



Apex Institute of Technology

Assignment 1

Student Name: Yash Gawali

UID: 23BAI70093

Branch: Computer Science & Engineering

Section/Group: 23AML-2 (B)

Semester: 6

Date of Performance: 12/02/2026

Subject Name: Full Stack Development - II

Questions:

1. Summarize the benefits of using design patterns in frontend development.
2. Classify the difference between global state and local state in React.
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.
4. Examine common component design patterns such as Container–Presentational, Higher-Order Components, and Render Props, and identify appropriate use cases for each pattern.
5. Demonstrate and develop a responsive navigation bar using Material UI components while applying appropriate styling and breakpoint configurations.
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.



Apex Institute of Technology

Answers:

Answer 1.

Design patterns are standardized, reusable solutions to recurring software design problems. In frontend development, they provide proven architectural approaches for structuring components, managing state, handling data flow, and organizing application logic. Rather than solving similar problems from scratch each time, developers rely on established patterns such as MVC, Flux, Container–Presentational, and Observer to ensure consistency, scalability, and maintainability. These patterns serve as blueprints that guide implementation decisions and improve software quality.

a) Maintainability

Design patterns encourage a clear structure and separation of responsibilities within an application. When code follows consistent architectural principles, it becomes easier to understand, debug, and modify. Future developers (or team members) can quickly grasp how components interact because the structure follows recognizable standards. This significantly reduces technical debt and makes long-term maintenance more efficient.

b) Reusability

Patterns promote modular design by breaking applications into smaller, independent components. Reusable UI components, hooks, or logic modules reduce redundancy and duplication across the application. For example, using component-based patterns in React allows developers to reuse the same navigation bar, form logic, or modal component in multiple parts of the system, improving development speed and consistency.

c) Scalability

As frontend applications grow in size and complexity, managing data flow and state becomes challenging. Architectural patterns such as Flux or Redux introduce predictable state management and unidirectional data flow. This prevents tightly coupled components and uncontrolled data mutations, making it easier to scale applications without introducing instability or performance issues.

d) Separation of Concerns

Design patterns clearly define boundaries between different parts of an application, such as UI rendering, business logic, and data management. For example, the Container–Presentational pattern separates data handling from visual components. This improves code clarity, allows independent development of logic and UI, and enhances testability by isolating responsibilities.



Apex Institute of Technology

e) Team Collaboration

When teams follow established design patterns, they share a common architectural language. This reduces onboarding time for new developers and improves communication among team members. Instead of debating structural decisions repeatedly, developers can rely on agreed-upon patterns, leading to more consistent codebases and smoother collaboration.

f) Testability

Design patterns often encourage decoupled and modular structures, which are easier to test. For example, separating business logic from UI components allows developers to write unit tests for logic without rendering the interface. Predictable state management patterns also simplify integration testing by ensuring deterministic behaviour.

Answer 2.

Aspect	Local State	Global State
Scope	Limited to a single component and shared only through props. Best for isolated component behavior.	Shared across multiple components in the application without prop drilling. Suitable for app-wide data.
Management	Managed with React hooks like <code>useState</code> and <code>useReducer</code> . Simple and requires no extra setup.	Managed with tools like Context API, Redux, or Redux Toolkit. Requires centralized store configuration.
Use Case	Used for UI-specific data such as form inputs, toggles, and temporary states.	Used for shared data such as authentication, user info, themes, and project data.
Performance Impact	Re-renders only the component where the state changes, making it efficient.	May trigger multiple component re-renders if not optimized with selectors or memoization.
Complexity	Easy to implement and ideal for small features.	More complex setup but beneficial for large-scale applications needing consistent shared state.



Apex Institute of Technology

Answer 3.

Routing determines how users navigate between views in a web application. In SPAs, routing can be handled on the client, the server, or through a hybrid approach. Each strategy has trade-offs in performance, SEO, scalability, and complexity.

1) Client-Side Routing (CSR)

In client-side routing, navigation is handled entirely in the browser using JavaScript frameworks such as React Router. The server typically serves a single HTML file, and content updates dynamically without full page reloads.

Advantages:

- Fast navigation after the initial load
- Smooth user experience (no full refresh)
- Reduced server workload

Disadvantages:

- Larger initial bundle size
- SEO challenges (unless pre-rendering is used)
- Slower first contentful paint on low-end devices

Best Use Cases:

Dashboards, SaaS platforms, admin panels, and applications where SEO is not critical.

2) Server-Side Routing (SSR)

With server-side routing, each request is processed by the server, which renders a new HTML page and sends it to the client. Traditional multi-page applications use this approach.

Advantages:

- Better SEO (search engines receive fully rendered HTML).
- Faster initial page load for users.
- Improved performance on low-powered devices.



Apex Institute of Technology

Disadvantages:

- Full page reloads on navigation.
- Higher server load.
- Less dynamic user experience.

Best Use Cases:

Content-heavy websites, blogs, news portals, and marketing pages where SEO and fast first load are priorities.

3) Hybrid Routing (CSR + SSR)

Hybrid routing combines both approaches. Frameworks like Next.js allow pages to be rendered on the server initially and then behave like a client-side SPA after hydration.

Advantages:

- Combines SEO benefits of SSR with smooth transitions of CSR.
- Flexible rendering strategies (SSR, SSG, ISR).
- Optimized performance for large applications.

Disadvantages:

- Increased architectural complexity.
- Requires more configuration and server setup.

Best Use Cases:

Large-scale applications requiring both strong SEO and interactive user experiences, such as e-commerce platforms or enterprise systems.



Apex Institute of Technology

Answer 4.

1) Container–Presentational Pattern

Container: Handles logic & state

Presentational: Handles UI

Use Case: Large applications where separation improves maintainability.

Example:

- UserContainer fetches data.
- UserList renders UI.

2) Higher-Order Components (HOC)

A function that takes a component and returns an enhanced component.

```
const withAuth = (Component) => (props) =>
```

```
  isAuthenticated ? <Component {...props} /> : <Login />;
```

Use Case: Cross-cutting concerns (auth, logging).

3) Render Props

Component shares logic via a function prop.

```
<DataFetcher render={({data}) => <List data={data} /> } />
```

Use Case: Sharing dynamic behavior between components.



Apex Institute of Technology

5.

The following code was used for a Single Page Application (SPA) consisting of a Profile page and a Dashboard page. The page has a navbar that allows the user to navigate between the two components of the SPA.

```
//App.jsx
```

```
import './App.css';
```

```
import { BrowserRouter as Router, Route, Routes, Link } from 'react-router-dom';
```

```
import Dashboard from './components/dashboard';
```

```
import Profile from './components/profile';
```

```
import Home from './components/home';
```

```
function App() {
```

```
  return (
```

```
    <Router>
```

```
      <div className="App">
```

```
        <nav className="navbar">
```

```
          <ul className="nav-list">
```

```
            <li>
```

```
              <Link to="/">Home</Link>
```

```
            </li>
```

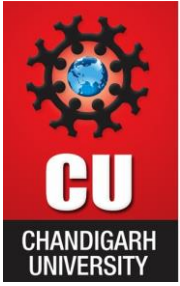
```
            <li>
```

```
              <Link to="/dashboard">Dashboard</Link>
```

```
            </li>
```

```
            <li>
```

```
              <Link to="/profile">Profile</Link>
```



Apex Institute of Technology

```
    </li>

  </ul>

</nav>

<Routes>

  <Route path="/" element={<Home />} />

  <Route path="/dashboard" element={<Dashboard />} />

  <Route path="/profile" element={<Profile />} />

</Routes>

</div>

</Router>

);

}

export default App;

//Dashboard.jsx

import React from "react";
import "./dashboard.css";

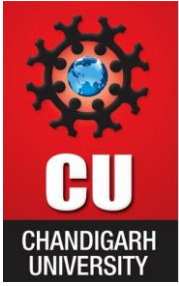
function Dashboard() {

  const skills = [

    "React",

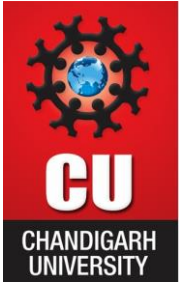
    "JavaScript",

    "HTML/CSS",
```

Apex Institute of Technology

```
"Node.js",  
  
"Python",  
  
"Git",  
  
"Blender",  
  
];  
  
return (  
  
  <div className="page">  
  
    <h1 style={{color:"#ffffff"}}>Skills Dashboard</h1>  
  
    <p style={{color: "#ffffff"}}>Welcome to the dashboard page!</p>  
  
    <div className="skills-container">  
  
      {skills.map((skill, index) => (  
  
        <div key={index} className="skill-box">  
  
          {skill}  
  
        </div>  
  
      ))}  
  
    </div>  
  
  </div>  
  
);  
  
}  
  
export default Dashboard;  
  
//Profile.jsx
```



Apex Institute of Technology

```
import React from "react";

import "./profile.css";

function Profile() {

  const projects = [

    {

      name: "Dream Partner",

      description:

        "A game platform to find your ideal partner based on compatibility.",

      language: "Godot, React, Node.js, Python",

    },

    {

      name: "Green prompt",

      description: "Making the world better one prompt at a time",

      language: "Python, Electron, React",

    },

    {

      name: "Muck",

      description:

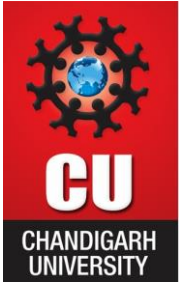
        "A website where you can collab with your peers to make documents",

      language: "React, Drizzle, Node.js, Docker",

    },

  ];

  return (
```



Apex Institute of Technology

```
<div className="profile-container">

  <div className="profile-sidebar">

    <div className="profile-avatar">

    </div>

    <h1 className="profile-name">Yash </h1>

    <a href="https://github.com/Dueyash">@Dueyash</a>

    <p className="profile-bio">

      Full-stack developer passionate about building web applications. Love

      React, Node.js, and open source.

    </p>

  </div>

  <div className="profile-main">

    <h2>Projects</h2>

    <div className="projects-grid">

      {projects.map((project, index) => (

        <div key={index} className="project-card">

          <h3>{project.name}</h3>

          <p className="project-description">{project.description}</p>

          <div className="project-footer">

            <span className="project-language">

              <span className="language-dot"></span>

              {project.language}

            </span>

          </div>

        </div>

      )

    )}

  </div>

</div>
```



Apex Institute of Technology

</div>

</div>

))}

</div>

</div>

</div>

);

}

export default Profile;

6.

a) SPA Structure

Routing:

- /login
- /dashboard
- /projects/:id
- /projects/:id/tasks
- Protected routes via RequireAuth wrapper

<Route element={<RequireAuth />}>

<Route path="/dashboard" element={<Dashboard />} />

</Route>

Use React Router v6 nested routing.



Apex Institute of Technology

b) Global State (Redux Toolkit)

Slices:

- authSlice
- projectSlice
- taskSlice
- notificationSlice

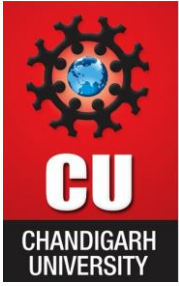
Middleware:

- Redux Thunk (async logic)
- WebSocket middleware (real-time updates)
- Logger (dev only)

```
configureStore({  
  reducer: { auth, projects, tasks },  
  middleware: (getDefaultMiddleware) =>  
    getDefaultMiddleware().concat(websocketMiddleware),  
});
```

c) Responsive UI with MUI + Custom Theme

```
const theme = createTheme({  
  palette: {  
    primary: { main: "#1976d2" },  
    secondary: { main: "#ff9800" },  
  },  
  typography: {
```



Apex Institute of Technology

```
fontFamily: "Roboto",  
  
},  
  
});
```

Use:

- Grid for layout.
- Drawer for sidebar.
- Dark/light mode toggle.

d) Performance Optimization

- Code splitting (React.lazy).
- Memoization (React.memo, useMemo).
- Virtualized lists (react-window).
- Debounced search.
- Pagination or infinite scroll.
- Normalized Redux state.

e) Scalability & Multi-User Concurrency

Real-Time:

- WebSockets (Socket.io).
- Optimistic UI updates.

Concurrency Handling:

- Versioning or timestamp conflict resolution.
- Server-authoritative state.

Scalability Improvements:

- Micro-frontend architecture (if large).



Apex Institute of Technology

- CDN for static assets.
- Caching with RTK Query.
- Load balancing backend.
- Event-driven updates.

Architectural Overview (High-Level)

Frontend:

- React + Redux Toolkit
- React Router.
- Material UI.
- WebSocket client.

Backend (assumed):

- REST + WebSocket server.
- Database with real-time sync.