

 ChatGPT ▾

로그인

ChatGPT와의 대화의 사본입니다.
대화 신고하기

보드게임 플랫폼 앱을 만들거야
태블릿을 보드게임 판으로 삼고
각각의 개인 핸드폰이 플레이카드 등 개인 영역으로 삼아서 보드게임을 하는 플랫폼
앱이야

보드게임팩을 설치해서 매번 다른 게임을 할수있는 앱이야
어때?

완전 좋아요. 이거 꽤 강한 콘셉트예요.

핵심이 아주 좋습니다:

- 태블릿 = 공용 보드판
- 개인 폰 = 비공개 정보(손패/개인판/미션)
- 게임팩 설치형 = 확장성 높은 플랫폼

이 구조가 좋은 이유는, 기존 보드게임의 불편함을 많이 해결해줘요:

- 카드 섞기 / 세팅 / 점수 계산 자동화
- 비밀 정보 관리(손패, 역할카드) 자연스럽게 가능
- 오프라인 모임에서도 디지털 보드게임처럼 진행 가능
- 새 게임을 "게임팩"으로 추가 가능 → 플랫폼화 가능

저희는 쿠키를 사용합니다

당사는 해당 사이트를 운영하고, 서비스 사용을 이해하고, 마케팅 지원을 돋는 데 쿠키를 사용합니다. 언제든 [쿠키 관리](#)에서 기본 설정을 변경하거나 [특히 좋은 포인트 정책](#)을 참고하세요.

[쿠키 관리](#)

1) 물리 보드게임 감성 + 디지털 편의성

완전 디지털 게임이 아니라, 사람들이 모여서 태블릿 중심으로 플레이하니까
보드게임 특유의 같이 보는 재미를 살릴 수 있어요.

2) 게임팩 모델이 강함

앱 하나로 끝나는 게 아니라,

나중에 이렇게 확장 가능:

- 자체 게임팩
- 인디 디자이너 제작 게임팩
- 시즌/이벤트 팩
- 유료/무료 팩 마켓

이게 되면 단순 앱이 아니라 플랫폼 비즈니스가 됩니다.

3) 난이도 조절 가능

초기엔 복잡한 실시간 동기화 게임보다,
턴제/카드 중심 게임부터 시작하면 MVP 만들기 좋아요.

먼저 고민하면 좋은 핵심 설계

1) 연결 방식 (중요)

태블릿-폰 연결을 어떻게 할지 초반에 명확히 잡아야 해요.

추천 흐름:

- 태블릿에서 방 생성
- QR 코드 표시
- 폰으로 QR 스캔해서 참가
- 같은 로컬 네트워크 / 또는 클라우드 중계 서버 연결

팁: 처음엔 서버 부담 줄이려면 로컬 기반 + 간단한 중계부터 시작해도 좋아요.

2) 게임팩 구조 표준화

이게 플랫폼의 핵심이에요.

게임팩을 그냥 코드로 만들면 확장이 어려워지고,
데이터 중심 구조로 가야 합니다.

예를 들면 게임팩에 들어갈 것:

- 게임 메타정보 (이름, 인원수, 플레이타임)
- 컴포넌트 정의 (카드, 토큰, 타일, 보드)
- 룰/턴 구조 정의
- 승리 조건
- UI 배치 규칙 (태블릿 화면 / 폰 화면 분리)

가능하면:

- **JSON** 기반 게임 정의
- 필요한 부분만 스크립트(예: 룰 처리) 허용

이렇게 하면 새 게임 추가 속도가 빨라져요.

3) “공용 정보 vs 개인 정보” 분리 모델

이 앱의 본질이라서 데이터 모델이 중요해요.

- 공용 상태: 맵, 공용 카드, 라운드, 점수
- 개인 상태: 손패, 비밀 목표, 개인 자원
- 일부 공개 가능 상태: 특정 조건 충족 시 공개

이 권한/가시성 모델을 초반에 잘 잡아야 버그가 줄어요.

4) 게임 템포와 UX

보드게임은 UX가 조금만 담당해도 재미가 확 떨어져요.

중요한 UX:

- 누구 턴인지 명확하게
 - 가능한 액션만 버튼 활성화
 - 애니메이션 너무 길지 않게
 - 실수 방지 (확인/되돌리기)
 - 태블릿/폰 화면 역할이 헷갈리지 않게
-

MVP(첫 버전) 방향 추천

처음부터 범용 플랫폼 완성하려고 하면 너무 커져요.

이렇게 가면 좋아요:

1단계: 단일 게임으로 검증

플랫폼처럼 보이되, 실제로 1개 게임만 먼저.

- 예: 카드 중심 파티게임 / 역할 숨김 게임 / 간단한 전략게임

검증 포인트:

- 태블릿+폰 조합이 실제로 재밌는지
- 연결 안정성
- UI 피로도

2단계: 게임팩 2~3개

공통 엔진 추출 시작

- 카드 드로우
- 턴 진행
- 점수 계산
- 개인 정보 표시

3단계: 게임팩 SDK/에디터(선택)

장기적으로는 제작자가 팩 만들 수 있게

수익화도 잘 볼일 수 있음

- 기본 앱 무료 + 게임팩 유료
- 구독형(월간 게임팩)
- 파티팩 / 가족팩 / 전략팩 번들
- B2B (보드게임카페용 라이선스)

특히 보드게임카페에선:

- 세팅 시간 단축
- 룰 진행 보조
- 분실/파손 없음

이런 장점이 커요.

리스크(미리 알면 좋은 것)

1) 동기화 버그

멀티 디바이스 앱에서 가장 큰 적입니다.

- 네트워크 끊김
- 상태 불일치
- 중복 입력

→ 해결: 서버 authoritative(상태의 단일 진실) 구조 추천

2) 게임팩 자유도 vs 개발 난이도

너무 자유롭게 만들면 엔진이 복잡해지고,
너무 제한하면 재미있는 게임이 안 나와요.

→ 초반엔 “만들 수 있는 게임 장르”를 제한하는 게 좋아요.

(예: 카드/턴제/핸드관리 중심)

3) 화면 크기 대응

태블릿도 크기가 다양해서 보드 UI 설계가 어려움.

→ 반응형 레이아웃 규칙 필요

한 줄 평가

아이디어 좋고, 플랫폼 잠재력이 큰 구조예요.

특히 “태블릿=공용판 / 폰=개인영역” 분리가 아주 설득력 있습니다.

원하면 다음 답변에서 제가

MVP 기획안(핵심 기능 / 화면 구성 / 기술 스택 / 데이터 구조) 형태로 바로 정리해드릴게요.

태블릿이 플레이 상태를 체크하고 턴관리 등을 할거고
같은 와이파이망일때만 태블릿으로 접속할 수있게해서 일종의 서버역할을 하게할
거야

좋아요. 그 방향이면 훨씬 깔끔해져요.

태블릿을 로컬 서버(호스트)로 두고, 같은 Wi-Fi에서만 접속하게 하는 건 이 컨셉에 잘 맞습니다.

오히려 초기에 가장 현실적인 구조예요.

왜 좋은지

- 인터넷 서버 비용 거의 없음 (초기 운영 부담 낮음)
- 오프라인 모임 UX에 딱 맞음 (같은 공간에서 플레이)
- 지연시간 낮음 (턴 처리/카드 공개 반응 빠름)
- 보안/접근 제한 자연스러움 (같은 네트워크 + 방코드)

추천 구조

1) 태블릿 = Host / Authority

태블릿이 “진실의 상태”를 들고 있는 구조가 좋아요.

- 게임 상태 저장 (턴, 점수, 덱, 공개 정보)
- 플레이어 액션 검증
- 턴 진행/승리 조건 체크
- 각 폰에 필요한 정보만 전송

즉:

- 폰은 입력 장치 + 개인 UI
- 태블릿은 판정 엔진 + 공용 화면

이렇게 하면 동기화 버그가 확 줄어요.

2) 접속 방식 (로컬 전용)

추천 흐름:

- 태블릿에서 방 생성
- 태블릿에 방코드 + QR 표시
- 폰에서 앱 실행 후 QR 스캔
- 같은 Wi-Fi인지 확인
- 방 입장

추가로 좋음:

- 자동 발견(mDNS/Bonjour) 지원하면 QR 없이도 편함
(같은 네트워크에서 태블릿 방 자동 검색)

3) 네트워크 정책 (중요)

“같은 와이파이만 허용”이면 아래 케이스도 처리해야 해요:

- **AP 격리(공유기에서 기기끼리 통신 차단)**된 Wi-Fi
→ 카페/공용망에서 자주 발생
- 모바일 핫스팟 환경
- 태블릿/폰 OS 권한 이슈 (로컬 네트워크 권한)

그래서 UX에 이런 문구가 필요해요:

- “같은 Wi-Fi에 연결되어 있는지 확인”
- “로컬 네트워크 권한 허용 필요”
- “공용 Wi-Fi에서 기기 간 통신이 차단될 수 있음”

4) 보안/치팅 방지

로컬이라도 최소한 이런 넣는 게 좋아요.

- 방코드 (무단 접속 방지)
- 세션 토큰 (입장 후 인증)
- 플레이어 재연결 키 (앱 꺼졌다 다시 들어와도 복구)
- 모든 액션은 태블릿에서 검증
 - 예: 내 턴 아닌데 카드 내기 요청 → 거절
 - 손패에 없는 카드 사용 요청 → 거절

5) 상태 동기화 방식

핵심은 “전체 상태를 매번 보내지 말고, 이벤트 중심”이 좋아요.

예시:

- PLAYER_JOINED
- TURN_STARTED
- CARD_DRAWN
- CARD_PLAYED
- SCORE_UPDATED

그리고 폰에는:

- 내 개인 상태
- 공용 상태 중 필요한 부분

만 보내기

이렇게 해야 확장할 때 안정적입니다.

6) 끊김/재접속 대비 (필수)

로컬망이라도 끊김은 자주 생겨요.

필수 기능:

- 플레이어 재접속 (닉네임/토큰으로 복구)
- 태블릿이 현재 상태 재전송
- 일정 시간 응답 없으면 “연결 끊김” 표시
- 호스트(태블릿) 종료 시 게임 종료 안내

추가로:

- 태블릿이 자동 저장(턴 시작마다) 하면 앱 틱거도 복구 쉬움

7) MVP에서 딱 필요한 범위

초기엔 기능 욕심 줄이고 이 정도만:

필수

- 방 생성 / 입장 (QR)
- 플레이어 목록 / 준비 상태
- 턴 관리
- 개인 손패 표시
- 공용 보드 표시
- 액션 요청/검증
- 재접속

나중에

- 관전자 모드
- 음성/채팅
- 클라우드 멀티플레이
- 원격 친구 초대

기술적으로도 방향 좋음

이 구조는 나중에 확장도 가능해요.

초기:

- 로컬 Host 모드만

나중:

- 같은 엔진을 그대로 써서
 - 클라우드 서버 모드 추가 가능
 - 태블릿 host → 전용 서버로 치환 가능

즉, 지금 설계가 나중 확장을 막지 않아요.

한 줄로 정리

지금 잡은 방향은 아주 좋고, 특히 초기 MVP에 최적화된 구조예요.

핵심은 ***태블릿 authoritative host + 폰은 클라이언트***를 끝까지 유지하는 겁니다.

원하면 다음으로

통신 메시지 구조(예시 JSON), 방/턴 상태 머신, MVP 화면 목록까지 바로 짜드릴게요.

응

바이브코딩을 할 수 있도록 기획을 구체화해서 짜줘

좋아요. 아래처럼 가면 바이브코딩(LLM+코파일럿)으로 바로 구현 시작할 수 있게 됩니다.

핵심은 “추상 아이디어”가 아니라 모듈 경계 / 데이터 구조 / 메시지 규격을 먼저 고정하는 거예요.

1) 제품 정의 (MVP 기준)

제품 한 줄

- 태블릿이 호스트(로컬 서버) 역할을 하고, 같은 Wi-Fi에 있는 각 플레이어의 폰이 개인 UI로 접속하는 보드게임 플랫폼

앱

MVP 목표

- 한 개의 게임팩(예: 카드 기반 턴제 게임)으로
- 방 생성 → 접속 → 준비 → 게임 시작 → 턴 진행 → 종료까지 안정적으로 돌아가게 만들기

MVP에서 일부러 안 하는 것

- 인터넷 원격 멀티플레이
- 게임팩 스토어/결제
- 음성 채팅
- 복잡한 애니메이션
- 다양한 게임 장르 지원 (처음엔 카드/턴제 중심만)

2) 시스템 구조 (핵심)

역할 분리

태블릿 (Host / Authority)

- 로컬 서버 실행
- 게임 상태의 단일 진실(source of truth)
- 룰 검증 / 턴 관리 / 승패 판정
- 공용 보드 UI 렌더링

핸드폰 (Client)

- 플레이어 입력 UI
- 개인 정보 표시 (손패, 비밀 목표 등)
- 서버에 액션 요청
- 서버 상태 반영

3) 기술 스택 추천 (MVP 친화)

바이브로딩 기준으로 **React Native + Node/TS**가 제일 무난합니다.

앱 프레임워크

- **React Native + Expo** (태블릿/폰 공통 코드베이스)
- TypeScript 필수

통신

- 로컬 네트워크 + **WebSocket**
- 태블릿이 WebSocket 서버 열기 (Host 모드)

로컬 발견(선택)

- 1차 MVP: QR코드로 접속 주소 전달
- 2차: mDNS/Bonjour 자동 발견

상태관리

- **Zustand** (클라이언트 상태)
- 서버는 순수 TS 상태 머신

데이터 저장

- 태블릿 로컬 저장: SQLite 또는 파일(JSON snapshot)
- 자동 저장(턴 시작/종료 시)

공통 패키지 (중요)

모노레포로 분리 추천:

- packages/shared-types (메시지 타입, 게임 상태 타입)
 - packages/game-engine (룰/턴 처리)
 - apps/mobile (태블릿/폰 앱)
-

4) MVP 게임팩 범위 (첫 게임 예시)

처음 게임은 카드 중심 턴제로 제한하세요.

예시 규칙(샘플 게임):

- 각 플레이어 손패 5장
- 내 턴에 카드 1장 사용
- 카드 효과로 점수/자원 변화
- 라운드 종료 후 점수 계산
- 3라운드 후 종료

이렇게 하면 엔진에서 필요한 공통 기능이 명확해집니다:

- 텍/드로우/셔플
 - 손패 관리
 - 턴 순서
 - 액션 검증
 - 점수 업데이트
 - 승리 조건
-

5) 화면 목록 (MVP)

태블릿 화면

1. 호스트 홈
 - 방 생성 버튼
 - 설치된 게임팩 목록
2. 로비
 - 방코드 / QR 표시
 - 접속 플레이어 목록
 - 준비 상태
 - 게임 시작 버튼
3. 게임 보드 화면
 - 현재 턴 플레이어
 - 공용 정보(점수, 라운드, 공용 카드 등)
 - 로그(최근 액션)
4. 게임 종료 화면
 - 순위 / 점수
 - 다시 시작 / 로비로

폰 화면

1. 참가 홈
 - QR 스캔 / 방코드 입력
2. 닉네임 입력/입장
3. 대기실
 - 준비 버튼
4. 개인 플레이 화면
 - 내 손패
 - 내 자원/개인 상태

- 가능한 액션 버튼
 - 턴 아님 표시

5. 결과 화면

 - 점수 / 결과

6) 상태 모델 (핵심 설계)

전체 게임 상태 (서버만 보유)

```
// TypeScript

gamerName: string,
hostStartedAt: number;

players: Record<string, PlayerState>;
playerOrder: string[];

game: GameState; // 게임팩별 상태 (공용+개인 포함)
turn: TurnState;

log: GameLogEntry[];
version: number; // 상태 버전 (동기화용)

};
```

플레이어 상태

```
// TypeScript

CONNECTION_ID: string,           // 연결 ID
isConnected: boolean;           // 연결 여부
isReady: boolean;

// 게임 중 개인 상태
handCardIds: string[];          // 손에 든 카드 ID 목록
secretObjectives?: string[];      // 비밀 목표 목록
resources: Record<string, number>; // 리소스 정보

// 재접속용
reconnectToken: string;
}.
```

터 상태

```
//> TypeScript

    activeLayerId: string,
    step: "START" | "MAIN" | "END",
    actionCountThisTurn: number;
}


```

7) 권한/가시성 모델 (중요)

LLM으로 코딩할 때 여기 훈들리면 버그 많이 납니다.

원칙

- 서버 상태는 하나
- 클라이언트에게는 **가공된 뷰(ViewModel)**만 전달

서버가 만드는 뷰

태블릿용 공용 뷰

- 점수판
- 현재 턴
- 공용 카드/보드
- 공개 로그

폰용 개인 뷰

- 내 손패 (내 것만)
- 내 비밀정보
- 공용 정보
- 가능한 액션 목록 (allowedActions)

즉, 폰은 전체 상태를 몰라도 됨.

8) 통신 프로토콜 (WebSocket 메시지 규격)

이거 먼저 고정하면 바이브코딩이 훨씬 쉬워집니다.

공통 규칙

모든 메시지 공통 필드:

```
// TypeScript
{
  sessionid: string,
  ts: number;
}
```

클라이언트 → 서버 메시지

1) 방 입장

```
// TypeScript
{
  roomcode: string,
  nickname: string;
  reconnectToken?: string;
}
```

2) 준비 상태

```
// TypeScript
r;
```

3) 액션 요청

```
// TypeScript

actionType: READ_CARD | END_TURN | SELECT_TARGET,
actionData: Record<string, any>;
clientActionId: string; // 중복 방지
};

};
```

4) 핑

</> TypeScript

5

서버 → 클라이언트 메시지

1) 입장 결과

</> TypeScript

```
    success: boolean,
    playerId?: string;
    reconnectToken?: string;
    errorCode?: string;
    errorMessage?: string;
  };
};
```

2) 로비 상태 업데이트

</> TypeScript

```
  roomCode: string,
  players: Array<{
    playerId: string;
    nickname: string;
    isReady: boolean;
    isConnected: boolean;
  }>;
  canStart: boolean;
};

};
```

3) 개인 상태 업데이트 (폰)

</> TypeScript

```

type PlayerViewMessage = BaseMessage & {
  type: "PLAYER_VIEW";
  payload: {
    phase: "LOBBY" | "IN_GAME" | "FINISHED";
    me: {
      playerId: string;
      nickname: string;
      hand: CardView[];
      resources: Record<string, number>;
    };
    shared: {
      round: number;
      activePlayerId: string;
      scores: Record<string, number>;
      publicBoard: any;
    };
    allowedActions: AllowedAction[];
    version: number;
  };
};

```

4) 보드 상태 업데이트 (태블릿)

typescript

```

phase: "LOBBY" | "IN_GAME" | "FINISHED",
board: any;
scores: Record<string, number>;
turn: TurnState;
log: GameLogEntry[];
version: number;
};
};

```

5) 액션 결과/에러

typescript

```

clientActionId: string,
reason: string;
code: "NOT_YOUR_TURN" | "INVALID_CARD" | "INVALID_TARGET" | "PHASE_MISMATCH";
};
};

```

9) 서버 처리 규칙 (중요)

액션 처리 파이프라인 (절대 고정)

1. 메시지 수신
2. 세션/플레이어 인증 확인
3. 중복 요청 체크 (`clientActionId`)
4. 현재 턴/단계 검증
5. 룰 검증 (손패에 카드 있는지 등)
6. 상태 변경 (immutable or controlled mutation)
7. 버전 증가
8. 태블릿/각 폰에 뷰 재계산 후 브로드캐스트
9. 로그 기록
- 10. 자동 저장**

이 흐름을 함수로 고정하세요:

« TypeScript

```
< MAIN_ACTION  
-> broadcast()  
-> persistSnapshot()
```



10) 상태 머신 (방/게임)

세션 상태 머신

« Plain text

```
< GAME_STARTING  
-> IN_GAME  
-> ROUND_END (optional)  
-> GAME_FINISHED  
-> BACK_TO_LOBBY
```



턴 상태 머신

« Plain text

```
-> NEXT_PLAYER
```



규칙 예시

- END_TURN 는 MAIN_ACTION 에서만 가능
- PLAY_CARD 성공 시 actionCountThisTurn += 1
- 액션 후 승리 조건 확인
- 승리 조건 충족 시 GAME_FINISHED

11) 게임팩 구조 (MVP 버전)

처음부터 완전 범용 엔진 말고, 반범용으로 갑니다.

게임팩 파일 구조 예시

« Plain text

```
rules.ts  
boardLayout.json  
strings.ko.json
```



manifest.json

« JSON

```

    "name": "샘플 카드 배틀",
    "minPlayers": 2,
    "maxPlayers": 4,
    "estimatedMinutes": 15,
    "version": "0.1.0"
}

```

cards.json

↔ JSON



J

rules.ts (게임팩별 룰 툐)

↔ TypeScript



```

    applicationState: GameSessionState, playerId: string, action: GameAction, gameSessionId: string): void;
checkGameEnd(state: GameSessionState): { ended: boolean; winnerIds?: string[] };
buildBoardView(state: GameSessionState): BoardView;
buildPlayerView(state: GameSessionState, playerId: string): PlayerView;
}

```

포인트:

- rules.ts 만 코드
- 나머지는 데이터(JSON)

이러면 AI가 게임팩 추가 코딩도 잘 해줍니다.

12) 폴더 구조 (바이브코딩 친화)

↔ Plain text



```

src/
  screens/
  components/
  host/
  client/
  hooks/
  packages/
    shared-types/
    src/
      messages.ts
      state.ts
      views.ts
  game-engine/
    src/
      session/
      transport/
      reducers/
      validators/
      persistence/
      gamepacks/
        sample-card-battle/

```

13) 구현 우선순위 (진짜 중요)

Sprint 1: 연결/로비

- 태블릿 Host 모드
- WebSocket 서버 구동
- QR로 접속 정보 공유
- 폰 Join/Ready
- 로비 상태 브로드캐스트

Sprint 2: 게임 루프 최소 버전

- 게임 시작
- 턴 표시
- 손패 표시
- 카드 1장 사용
- 턴 종료
- 점수 업데이트

Sprint 3: 안정화

- 재접속
- 중복 액션 방지
- 자동 저장/복구
- 에러 처리 메시지
- 연결 끊김 UX

Sprint 4: 게임팩 정리

- manifest/cards/rules 구조 확정
- 샘플 게임팩 1개 분리
- 게임팩 로더 작성

14) 바이브코딩용 작업 단위 쪼개기 (LLM 프롬프트 잘 먹는 방식)

AI에게 한 번에 “앱 전체 만들어줘” 하면 망합니다.

아래처럼 작게 쪼개서 시키세요.

좋은 작업 단위 예시

1. shared-types/messages.ts 타입 정의
2. WebSocket 서버 래퍼 구현
3. 로비 세션 매니저 구현
4. JOIN_ROOM 핸들러 구현
5. READY 토큰 핸들러 구현
6. 태블릿 로비 UI 구현
7. 폰 참가 UI 구현
8. 게임 시작 reducer 구현
9. 샘플 카드 게임 룰 구현
10. PLAYER_VIEW/BOARD_VIEW 빌더 구현

15) AI 코딩 프롬프트 템플릿 (실전용)

템플릿 1: 타입 정의 요청

TypeScript로 shared-types/messages.ts 를 작성해줘.

WebSocket 메시지 규격은 JOIN_ROOM, JOIN_ROOM_ACK, LOBBY_STATE, SUBMIT_ACTION,

PLAYER_VIEW, BOARD_VIEW, ACTION_REJECTED를 포함해야 해.
각 메시지는 type, ts, sessionId?, requestId? 필드를 공통으로 갖고 discriminated union으로 구성해줘.
export type ClientToServerMessage, ServerToClientMessage 를 만들어줘.

템플릿 2: 서버 핸들러 요청

Node/TypeScript로 handleJoinRoom 함수를 구현해줘.

입력: 현재 세션 상태, 소켓 연결 정보, JOIN_ROOM 메시지.

요구사항:

- “roomCode 검증”
 - “reconnectToken 있으면 재접속 처리”
 - “없으면 신규 플레이어 생성”
 - “reconnectToken 발급”
 - “LOBBY_STATE 브로드캐스트용 payload 반환”
 - “예외 대신 Result 타입 반환
- 타입 안정성 높게 작성해줘.”

템플릿 3: 룰 엔진 요청

sample-card-battle 게임팩의 getAllowedActions / applyAction 구현해줘.

규칙:

- “내 턴이고 MAIN 단계일 때만 PLAY_CARD, END_TURN 가능”
 - “PLAY_CARD는 손패에 있는 카드만 가능”
 - “공격 카드는 점수 +1”
 - “드로우 카드는 덱에서 1장 뽑기”
 - “액션 후 상태 버전 증가하지 말고, 순수 상태 변경만 반환
- 함수는 순수 함수로 작성하고 테스트 코드도 같이 작성해줘.”

16) 테스트 전략 (MVP 필수)

UI보다 먼저 서버/엔진 테스트를 깔아두면 훨씬 안정적입니다.

단위 테스트 대상

- JOIN_ROOM (신규/재접속)
- SET_READY
- SUBMIT_ACTION 검증
- 턴 진행
- 카드 사용 룰
- 승리 조건 판정

시나리오 테스트

- 2명 입장 → 준비 → 시작 → 턴 진행 → 종료
- 잘못된 턴 액션 거절
- 연결 끊김 후 재접속 복구

17) 실패 포인트 미리 방지 체크리스트

네트워크

- 같은 Wi-Fi인데도 AP 격리로 안 될 수 있음 안내
- 로컬 네트워크 권한 요청 안내(iOS)
- 연결 끊김 감지(heartbeat)

동기화

- 서버 authoritative 유지
- clientActionId 중복 방지

- 상태 버전(version) 관리

UX

- “내 턴/아님” 명확히 표시
 - 가능한 액션만 활성화
 - 에러 메시지 사용자 친화적으로 표시
-

18) MVP 완료 기준 (Definition of Done)

아래가 되면 “첫 버전 성공”입니다.

- 태블릿에서 방 생성 가능
 - 같은 Wi-Fi의 폰 2~4대 접속 가능
 - 로비 준비 상태 관리 가능
 - 게임 시작 후 턴/라운드 진행 가능
 - 각 폰에서 손패만 보임 (비공개 보장)
 - 잘못된 액션은 서버가 거절
 - 재접속 시 플레이어 상태 복구
 - 한 개 게임팩 완주 가능
-

19) 다음 단계 (MVP 이후)

1. 게임팩 2개째 추가 (엔진 범용성 검증)
 2. 자동 발견(mDNS)
 3. 게임팩 설치/업데이트 UX
 4. 게임팩 SDK 문서화
 5. 나중에 클라우드 모드 확장
-

원하면 다음 답변에서 바로 이어서

shared-types 타입 코드 초안 + 샘플 게임팩 상태/액션 타입까지 실제 코드 형태로 짜드릴게요.