Development Guide

Version: 0.1.0
Generated: 7/22/2025

Development Overview

This document provides comprehensive guidelines and best practices for developing the Blog Web Application. It covers setup instructions, architecture, coding standards, and workflows for contributors.

Development Environment Setup

To set up your development environment, follow these steps:

Prerequisites

- **Node.js v16.14.0 or higher (LTS recommended)**
- **npm v8.3.0 or higher (comes with Node.js)**
- **Git v2.30.0 or higher**
- **Visual Studio Code (recommended) with ESLint and Prettier extensions**
- **Chrome or Firefox with React Developer Tools extension**

Installation Steps

- **Clone the repository: git clone https://github.com/organization/blog-web-app.git**
- **Navigate to the project directory: cd blog-web-app**
- **Install dependencies: npm install**
- **Create a .env.local file based on .env.example and configure environment variables**
- **Start the development server: npm run dev**
- **The application will be available at http://localhost:3000**

Environment Variables

Create a .env.local file in the project root with the following variables:

```
NEXT_PUBLIC_API_URL=http://localhost:4000/api
NEXT_PUBLIC_SOCKET_URL=http://localhost:4000
NEXT_PUBLIC_ASSET_URL=http://localhost:4000/assets
NEXT_PUBLIC_GOOGLE_ANALYTICS_ID=UA-XXXXXXXXX-X
NEXT_PUBLIC_GOOGLE_CLIENT_ID=your-google-client-id.apps.googleusercontent.com
NEXT_PUBLIC_ENABLE_MOCK_API=true # Set to false when using real API
```

Project Structure

The project follows a feature-based organization with the following structure:

- src/ - Source code directory
- % % % Animations/ - Animation components and effects
- % % % components/ - React components organized by feature
- % % % % AccountIssue/ - Authentication components
- % % % % Contents/ - Content display components
- % % % % Errors/ - Error handling and display components
- % % % % Header/ - Navigation and header components
- % % % % redux/ - Redux store, slices, and actions
- % % % % ... - Other component categories
- % % % Providers/ - React context providers
- % % % % NotificationProvider.jsx - Notification system provider

- % % % % SearchProvider.js - Search context provider
- % % % Themes/ - Theme definitions and theming context
- % % % util/ - Utility functions and helpers
- % % % % data_structures/ - Custom data structure implementations
- % % % % io_utils/ - API and I/O utilities
- % % % App.js - Main application component
- % % % index.js - Application entry point
- public/ - Static assets and files
- % % % images/ - Image assets
- % % % serviceworkers/ - Service worker scripts
- % % % webworkers/ - Web worker scripts

Feature Organization

Each feature is organized following this structure:

- components/FeatureName/ - Root feature directory
- % % % index.jsx - Main entry point/container component
- % % % FeatureName.module.css - Feature-specific styles (if not using styled components)
- % % % SubComponent/ - Sub-component directory
- % % % % index.jsx - Sub-component implementation
- % % % % SubComponent.test.jsx - Component tests
- % % % hooks/ - Feature-specific custom hooks
- % % % utils/ - Feature-specific utility functions

Development Workflow

Follow these guidelines when developing new features or fixing bugs:

Git Workflow

- **Always branch from develop: git checkout -b feature/your-feature-name**
- **Use conventional commit messages: feat(component): add new feature**
- **Submit pull requests against the develop branch**
- **Ensure CI checks pass before requesting review**
- **Squash commits before merging**

Branch Naming Convention

- **feature/feature-name - For new features**
- **bugfix/issue-description - For bug fixes**
- **hotfix/issue-description - For critical production fixes**
- **refactor/component-name - For code refactoring**
- **docs/documentation-description - For documentation updates**

Commit Message Format

Follow the Conventional Commits specification:

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Types include:

- feat: A new feature
- fix: A bug fix
- docs: Documentation changes
- style: Changes that do not affect the meaning of the code
- refactor: Code change that neither fixes a bug nor adds a feature
- perf: Code change that improves performance

- test: Adding or correcting tests
- build: Changes to the build system or external dependencies
- ci: Changes to CI configuration files and scripts

Coding Standards

Adhere to the following coding standards to maintain code quality:

General Guidelines

- **Use functional components with hooks instead of class components**
- **Use TypeScript for type safety when adding new components**
- **Follow the principle of single responsibility (each component does one thing well)**
- **Keep components small and focused (< 300 lines recommended)**
- **Use named exports instead of default exports for better refactoring support**
- **Avoid prop drilling; use context or state management when props are passed through many layers**
- **Prefer composition over inheritance for component reuse**
- **Use error boundaries to gracefully handle component errors**

React Best Practices

- **Memoize expensive components using React.memo()**
- **Use the useCallback() hook for event handlers passed to child components**
- **Use the useMemo() hook for expensive calculations**
- **Lazy load components for code splitting: const Component = React.lazy(() => import('./Component'))**
- **Add meaningful alt text to all images for accessibility**
- **Use semantic HTML elements (e.g., <button> instead of <div onClick={...}>)**
- **Use proper ARIA attributes for accessibility**
- **Avoid direct DOM manipulation; use refs when necessary**

State Management

- **Use local state (useState) for component-specific state**
- **Use context (useContext) for state shared between a few components**
- **Use Redux for application-wide state or complex state logic**
- **Follow the Redux Toolkit guidelines for reducer organization**
- **Use createSlice() to define reducers and actions**
- **Normalize complex state structures in Redux**
- **Use selectors for accessing and computing derived state**

Testing

The project uses Jest and React Testing Library for testing. Run tests with:

```
npm test              # Run all tests
npm test -- --watch  # Run tests in watch mode
```

Follow these guidelines for writing tests:

Component Testing

- **Test behavior rather than implementation details**
- **Use React Testing Library's queries in this order of preference:**
- **1. Accessible queries (getByRole, getByLabelText, getByPlaceholderText, getByText)**
- **2. Test ID queries (getByTestId) as a last resort**
- **Simulate user interactions using userEvent rather than fireEvent when possible**

- **Test alternative states (loading, error, empty, etc.)**
- **Use mock data consistently across tests**
- **Isolate component tests by mocking external dependencies**

Test Organization
- **Co-locate tests with the components they test**
- **Use descriptive test names following the pattern: "renders/behaves/handles [expected behavior] when [condition]"**
- **Group related tests using describe blocks**
- **Use beforeEach for common setup logic**
- **Keep tests independent of each other**

Example Test
```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import LoginForm from './LoginForm';

describe('LoginForm', () => {
  const mockLogin = jest.fn();

  beforeEach(() => {
    mockLogin.mockClear();
  });

  it('submits username and password when form is submitted', async () => {
    render(<LoginForm onSubmit={mockLogin} />);

    // Fill out form
    await userEvent.type(screen.getByLabelText(/username/i), 'testuser');
    await userEvent.type(screen.getByLabelText(/password/i), 'password123');

    // Submit form
    await userEvent.click(screen.getByRole('button', { name: /log in/i }));

    // Assert
    expect(mockLogin).toHaveBeenCalledWith({
      username: 'testuser',
      password: 'password123',
    });
  });
});
```

Performance Optimization

Follow these guidelines to ensure optimal application performance:

- Use React.lazy() and Suspense for code splitting
- Implement windowing for long lists with react-window or react-virtualized
- Optimize images using WebP format and responsive sizes
- Use pagination or infinite scrolling for large datasets
- Implement proper caching strategies for API calls
- Memoize expensive calculations with useMemo()
- Use service workers for offline support and caching
- Minimize bundle size by tree-shaking and dynamic imports
- Optimize critical rendering path by deferring non-essential resources

Performance Monitoring

Use the following tools to monitor and improve performance:
- **Lighthouse in Chrome DevTools for overall performance audits**
- **React DevTools Profiler for component rendering performance**
- **WebPageTest for real-world performance testing**

- **Bundle analyzer: npm run analyze**

Troubleshooting

Common development issues and solutions:

Node Module Issues
If you encounter dependency issues:

- **Delete node_modules directory: rm -rf node_modules**
- **Clear npm cache: npm cache clean --force**
- **Reinstall dependencies: npm install**

API Connection Issues
If the application cannot connect to the API:

- **Ensure the API server is running**
- **Check that NEXT_PUBLIC_API_URL is correctly set in .env.local**
- **Verify network connectivity and CORS configuration**
- **Check browser console for specific error messages**
- **Set NEXT_PUBLIC_ENABLE_MOCK_API=true to use mock data for development**