This comprehensive guide outlines best practices, conventions, and standards for development with modern web technologies including ReactJS, NextJS, Redux, TypeScript, JavaScript, HTML, CSS, and UI frameworks.

Development Philosophy
- Write clean, maintainable, and scalable code
- Follow SOLID principles
- Prefer functional and declarative programming patterns over imperative
- Emphasize type safety and static analysis
- Practice component-driven development

Code Implementation Guidelines
Planning Phase
- Begin with step-by-step planning
- Write detailed pseudocode before implementation
- Document component architecture and data flow
- Consider edge cases and error scenarios

Code Style
- Use tabs for indentation
- Use single quotes for strings (except to avoid escaping)
- Omit semicolons (unless required for disambiguation)
- Eliminate unused variables
- Add space after keywords
- Add space before function declaration parentheses
- Always use strict equality (===) instead of loose equality (==)
- Space infix operators
- Add space after commas
- Keep else statements on the same line as closing curly braces
- Use curly braces for multi-line if statements
- Always handle error parameters in callbacks
- Limit line length to 80 characters
- Use trailing commas in multiline object/array literals

Naming Conventions
General Rules
- Use PascalCase for:

  - Components
  - Type definitions
  - Interfaces
- Use kebab-case for:
  - Directory names (e.g., components/auth-wizard)
  - File names (e.g., user-profile.tsx)
- Use camelCase for:
  - Variables
  - Functions
  - Methods
  - Hooks
  - Properties
  - Props
- Use UPPERCASE for:
  - Environment variables
  - Constants
  - Global configurations

Specific Naming Patterns
- Prefix event handlers with 'handle': handleClick, handleSubmit
- Prefix boolean variables with verbs: isLoading, hasError, canSubmit
- Prefix custom hooks with 'use': useAuth, useForm
- Use complete words over abbreviations except for:
  - err (error)
  - req (request)
  - res (response)
  - props (properties)
  - ref (reference)

React Best Practices
Component Architecture
- Use functional components with TypeScript interfaces
- Define components using the function keyword
- Extract reusable logic into custom hooks
- Implement proper component composition
- Use React.memo() strategically for performance
- Implement proper cleanup in useEffect hooks

React Performance Optimization

- Use useCallback for memoizing callback functions
- Implement useMemo for expensive computations
- Avoid inline function definitions in JSX
- Implement code splitting using dynamic imports
- Implement proper key props in lists (avoid using index as key)

Next.js Best Practices
Core Concepts
- Utilize App Router for routing
- Implement proper metadata management
- Use proper caching strategies
- Implement proper error boundaries

Components and Features
- Use Next.js built-in components:
  - Image component for optimized images
  - Link component for client-side navigation
  - Script component for external scripts
  - Head component for metadata
- Implement proper loading states
- Use proper data fetching methods

Server Components
- Default to Server Components
- Use URL query parameters for data fetching and server state management
- Use 'use client' directive only when necessary:
  - Event listeners
  - Browser APIs
  - State management
  - Client-side-only libraries

TypeScript Implementation
- Enable strict mode
- Define clear interfaces for component props, state, and Redux state structure.
- Use type guards to handle potential undefined or null values safely.
- Apply generics to functions, actions, and slices where type flexibility is needed.

- Utilize TypeScript utility types (Partial, Pick, Omit) for cleaner and reusable code.
- Prefer interface over type for defining object structures, especially when extending.
- Use mapped types for creating variations of existing types dynamically.

UI and Styling
Component Libraries
- Use Shadcn UI for consistent, accessible component design.
- Integrate Radix UI primitives for customizable, accessible UI elements.
- Apply composition patterns to create modular, reusable components.

Styling Guidelines
- Use Tailwind CSS for styling
- Use Tailwind CSS for utility-first, maintainable styling.
- Design with mobile-first, responsive principles for flexibility across devices.
- Implement dark mode using CSS variables or Tailwind's dark mode features.
- Ensure color contrast ratios meet accessibility standards for readability.
- Maintain consistent spacing values to establish visual harmony.
- Define CSS variables for theme colors and spacing to support easy theming and maintainability.

State Management
Local State
- Use useState for component-level state
- Implement useReducer for complex state
- Use useContext for shared state
- Implement proper state initialization

Global State
- Use Redux Toolkit for global state
- Use createSlice to define state, reducers, and actions together.
- Avoid using createReducer and createAction unless necessary.

- Normalize state structure to avoid deeply nested data.
- Use selectors to encapsulate state access.
- Avoid large, all-encompassing slices; separate concerns by feature.

Error Handling and Validation

Form Validation
- Use Zod for schema validation
- Implement proper error messages
- Use proper form libraries (e.g., React Hook Form)

Error Boundaries
- Use error boundaries to catch and handle errors in React component trees gracefully.
- Log caught errors to an external service (e.g., Sentry) for tracking and debugging.
- Design user-friendly fallback UIs to display when errors occur, keeping users informed without breaking the app.

Testing

Unit Testing
- Write thorough unit tests to validate individual functions and components.
- Use Jest and React Testing Library for reliable and efficient testing of React components.
- Follow patterns like Arrange-Act-Assert to ensure clarity and consistency in tests.
- Mock external dependencies and API calls to isolate unit tests.

Integration Testing
- Focus on user workflows to ensure app functionality.
- Set up and tear down test environments properly to maintain test independence.
- Use snapshot testing selectively to catch unintended UI changes without over-relying on it.
- Leverage testing utilities (e.g., screen in RTL) for cleaner and more readable tests.

Accessibility (a11y)

Core Requirements
- Use semantic HTML for meaningful structure.
- Apply accurate ARIA attributes where needed.
- Ensure full keyboard navigation support.
- Manage focus order and visibility effectively.
- Maintain accessible color contrast ratios.
- Follow a logical heading hierarchy.
- Make all interactive elements accessible.
- Provide clear and accessible error feedback.

Security
- Implement input sanitization to prevent XSS attacks.
- Use DOMPurify for sanitizing HTML content.
- Use proper authentication methods.

Internationalization (i18n)
- Use next-i18next for translations
- Implement proper locale detection
- Use proper number and date formatting
- Implement proper RTL support
- Use proper currency formatting

Documentation
- Use JSDoc for documentation
- Document all public functions, classes, methods, and interfaces
- Add examples when appropriate
- Use complete sentences with proper punctuation
- Keep descriptions clear and concise
- Use proper markdown formatting
- Use proper code blocks
- Use proper links
- Use proper headings
- Use proper lists