4-1. 프론트 시작 - Part 1: 개발 시작 전 반드시 정해야 할 것들

들어가며

디자이너가 Figma 링크를 공유했다.
예쁜 디자인이 완성되어 있다.

"이제 코딩하면 되겠네!"

잠깐. 바로 코딩하면 안 된다.

프론트엔드 개발자들이 자주 하는 실수가 있다. 디자인만 있으면 바로 개발을 시작할 수 있다고 생각하는 것이다. 하지만 2주 후에 이런 대화가 오간다.

"이 버튼 컴포넌트 어디 있어요?"
"저는 BlueButton이라고 만들었는데요."
"저는 PrimaryButton으로 만들었어요."
"저는 그냥 Button으로 만들었고요."

같은 버튼인데 3개의 컴포넌트가 존재한다. 디자이너가 "버튼 색상을 바꿔주세요"라고 하면? 3곳을 모두 수정해야 한다.

이 가이드는 총 3부작으로 구성되어 있다.

- **Part 1: 개발 시작 전 반드시 정해야 할 것들** (이 문서)
- **Part 2: 개발하면서 정해도 되는 것들**
- **Part 3: 개발 시작**

Part 1에서는 코드를 한 줄도 작성하기 전에 팀 전체가 합의해야 하는 것들을 다룬다. 이것들을 정하지 않고 개발을 시작하면, 나중에 엄청난 시간 낭비와 혼란을 겪게 된다.

---

1. 기술 명세서 작성

1.1 왜 가장 먼저 해야 하나?

"React 쓰면 되지 뭐"라고 생각하면 안 된다. React만 해도 선택지가 많다.

- Create React App vs Vite vs Next.js
- 상태 관리는? Context API vs Zustand vs Redux
- 스타일링은? Tailwind vs Styled-components vs CSS Modules
- 라우팅은? React Router vs TanStack Router

팀원마다 다른 도구를 쓰면 코드가 일관성 없이 뒤죽박죽이 된다.

1.2 무엇을 정해야 하나?

1) 프레임워크/라이브러리 선택

React vs Vue vs Svelte

```
선택: React 18
이유:
- 팀원 4명 중 3명이 React 경험 보유
```

- 생태계가 가장 크고 자료가 풍부
- 채용 공고에서 가장 많이 요구

고려했지만 선택하지 않은 것:
- Vue: 러닝 커브 낮지만 팀 경험 부족
- Svelte: 최신 기술이지만 생태계 작음

왜 React 버전까지 명시하나?

React 18은 Concurrent Features가 추가되었다. 이전 버전과 동작이 다를 수 있다. 명확히 버전을 명시하면 나중에 "왜 안 되지?"를 줄일 수 있다.

2) 빌드 도구 선택

Create React App vs Vite vs Next.js

선택: Vite
이유:
- 빌드 속도가 CRA보다 10배 이상 빠름
- HMR(Hot Module Replacement) 속도 빠름
- 간단한 설정

왜 CRA가 아닌가?
- 2023년부터 React 공식 문서에서 권장하지 않음
- 빌드 속도 느림
- 설정 변경 어려움

왜 Next.js가 아닌가?
- SSR 불필요 (SEO 우선순위 낮음)
- 학습 곡선 높음
- MVP 단계에는 과도함

3) 상태 관리 선택

Context API vs Zustand vs Redux

선택: Zustand
이유:
- Redux보다 간단함 (보일러플레이트 적음)
- Context API보다 성능 좋음
- 학습 곡선 낮음

단계별 전략:
- Phase 1: useState + Context API
- Phase 2: 복잡해지면 Zustand 도입
- Phase 3: 정말 필요하면 Redux 고려

왜 처음부터 Redux를 안 쓰나?

Redux는 강력하지만 복잡하다. action, reducer, store를 모두 만들어야 한다. MVP 단계에서는 과도하다. 상태 관리가 정말 복잡해졌을 때 도입해도 늦지 않다.

4) 스타일링 방식 선택

Tailwind vs Styled-components vs CSS Modules

```
선택: Tailwind CSS
이유:
- 빠른 개발 속도 (클래스만 추가)
- 일관된 디자인 시스템
- 파일 왕복 불필요 (HTML에 바로 작성)

고려사항:
- 처음엔 클래스명이 길어 보임
- 하지만 익숙해지면 가장 빠름
```

왜 Styled-components가 아닌가?

Styled-components는 CSS-in-JS다. 런타임에 스타일이 생성되어 성능에 영향을 줄 수 있다. Tailwind는 빌드 타임에 필요한 CSS만 생성되어 더 빠르다.

5) HTTP 클라이언트 선택

axios vs fetch vs React Query

```
선택: axios + TanStack Query (React Query)
이유:
- axios: 인터셉터로 토큰 자동 추가 가능
- React Query: 캐싱, 리페칭 자동 처리

왜 fetch가 아닌가?
- 기본 기능만 제공
- 인터셉터 직접 구현해야 함
- 타임아웃 설정 복잡
```

1.3 기술 명세서 템플릿

```
# 개그 배틀 프론트엔드 기술 명세서

## 기술 스택

### 코어
- React 18.2
- TypeScript 5.0
- Vite 5.0

### 상태 관리
- Zustand 4.4
```

```
### 스타일링
- Tailwind CSS 3.4

### HTTP 클라이언트
- axios 1.6
- TanStack Query 5.0

### 라우팅
- React Router 6.20

### 폼 관리
- React Hook Form 7.48

### 실시간 통신
- Socket.io-client 4.6

### 코드 품질
- ESLint 8.55
- Prettier 3.1

### 배포
- Vercel (자동 배포)

## 브라우저 지원
- Chrome 최신 2버전
- Safari 최신 2버전
- Firefox 최신 2버전
- Edge 최신 2버전
- 모바일: iOS Safari 15+, Chrome Android 최신

## 성능 목표
- Lighthouse 점수 90+ (Performance)
- First Contentful Paint: 1.5초 이내
- Time to Interactive: 3초 이내
```

---

2. 디렉토리 구조 선정

2.1 왜 미리 정해야 하나?

디렉토리 구조는 나중에 바꾸기 매우 어렵다. 파일이 10개일 때는 쉽다. 하지만 100개가 되면? import 경로가 전부 깨진다.

2.2 선택지와 의사결정

Feature-based vs Layer-based

Layer-based 구조:

```
components/
  - Button.tsx
  - Input.tsx
  - UserCard.tsx
  - MatchCard.tsx
hooks/
pages/
utils/
```

문제점: User 기능을 수정하려면 여러 폴더를 오가야 한다.

Feature-based 구조:

```
features/
  user/
    - UserCard.tsx
    - useUser.ts
    - userApi.ts
  match/
    - MatchCard.tsx
    - useMatch.ts
```

장점: 관련된 코드가 한 곳에 모여있다.

MVP 단계 권장: Feature-based (하이브리드)

```
src/
├── features/        # 기능별 모듈
│   ├── auth/
│   ├── match/
│   └── user/
├── shared/          # 공통 코드
│   ├── components/  # 공용 컴포넌트
│   ├── hooks/
│   ├── utils/
│   └── api/
├── pages/           # 라우트 페이지
├── stores/          # 전역 상태
└── styles/          # 글로벌 스타일
```

2.3 왜 이 구조인가?

features 폴더

User 기능을 수정할 때 `features/user`만 보면 된다. 관련 컴포넌트, 훅, API가 모두 여기 있다.

shared 폴더

Button, Input 같은 공용 컴포넌트는 shared에 둔다. 여러 feature에서 사용되기 때문이다.

pages 폴더

Next.js처럼 파일 기반 라우팅은 아니지만, 페이지를 한눈에 보기 위해 따로 둔다.

---

3. 공용 컴포넌트 설계 전략

3.1 왜 가장 먼저 정해야 하나?

실제 사례

A 개발자: "버튼 필요해서 만들었어요"
B 개발자: "저도 버튼 필요해서 만들었어요"
C 개발자: "저도요"

2주 후 코드 리뷰:
"어? 버튼 컴포넌트가 왜 7개나 있어요?"

디자이너: "버튼 색상 바꿔주세요"
개발자들: "...7곳을 다 바꿔야 하나요?"

공용 컴포넌트를 미리 정하지 않으면 이런 일이 생긴다.

3.2 Atomic Design으로 접근

atoms (원자) - 더 이상 쪼갤 수 없는 것

- Button, Input, Label, Icon, Badge, Avatar

molecules (분자) - atoms의 조합

- SearchBar (Input + Button)
- FormField (Label + Input + ErrorText)
- UserProfile (Avatar + Name + Badge)

organisms (유기체) - 복잡한 UI

- Header, Footer, Sidebar
- MatchCard (여러 molecule의 조합)

templates (템플릿) - 레이아웃

- PageLayout, AuthLayout, DashboardLayout

pages (페이지) - 실제 페이지

- HomePage, MatchPage, ProfilePage

3.3 왜 이렇게 나누나?

Button은 왜 atom인가?

Button은 더 이상 쪼갤 수 없다. 어디서든 재사용되어야 한다. 로그인 버튼, 매칭 버튼, 취소 버튼 모두 같은 Button 컴포넌트를 쓴다.

SearchBar는 왜 molecule인가?

SearchBar는 Input + Button의 조합이다. 하지만 이 둘은 항상 함께 쓰인다. 매번 Input과 Button을 따로 import하는 건 번거롭다. 하나로 묶는다.

Header는 왜 organism인가?

Header는 로그인 여부에 따라 다른 UI를 보여준다. 비즈니스 로직이 들어간다. molecule보다 복잡하다.

3.4 공용 컴포넌트 설계 원칙

1. Props는 최소한으로

나쁜 예:

```
<Button
  text="클릭"
  color="blue"
  size="medium"
  rounded={true}
  shadow={true}
/>
```

좋은 예:

```
<Button variant="primary" size="md">
  클릭
</Button>
```

variant 하나로 color, shadow, hover 효과를 모두 포함한다.

2. children으로 유연성 확보

제한적:

```
<Button text="로그인" />
```

유연함:

```
<Button>
  <Icon name="login" />
  <span>로그인</span>
</Button>
```

3. Compound Component 패턴

복잡한 컴포넌트는 쪼갠다:

```
<Modal>
  <Modal.Header>제목</Modal.Header>
  <Modal.Body>내용</Modal.Body>
  <Modal.Footer>
    <Button>확인</Button>
  </Modal.Footer>
</Modal>
```

3.5 3번 규칙 (Rule of Three)

언제 공용 컴포넌트로 만드나?

1번째: 그냥 만든다
2번째: 복붙한다 (아직 패턴이 명확하지 않음)
3번째: "아, 이게 패턴이구나" → 공용 컴포넌트로 추출

처음부터 공용 컴포넌트로 만들면 안 되나? 안 된다. 요구사항이 명확하지 않은 상태에서 공용 컴포넌트를 만들면, 나중에 수정할 때 모든 곳이 영향을 받는다.

3.6 디자이너와 용어 통일

실제 사례

디자이너: "Primary Button 수정 부탁드려요"
개발자: "Primary Button이 뭐죠?"
디자이너: "이 파란색 버튼이요"
개발자: "아... 저희는 BlueButton이라고 부르는데..."

해결책

Figma 컴포넌트 이름 = React 컴포넌트 이름

```
Figma: Button/Primary
React: <Button variant="primary" />

Figma: Input/TextField
React: <Input type="text" />

Figma: Card/Match
React: <MatchCard />
```

3.7 공용 컴포넌트 문서화

README에 간단히 적는다:

```
# Button 컴포넌트

## 사용법
<Button variant="primary" size="md">클릭</Button>

## Props
- variant: "primary" | "secondary" | "danger"
- size: "sm" | "md" | "lg"
- disabled: boolean

## 예시
<Button variant="primary">로그인</Button>
<Button variant="secondary" size="sm">취소</Button>
```

Storybook은 MVP 단계에서는 과도하다. 나중에 필요하면 추가한다.

---

4. 코드 컨벤션 정하기

4.1 왜 필요한가?

A 개발자: `getUserData()`
B 개발자: `fetchUser()`
C 개발자: `loadUserInfo()`

같은 기능인데 이름이 다 다르다. 3개월 후 신입 개발자: "저는 뭘 써야 하나요?"

4.2 네이밍 규칙

컴포넌트

- PascalCase 사용
- 의미 명확히: `Button`, `UserCard`, `MatchList`

함수

- camelCase 사용
- 동사로 시작: `handleClick`, `fetchUser`, `validateEmail`

변수

- camelCase 사용
- boolean은 `is`, `has`, `should`로 시작: `isLoading`, `hasError`

상수

- UPPER_SNAKE_CASE
- `API_BASE_URL`, `MAX_RETRY_COUNT`

파일명

- 컴포넌트: PascalCase → `UserCard.tsx`
- 훅: camelCase → `useAuth.ts`
- 유틸: camelCase → `formatDate.ts`

4.3 폴더 및 파일 구조 규칙

컴포넌트 폴더

```
UserCard/
  - index.tsx       # 컴포넌트
  - UserCard.test.tsx
  - UserCard.stories.tsx  # (선택)
  - types.ts        # 타입 정의
```

왜 index.tsx를 쓰나?

import 경로가 깔끔해진다:

```
import UserCard from '@/components/UserCard'
// vs
import UserCard from '@/components/UserCard/UserCard'
```

4.4 Import 순서

```
// 1. React 관련
import React, { useState } from 'react'

// 2. 외부 라이브러리
import axios from 'axios'
import { useQuery } from '@tanstack/react-query'

// 3. 내부 절대 경로
import { Button } from '@/shared/components'
import { useAuth } from '@/features/auth'

// 4. 상대 경로
import { UserCard } from './UserCard'
import type { User } from './types'

// 5. 스타일
import styles from './styles.module.css'
```

4.5 주석 작성 규칙

언제 쓰나?

- 복잡한 비즈니스 로직
- 외부 API 스펙 참조
- 임시 해결책 (TODO와 함께)

언제 안 쓰나?

- 코드로 명확한 경우
- 이름이 의미를 잘 전달하는 경우

나쁜 예:

```
// 사용자 이름 표시
<span>{user.name}</span>
```

좋은 예:

```
// iOS Safari 버그 회피: WebRTC 연결 전 100ms 대기 필요
// https://bugs.webkit.org/show_bug.cgi?id=179363
await delay(100)
```

---

5. 디자인 시스템 합의

5.1 왜 필요한가?

디자이너: "이 버튼은 primary-blue예요"
개발자: "#0066FF를 쓰면 되나요?"
디자이너: "아니요, #0052CC예요"
개발자: "그럼 이 버튼은요?"
디자이너: "그건 #0047B3이에요"

색상 코드를 하드코딩하면 나중에 "브랜드 색상이 바뀌었어요"라고 할 때 100곳을 수정해야 한다.

5.2 색상 팔레트

Tailwind config에 정의:

```
// tailwind.config.js
module.exports = {
  theme: {
    extend: {
      colors: {
        primary: {
          50: '#EFF6FF',
          500: '#0066FF',
          600: '#0052CC',
          700: '#0047B3',
        },
        gray: {
          50: '#F9FAFB',
          500: '#6B7280',
          900: '#111827',
        }
      }
    }
  }
}
```

이제 `className="bg-primary-500"`로 사용한다.

5.3 타이포그래피

```
fontSize: {
  'heading-1': ['32px', { lineHeight: '40px', fontWeight: '700' }],
  'heading-2': ['24px', { lineHeight: '32px', fontWeight: '600' }],
  'body': ['16px', { lineHeight: '24px', fontWeight: '400' }],
  'caption': ['14px', { lineHeight: '20px', fontWeight: '400' }],
}
```

5.4 반응형 브레이크포인트

```
screens: {
  'sm': '640px',   // 모바일
  'md': '768px',   // 태블릿
```

```
  'lg': '1024px',  // 데스크톱
  'xl': '1280px',  // 큰 데스크톱
}
```

디자이너와 합의한 기준을 명확히 정한다.

---

6. API 통신 규칙

6.1 API 호출은 어디서?

선택지

1. 컴포넌트에서 직접
2. Custom Hook
3. Service 레이어

권장: Custom Hook + Service 레이어

```
// features/user/api/userApi.ts
export const userApi = {
  getUser: (id: string) =>
    axios.get(`/api/v1/users/${id}`),

  createUser: (data: UserCreateRequest) =>
    axios.post('/api/v1/users', data)
}

// features/user/hooks/useUser.ts
export const useUser = (id: string) => {
  return useQuery({
    queryKey: ['user', id],
    queryFn: () => userApi.getUser(id)
  })
}

// 컴포넌트
const { data: user } = useUser(userId)
```

6.2 에러 처리 전략

axios 인터셉터로 공통 처리

```
axios.interceptors.response.use(
  response => response,
  error => {
    if (error.response?.status === 401) {
      // 로그아웃 처리
    }
    if (error.response?.status === 500) {
```

```
      // 에러 토스트
    }
    return Promise.reject(error)
  }
)
```

6.3 로딩 상태 관리

React Query가 자동으로 처리:

```
const { data, isLoading, error } = useUser(id)

if (isLoading) return <Skeleton />
if (error) return <ErrorMessage />
return <UserCard user={data} />
```

---

7. 환경 분리 전략

7.1 .env 파일 관리

```
.env.local          # 로컬 개발
.env.development    # 개발 서버
.env.staging        # 스테이징
.env.production     # 운영
```

```
VITE_API_BASE_URL=http://localhost:8080
VITE_WS_URL=ws://localhost:8080
```

Vite는 `VITE_` 접두사가 필수다.

7.2 환경별 설정

```
const config = {
  apiBaseUrl: import.meta.env.VITE_API_BASE_URL,
  wsUrl: import.meta.env.VITE_WS_URL,
  isDev: import.meta.env.DEV,
  isProd: import.meta.env.PROD,
}
```

---

8. Git 전략

8.1 브랜치 전략

```
main
  ├── feature/user-profile
  ├── feature/match-system
```

```
├── fix/button-click
└── hotfix/login-error
```

8.2 커밋 메시지

```
feat(user): 사용자 프로필 페이지 구현
fix(match): 매칭 버튼 클릭 안 되는 버그 수정
style(button): 버튼 padding 조정
```

Part 1 체크리스트

- [ ] 기술 명세서 작성 완료
- [ ] 디렉토리 구조 합의
- [ ] 공용 컴포넌트 목록 작성
- [ ] Atomic Design 레벨 정의
- [ ] 코드 컨벤션 문서화
- [ ] 디자인 시스템 (색상, 폰트) 합의
- [ ] API 통신 규칙 정립
- [ ] 환경 분리 전략 수립
- [ ] Git 브랜치 전략 합의

마치며

"계획에 1시간을 쓰면, 실행에 10시간을 절약할 수 있다."

지금 30분 투자해서 공용 컴포넌트를 정하면, 나중에 "버튼 컴포넌트가 7개나 있어요"를 막을 수 있다.

지금 1시간 투자해서 디자인 시스템을 합의하면, 나중에 "이 색상 코드가 뭐였더라?"를 100번 검색하지 않아도 된다.

준비는 지루하지만, 그 지루함을 견디는 팀이 결국 더 빠르게 달린다.

Part 2에서는 성능 최적화, 테스트 전략처럼 개발하면서 정해도 되는 것들을 다룬다. 🚀