APEX INSTITUTE OF TECHNOLOGY

ASSIGNMENT - 1

Student Name: Aditya **UID:** 23BAI70127

Branch: CSE AIML **Section/Group:** 23AML – 3A

Semester: 6th **Subject Code:** 23CSH-382

Subject Name: Full stack

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

Design patterns exist because frontend complexity grows non-linearly. Without patterns, UI code degenerates into tightly coupled, unreadable logic.

Key Benefits

a) Separation of Concerns

Patterns enforce boundaries:

- UI rendering
- Business logic
- State handling
- Side effects

b) Scalability

Patterns let you:

- Add features without rewriting existing code
- Split work across teams
- Maintain consistency

c) Maintainability

Well-applied patterns:

- Reduce regression bugs

- Make refactoring predictable
- Localize changes

d) Reusability

Reusable logic reduces:

- Code duplication
- Inconsistent behavior
- Testing effort

e) Testability

Patterns enable:

- Unit testing logic without DOM
- Mocking data sources
- Predictable outputs

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

Local State

- Scoped to a component
- Managed via `useState`, `useReducer`
- Short-lived, UI-specific

Use for:

- Form inputs
- Toggle states
- Modal visibility
- Component-only UI behaviour

Global State

- Shared across components
- Managed via tools like **Redux Toolkit**, Context, Zustand
- Long-lived, application-wide

Use for:

- Auth state
- User profile
- Theme
- Notifications
- Cached API data

3) 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.

Client-Side Routing

Handled entirely in the browser (React Router).

Pros

- Fast navigation
- No full reloads
- Better UX

Cons

- SEO requires extra work
- Initial bundle size matters

Use when

- Dashboards
- Authenticated apps
- Internal tools

Server-Side Routing

Server decides routes and returns full pages.

Pros

- Excellent SEO
- Faster first content paint

Cons

- Slower navigation
- More server load

Use when

- Content-heavy sites
- Marketing pages

Hybrid Routing

Combination (SSR + client hydration).

Pros

- Best SEO + UX
- Scales well

Cons

- Complex architecture
- Higher dev cost

Use when

- SaaS products
- Public-facing apps with dashboards

Hard truth:

If SEO matters and you still use pure CSR → you made a business mistake.

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

Container–Presentational

Logic separated from UI.

Use when

- Same UI with different data sources
- Testability matters

Avoid when

- Small components (overkill)

Higher-Order Components (HOC)

Component wrapped with extra behaviour.

Use when

- Cross-cutting concerns (auth, logging)

Problems

- Wrapper hell
- Debugging pain

Modern verdict:
Hooks replace 90% of HOC use cases.

Render Props

Behavior passed as a function.

Use when

- Highly reusable logic with flexible UI

Cons

- JSX nesting
- Readability drops fast

Blunt reality:

Hooks > Render Props > HOCs

That's the hierarchy in modern React.

5) Implementation: Responsive Material UI Navigation.

Using Material UI

```
import React from "react";
import {
  AppBar,
  Toolbar,
  Typography,
  IconButton,
  Button,
  Box,
  Drawer,
  List,
  ListItem,
  ListItemText,
  useTheme,
  useMediaQuery
} from "@mui/material";
import MenuIcon from "@mui/icons-material/Menu";
const Navbar = () => {
  const theme = useTheme();
```

```
const isMobile = useMediaQuery(theme.breakpoints.down("md"));
const [open, setOpen] = React.useState(false);
const navItems = ["Dashboard", "Projects", "Teams", "Settings"];
return (
  <>
    <AppBar position="static">
      <Toolbar>
        {isMobile && (
          <IconButton color="inherit" onClick={() => setOpen(true)}>
            <MenuIcon />
          </IconButton>
        )
        <Typography variant="h6" sx={{ flexGrow: 1 }}>
          ProjectManager
        </Typography>
        {!isMobile &&
          navItems.map(item => (
            <Button key={item} color="inherit">
              {item}
            </Button>
          ))}
      </Toolbar>
    </AppBar>
```

```
      <Drawer anchor="left" open={open} onClose={() => setOpen(false)}>
        <Box sx={{ width: 250 }}>
          <List>
            {navItems.map(item => (
              <ListItem button key={item}>
                <ListItemText primary={item} />
              </ListItem>
            ))}
          </List>
        </Box>
      </Drawer>
    </>
  );
};
export default Navbar;
```

6) Frontend Architecture: Collaborative Project Management Tool

a) SPA Structure

- React Router with nested routes
- Protected routes using auth guards

/login

/app

├── dashboard

├── projects/:id

```
│   ├── tasks
│   └── activity
└── settings
```

b) Global State (Redux Toolkit)

- Auth slice
- Projects slice
- Tasks slice
- WebSocket middleware

Middleware responsibilities

- Handle real-time updates
- Sync optimistic UI
- Rollback on failure

c) Responsive UI + Theming

- Central MUI theme
- Dark/light mode
- Design tokens (spacing, colors)
- Grid + Flex layouts only (no absolute chaos)

d) Performance Optimization

- Virtualized lists (for tasks)
- Memoization (useMemo, useCallback)
- Normalized Redux state
- WebSocket diff updates (not full payloads)
- Lazy-loaded routes

If you skip this:
Large datasets will **destroy your FPS**.

e) Scalability & Multi-User Concurrency

Problems to expect

- Race conditions
- Conflicting updates
- UI desync

Solutions

- Server authoritative state
- WebSockets with versioning
- Optimistic UI + reconciliation
- Fine-grained subscriptions
- Role-based access control

Blunt truth:
Real-time apps fail not because of UI, but because **state sync logic is naïve**.