Routing, Data Fetching & State Management

React + TypeScript (Vite)

Why This Module Matters

Modern applications require:

- Multiple pages and layouts
- Reliable API communication
- Shared and predictable state

Without proper patterns:

- Apps become hard to maintain
- Bugs increase as features grow
- Performance degrades

📌 This module introduces app-level architecture

Routing in React

Routing allows:

- Navigation between views
- URL-based state
- Deep linking & bookmarking

Key Features:

- Client-side routing
- Dynamic routes
- Nested layouts

📌 React apps behave like multi-page apps without reloads

React Router Concepts

Core building blocks:

- Router provider
- Routes & Route definitions
- Link & NavLink
- Route parameters

Best Practices:

- Centralized route configuration
- Clear route naming
- Layout-based routing

Routing with TypeScript

TypeScript advantages:

- Typed route params
- Safer navigation
- Fewer runtime errors

Design Considerations:

- Consistent route structures
- Typed navigation helpers
- Error boundaries for routes

Common Routing Pitfalls

❌ Hard-coding paths everywhere
❌ Overusing nested routes
❌ Missing error or fallback routes
❌ Mixing routing logic with UI logic

✅ Centralize routes
✅ Separate layout and page components

What Is Data Fetching?

Data Fetching means:

- Retrieving data from APIs
- Syncing UI with backend state
- Handling async operations

Common Sources:

- REST APIs
- Internal services
- Mock servers (during development)

Data Fetching Patterns

Common patterns:

- Fetch on component mount
- Fetch on route change
- Fetch on user action

Key Considerations:

- Loading states
- Error handling
- Caching strategies

Data Fetching with TypeScript

Why TypeScript is critical:

- Typed API responses
- Safer transformations
- Predictable UI rendering

Best Practices:

- Define API response interfaces
- Handle optional/null values
- Centralize API logic

Data Fetching Anti-Patterns

❌ Fetching data inside render
❌ Duplicate API calls
❌ Ignoring loading/error states
❌ Mixing fetch logic with UI

✅ Separate data layer
✅ Reuse fetching logic

What Is State Management?

State Management handles:

- Shared data across components
- Global application state
- Predictable updates

Types of State:

- Local component state
- Shared state
- Global state

When State Becomes a Problem

State issues arise when:

- Multiple components depend on same data
- Prop drilling becomes deep
- Updates become hard to track

📌 This is where structured state management helps

State Management Approaches

Common approaches:

- Lifting state up
- Context API
- External state libraries

Choosing depends on:

- App size
- Data complexity
- Team familiarity

State Management with TypeScript

Benefits:

- Strongly typed state models
- Safer reducers and actions
- Easier refactoring

Best Practices:

- Define clear state interfaces
- Avoid overly large global state
- Keep state normalized

Integration Points & Dependencies

Key integration areas:

- Backend APIs
- Authentication systems
- Environment configs (Vite)
- Error monitoring tools

Design Impact:

- State ownership
- Performance
- Security

Common Anti-Patterns

❌ Storing server data as global state unnecessarily
❌ Over-fetching data
❌ Tight coupling between routes and API logic
❌ Ignoring caching strategies

✅ Fetch close to where data is used
✅ Share state only when necessary

Preparing Assets for Capstone

You should prepare:

- Typed route configuration
- Reusable API service layer
- Shared state models
- Loading & error UI patterns

These assets will directly support:

👉 Capstone Application Architecture

Module Summary

- Routing enables navigation and structure
- Data fetching connects UI to backend
- State management ensures consistency
- TypeScript enforces correctness at scale

This module completes the core application flow