4-2 프론트엔드 시작 - Part 2: 개발하면서 정해도 되는 것들

들어가며

Part 1에서는 개발 시작 전에 반드시 정해야 할 것들을 다뤘다. 기술 스택, 디렉토리 구조, 공용 컴포넌트 설계 같은 것들이다.

하지만 모든 것을 처음부터 완벽하게 정할 필요는 없다.

"memo를 어디에 써야 하지?"
"컴포넌트가 너무 커지는데 어떻게 쪼개지?"
"이 상태는 전역으로 관리해야 하나?"

이런 것들은 개발하면서 정해도 된다. 아니, 오히려 **개발하면서 정해야** 한다. 실제로 코드를 작성해봐야 어떤 문제가 있는지 알 수 있기 때문이다.

Part 2에서는 개발하면서 점진적으로 개선해도 되는 것들을 다룬다. 이것들을 처음부터 완벽하게 하려고 하면, 개발 속도만 느려지고 출시는 늦어진다.

---

1. 성능 최적화

1.1 왜 나중에 해도 되나?

처음부터 "최적화된 코드"를 작성하려고 하면? 개발 속도가 절반으로 느려진다.

"이 컴포넌트에 memo를 써야 하나?"
"여기서 useMemo를 써야 하나?"
"이건 lazy loading 해야 하나?"

이런 고민으로 하루를 보내면, 정작 기능은 만들지 못한다.

실제 사례

어떤 개발자가 처음부터 모든 컴포넌트에 `React.memo`를 감쌌다. "성능을 위해서"라고 했다. 하지만 실제로 측정해보니 성능 차이가 없었다. 오히려 코드만 복잡해졌다.

1.2 언제 최적화하나?

1) 사용자가 느릴 때

"페이지 로딩이 너무 느려요"
"스크롤이 버벅거려요"

이런 불평이 나올 때 최적화한다.

2) 개발자 도구로 확인했을 때

React DevTools Profiler로 보니 특정 컴포넌트가 불필요하게 10번씩 렌더링된다. 그때 `memo`를 추가한다.

3) Lighthouse 점수가 낮을 때

Lighthouse에서 Performance 점수가 50점이 나온다. 그때 원인을 분석하고 개선한다.

1.3 흔한 최적화 포인트

React.memo는 언제?

불필요한 리렌더링이 실제로 발생할 때만 쓴다.

```
// 부모가 리렌더링될 때마다 자식도 리렌더링
const UserCard = ({ user }) => {
  console.log('렌더링!')
  return <div>{user.name}</div>
}

// user props가 안 바뀌면 리렌더링 안 함
const UserCard = memo(({ user }) => {
  return <div>{user.name}</div>
})
```

하지만 처음부터 모든 컴포넌트를 memo로 감싸지 말자. 실제로 문제가 보일 때 추가한다.

useMemo는 언제?

계산 비용이 높은 연산에만 쓴다.

```
// 나쁜 예: 간단한 연산에 useMemo
const fullName = useMemo(() =>
  `${firstName} ${lastName}`,
  [firstName, lastName]
)

// 좋은 예: 복잡한 연산에만 useMemo
const sortedList = useMemo(() =>
  hugeArray.sort().filter(...).map(...),
  [hugeArray]
)
```

간단한 연산은 그냥 계산하는 게 오히려 빠르다. useMemo 자체도 비용이 들기 때문이다.

Code Splitting은 언제?

처음부터 모든 페이지를 lazy loading할 필요 없다. 번들 크기가 커졌을 때 추가한다.

```
// 처음: 모두 import
import HomePage from './pages/HomePage'
import MatchPage from './pages/MatchPage'

// 나중에: lazy loading
const HomePage = lazy(() => import('./pages/HomePage'))
const MatchPage = lazy(() => import('./pages/MatchPage'))
```

1.4 이미지 최적화

처음에는 신경 안 써도 됨

```
<img src="/images/user.jpg" alt="사용자" />
```

나중에 개선

```
<img
  src="/images/user.jpg"
  loading="lazy"
  width={200}
  height={200}
  alt="사용자"
/>
```

width, height를 명시하면 Layout Shift를 방지할 수 있다. loading="lazy"는 화면에 보일 때만 로드한다.

---

2. 상태 관리 복잡도

2.1 왜 나중에 정해도 되나?

처음부터 "전역 상태 관리가 필요해!"라고 Redux를 도입하면? 간단한 카운터 하나 만드는 데 파일 5개를 만들어야 한다.

2.2 단계별 상태 관리 전략

Phase 1: useState로 시작

```
const [user, setUser] = useState(null)
const [isLoading, setIsLoading] = useState(false)
```

대부분의 상태는 useState로 충분하다.

Phase 2: Props Drilling이 괴로울 때 Context API

3단계 이상 props를 내려보내고 있다면?

```
// Context 생성
const UserContext = createContext()

// Provider로 감싸기
<UserContext.Provider value={user}>
  <ChildComponent />
</UserContext.Provider>

// 사용
const user = useContext(UserContext)
```

Phase 3: Context도 복잡하면 Zustand

Context가 10개 넘게 생겼다면 Zustand를 고려한다.

```
// store 정의
const useUserStore = create((set) => ({
  user: null,
  setUser: (user) => set({ user }),
}))
```

```
// 사용
const user = useUserStore(state => state.user)
```

Phase 4: 정말 복잡하면 Redux

Redux는 최후의 선택지다. 대부분 프로젝트는 Zustand로 충분하다.

2.3 언제 전역 상태로 올리나?

전역 상태로 만들어야 하는 것

- 사용자 정보 (여러 컴포넌트에서 사용)
- 인증 토큰
- 테마 설정 (다크모드 등)

로컬 상태로 두어야 하는 것

- 폼 입력값
- 모달 open/close 상태
- 탭 선택 상태

기준: **3개 이상의 컴포넌트에서 사용되면** 전역 상태로 올린다.

---

3. 컴포넌트 분리 전략

3.1 왜 나중에 정해도 되나?

처음부터 "이 컴포넌트를 어떻게 쪼개지?"를 고민하면 시간만 간다. 일단 하나의 컴포넌트로 만들고, 복잡해지면 쪼갠다.

3.2 언제 쪼개나?

1) 컴포넌트가 200줄 넘을 때

100줄까지는 괜찮다. 200줄 넘어가면 읽기 어려워진다. 그때 쪼갠다.

2) 같은 로직이 반복될 때

같은 JSX가 3번 이상 반복되면 컴포넌트로 분리한다.

3) 책임이 명확히 나뉠 때

UserProfile 컴포넌트가 "프로필 표시 + 수정 + 삭제"를 모두 한다면? ProfileView, ProfileEdit, ProfileDelete로 쪼갠다.

3.3 어떻게 쪼개나?

나쁜 예: 너무 잘게 쪼갬

```
const UserName = ({ name }) => <span>{name}</span>
const UserEmail = ({ email }) => <span>{email}</span>
const UserAge = ({ age }) => <span>{age}</span>
```

이렇게 하면 컴포넌트만 늘어나고 가독성이 떨어진다.

좋은 예: 의미 있는 단위로

```
const UserBasicInfo = ({ user }) => (
  <div>
    <span>{user.name}</span>
    <span>{user.email}</span>
    <span>{user.age}</span>
  </div>
)
```

3.4 Custom Hook으로 로직 분리

컴포넌트는 UI만, 로직은 Hook으로 분리한다.

```
// Before: 컴포넌트에 로직이 섞임
const UserProfile = () => {
  const [user, setUser] = useState(null)
  const [isLoading, setIsLoading] = useState(false)

  useEffect(() => {
    setIsLoading(true)
    fetchUser().then(setUser).finally(() => setIsLoading(false))
  }, [])

  return <div>{user?.name}</div>
}

// After: Hook으로 로직 분리
const useUser = () => {
  const [user, setUser] = useState(null)
  const [isLoading, setIsLoading] = useState(false)
  // ... 로직
  return { user, isLoading }
}

const UserProfile = () => {
  const { user, isLoading } = useUser()
  return <div>{user?.name}</div>
}
```

---

4. 테스트 전략

4.1 왜 나중에 정해도 되나?

"테스트 커버리지 100%"를 목표로 하면? MVP는 영원히 출시되지 않는다.

테스트도 코드다. 테스트 코드 작성에도 시간이 든다. 처음부터 모든 컴포넌트에 테스트를 작성하면 개발 속도가 절반으로 느려진다.

4.2 MVP 단계의 테스트 전략

핵심 기능만 E2E 테스트

회원가입 → 로그인 → 매칭 → 게임 같은 주요 흐름만 테스트한다.

Playwright나 Cypress로:

```
test('사용자가 로그인할 수 있다', async ({ page }) => {
  await page.goto('/login')
  await page.fill('[name="email"]', 'test@example.com')
  await page.fill('[name="password"]', 'password123')
  await page.click('button[type="submit"]')
  await expect(page).toHaveURL('/home')
})
```

복잡한 유틸 함수만 단위 테스트

날짜 포맷팅, 유효성 검사 같은 순수 함수는 단위 테스트를 작성한다.

```
test('이메일 유효성 검사', () => {
  expect(isValidEmail('test@example.com')).toBe(true)
  expect(isValidEmail('invalid')).toBe(false)
})
```

컴포넌트 테스트는 나중에

단순 UI 컴포넌트는 테스트를 나중에 작성해도 된다. 버그가 적고, 시각적 확인이 더 정확하다.

4.3 테스트 커버리지 목표

처음부터 80%를 목표로 하지 말자

MVP 단계에서는 30-40%면 충분하다. 핵심 기능 위주로 테스트를 작성한다.

점진적으로 올린다

서비스가 안정화되면 50%, 60%로 올린다. 하지만 100%는 목표로 하지 않는다.

왜 100%가 아닌가?

간단한 컴포넌트, 타입 정의, 상수 파일. 이런 것들을 테스트하는 건 시간 낭비다. 의미 있는 로직만 테스트한다.

4.4 언제 테스트를 추가하나?

버그가 발견됐을 때

운영 중에 버그가 발견되면, 먼저 그 버그를 재현하는 테스트를 작성한다. 그다음 버그를 고친다. 이렇게 하면 같은 버그가 다시 발생하지 않는다.

리팩토링하기 전에

큰 리팩토링을 하기 전에, 기존 동작을 보장하는 테스트를 먼저 작성한다.

---

5. 에러 처리 전략

5.1 왜 나중에 개선해도 되나?

처음에는 단순하게 시작한다.

```
try {
  await fetchUser()
} catch (error) {
  console.error(error)
  alert('에러가 발생했습니다')
}
```

이 정도면 충분하다. 나중에 사용자가 늘어나면 개선한다.

5.2 점진적으로 개선하기

처음: try-catch + alert

```
try {
  await fetchUser()
} catch (error) {
  alert('에러가 발생했습니다')
}
```

개선 1: Toast UI 라이브러리

```
try {
  await fetchUser()
} catch (error) {
  toast.error('사용자 정보를 불러올 수 없습니다')
}
```

개선 2: Error Boundary

```
class ErrorBoundary extends Component {
  componentDidCatch(error) {
    console.error(error)
    this.setState({ hasError: true })
  }

  render() {
    if (this.state.hasError) {
      return <ErrorFallback />
    }
    return this.props.children
  }
}
```

개선 3: Sentry 연동

에러가 발생하면 자동으로 Sentry에 리포트된다.

5.3 언제 개선하나?

사용자가 혼란스러워할 때

"에러가 났는데 무슨 에러인지 모르겠어요"

이런 불평이 나오면 에러 메시지를 개선한다.

같은 에러가 자주 발생할 때

같은 에러가 하루에 100번씩 발생한다면, Sentry로 추적하고 근본 원인을 찾는다.

---

6. 접근성 (Accessibility)

6.1 왜 나중에 해도 되나?

접근성은 중요하다. 하지만 MVP 단계에서 완벽한 접근성을 구현하면 출시가 늦어진다.

6.2 최소한만 처음부터

시맨틱 HTML은 처음부터

```
// Bad
<div onClick={handleClick}>버튼</div>

// Good
<button onClick={handleClick}>버튼</button>
```

시맨틱 HTML을 쓰는 건 어렵지 않다. 처음부터 습관화한다.

alt 텍스트는 처음부터

```
<img src="/user.jpg" alt="사용자 프로필" />
```

이것도 어렵지 않다. 처음부터 작성한다.

6.3 나중에 개선

키보드 내비게이션

Tab으로 모든 요소에 접근할 수 있는가? 나중에 확인한다.

스크린 리더 지원

aria-label, aria-describedby 같은 것들. 나중에 추가한다.

색상 대비

WCAG 기준을 만족하는가? 디자인 시스템 확정 후 확인한다.

6.4 언제 개선하나?

사용자가 불편을 호소할 때

"키보드로 사용할 수 없어요"
"스크린 리더로 읽히지 않아요"

이런 피드백이 오면 개선한다.

법적 요구사항이 있을 때

공공기관이나 대기업 대상 서비스라면 웹 접근성 인증이 필요할 수 있다. 그때 본격적으로 개선한다.

---

7. SEO 최적화

7.1 왜 나중에 해도 되나?

SEO는 검색 엔진 최적화다. 하지만 초기 서비스는 검색보다 직접 유입이 많다.

사용자가 1,000명일 때는 SNS, 커뮤니티를 통해 들어온다. 검색으로 들어오는 비율은 5% 미만이다.

7.2 최소한만 처음부터

title과 description

```
<Helmet>
  <title>개그 배틀 - 웃음 참기 대결</title>
  <meta name="description" content="친구와 웃음 참기 대결을 해보세요" />
</Helmet>
```

이 정도만 해도 충분하다.

7.3 나중에 개선

SSR (Server-Side Rendering)

검색 엔진 크롤러가 JavaScript를 실행하지 못하면, CSR(Client-Side Rendering)은 SEO에 불리하다.

하지만 요즘은 대부분의 검색 엔진이 JavaScript를 실행한다. SSR은 꼭 필요할 때만 도입한다.

Open Graph 태그

SNS에 공유했을 때 예쁘게 보이려면 Open Graph 태그가 필요하다. 하지만 나중에 추가해도 된다.

구조화된 데이터

JSON-LD 같은 구조화된 데이터. 검색 결과에 별점, 이미지를 표시하려면 필요하다. 하지만 초기에는 과도하다.

7.4 언제 개선하나?

검색 유입이 중요해질 때

사용자가 10,000명을 넘어가면 검색 유입 비율이 올라간다. 그때 SEO를 본격적으로 개선한다.

---

8. 로깅 및 모니터링

8.1 왜 나중에 해도 되나?

사용자가 10명일 때는 모니터링이 필요 없다. 문제가 생기면 사용자가 직접 알려준다.

하지만 사용자가 1,000명이 되면? 모든 사용자의 불평을 직접 듣기 어렵다. 그때 모니터링이 필요하다.

8.2 단계별 모니터링

Phase 1 (MVP): console.log

```
console.log('사용자 로그인:', user)
console.error('API 에러:', error)
```

이 정도면 충분하다.

Phase 2 (성장기): 에러 추적

Sentry를 추가한다.

```
Sentry.init({
  dsn: 'your-dsn',
})

// 에러 자동 캡처
try {
  await fetchUser()
} catch (error) {
  Sentry.captureException(error)
}
```

Phase 3 (확장기): 분석 도구

Google Analytics, Mixpanel 같은 도구로 사용자 행동을 분석한다.

8.3 언제 추가하나?

버그를 찾기 어려울 때

"사용자가 에러가 났다고 하는데, 무슨 에러인지 모르겠어요"

이럴 때 Sentry를 추가한다.

데이터 기반 의사결정이 필요할 때

"어떤 기능을 가장 많이 사용하나요?"
"사용자들이 어디서 이탈하나요?"

이런 질문에 답하려면 분석 도구가 필요하다.

---

9. 폴더 구조 세분화

9.1 왜 나중에 해도 되나?

처음부터 완벽한 폴더 구조를 만들 필요 없다. 파일이 늘어나면서 자연스럽게 구조가 잡힌다.

9.2 점진적으로 세분화

처음: 단순하게

```
src/
  components/
  pages/
  hooks/
  utils/
```

파일이 늘어나면: 기능별로

```
src/
  features/
    user/
    match/
```

```
  shared/
    components/
    hooks/
```

더 복잡해지면: 세분화

```
features/
  user/
    components/
    hooks/
    api/
    types/
    utils/
```

9.3 언제 세분화하나?

한 폴더에 파일이 10개 넘을 때

components 폴더에 파일이 30개 있으면 찾기 어렵다. 그때 atoms, molecules로 나눈다.

기능별로 명확히 구분될 때

User 기능과 Match 기능이 완전히 독립적이라면, features로 분리한다.

---

10. 타입 정의 세밀화

10.1 왜 나중에 해도 되나?

처음부터 완벽한 타입을 만들 필요 없다. 일단 any로 시작해도 된다.

```
// 처음
const user: any = fetchUser()

// 나중에 개선
interface User {
  id: string
  name: string
  email: string
}
const user: User = fetchUser()
```

10.2 언제 개선하나?

타입 에러가 자주 발생할 때

"undefined를 읽을 수 없습니다" 에러가 자주 나오면, 타입을 명확히 정의한다.

API 스펙이 확정됐을 때

백엔드 API가 확정되면, 그때 정확한 타입을 정의한다.

Part 2 체크리스트

당장은 아니지만 곧 필요한 것들

- [ ] 주요 흐름 E2E 테스트 작성
- [ ] 에러 처리 Toast UI 추가
- [ ] 기본 SEO 태그 추가

사용자가 늘어나면 필요한 것들

- [ ] React.memo 최적화
- [ ] Code Splitting 추가
- [ ] Sentry 에러 추적
- [ ] Google Analytics 연동

성능 문제가 보이면 필요한 것들

- [ ] 이미지 lazy loading
- [ ] 번들 크기 분석 및 최적화
- [ ] Lighthouse 점수 개선
- [ ] 접근성 개선

마치며

"완벽함은 선의 적이다" - Voltaire

처음부터 완벽하게 만들려고 하지 말자.

일단 작동하게 만든다. 그다음 사용자 피드백을 받는다. 문제가 보이면 그때 개선한다.

"나중에 성능 문제가 생기면 어떡하지?"라고 걱정하지 말자. 성능 문제는 **생겼을 때** 해결하면 된다. 지금은 **출시**가 목표다.

React.memo를 언제 써야 하는지 고민하는 시간에, 하나의 기능을 더 만드는 게 낫다.

MVP를 출시하지 못하면, 최적화는 의미가 없다. 아무도 사용하지 않는 완벽한 앱보다, 많은 사람이 사용하는 개선 여지가 있는 앱이 낫다.

Part 3에서는 실제로 프로젝트를 생성하고, 첫 페이지를 만들고, 배포하는 과정을 다룬다. Hello World부터 Vercel 배포까지. 🚀