API Integration Standards

API Client Structure

- Use Axios as the primary HTTP client
- Create typed API functions in `src/api/` directory
- Implement proper request/response interceptors
- Handle authentication tokens consistently

Error Handling

- Implement comprehensive error handling for all API calls
- Use consistent error response format
- Log errors appropriately for debugging
- Provide meaningful error messages to users

Referenced Templates

@api-client-template.ts

Custom Hooks Standards

Hook Naming and Structure

- Hook names must start with `use` prefix
- Use descriptive names that indicate the hook's purpose
- Place hooks in `src/hooks/` directory
- Export both the hook and its types

Hook Guidelines

- Return objects with named properties for multiple values (not arrays)
- Include proper TypeScript typing for parameters and return values
- Handle loading states and errors consistently
- Use proper cleanup in useEffect hooks
- Document complex hooks with JSDoc comments

Referenced Template

@hook-template.ts

File Structure Standards

Directory Organization

```
src/
├── components/          # Reusable UI components
│   ├── common/         # Generic, reusable components
│   ├── layout/         # Layout-specific components
│   └── ui/             # Basic UI elements
├── pages/              # Top-level page components
├── hooks/              # Custom React hooks
├── context/            # React context providers
├── api/                # API client and service functions
├── utils/              # Utility functions and helpers
├── types/              # TypeScript type definitions
└── assets/             # Static assets
```

File Naming Conventions

- **Components**: PascalCase (`UserProfile.tsx`)
- **Hooks**: camelCase with `use` prefix (`useUserData.ts`)
- **Utilities**: camelCase (`dateHelpers.ts`)
- **Types**: PascalCase (`UserTypes.ts`)
- **Tests**: Same as source with `.test` or `.spec` suffix

React Component Standards

Component Structure

React components should follow this standardized structure:

1. **Imports** - External libraries first, then internal imports
2. **Types/Interfaces** - Component props and local types
3. **Component Definition** - Functional component with proper typing
4. **Export** - Named export (preferred over default)

TypeScript Requirements

- All components must be properly typed
- Use `React.FC<Props>` or explicit return type annotation
- Define props interface above component
- Avoid `any` type - use proper typing or `unknown`

Component Guidelines

- Use functional components with hooks exclusively
- Keep components focused and single-responsibility
- Extract complex logic into custom hooks
- Use descriptive, PascalCase component names
- Include JSDoc comments for complex components

Referenced Template

@component-template.tsx

Testing Standards

Testing Framework

- Use Vitest as the primary testing framework
- Use React Testing Library for component testing
- Use MSW (Mock Service Worker) for API mocking

Test Structure

- Follow AAA pattern: Arrange, Act, Assert
- Use descriptive test names that explain the expected behavior
- Group related tests using `describe` blocks

Component Testing

- Test user interactions, not implementation details
- Query elements by role, label, or text (accessibility-first)
- Mock external dependencies appropriately

Referenced Templates

@component-test-template.tsx @hook-test-template.ts

TypeScript Standards

Type Definitions

- Use `interface` over `type` for object definitions
- Use `type` for unions, primitives, and computed types
- Define types close to where they're used
- Use PascalCase for interface and type names

Strict TypeScript

- Enable strict mode in tsconfig.json
- Avoid `any` type - use `unknown` when type is truly unknown
- Use type assertions sparingly and with type guards
- Prefer type narrowing over type assertions

Function Typing

- Always type function parameters and return values
- Use function overloads when necessary
- Prefer arrow functions for inline functions
- Use proper typing for async functions