

React→Next.js 마이그레이션 비용 분석 & 모바일 전략

1. React+Vite → Next.js 마이그레이션 비용

규모별 마이그레이션 시간 추정

앱 규모	컴포넌트 수	라우트 수	마이그레이션 시간	비용 (1인 개발)	영향도
소형	20-50	5-10	3-5일	\$3K-5K	낮음
중형	50-150	10-30	1-2주	\$5K-10K	중간
대형	150-500	30-100	3-8주	\$15K-40K	높음
초대형	500+	100+	2-3개월	\$40K+	매우 높음

구체적 마이그레이션 작업 분해

1단계: 프로젝트 설정 (1-2일)

React+Vite 프로젝트

package.json 재구성

Vite 설정 제거

Next.js 초기화

tsconfig.json 수정

비용: 낮음 (자동화 가능)

2단계: 폴더 구조 변경 (2-3일)

Before (React+Vite):

src/

```
|— components/
|   |— Dashboard.tsx
|   |— Portfolio.tsx
|   |— ...
|— pages/
|   |— dashboard.tsx
|   |— portfolio.tsx
|— hooks/
|— utils/
|— App.tsx + Router 설정
```

After (Next.js):

app/

```
|— layout.tsx (새로 작성)
|— page.tsx (대시보드)
|— dashboard/
|   |— page.tsx
|— portfolio/
|   |— page.tsx
|— api/ (새 영역)
```

components/

utils/

hooks/

작업:

- 라우팅 로직 변경 (React Router → Next.js App Router)
- 폴더 구조 재조직
- import 경로 모두 수정

비용: 중간 (수작업 많음)

3단계: 라우팅 변경 (3-5일)

typescript

```
// Before (React Router)
import { BrowserRouter, Routes, Route } from 'react-router-dom'

export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element=<Dashboard /> />
        <Route path="/portfolio" element=<Portfolio /> />
        <Route path="/portfolio/:id" element=<PortfolioDetail /> />
      </Routes>
    </BrowserRouter>
  )
}

// After (Next.js)
// app/page.tsx
export default function Dashboard() { return <Dashboard /> }

// app/portfolio/page.tsx
export default function PortfolioList() { return <Portfolio /> }

// app/portfolio/[id]/page.tsx
interface Props { params: { id: string } }
export default function PortfolioDetail({ params }: Props) { ... }
```

모든 라우트를 파일 기반으로 변환

비용: 높음 (라우트 수에 비례)

4단계: 상태관리 유지 (0-3일)

```
typescript

// Zustand는 그대로 사용 가능
// pages/dashboard.tsx에서
import { usePortfolioStore } from '@/store'

export default function Dashboard() {
  const portfolio = usePortfolioStore()
  // 그대로 사용 가능
}
```

비용: 낮음 (Zustand는 호환성 완벽)

5단계: 데이터 페칭 변경 (3-5일)

typescript

```
// Before (클라이언트 전용)
useEffect(() => {
  fetch('/api/portfolio')
    .then(res => res.json())
    .then(data => setPortfolio(data))
}, [])

// After (서버 컴포넌트 권장)
import { getPortfolio } from '@lib/api'

export default async function Dashboard() {
  const portfolio = await getPortfolio() // 서버에서 데이터 페칭
  return <DashboardClient portfolio={portfolio} />
}
```

비용: 높음 (모든 데이터 페칭 지점 변경)

6단계: API 라우트 생성 (1-3일)

Next.js는 `/pages/api`가 없어도 되지만
기존 API 호출을 그대로 유지하려면:

```
// app/api/portfolio/route.ts
export async function POST(request: Request) {
  const body = await request.json()
  // 백엔드 호출
  return Response.json(result)
}
```

비용: 낮음 (기존 API 엔드포인트 유지하면 변경 최소)

7단계: 테스트 및 디버깅 (3-7일)

- 모든 라우트 동작 확인
- 상태 관리 재검증
- 성능 최적화 확인
- 배포 파이프라인 재구성

비용: 높음

마이그레이션 시간 요약

프로토타입 규모 (소형)

- 컴포넌트 30개
- 라우트 8개
- 상태관리 간단 (Zustand)
- 마이그레이션: 3-5일 (당신 혼자)

실제 시간 분해:

- 셋업 및 폴더 구조: 1일
- 라우팅 변경: 1-2일
- 데이터 페칭 수정: 1-2일
- 테스트: 1일
- 버그 수정 및 최적화: 1일

결론: 프로토타입 단계면 3-5일, 큰 앱이면 2-3주

2. 모바일 앱 병행의 영향

모바일 전략이 현재 기술 선택에 미치는 영향

시나리오: 웹 SPA → 모바일 앱 확장

고려사항

- 코드 재사용성
- 데이터 페칭 로직 공유
- 비즈니스 로직 공유
- UI 컴포넌트 재사용

Option 1: React Web + React Native (모바일)

공유 가능한 것:

- 비즈니스 로직 (Zustand 스토어)
- API 클라이언트 (axios/fetch 로직)
- 타입 정의 (TypeScript 타입)
- 유틸 함수

UI는 별도:

- 웹: React+Vite (또는 Next.js)
- 모바일: React Native (완전히 다른 UI)

평가

코드 재사용율: 30-40% (비즈니스 로직만)
개발 비용: 높음 (두 개 앱 병행)
팀 크기: 웹 개발자 + 모바일 개발자

Option 2: Monorepo 구조 (권장)

구조 예시

```

portfolio-monorepo/
├── packages/
│   ├── core/           # 공유 코드 ★
│   │   ├── store/      # Zustand 스토어
│   │   ├── api/        # API 클라이언트
│   │   ├── types/      # TypeScript 타입
│   │   └── utils/       # 공유 함수
│   ├── ui/             # 웹 UI 컴포넌트 (선택)
│   │   ├── Button
│   │   ├── Card
│   │   └── Chart
│   └── mobile-ui/       # 모바일 UI 컴포넌트
│       ├── Button
│       ├── Card
│       └── Chart
├── apps/
│   ├── web/            # 웹 앱 (React+Vite 또는 Next.js)
│   │   ├── src/
│   │   │   ├── pages/
│   │   │   ├── components/
│   │   │   └── App.tsx
│   └── mobile/          # 모바일 앱 (React Native)
│       ├── src/
│       │   ├── screens/
│       │   ├── components/
│       │   └── App.tsx
└── turbo.json           # Monorepo 설정
  
```

장점

- ✅ 코드 재사용율: 50-70% (core 패키지 공유)
- ✅ 한 번에 관리 (git, 버전 관리)
- ✅ 타입 안전성 (공유 타입 정의)
- ✅ 배포 자동화 (영향받는 앱만 배포)

예시: API 클라이언트 공유

typescript

```
// packages/core/api/client.ts
export const api = axios.create({
  baseURL: process.env.REACT_APP_API_URL,
})

export const portfolioAPI = {
  list: () => api.get('/portfolio'),
  detail: (id: string) => api.get(`/portfolio/${id}`),
  optimize: (data: Portfolio) => api.post('/portfolio/optimize', data),
}

// apps/web/src/hooks/usePortfolio.ts
import { portfolioAPI } from '@core/api'
export const usePortfolios = () => {
  return useQuery({
    queryKey: ['portfolios'],
    queryFn: portfolioAPI.list,
  })
}

// apps/mobile/src/hooks/usePortfolio.ts
import { portfolioAPI } from '@core/api'
export const usePortfolios = () => {
  return useQuery({
    queryKey: ['portfolios'],
    queryFn: portfolioAPI.list,
  })
}
```

비즈니스 로직 공유

typescript


```
// packages/core/store/portfolioStore.ts
import { create } from 'zustand'

export const usePortfolioStore = create((set) => ({
  portfolio: null,
  setPortfolio: (data) => set({ portfolio: data }),
  optimize: async (constraints) => {
    const result = await portfolioAPI.optimize(constraints)
    set({ portfolio: result })
    return result
  },
}))

// 웹과 모바일에서 동일하게 사용
import { usePortfolioStore } from '@core/store'

// 웹에서
export function WebDashboard() {
  const { portfolio, optimize } = usePortfolioStore()
  return <Dashboard data={portfolio} onOptimize={optimize} />
}

// 모바일에서
export function MobileDashboard() {
  const { portfolio, optimize } = usePortfolioStore()
  return <Dashboard data={portfolio} onOptimize={optimize} />
}
```

3. 기술 선택: 모바일까지 고려한 전략

현 상황 종합 분석

당신의 상황:

- └── 혼자 개발
- └── 프로토타입 단계
- └── 웹 SPA 먼저 필요
- └── 향후 모바일 앱 고려
- └── 장기적으로 스케일 예상

추천 경로 (3가지)

경로 1: React+Vite → Monorepo로 확장 (추천 ★★★★★)

Timeline

- Month 0-2: React+Vite로 웹 프로토타입 완성
- Month 2-4: Monorepo 구조로 리팩토링 (코드 재사용 준비)
- Month 4-6: React Native 모바일 앱 병행 개발
- Month 6+: 웹 + 모바일 병렬 운영

작업 분해

- Month 0-2 (프로토타입):
- React+Vite 개발
 - 비용: 저 (현재 계획대로)
- Month 2-4 (리팩토링):
- packages/core 생성
 - API 클라이언트 추출
 - 상태관리 정리
 - 공유 타입 정의
 - 비용: 1주 (코드 재구성)
- Month 4-6 (모바일):
- React Native 앱 개발
 - packages/core 재사용
 - 비용: 낮음 (코드 50% 재사용)

마이그레이션 비용: 없음 (React+Vite 유지, Monorepo 추가)

장점

- ✓ 초기 개발 속도 빠름 (React+Vite)
- ✓ 모바일 개발 시 코드 재사용 (50-70%)
- ✓ 장기적으로 확장성 높음
- ✓ 팀 성장 시에도 대응 가능

구체적 예시

packages/core/api/portfolioAPI.ts

- 웹에서 사용
- 모바일에서도 사용

apps/web/src/pages/Dashboard.tsx

- React+Vite 개발

apps/mobile/src/screens/Dashboard.tsx

- React Native 개발

경로 2: 지금 바로 Next.js + Monorepo 시작 (전략적)

이 경우가 좋은 이유

- ✓ 모바일도 고려하면 Monorepo 필수
- ✓ Monorepo = Next.js와 궁합 좋음
- ✓ Next.js의 API 라우트 활용 가능
- ✓ 나중에 마이그레이션 할 필요 없음

구조

```
portfolio-monorepo/  
├── packages/  
│   ├── core/ (API, 상태관리, 타입)  
│   └── ui-web/ (Next.js UI 컴포넌트)  
├── apps/  
│   ├── web/ (Next.js)  
│   └── mobile/ (React Native)  
└── turbo.json
```

Timeline

- Month 0-1: Monorepo + Next.js 셋업
- Month 1-2: 프로토타입 개발 (약간 느낌)
- Month 2-4: 모바일 추가 (코드 50% 재사용)

비용

초기 셋업: +2-3일
장기 이득: +50% 코드 재사용

장점

- ✓ 처음부터 올바른 구조
- ✓ 마이그레이션 비용 0
- ✓ 모바일 개발 시 효율 극대
- ✓ 팀 확대 시 구조 완벽

단점

- ✗ 초기 셋업 복잡 (당신 혼자기 힘들 수 있음)
- ✗ 초기 개발 속도 약간 느림
- ✗ 학습곡선 높음 (Turborepo, Next.js)

경로 3: React+Vite만 진행, 나중에 판단

이 경우

Month 0-2: React+Vite 웹 완성
Month 2+: 모바일 필요성 판단 후 결정

위험성

- ✗ 나중에 Monorepo 구조로 리팩토링 필요 (2-3주)
- ✗ 코드 재사용 구조 미리 설계 못함
- ✗ 마이그레이션 비용 발생

4. 최종 추천: 당신의 상황에 최적 경로

당신의 조건

- 혼자 개발 (협업 X)
- 빠른 프로토타입 필요
- 향후 모바일 앱 고려
- 장기 유지보수 필요

🎯 추천: 경로 1 (React+Vite → Monorepo)

NOW (Month 0-2): React+Vite로 신속하게 프로토타입 완성

- └─ 장점: 빠름, 간단, 배우기 쉬움
- └─ 아키텍처: 일단 신경 쓰지 않음
- └─ 목표: 비즈니스 로직 검증

LATER (Month 2-4): Monorepo로 리팩토링

- └─ 장점: 코드 재사용 준비
- └─ 비용: 1주 (관리 가능)
- └─ 목표: 모바일 개발 준비

NEXT (Month 4-6+): React Native 모바일 앱

- └─ 장점: 50-70% 코드 재사용
- └─ 비용: 낮음
- └─ 목표: 웹 + 모바일 동시 운영

5. 구체적인 실행 계획

Phase 1: 웹 프로토타입 (Month 0-2) - 지금

```
bash

npm create vite@latest portfolio-web -- --template react-ts
cd portfolio-web
npm install axios zustand react-router-dom
npm run dev
```

폴더 구조 (의도적으로 간단하게)

```
src/
└─ components/
└─ pages/
└─ hooks/
└─ store/
└─ utils/
└─ api/
```

Phase 2: Monorepo 리팩토링 (Month 2-4)

```
bash
```

```
# Turborepo 마이그레이션
```

```
npx create-turbo@latest
```

```
# 결과 구조
```

```
├── packages/  
│   ├── core/  
│   │   ├── api/  
│   │   ├── store/  
│   │   ├── types/  
│   │   └── utils/  
│   └── tsconfig/  
└── apps/  
    └── web/ (기존 React+Vite)
```

구체적 작업

1. `packages/core/api/` 에 기존 API 클라이언트 이동
2. `packages/core/store/` 에 Zustand 스토어 이동
3. `packages/core/types/` 에 TypeScript 타입 정의
4. `apps/web/` 에서 `@core/api`, `@core/store` import

코드 예시

```
typescript
```

```
// 기존 (apps/web/src/api/client.ts)
```

```
export const api = axios.create(...)
```

```
// 이동 후 (packages/core/api/client.ts)
```

```
export const api = axios.create(...)
```

```
// 웹에서 import
```

```
import { api } from '@core/api'
```

Phase 3: 모바일 앱 (Month 4-6+)

```
bash
```

```
npx create-expo-app@latest mobile
```

결과

apps/

├── web/ (React+Vite)

└── mobile/ (React Native/Expo)

모바일에서도 동일하게 사용

typescript

```
// apps/mobile/src/screens/Dashboard.tsx
```

```
import { usePortfolioStore } from '@core/store'
```

```
import { portfolioAPI } from '@core/api'
```

```
export function Dashboard() {
```

```
  const portfolio = usePortfolioStore()
```

```
  // 웹과 동일한 로직
```

```
}
```

코드 재사용율

packages/core: 100% 재사용

├── API 클라이언트

├── 상태관리

├── 타입 정의

└── 유틸 함수

UI 컴포넌트: 0% (별도로 개발)

├── 웹: React 컴포넌트

└── 모바일: React Native 컴포넌트

결론: 당신의 상황에 최적 선택

당신의 조건 재분석

- 혼자 개발
- 프로토타입 단계
- 모바일 필요성 불확실
- 두 트랙 병행 시 리소스 분산

최종 권장: React+Vite 단일 트랙

지금: React+Vite로 집중

└─ 개발 속도 최고

└─ 번들 크기 최소





└─ 의사결정 최소

Month 2-3: 프로토타입 완성 후 판단






└─ 모바일 필요? YES → Monorepo 리팩토링 (1주)

└─ 모바일 불필요? NO → 그대로 진행

결과:

-  하나에 집중 (완성도 높음)
-  유연한 의사결정 (나중에 선택 가능)
-  리소스 효율 최고
-  Monorepo 미리 준비 가능 (구조)

두 트랙 병행의 현실

-  리소스 분산 (50% + 50%)
-  둘 다 70% 수준 완성
-  의사결정 어려움
-  시간 낭비
-  혼자 개발에는 비현실적

추천 구체 액션

1. **지금:** React+Vite로 시작
2. **코드 작성 시:** 나중에 Monorepo 이동 가능하도록 설계
3. **Month 2-3:** 모바일 필요성 판단
4. **필요 시:** Monorepo 리팩토링 (1주)
5. **불필요 시:** 그대로 진행 또는 팀 확대 시 다시 검토