DEPARTMENT OF APEX INSTITUTE
OF TECHNOLOGY AI & ML

Name: Mokshda Sharma
UID: 23BAI70178

Assignment -1

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

Ans: Design patterns in frontend development promote code reusability by enabling modular components that can be used across projects, reducing development time. They enhance maintainability through clear structures that make debugging and updates easier for teams. Scalability improves as patterns manage growing complexity without introducing bugs, and readability follows established conventions for better collaboration. Performance benefits arise from optimizations like reduced memory usage in patterns such as Flyweight.

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

Ans: **Global state** is shared across multiple components via tools like Redux, suitable for app-wide data such as user authentication or themes, but it adds complexity and can trigger widespread re-renders. **Local state**, managed with useState in a single component, handles UI-specific data like form inputs with simpler encapsulation and fewer performance impacts.

Aspect	Global State	Local State
Scope	App-wide, accessible anywhere cursa	Component-only chucksacademy
Complexity	Higher setup (e.g., Redux) cursa	Simpler, less boilerplate chucksacademy
Performance	May re-render many components cursa	Limited to owning component chucksacademy
Use Cases	Auth, themes, shared data chucksacademy	Forms, toggles cursa

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: **Client-side routing** (e.g., React Router) handles navigation in the browser without full page reloads, offering fast transitions and app-like feel but poor initial SEO and relies on JavaScript. **Server-side routing** pre-renders pages on the server for better SEO and fast first loads, though it suffers slower interactions due to reloads. **Hybrid routing** (e.g., Next.js)

combines both via SSR/SSG for initial loads and CSR for interactions, balancing SEO, speed, and dynamism.

Strategy	Trade-offs	Use Cases
Client-side	Fast UX, poor SEO, JS-dependent telerik	Dashboards, internal apps telerik
Server-side	Good SEO, slow navigation telerik	Content sites, public pages
Hybrid	Best of both, higher complexity telerik	E-commerce, blogs with interactivity

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: **Container-Presentational** separates logic-fetching "container" components from dumb "presentational" ones focused on UI, ideal for data-heavy views to improve testability.

Higher-Order Components (HOC) wrap components to reuse logic like auth or data fetching, great for cross-cutting concerns but can cause wrapper hell.

Render Props passes a render function as a prop for flexible logic sharing (e.g., mouse tracking), suiting dynamic UIs without deep nesting.

Pattern	Use Cases
Container-Presentational	Data fetching + display reddit
HOC	Reusable enhancers (auth) refine
Render Props	Flexible behavior sharing

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, useTheme, useMediaQuery } from '@mui/material';
import MenuIcon from '@mui/icons-material/Menu';

const ResponsiveNavBar = () => {
  const [drawerOpen, setDrawerOpen] = useState(false);
  const theme = useTheme();
  const isMobile = useMediaQuery(theme.breakpoints.down('md'));

  const navItems = ['Home', 'About', 'Services', 'Contact'];

  return (
    <AppBar position="static">
      <Toolbar>
        <Typography variant="h6" sx={{ flexGrow: 1 }}>Logo</Typography>
        {isMobile ? (
          <>
            <IconButton color="inherit" onClick={() => setDrawerOpen(true)}>
              <MenuIcon />
            </IconButton>
            <Drawer anchor="right" open={drawerOpen} onClose={() => setDrawerOpen(false)}>
              <List>
                {navItems.map((item) => (
                  <ListItem button key={item}>
                    <ListItemText primary={item} />
                  </ListItem>
                ))}
              </List>
            </Drawer>
```

```
      </>
    ) : (
      navItems.map((item) => (
        <Button color="inherit" key={item}>{item}</Button>
      ))
    )}
    <Button color="inherit">Login</Button>
  </Toolbar>
 </AppBar>
 );
};
export default ResponsiveNavBar;
```

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

Ans: Use React Router v6 for nested routes: /dashboard/projects/:id/tasks with loaders. Protected routes check auth context, redirecting unauth users.

```
<Routes>
  <Route path="/login" element={<Login />} />
  <Route element={<ProtectedRoute />}>
    <Route path="/" element={<Dashboard />}>
      <Route path="projects/:id" element={<Project />} />
    </Route>
  </Route>
</Routes>
```

b) Global state management using Redux Toolkit with middleware

Ans: Redux Toolkit slices for projects/tasks/users. Add Socket.io middleware for real-time: dispatch actions on WebSocket events (e.g., task updates).

```
// socketMiddleware.js
const socketMiddleware = (socket) => (store) => (next) => (action) => {
```

```
  // Listen for events, dispatch updates
  socket.on('taskUpdate', (data) => store.dispatch(updateTask(data)));
  return next(action);
};
```

c) Responsive UI design using Material UI with custom theming

Ans: Material UI's createTheme customizes colors, typography, and breakpoints for responsive design. Define a palette (primary/secondary), extend spacing, and set responsive breakpoints like xs: 0, sm: 600, md: 960, lg: 1280, xl: 1920 to adapt layouts.

Example theme setup:

```
import { createTheme } from '@mui/material/styles';
const theme = createTheme({
  palette: {
    primary: { main: '#1976d2' },
    secondary: { main: '#dc004e' },
    background: { default: '#f5f5f5' },
  },
  typography: {
    fontFamily: 'Roboto, sans-serif',
    h1: { responsive: true },
  },
  breakpoints: {
    values: { xs: 0, sm: 600, md: 900, lg: 1200, xl: 1536 },
  },
  components: {
    MuiGrid: { styleOverrides: { container: { spacing: 2 } } },
  },
});
<ThemeProvider theme={theme}>
  {/* App components */}
</ThemeProvider>
```

Use useMediaQuery for conditional rendering, Grid with xs={12} sm={6} for responsive grids, and Box sx={{ display: { xs: 'block', md: 'flex' } }} for adaptive layouts. This creates fluid UIs for project management dashboards, cards, and lists that stack on mobile.

d) Performance optimization techniques for large datasets

Ans: For project tasks (e.g., 10k+ items), avoid full re-renders with virtualization: MUI DataGridPro or react-window renders only visible rows, slashing memory by 90%.

Key techniques:

- **Virtualization**: FixedSizeList from react-window; limits DOM nodes.
- **Memoization**: React.memo, useMemo for expensive computations like filtered lists.
- **Lazy Loading**: React.lazy + Suspense for routes; infinite scroll with react-query.
- **Pagination**: Server-side via RTK Query, client-side with useTable from TanStack.
- **Code Splitting**: Dynamic imports; tree-shaking with Vite/Webpack.

```
import { FixedSizeList as List } from 'react-window';

<List
  height={500}
  itemCount={tasks.length}
  itemSize={50}
>
  {({ index, style }) => <div style={style}>{tasks[index].name}</div>}
</List>
```

e) Analyze scalability and recommend improvements for multi-user concurrent access.

Ans: Multi-user concurrency in project management risks conflicts (e.g., simultaneous task edits); optimistic UI updates (dispatch immediately, rollback on error) with Socket.io for real-time sync handle 100+ users per project.

Analysis: Redux Toolkit + Socket middleware broadcasts changes; WebSockets scale better than polling (low latency <100ms). Bottlenecks: state bloat (use normalized schemas via normalizr), re-renders (selectors with reselect).

Recommendations:

- **Real-time Layer**: Socket.io with rooms (join(projectId)); fallback to Server-Sent Events.
- **Conflict Resolution**: Operational Transforms (Yjs lib) or CRDTs for merges; presence indicators.

- **State Normalization**: RTK entities for O(1) lookups; pagination for lists.
- **Backend Sync**: Webhooks + RTK Query invalidation; optimistic mutations with rollback.
- **Horizontal Scale**: Deploy to Vercel/Render; use Redis pub/sub for Socket scaling; monitor with Sentry.
- **Edge Cases**: Offline support via IndexedDB + service workers; reconnection logic.