Full Stack-II

Assignment-1

Name- Gurleen Kaur

Section- 23AML-2(B)

UID- 23BAI70141

1. Summarize the benefits of using design patterns in front-end development.

Ans. The core benefits of utilizing design patterns in frontend development:

1. Code Reusability

Patterns like the **Component Pattern** allow developers to reuse UI elements (buttons, forms, cards) instead of rewriting code.
→ Reduces duplication and speeds up development.

2. Better Maintainability

Architectures such as **MVC/MVVM** separate data, UI, and logic.
→ Bugs can be identified and fixed easily.

3. Scalability

Patterns like **Flux/Redux** manage application state in a structured way.
→ Projects can grow without becoming messy.

4. Team Collaboration

Standard structures are familiar to developers.
→ Easier teamwork and onboarding.

5. Readability

Well-known patterns make code easier to understand.
→ Improves clarity for large projects.

6. Faster Development

Developers use proven solutions instead of designing from scratch.
→ Saves time and reduces errors.

7. Easier Testing

Modular code allows independent testing of components and logic.
→ More reliable applications.

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

Ans. In React, *state* stores data that controls how components behave and render. It is mainly classified into **Local State** and **Global State** based on scope and usage.

Basis	Local State	Global State
Definition	State managed inside a single component	State shared across multiple components
Scope	Limited to that component only	Accessible throughout the application
Creation	Created using useState()	Managed using Context API, Redux, Zustand, etc.
Data Sharing	Passed through props (prop drilling)	Directly accessed by any component
Complexity	Simple and easy to manage	More complex but structured
Performance	Faster for small UI updates	Useful for large apps but may cause re-renders if poorly managed
Example Usage	Form inputs, toggle button, modal open/close	User authentication, theme, cart items, language settings

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.

Ans. Routing decides how a web application loads different pages (views) when the URL changes. In modern frontend frameworks (React, Angular, Vue), three main routing strategies are used.

1. Client-Side Routing (CSR)

Navigation is handled completely inside the browser using JavaScript without reloading the page.

Examples: React Router, Angular Router

Advantages:

- Very fast navigation (no full page reload)
- Smooth user experience
- Reduced server load
- Works well for interactive apps

Disadvantages:

- Poor SEO (content not initially visible to search engines)
- Slower first load (large JS bundle)
- Depends heavily on JavaScript

Suitable Use Cases:

- Dashboards (admin panels)
- Social media apps
- Email applications

2. Server-Side Routing (SSR)

Each URL request goes to the server, which returns to a fully rendered HTML page.

Examples: Traditional websites, PHP, JSP, Django templates

Advantages:

- Excellent SEO
- Fast initial page load
- Works even if JavaScript disabled

Disadvantages:

- Full pages reload on navigation
- Slower user experience
- Higher server load

Suitable Use Cases:

- Blogs and news websites
- Marketing websites
- Content-heavy platforms

3. Hybrid Routing (CSR + SSR)

Server renders the first page, then client-side routing handles further navigation.

Examples: Next.js, Nuxt.js

Advantages:

- Good SEO + fast navigation
- Faster initial load than CSR
- Better performance balance

Disadvantages:

- More complex architecture
- Requires server + frontend configuration
- Higher development effort

Suitable Use Cases:

- E-commerce websites
- Large production apps
- Platforms needing SEO and interactivity

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

Ans. **React Component Design Patterns and Use Cases:**

1. Container–Presentational Pattern

Separates **logic** and **UI** into two components.

- Container → handles state, API, business logic
- Presentational → displays data using props

Use Cases:

- Data fetching pages (lists, dashboards)
- Forms with validation logic
- Large apps needing clean structure

2. Higher-Order Component (HOC)

A function that takes a component and returns an enhanced component to reuse behavior.

Use Cases:

- Authentication/authorization
- Loading spinners
- Logging or analytics
- Permission-based UI

3. Render Props

A component shares logic using a **function prop** to decide what UI to render.

Use Cases:

- Reusable form handling
- Mouse/scroll tracking
- Data fetching with different layouts
- Animations

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

Ans. Below is a simple **responsive navigation bar** built using **Material UI (MUI v5):**

It adapts to screen size:

- **Desktop** → shows menu items on the top bar
- **Mobile** → shows hamburger icon → opens drawer menu

1. Install Dependencies

npm install @mui/material @mui/icons-material @emotion/react @emotion/styled

2. Responsive Navbar Component

```
import React, { useState } from "react";
import {
 AppBar,
  Toolbar,
  Typography,
  IconButton,
  Button,
  Drawer,
  List,
  ListItem,
```

```
  ListItemButton,
  ListItemText,
  Box,
  useTheme,
  useMediaQuery
} from "@mui/material";
import MenuIcon from "@mui/icons-material/Menu";

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

export default function ResponsiveNavbar() {
  const [open, setOpen] = useState(false);

  const theme = useTheme();
  const isMobile = useMediaQuery(theme.breakpoints.down("md"));

  const drawer = (
    <Box sx={{ width: 250 }} onClick={() => setOpen(false)}>
      <List>
        {navItems.map((item) => (
          <ListItem key={item} disablePadding>
            <ListItemButton>
              <ListItemText primary={item} />
            </ListItemButton>
          </ListItem>
        ))}
      </List>
    </Box>
  );

  return (
    <>
      <AppBar position="static" sx={{ bgcolor: "primary.main" }}>
        <Toolbar>
```

```
        {/* Logo */}
        <Typography variant="h6" sx={{ flexGrow: 1 }}>
          MyWebsite
        </Typography>

        {/* Desktop Menu */}
        {!isMobile && (
          <Box>
            {navItems.map((item) => (
              <Button key={item} color="inherit" sx={{ ml: 2 }}>
                {item}
              </Button>
            ))}
          </Box>
        )}

        {/* Mobile Menu Icon */}
        {isMobile && (
          <IconButton color="inherit" onClick={() => setOpen(true)}>
            <MenuIcon />
          </IconButton>
        )}
      </Toolbar>
    </AppBar>

    {/* Drawer for Mobile */}
    <Drawer anchor="right" open={open} onClose={() => setOpen(false)}>
      {drawer}
    </Drawer>
  </>
 );
}
```

3. Breakpoint Behavior

Material UI default breakpoints:

Size	Width
xs	0px
sm	600px
md	900px
lg	1200px
xl	1536px

Here we used:

const isMobile = useMediaQuery(theme.breakpoints.down("md"));

➞ Screens **below 900px** show hamburger menu

➞ Screens **above 900px** show full navbar

4. Styling Applied

- sx prop for inline styling
- flexGrow: 1 pushes menu to right
- ml: 2 spacing between menu items
- Drawer provides mobile navigation panel

Result:

- Fully responsive navigation bar
- Clean Material UI styling
- Optimized for mobile and desktop

6. Evaluate and design a complete front-end 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. **Frontend Architecture Design – Collaborative Project Management Tool (with Real-Time Updates)**

Goal: Build a scalable SPA similar to Trello/Jira where multiple users can edit tasks simultaneously with real-time updates.

A) SPA Structure (Nested Routing + Protected Routes)

Folder Structure:

```
src/
 ├── app/ (store configuration)
 ├── features/
 │    ├── auth/
 │    ├── projects/
 │    ├── tasks/
 │    ├── comments/
 │    └── users/
 ├── layouts/
 │    ├── DashboardLayout.jsx
 │    └── AuthLayout.jsx
 ├── routes/
 │    └── AppRouter.jsx
 ├── components/
 │    └── common UI components
 └── services/
      └── socket.js
```

Nested Routing (React Router v6):

```
<Routes>

  {/* Public */}
  <Route element={<AuthLayout />}>
    <Route path="/login" element={<Login />} />
```

```
   <Route path="/register" element={<Register />} />
  </Route>

  {/* Protected */}
  <Route element={<ProtectedRoute />}>
   <Route path="/dashboard" element={<DashboardLayout />}>

    <Route index element={<ProjectList />} />
    <Route path="project/:projectId" element={<ProjectBoard />}>

     <Route path="task/:taskId" element={<TaskDetails />} />
    </Route>

    <Route path="settings" element={<Settings />} />
   </Route>
  </Route>

</Routes>
```

Protected Route Logic:

```
const ProtectedRoute = () => {
  const isAuth = useSelector(state => state.auth.isAuthenticated);
  return isAuth ? <Outlet />: <Navigate to="/login" replace />;
};
```

Benefits:

- Clear hierarchy
- Lazy loading possible
- Access control security

B) Global State Management (Redux Toolkit + Middleware)

Store Setup:

```
export const store = configureStore({
  reducer: {
    auth: authReducer,
    projects: projectReducer,
    tasks: taskReducer,
    realtime: socketReducer
  },
  middleware: (getDefault) =>
    getDefault().concat(socketMiddleware)
});
```

Real-Time Middleware (WebSocket):

```
const socketMiddleware = store => next => action => {
  if(action.type === "socket/connect"){
     socket.connect();
  }

  if(action.type === "task/update"){
     socket.emit("task:update", action.payload);
  }

  socket.on("taskUpdated", data => {
     store.dispatch(taskUpdatedFromServer(data));
  });

  return next(action);
};
```

Why Redux Toolkit:

- Centralized shared state
- Predictable updates
- DevTools debugging
- Middleware handles real-time sync

C) Responsive UI (Material UI + Custom Theme)

Custom Theme:

```
const theme = createTheme({
  palette: {
    primary: { main: "#3f51b5" },
    secondary: { main: "#ff9800" },
    background: { default: "#f5f6fa" }
  },
  typography: {
    fontFamily: "Inter, sans-serif"
  }
});
```

Responsive Layout:

```
const isMobile = useMediaQuery(theme.breakpoints.down("md"));

<Grid container spacing={2}>
  <Grid item xs={12} md={3}>
    <Sidebar collapse={isMobile} />
  </Grid>

  <Grid item xs={12} md={9}>
    <TaskBoard />
  </Grid>
</Grid>
```

UI Features:

- Drawer sidebar (mobile)
- Kanban board drag-drop
- Dark/light theme toggle
- Adaptive typography

D) Performance Optimization (Large Datasets)

Techniques:

1. Virtualization

Render only visible tasks

react-window / react-virtualized

2. Memoization

export default React.memo(TaskCard);

3. Normalized Redux State

```
tasks: {
  byId: {},
  allIds: []
}
```

4. Pagination + Infinite Scroll

Load 50 tasks at a time

5. Debounced Updates

Prevent API flooding

debounce(taskUpdate, 300ms)

6. Code Splitting

const ProjectBoard = lazy(() => import("./ProjectBoard"));

E) Scalability & Multi-User Concurrency Analysis

Challenges:

- Simultaneous editing
- Data conflicts

- High socket traffic
- UI freezing with many updates

Recommended Solutions:

Problem	Solution
Two users edit same task	Optimistic updates + versioning
Rapid updates	Throttled socket events
Many users online	Room-based WebSocket channels
UI flicker	Patch updates instead of full refresh
Offline edits	Conflict resolution queue
Large teams	Server event batching