이력서 바탕 QnA

1. MVVM / UseCase / 레이어드 아키텍처 이해도

Q1. “어딧삼 프로젝트에서 MVVM + UseCase 레이어를 적용했다고 하셨는데, 각 레이어의 책임을 어떻게 나누셨는지 구체적으로 설명해 주실 수 있을까요?”

- **메인 답변 (10점) [추론]**

  “어딧삼에서는 크게 세 레이어로 나눴습니다.

  1. **Presentation 레이어 (View / ViewModel)**
     - View는 말 그대로 ‘그리기’와 사용자 입력만 담당했습니다.
       상태는 전부 ViewModel에서 받아서 `build`에 반영하는 형태였고요.
     - ViewModel은
       - 화면에서 필요한 상태를 들고 있고
       - 유즈케이스를 호출해서 도메인 로직을 실행하고
       - 결과를 Presentation-friendly 한 상태로 변환하는 역할에 집중시켰습니다.
         여기에는 네트워크/DB 호출이 직접 들어가지 않도록 했습니다.
  2. **Domain 레이어 (UseCase)**
     - 비즈니스 규칙이 들어가는 곳이라,
       ‘사용자 차단 여부에 따라 채팅 방에 입장할 수 있는가?’ 같은 질문을
       UseCase로 분리했습니다.
     - 여러 화면에서 동일한 규칙을 써야 하는 로직도 이 레이어에 두고,
       ViewModel은 단순히 ‘이 UseCase를 언제 호출할 것인가’를 결정하는 정도로 줄였습니다.
  3. **Data 레이어 (Repository + DataSource)**
     - Firebase / Cloud Functions / REST API와 통신하는 부분은
       Repository가 담당했고,
       그 안에서 DTO ↔ 도메인 모델 변환이 일어났습니다.

- ViewModel은 Repository 인터페이스에만 의존하게 해서,
  테스트 시에는 Fake/MOCK Repository를 주입할 수 있도록 했습니다.

특히 사용자 차단 기능이 추가되면서 if/else가 폭발하는 구간이 있었는데,

그때 '이건 이제 ViewModel 책임이 아니다'라고 보고

UseCase로 규칙을 옮기면서 ViewModel을 다시 얇게 만들었습니다."

- **꼬리질문 1**

  "같은 구조를 저희 '타이머 → 포만감 → 기록' 흐름에 적용한다면, UseCase는 어떤 단위로 나누고, ViewModel(혹은 Provider)은 어디까지 책임지게 설계하실 것 같나요?"

  - **답변 (10점) [추론]**

    "LoopOS 흐름에 그대로 적용해보면,

    - **UseCase**는

      1. `StartMealSessionUseCase` – 식사 세션 시작(타이머 초기화, 설정값 로드)
      2. `CompleteBiteUseCase` – 한입 종료 시 기록 추가 / 리듬 평가
      3. `CompleteMealUseCase` – 전체 타이머 종료, 실제 duration 계산
      4. `SaveMealRecordUseCase` – 포만감/감정 선택 포함해서 최종 기록 저장

      정도로 쪼갤 수 있을 것 같습니다.

    - **ViewModel/Provider**는
      - 지금 사용자가 어느 단계(타이머/포만감/리뷰)에 있는지 상태를 들고 있고,
      - 버튼/탭 이벤트를 받아서 어느 UseCase를 호출할지만 결정합니다.
      - 호출 결과를
        - 타이머 남은 시간
        - 한입 리스트
        - 포만감 선택값 등

        화면에서 바로 쓸 수 있는 형태로 노출하는 역할이 되겠죠.

    요약하면,

    - '시간이 어떻게 흘렀는가 / 어떤 데이터를 남길 것인가' 같은 규칙은 UseCase,

    - ‘언제 그 규칙을 실행할 것인가 / 어떤 화면을 보여줄 것인가’는 ViewModel/Provider 책임으로 나누고 싶습니다.”

- **꼬리질문 2**

  “레거시 프로젝트에서 MVVM/UseCase로 리팩터링할 때, **‘처음 손대는 지점’**은 보통 어디라고 보시나요? 실제 경험 기준으로 얘기해 주세요.”

  - **답변 (10점) [추론]**

    “경험상 처음 손대는 지점은 두 가지입니다.

    1. **ViewModel에서 가장 지저분한 메서드**
        - if/else가 길고,
        - API 호출 / 로컬 저장 / 상태 변경이 뒤섞여 있는 메서드를 골라서 그 로직을 UseCase로 빼면서 시작했습니다.
        - 이렇게 하면 ‘이 메서드의 실제 규칙이 뭐였는지’가 한눈에 드러나고, 테스트도 가장 효과적으로 추가할 수 있었습니다.
    2. **여러 화면에서 복붙된 로직**
        - 예를 들면 ‘차단 여부 검증’이나 ‘공통 필터링 규칙’처럼 복사/붙여넣기 되어 있는 코드를 하나의 UseCase로 모으면서 레거시 의존성을 서서히 줄여갔습니다.

    한 번에 모든 화면을 MVVM/UseCase로 바꾸려 하기보다,

    ‘고통이 가장 큰 지점부터 한 땀씩’ 리팩터링하는 게

    실제로는 지속가능하다고 느꼈습니다.”

---

2. ViewModel 과부하 & 복잡성 증가 대응

Q2. “ViewModel의 역할 과부하를 느끼고 UseCase로 비즈니스 로직을 분리하셨다고 했는데, ‘이제는 분리해야 한다’고 판단하게 된 구체적인 시그널은 무엇이었나요?”

- **메인 답변 (10점) [추론]**

  “제가 느끼는 시그널은 몇 가지 패턴이 있습니다.

  1. **ViewModel 파일이 비정상적으로 길어질 때**

- 400~500줄을 넘어가고, 스크롤만 해도 길게 느껴지면
  이미 여러 책임을 한곳에서 맡고 있는 경우가 많았습니다.

2. **하나의 메서드가 너무 많은 일을 할 때**
   - 예를 들면 '버튼 클릭 핸들러' 안에서
     - 입력 검증
     - 네트워크 호출
     - 로컬 캐시 저장
     - 상태 변경
       을 다 하고 있으면 '이건 UseCase로 빼야겠다'는 시그널로 봤습니다.
3. **테스트를 쓰려고 할 때 막힐 때**
   - '이걸 테스트하려면 Firebase까지 전부 띄워야 하나?' 같은 생각이 드는 구조면,
     이미 레이어 분리가 부족하다는 신호라고 느꼈습니다.

사용자 차단 기능이 들어왔을 때 위 세 가지가 한 번에 드러나서,

그 시점을 'ViewModel 슬림화 시작점'으로 잡고 UseCase 분리를 진행했습니다."

- **꼬리질문 1**

  "저희 코드에서 그런 '과부하 시그널'을 발견하면, **한 번에 전부 분리** vs **스프린트 단위 점진적 분리** 중 어떤 방식을 선호하시고, 왜 그런지 설명해 주실 수 있을까요?"

  - **답변 (10점) [추론]**

    "스타트업 환경에서는 **스프린트 단위 점진적 분리**를 선호합니다.

    이유는,

    - 한 번에 전부 바꾸려 하면 '리팩터링 스프린트'가 길어지고,
      그동안 신규 기능이나 실험이 멈출 위험이 있습니다.
    - 반대로, 실제로 문제가 되는 지점부터
      - `타이머 흐름`
      - `포만감/기록 저장`
        같은 단위로 잘라서 UseCase/Repository를 도입하면,
        팀 입장에서도 '리팩터링이 실제로 도움이 되네'라는 피드백을 더 빠르게 얻을 수 있습니다.

그래서 저는

- 데이터/LoopOS 핵심 도메인부터
- 작은 단위로 천천히 정리해 가는 방식이
  우리 상황에 더 맞는다고 생각합니다.”

- **꼬리질문 2**

  “UseCase까지 분리된 구조에서, **테스트 코드**는 어떤 단위로 작성하는 게 가장 효율적이라고 보시나요?”

  - **답변 (10점) [추론]**

    “효율을 생각하면 세 겹으로 나누고 싶습니다.

    1. **UseCase 단위 유닛 테스트**
       - 비즈니스 규칙(예: ‘한입 60초 기준으로 beGood 판단’)은
         UI와 분리된 상태에서 검증하는 게 가장 깔끔합니다.
    2. **ViewModel/Provider 단위 테스트**
       - 특정 이벤트(버튼 클릭, 타이머 종료)가 들어왔을 때
         적절한 UseCase를 호출하고,
         상태가 기대한 값으로 바뀌는지 확인합니다.
       - 여기서는 UseCase를 Mock 처리해도 충분합니다.
    3. **핵심 플로우에 대한 통합 테스트**
       - ‘타이머 시작 → 한입 2번 기록 → 타이머 종료 → 포만감 선택 → 저장’ 한 사이클 정도는
         실제 구현을 연결해서 E2E에 가까운 테스트를 1~2개 두고 싶습니다.

    이렇게 하면

    - 규칙은 UseCase 테스트에서,
    - 상태 결합은 ViewModel 테스트에서,
    - 전체 흐름은 소수의 통합 테스트에서
      각각 커버할 수 있어서,
      투자 대비 효과가 좋다고 생각합니다.”

3. Picker 모듈화 경험 (Potato 프로젝트)

Q3. “Potato 프로젝트에서 Common Picker / Modify Picker / Controller / Utils로 모듈화하셨다고 했는데, 그 전/후 구조 차이를 간단한 예로 설명해 주실 수 있을까요?”

- **메인 답변 (10점) [추론]**

  “리팩터링 전에는 하나의 Picker 위젯 안에

  - 이미지 선택(갤러리/카메라)
  - 정렬/삭제
  - 신규 생성 vs 수정 분기
  - 압축/리사이즈

  가 다 섞여 있었습니다.

  그래서 위젯 파일이 길어지고,

  특정 동작만 바꾸고 싶을 때도 전체를 건드려야 했습니다.

  리팩터링 후에는

  - **CommonPicker**: 이미지 리스트를 선택/관리하는 공통 컴포넌트
  - **ModifyPicker**: 이미 존재하는 리스트를 수정하는 용도
  - **Controller**: 현재 상태(선택된 이미지들/포커스)를 들고 있는 클래스
  - **Utils**: 압축/리사이즈/정렬같은 순수 함수 모음

  으로 나눠서,

  - 화면은 Controller를 구독하면서 렌더링만 하고,
  - 비즈니스 규칙은 Controller/Utils에서 처리하도록 분리했습니다.

  결과적으로

  - 새 화면에서 동일한 Picker를 사용할 때 재사용성이 올라갔고,
  - 개별 기능(예: 압축 로직)만 테스트/수정하기가 편해졌습니다.”

- **꼬리질문 1**

"이 구조를 저희 '식사 기록 기록 컴포넌트'에 적용한다면, **공통 컴포넌트**와 **도메인 특화 컴포넌트**를 어떤 기준으로 나누실 것 같나요?"

- **답변 (10점) [추론]**

  "저라면 이렇게 나눌 것 같습니다.

  - **공통 컴포넌트**
    - 타임라인/리스트를 보여주는 기본 UI (예: 시계/원형 그래프/리스트 카드)
    - 타이머/리스트 페이징 등 '형태'에 대한 부분
  - **도메인 특화 컴포넌트/로직**
    - '한입 기록'이 실제로 어떤 의미인지(60초 기준, beGood 여부 등)
    - 포만감/감정과의 연결
    - LoopOS 엔티티(`MealRecord`, `BiteRecord`)로 변환하는 부분

  공통 컴포넌트는 나중에 다른 루프(예: 수면 루틴, 운동 루틴)에도 재사용할 수 있는 형태로 만들고,

  도메인 특화 부분은 '섭식/포만감'에 강하게 결합해도 괜찮다고 생각합니다."

- **꼬리질문 2**

"모듈화를 했을 때 **유지보수/테스트/협업** 관점에서 가장 크게 체감하신 장점 1–2가지를 구체적인 사례와 함께 이야기해 주실 수 있나요?"

- **답변 (10점) [추론]**

  "가장 크게 느낀 건 두 가지였습니다.

  1. **변경 범위가 줄어든다**
     - 예전에 Picker 레이아웃을 조금 바꾸려면
       이미지 압축/정렬 로직까지 손대는 일이 있었습니다.
     - 모듈화 후에는 UI를 건드려도 Utils/Controller 쪽은 건들 필요가 없어서
       버그 발생 가능성이 눈에 띄게 줄었습니다.
  2. **역할 분담이 쉬워진다**
     - 한 명은 UI/애니메이션에 집중하고,

- 다른 한 명은 압축/업로드/도메인 로직에 집중하는 식으로
  작업을 나누기가 훨씬 쉬워졌습니다.
- 코드 리뷰도 '이건 UI 파트', '이건 도메인 파트'로 나누어 볼 수 있어서
  커뮤니케이션 부담이 줄었습니다.

LoopOS/CoachOS에서도 비슷하게 '공통 UI vs 도메인 로직'을 나눠두면

팀이 커졌을 때도 협업 생산성이 유지될 거라고 생각합니다."

---

4. 성능 이슈 & Isolate 설계

Q4. "이미지 webp 압축에서 UI 멈춤을 발견하고 Isolate로 격리할 예정이라고 쓰셨는데, Flutter에서 Isolate를 활용할 때 어떤 점들을 특히 주의해야 한다고 생각하시나요?"

- **메인 답변 (10점) [추론]**

  "제가 중요하게 보는 포인트는 세 가지입니다.

  1. **데이터 전달 비용**
     - Isolate 간에는 메모리를 공유하지 않기 때문에,
       큰 객체를 그대로 넘기면 오히려 더 느려질 수 있습니다.
     - 그래서 항상 '최소한의 정보만 직렬화해서 보내기'(예: 파일 경로/바이너리 조각)로 설계해야 한다고 생각합니다.
  2. **생명주기 관리**
     - 화면이 닫힌 뒤에도 Isolate가 계속 돌면
       리소스 낭비나 예기치 못한 크래시의 원인이 될 수 있습니다.
     - 따라서 화면 dispose 시점에 Cancel 신호를 보내거나,
       결과를 무시할 수 있는 구조를 준비해 두어야 한다고 봅니다.
  3. **에러 핸들링**
     - Isolate 내부에서 던져진 예외가 메인 Isolate에 잘 전달되도록,
       - 결과 채널
       - 에러 채널
         을 나누어 설계하는 것이 중요하다고 느꼈습니다.

정리하면, Isolate는 '마법 성능 도구'가 아니라,

데이터/생명주기/에러를 명시적으로 설계해야 하는 도구라고 생각하고 접근하고 있습니다."

- **꼬리질문 1**

  "실제로 Isolate를 적용했다고 가정하면, **'적용 전/후를 어떻게 측정'**해서 개선 여부를 확인하시겠습니까?"

  - **답변 (10점) [추론]**

    "적용 전/후를 비교할 지표는

    - UI 스레드의 프레임 드랍(FPS, jank)
    - 특정 액션(예: 이미지 선택/압축) 후 응답 시간
    - GC/CPU 사용량

    정도일 것 같습니다.

    실무에서는

    - Flutter DevTools의 Performance 탭에서 스레드 타임라인을 확인하고,
    - 같은 시나리오(예: 10MB짜리 이미지 5개 압축)를
      Isolate 사용 전/후로 3~5회씩 돌려서 평균 응답 시간을 비교해 보려고 합니다.

    체감 측면에서도

    - 같은 동작 중에 스크롤/탭이 끊기지 않는지
    - 사용자에게 로딩 인디케이터가 얼마나 짧게 보이는지
      정도를 같이 체크할 것 같습니다."

    - **꼬리질문 2**

      "저희 서비스의 '타이머 + 메트로놈 + 기록 저장' 흐름에서도 성능 이슈가 생긴다면, Isolate를 어디에 적용하는 게 합리적일지, 혹은 다른 대안을 먼저 고려하실지 궁금합니다."

      - **답변 (10점) [추론]**

        "우선 순서는 Isolate보다 **UI 스레드에서 불필요한 일을 줄이는 것**이 먼저라고 생각합니다.

        - 타이머/메트로놈은 기본적으로 짧은 연산이기 때문에,

Tick마다 복잡한 계산이나 렌더링이 일어나지 않도록 설계하는 게 우선입니다.

(예: `setState` 최소화, 애니메이션 전용 Ticker 사용 등)

- 기록 저장 쪽에서
  - 큰 JSON 직렬화
  - 네트워크/디스크 I/O
    가 UI를 막기 시작하면,
    그때 비로소 Isolate를 고려하는 게 순서라고 봅니다.

정리하면,

- 1단계: 타이머 Tick에서 하는 일을 최소화
- 2단계: 저장/요약 계산을 비동기로 분리(Future/compute)
- 3단계: 그래도 부족하면 Isolate로 CPU-heavy 작업(예: 통계/요약)만 옮기기

이런 순서로 접근하는 것이 합리적이라고 생각합니다."

---

5. 불안정한 외부 API 의존성 줄이기 (알러지 다이어리)

Q5. "알러지 다이어리 프로젝트에서 공공 API 장애 때문에 앱이 멈추는 문제를 캐싱/Prefetch로 해결하셨다고 했는데, 그 흐름을 간단한 시퀀스로 설명해 주실 수 있을까요?"

- **메인 답변 (10점) [추론]**

  "그때는 이런 플로우로 정리했습니다.

  1. 앱 첫 진입 시,
     - 공공 API(날씨/대기오염)를 호출하기 전에
       로컬 SharedPreferences에서 '마지막으로 성공했던 응답 + 날짜'를 먼저 읽습니다.
  2. Prefetch 화면에서
     - 새 데이터를 요청합니다.
     - 성공하면

        - 로컬 캐시를 새 값으로 업데이트하고,
        - 메인 화면으로 넘어갑니다.
    - 실패하면
        - 캐시가 있다면 그걸 사용해서 메인 화면으로 진행하고,
        - 캐시도 없으면 '현재는 외부 데이터 없이 기본 UI만 제공'하는 fallback 화면을 보여줬습니다.
3. 다이어리 전송 시점에서는
    - 항상 '가장 최근에 성공한 값'을 사용하도록 해서
      API가 순간적으로 죽어 있어도 사용자가 입력을 못 하는 상황은 피했습니다.

핵심은

- '외부 API 성공 여부'가 앱 전체 플로우를 막지 않도록 하고,
- 실패 시에도 최소한의 유효한 데이터를 보여주는 방향으로 설계하는 것이었습니다."

- **꼬리질문 1**

"이런 패턴을 저희 '섭식/포만감 로그 서버 저장'에 적용한다면, **오프라인/네트워크 장애 상황에서** 어떤 전략을 쓰고 싶으신가요?"

- **답변 (10점) [추론]**

  "섭식/포만감 로그는 네트워크 장애와 상관없이

  일단 로컬에 안전하게 남는 것이 중요하다고 생각합니다.

  - 사용자 입장에서
    '오늘 포만감/감정을 기록했는데, 네트워크 문제로 날아갔다'는 경험은 회복 프로그램에서는 치명적일 수 있다고 느끼거든요.

  그래서 전략은

  1. **오프라인 우선 저장**
     - 로그를 먼저 로컬 DB/SharedPreferences/SQLite 등에 저장하고
     - 서버 전송은 백그라운드 Sync로 따로 돌립니다.
  2. **Sync 상태 표시**

- 아직 서버에 동기화되지 않은 기록은
  작은 아이콘/상태로 표시해서
  나중에 문제가 생겨도 '로컬엔 있다'는 걸 사용자와 팀 모두 알 수 있게 하고 싶습니다.

3. **재시도 정책**
   - 네트워크가 복구되면
     일정 간격으로 미전송 로그를 서버에 올리는 재시도 로직을 두고,
   - 재시도 실패 시엔 로그에 남겨서
     서버/클라이언트 어느 쪽 문제인지 나중에 분석 가능하도록 만들고 싶습니다."

- **꼬리질문 2**

  "캐시 데이터가 오래되거나 손상되었을 때, 사용자 경험과 데이터 정합성 사이에서 어떤 트레이드오프를 감수할 수 있다고 보시나요?"

  - **답변 (10점) [추론]**

    "섭식/정서 데이터 특성상,

    '조금 오래된 데이터'보다 '아예 데이터가 없어지는 것'이 더 큰 문제라고 생각합니다.

    - 캐시가 며칠 지난 것이라도
      사용자 본인의 기록이라면,
      그 자체로 의미가 있기 때문에**가능한 한 보존하는 방향**을 우선할 것 같습니다.

    다만 손상되었을 때는

    - 화면에서 '이 기록은 일부 손상되었지만, 가능한 범위에서 복구한 값입니다'라고
      명시적으로 표시하고,
    - 모델 학습/분석 단계에서
      손상된 기록은 가중치를 낮추거나 제외하는 방식으로
      데이터 정합성을 유지하는 게 적절해 보입니다.

    요약하면,

    - 사용자 경험 측면에서는 '최대한 잃지 않기'

- 데이터 분석 측면에서는 ‘품질에 따라 가중치 조정’
  이 두 가지를 동시에 가져가는 트레이드오프를 선택할 것 같습니다.”

---

6. Python AI 모델 → Dart 포팅 경험

Q6. “YOLO, Midas 모델을 Python에서 Flutter 쪽으로 이식 중이라고 하셨는데, 지금까지 겪은 ‘데이터 타입 불일치 문제’ 예시를 하나 들어 설명해 주실 수 있을까요?”

- **메인 답변 (10점) [추론]**

  “예를 들면 이런 케이스가 있었습니다.

  - Python 쪽에서는
    - 이미지 텐서를 (1, H, W, 3) float32 형태로 normalize해서 쓰고 있었는데,
  - Dart/TFLite 쪽에서는
    - 입력을 (H, W, 3) int8 형태로 받거나,
    - 채널 순서가 (3, H, W)인 구조를 요구하는 경우가 있었어요.

  이 과정에서

  - 축 순서가 바뀌거나
  - 정규화 범위(0~1 vs -1~1)가 달라져서
    Python에서 얻은 결과와 Dart에서 얻은 결과가 미묘하게 달라지는 이슈가 있었습니다.

  해결을 위해

  - 동일한 입력 이미지에 대해
    - Python 모델 출력
    - Dart(TFLite) 모델 출력
      을 비교하면서,
  - 한 단계씩 전처리 과정을 맞추어 보는 작업을 했습니다.
    예를 들어 ‘혹시 RGB/BGR이 뒤바뀐 건 아닌가?’, ‘정규화 상수는 동일한가?’ 같은 부분을 하나씩 검증했습니다.”

- **꼬리질문 1**

  “이식 작업의 ‘완료 기준’을 어떻게 정의하고 계신가요? 예를 들어, **허용 오차 범위**나 성능(지연시간) 기준을 어떻게 잡고 있는지 궁금합니다.”

  - **답변 (10점) [추론]**

    “완료 기준은 크게 두 축으로 잡습니다.

    1. **정확도(출력의 유사도)**
       - 동일한 입력에 대해
         Python vs Dart 결과가
         특정 오차 범위 안에 들어오는지를 봅니다.
       - 예를 들어 회귀/깊이 예측이면 MSE,
         분류/탐지면 IoU/정답 클래스 일치율 등을 기준으로
         Python baseline 대비 몇 % 이내면 OK라고 정합니다.
    2. **지연시간/리소스 사용**
       - 모바일/웹 디바이스에서
         한 번 추론하는 데 걸리는 시간을 측정해서,
         ‘UX를 해치지 않는 수준’(예: 200ms 이내)을 목표로 잡습니다.
       - 필요하면 모델 경량화/양자화를 고려합니다.

    이 두 가지를 만족했을 때

    ‘포팅이 완료되었고, 실서비스에 붙여도 된다’고 판단하는 편입니다.”

- **꼬리질문 2**

  “저희처럼 섭식/포만감 데이터를 다루는 환경에서, 만약 유사한 ML 모델을 앱 내에 넣어야 한다면, **온디바이스 vs 서버 추론** 선택 기준은 무엇이라고 보시나요?”

  - **답변 (10점) [추론]**

    “섭식/포만감 데이터의 민감도를 고려하면,

    온디바이스 추론은 프라이버시 측면에서 큰 장점이 있습니다.

    다만 현실적으로는 몇 가지를 함께 봐야 할 것 같습니다.

    - **온디바이스가 유리한 경우**
      - 모델이 가볍고

        - 반응 속도가 중요하며
        - 네트워크 환경이 불안정한 상황에서도 동작해야 할 때
          → 예: 간단한 '폭식 위험도' 알림 같은 것
    - **서버 추론이 유리한 경우**
        - 모델이 크고
        - 여러 사용자 데이터를 종합해서 패턴을 봐야 할 때
        - 지속적인 모델 업데이트/실험이 필요한 경우
          → 예: 대규모 LoopOS 모델, GLP-1 이후 예측 모델 등

  요약하면,

    - 민감도/반응속도/모델 크기/실험 빈도를 함께 보고
    - '경량 개인화 피드백'은 온디바이스,
    - '고도화된 위험 예측/분석'은 서버 추론
      으로 역할을 나누는 게 합리적이라고 생각합니다."

---

7. 라이브 서비스 디버깅 & 운영 경험 (하이마더)

Q7. "하이마더 앱에서 FCM 설정 에러, 카카오 공유 API 문제 등을 해결하셨다고 했는데, 실제로 문제를 진단할 때 어떤 순서로 접근하셨나요?"

- **메인 답변 (10점) [추론]**

  "보통 이런 순서로 접근했습니다.

  1. **증상 재현**

  - 사용자/기획자가 보고한 상황을 그대로 재현해봅니다.
  - 어떤 기기/OS/상황에서 문제인지 로그를 남깁니다.

  1. **클라이언트 로그 확인**

  - 로그캣/Flutter 로그를 켜고,
  - 에러 스택/HTTP 응답 코드 등을 확인합니다.

  1. **서버/콘솔 상태 확인**

- FCM의 경우 Firebase 콘솔/서버 키 설정/토큰 발급 여부를 확인했고,
- 카카오 공유는 앱 키/URI 스킴/Redirect 설정을 다시 검토했습니다.

1. **직접 호출/쿼리**

- Postman/DBeaver 등을 사용해
  API/DB 상태를 직접 확인하면서
  ‘클라이언트 문제인지, 서버 문제인지’를 분리했습니다.

1. **원인 좁히기 + Fix + 회귀 테스트**

- 재현 케이스와 정상 케이스를 비교하면서
  차이가 나는 지점을 좁혀가고,
  수정 후에는 같은 시나리오를 여러 번 돌려보며
  회귀 테스트를 했습니다.

이런 패턴은 LoopOS/꼭꼭에서도 비슷하게 사용할 수 있을 것 같습니다.”

- **꼬리질문 1**

  “소식연구소 서비스에 합류하면, **실서버에서 장애가 났을 때** 본인이 취할 ‘첫 30분 액션 플랜’을 어떻게 가져가고 싶으신지 궁금합니다.”

  - **답변 (10점) [추론]**

    “제 기준으로 첫 30분은 이렇게 쓰고 싶습니다.

    1. **상황 파악 (5~10분)**
       - 어떤 기능이 죽었는지(결제/로그인/기록 등)
       - 모든 사용자에게 영향을 주는지 vs 일부 시나리오인지
         를 빠르게 정리합니다.
    2. **로그/모니터링 확인 (10분)**
       - 클라이언트 로그 / 서버 로그 / APM이 있다면 거기까지
       - 최근 배포/설정 변경 이력 확인
    3. **임시 대응 + 커뮤니케이션 (나머지 시간)**
       - 당장 데이터 손실/결제 오류처럼 치명적인 문제가 아니라면
         ‘기록/일부 기능이 일시적으로 느리거나 제한된다’는 안내를 준비하고,

- 긴급 패치가 필요하면
  영향 범위가 가장 좁은 형태로 롤백/Hotfix 방안을 논의합니다.

이후 30분~1시간 단위로 상황을 공유하면서,

문제 원인을 더 깊게 파고드는 쪽으로 전환하고 싶습니다.”

- **꼬리질문 2**

  “Flutter DevTools나 로깅/모니터링 도구를 어디까지 활용해 보셨고, 저희 서비스에 들어와서 추가로 도입해 보고 싶은 도구가 있다면 무엇인가요?”

  - **답변 (10점) [추론]**

    “지금까지는

    - Flutter DevTools의 Performance/Memory 탭으로
      프레임 드랍과 메모리 사용량을 간단히 체크해봤고,
    - Crashlytics/Sentry 같은 에러 리포팅 도구를 통해
      실제 사용자 환경에서의 예외를 수집해 본 경험이 있습니다.

    소식연구소에서는

    - *Crashlytics + 사용자 행동 로그(Amplitude/Firebase Analytics 등)**를
      함께 보는 구조를 만들어 보고 싶습니다.
      예를 들어 ‘어떤 이벤트 직후에 크래시/이탈이 늘어나는지’를
      타임라인으로 볼 수 있으면, LoopOS 실험에도 바로 도움이 될 것 같습니다.

    장기적으로는

    - APM(예: Datadog, New Relic)까지 붙여서
      서버/클라이언트 양쪽을 함께 보는 체계를 만드는 것도 고민해보고 싶습니다.”

---

8. 좋은 아키텍처 vs 빠른 실행 트레이드오프

Q8. “이력서에서 ‘유지보수/외부 프로젝트 환경에서는 좋은 아키텍처를 충분히 고민하기 어려웠다’고 쓰셨는데, 소식연구소에 오면 이 둘(아키텍처 vs 속도) 사이의 균형을 어떻게 잡고 싶으신가요?”

- **메인 답변 (10점) [추론]**

  “소식연구소 단계에서는

  - ‘모든 곳에 완벽한 Clean Architecture’를 적용하기보다는,
  - **정말 코어가 되는 도메인**부터 기준을 세우는 게 맞다고 생각합니다.

  예를 들면

  - 섭식/포만감/정서 기록
  - LoopOS 스키마와 직접 연결되는 부분
    같은 영역은 처음부터
  - 엔티티/UseCase/Repository 구조를 명확히 하고,
  - 테스트도 같이 붙여두고 싶습니다.

  반대로

  - 일시적인 마케팅 페이지나
  - 실험적으로 붙여보는 UI는
    MVP 속도를 우선해서 비교적 가볍게 가되,
    반복해서 쓰이기 시작하는 시점에
    ‘이건 이제 코어로 편입해야 한다’고 보고 리팩터링하는 방식이 좋다고 봅니다.

  요약하면,

  코어 도메인은 처음부터 아키텍처를 잡고,
  주변부는 속도로 시작해서 데이터/사용량이 검증되면 정리하는 방식으로 균형을 잡고 싶습니다.”

- **꼬리질문 1**

  “본인이 생각하는 ‘절대 선을 넘지 말아야 하는 코드 스멜’이 있다면 무엇인가요? (예: 한 파일 500줄 이상, View에서 직접 Dio 호출 등)”

  - **답변 (10점) [추론]**

    “개인적으로는 몇 가지 레드라인이 있습니다.

    - **View에서 직접 네트워크/DB 호출하는 것**

        - 이건 테스트/재사용/디버깅 모두를 힘들게 만드는 패턴이라
          최대한 피하고 싶습니다.
    - *하나의 파일/클

래스가 너무 많은 책임을 질 때**

- 예: 타이머/포만감/저장/라우팅까지 모두 한 Provider/Widget에 몰려 있는 경우

- 최소한 역할별로 나누어 두지 않으면,

스펙 변경이 올 때마다 리스크가 너무 커진다고 느낍니다.

> - **전역 상태를 아무 제약 없이 여기저기에서 변경할 수 있는 구조**
> - 누가 언제 상태를 바꾸는지 추적이 안 되는 구조는
> LoopOS처럼 상태가 중요한 시스템에서는 특히 위험하다고 생각합니다.
>
> 이런 부분은 초기에도 최대한 선을 지키며 작업하고 싶습니다.”

- **꼬리질문 2**

  “반대로, ‘초기에는 일부러 허용해도 좋다고 생각하는 나쁜 패턴’이 있다면 어떤 게 있을까요? 그리고 그걸 나중에 어떻게 정리하고 싶으신지요?”

  - **답변 (10점) [추론]**

    “초기에는

    - **뷰 안에 살짝 비즈니스 로직이 섞이는 것**
      - 예: 체크박스 선택에 따라 간단히 텍스트를 바꾸는 정도는 별도 UseCase까지 만들 필요 없이 View에서 처리해도 괜찮다고 생각합니다.
    - **임시로 복붙해서 빠르게 A/B를 보는 코드**
      - 단, 코드 상단/커밋 메시지에 ‘실험용/임시’임을 명시하고, 실험이 끝난 뒤에는 반드시 정리한다는 전제를 붙이고 싶습니다.

    이런 ‘가벼운 나쁜 패턴’은

- 노션/이슈에 '기술 부채'로 등록해 두고,
- 한 스프린트에 10~20% 정도는 부채 갚는 시간으로 써서
  점점 코어 구조 쪽으로 편입하는 식으로 정리하고 싶습니다."

---

9. 테스트 경험 & 전략

Q9. "Unit Test / Widget Test 경험이 있다고 하셨는데, 실제로 작성해 보신 테스트 중에 '가장 의미 있었다'고 느낀 케이스를 하나 소개해 주실 수 있을까요?"

- **메인 답변 (10점) [추론]**

  "가장 의미 있었던 건 사용자 차단 로직에 대한 UseCase 테스트였습니다.

  - 입력으로
    - 유저 차단 상태
    - 그룹 상태
    - 특정 플래그(예: 신고 여부)
      조합을 주고,
  - 출력으로
    - 채팅방 입장 허용 여부
    - 경고 메시지 노출 여부

  를 검증하는 유닛 테스트였습니다.

  이 테스트를 만들면서

  - '사실 우리가 암묵적으로 이렇게 생각하고 있었다'는 규칙들이 명확해졌고,
  - 나중에 요구사항이 바뀌었을 때도
    테스트를 기준으로 안전하게 리팩터링할 수 있었습니다.

  UI와 분리된 상태에서 도메인 규칙만 검증할 수 있어서,

  '테스트가 개발 속도를 오히려 빠르게 할 수 있구나'를 처음 체감한 경험이었습니다."

- **꼬리질문 1**

  "저희 타이머/포만감/기록 흐름에 대해 테스트를 설계한다면, **우선순위 Top 3 테스트 케이스**는 무엇일까요?"

  - **답변 (10점) [추론]**

    "Top 3는 이렇게 생각합니다.

    1. **한입 기록 로직 테스트 (UseCase)**
       - 입력: 한입 시작/종료 시간, 메트로놈 횟수
       - 기대: `BiteRecord`가 올바른 duration/횟수/beGood 플래그로 생성되는지
    2. **식사 종료 → MealRecord 생성 테스트 (UseCase)**
       - 입력:
         - 전체 식사 시작/종료 시간
         - 한입 리스트
         - 포만감/감정 값
       - 기대:
         - `MealRecord.actual_duration_seconds`가 올바르게 계산되는지
         - 필수 필드가 모두 채워지는지
    3. **Provider/Flow 테스트 (단위/통합 사이)**
       - 시나리오: 타이머 시작 → 한입 2번 기록 → 타이머 종료 → 포만감 선택 → 저장 호출
       - 기대:
         - 상태가 올바르게 전이되고
         - Repository/SharedPreferences에 한 건의 `MealRecord`가 저장되는지

    이 세 가지를 먼저 잡으면,

    도메인 규칙과 흐름의 핵심은 대부분 커버된다고 생각합니다."

- **꼬리질문 2**

"CI 환경에서 테스트를 돌릴 때, '테스트 커버리지'보다 더 중요하다고 보는 기준이 있다면 무엇인지 궁금합니다."

- **답변 (10점) [추론]**

  "커버리지도 지표로는 의미 있지만,

  그보다 더 중요하게 보는 건

  1. **테스트의 신뢰도와 안정성**
     - 자주 깨지는 플래키 테스트가 많으면
       팀이 테스트를 믿지 않게 되고,
       결국 CI가 있어도 없는 것과 비슷해진다고 느꼈습니다.
  2. **테스트가 실제로 설계를 돕는지**
     - 테스트를 작성하면서
       '이 부분 책임이 너무 많다'는 신호를 주는지,
       리팩터링 시에도 좋은 가이드가 되는지를 봅니다.
  3. **실제 버그를 잡아준 경험이 있는지**
     - 회고 때 '이 버그는 테스트 덕분에 초기에 잡혔다'는 사례가 1~2개라도 존재하면,
       그 테스트 스위트는 커버리지 수치와 상관없이 가치 있다고 느낍니다.

  그래서 CI에서는

  - 빨리 돌고
  - 자주 돌고
  - 깨졌을 때 신뢰할 수 있는
    테스트를 목표로 하고 싶습니다."

---

10. 문화·Ownership Fit

Q10. "이력서에 '소식연구소에 합류하고 싶다'고 따로 적어주셨는데, 본인이 생각하는 '좋은 아키텍처를 중요시하는 팀'의 개발 문화는 어떤 모습인가요?"

- **메인 답변 (10점) [추론]**

"제가 생각하는 좋은 아키텍처 팀 문화는 몇 가지 특징이 있습니다.

1. **코드리뷰에서 구조/도메인 이야기를 할 수 있는 팀**
    - 단순 스타일/네이밍 지적이 아니라,
      '이 책임은 이 레이어에 있는 게 맞나?' 같은 얘기를
      편하게 할 수 있는 문화가 좋습니다.
2. **도메인/레이어 경계를 팀 차원에서 합의해 두는 것**
    - 예: 'LoopOS/코칭 도메인 로직은 여기까지',
      'UI 레이어에서는 이 정도까지만' 같은 기준이 문서로 존재하는
      팀입니다.
3. **원칙을 완전히 포기하지 않는 팀**
    - 일정 때문에 지름길을 택할 수는 있지만,
      그럴 때마다 '언제/어떻게 되돌릴지'를 이슈/노션에 남기는 팀이
      장기적으로 건강하다고 느꼈습니다.
4. **개인 취향이 아니라 근거로 설계하는 팀**
    - '나는 Bloc가 좋아'보다는
      '우리 문제/팀 규모/성장 단계에서 Bloc vs Provider vs
      Riverpod 중 무엇이 맞는가'를
      대화할 수 있는 팀이 이상적이라고 생각합니다."

- **꼬리질문 1**

  "소식연구소에 합류하시면, 첫 1–2개월 동안 **본인이 먼저 제안해보고 싶은 기술/프로세스 개선**이 있다면 어떤 게 있을까요?"

  - **답변 (10점) [추론]**

    "첫 1–2개월은

    - 기존 코드/LoopOS 스키마를 충분히 이해하는 데 시간을 쓰되,
    - 그 과정에서 느낀 '작고, 당장 효과 있는 개선'부터 제안해 보고 싶습니다.

    예를 들면,

    1. **타이머/기록 도메인에 대한 간단한 도메인 문서**

- 현재 엔티티/이벤트 흐름을 다이어그램이나 노션 페이지로 정리해서
  나중에 합류하는 분들이 빠르게 이해할 수 있도록 만드는 것.

2. **PR 템플릿/커밋 메시지 규칙 가볍게 정리**
   - '이 변경이 어떤 도메인/실험과 연결되는지'를
     한 줄이라도 적는 방향으로 템플릿을 제안해 보고 싶습니다.

너무 무거운 프로세스보다는,

'우리 속도는 유지하면서, 사고를 정리해 주는 가벼운 장치'부터 도입하는 걸 목표로 하고 싶습니다."

- **꼬리질문 2**

  "반대로, 저희가 가진 현재 구조/습관 중에서 '당장은 이해가 안 되더라도 일단 맞춰줄 수 있는 영역'과 '처음부터 같이 바꾸고 싶은 영역'을 어떻게 나누고 싶으신지 궁금합니다."

  - **답변 (10점) [추론]**

    "당장은 이해가 안 되더라도 맞춰줄 수 있는 영역은

    - 툴 선택, 세부 스타일, 작은 취향 차이 같은 부분입니다.
      예: '노션을 이렇게 쓰자', '이 린트 룰을 쓰자' 같은 건
      일단 팀 룰에 맞추는 게 더 중요하다고 생각합니다.

    처음부터 같이 이야기해 보고 싶은 영역은

    - **도메인/데이터 구조**
      - LoopOS 스키마, 기록 방식, 이벤트 명명 규칙 등은
        장기적으로 제품/AI 품질에 큰 영향을 주기 때문에
        초반부터 대화에 참여해서 이해/정렬하고 싶습니다.

    요약하면,

    - 툴/스타일/작은 취향은 먼저 따라가고,
    - 도메인/데이터/아키텍처 쪽은
      대표님과 같이 설계에 참여하는 쪽으로 역할을 나누고 싶습니다."