Submitted By: Shreyas Gautam
UID: 23BAI70374
Section: 23AML-2(A)

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

Ans. Design Patterns are reusable templates for solving architectural and logical problems. They help in making the design pattern more structured. Main benefits include:

- Maintainability and Scalability: Patterns allow us to create a decoupled codebase which means adding new features and making changes to the codebase does not lead to unintended side effects in unrelated modules.
- Standardization and Communication: Patterns are a universal standard that streamline communication between teams and ensure that anyone developer looking at the codebase can understand its data flow instantly improving accessibility and reducing the need for extensive documentation.
- Code Reusability: Patterns allow creation of pluggable components and logic that allow us to reuse the same logic across different parts of the application with minimal configuration changes.
- Predictability and Debugging: Patterns ensure that state transitions and actions are controlled and predictable. This makes it easier to trace bugs and resolve bugs.
- Optimization: Patterns also allow us to minimize redundant computations and improve responsiveness in browser.

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

Ans. State is an internal and mutable object that holds information about a component. When state changes, React automatically re-renders the UI to reflect the new data.

| Feature | Local State | Global State |
|---|---|---|
| Definition | It is the data encapsulated within a single component. | It represents shared data between multiple related or unrelated components. |
| Mechanics | It lives and dies within a components Mount/Unmount lifecycle and is managed using useState or useReducer hooks. | It is stored outside of the specific component tree and is managed by the Context API or libraries like Redux Toolkit. |
| Data Flow | It follows top-down and unidirectional flow so to share it you must pass it as props to children. | It can broadcast changes and any component can subscribe to listen to changes in the global 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.

Ans.

| Feature | Client-Side Routing | Server-Side Routing | Hybrid Routing |
|---|---|---|---|
| Definition | Navigation is handled by the browser without | A full-page reload is triggered with | The server handles the initial load while |

| | | | |
|---|---|---|---|
| | asking the server for a new HTML page. | every URL change where the server sends back a completely new HTML file thus following a multi-page model. | JavaScript handles the client-side navigation afterwards. |
| Mechanism | History API is used to handle the URL then JavaScript intercepts the click and swaps out the components on the screen. | Uses a request-response cycle where navigation triggers a GET request that the server processes to generate and return a complete HTML document. | Server generates html and critical CSS to deliver a complete page to the browser. The browser then attaches javascript to the page making it interactive. Now, all subsequent actions are handled locally by client-side routing. |
| Use Case | Used in standard SPA's where speed after the initial load is a priority. | Used in standard MPA's that are content-heavy with a requirement for SEO. | Used in websites where you require both SEO and smooth user experience like in E-commerce or social media. |
| Trade-Offs | Fast transitions but slow initial load and poor SEO. | Better SEO and Time to Interactive(initial load speed) but navigation faces increased latency due to full-page refresh on every request. | Best performance but adds architectural complexity. |

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

Ans.

1. Container-Presentational Pattern: It helps in achieving separation of concerns. It divides the components into two distinct roles: one handles the logic(how things work) while the other handles the UI(how things look).
   Container components fetch data, handle state transitions, and communicate with external services. They do not contain any CSS or HTML except the presentational component they wrap. They are referred to as "Smart" components.
   Presentational component: They receive data strictly through props. They are stateless functional components that provide layout and styling. They are referred to as "Dumb" components.
   Use Case: They are used for reusability and testability. By isolating the UI, we can test the look of a component without needing a mock server or state management.

2. Higher-Order Components(HOC): Specialized functions that take a component as an argument and return an improved version of it.
   It follows the functional programming principle of pure functions.
   Example: *const ProtectedRoute=withAuth(Dashboard);*
   The HOC wraps the logic and passes on the necessary data.
   Use Case: Used to address cross-cutting concerns. These are features that multiple unrelated components need such as Authentication checks etc. It prevents code duplication and centralizes common logic.

3. Render Props: It is used for dynamic content injection. Instead of hardcoding child components, a component is given a prop(called render) that is a function. This function then tells the component what to display.
   It uses a function a function as a prop to share state between the component and what it renders.
   Example: You write logic for a component <Toggle/> that true/false state. It doesn't have html/css of its own. It can work for Menu, Bio, and Theme toggle without having to hardcode the logic for each.
   Use Case: It is used for flexible state sharing. Used when a single source needs to support a variety of UI layouts.

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

Ans.

```
import React, { useState } from 'react';

import {
  AppBar, Toolbar, Typography, Button, IconButton,
  Drawer, List, ListItem, ListItemText, Box, useMediaQuery, useTheme
} from '@mui/material';
import MenuIcon from '@mui/icons-material/Menu';
const Navbar = () => {
  const [mobileOpen, setMobileOpen] = useState(false);
  const theme = useTheme();
  const isMobile = useMediaQuery(theme.breakpoints.down('md'));
  const navItems = ['Dashboard', 'Projects', 'Tasks', 'Settings'];
```

```
  const handleDrawerToggle = () => {
    setMobileOpen(!mobileOpen);
  };
  const drawer = (
    <Box onClick={handleDrawerToggle} sx={{ textAlign: 'center', width: 250
}}>
      <Typography variant="h6" sx={{ my: 2 }}>PRO-MANAGE</Typography>
      <List>
        {navItems.map((item) => (
          <ListItem key={item} disablePadding>
            <ListItemText primary={item} sx={{ textAlign: 'center', py: 1 }}
/>
          </ListItem>
        ))}
      </List>
    </Box>
  );
  return (
    <>
      <AppBar position="static" sx={{ bgcolor: '#1a237e' }}>
        <Toolbar>
          <IconButton
            color="inherit"
            edge="start"
            onClick={handleDrawerToggle}
            sx={{ mr: 2, display: { md: 'none' } }}
          >
            <MenuIcon />
          </IconButton>
          <Typography variant="h6" sx={{ flexGrow: 1 }}>
            Collaborative Hub
          </Typography>
          <Box sx={{ display: { xs: 'none', md: 'block' } }}>
            {navItems.map((item) => (
              <Button key={item} color="inherit">{item}</Button>
            ))}
          </Box>
        </Toolbar>
      </AppBar>
      <Drawer
        variant="temporary"
        open={mobileOpen}
        onClose={handleDrawerToggle}
        ModalProps={{ keepMounted: true }}
      >
        {drawer}
      </Drawer>
    </>
```

```
  );
};
export default Navbar;
```

Q6. Evaluate and design a complete frontend architecture for a collaborative project management tool with real-time updates. Include: a) SPA structure with nested routing and protected routes b) Global state management using Redux Toolkit with middleware c) Responsive UI design using Material UI with custom theming d) Performance optimization techniques for large datasets e) Analyze scalability and recommend improvements for multi-user concurrent access.

Ans.

This architecture prioritizes modularity, state consistency, and high-performance rendering for real-time collaboration.

a) SPA Structure: Nested & Protected Routes

The system utilizes React Router v6 to implement a declarative hierarchy. It uses a higher-order <ProtectedRoute> component intercepting navigation by validating authentication tokens. This layout uses an /app wrapper for persistent navigation along with nested outlets for managing dynamic views like /app/projects/:id.

b) Global State: Redux Toolkit (RTK) with Middleware

A state is partitioned into RTK Slices (Auth, Projects, Tasks), the RTK Query manages server-state caching, while custom Socket.io middleware syncs real-time events. This ensures that teammate actions trigger immediate state updates without manual polling.

c) Responsive UI Design: Material UI (MUI) Custom Theming

The visual layer uses MUI's ThemeProvider which allows provides a centralized createTheme configuration to override default palettes and typography resulting in a professional SaaS aesthetic. Meanwhile, responsive layouts use MUI's breakpoint system for cross-device compatibility.

d) Performance Optimization for Large Datasets

The architecture makes use of DOM Virtualization via react-window to maintain a 60 fps framerate. By rendering only those components that are within the active viewport, the memory overhead and DOM reconciliation time of the application is reduced.

e) Scalability and Multi-User Concurrency

For high-concurrency environments Optimistic Updates provide immediate feedback. To resolve synchronization conflicts in such environments we should implement CRDTs (Conflict-free Replicated Data Types) to provide consistent state convergence during simultaneous edits.

Example:

```
// src/routes/ProtectedRoute.jsx
```

```jsx
import { Navigate, Outlet } from 'react-router-dom';
import { useSelector } from 'react-redux';
const ProtectedRoute = () => {
  const { isAuthenticated } = useSelector((state) => state.auth);
  return isAuthenticated ? <Outlet /> : <Navigate to="/login" />;
};
// src/App.jsx
const AppRoutes = () => (
  <Routes>
    <Route path="/login" element={<Login />} />
    <Route element={<ProtectedRoute />}>
      <Route path="/dashboard" element={<DashboardLayout />}>
        <Route index element={<ProjectList />} />
        <Route path="project/:id" element={<ProjectBoard />} />
      </Route>
    </Route>
  </Routes>
);
```

```jsx
// src/store/middleware/socketMiddleware.js
export const socketMiddleware = (socket) => (store) => {
  socket.on('task_updated', (updatedTask) => {
    store.dispatch(updateTaskLocal(updatedTask));
  });
  return (next) => (action) => {
    if (action.type === 'tasks/moveTask') {
      socket.emit('move_task', action.payload);
    }
    return next(action);
  };
};
```

```jsx
import { createTheme } from '@mui/material/styles';
export const theme = createTheme({
  palette: {
    primary: { main: '#2c3e50' },
    secondary: { main: '#e74c3c' },
  },
  components: {
    MuiCard: {
      styleOverrides: {
        root: { borderRadius: 12, boxShadow: '0 4px 20px rgba(0,0,0,0.08)' }
      }
    }
  }
});
```