

W06  
리팩토링과 아키텍처

다음 기능을 추가하기 전에

# 의도대로 동작하면 출시가 완료되지만

지금까지는 "동작하는 것"에 집중했다.

- 로그인이 되는가?
- 데이터가 저장되는가?
- 화면이 제대로 나오는가?

하지만 "어떻게 만들어졌는가"는...

- 코드가 깔끔하게 정리되어 있는가?
- 파일들이 일관된 방식으로 연결되어 있는가?
- 다음 기능을 넣기 좋은 상태인가?

→ 아무도 확인 안 했다. AI도 신경 쓰지 않았다.

# 기술 부채

새 기능 추가해달라고 했더니, 되던 기능이 안 된다

- "이 부분 수정해줘" 했더니 AI가 엉뚱한 파일을 건드린다
- 비슷한 화면인데 이전이랑 완전 다른 방식으로 만들어놨다
- AI 응답이 점점 길어지고, 뭘 바꿨는지 따라가기 어렵다

기술 부채(Technical Debt) = 당장은 동작하지만, 나중에 문제가 될 수 있는 기술적 결함

- 빚처럼 쌓인다
- 지금은 티가 안 난다
- 하지만 이자가 붙는다 (점점 고치기 어려워진다)

"다음 기능 추가하기 전에, 지금 상태를 점검해야 한다"

# 리팩토링이란

- 소프트웨어의 겉으로 드러나는 기능은 그대로 유지하면서 내부 코드 구조를 개선하여 가독성, 유지보수성, 효율성을 높이는 체계적인 프로세스

기능은 그대로 두고, 코드만 정리하는 것

- 사용자가 보기에는 아무것도 안 바뀐다
- 로그인이 되던 건 여전히 된다
- 게시판이 되던 것도 여전히 된다
- 하지만 안쪽 코드는 깔끔해진다

# 리팩토링을 하는 이유

리팩토링을 안 하면:

- 같은 로직이 3곳에 흩어져 있음 → 한 곳만 고쳐달라고 하면 나머지는 그대로 → 버그
- 파일 하나가 500줄 → AI가 어디를 고쳐야 할지 헷갈림 → 엉뚱한 곳 수정
- 비슷한 기능이 다른 방식 → 새 기능도 또 다른 방식 → 점점 혼란

리팩토링 원칙

- **가독성** 향상: 코드를 이해하기 쉽게 만들어 다른 개발자(혹은 미래의 나)가 쉽게 파악하고 수정할 수 있도록 합니다.
- **유지보수성** 개선: 변경 사항이 시스템 전체에 미치는 영향을 최소화하고, 버그를 수정하거나 기능을 추가하기 용이하게 만듭니다.
- **효율성** 증가: 중복 코드를 제거하고, 로직을 단순화하여 코드의 복잡도를 낮추고 효율성을 높입니다.

# 리팩토링은 안전함이 핵심

## 리팩토링 목적은 정리

- 기능이 바뀌면 안 된다: 리팩토링 전에 되던 것 → 리팩토링 후에도 되어야 함
- 한 번에 조금씩: 전체를 한꺼번에 고치면, 문제가 생겼을 때 어디가 원인인지 모름
- 바꿀 때마다 확인: 하나 고침 → 동작 확인 → 다음 거 고침

정리하다가 망가뜨리면 의미가 없다.

# 질문 1. 현재 코드를 분석해 리팩토링 요소 발견하기

“이 프로젝트 전체 파일을 모두 탐색해서 가독성, 유지보수성, 효율성을 높이기 위해 리팩토링해야 할 것들에 대해 정리하고 우선순위를 선정해줘.”

## 질문 2. 안전한 리팩토링 계획 수립하기

“우선순위 X, X, X 번의 리팩토링을 진행하기 위한 계획을 수립해줘. 리팩토링 과정에서 기능이 동작하지 않거나 오류가 생기지 않고 목적을 달성할 수 있는 안전하면서도 효율적인 절차를 명확하게 설계해줘.”

“계획을 보고 다시한번 검토 후 납득이 되면 진행해줘”

“앞서 설계한 계획에 맞춰 리팩토링을 진행해줘”

“리팩토링이 완료된 이후에도 기능이 정상동작하는지 확인해줘”

# 프로젝트가 너무 크다면?

단계적으로 파악하는 절차를 설계해서 AI의 맥락 유지를 돋는다

1. 리팩토링을 하기 위해 전체 구조를 파악해줘
2. 파악한 구조를 바탕으로 코드를 단계적으로 탐색할 방안을 수립해줘
3. 수립한 방안에 맞게 단계적으로 코드를 모두 검토해서 리팩토링이 필요한 부분을 파악해줘
4. 파악한 내용들을 종합해서 우선순위를 정해줘

# 리팩터링으로 무엇이 변하는지 알고 있어야 한다

리팩토링 전에 확인할 것:

- 어떤 파일이 바뀌는가?
- 그 파일과 연결된 다른 기능은?

리팩토링 후에 확인할 것:

- 기존에 되던 것이 여전히 되는가?
- AI가 말한 대로 바뀌었는가?

"뭐가 바뀌는지 모르면, 뭐가 망가졌는지도 모른다"

- 각 리팩터링 내용에 대해 충분히 학습하고 진행한다
- 문제가 생기면? Git으로 되돌린다

# 지금 상태에 새로운 기능을 더 추가할 수 있을까?

- 리팩토링은 "지금 있는 코드를 정리"하는 것이었다.

하지만 다른 문제도 존재

- 지금은 로그인 없이 만들었다 → 회원별 데이터 저장 기능을 추가하려면?
- 지금은 브라우저에만 저장한다 → 다른 기기에서도 쓰려면?
- 지금은 혼자 쓴다 → 여러 사람이 동시에 쓰려면?

→ 코드를 아무리 정리해도, 애초에 구조가 안 받쳐주면 못 만든다

# AI의 선택은 '지금' 만드는 것에 초점을 두고 있다

AI가 코드를 만들 때:

- 첫 번째 PRD에 있는 기능을 만드는데 집중했다
- 미래에 뭘 추가할지는 고려하지 않았다
- "지금 당장 동작하는 가장 빠른 방법"을 선택했다

새로운 기능을 추가하면

- 지금 기술로는 구현 불가능한 기능이 있을 수 있다
- 지금 구조로는 새 기능을 넣을 자리가 없을 수 있다
- 기술 스택은 한번 정하면 바꾸기가 힘들지만...

# 서비스 확장을 위해서 어떤 것들이 달라져야 할까

기술 스택 = 지금 쓰는 도구로 다음 기능을 만들 수 있는가?

- 프레임워크, 라이브러리, 데이터베이스 등
- “무엇으로 만들었는가”

아키텍처 = 지금 구조에 다음 기능을 넣을 자리가 있는가?

- 파일들이 어떻게 연결되어 있는가
- 데이터가 어떻게 흐르는가
- "어떤 구조로 만들어졌는가"

# 기술 스택의 변화

기술 스택 = 프로젝트를 만드는데 쓰는 도구들 (프레임워크, 라이브러리, 데이터베이스)

- 예시: `localStorage` → `Supabase`로 변경
- 지금: 데이터를 브라우저 `localStorage`에 저장하고 있다
- 문제: 다른 기기에서 접속하면 데이터가 없다, 여러 사용자가 못 쓴다

변경하면 실제로 일어나는 일:

- `Supabase` 라이브러리 설치
- 데이터 저장하는 코드 전부 다시 작성
- 데이터 불러오는 코드 전부 다시 작성
- 기존 `localStorage`에 있던 데이터는 날아감

# 소프트웨어 아키텍처

- 시스템의 전체적인 구조와 구성 요소 간의 관계, 그리고 이들이 어떻게 상호작용하는지를 결정하는 고차원의 설계 원칙

소프트웨어 아키텍처가 왜 중요한가?

- 복잡성 관리: 시스템이 커져도 구조가 명확하면 관리가 쉽습니다.
- 효율적인 협업: 개발자들이 각자 어떤 부분을 담당하고 어떻게 연결되는지 이해할 수 있습니다.
- 변경 유연성: 기술 스택을 바꾸거나 기능을 추가할 때 시스템 전체가 무너지는 것을 방지합니다.

# 아키텍처(구조)의 변화로 생기는 일

- 예시: 폴더 하나에 모두 작성 → 클린 아키텍처로 변경
- 지금: App.js 하나에 화면, 로직, API 호출이 전부 들어있다 (500줄)
- 문제: AI가 어디를 고쳐야 할지 헷갈림, 수정할 때마다 다른 곳이 깨짐

변경하면 실제로 일어나는 일:

- components, hooks, api 폴더 새로 생성
- 500줄짜리 파일을 10개 파일로 쪼갬
- 파일 간 import/export 연결 전부 새로 작성
- 쪼개는 과정에서 잘 되던 기능이 안 될 수 있음

# 질문 3. 현재 코드의 기술 스택과 아키텍처 파악하기

“지금 프로젝트 전체를 분석해서 어떤 기술 스택과 아키텍처로 구성되어 있는지 정리해줘”

“각 정의된 기술 스택과 아키텍처가 어떤 이유로 선정되었는지를 설명해줘”

## 질문 4. 확장성을 위해 바뀌어야 하는 아키텍처, 기술 스택

“앞으로 XXX, XXXX, XXXXXX 등의 기능이 추가되고, 지금 시스템 개요의 내용대로 서비스가 확장해 나가는 것을 감안할 때 변경해야 하는 아키텍처와 기술 스택이 있는지 검토해줘. 미래 기능을 너무 확대 해석해서 무리한 기술 스택과 아키텍처를 선정하지 않도록 조심해줘”

- 변경이 필요한 것들 중 필수적, 선택적인 것이 무엇인지 나누어 확인한다
- 구체적으로 어떤 기능의 추가가 있을 때 변경이 필요한지 파악한다.
- 변경 난이도에 맞춰 기능 구현 로드맵을 조정하는 것도 고려한다.

# 어떤 이유로 바뀌어야 하는지 확실히 파악해야 한다

기술 스택, 아키텍처의 변화는 엄청나게 큰 변화

- 지금까지 만든 코드 상당 부분을 다시 작성해야 할 수 있다
- 잘 되던 기능이 갑자기 안 될 수 있다
- 한번 바꾸면 되돌리기 어렵다

바뀌어야 하는 이유가 확실해야만 한다

- AI가 "바꾸는 게 좋다"고 해서 바꾸는 게 아니다
- "이 기능을 만들려면 반드시 필요하다"는 근거가 있어야 한다
- 더 작은 변경으로 해결할 수 있는지 먼저 확인한다

안전하게 바꾸지 않으면 프로젝트 전체가 무너진다

- 브랜치 만들고 시작한다
- 한 번에 전부 바꾸지 않는다
- 단계마다 동작 확인한다

## 질문 5. 아키텍처, 기술 스택 변경을 위한 체계적 계획 수립

"XXXX로을 진행하기 위한 계획을 수립해줘. 변경 과정에서 기능이 동작하지 않거나 오류가 생기지 않고 목적을 달성할 수 있는 안전하면서도 효율적인 절차를 명확하게 설계해줘. 한 번에 전부 바꾸지 말고, 단계별로 나눠서 진행할 수 있게 설계해줘."

# 오늘 해결해야 할 문제

1. 내 프로젝트에서 리팩토링이 필요한 부분을 AI에게 분석 요청하기
2. 우선순위가 높은 것 1~2개를 선정해서 리팩토링 계획 수립하기
3. 필수 개선에 대해 실제 리팩토링 실행하기
4. 현재 기술 스택과 아키텍처를 AI에게 정리 요청하기
5. 로드맵의 다음 기능을 위해 변경이 필요한 기술 스택, 아키텍처가 있는지 확인하기

# Q&A