React Application Development with TypeScript

Duration: 5 Days | **40 Hours**

Table of contents

Day 1 – Advanced TypeScript & Modern React

Module 1: Advanced TypeScript for React Development

- Role of TypeScript in enterprise React applications
- Type inference and type annotations
- Primitive, object, array and function types
- Interfaces vs Type Aliases
- Optional, readonly and index properties
- Union and Intersection Types
- Literal Types and Enums
- Tuples and readonly tuples
- any vs unknown vs never
- Type assertions and type narrowing
- Type Guards and custom type predicates
- Generics and generic constraints
- Utility Types
    - Partial
    - Required
    - Pick
    - Omit
    - Record
    - Readonly
- keyof, typeof and indexed access types
- Discriminated unions
- Typing functions, callbacks and event handlers
- Type-safe module organization

- TypeScript configuration for React

Lab Exercise 1: Build a Type-Safe Employee Domain

Scenario:
Create the TypeScript foundation for an enterprise Employee Management Portal.

Tasks:

1. Define Employee, Department, Address and Role interfaces.
2. Create union types for employee status and user roles.
3. Use optional and readonly properties where appropriate.
4. Create generic types for API responses.
5. Use Partial, Pick and Omit to create employee create/update models.
6. Implement a type guard to distinguish Admin, Manager and Employee users.
7. Use discriminated unions for different employee actions.
8. Create strongly typed utility functions for employee filtering and sorting.
9. Configure TypeScript using strict compiler options.
10. Organize the models and utilities into separate modules.

Expected Outcome:
A strongly typed domain layer that can be reused throughout the React application.

Module 2: Modern React & Component Architecture

- Creating React + TypeScript applications with Vite
- Functional components and JSX
- Props and children typing
- Component composition
- Component boundaries and responsibilities
- Reusable and configurable components
- Controlled vs uncontrolled components
- Event handling with TypeScript
- Conditional and list rendering
- useState
- useReducer

- useRef
- DOM references
- useEffect
- Effect cleanup and dependency management
- Custom Hooks
- Designing reusable typed Hooks
- React Strict Mode
- Modern React development practices

Lab Exercise 2: Create a Reusable Enterprise UI Component Set

Scenario:
Develop the reusable UI foundation for the Employee Management Portal.

Tasks:

1. Create the React + TypeScript project using Vite.
2. Create reusable components:
    - Button
    - Input
    - Card
    - Modal
    - Badge
    - DataTable
3. Define strongly typed props for each component.
4. Implement typed event handlers.
5. Build an employee listing component.
6. Use useState to manage search and selection.
7. Use useReducer for complex employee UI state.
8. Create a reusable useLocalStorage<T> custom Hook.
9. Use useRef for input focus management.
10. Implement useEffect for synchronizing application state.

Expected Outcome:
Participants create a reusable, strongly typed component foundation for an enterprise React application.

Day 2 – Advanced React Patterns, Architecture & Routing

Module 3: Advanced React Patterns

- Container vs Presentational Components
- Composition over inheritance
- Compound Components
- Render Props
- Higher-Order Components
- Custom Hook patterns
- Headless component architecture
- Reusable component APIs
- Generic React components
- Type-safe component APIs
- Polymorphic components
- Discriminated-union component APIs
- Error Boundaries

Lab Exercise 3: Build Reusable Advanced React Components

Scenario:
The enterprise application needs reusable components that can support multiple business modules.

Tasks:

1. Create a reusable Compound Component for employee filters.
2. Implement a reusable Tabs component using composition.
3. Create a generic DataTable<T> component.
4. Add configurable columns using TypeScript generics.
5. Create a reusable Modal component.
6. Implement a custom Hook for modal state.

7. Create a reusable notification component.
8. Implement an Error Boundary around the application modules.
9. Demonstrate the difference between container and presentational components.
10. Refactor duplicated component logic into reusable Hooks.

Expected Outcome:
Participants will understand how to design reusable and scalable React components suitable for enterprise applications.

Module 4: Enterprise React Architecture

- Feature-based architecture
- Separation of UI, business logic and data-access layers
- Scalable folder structure
- Reusable component architecture
- SOLID principles for React
- Enterprise React project structure
- Design-system concepts
- Shared components and utilities
- Managing large-scale React applications

Lab Exercise 4: Refactor a React Application into Enterprise Architecture

Scenario:
A growing Employee Management Portal has become difficult to maintain because components, API logic and business logic are mixed together.

Tasks:

1. Create a feature-based folder structure.
2. Separate:
    - Components
    - Hooks
    - Services
    - Models
    - Utilities
    - Pages

3. Create separate employee and task features.
4. Move API-related logic into a service layer.
5. Separate business logic from presentation components.
6. Create shared UI components.
7. Create shared TypeScript models.
8. Apply SOLID principles where applicable.
9. Establish coding conventions for scalable development.
10. Refactor existing components into the new architecture.

Expected Outcome:
A maintainable enterprise React project structure that can scale as new business features are added.

Module 5: React Router & Application Navigation

- React Router architecture
- Application routes
- Nested routes
- Dynamic route parameters
- Query parameters and URL state
- Programmatic navigation
- Route-based layouts
- Protected routes
- Authentication-aware routing
- Role-based route authorization
- Route loaders and actions
- Error and Not Found routes
- Redirects
- Lazy-loaded routes
- Route-level code splitting
- Suspense-based loading

Lab Exercise 5: Implement Multi-Role Application Navigation

Scenario:
The Employee Management Portal has three types of users: **Admin, Manager and Employee**. Each role must have different access.

Tasks:

1. Configure routes for Dashboard, Employees, Tasks and Reports.
2. Create nested employee routes.
3. Implement dynamic employee detail routes.
4. Add query parameters for employee filtering.
5. Create protected routes.
6. Implement role-based route authorization.
7. Create separate navigation menus for each role.
8. Implement redirects for unauthorized users.
9. Add Not Found and error routes.
10. Lazy-load the Employee and Reports modules.
11. Implement route-level loading using Suspense.

Expected Outcome:
A complete role-aware navigation system with protected and lazy-loaded routes.

Day 3 – Forms, State Management & REST APIs

Module 6: Type-Safe Forms & Validation

- Modern form architecture
- Controlled vs uncontrolled forms
- React Hook Form
- Type-safe form models
- Zod schema validation
- Type inference
- Field-level validation
- Form-level validation
- Dynamic forms

- Nested form fields
- Conditional fields
- Async validation
- Server-side validation errors
- File upload forms
- Accessible forms

Lab Exercise 6: Build an Employee Onboarding Form

Scenario:
HR needs a digital employee onboarding form to register new employees.

Tasks:

1. Build an employee registration form using React Hook Form.
2. Define a strongly typed employee model.
3. Create a Zod validation schema.
4. Validate:
    - Employee name
    - Email
    - Phone
    - Department
    - Designation
    - Joining date
    - Salary
5. Implement field-level error messages.
6. Add conditional fields based on employee type.
7. Implement nested address fields.
8. Add profile photo upload.
9. Implement asynchronous email uniqueness validation.
10. Display server-side validation errors.
11. Create reusable form input components.

Expected Outcome:
A production-style, type-safe employee onboarding form with comprehensive validation.

Module 7: State Management with Redux Toolkit

- Local state vs global state
- Redux architecture
- Redux Toolkit
- Store configuration
- Slices and reducers
- Actions and action creators
- Typed useSelector and useDispatch
- Normalized application state
- createAsyncThunk
- Async state handling
- Loading, success and error states
- Redux middleware
- Redux DevTools
- Redux Toolkit best practices
- RTK Query overview
- Redux vs Context vs local state

Lab Exercise 7: Implement Enterprise State Management

Scenario:
The Employee Portal needs centralized state management for employee, task and application settings.

Tasks:

1. Configure a Redux Toolkit store.
2. Create an employee slice.
3. Create a task slice.
4. Define strongly typed Redux hooks.

5. Implement employee CRUD state operations.
6. Manage loading, success and error states.
7. Implement asynchronous operations using createAsyncThunk.
8. Normalize employee-related state.
9. Use Redux DevTools to inspect state changes.
10. Identify which state should remain local and which should be global.
11. Implement middleware for application-level logging or monitoring.

Expected Outcome:
A scalable Redux Toolkit state architecture with strongly typed state management.

Module 8: REST API Integration

- RESTful API architecture
- HTTP methods and status codes
- Type-safe API request and response models
- Fetch vs Axios
- Axios instances
- Axios interceptors
- API service layer
- Request/response transformation
- Centralized API error handling
- Request cancellation
- Pagination
- Filtering and sorting
- Search APIs
- File upload/download

Lab Exercise 8: Integrate Employee REST APIs

Scenario:
Connect the Employee Management Portal to a RESTful backend.

Tasks:

1. Define TypeScript interfaces for API requests and responses.
2. Create an Axios instance.
3. Create a centralized employee API service.
4. Implement:
    - Get employees
    - Get employee by ID
    - Create employee
    - Update employee
    - Delete employee
5. Configure Axios interceptors.
6. Implement centralized error handling.
7. Implement API loading states.
8. Add employee search and filtering.
9. Add server-side pagination.
10. Implement request cancellation.
11. Handle API validation errors in React forms.

Expected Outcome:
A fully integrated React frontend communicating with a REST API through a type-safe service layer.

Day 4 – Server State, Authentication & Security

Module 9: TanStack Query / Server State Management

- Client state vs server state
- Query and mutation concepts
- Query keys
- Caching and stale data

- Background refetching
- Mutations
- Query invalidation
- Optimistic updates
- Pagination
- Infinite queries
- Prefetching
- Dependent queries
- Retry and error handling

Lab Exercise 9: Implement Server State with TanStack Query

Scenario:
The Employee Portal receives frequent updates from the backend. The application should minimize unnecessary API requests while keeping data fresh.

Tasks:

1. Configure TanStack Query.
2. Create employee queries.
3. Define reusable query keys.
4. Implement employee mutations.
5. Configure caching and stale times.
6. Implement background refetching.
7. Invalidate queries after employee updates.
8. Implement optimistic updates.
9. Add server-side pagination.
10. Implement dependent queries.
11. Add retry and error-handling strategies.
12. Implement prefetching for employee detail pages.

Expected Outcome:
Participants will implement efficient server-state management with caching, mutations and background synchronization.

Module 10: Authentication & Authorization

- Authentication vs Authorization
- Login/logout architecture
- JWT authentication
- Access and refresh token concepts
- Protected API calls
- Role-Based Access Control
- Permission-based UI rendering
- Secure token handling
- Session management
- Authentication failure handling
- Protected routes

Lab Exercise 10: Implement JWT Authentication & RBAC

Scenario:
Secure the Employee Management Portal so that users can access only the features permitted by their roles.

Tasks:

1. Create a login page.
2. Authenticate users against an API.
3. Implement JWT-based authentication.
4. Manage authentication state.
5. Protect application routes.
6. Implement Admin, Manager and Employee roles.
7. Create permission-based UI elements.
8. Attach authentication credentials to API requests.
9. Handle expired/invalid authentication.
10. Implement logout.
11. Redirect unauthorized users.
12. Maintain authenticated application state.

Expected Outcome:
A secure authentication and authorization workflow supporting role-based enterprise access.

Module 11: React Application Security

- CORS considerations
- XSS protection
- CSRF concepts
- Secure form handling
- Sensitive data protection
- API security considerations
- Frontend security best practices

Lab Exercise 11: Perform a React Security Hardening Exercise

Scenario:
Perform a security review of the Employee Portal before it is released to production.

Tasks:

1. Identify sensitive information exposed in frontend code.
2. Review authentication/token handling.
3. Identify potential XSS risks.
4. Review user-generated content handling.
5. Analyze CORS configuration requirements.
6. Review form security.
7. Identify insecure API usage patterns.
8. Implement appropriate frontend security controls.
9. Remove sensitive debugging information.
10. Create a frontend security checklist.

Expected Outcome:
Participants will identify common React application security risks and apply practical mitigation techniques.

Day 5 – Performance, Testing & Production Readiness

Module 12: React Performance Optimization

- Understanding React rendering
- Component re-render analysis
- React DevTools Profiler
- React.memo
- useMemo
- useCallback
- Avoiding unnecessary renders
- State placement strategies
- Context performance
- Code splitting
- Lazy loading
- Suspense
- Dynamic imports
- Bundle optimization
- Tree shaking
- Virtualized lists
- Image and asset optimization
- Core Web Vitals
- Production performance troubleshooting

Lab Exercise 12: Optimize an Enterprise React Dashboard

Scenario:
The Employee Portal has become slow because it displays thousands of employees and contains several data-heavy components.

Tasks:

1. Use React DevTools Profiler to identify slow components.
2. Identify unnecessary component re-renders.
3. Apply React.memo where appropriate.

4. Optimize expensive calculations using useMemo.
5. Optimize callback references using useCallback.
6. Implement lazy loading for application modules.
7. Configure route-level code splitting.
8. Implement list virtualization for large employee datasets.
9. Analyze the application bundle.
10. Optimize static assets.
11. Review Core Web Vitals.
12. Measure performance before and after optimization.

Expected Outcome:
A significantly optimized React application with measurable improvements in rendering and loading performance.

Module 13: Testing React Applications

- React testing strategy
- Unit testing
- Integration testing
- End-to-end testing
- React Testing Library
- Testing user interactions
- Testing custom Hooks
- Mocking REST APIs
- MSW (Mock Service Worker)
- Testing Redux state
- Testing forms and validation
- Testing routing
- Playwright/Cypress overview

Lab Exercise 13: Create an Automated Test Suite

Scenario:
Before production deployment, the Employee Portal must have automated tests covering its critical business workflows.

Tasks:

1. Configure React Testing Library.
2. Create component tests.
3. Test employee registration.
4. Test form validation.
5. Test login functionality.
6. Test protected routes.
7. Test Redux state changes.
8. Mock REST APIs using MSW.
9. Test employee CRUD operations.
10. Test custom Hooks.
11. Test role-based UI behavior.
12. Create an end-to-end test for the employee onboarding workflow using Playwright/Cypress.

Expected Outcome:
A reliable automated test suite covering critical application functionality.

Module 14: Code Quality & Production Practices

- TypeScript strict mode
- ESLint
- Prettier
- Code quality standards
- Environment configuration
- Production builds
- Source maps and debugging

Lab Exercise 14: Prepare the Application for Production

Scenario:
The Employee Management Portal is ready for release and needs to pass a production-readiness checklist.

Tasks:

1. Enable TypeScript strict mode.
2. Configure ESLint.
3. Configure Prettier.
4. Fix linting and formatting issues.
5. Configure environment-specific variables.
6. Create development and production configurations.
7. Generate an optimized production build.
8. Analyze the production bundle.
9. Run the complete automated test suite.
10. Configure build and test validation for CI/CD.
11. Perform a final code-quality review.
12. Prepare the application for deployment.

Expected Outcome:
A clean, tested, optimized and production-ready React + TypeScript application.