

Assignment

of
Full stack-II {23CSH-382}

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
IN
COMPUTER SCIENCE WITH SPECIALIZATION IN
ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Submitted by:
23BAI70412 Kumar Aditya

Under the Supervision of:
Er. Akash Mahadev Patil



CHANDIGARH
UNIVERSITY
Discover. Learn. Empower.

**CHANDIGARH UNIVERSITY,
GHARUAN, MOHALI - 140413,**

PUNJAB

04 February , 2026

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

Frontend Design Patterns are UI related solutions which are reusable, maintainable, scalable and consistent which leads to better readability, lesser bugs from UI and help in a faster deployment.

Design patterns can be stated as a reusable solution to the common UI and architectural problems as

- Design patterns are not just code of block but conceptual templates which have proper position and scaling of the objects and components to be well structured and ready to interact
- The pattern defines the component roles, data flow and interaction rules which can be change and modify as per needs.
- It lessens the development time as it is already tested and proven and can be reused in multiple places.
- It is well documented and have a clean architecture so easier to fix and understand the code.
- It helps in faster deployment as there is no need of re-invent the logic or pattern it helps in reduce design and decision time.

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

React is a JavaScript library for a building user interface, usually for single-page applications. It helps in creating reusable UI components that manage their own state, making it easier to build dynamic and interactive web applications.

State represents dynamic data that controls a component behavior and rendering.

- Local State

It is a state that is owned and managed by a single component and is not directly accessible by other components

- ❖ It is used for UI-specific or temporary data
- ❖ It is simple and lightweight
- ❖ Doesn't affect unrelated components
- ❖ It is managed using `useState()` or `useReducer()`
- ❖ Only the component re-renders when state changes.
- ❖ Logic and data remain tightly bound to the component.
- ❖ It is applicable for small and medium components.

It is commonly used for

- Form input values
- Toggle buttons
- Dropdown selection
- Component level UI states (loading, error flags)

Limitations of Local State

- Can't be shared with distant components
- Not suitable for app wide data

- Global State

It is a state that is shared across multiple components and often across different levels of the component tree

- ❖ It is accessible by many components
- ❖ It is stored outside of individual components
- ❖ It is managed by Context API, Redux etc(state management libraries)
- ❖ Retains as long as the app is running
- ❖ It has centralized data management
- ❖ Components can access data directly
- ❖ It has consistency across UI
- ❖ It is scalable for large applications

The limitations of global state are

- It can lead to performance overhead
- It is complex
- Changes in State can affect many components
- All state shouldn't be global

The local state and global state serve different architectural purpose in React

The local state improves simplicity, performance and component isolation meanwhile the global state improves consistency, scalability and shared access.

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.

A routing strategy is an architectural approach that determines how URL navigation is handled, rendered, and managed within a web application.

There are three major routing strategies used in SPA:

1. Client-Side Routing:

In client-side routing, routing logic is handled inside the browser using JavaScript. The server always returns a single HTML file, and the frontend framework decides which component to display.

The working of CSR is like Browser loads index.html and JavaScript router listens to URL changes while corresponding component is rendered so no full page reload occurs while routing

Example

- React Router
- Vue Router
- Angular Router

Trade-Offs

- Faster navigation after initial load but Slower first load
- Reduced server load but leads to SEO challenges
- Rich SPA experience but requires JavaScript to work

Use of Client-Side Routing required when:

- Application is interactive
- SEO is not critical
- Mostly authenticated users uses

2. Server-Side Routing

In server-side routing, each URL request is handled by the server, which returns a fully rendered HTML page.

It works as when a browser requests a URL then server processes the route and HTML page is generated then Browser reloads the page

Example

- Express.js routes
- Django URLs
- Traditional PHP routing

Trade-Offs

- SEO-friendly but slower navigation due to page reloads
- Secure server logic but higher server load
- Simple rendering flow but less interactive UI
- Use of Server-Side Routing is required when:
- Content is SEO-focused
- Simplicity is required
- JavaScript may be disabled

3. Hybrid Routing

Hybrid routing combines server-side rendering (SSR) with client-side routing (CSR). Initial page is rendered on the server, and subsequent navigation is handled on the client.

It works as a server sends pre-rendered HTML then browser displays content immediately to which JavaScript hydrates the page and further navigation uses client-side routing

Example

- Next.js
- Nuxt.js

Trade-Offs

- Have excellent SEO but have high architectural complexity

- Fast initial load but can have hydration issues possible
- Rich interactivity but requires more setup and maintenance

Use of Hybrid Routing is applicable:

- When we require SEO and SPA experience
- When app is large and scalable
- When performance is critical

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

Component design patterns define standard ways to structure, reuse, and share logic among React components. They help solve recurring problems such as separation of concerns, logic reuse, and maintainability in large applications.

The three widely used patterns:

1. Container-Presentational Pattern

This pattern divides components into two types:

Container components handle data, state, and logic

Presentational components handle only UI and display

It works

Container fetches data and manages state while Presentational component receives data via props and UI and logic are kept separate

We generally use this pattern when

- When UI needs to be reused in many places
- When you want clear separation between logic and UI
- In large applications with complex data handling

2. Higher-Order Components (HOC)

A Higher-Order Component is a function that takes a component and returns a new component with added functionality.

It works as A Common logic is written once then that logic is wrapped around multiple components then those components receive extra behaviour through the wrapper.

We use them when

- Authentication and authorization
- Logging and analytics
- Reusing logic across many components

3. Render Props Pattern

The Render Props pattern shares logic by passing a function as a prop, which decides how UI should be rendered.

It works like

- One component manages logic
- Another component controls how data is displayed
- Rendering is fully customizable

We use it when

- When UI rendering needs high flexibility
- For reusable behaviour like data fetching or event tracking
- When different layouts are required for the same logic

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

```

1  import { Navbar, Nav, Container, Card, Button, Form, Row, Col } from 'react-bootstrap';
2
3  function App() {
4    const randomTexts = [
5      "Exploring creative horizons with code and design.",
6      "Building scalable solutions for modern web challenges.",
7      "Innovating one component at a time with passion."
8    ];
9
10   return (
11     <div className="d-flex flex-column min-vh-100">
12       { /* Never Expand Navbar (Fixed hamburger menu) */ }
13       <Navbar bg="dark" variant="dark" expand={false} className="shadow-sm">
14         <Container>
15           <Navbar.Brand href="#home" className="fw-bold">Exp-4: Kumar Aditya</Navbar.Brand>
16           <Navbar.Toggle aria-controls="offcanvasNavbar" />
17           <Navbar.Collapse id="offcanvasNavbar">
18             <Nav className="ms-auto p-3">
19               <Nav.Link href="#home">Home</Nav.Link>
20               <Nav.Link href="#profile">Profile</Nav.Link>
21               <Nav.Link href="#settings">Settings</Nav.Link>
22               <Nav.Link href="#contact">Contact</Nav.Link>
23             </Nav>
24           </Navbar.Collapse>
25         </Container>
26       </Navbar>
27     </div>
  
```

6) 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.

A collaborative project management tool requires:

- Smooth navigation without reloads
- Secure access control
- Real-time multi-user updates
- Scalable global state handling
- High performance with large datasets

A Single Page Application (SPA) architecture with Redux Toolkit, Material UI, and real-time communication is suitable for this use case.

(a) SPA Structure with Nested Routing and Protected Routes

SPA Structure

The application is structured as a Single Page Application to ensure:

- Fast navigation
- Smooth user experience
- No full page reloads

The routes can be expressed as :

Public Routes

- Login
- Register

Protected Routes

- Dashboard
- Projects
- Project Overview
- Task Board
- Team Members
- Profile
- Settings

The nested routing is used to:

- Keep parent layouts persistent (navbar, sidebar)
- Load child views dynamically
- Improve code organization

b) Global State Management using Redux Toolkit with Middleware

The global state is required as in a collaborative tool, many components need shared data

- User authentication
- Project lists
- Tasks and comments
- Real-time updates
- Notifications

Redux Toolkit Architecture

Slices for modular state management

- authSlice
- projectsSlice
- tasksSlice
- usersSlice
- notificationsSlice

Store as a single source of truth

Middleware Usage

Middleware handles side effects and async logic:

- Thunk Middleware
 - API calls (fetch projects, tasks)
 - Error handling
 - Loading states
- WebSocket / Real-Time Middleware
 - Sync updates across users
 - Handle events like;
 - Task updates
 - Comments added
 - Status changes

C) Responsive UI Design using Material UI with Custom Theming

UI Design Requirement

- Works on desktop, tablet and mobile
- Consistent branding
- Accessibility support

Material UI Usage

Material UI is used because it provides:

- Prebuilt components (AppBar, Drawer, Cards)
- Responsive breakpoints
- Accessibility compliance

Responsive Layout Strategy

- Sidebar collapses into drawer on small screens
- Task boards switch from grid to list on mobile
- Breakpoints used: xs, sm, md, lg

Custom Theming

Custom theme includes:

- Brand colors
- Typography styles
- Dark / light mode support
- Consistent spacing and shadows

d) Performance Optimization Techniques for Large Datasets

Project management tools often handle, hundreds of tasks, multiple users and frequent updates these processes require optimization

Optimization Techniques

1. List Virtualization

- Render only visible items in task lists
- Reduces DOM size and memory usage

2. Memoization

- Prevent unnecessary re-renders
- Use memoized selectors and components

3. Normalized State Structure

- Store entities by ID
- Avoid deep nested objects
- Faster lookups and updates

4. Lazy Loading

- Load routes and components only when needed
- Reduces initial bundle size

5. Debouncing and Throttling

- Limit frequent updates (search, drag events)
- Improves responsiveness

e) Scalability Analysis and Recommendations for Multi-User Concurrent Access

Challenges faced in Multi-User Collaboration can be conflicting updates, State synchronization, Network latency, High event frequency

Scalability Analysis

Current Strengths

- Centralized state management
- Real-time updates via WebSockets
- Modular routing and UI structure

Potential Bottlenecks

- Redux store overload
- Frequent re-renders
- WebSocket event flooding

Recommended Improvements

1. Event-Based State Updates
 - Update only affected entities
 - Avoid full state refresh
2. Optimistic UI Updates
 - Update UI before server confirmation
 - Roll back if update fails
3. Role-Based Data Fetching
 - Load only relevant data per user role
 - Reduces payload size
4. State Partitioning
 - Separate real-time state from static state
 - Improves maintainability
5. Horizontal Scaling Support
 - Design frontend to handle:
 - Multiple WebSocket connections
 - Reconnection strategies
 - Conflict resolution feedback