사용자 웹 개발 표준 정의서

1. 개요

1.1. 목적

1. 프로젝트의 개발 부문에 적용할 표준을 정의하는 것을 목적으로 합니다.
2. 개발자 간 코드의 일관성 및 유지보수성 확보를 위해 개발 표준을 공유하고자 합니다.
3. 이를 통해 프로젝트의 품질 향상과 생산성 향상을 도모하고자 합니다.
4. 신규 입사자 및 프로젝트 합류자의 온보딩 과정을 용이하게 합니다.
5. 프로젝트의 장기적인 유지보수 및 확장성을 고려한 개발 환경을 구축합니다.
6. 코드 리뷰 프로세스를 효율화하고 일관된 기준을 제공합니다.

1.2. 적용 범위

1. 본 개발 표준은 React.js 기반의 웹 개발 프로젝트 전체에 적용됩니다.
2. 오픈소스 활용 부분은 별도의 기준에 따라 관리되므로 본 표준의 적용 범위에서 제외됩니다.
3. 본 표준은 신규 개발 프로젝트 및 유지보수 단계에 있는 기존 프로젝트에도 점진적으로 적용됩니다.
4. 프로젝트의 특성에 따라 일부 예외 사항이 있을 수 있으며, 이는 프로젝트 책임자의 승인을 받아야 합니다.

2. 개발 환경 및 기술 스택

2.1. 개발 환경

React.js 개발 환경

1. 운영 체제:
    - MacOS: 최신 안정 버전 권장 (최소 macOS Monterey 이상)
    - Windows: Windows 10, 11
2. 개발 툴:
    - IntelliJ IDEA Ultimate (버전 2024.1 이상)
    - Visual Studio Code (최신 안정 버전)
    - 권장 확장 프로그램:
        - ESLint
        - Prettier - Code formatter
        - GitLens
        - Tailwind CSS IntelliSense
        - Import Cost
3. 패키지 매니저:
    - npm (버전 9.0 이상 권장) - 현재 프로젝트는 npm 사용
4. 코드 품질 관리:
    - ESLint (버전 8.57.0)

    - Prettier (버전 3.3.3)
        - **주요 설정:** 세미콜론 미사용 (`semi: false`), 한 줄 최대 80자 (`printWidth: 80`)
    - Husky (버전 9.1.7)
        - **설정:** `pre-push` 훅에서 `npm run build` 실행 (커밋 전 검사 없음)
    - **(주의)** ESLint에서 `no-unused-vars` 규칙 비활성화 상태

5. 버전 관리:
    - Git (버전 2.30 이상)
    - GitHub Enterprise (또는 GitHub)
    - 브랜칭 전략: GitHub Flow (권장)
6. 의존성 관리:
    - npm-check (권장)
    - Dependabot (자동 의존성 업데이트 권장)
7. Node.js: LTS 버전 권장 (최소 v20.0 이상)
8. 개발 서버:
    - Vite (버전 5.3.1)
9. 빌드 도구:
    - Vite (버전 5.3.1)
10. 테스트 도구:
    - **(현재 미설정)** 향후 Jest, React Testing Library, Cypress 도입 고려 (표준 권장 사항)

2.2. 기술 스택

React.js 기술 스택

1. 언어:
    - Javascript (ES2020) - `tsconfig.json` 기준
    - Typescript (버전 5.7.3)
2. 상태 관리:
    - Zustand (버전 5.0.2) - 주 상태 관리 라이브러리
    - Context API + Hooks (간단한 전역 상태 또는 특정 하위 트리 상태)
3. API 통신:
    - Axios (버전 1.7.7)
    - TanStack Query / React Query (버전 5.59.20) - 서버 상태 관리
4. 유틸리티 라이브러리:
    - date-fns (버전 4.1.0) - 날짜 처리

- Day.js (버전 1.11.13) - 날짜 처리 (date-fns와 함께 사용)

5. UI 프레임워크:
    - React (버전 18.3.1)
6. 주요 라이브러리:
    - React Router DOM (버전 6.26.0) - 라우팅
    - MUI (Material-UI, 버전 6.1.1) - UI 컴포넌트 (`@mui/x-date-pickers` 포함)
    - Tailwind CSS (버전 3.4.12) - 유틸리티 CSS (스타일링 주력)
    - Emotion (`@emotion/react`, `@emotion/styled`) - MUI 의존성 및 일부 커스텀 스타일링
    - Firebase (버전 11.3.1) - 인증 (`@firebase/auth`), 메시징 (`@firebase/messaging`)
    - React Daum Postcode (버전 3.2.0) - 주소 검색 UI
    - Swiper (버전 11.1.14) / Slick Carousel (버전 1.8.1) - 캐러셀/슬라이더
    - react-window (버전 1.8.11) - 리스트 가상화
    - Naver Maps API (타입 정의 `@types/navermaps` 사용) - 지도 연동
7. 국제화:
    - **(현재 미적용)** 향후 필요시 react-i18next 등 도입 고려

2.2.1. 라이브러리 선정 기준 및 표준

1. **선정 기준**: 프로젝트에 새로운 라이브러리를 도입할 경우 다음 기준을 고려합니다.
    - **성능**: 애플리케이션 성능에 미치는 영향 최소화
    - **커뮤니티 활성도**: 활발한 커뮤니티 지원 및 지속적인 업데이트 여부
    - **유지보수성**: 문서화 수준, 안정성, 프로젝트 요구사항 부합 여부
    - **라이선스**: 프로젝트 라이선스와 호환되는지 확인
2. **표준 라이브러리**: 섹션 2.2에 명시된 라이브러리들은 본 프로젝트의 표준으로 권장됩니다. 특별한 사유 없이 유사 기능을 가진 다른 라이브러리를 중복 도입하는 것은 지양합니다.

3. 코딩 컨벤션

3.1. 코딩 스타일 가이드

1. 들여쓰기 및 포맷팅
    - 들여쓰기는 공백 2칸을 사용한다. (`.prettierrc.json`: `tabWidth: 2`)
    - 한 줄의 최대 길이는 80자를 넘지 않도록 한다. (`.prettierrc.json`: `printWidth: 80`)
    - 코드 블록의 중괄호({})는 같은 줄에 시작하고 새로운 줄에서 끝낸다.
    - 모든 문장의 끝에는 세미콜론(;)을 **사용하지 않는다**. (`.prettierrc.json`: `semi: false`)
2. 문자열 표현
    - 문자열을 표현할 때는 쌍따옴표(" ")를 사용한다. (`.prettierrc.json`: `singleQuote: false`)
    - 템플릿 리터럴이 필요한 경우에는 백틱(`)을 사용한다.
    - 동적 문자열 연결 시 템플릿 리터럴을 우선 사용한다.
3. 함수 선언

- 함수 선언 시 화살표 함수를 표준으로 작성한다.

```
// 권장
const calculateTotal = (a: number, b: number): number => {
  return a + b
}

// 가능 - 한 줄 화살표 함수의 경우 중괄호와 return 생략 가능
const multiply = (a: number, b: number): number => a * b

// 클래스 메서드 또는 생명주기 메서드의 경우 일반 함수 선언 사용
class Component extends React.Component {
  handleClick() {
    // 처리 로직
  }
}
```

4. 연산자 및 공백

- 연산자 앞뒤로 공백을 넣는다.
- 콤마(,) 뒤에는 공백을 넣는다.
- 코드 블록의 중괄호({}) 안쪽에는 공백을 넣는다.
- 함수 호출 시 괄호와 인자 사이에 공백을 넣지 않는다.

5. JSX 스타일

- JSX element 사이에는 줄바꿈을 하여 가독성을 높인다.
- 속성이 3개 이상인 경우 각 속성을 새 줄에 배치한다.
- 자식 컴포넌트가 있는 경우 들여쓰기하여 계층 구조를 명확히 한다.

```
// 권장
<Component prop1="value1" prop2="value2" prop3="value3">
  <ChildComponent />
</Component>
```

6. 조건문 및 루프

- 복잡한 조건문은 변수로 추출하여 가독성을 높인다.
- 중첩 조건문은 최소화하고, 조기 반환 패턴을 활용한다.
- 반복문보다 배열 메서드(map, filter, reduce 등)를 우선적으로 사용한다.

3.2. 이름 지정 가이드

1. 함수 명명

- 함수 명명은 camelCase를 사용한다.
- 동사 또는 동사구로 시작하며 함수의 목적을 명확히 나타낸다.
- 예시: `getUserData()`, `calculateTotalPrice()`, `handleSubmit()`

2. 메서드 명명

    - 메서드 명명은 camelCase를 사용한다.
    - 이벤트 핸들러는 `handle` 접두사를 사용한다. (예: `handleClick()`)
    - 상태 변경 메서드는 `set` 접두사를 사용한다. (예: `setUserStatus()`)

3. 함수 파라미터

    - 함수 파라미터는 camelCase를 사용한다.
    - 파라미터 이름은 의미를 명확히 전달해야 한다.
    - 불리언 타입 파라미터는 `is`, `has`, `should` 접두사를 사용한다.

4. 변수명

    - 변수명은 camelCase를 사용한다.
    - 명사 또는 명사구로 시작하며 변수의 내용을 명확히 나타낸다.
    - 약어는 가급적 사용하지 않으며, 불가피한 경우 널리 알려진 약어만 사용한다.
    - 단일 문자 변수명(i, j, k 등)은 반복문의 인덱스로만 사용한다.

5. 상수 값

    - 상수 값은 대문자 SNAKE_CASE를 사용한다.
    - 모듈 레벨에서 선언된 상수에만 적용한다.
    - 예시: `MAX_RETRY_COUNT`, `API_BASE_URL`, `DEFAULT_TIMEOUT`

6. 컴포넌트 명명

    - 컴포넌트 명명은 PascalCase를 사용한다.
    - 명사 또는 명사구로 시작한다.
    - 예시: `UserProfile`, `NavigationBar`, `ProductCard`

7. 파일 및 폴더 명명

    - 컴포넌트 파일은 PascalCase를 사용한다. (예: `UserProfile.tsx`)
    - 유틸리티, 훅 등의 파일은 camelCase를 사용한다. (예: `authUtils.ts`, `useFormValidation.ts`)
    - 폴더 이름은 소문자 kebab-case를 사용한다. (예: `user-profile`, `auth-services`)

3.3. 주석 가이드

1. 코드 문서화

    - 필요 시 함수, 클래스, 인터페이스 위에 역할을 설명하는 주석을 달아 문서화한다.
    - 복잡한 알고리즘이나 비즈니스 로직에는 작동 방식을 설명하는 주석을 추가한다.
    - 주석은 '무엇을'이 아닌 '왜'에 초점을 맞춰 작성한다.

2. JSDoc 활용

    - 함수 입력 파라미터에 대한 설명은 JSDoc의 @param 태그를 사용한다.
    - 반환값 설명에는 @returns 태그를 사용한다.
    - 예외 발생 가능성이 있는 경우 @throws 태그를 사용한다.

```
/**
 * 사용자의 총 주문 금액을 계산합니다.
 * @param userId - 사용자 고유 식별자
 * @param includeDiscounts - 할인 적용 여부
 * @returns 총 주문 금액(원화)
 * @throws {ApiError} API 연결 오류 발생 시
 */
const calculateUserOrderTotal = async (
  userId: string,
  includeDiscounts: boolean = true,
): Promise<number> => {
  // 구현 로직
}
```

3. 주석 스타일
    - 한 줄 주석은 //를 사용하고, 주석 앞에 공백을 넣는다.
    - 여러 줄 주석은 /** */를 사용한다.
    - 주석은 문장 형태로 작성하고 마침표로 끝낸다.
4. 특수 주석 태그
    - TODO: 향후 구현이 필요한 기능
    - FIXME: 수정이 필요한 문제점
    - NOTE: 중요한 정보나 설명
    - HACK: 임시방편으로 사용된 코드

```
// TODO: 사용자 권한 검증 로직 추가 필요
// FIXME: Safari에서 렌더링 이슈 발생
// NOTE: 이 로직은 결제 프로세스 변경 시 수정 필요
```

3.4. 기타 가이드

1. 비동기 처리
    - 비동기 처리 시 항상 예외 처리를 통해 오류 핸들링을 한다.
    - Promise 체인보다 async/await 구문을 우선적으로 사용한다.
    - try-catch 블록을 사용하여 오류를 명시적으로 처리한다.

```
const fetchUserData = async (userId: string): Promise<UserData | null>
=> {
  try {
    const response = await api.get(`/users/${userId}`)
    return response.data
  } catch (error) {
    logError("사용자 데이터 조회 실패", error)
    notifyError("사용자 정보를 불러오는데 실패했습니다.")
```

```
      return null
    }
  }
```

2. 타입스크립트 활용

    - any 타입은 가능한 사용하지 않는다 (`strict: true` 설정).
    - 함수 파라미터와 반환값에 명시적 타입을 지정한다.
    - 인터페이스나 타입 별칭을 사용하여 복잡한 객체 구조를 정의한다 (`src/types` 디렉토리 활용).
    - 유니온 타입과 제네릭을 적절히 활용하여 타입 안전성을 높인다.
    - ECMAScript 타겟 버전은 "ES2020"이다.

3. 버전 관리 및 의존성

    - node_modules 및 빌드 결과물들은 .gitignore를 통해 git에서 제외한다.
    - package.json에 의존성 버전을 명확히 명시한다. (`package-lock.json` 활용)
    - 패키지 업데이트 시 충분한 테스트를 거친 후 적용한다.

4. 디버깅 및 로깅

    - 중요 로직에는 적절한 로깅을 추가한다.
    - 개발 환경에서는 console.log를 활용할 수 있으나, 프로덕션 빌드 시에는 제거된다 (`vite.config.ts`의 terser 설정).
    - 로깅 시 중요 정보(개인정보, 인증 정보 등)가 노출되지 않도록 주의한다.
    - 구조화된 로깅을 위해 로깅 라이브러리 사용을 고려한다.

5. 코드 품질 및 테스트

    - 핵심 기능과 복잡한 로직에 대한 단위 테스트 작성을 **권장**한다 (현재 미설정).
    - ESLint와 Prettier를 활용하여 코드 스타일을 일관되게 유지한다.
        - **(주의)** ESLint에서 `no-unused-vars` 규칙이 비활성화되어 있어, 미사용 변수/타입 검출이 안 될 수 있다. `tsconfig.json`의 `noUnusedLocals/Parameters`는 활성화되어 있으나 빌드 시에만 확인될 수 있다.
    - Git Push 전 빌드 검사를 수행한다 (`husky` pre-push hook).
    - 지속적 통합(CI)을 통해 코드 품질 관리를 **권장**한다.

4. 파일 및 디렉토리 구조화 규칙

4.1. 기본 원칙

1. 기능별 구조화

    - 기능 별로 파일 및 폴더를 구분하여 관리한다.
    - 관련된 기능과 컴포넌트는 동일한 디렉토리에 위치시킨다.
    - 확장성을 고려하여 구조를 설계한다.

2. 파일 크기 및 이름 규칙

    - 파일의 이름은 50자를 넘지 않도록 한다.
    - 파일 내 소스코드는 500 line을 넘기지 않는다. (주석 제외)

- 500 line을 초과하는 경우 더 작은 단위로 분리한다.
- 하나의 파일에는 하나의 컴포넌트만 포함한다.

3. 기술적 규칙

- File Encoding은 UTF-8을 기본으로 한다.
- 모든 텍스트 파일은 새 줄(LF)로 끝난다.
- 파일 확장자는 기능에 따라 다음과 같이 사용한다:
  - TypeScript 리액트 컴포넌트: .tsx
  - JavaScript 리액트 컴포넌트: .jsx
  - TypeScript 모듈: .ts
  - JavaScript 모듈: .js
  - 스타일: .css, .module.css (주로 Tailwind 사용)
  - 환경설정: .env, .json, .js, .ts, .cjs

4. 모듈화 원칙

- 재사용 가능한 로직은 별도의 훅(`src/hooks`)이나 유틸리티 함수(`src/utils`)로 분리한다.
- 공통 인터페이스와 타입은 `src/types` 디렉토리에서 관리한다.
- 단일 책임 원칙을 준수하여 각 모듈이 한 가지 역할만 담당하도록 한다.

4.2. React 프로젝트 디렉토리 구조

각 디렉토리의 역할과 세부 구조에 대한 설명입니다. (therapist-frontend 프로젝트 기준)

1. **pages**: 애플리케이션의 각 페이지(라우트)를 담당하는 컴포넌트를 관리

- 각 페이지는 개별 폴더로 구성하여 관련 파일을 함께 관리
- 페이지별 상태 관리(필요시 로컬 Zustand 스토어 또는 Context) 및 데이터 요청 로직(TanStack Query 훅 사용) 포함
- 컴포넌트 조합 역할을 담당하며 실제 UI 구현은 최소화

2. **components**: UI 컴포넌트를 관리

- `components/[페이지명]`: 특정 페이지에서만 사용되는 컴포넌트
- `components/common`: 여러 페이지에서 공통으로 사용되는 컴포넌트
- `components/layout`: 레이아웃 관련 컴포넌트 (헤더, 푸터, 네비게이션 바 등)
- `components/form`: 폼 요소 관련 컴포넌트 (Input, Select 등)
- 각 컴포넌트는 다음 파일들을 포함할 수 있음:
  - `[ComponentName].tsx`: 컴포넌트 구현
  - `[ComponentName].stories.tsx`: (선택) Storybook 파일
  - `[ComponentName].test.tsx`: (선택) 컴포넌트 테스트
  - `index.ts`: 컴포넌트 재내보내기(re-export)

3. **assets**: 정적 자산 관리

- `assets/images`: 이미지 파일 (JPG, PNG, WebP 등)
- `assets/fonts`: 폰트 파일
- `assets/icons`: 아이콘 SVG 파일 (vite-plugin-svgr로 컴포넌트화 가능)

4. **styles**: 전역 스타일 및 Tailwind CSS 관련 설정

    - `styles/global.css`: 전역 CSS 규칙 (Tailwind base, components, utilities import 포함)
    - `tailwind.config.js` (루트): Tailwind 테마 및 플러그인 설정

5. **apis**: API 연동 관련 로직 관리
    - API 클라이언트 설정 및 Axios 인스턴스 (`apis/client.ts`)
    - 기능 또는 도메인별로 API 호출 함수 구성 (`apis/auth.ts`, `apis/users.ts` 등)
    - API 요청 및 응답 타입 정의는 `src/types` 활용
    - 에러 핸들링 및 인터셉터 설정 포함
    - **표준**: 모든 백엔드 API 연동은 `src/apis` 디렉토리 내 함수를 통해 이루어져야 하며, 데이터 페칭 및 캐싱은 `src/queries`의 TanStack Query 훅 사용을 권장한다.
6. **types**: 데이터 모델 및 전역 타입 정의
    - 도메인별 인터페이스 및 타입 정의 (`types/user.ts`, `types/product.ts` 등)
    - API 응답/요청 공통 타입 (`types/api.ts`)
    - 타입 가드 및 타입 유틸리티 함수 포함 가능
7. **contexts**: React Context API 기반 상태 관리
    - 간단한 전역 상태나 특정 하위 트리 상태 관리에 사용
    - 각 컨텍스트마다 별도 파일 또는 폴더로 구성
    - `contexts/[기능명]Context.tsx` 형태로 구성
    - Provider 컴포넌트 및 관련 커스텀 훅 포함
8. **utils**: 공용 유틸리티 함수
    - 날짜, 문자열, 숫자 등 데이터 처리 함수
    - 브라우저 API 래퍼 함수 (LocalStorage 등)
    - 검증(validation) 함수
    - 기타 재사용 가능한 도우미 함수
9. **hooks**: 커스텀 Hook 관리
    - 각 훅은 별도 파일로 관리
    - `hooks/use[기능명].ts` 형태로 구성
    - 재사용 가능한 로직(상태, 이펙트 등) 캡슐화
10. **constants**: 상수 값 관리
    - 라우트 경로 (`constants/routes.ts`)
    - API 엔드포인트 (`constants/apiEndpoints.ts`)
    - 메시지 및 텍스트 상수 (`constants/messages.ts`)
    - 환경 변수 관련 상수
    - 기타 고정 값
11. **stores**: Zustand 기반 전역 상태 관리
    - 애플리케이션의 주요 전역 상태 관리
    - 기능별 또는 도메인별 스토어 파일 구성 (`stores/authStore.ts`, `stores/cartStore.ts` 등)
    - 상태, 액션, 계산된 값(getter) 등을 정의

12. **queries**: TanStack Query 관련 설정 및 훅 관리
    - 쿼리 키 팩토리 (`queries/queryKeys.ts`)
    - 재사용 가능한 커스텀 쿼리/뮤테이션 훅 정의 (`queries/useUserQuery.ts`, `queries/useUpdateUserMutation.ts` 등)
    - `src/apis`의 API 함수와 연동
13. **router**: React Router DOM 설정 및 라우트 정의
    - 애플리케이션 라우트 구조 정의 (`router/index.tsx` 또는 `router/routes.ts`)
    - 라우트 가드(Route Guard) 또는 권한 기반 라우팅 로직 포함 가능
    - 레이아웃 컴포넌트와 페이지 컴포넌트 연결
14. **mappers**: 데이터 형식 변환 로직 관리
    - API 응답 데이터를 UI 모델 또는 내부 데이터 구조로 변환하는 함수
    - 데이터 전송 시 내부 데이터를 API 요청 형식으로 변환하는 함수
    - `mappers/[도메인명]Mapper.ts` 형태로 구성
15. **libs**: 외부 라이브러리 연동 또는 자체 라이브러리성 코드
    - 특정 라이브러리 설정 또는 래퍼(wrapper) 함수
    - 프로젝트 전반에서 사용될 수 있는 독립적인 모듈

4.2.1 React 프로젝트 디렉토리 상세 구성도

```
src/
├── pages/
│   ├── HomePage/
│   │   ├── HomePage.tsx
│   │   └── index.ts
│   ├── LoginPage/
│   │   ├── LoginPage.tsx
│   │   └── index.ts
│   └── ... (다른 페이지)
├── components/
│   ├── home/
│   ├── login/
│   ├── common/ (Button, Modal, Card 등)
│   ├── layout/ (Header, Footer, Navigation 등)
│   └── form/ (Input, Select 등)
├── assets/
│   ├── images/
│   ├── fonts/
│   └── icons/
├── styles/
│   └── global.css
├── apis/
│   ├── client.ts
│   ├── auth.ts
│   ├── users.ts
│   └── ... (다른 API 모듈)
```

```
├── types/
│   ├── user.ts
│   ├── product.ts
│   ├── api.ts
│   └── common.ts
├── contexts/
│   ├── ThemeContext.tsx
│   └── NotificationContext.tsx
├── utils/
│   ├── date.ts
│   ├── string.ts
│   ├── validation.ts
│   └── storage.ts
├── hooks/
│   ├── useAuthChange.ts
│   ├── useDebounce.ts
│   └── ... (다른 커스텀 훅)
├── constants/
│   ├── routes.ts
│   ├── apiEndpoints.ts
│   ├── messages.ts
│   └── queryKeys.ts (또는 queries/ 디렉토리 내부에 정의)
├── stores/
│   ├── authStore.ts
│   └── uiStore.ts
├── queries/
│   ├── queryKeys.ts
│   ├── useUserQuery.ts
│   └── useProductsQuery.ts
├── router/
│   ├── index.tsx
│   └── protectedRoute.tsx
├── mappers/
│   ├── userMapper.ts
│   └── productMapper.ts
├── libs/
│   └── firebase.ts (Firebase 초기 설정 등)
├── App.tsx
└── main.tsx (애플리케이션 진입점)
```

4.8. 주요 기능 구현 표준

1. **아키텍처 패턴 준수**: 섹션 `4.1.5 아키텍처 패턴`에서 정의된 원칙을 따릅니다.
2. **API 연동**: `src/apis`에 정의된 함수와 `src/queries`의 TanStack Query 훅을 사용하여 서버와 통신합니다. API 응답 데이터는 필요 시 `src/mappers`를 통해 UI 모델로 변환하여 사용합니다.
3. **상태 관리**: 섹션 `4.7 상태 관리`의 가이드라인(로컬 상태, 전역 상태 - Zustand/Context, 서버 상태 - TanStack Query)에 따라 적절한 상태 관리 도구를 선택하고 표준 패턴을 따릅니다.
4. **재사용성**: 공통 로직은 `src/hooks` 또는 `src/utils`로 분리하고, 공통 UI 요소는 `src/components/common`에 작성하여 재사용성을 높입니다.

5. 프로젝트 품질 관리 및 배포

5.1. 코드 품질 관리

1. 코드 리뷰
    - 모든 코드는 최소 1명 이상의 리뷰어에게 리뷰를 받아야 한다.
    - **프로세스**: Pull Request(PR) 기반 코드 리뷰를 원칙으로 합니다.
    - **체크리스트 활용**: 일관된 품질을 위해 다음 항목을 포함한 리뷰 체크리스트 활용을 권장합니다.
        - 코딩 컨벤션 준수 여부 (본 문서 기준)
        - 요구사항 및 설계 부합 여부
        - 가독성 및 유지보수성
        - 성능 및 최적화 고려 여부
        - 보안 취약점 유무 (입력값 검증, 권한 체크 등)
        - 테스트 코드 작성 여부 및 커버리지 (테스트 도입 시)
        - 중복 코드 및 불필요한 로직 제거 여부
        - 타입 정의의 적절성 (`any` 타입 최소화)
2. 자동화된 코드 품질 도구
    - ESLint: 코드 품질 및 스타일 검사 (일부 규칙 비활성화 상태 확인 필요)
    - Prettier: 코드 포맷팅
    - TypeScript: 타입 체크 (`strict: true`)
    - Husky: Git `pre-push` 훅을 활용한 빌드 검사 (커밋 전 검사 없음)
    - **(미적용)** `lint-staged`: 표준 문서에서 권장되었으나 현재 미사용
3. 테스트 전략
    - **표준**: 핵심 비즈니스 로직, 복잡한 컴포넌트, 공용 유틸리티 및 훅에 대한 테스트 코드 작성을 표준으로 권장합니다.
    - **(현재 미설정)** 단위 테스트, 통합 테스트, E2E 테스트 도입을 **강력히 권장**한다.
    - **도구**: 도입 시 Jest + React Testing Library (단위/통합), Cypress (E2E) 고려
    - **커버리지 목표**: 도입 시 기능 안정성 확보를 위해 테스트 커버리지 목표(예: 주요 로직 기준 70% 이상) 설정을 권장합니다.
4. 타입 관리
    - **원칙**: TypeScript의 타입 시스템을 최대한 활용하여 코드 안정성을 높입니다.
    - `any` 타입 사용은 불가피한 경우를 제외하고 **엄격히 제한**합니다. `unknown` 타입을 우선 사용하고 타입 가드를 통해 타입을 좁혀나갑니다.
    - 모든 함수 파라미터, 반환값, 변수에는 명시적인 타입을 정의하는 것을 원칙으로 합니다. (`strict: true` 설정 유지)
    - 복잡한 타입이나 여러 곳에서 사용되는 타입은 `src/types` 디렉토리에서 인터페이스 또는 타입 별칭으로 정의하여 재사용합니다.

5.2. 배포 프로세스

1. 개발 환경 구분
    - 개발(Development): 개발자 로컬 환경 (Vite 개발 서버)
    - 테스트(Test): QA 및 테스트용 환경 (별도 배포 환경 구성 필요)
    - 스테이징(Staging): 프로덕션과 유사한 환경으로 최종 테스트 (별도 배포 환경 구성 필요)

- 프로덕션(Production): 실 사용자 대상 환경 (Vercel 사용 중 - `vercel.json` 확인)
- **원칙**: 각 환경은 독립적으로 구성되며, 환경별 설정(`.env` 파일 등)을 통해 관리합니다. (상세 내용은 `6.3 환경 변수 및 설정 관리 표준` 참조)

2. CI/CD 파이프라인

- **표준**: 코드 변경 사항을 통합하고 테스트하며, 각 환경(Test, Staging, Production)으로 자동으로 배포하는 CI/CD 파이프라인 구축 및 활용을 표준으로 합니다.
- **구성 가이드라인**:
  - **빌드 자동화 원칙**: 소스 코드 변경 시 자동으로 빌드 프로세스가 실행되어야 합니다. (`npm run build`)
  - 테스트 자동화: 빌드 성공 후 자동화된 테스트(단위, 통합 등)를 실행합니다. (테스트 도입 시)
  - 배포 자동화: 테스트 통과 후 지정된 환경으로 자동 배포를 수행합니다. (예: Main 브랜치 Push 시 Production 배포)
  - **배포 프로세스 표준화**: 각 환경별 배포 절차를 명확히 정의하고 자동화하여 휴먼 에러를 최소화합니다.
- **(설정 확인 필요)** GitHub Actions (`.github/` 디렉토리 확인) 또는 Vercel CI/CD 활용 가능성 높음
- 환경별 설정 관리 (`.env.local`, `.env.production` 활용)

3. 버전 관리 및 릴리스

- Semantic Versioning(SemVer) 적용 권장: Major.Minor.Patch
- 릴리스 노트 작성 및 관리 권장
- 핫픽스 절차 정의 권장

4. 모니터링 및 에러 추적

- 에러 로깅 및 모니터링 도구 활용 권장 (Sentry, LogRocket 등)
- 성능 모니터링 (Google Analytics, Vercel Analytics 등)
- 사용자 피드백 수집 체계 구축 권장

6. 보안 가이드라인

6.1. 인증 및 권한 관리

1. 인증(Authentication) - 사용자 인증 구현 표준

- **표준 방식**: JWT(JSON Web Token) 또는 OAuth 2.0 기반의 인증 시스템 활용을 표준으로 합니다. (현재 Firebase Authentication 사용 중)
- **토큰 관리 표준**:
  - 토큰 만료 시간 설정 및 자동 갱신(Refresh Token) 전략을 구현합니다.
  - 토큰은 안전한 방식으로 저장 및 관리합니다. (예: HttpOnly 쿠키, 브라우저 Secure Storage 고려) 클라이언트 사이드 스크립트에서 직접 접근 가능한 저장소(LocalStorage, SessionStorage)는 탈취 위험이 있어 민감한 토큰 저장에 권장되지 않습니다.
- 비밀번호 등 민감 정보는 클라이언트에서 직접 처리하지 않고 서버에서 안전하게 처리합니다.

2. 권한(Authorization) - 권한 관리 아키텍처 가이드라인

- **표준 방식**: 역할 기반 접근 제어(RBAC) 또는 속성 기반 접근 제어(ABAC) 적용을 표준으로 합니다. 사용자 역할(Role)에 따라 접근 가능한 리소스와 기능을 제어합니다.
- **구현**:

- **라우팅 레벨**: React Router 등을 활용하여 특정 경로 접근 시 필요한 권한을 체크하는 라우트 가드(Route Guard)를 구현합니다.
- **컴포넌트 레벨**: 사용자 권한에 따라 특정 UI 요소(버튼, 메뉴 등)를 조건부로 렌더링하거나 비활성화합니다.
- API 요청 시 백엔드에서 최종적인 권한 검증이 이루어져야 합니다.

6.2. 데이터 보안

1. 입력 데이터 검증
    - 모든 사용자 입력은 클라이언트와 서버 양쪽에서 검증
    - XSS 및 인젝션 공격 방지 대책 수립
    - 보안 라이브러리 활용 (DOMPurify, validator.js 등)
2. API 보안
    - API 요청에 CSRF 토큰 적용
    - 민감한 데이터는 암호화하여 전송
    - HTTPS 사용 강제
    - API 속도 제한 및 모니터링
3. 환경 변수 관리
    - 비밀 키와 API 키는 환경 변수로 관리
    - .env 파일은 버전 관리에서 제외
    - 프로덕션 빌드 시 환경 변수 주입 방식 사용

6.3. 환경 변수 및 설정 관리 표준

1. **관리 원칙**:
    - API 키, 비밀 키 등 민감 정보는 소스 코드에 직접 포함하지 않고 환경 변수를 통해 관리합니다.
    - 환경 변수 파일(`.env`, `.env.local`, `.env.production` 등)을 사용하여 환경별 설정을 분리합니다.
2. **민감 정보 처리 가이드라인**:
    - `.env` 파일 자체는 샘플 구성이나 기본값만 포함하고, 실제 민감 정보가 담긴 `.env.local` 등은 `.gitignore`에 추가하여 버전 관리에서 제외합니다.
    - 프로덕션 빌드 시 환경 변수 주입 방식 사용
    - 클라이언트로 노출되는 환경 변수에는 민감 정보를 포함하지 않도록 주의합니다. (Vite는 `VITE_` 접두사가 붙은 변수만 클라이언트에 노출)
3. **환경 분리 원칙**:
    - 개발(Development), 테스트(Test), 스테이징(Staging), 운영(Production) 환경별로 필요한 설정(API 엔드포인트, 기능 플래그 등)을 명확히 분리하여 관리합니다.
    - 각 환경의 목적에 맞게 설정을 최적화합니다. (예: 운영 환경에서는 디버깅 로그 비활성화)

7. 성능 최적화 가이드라인

7.1. 렌더링 최적화

1. 컴포넌트 메모이제이션
    - React.memo, useMemo, useCallback 적절히 활용

- 큰 리스트는 가상화 기법 적용 (`react-window` 사용 중)
- 불필요한 리렌더링 방지 (React DevTools Profiler 활용)

2. 코드 분할

- React.lazy와 Suspense를 활용한 동적 import
- Vite의 `build.rollupOptions.output.manualChunks` 설정을 활용한 청크 분할 (현재 설정됨)
- 라우트 기반 코드 분할 구현 (React Router와 연계)

3. 이미지 및 에셋 최적화

- 이미지 최적화 및 압축 (`vite-plugin-image-optimizer` 사용 중)
- WebP 또는 AVIF 포맷 사용 권장 (현재 Vite 설정에 WebP 최적화 포함)
- 적절한 이미지 크기 및 해상도 사용 (반응형 이미지 처리 고려)

7.2. 네트워크 최적화

1. 데이터 요청

- TanStack Query를 활용한 데이터 캐싱 전략 적극 활용
- 불필요한 API 요청 최소화 (캐시 활용, 요청 병합 등)
- API 설계 시 필요한 데이터만 요청하도록 백엔드와 협의

2. 번들 최적화

- Vite 빌드 결과 분석 및 번들 크기 최적화 (Vite Analyzer 플러그인 활용 고려)
- 트리 쉐이킹(Tree Shaking)은 Vite에서 기본 지원
- 의존성 최소화 및 불필요한 패키지 제거 (정기적 검토)
- Vite의 압축 플러그인(`vite-plugin-compression`) 활용한 Gzip, Brotli 압축 (현재 설정됨)

8. 접근성 및 국제화

8.1. 웹 접근성

1. 접근성 표준

- WCAG 2.1 AA 수준 준수
- 시맨틱 HTML 요소 사용
- 키보드 네비게이션 지원

2. 접근성 테스트

- 스크린 리더 호환성 테스트
- 접근성 자동화 도구 활용 (axe, lighthouse)
- 수동 접근성 테스트 수행

8.2. 국제화(i18n) 및 지역화(l10n)

1. 다국어 지원

- react-i18next 또는 react-intl 라이브러리 활용
- 번역 키 관리 체계 구축

- 동적 언어 전환 지원

2. 지역화 고려사항
    - 날짜, 시간, 숫자, 통화 포맷
    - RTL(Right-to-Left) 언어 지원
    - 문화적 차이 고려

8.3. 웹 플랫폼 기능 연동 (Web API 활용)

1. **푸시 알림 구현 표준**:
    - 웹 푸시 알림 기능 구현 시, Firebase Cloud Messaging(FCM) (`@firebase/messaging`) 또는 웹 표준 Push API 활용을 표준으로 합니다.
    - 사용자 동의(Permission) 요청 프로세스를 명확히 구현하고, 알림 설정 관리 기능을 제공합니다.
2. **네이티브 기능 연동 표준**:
    - 카메라, 마이크, 위치 정보 등 브라우저가 제공하는 네이티브 기능 연동 시, 관련 Web API (예: MediaDevices API, Geolocation API) 사용을 표준으로 합니다.
    - 기능 사용 전 반드시 사용자에게 명시적인 권한 요청 및 동의를 받아야 합니다.
    - API 사용 가능 여부(feature detection) 및 오류 처리를 철저히 구현합니다.
    - 민감 정보(위치 등) 접근 시 사용자 프라이버시 보호를 최우선으로 고려합니다.

9. 문서화 및 지식 관리

9.1. 코드 문서화

1. 인라인 문서화
    - JSDoc 주석 활용
    - 복잡한 알고리즘 및 비즈니스 로직 설명
    - API 인터페이스 문서화

2. 컴포넌트 문서화
    - Storybook을 활용한 컴포넌트 문서화
    - 컴포넌트 사용 예시 및 프로퍼티 설명
    - 디자인 시스템과 연계

9.2. 프로젝트 문서화

1. README 및 개발 가이드
    - 설치 및 설정 방법
    - 개발 환경 구성 안내
    - 주요 명령어 및 스크립트 설명

2. API 문서화
    - API 엔드포인트 설명
    - 요청 및 응답 예시
    - 오류 코드 및 처리 방법

3. 아키텍처 문서
    - 시스템 구조 다이어그램
    - 주요 컴포넌트 및 모듈 설명
    - 데이터 흐름 도식화