):

- **Mechanism:** The server generates the full HTML for every request and sends it to the browser.
- **Trade-offs:** Excellent SEO and faster "First Contentful Paint," but higher server load and slower page transitions compared to CSR.
- **Use Case:** Content-heavy sites like news portals or e-commerce product pages (e.g., Amazon listings).

3. Hybrid (SSG/ISR):

- **Mechanism:** Combines both. Static generation (SSG) builds pages at compile time, while Incremental Static Regeneration (ISR) updates them in the background.
- **Trade-offs:** Best performance and SEO, but complex architecture.
- **Use Case:** Blogs, marketing websites, and documentation sites (e.g., Next.js documentation).

Q4) Examine common component design patterns such as Container–Presentational, Higher-Order Components, and Render Props, and identify appropriate use cases for each pattern.

Answer:

1. Container–Presentational Pattern:

- **Concept:** Separates logic (Container) from the view (Presentational). The Container fetches data and handles state; the Presentational component just renders props.
- **Use Case:** A UserListContainer fetches user data from an API and passes it to a UserList component which simply maps over the array to display UI cards.

2. Higher-Order Components (HOC):

- **Concept:** A function that takes a component and returns a new component with enhanced props or logic.
- **Use Case:** withAuth(ProfilePage) – wrapping a component to check if a user is logged in before rendering it. Also common for logging or styling (e.g., withStyles).

3. Render Props:

- **Concept:** A component with a prop (usually called render or children) that takes a function returning a React element. It allows sharing code between components using a prop whose value is a function.
- **Use Case:** A MouseTracker component that tracks the cursor position and exposes the (x, y) coordinates to any component placed inside it via a render function.

Q5) Demonstrate and develop a responsive navigation bar using Material UI components while applying appropriate styling and breakpoint configurations.

Answer:

Below is the implementation of a responsive Navbar. It shows a full menu on desktop and switches to a hamburger drawer on mobile devices.

JavaScript

```
import React, { useState } from 'react';
import {
 AppBar, Toolbar, Typography, Button, IconButton,
 Drawer, List, ListItem, ListItemText, useMediaQuery, useTheme
} from '@mui/material';
import MenuIcon from '@mui/icons-material/Menu';

const Navbar = () => {
 const [openDrawer, setOpenDrawer] = useState(false);
 const theme = useTheme();
 // Returns true if screen width is below 'md' (900px)
 const isMobile = useMediaQuery(theme.breakpoints.down('md'));

 const menuItems = ['Dashboard', 'Projects', 'Team', 'Reports'];

 return (
  <>
   <AppBar position="static" sx={{ backgroundColor: '#2E3B55' }}>
    <Toolbar>
     <Typography variant="h6" sx={{ flexGrow: 1, fontWeight: 'bold' }}>
      ProjectFlow
     </Typography>

     {isMobile ? (
      <IconButton edge="start" color="inherit" onClick={() => setOpenDrawer(true)}>
       <MenuIcon />
      </IconButton>
     ) : (
      <div>
       {menuItems.map((item) => (
```

```
            <Button key={item} color="inherit" sx={{ marginLeft: 2 }}>
              {item}
            </Button>
          ))}
        </div>
      )}
    </Toolbar>
  </AppBar>

  {/* Mobile Drawer */}
  <Drawer anchor="right" open={openDrawer} onClose={() => setOpenDrawer(false)}>
    <List sx={{ width: 250 }}>
      {menuItems.map((item) => (
        <ListItem button key={item} onClick={() => setOpenDrawer(false)}>
          <ListItemText primary={item} />
        </ListItem>
      ))}
    </List>
  </Drawer>
 </>
 );
};

export default Navbar;
```

Q6) Evaluate and design a complete frontend architecture for a collaborative project management tool with real-time updates.

Answer:

A) SPA Structure with Nested Routing:

The app follows a feature-based directory structure to ensure modularity.

- **Router:** Use react-router-dom.
- **Structure:**
  - /login: Public route.
  - /app: Protected layout (requires auth token).
    - /app/dashboard: Overview.
    - /app/project/:id: Project details (nested routes for Kanban/List views).
- **Protection:** Wrap protected routes in a <RequireAuth> component that checks Redux state for a valid JWT.

B) Global State Management (Redux Toolkit):

We use Redux Toolkit (RTK) to minimize boilerplate.

- **Slices:**
  - authSlice: Stores user token and profile.
  - projectSlice: Stores list of projects (normalized data structure).
  - uiSlice: Manages modals, sidebars, and toast notifications.
- **Middleware:** Use **RTK Query** for data fetching. It handles caching, polling, and invalidation automatically, which is superior to manual useEffect fetching.

C) Responsive UI (Material UI):

- **Theming:** Create a theme.js using createTheme to define brand colors (primary: #1976d2, secondary: #dc004e) and typography.
- **Layout:** Use MUI Grid (v2) or Stack for consistent spacing.
- **Custom Hooks:** Create hooks like useResponsiveLayout to programmatically adjust Kanban column widths based on the device.

D) Performance Optimization:

- **Virtualization:** Since project boards can have thousands of tasks, use react-window to only render the tasks currently visible in the viewport.
- **Lazy Loading:** Use React.lazy() and Suspense for heavy routes (e.g., the Analytics dashboard) to reduce the initial bundle size.
- **Memoization:** Use React.memo for individual Task Cards to prevent re-rendering the entire list when only one task changes.

E) Scalability & Real-Time Access:

- **WebSockets:** Integrate **Socket.io-client**.
  - When User A moves a task, the server emits a task_moved event.
  - Client listens for this event and dispatches a Redux action to update the store immediately.
- **Optimistic UI:** When a user performs an action (like deleting a task), update the UI immediately *before* the server responds. If the server fails, roll back the change and show an error. This makes the app feel instant even on slow networks.
- **Concurrency:** Implement "locking" logic. If User A is editing a task description, show a "Being edited by User A" badge to User B to prevent overwrite conflicts.