PRINTABLE STUDY PACKET

Android 앱 개발 공부 가이드 with `zipkr`

안드로이드 기초부터 `zipkr` 코드베이스 해부, 그리고 2026년 5월 기준 최신 안드로이드 개발 흐름까지 한 번에 따라갈 수 있도록 만든 인쇄용 학습 문서.

문서 목적
프린트해서 밑줄 치고 메모하며 읽을 수 있는 개인 학습용 기준 문서.

기준 날짜
2026-05-06. 최신 트렌드/버전 정보는 이 날짜 기준으로 공식 문서를 확인해 반영.

학습 방향
`앱 진입점(안드로이드 기초) -> zipkr 구조 -> 실전 기술 -> 최신 트렌드`

사용 방법
1차 정독 후 체크리스트 표시, 2차부터는 실제 파일을 열어 줄 단위로 함께 읽기.

작성 대상: jewan 개인 학습용. 제품 문서가 아니라 별도 학습 기록이며, 블로그 공개용 원고의 초안이 아니다.

1. 이 문서로 무엇을 얻어야 하는가

이 문서의 최종 목표는 단순히 "안드로이드 앱을 따라 쳐볼 수 있다"가 아니다. 목표는 세 가지다.

1. 안드로이드 OS가 앱을 어떻게 실행하는지 설명할 수 있어야 한다.
2. `zipkr`의 각 파일이 왜 존재하고 왜 그 위치에 있는지 설명할 수 있어야 한다.
3. 작은 실전 앱을 비슷한 구조로 스스로 다시 만들 수 있어야 한다.

핵심 원칙
안드로이드 일반론만 따로 오래 공부하지 않는다. `zipkr`를 해부하면서 필요한 기초를 그때그때 정확히 붙인다. 즉, `기초 -> 코드 -> 다시 기초 -> 다시 코드`의 반복으로 간다.

이 문서가 다루는 범위

Android 기본 구조 Manifest Application Activity Jetpack Compose MVVM StateFlow Repository Hilt Retrofit DataStore Gradle 테스트 최신 트렌드

2. 현재 `zipkr`를 어떻게 봐야 하는가

`zipkr`는 거대한 상용 서비스 코드베이스가 아니다. 대신 아주 좋은 학습 재료다. 이유는 기능은 비교적 작지만 실전 앱에서 필요한 경계들이 다 들어 있기 때문이다.

- 앱 부팅 진입점이 있다: `AndroidManifest.xml`, `ZipkrApp.kt`, `MainActivity.kt`
- Compose UI가 있다: `ui/search`, `ui/detail`, `ui/components`
- 상태 관리가 있다: `ViewModel`, `StateFlow`
- 도메인 모델이 있다: `Address`, `Coordinate`, `AppError`
- 외부 API 연동이 있다: 행안부 주소 API, 카카오 API
- 저장소가 있다: `DataStore` 기반 최근/즐겨찾기
- DI가 있다: `Hilt`
- 운영 요소가 있다: `AdMob`, `Crashlytics`, 로깅
- 테스트가 있다: ViewModel/Repository/Util 테스트

지금 가장 중요한 것은 "최신 버전으로 빨리 올리기"가 아니다. 먼저 현재 코드가 왜 돌아가는지 이해해야 한다. 업그레이드는 이해 이후에 하는 것이 좋다.

3. 큰 공부 로드맵

가장 효율적인 순서는 다음과 같다.

Phase	핵심 주제	대표 파일	완료 기준

1	앱 진입점 이해	`AndroidManifest.xml`
		`ZipkrApp.kt`
		`MainActivity.kt`
			앱 아이콘을 눌렀을 때 실행 순서를 설명할 수 있다.

2	Compose 화면 구조	`SearchScreen.kt`
		`ui/components/*`
		`ui/theme/*`
			화면과 컴포넌트의 차이, `remember` 계열의 역할을 설명할 수 있다.

3	상태 관리와 MVVM	`SearchViewModel.kt`
		`SearchUiState.kt`
		`DetailViewModel.kt`
			UI는 상태를 그리고, ViewModel은 흐름을 다룬다는 구조를 설명할 수 있다.

Phase	핵심 주제	대표 파일	완료 기준

4	도메인 모델과 앱 언어	`Address.kt`
		`AppError.kt`
		`Result.kt`
			외부 API 응답과 내부 모델을 왜 분리하는지 설명할 수 있다.

5	Repository와 저장소	`AddressRepository*.kt`
		`CoordinateRepository*.kt`
		`DataStoreSearchHistoryRepository.kt`
			UI가 데이터 출처를 몰라도 되는 이유를 설명할 수 있다.

6	네트워크 계층	`provider/juso/*`
		`provider/kakao/*`
		`NetworkModule.kt`
			Retrofit, OkHttp, JSON 직렬화의 연결 구조를 설명할 수 있다.

7	DI와 객체 그래프	`DataModule.kt`
		`PersistenceModule.kt`
		`DiQualifiers.kt`
			객체가 누가 만들어서 누구에게 주입되는지 설명할 수 있다.

8	상세 화면과 UX 확장	`ui/detail/*`
		`KakaoMapWebView.kt`
		`MapDeepLinks.kt`
			바텀시트, WebView, 외부 앱 연계를 설명할 수 있다.

9	빌드/운영/배포 준비	`app/build.gradle.kts` `libs.versions.toml`	debug/release, BuildConfig, 키 관리 구조를 설명할 수 있다.
10	테스트와 검증 습관	`app/src/test/*`	무엇이 테스트되고 무엇이 비어 있는지 설명할 수 있다.

권장 운영 방식

- 1회독: 전체 흐름을 이해한다.
- 2회독: 파일별 역할과 연결 관계를 설명할 수 있게 만든다.
- 3회독: 왜 이렇게 설계했는지와 대안을 같이 본다.
- 4회독: 비슷한 코드를 직접 다시 작성해본다.

4. 파일 하나를 공부할 때의 표준 질문

매 파일마다 최소한 아래 질문을 고정으로 던지면 이해 속도가 빨라진다.

1. 이 파일은 왜 존재하는가?
2. 왜 이 폴더에 있는가?
3. 왜 이런 이름을 썼는가?
4. 누가 이 파일을 호출하는가?
5. 이 파일은 누구를 호출하는가?
6. 안드로이드 기본 개념 중 무엇과 연결되는가?
7. 이 코드가 빠지면 어떤 기능이 깨지는가?
8. 더 단순한 대안은 있었는가?
9. 현재 나는 이 파일을 1~5점 중 몇 점까지 이해했는가?

이해 점수 기준

점수	의미
1	코드가 낯설고 용어도 모른다.
2	대충 역할은 알지만 줄 단위 동작은 모른다.
3	줄 단위로 읽고 따라갈 수 있다.
4	왜 이렇게 설계했는지 설명할 수 있다.
5	비슷한 파일을 혼자 다시 작성할 수 있다.

5. `zipkr` 파일 읽기 순서

1차: 앱 부팅과 첫 화면

1. `zipkr/app/src/main/AndroidManifest.xml`
2. `zipkr/app/src/main/kotlin/com/jewan/zipkr/ZipkrApp.kt`
3. `zipkr/app/src/main/kotlin/com/jewan/zipkr/MainActivity.kt`

2차: 검색 화면과 상태

1. `zipkr/app/src/main/kotlin/com/jewan/zipkr/ui/search/SearchScreen.kt`
2. `zipkr/app/src/main/kotlin/com/jewan/zipkr/ui/search/SearchUiState.kt`
3. `zipkr/app/src/main/kotlin/com/jewan/zipkr/ui/search/SearchViewModel.kt`

3차: 재사용 컴포넌트와 테마

1. `ui/components/AddressResultCard.kt`
2. `ui/components/CopyBar.kt`
3. `ui/components/ErrorView.kt`
4. `ui/theme/Theme.kt`, `Color.kt`, `Type.kt`, `Spacing.kt`

4차: 도메인 모델

1. `data/Address.kt`
2. `data/Coordinate.kt`
3. `data/Sido.kt`
4. `data/AddressPage.kt`
5. `data/Result.kt`
6. `data/AppError.kt`

5차: Repository와 저장소

1. `data/AddressRepository.kt`
2. `data/AddressRepositoryImpl.kt`
3. `data/CoordinateRepository.kt`
4. `data/CoordinateRepositoryImpl.kt`
5. `data/SearchHistoryRepository.kt`
6. `data/DataStoreSearchHistoryRepository.kt`

6차: 네트워크와 Provider

1. `data/provider/AddressProvider.kt`
2. `data/provider/juso/JusoApi.kt`
3. `data/provider/juso/JusoModels.kt`
4. `data/provider/juso/JusoApiProvider.kt`
5. `data/provider/kakao/KakaoLocalApi.kt`
6. `data/provider/kakao/KakaoModels.kt`

7차: DI와 빌드

1. `di/DataModule.kt`
2. `di/PersistenceModule.kt`
3. `di/NetworkModule.kt`
4. `di/DiQualifiers.kt`
5. `app/build.gradle.kts`
6. `gradle/libs.versions.toml`

8차: 상세 화면과 실전 UX

1. `ui/detail/DetailSheet.kt`
2. `ui/detail/DetailHeader.kt`
3. `ui/detail/DetailViewModel.kt`
4. `ui/detail/DetailUiState.kt`
5. `ui/components/KakaoMapWebView.kt`
6. `ui/components/MapDeepLinkButtons.kt`
7. `util/MapDeepLinks.kt`
8. `util/KakaoMapHtmlBuilder.kt`

6. Phase별로 무엇을 공부해야 하는가

Phase 1. 앱 진입점

- **반드시 아는 것**: Manifest, Application, Activity, `onCreate`, intent-filter, theme
- **왜 중요한가**: 앱이 "어디서부터 시작되는가"를 모르면 나머지 구조가 전부 떠 있다.
- **학습 포인트**: OS가 앱을 어떻게 띄우는지, 전역 초기화는 어디서 하는지, 첫 화면은 어디서 여는지

Phase 2. Compose UI

- **반드시 아는 것**: `@Composable`, `Scaffold`, `Modifier`, `remember`, `rememberSaveable`, `LaunchedEffect`
- **왜 중요한가**: 이 앱의 UI는 XML이 아니라 Kotlin 함수로 구성된다.
- **학습 포인트**: 화면과 상태가 분리되고, UI는 함수처럼 조립된다는 감각

Phase 3. MVVM과 상태

- **반드시 아는 것**: ViewModel, StateFlow, lifecycle-aware collect, coroutine scope
- **왜 중요한가**: Compose에서는 "상태가 바뀌면 UI가 다시 그려진다"가 핵심이다.
- **학습 포인트**: Composable이 직접 네트워크를 때리지 않고 ViewModel을 경유하는 이유

Phase 4. 도메인 모델

- **반드시 아는 것**: DTO vs Domain Model, sealed class, typed error, pagination model
- **왜 중요한가**: 앱 내부 언어가 정리돼 있어야 UI와 API가 느슨하게 결합된다.

Phase 5. Repository와 저장소

- **반드시 아는 것**: repository abstraction, cache, DataStore, interface/implementation 분리
- **왜 중요한가**: UI가 네트워크와 저장소의 세부 구현을 몰라도 되게 만든다.

Phase 6. 네트워크

- **반드시 아는 것**: Retrofit interface, OkHttp interceptor, JSON serialization, API key injection
- **왜 중요한가**: 외부 서비스와 연결되는 경계는 앱 구조를 가장 많이 드러내는 곳이다.

Phase 7. DI(Hilt)

- **반드시 아는 것**: `@Inject`, `@Provides`, `@Binds`, `@Singleton`, `@Named`
- **왜 중요한가**: 객체 생성 책임이 분리돼야 테스트와 유지보수가 쉬워진다.

Phase 8. UX 확장

- **반드시 아는 것**: ModalBottomSheet, WebView, deep link, external intent
- **왜 중요한가**: "단순 CRUD"를 넘어 실전 모바일 UX 요소가 들어오기 시작한다.

Phase 9. 빌드/운영

- **반드시 아는 것**: Gradle, version catalog, debug/release, BuildConfig, manifest placeholders, signing
- **왜 중요한가**: 실행되는 코드와 배포되는 앱은 빌드 설정에 의해 크게 달라진다.

Phase 10. 테스트

- **반드시 아는 것**: JUnit, MockK, Truth, Turbine, coroutine test
- **왜 중요한가**: 앱 코드를 이해하려면 "무엇을 보장하려고 테스트를 쓰는지"를 알아야 한다.

7. 현재 `zipkr` 스택과 최신 흐름 비교

아래 표는 `zipkr` 현재 의존성 버전과, 2026-05-06 기준 공식 문서에서 확인되는 최신 흐름을 나란히 둔 것이다. 여기서 중요한 것은 "지금 당장 다 올려야 한다"가 아니라, **내 프로젝트가 시대 흐름에서 어디쯤 있는지 감각을 잡는 것**이다.

영역	zipkr 현재	공식 최신 흐름	학습 해석
Android Gradle Plugin	`8.5.2`	AGP `9.2.0` stable (2026-05-05 기준 최신 업데이트 페이지)	지금 프로젝트는 아주 낡은 수준은 아니지만 최신 도구 체인과는 차이가 있다.
Android Studio	프로젝트 파일상 직접 고정 없음	Android Studio `Panda 4 | 2025.3.4 Patch 1` stable, Canary는 `Quail 1 | 2026.1.1 Canary 3`	안정판과 실험판 채널이 분리되어 있다는 사실 자체를 이해하는 것이 중요하다.
Compose BOM	`2024.08.00`	Compose April '26 stable, BOM `2026.04.01`, core Compose `1.11.0`	UI 툴킷은 현재 공식 흐름보다 꽤 뒤에 있다. 다만 학습용으론 충분하다.
Compose compiler	`1.5.14`	release 페이지 기준 stable `1.5.15`	큰 차이는 아니지만, 최신 문서와 비교해 보는 연습은 필요하다.
Activity Compose	`1.9.1`	`1.13.0` stable	앱 진입점/Compose 연결 영역의 라이브러리도 계속 진화한다.
Lifecycle	`2.8.4`	`2.10.0` stable	상태 관리와 lifecycle 수집 패턴은 계속 업데이트된다.
Navigation	`navigation-compose 2.7.7`	기존 Navigation stable `2.9.7`, 별도로 Navigation 3 stable `1.1.1`	현재는 기존 Navigation 계열, 최신 Compose 흐름은 Navigation 3도 반드시 알아둘 가치가 있다.

> **권장 태도**
> 먼저 현재 버전으로 왜 돌아가는지 이해하고, 그다음 별도 브랜치에서 최신 흐름과 비교하며 올리는 것이 좋다. 이해 없이 버전부터 올리면 공부가 아니라 추적이 된다.

8. 2026년 5월 기준 최신 안드로이드 개발 트렌드

아래 내용은 2026-05-06 기준 공식 Android Developers 문서와 블로그를 확인해 정리했다.

트렌드 1. Jetpack Compose는 여전히 기본 선택지다

Compose는 안드로이드의 권장 현대 UI 툴킷이다. 2026년 4월 stable 릴리스 기준으로 core Compose는 `1.11.0`이며, BOM은 `2026.04.01`을 사용한다. 핵심 포인트는 단순히 "XML 대신 Kotlin"이 아니라, 상태 중심 UI와 함수형 조립 방식이 안드로이드 주류가 되었다는 점이다.

- 테스트 타이밍 API v2가 기본이 되면서, Compose 테스트는 점점 더 실제 coroutine 동작에 가까워졌다.
- shared element 디버깅 도구가 강화됐다.

- trackpad 입력과 제스처 지원이 강화되며 대형 기기/다양한 입력장치 대응이 중요해지고 있다.

트렌드 2. 적응형 UI는 선택이 아니라 기본 역량이 되고 있다

Android Developers는 adaptive apps를 강하게 밀고 있다. 공식 문서는 3억 대 이상의 large screen Android device를 언급하며, 제대로 적응형 UI를 지원하면 사용자 만족도와 Play 노출 측면 모두 이점이 있다고 설명한다.

- 폰만 보는 설계는 점점 덜 유리하다.
- tablet, foldable, ChromeOS, 데스크톱형 경험을 함께 고려해야 한다.
- 화면을 "한 크기"가 아니라 "window size와 layout strategy" 관점으로 보는 습관이 중요하다.

트렌드 3. Navigation 3는 Compose-first 흐름을 대표한다

Navigation 3는 Compose를 위해 설계된 새로운 navigation 라이브러리다. 공식 가이드는 개발자가 back stack을 직접 통제하고, adaptive layout까지 자연스럽게 연결할 수 있다는 점을 강조한다. 2026-04-22 기준 stable은 `1.1.1`이고, 2026-04-08에 `1.1.0`이 stable이 되었다.

- back stack이 더 명시적이다.
- 상태 보존과 적응형 레이아웃 연결이 쉬워진다.
- 지금 당장 도입 여부와 별개로, 최신 Compose 구조를 공부할 때 반드시 알아둘 주제다.

트렌드 4. 성능 최적화는 Baseline Profiles가 기본 축이 됐다

공식 문서는 Baseline Profiles가 첫 실행부터 코드 실행 속도를 약 30% 개선할 수 있다고 설명한다. 단순 미세 최적화가 아니라, 출시 품질의 기본 축으로 보는 분위기다. 특히 Compose 앱은 라이브러리 기반이기 때문에 성능 최적화의 체감 이점이 크다.

- 앱 시작 속도
- 화면 전환
- 스크롤 및 상호작용 부드러움
- Startup Profiles와 함께 쓰는 방향

트렌드 5. Android 16 대응은 현실이고, Android 17 준비도 빨라지고 있다

공식 최신 업데이트 페이지 기준 Android 16은 stable 플랫폼이다. 동시에 2026-02-13에는 Android 17 Beta 1이 공개되었고, 전통적인 Developer Preview 대신 연속적인 Canary 채널로 전환하는 방향이 발표됐다.

- 앞으로는 "연 1회 크게 보고 끝"이 아니라 "더 자주, 더 일찍 대응"하는 흐름을 익혀야 한다.
- Android 16 쪽에서는 progress-centric notifications, predictive back 관련 업데이트, key sharing API 같은 변화가 보인다.
- 즉, 플랫폼 학습은 정적 문법 공부가 아니라 변화 대응 루틴까지 포함한다.

트렌드 6. 도구 체인 자체도 빠르게 변한다

2026-05-05 기준 Android Developers 최신 업데이트 페이지에는 Android Studio 안정판으로 `Panda 4 | 2025.3.4 Patch 1`, Android Gradle Plugin stable로 `9.2.0`이 올라와 있다. 이제 IDE와 AGP는 앱 코드 못지않게 지속적으로 업데이트되는 학습 대상이다.

트렌드 7. "최신"을 무작정 따라가기보다, 주류 패턴을 먼저 잡아야 한다

가장 큰 함정은 버전 숫자를 외우는 것이다. 실제로 더 중요한 것은 아래 순서다.

1. 현재 코드가 왜 돌아가는지 이해한다.
2. 공식이 어떤 방향으로 생태계를 끌고 가는지 본다.
3. 그 차이를 문서화한다.
4. 필요할 때만 업그레이드한다.

9. 지금 당장 무엇부터 공부해야 하는가

우선순위 A: 가장 먼저 잡아야 할 것

1. 앱 진입점: Manifest, Application, Activity
2. Compose 기본 문법: Composable, state, remember
3. ViewModel과 StateFlow

우선순위 B: 앱 구조를 이해하기 위해 바로 이어서 볼 것

1. 도메인 모델: `Address`, `AppError`, `Result`
2. Repository 패턴
3. Retrofit/OkHttp
4. Hilt

우선순위 C: 실전 앱 완성도를 위해 꼭 붙여야 할 것

1. DataStore
2. Gradle과 BuildConfig
3. 테스트
4. 성능 최적화(Baseline Profiles)

우선순위 D: 최신 흐름으로 확장할 때 공부할 것

1. Adaptive UI
2. Navigation 3
3. Android 16/17 변화 추적
4. Compose 최신 릴리스 노트 읽는 습관

10. 추천 주간 플랜

주차	집중 주제	산출물
1-2주차	앱 진입점 + Compose 기초	`AndroidManifest.xml`, `ZipkrApp.kt`, `MainActivity.kt`, `SearchScreen.kt` 해설 완료
3-4주차	상태 관리 + ViewModel	`SearchUiState.kt`, `SearchViewModel.kt`, `DetailViewModel.kt` 해설 완료
5-6주차	도메인 모델 + Repository	도메인 타입과 저장소 흐름을 말로 설명할 수 있음
7-8주차	네트워크 + Hilt	API 호출 경계와 DI 흐름을 설명할 수 있음
9-10주차	상세 화면 + 빌드/운영 + 테스트	앱 전체 구조를 처음부터 끝까지 설명 가능

> **더 현실적인 운영 팁**
> 주차보다 체크포인트가 더 중요하다. 예를 들어 "MainActivity까지 설명 가능", "SearchViewModel 설명 가능", "Repository와 Provider 차이 설명 가능"처럼 끊으면 흔들리지 않는다.

11. 체크포인트

Checkpoint A. 앱 진입점

- [ ] Manifest가 무엇인지 설명할 수 있다.
- [ ] Application과 Activity 차이를 설명할 수 있다.
- [ ] 앱 아이콘을 누른 뒤 첫 화면이 뜰 때까지의 순서를 말할 수 있다.

Checkpoint B. Compose와 상태

- [ ] `@Composable`가 무엇인지 설명할 수 있다.
- [ ] `remember`와 `rememberSaveable` 차이를 말할 수 있다.
- [ ] ViewModel이 필요한 이유를 설명할 수 있다.

Checkpoint C. 데이터 흐름

- [ ] DTO와 도메인 모델 차이를 설명할 수 있다.
- [ ] Repository가 왜 필요한지 설명할 수 있다.
- [ ] API 실패를 왜 `AppError`로 바꾸는지 설명할 수 있다.

Checkpoint D. 인프라

- [ ] Hilt가 무엇을 해주는지 설명할 수 있다.
- [ ] `BuildConfig`와 `local.properties`를 왜 쓰는지 설명할 수 있다.
- [ ] DataStore를 왜 선택했는지 설명할 수 있다.

Checkpoint E. 최신 흐름

- [ ] Compose 1.11과 BOM 개념을 이해한다.
- [ ] Adaptive UI가 왜 중요한지 설명할 수 있다.
- [ ] Baseline Profiles가 왜 중요한지 설명할 수 있다.
- [ ] Android 16/17 대응이 왜 "지속적인 변화 추적" 문제인지 설명할 수 있다.

12. 공부하면서 절대 잊지 말아야 할 것

1. 코드를 이해하지 못한 채 최신 버전으로 점프하지 않는다.
2. 한 파일을 볼 때는 먼저 "책임"부터 본다.
3. 모르는 용어가 나오면 바로 정의를 옆에 적는다.
4. UI 코드는 상태와 분리해서 본다.
5. 네트워크 코드는 DTO와 도메인 모델을 구분해서 본다.
6. 빌드 스크립트는 코드가 아니라 "앱이 어떤 조건으로 만들어지는지"를 선언하는 문서라고 생각한다.
7. 최신 트렌드는 암기 대상이 아니라 "방향 감각"을 잡는 재료다.

13. 지금 이 문서를 읽고 바로 할 일

1. 이 PDF를 출력한다.
2. 1장과 3장을 읽고 공부 순서에 동의하는지 표시한다.
3. 5장의 파일 읽기 순서에 따라 첫 대상 파일을 연다.
4. 각 파일마다 "역할 / 연결 / 안드로이드 개념 / 모르는 점"을 메모한다.
5. 이해 점수를 1~5로 적는다.
6. 모르는 점은 다음 세션에서 바로 해설 대상으로 넘긴다.

> 권장하지 않는 방식은 "오늘은 이것도 조금, 저것도 조금"이다. 지금은 넓게 훑는 대신, 파일 하나를 끝까지 정확하게 보는 것이 훨씬 낫다.

14. 최신 공식 참고 자료

아래 자료는 2026-05-06에 확인한 공식 문서/블로그 기준이다.

- Jetpack Compose overview
- Compose release notes
- What's new in the Jetpack Compose April '26 release
- Adaptive apps
- Navigation 3 guide
- Navigation 3 release notes
- Baseline Profiles overview
- Android Developers latest updates
- Android 16 features and APIs
- The First Beta of Android 17

문서 내 "최신" 정보는 시간이 지나면 바뀐다. 버전이나 툴 이름을 실제로 쓸 때는 항상 다시 공식 페이지를 확인한다.

PART 2

Android Phase 1 Entrypoint Workbook

PHASE 1 DETAILED WORKBOOK

Android 앱 진입점 워크북
Manifest, Application, Activity

`zipkr`의 가장 첫 시작점만 따로 분리한 인쇄용 상세 자료. `AndroidManifest.xml`, `ZipkrApp.kt`, `MainActivity.kt`를 줄 단위에 가깝게 읽으면서, 2026년 5월 기준 공식 권장 사항과 비교까지 할 수 있도록 구성했다.

학습 목표
앱 아이콘을 눌렀을 때 OS가 어떤 순서로 무엇을 실행하는지 설명할 수 있어야 한다.

대상 파일
`AndroidManifest.xml`, `ZipkrApp.kt`, `MainActivity.kt`

핵심 개념
Manifest, Application, Activity, SplashScreen API, edge-to-edge, Hilt, app startup

읽는 방식
한 섹션씩 읽고 바로 실제 파일을 열어 같은 줄을 눈으로 대조한다.

작성일: 2026-05-06. 공식 최신 권장 사항은 Android Developers 문서를 기준으로 다시 확인해 반영했다.

1. 먼저 큰 그림부터

안드로이드 앱은 보통 아래 순서로 시작한다.

```
1. 사용자가 홈 화면에서 앱 아이콘을 누른다.
2. Android OS가 앱의 AndroidManifest.xml을 읽는다.
3. 어떤 Application 클래스를 쓸지 확인한다.
4. 앱 프로세스를 만들고 Application 객체를 생성한다.
5. Launcher로 등록된 MainActivity를 시작한다.
6. MainActivity.onCreate()가 호출된다.
7. setContent { ... } 안에서 Compose UI가 시작된다.
```

> **이 문서에서 반드시 잡아야 하는 감각**
> `Manifest`는 앱의 신분증, `Application`은 앱 전역 객체, `Activity`는 안드로이드 화면 컨테이너, `Composable`은 그 안에 그려지는 실제 UI 함수다.

왜 이 순서가 중요한가

- `Manifest`를 모르면 앱이 어디서 시작되는지 모른다.
- `Application`을 모르면 전역 초기화 위치를 헷갈리게 된다.
- `Activity`를 모르면 Compose의 시작 위치를 놓치게 된다.
- 이 세 가지를 먼저 이해하면 이후 모든 파일이 "큰 구조 안의 한 조각"으로 보인다.

2. 현재 코드 기준 실행 흐름

`zipkr`에서 실제로 벌어지는 일

1. `AndroidManifest.xml` 7~16행에서 앱 전역 설정과 `ZipkrApp`을 선언한다.
2. `AndroidManifest.xml` 31~39행에서 `MainActivity`를 launcher activity로 선언한다.
3. 프로세스가 뜨면 `ZipkrApp.onCreate()`가 실행된다.
4. 여기서 Timber 로깅 트리와 AdMob SDK를 초기화한다.
5. 그다음 `MainActivity.onCreate()`가 실행된다.
6. `installSplashScreen()`, `enableEdgeToEdge()`, `setContent { ZipkrTheme { SearchScreen() } }`가 순서대로 적용된다.
7. 최종적으로 사용자가 보는 첫 화면은 `SearchScreen()`이다.

현재 코드와 최신 권장 사항의 차이

주제	현재 zipkr	2026-05 공식 권장 흐름
edge-to-edge	`enableEdgeToEdge()` 사용 중	좋다. 다만 공식 문서는 Activity manifest에 `android:windowSoftInputMode="adjustResize"`도 함께 설정하라고 권장한다.
target SDK	`targetSdk = 34`	공식 edge-to-edge 문서는 Android 15(API 35)+ 타깃 시 edge-to-edge enforced를 설명한다. 즉, 이후 업그레이드 학습이 필요하다.
splash	`installSplashScreen()`를 `super.onCreate()` 전에 호출	공식 권장과 일치한다.
Application 초기화	Timber, CrashlyticsTree, AdMob 초기화 수행	가능하지만 공식 스타트업 최적화 문서는 `Application`에서 불필요한 eager initialization을 피하라고 권장한다. 무거운 초기화는 항상 의심해 봐야 한다.
Hilt	`@HiltAndroidApp` 사용	공식 Hilt 사용 방식과 일치한다.

3. AndroidManifest.xml 먼저 이해하기

대상 파일: `zipkr/app/src/main/AndroidManifest.xml`

원문

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:tools="http://schemas.android.com/tools">
4
5      <uses-permission android:name="android.permission.INTERNET" />
6
7      <application
8          android:name=".ZipkrApp"
9          android:allowBackup="true"
10         android:dataExtractionRules="@xml/data_extraction_rules"
11         android:fullBackupContent="@xml/backup_rules"
12         android:icon="@mipmap/ic_launcher"
13         android:label="@string/app_name"
14         android:roundIcon="@mipmap/ic_launcher_round"
15         android:supportsRtl="true"
16         android:theme="@style/Theme.Zipkr">
17
18         <meta-data
19             android:name="com.google.android.gms.ads.APPLICATION_ID"
20             android:value="${admobAppId}" />
21
22         <property
23             android:name="android.adservices.AD_SERVICES_CONFIG"
24             android:resource="@xml/gma_ad_services_config"
25             tools:replace="android:resource" />
26
27         <activity
28             android:name=".MainActivity"
29             android:exported="true"
30             android:theme="@style/Theme.Zipkr.Splash">
31             <intent-filter>
32                 <action android:name="android.intent.action.MAIN" />
33                 <category android:name="android.intent.category.LAUNCHER" />
34             </intent-filter>
35         </activity>
36     </application>
37 </manifest>
```

Manifest의 본질

Manifest는 "앱의 신분증"이자 "운영체제에게 제출하는 선언서"다. 이 파일에는 앱의 이름, 아이콘, 권한, 시작 Activity, 앱 전역 클래스, 테마 같은 정보가 들어간다. 안드로이드 OS는 이 파일을 보고 앱을 어떻게 띄울지 결정한다.

> **중요**
> Manifest는 Kotlin 코드가 아니다. 그런데 앱 실행 구조를 가장 먼저 결정한다. 즉, 안드로이드에서는 "코드 이전의 선언"도 앱 구조의 일부다.

묶음별 해설

줄	코드	의미
1	XML 선언	이 파일이 XML 문서라는 선언이다. 안드로이드 Manifest는 XML 형식으로 작성한다.
2-3	`<manifest ...>`	루트 태그다. `android:`, `tools:` 속성을 쓰기 위해 네임스페이스를 선언한다.
5	`INTERNET` permission	이 앱이 인터넷에 접근하도록 허용한다. 주소 API, 카카오 지도, AdMob, Firebase에 필요하다.
7-16	`<application ...>`	앱 전역 설정이다. Application 클래스, 백업 정책, 아이콘, 앱 이름, RTL 지원, 기본 테마를 선언한다.

줄	코드	의미
8	`android:name=".ZipkrApp"`	앱 전역 `Application` 클래스로 `ZipkrApp`을 사용하겠다는 뜻이다. Hilt와 전역 초기화가 여기서 연결된다.
9-11	backup/data extraction 관련	백업 허용과 데이터 추출 규칙을 지정한다. 앱 데이터를 OS 백업 체계와 어떻게 연결할지 선언하는 부분이다.
12-15	icon/label/RTL	사용자가 보는 앱 외형과 언어 방향성을 정한다.
16	`Theme.Zipkr`	앱 전체 기본 테마다. 특별히 다른 Activity 테마를 주지 않으면 이것이 적용된다.
18-20	AdMob `meta-data`	Google Mobile Ads SDK가 읽는 App ID 설정이다. 문자열을 코드에 박지 않고 Gradle에서 placeholder로 주입한다.
22-25	`AD_SERVICES_CONFIG`	광고 관련 시스템 설정 리소스를 지정한다. `tools:replace`는 manifest 병합 충돌 시 이 값을 우선 적용하겠다는 뜻이다.
27-35	`<activity ...>`	앱의 첫 화면 컨테이너인 `MainActivity` 선언이다.
28	`android:name=".MainActivity"`	실제 클래스는 `com.jewan.zipkr.MainActivity`다. 점 표기는 패키지 축약 표기다.
29	`android:exported="true"`	런처처럼 앱 바깥에서 이 Activity를 실행할 수 있게 허용한다. Android 12+에서는 intent-filter가 있으면 명시가 필수다.
30	`Theme.Zipkr.Splash`	이 Activity 시작 시에는 일반 테마가 아니라 스플래시 전용 테마를 적용한다.
31-34	`intent-filter`	이 Activity가 어떤 intent에 반응할지 정의한다. `MAIN + LAUNCHER` 조합은 "앱 아이콘을 눌렀을 때 시작되는 진입점"이라는 뜻이다.

이 파일을 보고 꼭 떠올려야 할 질문

1. 왜 `ZipkrApp`을 따로 만들었나?
2. 왜 `MainActivity`에 스플래시 테마를 따로 줬나?
3. 왜 AdMob ID를 코드에 직접 쓰지 않고 placeholder로 넣나?
4. 왜 launcher entry는 Activity이지 Composable이 아닌가?

메모

4. `ZipkrApp.kt` 이해하기

대상 파일: `zipkr/app/src/main/kotlin/com/jewan/zipkr/ZipkrApp.kt`

원문

```kotlin
package com.jewan.zipkr

import android.app.Application
import com.google.android.gms.ads.MobileAds
import com.google.firebase.crashlytics.FirebaseCrashlytics
import dagger.hilt.android.HiltAndroidApp
import timber.log.Timber

@HiltAndroidApp
class ZipkrApp : Application() {
    override fun onCreate() {
        super.onCreate()
        if (BuildConfig.DEBUG) {
            Timber.plant(Timber.DebugTree())
        } else {
            Timber.plant(CrashlyticsTree())
        }
        MobileAds.initialize(this) {}
    }
}

private class CrashlyticsTree : Timber.Tree() {
    override fun log(
        priority: Int,
        tag: String?,
        message: String,
        t: Throwable?,
    ) {
        if (priority < android.util.Log.WARN) return
        FirebaseCrashlytics.getInstance().log("[$tag] $message")
        if (t != null) FirebaseCrashlytics.getInstance().recordException(t)
    }
}
```

Application이란 무엇인가

`Application`은 앱 프로세스가 살아 있는 동안 유지되는 앱 전역 객체다. 화면 하나를 나타내는 `Activity`와 달리, 특정 화면에 묶이지 않는다. 보통 아래와 같은 전역 초기화 작업을 여기에 둔다.

- DI 프레임워크 시작
- 전역 로깅 설정
- 분석/크래시 리포팅 SDK 초기화
- 앱 전체에서 공통으로 쓰는 전역 설정

> 하지만 `Application.onCreate()`는 무조건 조심해야 하는 구간이다. 공식 스타트업 최적화 문서는 이 지점에서 불필요한 eager initialization을 피하라고 분명히 말한다. 즉, "전역이니까 아무거나 넣는 곳"이 절대 아니다.

줄 단위 해설

줄	코드	의미
1	`package com.jewan.zipkr`	이 클래스의 패키지다. Manifest의 `.ZipkrApp` 축약 표기는 이 패키지와 합쳐져 실제 클래스를 찾는다.
3	`Application` import	안드로이드 앱 전역 객체의 기반 클래스다.
4-7	SDK import들	AdMob, Crashlytics, Hilt, Timber를 이 파일에서 사용한다는 뜻이다. 즉, 이 파일의 책임은 "전역 초기화"다.
9	`@HiltAndroidApp`	Hilt를 앱 전역에서 시작하겠다는 선언이다. 공식 문서에 따르면 이 어노테이션이 Hilt code generation과 application-level dependency container 생성을 트리거한다.
10	`class ZipkrApp : Application()`	`ZipkrApp`은 `Application`을 상속받는다. 즉, 이 클래스는 앱 전체 수명주기에 묶인 객체다.
11	`override fun onCreate()`	앱 프로세스 생성 시 초기화 지점이다. 화면 단위가 아니라 앱 전역 단위다.
12	`super.onCreate()`	부모 클래스 초기화를 먼저 수행한다. 일반적으로 반드시 호출해야 한다.
13	`if (BuildConfig.DEBUG)`	현재 빌드가 debug인지 release인지에 따라 동작을 나눈다. `BuildConfig`는 Gradle이 생성하는 빌드 상수 클래스다.
14	`Timber.plant(Timber.DebugTree())`	디버그 빌드에서는 로컬 콘솔용 로깅 트리를 심는다. 쉽게 말해 "로그 출력을 어디로 보낼지" 설정하는 행위다.
15-16	`CrashlyticsTree()`	릴리스에서는 디버그용 verbose 로그 대신 Crashlytics로 보내는 트리를 심는다.
18	`MobileAds.initialize(this) {}`	AdMob SDK를 앱 시작 시 한 번 초기화한다. 화면마다 반복해서 초기화하지 않겠다는 의도다.
22	`private class CrashlyticsTree`	이 파일 내부 전용 로깅 구현체다. 외부에서 직접 쓸 필요가 없으므로 `private`다.
23-28	`override fun log(...)`	Timber가 실제 로그를 전달하는 진입점이다. priority, tag, message, throwable을 받는다.
29	`if (priority < WARN) return`	INFO/DEBUG/VERBOSE는 운영 노이즈이므로 버리고, WARN 이상만 보낸다.
30	`FirebaseCrashlytics.getInstance().log(...)`	문제 로그를 Crashlytics 비치명 이벤트로 전송한다.
31	`recordException(t)`	Throwable이 있으면 실제 예외로 기록한다. 나중에 운영 중 문제 분석에 사용된다.

왜 이 파일이 필요한가

- Hilt 앱 수준 초기화 지점이 필요하다.
- 로깅 전략을 빌드 타입별로 나누는 전역 지점이 필요하다.
- AdMob처럼 앱 전역에서 한 번 초기화할 SDK가 있다.

이 파일이 없으면?

- `@HiltAndroidApp`가 없어 Hilt 진입 구성이 깨진다.
- Timber 트리가 심어지지 않아 로그 전략이 없어질 수 있다.
- AdMob 초기화 위치를 다른 화면으로 옮겨야 한다.
- 전역 초기화 책임이 각 Activity로 흩어질 위험이 커진다.

최신 권장 사항과 연결

- Hilt 공식 문서는 모든 Hilt 앱이 `@HiltAndroidApp`가 붙은 `Application` 클래스를 가져야 한다고 설명한다.
- 공식 스타트업 최적화 문서는 `Application`에서 unnecessary eager initialization을 피하라고 권장한다.
- 따라서 이 파일을 읽을 때는 "지금 넣은 초기화가 충분히 가벼운가?"를 항상 같이 물어봐야 한다.

메모

5. `MainActivity.kt` 이해하기

대상 파일: `zipkr/app/src/main/kotlin/com/jewan/zipkr/MainActivity.kt`

원문

```kotlin
package com.jewan.zipkr

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.core.splashscreen.SplashScreen.Companion.installSplashScreen
import com.jewan.zipkr.ui.search.SearchScreen
import com.jewan.zipkr.ui.theme.ZipkrTheme
import dagger.hilt.android.AndroidEntryPoint

@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        installSplashScreen()
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            ZipkrTheme {
                SearchScreen()
            }
        }
    }
}
```

Activity란 무엇인가

Activity는 안드로이드 화면의 가장 기본적인 컨테이너다. 사용자가 보는 실제 UI는 Compose라 하더라도, 그 Compose를 붙일 "안드로이드 레벨의 화면 호스트"는 여전히 Activity다.

- Manifest에 launcher로 등록된다.
- OS가 생명주기를 관리한다.
- Back stack 안에서 이동/복귀 단위가 된다.
- Compose UI는 보통 Activity의 `setContent { ... }`에서 시작된다.

줄 단위 해설

줄	코드	의미
4	`ComponentActivity`	Compose와 잘 맞는 현대 AndroidX Activity 기반 클래스다.
5	`setContent`	여기서 Compose UI 루트를 붙인다. XML의 `setContentView()`와 대응되는 개념이지만, 대상이 View가 아니라 Composable tree다.
6	`enableEdgeToEdge`	시스템 바 뒤까지 UI를 그릴 수 있게 해주는 최신 권장 API다. 공식 문서는 backward compatible한 modern way로 설명한다.
7	`installSplashScreen`	Android 12+ 스타일 스플래시를 제어하는 API다. 공식 문서는 반드시 `super.onCreate()` 직전에 호출하라고 안내한다.
8-9	`SearchScreen`, `ZipkrTheme`	이 Activity가 최종적으로 검색 화면을 띄우고, 앱 테마를 감싸는 진입점이라는 뜻이다.
10	`@AndroidEntryPoint`	이 Activity도 Hilt 주입 대상이라는 뜻이다. Hilt가 Activity 스코프 의존성을 제공할 수 있게 된다.
13	`class MainActivity : ComponentActivity()`	앱 첫 화면 컨테이너다. Manifest에서 launcher로 연결된 바로 그 클래스다.
14	`onCreate(savedInstanceState: Bundle?)`	Activity 생성 시점 콜백이다. 최초 생성뿐 아니라 구성 변경 후 복원 시에도 다시 호출될 수 있다.
15	`installSplashScreen()`	공식 권장대로 `super.onCreate()` 전에 호출한다. 시스템 스플래시를 정확하게 적용하려는 의도다.
16	`super.onCreate(savedInstanceState)`	부모 Activity 초기화를 수행한다.
17	`enableEdgeToEdge()`	시스템 바 영역까지 컨텐츠를 그리는 환경을 연다. Android 15(API 35)+에선 edge-to-edge가 enforced이므로, 현재 안드로이드 UI 학습의 핵심 주제다.
18-22	`setContent { ... }`	Compose UI의 시작점이다. 여기서부터는 View XML이 아니라 Composable 함수 트리가 화면을 만든다.
19	`ZipkrTheme { ... }`	앱의 색상, 타이포, shape 같은 테마를 전체 UI에 적용한다.
20	`SearchScreen()`	사용자가 실제로 처음 보는 화면이다. 즉, `MainActivity`는 "호스트", `SearchScreen`은 "화면 내용"이다.

이 파일이 특히 좋은 이유

- 작고 명확하다. 그래서 Activity가 무엇을 해야 하는지와 하지 말아야 하는지를 동시에 보여준다.

- 실제 비즈니스 로직은 거의 없다. Activity는 호스트 역할에 집중한다.
- 스플래시, edge-to-edge, Compose 진입점, Hilt 연결이 한 파일에 모여 있다.

현재 코드에서 눈여겨볼 개선 포인트

- 공식 edge-to-edge 문서는 Activity manifest에 `android:windowSoftInputMode="adjustResize"`를 추가하라고 안내한다. 현재 `zipkr` manifest에는 없다.
- `targetSdk = 34`라서 Android 15 enforced edge-to-edge 환경은 아직 직접 맞고 있지 않다.
- 즉, 이 파일은 "현재도 꽤 현대적"이지만 "최신 권장 사항과의 차이도 있는" 좋은 학습 예제다.

메모

6. 세 파일을 함께 보면 보이는 구조

파일	책임	호출/연결	실수하기 쉬운 포인트
`AndroidManifest.xml`	앱의 신분과 시작 선언	OS가 먼저 읽는다	코드가 아니라 선언인데도 실행 구조를 결정한다는 사실을 놓치기 쉽다.
`ZipkrApp.kt`	앱 전역 초기화	Manifest의 `android:name`으로 연결	무거운 초기화를 너무 많이 넣으면 앱 시작이 느려질 수 있다.
`MainActivity.kt`	첫 화면 호스트	Manifest의 launcher activity로 연결	Activity가 로직까지 떠안기 시작하면 구조가 빠르게 무너진다.

이 세 파일의 역할 구분을 문장으로 외워두기

- **Manifest**: OS에게 앱 정보를 선언한다.
- **Application**: 앱 전역 초기화를 담당한다.
- **Activity**: 화면 컨테이너를 만들고 UI 루트를 붙인다.
- **Composable**: 사용자가 실제로 보는 UI를 그린다.

> **암기용 한 줄**
> `Manifest가 시작을 선언하고, Application이 앱 전역을 준비하고, Activity가 화면을 열고, Compose가 내용을 그린다.`

7. 최신 공식 권장 사항 정리

2026-05-06 기준 Android Developers 공식 문서 요약

1. SplashScreen API

- `installSplashScreen()`는 시작 Activity의 `onCreate`에서 `super.onCreate()` 직전에 호출한다.
- Android 12+(API 31+)에서는 플랫폼 스플래시를 사용하고, 그 이전 버전엔 라이브러리가 유사 동작을 재현한다.
- Manifest에서 시작 Activity나 앱에 시작 테마를 설정해야 한다.

2. edge-to-edge

- Android 15(API 35)+를 타깃하면 edge-to-edge가 enforced된다.
- 이전 버전까지 포함해 호환성 있게 가려면 Activity `onCreate()`에서 `enableEdgeToEdge()`를 호출한다.
- 공식 문서는 Activity manifest에 `android:windowSoftInputMode="adjustResize"` 설정을 권장한다.
- system bars와 IME inset을 고려한 레이아웃 처리가 중요하다.

3. Hilt

- 모든 Hilt 앱은 `@HiltAndroidApp`가 붙은 `Application` 클래스를 가져야 한다.
- `@AndroidEntryPoint`가 붙은 Activity/Fragment/View 등은 Hilt가 의존성을 제공할 수 있다.
- 즉, `ZipkrApp`과 `MainActivity`의 어노테이션 조합은 공식 방식과 맞는다.

4. 앱 시작 성능

- Baseline Profiles는 첫 실행부터 코드 실행 속도를 약 30% 개선할 수 있다고 공식 문서가 설명한다.
- 무거운 작업은 `Application`에서 eager init 하지 않는 것이 좋다.
- 필요하다면 App Startup library, Startup profile, cold start trace 같은 주제를 이어서 공부할 가치가 있다.

8. 지금 코드에서 직접 체크해볼 것

체크리스트

- [ ] 왜 `ZipkrApp`이 Manifest에 등록돼야 하는지 설명할 수 있는가?
- [ ] 왜 `MainActivity`가 launcher activity인지 설명할 수 있는가?
- [ ] 왜 `SearchScreen()`이 Manifest에 직접 나오지 않는지 설명할 수 있는가?
- [ ] 왜 `installSplashScreen()`가 `super.onCreate()` 앞에 있는지 설명할 수 있는가?
- [ ] 왜 `enableEdgeToEdge()`가 필요한지 설명할 수 있는가?
- [ ] 왜 Hilt가 `Application`과 `Activity` 양쪽 선언을 필요로 하는지 설명할 수 있는가?

- [ ] 왜 `Application`에 너무 많은 작업을 넣으면 위험한지 설명할 수 있는가?

직접 적어보기

Q1. 앱 아이콘을 눌렀을 때 `SearchScreen()`이 그려질 때까지의 순서를 직접 써보라.

Q2. `Application`과 `Activity`의 차이를 한 문장씩 적어보라.

Q3. 현재 `zipkr`에서 최신 edge-to-edge 권장 사항과 다른 점을 적어보라.

9. 다음 단계 예고

Phase 10이 끝나면 바로 다음은 Compose 첫 화면이다.

1. `SearchScreen.kt`에서 `setContent { ... }` 안쪽 세상을 본다.
2. 여기서 `Composable`, `Scaffold`, `remember`, `rememberSaveable`, `LaunchedEffect`를 본다.
3. 그다음 `SearchUiState.kt`와 `SearchViewModel.kt`로 내려간다.

> **다음 문서 후보**
> `Phase 2 Compose 화면 워크북`
> 대상: `SearchScreen.kt`, `ui/components/*`, `ui/theme/*`

10. 공식 참고 자료

- SplashScreen API reference
- Set up Edge-to-edge
- Use insets in Views and Compose
- Dependency injection with Hilt
- Tasks and the back stack
- Best practices for app optimization
- Android Developers latest updates

최신 권장 사항은 계속 바뀐다. 특히 edge-to-edge, target SDK, Android Studio, AGP 같은 항목은 이후 다시 확인해야 한다.

PART 3

Android Phase 2 Compose SearchScreen Workbook

PHASE 2 DETAILED WORKBOOK

Compose 첫 화면 워크북 SearchScreen.kt 중심

`zipkr`의 첫 실제 화면인 `SearchScreen.kt`를 기준으로 Jetpack Compose의 핵심 개념을 배우는 인쇄용 상세 자료. 상태 수명, `remember`, `rememberSaveable`, `LaunchedEffect`, `collectAsStateWithLifecycle()`, 상태 끌어올리기까지 함께 묶었다.

학습 목표
Compose 화면이 어떻게 함수처럼 조립되고 상태에 반응하는지 설명할 수 있어야 한다.

대상 파일
`SearchScreen.kt`, `SearchUiState.kt`, `Theme.kt`, `SidoAnchor.kt`

핵심 개념
Composable, recomposition, state lifespans, state hoisting, side effects, lifecycle-aware Flow collection

읽는 방식
파일 원문과 문서 해설을 같이 놓고, 각 Compose API가 왜 거기 있는지 체크한다.

작성일: 2026-05-06. Compose 최신 상태 수명 가이드, state hoisting, side effects, lifecycle-aware collect 공식 문서를 반영했다.

1. Compose 화면을 볼 때 먼저 잡아야 하는 관점

Compose는 "View를 객체로 만들고 붙이는 방식"보다 "상태를 받아 UI 함수를 다시 호출해 그리는 방식"에 가깝다. 즉, 화면을 이해하려면 먼저 **상태가 어디에 있고, 누가 읽고, 누가 바꾸는지**를 봐야 한다.

> **Compose 핵심 문장**
> UI는 상태의 함수다. 상태가 바뀌면 Compose는 필요한 부분만 다시 그린다.

SearchScreen을 보는 첫 질문

1. 이 화면이 직접 들고 있는 상태는 무엇인가?
2. ViewModel에서 받는 상태는 무엇인가?
3. 왜 어떤 상태는 `rememberSaveable` 이고, 어떤 상태는 ViewModel인가?
4. 왜 어떤 작업은 `LaunchedEffect` 안에서 일어나는가?
5. 왜 Flow는 그냥 읽지 않고 `collectAsStateWithLifecycle()` 로 읽는가?

2. 현재 파일들의 역할

파일	역할	이 파일에서 꼭 배워야 할 것
`SearchScreen.kt`	검색 메인 화면을 Compose로 조립하는 루트 화면	Composable 구조, 화면 상태 조립, local API, saveable UI state, callbacks 묶기
`SearchUiState.kt`	검색 화면의 단일 상태 모델	상태 머신, UI phase, 불변 상태 모델
`Theme.kt`	앱 전역 Material theme 진입점	Material3, dynamic color, light/dark theme
`SidoAnchor.kt`	검색바 옆 지역 선택 컴포넌트	재사용 컴포넌트, 토큰 공유, 화면과 컴포넌트 책임 분리

3. SearchScreen 전체 구조 먼저 보기

이 파일은 길어 보이지만 구조는 단순하다. 크게 다섯 덩어리다.

```
1. rememberAutoFocusKeyboard()
   - 입력창 자동 포커스와 키보드 표시

2. SearchScreen()
   - ViewModel 상태 수집
   - 화면 내부 saveable state 보유
   - 콜백 묶음 생성
   - Scaffold로 전체 레이아웃 구성

3. SearchOverlays()
   - 시·도 선택 시트 / 상세 시트 표시

4. SearchScreenContent()
   - 실제 본문 레이아웃 조립

5. SearchBar(), SearchBody(), SearchCallbacks
   - 작은 조각들로 화면 로직 분리
```

> **좋은 Compose 파일의 전형적인 모습**
> 루트 화면이 모든 걸 한 함수에서 끝내지 않는다. 상태 수집, 레이아웃 조립, 작은 UI 조각, 오버레이를 분리해 재구성 비용과 가독성을 동시에 관리한다.

실제 실행 흐름

1. `MainActivity.setContent` 에서 `SearchScreen()` 이 호출된다.
2. `SearchScreen()` 이 ViewModel 상태를 Compose state로 변환해 읽는다.
3. 필요한 UI 로컬 상태를 `rememberSaveable` 로 만든다.
4. `Scaffold` 안에서 상단 바와 본문을 구성한다.
5. 본문은 `SearchScreenContent()` 가 담당한다.
6. 시트는 `SearchOverlays()` 가 별도로 렌더링한다.
7. `phase` 에 따라 `SearchBody()` 가 다른 UI를 그린다.

4. 꼭 알아야 할 Compose 최신 개념

1. 상태 수명: `remember` vs `rememberSaveable`

2026년 공식 Compose state lifespans 문서는 `remember`, `retain`, `rememberSaveable`, `rememberSerializable` 의 차이를 분리해서 설명한다.

API	recomposition survive	activity recreation survive	주된 용도
`remember`	예	아니오	화면 구성 중 잠깐 유지할 값, 재구성 동안만 유지하면 되는 값
`rememberSaveable`	예	예	사용자 입력, 선택 상태, 시트 열림 여부처럼 화면 재생성에도 남아야 하는 값
`rememberSerializable`	예	예	`@Serializable` 기반 복잡한 타입 상태 저장

현재 `SearchScreen` 은 `sheetOpen` 과 `detailSheetAddress` 를 `rememberSaveable` 로 관리한다. 이건 "순수 UI 상태지만, 화면 재생성에도 살아남아야 한다"는 판단이다.

2. 상태 끌어올리기(State Hoisting)

공식 state hoisting 가이드는 **상태는 읽고 쓰는 모든 composable의 가장 낮은 공통 조상으로 올리라**고 설명한다. 그리고 business logic이 끼면 그 공통 조상은 Composition 바깥, 즉 ViewModel이 될 수 있다.

- `query`, `selectedSido`, `phase`는 business logic이 있으므로 ViewModel 쪽 상태다.
- `sheetOpen`, `detailSheetAddress`는 순수 화면 표시 상태라 화면 내부에 둔다.
- 즉, 이 파일은 state hoisting의 실전 예제다.

3. Side effect와 `LaunchedEffect`

공식 side-effects 문서는 composable은 ideally side-effect free여야 한다고 설명한다. 하지만 UI 수명주기에 맞춰 한 번 실행해야 하는 작업은 effect API로 다뤄야 한다.

- `rememberAutoFocusKeyboard()` 안의 `LaunchedEffect(Unit)`는 입력창 포커스와 키보드 표시를 "조합 완료 후" 실행한다.
- 그냥 함수 본문에서 바로 키보드를 띄우면 recomposition 타이밍과 꼬일 수 있다.

4. Flow 수집과 `collectAsStateWithLifecycle()`

공식 Compose state 문서는 Android 앱에서 Flow를 Compose State로 읽을 때 `collectAsStateWithLifecycle()`를 권장한다. 이유는 lifecycle-aware collection이기 때문이다.

- 화면이 안 보일 때 불필요한 수집을 줄인다.
- Compose가 자동으로 최신 값을 State로 노출한다.
- `SearchScreen`은 `uiState`, `favorites`, `recent`를 모두 이 방식으로 읽는다.

5. SearchScreen 상단부 해설

```
61 // SidoAnchor와 검색바가 같은 radius를 공유
62 private val SEARCH_BAR_RADIUS = 14.dp

68 @Composable
69 private fun rememberAutoFocusKeyboard(): Pair<FocusRequester, SoftwareKeyboardController?> {
70     val keyboard = LocalSoftwareKeyboardController.current
71     val focus = remember { FocusRequester() }
72     LaunchedEffect(Unit) {
73         focus.requestFocus()
74         keyboard?.show()
75     }
76     return focus to keyboard
77 }
```

61-62행: 디자인 토큰 공유

`SEARCH_BAR_RADIUS`는 단순 숫자가 아니다. 화면과 컴포넌트 사이에 시각 규칙을 공유하는 디자인 토큰 역할을 한다. `SidoAnchor`와 같은 radius를 써서 한 줄에서 "하나의 조합 컴포넌트"처럼 읽히게 한다.

69-77행: `rememberAutoFocusKeyboard()`

줄	코드	의미
69	함수 반환 타입	`FocusRequester`와 키보드 컨트롤러를 함께 반환한다. 즉, 이 함수는 "자동 포커스/키보드용 hook"이다.
70	`LocalSoftwareKeyboardController.current`	현재 Compose 트리에서 소프트 키보드를 제어할 수 있는 컨트롤러를 가져온다.
71	`remember { FocusRequester() }`	recomposition 때마다 새 FocusRequester를 만들지 않게 기억한다.
72-75	`LaunchedEffect(Unit)`	composition에 들어온 뒤 한 번 코루틴을 실행해 포커스를 요청하고 키보드를 띄운다.
76	Pair 반환	호출부에서 입력창 연결용 focus와 hide/show 용 keyboard를 같이 받게 한다.

> **여기서 배울 핵심**
> Compose에서도 "라이프사이클에 맞춘 일회성 작업"이 필요하다. 그때 그냥 일반 함수처럼 하지 않고 `LaunchedEffect` 같은 effect API를 쓴다.

6. SearchScreen 본체 해설

```
86 @OptIn(ExperimentalMaterial3Api::class)
87 @Composable
88 fun SearchScreen(viewModel: SearchViewModel = hiltViewModel()) {
89     val state by viewModel.uiState.collectAsStateWithLifecycle()
90     val favorites by viewModel.favorites.collectAsStateWithLifecycle()
91     val recent by viewModel.recent.collectAsStateWithLifecycle()
92     val context = LocalContext.current
93     val view = LocalView.current
94     val (focus, keyboard) = rememberAutoFocusKeyboard()
95     val copyToastTemplate = stringResource(R.string.copy_toast)
96
97     var sheetOpen by rememberSaveable { mutableStateOf(false) }
98     var detailSheetAddress by rememberSaveable { mutableStateOf<Address?>(null) }
99
100    val callbacks = rememberSearchCallbacks(...)
101
118    Scaffold(...) { padding ->
119        SearchScreenContent(...)
120    }
121
```

```
131     SearchOverlays(...)
132 }
```

88행: ViewModel 기본값에 `hiltViewModel()`

이 화면은 기본적으로 Hilt가 제공하는 `SearchViewModel`을 사용한다. 즉, 화면은 ViewModel을 직접 생성하지 않는다. 이건 DI와 state hoisting이 만나는 지점이다.

89-91행: ViewModel 상태 수집

- `state` : 검색 화면의 단일 진실 상태
- `favorites` : 즐겨찾기 리스트
- `recent` : 최근 본 주소 리스트

여기서 중요한 건 Flow를 그대로 쓰지 않고 Compose `State`로 변환한다는 점이다. 그래야 값이 바뀔 때 화면이 자동으로 재구성된다.

92-95행: Compose Local과 리소스

- `LocalContext.current` : Android `Context` 접근
- `LocalView.current` : 현재 Android View 접근
- `stringResource(...)` : 리소스 문자열 읽기
- `rememberAutoFocusKeyboard()` : 위에서 만든 화면용 hook 사용

즉, Compose는 순수 선언형이지만 필요할 때 Android platform 객체와 연결할 수 있는 "Local" 진입점을 제공한다.

97-98행: 왜 `rememberSaveable`인가

`sheetOpen`과 `detailSheetAddress`는 ViewModel 상태가 아니라 순수 UI 상태다. 하지만 회전이나 Activity 재생성에도 보존되면 좋다. 그래서 `remember`가 아니라 `rememberSaveable`를 썼다.

- `sheetOpen` : 상세/시도 시트가 열렸는지 여부
- `detailSheetAddress` : 현재 상세 시트에 띄울 주소

> 공식 최신 Compose 문서를 보면 `rememberSerializable`도 새 선택지다. 하지만 현재 코드는 `Address`가 `Parcelable`이라 `rememberSaveable`만으로도 충분히 성립한다.

100행: 왜 callbacks를 묶는가

화면이 자식 composable들에게 넘겨야 할 함수가 많아지면 시그니처가 급격히 지저분해진다. 그래서 `SearchCallbacks`라는 data class로 한 번 묶었다. 이건 Compose 고유 기술이라기보다 "큰 함수의 의존성을 관리하는 일반적인 코드 정리 전략"이다.

118-120행: `Scaffold`

Material3의 화면 뼈대 컴포넌트다. 상단 바, 하단 바, FAB, 본문 영역 같은 화면 골격을 관리한다. 여기서는 상단 바와 본문 패딩 전달에 사용한다.

131행 이후: 오버레이를 본문 밖으로 분리

`SearchOverlays()`를 Scaffold 본문 바깥에 두어 시트 렌더링 책임을 분리했다. 본문 길이를 줄이고, "메인 화면"과 "모달 레이어"를 분리해서 읽을 수 있게 만든다.

7. SearchOverlays와 SearchScreenContent

```
145 @Composable
146 private fun SearchOverlays(...) {
154     if (sheetOpen) {
155         SidoSelectorSheet(...)
160     }
161     detailAddress?.let { addr ->
162         DetailSheet(...)
167     }
168 }

170 @Composable
171 private fun SearchScreenContent(...) {
179     Column(...)
184         Row(...) {
188             SidoAnchor(...)
189             SearchBar(...)
196         }
199         Box(modifier = Modifier.weight(1f)) {
200             SearchBody(...)
207         }
208         AdBanner()
209     }
210 }
```

SearchOverlays: 모달 레이어 분리

- `sheetOpen`이 true면 시·도 선택 바텀시트를 띄운다.
- `detailAddress`가 null이 아니면 상세 시트를 띄운다.
- 즉, 이 함수는 "어떤 오버레이가 지금 보일지"만 담당한다.

SearchScreenContent: 화면 본문 조립

- `Column` : 세로 방향 레이아웃
- `Row` : 시·도 선택 앵커 + 검색바를 한 줄에 둔다

- `Box(weight = 1f)` : 본문이 남은 공간을 차지한다
- `AdBanner()` : 광고를 하단에 고정한다

이 구조는 "상단 검색 조작부 / 가변 본문 / 하단 광고"라는 화면 리듬을 만든다. Compose에선 이런 구조를 View tree 대신 함수 호출 계층으로 표현한다.

weight(1f)를 왜 쓰는가

코드 주석에도 적혀 있듯이, `SearchBody` 쪽 일부 상태 화면은 `fillMaxSize()` 를 쓸 수 있다. 그래서 본문 영역을 `Box(modifier = Modifier.weight(1f))` 로 감싸 두어 마지막 `AdBanner()` 가 아래로 밀려나지 않게 만든다.

8. rememberSearchCallbacks와 SearchBar

```
212 @Composable
213 private fun rememberSearchCallbacks(...) : SearchCallbacks =
222     remember(...) {
223         SearchCallbacks(
224             onQueryChange = viewModel::onQueryChange,
225             onSubmit = {
226                 viewModel.searchNow()
227                 onSubmitExtra()
228             },
229             onCopyAddress = { label, text ->
230                 context.copyToClipboard(label, text)
231                 view.lightHaptic()
232                 if (Build.VERSION.SDK_INT < Build.VERSION_CODES.TIRAMISU) {
233                     Toast.makeText(...).show()
234                 }
235             },
236             ...
241         )
242     }
```

왜 `remember(...)` 로 콜백 객체를 묶는가

이 함수는 단순히 보기 좋게 묶는 것 이상으로, recomposition 때마다 새 callback 묶음을 무조건 다시 만들지 않게 해준다. 다만 가장 중요한 목적은 여전히 **가독성과 인자 폭발 방지**다.

onCopyAddress 안에서 하는 일

- 클립보드 복사
- 햅틱 피드백
- Android 13 미만에서만 수동 토스트 표시

플랫폼 버전 차이를 UI 이벤트 처리에서 직접 흡수하는 예제다.

SearchBar 해설

```
272 @Composable
273 private fun SearchBar(...) {
280     OutlinedTextField(
281         value = query,
282         onValueChange = onQueryChange,
283         placeholder = { Text(stringResource(...)) },
284         singleLine = true,
286         shape = RoundedCornerShape(SEARCH_BAR_RADIUS),
287         modifier = modifier.focusRequester(focus),
288         keyboardOptions = KeyboardOptions(imeAction = ImeAction.Search),
290         keyboardActions = KeyboardActions(onSearch = { onSearch() }),
292         trailingIcon = {
293             if (query.isNotEmpty()) {
294                 IconButton(onClick = { onQueryChange("") }) { ... }
300             }
301         },
302     )
303 }
```

코드 포인트	의미
`value / onValueChange`	Compose의 기본적인 state hoisting 패턴이다. TextField가 자기 상태를 직접 가지지 않고, 바깥 상태를 받고 이벤트만 올린다.
`placeholder`	로컬 문자열 리소스로 힌트 텍스트를 넣는다.
`singleLine`	검색 입력창이 여러 줄로 늘어나지 않게 한다.
`shape`	`SidoAnchor` 와 같은 radius를 써서 한 줄에서 시각적 통일감을 만든다.
`focusRequester`	앞에서 만든 `FocusRequester` 와 연결해 자동 포커스를 받게 한다.
`ImeAction.Search`	키보드 action 버튼을 검색 모양으로 바꾼다.
`KeyboardActions(onSearch = ...)`	사용자가 키보드 검색 버튼을 누르면 debounce를 기다리지 않고 즉시 검색을 실행한다.
`trailingIcon`	입력값이 있을 때만 X 버튼을 보여준다. UI 노이즈를 줄이는 작은 결정이다.

9. SearchBody와 SearchUiState를 함께 보기

`SearchBody()`를 제대로 이해하려면 `SearchUiState.Phase`를 같이 봐야 한다. 이 둘은 사실상 한 세트다.

```
// SearchUiState.kt
sealed interface Phase {
    data object Idle : Phase
    data object Loading : Phase
    data class Success(...) : Phase
    data object Empty : Phase
    data object PostalCodeUnsupported : Phase
    data class Error(val error: AppError) : Phase
}
```

```
// SearchScreen.kt 일부
327 when (phase) {
328     SearchUiState.Phase.Idle -> ...
342     SearchUiState.Phase.Loading -> ...
350     SearchUiState.Phase.Empty -> ...
355     SearchUiState.Phase.PostalCodeUnsupported -> ...
360     is SearchUiState.Phase.Error -> ...
365     is SearchUiState.Phase.Success -> ...
}
```

이 구조를 뭐라고 봐야 하나

이건 거의 상태 머신이다. 즉, "화면은 지금 어떤 phase인가?"를 하나의 타입으로 명시하고, 그 타입에 따라 어떤 UI를 그릴지 `when`으로 분기한다.

왜 좋은가

- 현재 화면 상태가 무엇인지 명확하다.
- 불가능한 상태 조합을 줄이기 쉽다.
- UI 테스트, ViewModel 테스트, 화면 설계가 같은 언어를 쓰게 된다.

SearchBody에서 보이는 UI 철학

- `Idle`: 첫 화면 또는 최근/즐겨찾기 보여주기
- `Loading`: 큰 스켈레톤 대신 작은 스피너로 노이즈 줄이기
- `Empty`: 검색 결과 없음
- `PostalCodeUnsupported`: 우편번호 검색은 별도 안내
- `Error`: 사용자 친화 메시지 + 재시도
- `Success`: 결과 리스트 + 무한 스크롤

> **여기서 가장 중요한 연결**
> ViewModel이 `Phase`를 만든다. 화면은 그 `Phase`를 보고 그리기만 한다. 즉, UI와 business logic의 분리가 타입 구조로 드러난다.

10. Theme와 재사용 컴포넌트까지 연결하기

`ZipkrTheme`

`MainActivity`에서 `ZipkrTheme { SearchScreen() }`로 감싼다는 것은, 이 화면이 Material3 theme 체계 안에서 동작한다는 뜻이다.

- light/dark theme 자동 추종
- dynamic color는 기본 꺼짐
- brand palette를 우선 적용

`SidoAnchor`

이 컴포넌트는 검색 화면 전체가 아니라 "검색바 옆 지역 선택 버튼"만 담당한다. 즉, 루트 화면은 조립하고, 세부 시각 표현은 재사용 컴포넌트가 맡는다.

- 선택 유무에 따라 color/border가 바뀐다.
- search bar와 height/radius 패턴을 공유한다.
- 컴포넌트 자체는 state를 많이 갖지 않고, `selected`와 `onOpen`만 받는다.

> **이 부분에서 배울 설계 감각**
> 화면은 조립과 연결을 하고, 컴포넌트는 표현을 한다. 이 분리가 잘 되면 파일이 커져도 읽을 수 있다.

11. 최신 공식 흐름과 현재 코드의 관계

주제	현재 zipkr 코드	2026-05 공식 흐름과 해석
Flow 수집	`collectAsStateWithLifecycle()` 사용	공식 권장과 맞는다. Android 앱에서 lifecycle-aware Flow collect의 권장 방식이다.
상태 보존	`remember`, `rememberSaveable` 사용	공식 state lifespans 문서와 잘 맞는다. 최신 문서에는 `rememberSerializable`와 `retain`도 추가로 나온다.
state hoisting	business state는 ViewModel, UI-only state는 화면 내부	공식 state hoisting 가이드의 좋은 사례다.

주제	현재 zipkr 코드	2026-05 공식 흐름과 해석
side effects	`LaunchedEffect(Unit)`로 키보드/포커스 처리	공식 side-effects 가이드와 맞는다. composable body에서 직접 부작용을 실행하지 않는다.
화면 구조	Scaffold + 작은 composable 분해	현대 Compose 앱의 전형적인 구조다. 한 함수에 전부 몰아넣지 않는다.

12. 직접 써보는 체크포인트

Q1. 왜 `sheetOpen`은 ViewModel이 아니라 `rememberSaveable`인가?

Q2. 왜 `uiState`는 `remember`로 만들지 않고 ViewModel에서 가져오는가?

Q3. `LaunchedEffect(Unit)`를 그냥 함수 본문 코드로 바꾸면 어떤 문제가 생길 수 있는가?

Q4. `value / onValueChange` 패턴이 왜 state hoisting의 예시인가?

Q5. `SearchBody()`와 `SearchUiState.Phase`를 함께 볼 때, 왜 이 구조를 상태 머신처럼 볼 수 있는가?

13. 다음 단계 예고

Phase 2 다음에는 자연스럽게 ViewModel 쪽으로 내려가야 한다.

1. `SearchViewModel.kt`에서 debounce, explicit search, loadMore, cancellation을 본다.
2. `SearchUiState.kt`를 다시 가져와 상태 전이를 연결해 본다.
3. 그다음 Repository와 네트워크 흐름으로 간다.

> **다음 문서 후보**
> `Phase 3 상태 관리 & SearchViewModel 워크북`
> 대상: `SearchViewModel.kt`, `SearchUiState.kt`, 일부 테스트 코드

14. 공식 참고 자료

- State and Jetpack Compose
- State lifespans in Compose
- Where to hoist state
- Side-effects in Compose
- State in Jetpack Compose codelab
- Set up Edge-to-edge

Compose는 빠르게 진화한다. 특히 상태 수명 관련 API인 `retain`, `rememberSerializable`처럼 최근 문서에 강화된 개념은 이후 실제 코드 적용 시 다시 확인하는 것이 좋다.

PART 4

Android Phase 3 SearchViewModel Workbook

PHASE 3 DETAILED WORKBOOK

상태 관리 워크북 SearchViewModel.kt 중심

`SearchViewModel.kt`를 기준으로 ViewModel, StateFlow, coroutines, debounce, cancellation, stateIn, 화면 상태 전이, 테스트(Turbine/MockK)까지 같이 배우는 인쇄용 상세 자료.

학습 목표
왜 이 로직이 Composable이 아니라 ViewModel에 있어야 하는지 설명할 수 있어야 한다.

대상 파일
`SearchViewModel.kt`, `SearchUiState.kt`, `SearchViewModelTest.kt`

핵심 개념

읽는 방식

ViewModel, StateFlow, viewModelScope, debounce, stateIn, UDF, cancellation	상태 정의와 상태 전이를 같이 보고, 테스트가 무엇을 증명하는지도 함께 본다.

작성일: 2026-05-06. ViewModel, StateFlow/SharedFlow, coroutines, UI state holder 관련 Android Developers 공식 문서를 반영했다.

1. 왜 ViewModel이 필요한가

Android Developers의 최신 ViewModel overview는 ViewModel을 **screen-level state holder**이자 business logic holder로 설명한다. 핵심은 화면이 사라졌다가 다시 만들어져도 UI state를 유지하고, 화면 관련 로직을 UI 함수 밖으로 분리한다는 점이다.

> **핵심 문장**
> Composable은 화면을 그린다. ViewModel은 화면 상태를 생산한다.

- 회전 등 configuration change 이후에도 상태 유지
- 비동기 작업을 UI에서 분리
- 테스트 가능한 screen logic 위치 제공
- UI와 데이터 계층 사이의 완충 지점

2. SearchViewModel이 하는 일

책임	설명
입력 상태 보유	`query`, `selectedSido`, `phase`가 들어 있는 `SearchUiState`를 유지한다.
자동 검색 트리거	`queryFlow.debounce(700ms)`를 통해 사용자의 입력을 일정 시간 기다렸다가 검색한다.
명시적 검색 처리	`searchNow()`로 키보드 검색 버튼/재시도 버튼의 즉시 검색을 처리한다.
시·도 필터 반영	`Sido`가 바뀌면 effective query를 다시 만들어 재검색한다.
페이징 처리	`loadMore()`로 다음 페이지를 누적 로드한다.
중복 호출 방지	`lastTriggeredQuery`, `inFlight`, `loadMoreJob`으로 debounce/명시적 검색/페이징 충돌을 막는다.
최근/즐겨찾기 상태 노출	DataStore Flow를 `stateIn`으로 hot state로 바꾸어 Compose에 제공한다.

3. ViewModel 전체 구조 먼저 보기

```
@HiltViewModel
class SearchViewModel @Inject constructor(
    private val repository: AddressRepository,
    private val historyRepository: SearchHistoryRepository,
) : ViewModel() {
    private val _uiState = MutableStateFlow(SearchUiState())
    val uiState: StateFlow<SearchUiState> = _uiState.asStateFlow()

    val favorites = historyRepository.observeFavorites().stateIn(...)
    val recent = historyRepository.observeRecent().stateIn(...)

    private val queryFlow = MutableStateFlow("")
    private var inFlight: Job? = null
    private var loadMoreJob: Job? = null
    private var lastTriggeredQuery: String? = null

    init { ... debounce pipeline ... }

    fun onQueryChange(query: String) { ... }
    fun searchNow() { ... }
    fun onSidoChange(sido: Sido?) { ... }
    fun loadMore() { ... }

    private fun runSearch(query: String) { ... }
    private fun mapLoadMore(...) { ... }
    private fun mapFirstPage(...) { ... }
    private fun effectiveQuery(...) { ... }
}
```

> **이 구조에서 먼저 보이는 것**
> 상태 정의, 이벤트 함수, 비공개 검색 실행 함수, 결과 매핑 함수가 명확히 분리돼 있다. 즉, "입력", "실행", "결과 변환"이 뒤섞이지 않는다.

4. StateFlow와 UDF 관점

Android Developers의 `StateFlow and SharedFlow` 문서는 `StateFlow`를 현재 상태와 새 상태를 내보내는 state-holder observable flow로 설명한다.

- `MutableStateFlow`는 내부에서 상태를 바꾼다.
- `StateFlow`는 외부에 읽기 전용으로 노출한다.
- UI는 `uiState`를 읽고, 이벤트 함수로만 상태 변화를 요청한다.

```
private val _uiState = MutableStateFlow(SearchUiState())
val uiState: StateFlow<SearchUiState> = _uiState.asStateFlow()
```

왜 이 패턴이 중요한가

- UI가 상태를 직접 덮어쓸 수 없다.
- 상태 변경 경로가 이벤트 함수로 제한된다.
- UDF(단방향 데이터 흐름)가 유지된다.

> UI가 `MutableStateFlow`를 직접 받으면 구조가 무너지기 쉽다. 읽는 쪽과 쓰는 쪽을 분리하는 것이 핵심이다.

5. favorites / recent와 `stateIn`

```
val favorites: StateFlow<List<Address>> =
    historyRepository
        .observeFavorites()
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(STOP_TIMEOUT_MS), emptyList())

val recent: StateFlow<List<Address>> =
    historyRepository
        .observeRecent()
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(STOP_TIMEOUT_MS), emptyList())
```

이 코드가 하는 일

- DataStore에서 오는 cold Flow를 화면에서 쓰기 좋은 hot state로 바꾼다.
- 초기값을 `emptyList()`로 명시한다.
- `WhileSubscribed(STOP_TIMEOUT_MS)`로 짧은 구독 해제 동안 upstream을 유지해 재구독 비용을 줄인다.

공식 StateFlow 문서는 cold flow를 `stateIn`으로 hot state로 바꿀 수 있다고 설명한다. 현재 코드의 주석도 같은 의도를 말한다. Composable이 여러 번 collect하더라도 DataStore를 매번 새로 읽지 않도록 하려는 것이다.

왜 ViewModel에서 하는가

UI는 "즐겨찾기 리스트가 지금 무엇인가"만 알면 된다. DataStore가 Flow를 어떻게 내보내는지, 그것을 언제 hot state로 바꿔야 하는지는 ViewModel 책임이다.

6. queryFlow와 debounce 파이프라인

```
init {
    viewModelScope.launch {
        queryFlow
            .debounce(DEBOUNCE_MS)
            .distinctUntilChanged()
            .filter { it.length >= MIN_QUERY_LEN }
            .filter { effectiveQuery(it, _uiState.value.selectedSido) != lastTriggeredQuery }
            .onEach { runSearch(it) }
            .collect { /* no-op */ }
    }
}
```

각 연산자의 의미

연산자	의미
`debounce(700ms)`	사용자가 타이핑을 멈춘 뒤 700ms 기다렸다가 다음 단계로 넘긴다. 한국어 조합 입력과 과도한 API 호출을 함께 고려한 값이다.
`distinctUntilChanged()`	같은 query가 연속으로 들어오면 중복 실행하지 않는다.
`filter { it.length >= 2 }`	한 글자 이하의 너무 짧은 검색은 막는다.
`filter { effectiveQuery(...) != lastTriggeredQuery }`	명시적 검색과 debounce 검색이 같은 effective query로 겹칠 때 중복 호출을 막는다.
`onEach { runSearch(it) }`	실제 검색 실행 함수를 호출한다.

왜 `effectiveQuery`로 비교하나

사용자가 같은 base query를 입력해도 `Sido`가 바뀌면 실제 API 호출 query는 달라진다. 예를 들어 `"강남"`과 `"서울특별시 강남"`은 다르다. 그래서 중복 차단 기준도 base query가 아니라 effective query여야 한다.

7. 이벤트 함수들 해설

`onQueryChange(query)`

```
fun onQueryChange(query: String) {
    _uiState.value = _uiState.value.copy(query = query)
    queryFlow.value = query
    if (query.length < MIN_QUERY_LEN) {
        _uiState.value = _uiState.value.copy(phase = SearchUiState.Phase.Idle)
```

```
    }
}
```

- UI state에 즉시 query를 반영한다.
- debounce 파이프라인용 `queryFlow`에도 값을 밀어 넣는다.
- 2글자 미만이면 API 호출 이전에 화면을 Idle로 돌려놓는다.

`searchNow()`

```
fun searchNow() {
    lastTriggeredQuery = null
    runSearch(_uiState.value.query)
}
```

- 사용자가 키보드 검색 버튼이나 재시도 버튼을 눌렀을 때 즉시 실행한다.
- `lastTriggeredQuery`를 비워 같은 query여도 강제 실행 가능하게 만든다.

`onSidoChange(sido)`

```
fun onSidoChange(sido: Sido?) {
    if (sido == _uiState.value.selectedSido) return
    _uiState.value = _uiState.value.copy(selectedSido = sido)
    if (_uiState.value.query.length >= MIN_QUERY_LEN) {
        runSearch(_uiState.value.query)
    }
}
```

- 같은 칩을 다시 누르면 의미 변화가 없으므로 return한다.
- 선택 시·도가 바뀌면 현재 query를 기준으로 즉시 재검색한다.
- debounce를 기다리지 않는 이유는 "필터 변경"이 명시적 사용자 액션이기 때문이다.

8. loadMore와 페이징 처리

```
fun loadMore() {
    val current = _uiState.value.phase as? SearchUiState.Phase.Success ?: return
    val canLoad =
        current.hasNext &&
        !current.isLoadingMore &&
        query.isNotBlank() &&
        query.length >= MIN_QUERY_LEN
    if (!canLoad) return

    _uiState.value = _uiState.value.copy(
        phase = current.copy(isLoadingMore = true, loadMoreError = null),
    )

    val effective = effectiveQuery(query, _uiState.value.selectedSido)
    loadMoreJob = viewModelScope.launch {
        val result = repository.search(effective, page = current.currentPage + 1, pageSize = PAGE_SIZE)
        val nextPhase = mapLoadMore(result, current)
        _uiState.value = _uiState.value.copy(phase = nextPhase)
    }
}
```

핵심 포인트

- `Success` 상태일 때만 가능하다.
- 이미 로딩 중이면 중복 호출을 막는다.
- 다음 페이지가 없으면 호출하지 않는다.
- 페이지 2 이상도 동일한 `effectiveQuery`를 유지한다.

`mapLoadMore()`가 하는 일

- 성공 시: 기존 결과 + 새 페이지 결과를 누적한다.
- 실패 시: 기존 결과는 유지하고 `loadMoreError`만 채운다.

> **좋은 UX 포인트**
> 다음 페이지 로딩 실패가 전체 검색 실패와 같지 않다는 점을 타입에 반영했다. 사용자가 이미 보고 있던 결과를 지우지 않는 것이 중요하다.

9. `runSearch()`는 왜 이렇게 생겼는가

```
private fun runSearch(query: String) {
    inFlight?.cancel()
    loadMoreJob?.cancel()
    if (query.isBlank()) {
        _uiState.value = _uiState.value.copy(phase = Idle)
        return
    }
    if (query.matches(POSTAL_CODE_PATTERN)) {
        lastTriggeredQuery = query
```

```
            _uiState.value = _uiState.value.copy(phase = PostalCodeUnsupported)
            return
        }
        val effective = effectiveQuery(query, _uiState.value.selectedSido)
        lastTriggeredQuery = effective
        _uiState.value = _uiState.value.copy(phase = Loading)
        inFlight = viewModelScope.launch {
            val result = repository.search(effective, page = 1, pageSize = PAGE_SIZE)
            _uiState.value = _uiState.value.copy(phase = mapFirstPage(result))
        }
    }
```

왜 취소부터 하나

공식 coroutine 가이드는 `viewModelScope`를 screen-level async work에 권장한다. 이 코드에서 중요한 것은 scope 자체보다 **이전 검색과 다음 검색이 겹칠 수 있다는 현실**이다.

- `inFlight?.cancel()` : 이전 첫 페이지 검색 취소
- `loadMoreJob?.cancel()` : 진행 중이던 추가 페이지 검색도 취소

이렇게 해야 오래 걸린 이전 요청 결과가 나중에 돌아와 현재 화면을 덮어쓰는 stale overwrite를 막을 수 있다.

우편번호 5자리 처리

`query.matches(POSTAL_CODE_PATTERN)`에 걸리면 API를 호출하지 않고 바로 `PostalCodeUnsupported` phase로 전환한다. 즉, 실패를 네트워크 결과로 기다리지 않고 intent-aware guard를 선행 적용했다.

`mapFirstPage()`

- 결과가 비어 있으면 `Empty`
- 결과가 있으면 `Success`
- 실패면 `Error(AppError)`

중요한 점은 ViewModel이 `Result<AddressPage>`를 그대로 UI에 넘기지 않고, UI가 바로 그릴 수 있는 phase로 바꾼다는 것이다.

10. SearchUiState와 함께 봐야 하는 이유

```
data class SearchUiState(
    val query: String = "",
    val selectedSido: Sido? = null,
    val phase: Phase = Phase.Idle,
)
```

ViewModel은 결국 `SearchUiState`를 생산하는 기계다. 이 파일의 진짜 목적은 "검색 로직 실행"이 아니라 "UI가 그릴 수 있는 상태를 올바르게 생산"하는 것이다.

지금 구조가 좋은 이유

- 검색어와 phase가 같은 객체 안에 있어 화면의 전체 상태를 한 번에 설명할 수 있다.
- `selectedSido = null`을 "전체"로 쓰는 선택도 불필요한 sentinel enum을 피한다.
- `Success` 안의 `isLoadingMore`, `loadMoreError`는 2차 로딩 UX를 따로 모델링한다.

11. 테스트는 무엇을 증명하나

`SearchViewModelTest.kt`는 단순히 "코드가 돌아간다"를 확인하는 것이 아니라, ViewModel이 생산하는 상태 전이가 기대대로 나오는지를 고정한다.

테스트 주제	무엇을 보호하는가
초기 상태는 Idle	ViewModel 생성 직후 화면이 예상 가능한 기본 상태에서 시작함을 보장
검색 결과가 있으면 Success	repository 성공 결과가 올바른 화면 상태로 매핑됨을 보장
결과가 비면 Empty	빈 결과를 성공과 혼동하지 않도록 보장
네트워크 실패면 Error	예외가 AppError/Phase.Error로 보존됨을 보장
5자리 숫자면 PostalCodeUnsupported	guard 로직이 API 호출 없이 먼저 동작함을 보장
loadMore 누적	페이지 2 이상 결과를 이어붙이는 로직 보호

왜 Turbine을 쓰나

Android Developers 공식 문서도 StateFlow/stateIn 테스트에 추가 주의가 필요하다고 말한다. Turbine은 Flow를 채널처럼 다루면서 emit 순서와 값을 안정적으로 검증하게 해준다.

왜 UnconfinedTestDispatcher를 쓰나

테스트에서 `viewModelScope`와 coroutine 실행을 통제하기 위해 메인 디스패처를 교체한다. 즉, 비동기 로직을 "테스트 가능한 동기 흐름처럼" 다루려는 의도다.

12. 최신 공식 흐름과 현재 코드의 관계

주제	현재 zipkr	2026-05 공식 흐름과 해석
ViewModel	`@HiltViewModel` + screen-level state holder	공식 ViewModel overview 및 UI state holder 가이드와 매우 잘 맞는다.

주제	현재 zipkr	2026-05 공식 흐름과 해석
StateFlow	내부는 `MutableStateFlow`, 외부는 `StateFlow`	공식 StateFlow 가이드의 전형적인 패턴이다.
Flow 수집	UI는 `collectAsStateWithLifecycle()`, VM은 `stateIn()`	현대 Android 권장 방식과 맞는다.
coroutines	`viewModelScope.launch` 사용	공식 coroutine 가이드의 기본 패턴이다.
UDF	상태는 내려가고 이벤트는 올라간다	공식 app architecture의 단방향 데이터 흐름 원칙과 맞는다.

13. 직접 적어보기

Q1. 왜 `queryFlow`와 `uiState.query`가 둘 다 필요한가?

Q2. `lastTriggeredQuery`가 없다면 어떤 중복 호출이 생길 수 있는가?

Q3. 왜 `loadMore()` 실패가 전체 화면 `Error`로 가지 않고 `loadMoreError`로만 남는가?

Q4. 왜 Composable에서 직접 debounce 검색을 구현하지 않고 ViewModel에서 하는가?

Q5. 이 ViewModel이 UDF를 어떻게 지키는지 설명해보라.

14. 다음 단계 예고

다음은 data 계층을 볼 차례다. 여기서부터는 "화면이 어떤 데이터를 먹는가"보다 "앱이 어떤 언어로 문제를 표현하는가"가 중요해진다.

1. `Address`, `Result`, `AppError`, `AddressPage`
2. `AddressRepository`와 `AddressRepositoryImpl`
3. `SearchHistoryRepository`와 `DataStoreSearchHistoryRepository`

15. 공식 참고 자료

- ViewModel overview
- State holders and UI state
- Kotlin coroutines on Android
- StateFlow and SharedFlow
- Guide to app architecture
- Recommendations for Android architecture

stateIn/shareIn, lifecycle-aware collect, coroutine dispatcher 전략은 실제 앱이 커질수록 더 중요해진다. 지금 이 ViewModel은 작은 앱에서 현대 Android 패턴을 꽤 잘 보여주는 예제다.

PART 5

Android Phase 4 Domain Repository Workbook

PHASE 4 DETAILED WORKBOOK

도메인 & Repository 워크북
data 패키지 중심

`zipkr`의 `data/` 패키지를 기준으로 도메인 모델, 결과 타입, 에러 타입, Repository 패턴, Provider 경계, DataStore, in-memory cache까지 한 번에 이해하는 인쇄용 상세 자료.

학습 목표
UI가 왜 데이터를 직접 다루지 않고 Repository와 도메인 모델을 거쳐야 하는지 설명할 수 있어야 한다.

대상 파일
`data/*`, `data/provider/AddressProvider.kt`, 일부 테스트

핵심 개념
도메인 모델, DTO 분리, repository, provider, typed error, DataStore, cache

읽는 방식
모델 -> repository 계약 -> 구현 -> 저장소/캐시 -> 테스트 순으로 본다.

작성일: 2026-05-06. Android app architecture, data layer, DataStore 공식 문서를 반영했다.

1. data 패키지는 왜 중요한가

UI는 결국 "어떤 데이터"를 받아 그리는 계층이다. 그런데 그 데이터를 어떤 이름과 어떤 규칙으로 다룰지 정하는 곳이 바로 `data` 패키지다. 여기서 잘못 설계하면 UI, 네트워크, 저장소가 전부 섞인다.

> **핵심 문장**
> `data` 패키지는 앱이 세상을 어떤 언어로 표현하는지 정의하는 곳이다.

공식 아키텍처 권장 사항

Android Developers의 최신 아키텍처 가이드는 최소한 두 계층을 권장한다.

- UI layer: 화면에 데이터를 보여주고 사용자 입력을 받는다.
- Data layer: 앱의 비즈니스 로직 대부분을 포함하고, 앱 데이터를 노출한다.

그리고 권장 사항 페이지는 **single data source만 있더라도 repository를 만들라**고 분명히 말한다. `zipkr`는 바로 그 구조를 따르고 있다.

2. 이 패키지의 큰 구조

```
data/
├─ Address.kt
├─ Coordinate.kt
├─ Sido.kt
├─ AddressPage.kt
├─ Result.kt
├─ AppError.kt
├─ AddressRepository.kt
├─ AddressRepositoryImpl.kt
├─ CoordinateRepository.kt
├─ CoordinateRepositoryImpl.kt
├─ SearchHistoryRepository.kt
├─ DataStoreSearchHistoryRepository.kt
└─ provider/
   └─ AddressProvider.kt
```

분류	무엇을 담당하는가
도메인 모델	`Address`, `Coordinate`, `Sido`, `AddressPage`처럼 앱 내부에서 믿고 쓰는 데이터 구조
결과/에러 타입	`Result`, `AppError`처럼 성공/실패와 예외 의미를 표준화하는 타입
계약 인터페이스	`AddressRepository`, `CoordinateRepository`, `SearchHistoryRepository`, `AddressProvider`
구현체	`AddressRepositoryImpl`, `CoordinateRepositoryImpl`, `DataStoreSearchHistoryRepository`

3. 도메인 모델이란 무엇인가

도메인 모델은 외부 API의 응답 형식이 아니라, **앱이 실제로 다루고 싶은 의미 있는 데이터 구조**다. 예를 들어 행안부 API가 어떤 필드명으로 주소를 주든, UI는 결국 `Address`라는 앱 언어만 보면 된다.

`Address.kt`

```
@Parcelize
@Serializable
data class Address(
    val zipCode: String,
    val roadAddress: String,
    val jibunAddress: String,
    val englishAddress: String,
    val buildingName: String,
    val sido: String,
    val sigungu: String,
    val eupmyeondong: String,
) : Parcelable {
    val stableKey: String
        get() = "$zipCode|$roadAddress|$buildingName"
}
```

왜 좋은 모델인가

- UI가 필요한 필드만 담고 있다.
- `@Parcelize` 덕분에 화면 간/시트 상태 보존에 넣기 쉽다.
- `@Serializable` 덕분에 DataStore JSON 저장에 쓸 수 있다.
- `stableKey`로 중복 제거 기준을 모델 안에 캡슐화했다.

Coordinate.kt

```
@Parcelize
data class Coordinate(
    val longitude: Double,
    val latitude: Double,
) : Parcelable
```

작은 모델이지만 좌표 의미를 분명하게 드러낸다. 단순한 `Pair<Double, Double>`보다 훨씬 읽기 좋다.

AddressPage.kt

```
data class AddressPage(
    val items: List<Address>,
    val currentPage: Int,
    val totalCount: Int,
) {
    fun hasNext(pageSize: Int): Boolean = currentPage * pageSize < totalCount
}
```

중요한 포인트는 `items`가 누적 결과가 아니라 **이번 페이지 결과만** 담는다는 점이다. 누적 책임은 ViewModel이 가진다. 즉, 데이터 모델과 UI 누적 전략을 분리했다.

4. 성공/실패를 타입으로 다루기

Result.kt

```
sealed class Result<out T> {
    data class Success<T>(val value: T) : Result<T>()
    data class Failure(val error: AppError) : Result<Nothing>()
}
```

이 구조는 "성공하면 값, 실패하면 에러"를 예외 던지기 대신 타입으로 표현한다. ViewModel은 이 타입을 받아 화면 상태로 바꾼다.

왜 좋은가

- 호출자가 성공/실패를 강제로 분기하게 만든다.
- 실패 의미를 `AppError`로 표준화할 수 있다.
- UI가 예외 메시지 문자열에 의존하지 않게 된다.

AppError.kt

```
sealed class AppError {
    abstract val cause: Throwable?

    class Network(...) : AppError()
    data class ApiBadResponse(val code: String) : AppError()
    class Unknown(...) : AppError()
}
```

왜 `AppError`가 필요한가

- 네트워크 예외와 API 비정상 응답과 알 수 없는 실패를 구분한다.
- 외부 API의 raw 메시지를 UI에 직접 노출하지 않게 한다.
- UI는 error code와 type만 보고 친화적인 메시지를 결정할 수 있다.

> 이 코드는 일부러 예외 원문을 그대로 사용자에게 보여주지 않는다. 이건 UX와 보안 둘 다를 위한 결정이다.

5. Repository는 왜 필요한가

공식 아키텍처 권장 사항은 UI 계층이 데이터 소스와 직접 대화하지 말고, 데이터 계층이 repository를 통해 앱 데이터를 노출하라고 한다.

```
interface AddressRepository {
    suspend fun search(
        query: String,
        page: Int,
        pageSize: Int,
    ): Result<AddressPage>
}
```

Repository가 하는 역할

- UI가 네트워크/캐시/DB/DataStore 세부 구현을 모르게 한다.
- 외부 데이터 소스 교체 가능성을 확보한다.
- 앱 내부에서 통일된 도메인 타입을 노출한다.

AddressRepositoryImpl.kt

```
class AddressRepositoryImpl @Inject constructor(
    private val provider: AddressProvider,
) : AddressRepository {
    override suspend fun search(
```

```
        query: String,
        page: Int,
        pageSize: Int,
    ): Result<AddressPage> = provider.search(query, page, pageSize)
}
```

지금은 단순 위임처럼 보이지만, 공식 권장 사항 기준으로도 이 계층은 의미가 있다. 현재 구현이 얇더라도 나중에 data source가 늘어나면 여기에서 흡수한다.

> **중요한 설계 감각**
> 지금 얇아 보여도 경계는 미리 만든다. 작은 앱일수록 "나중에 필요할 때 만들자"는 유혹이 크지만, 오히려 이 시점에 경계를 깔아두면 이후 변경 비용이 낮아진다.

6. Provider는 무엇이 다른가

```
interface AddressProvider {
    suspend fun search(
        query: String,
        page: Int,
        pageSize: Int,
    ): Result<AddressPage>
}
```

이름이 비슷해서 헷갈리지만 `Repository`와 `Provider`는 다르다.

구분	Repository	Provider
관점	앱 내부에서 데이터를 어떻게 제공할 것인가	외부 소스 하나에 어떻게 붙을 것인가
대상 독자	UI/ViewModel	Repository
교체 단위	저장소 전략	특정 API 구현
지금 zipkr에서	`AddressRepository`	`AddressProvider`, 실제 구현은 Juso 쪽

즉, Repository는 "앱의 데이터 창구"이고, Provider는 "특정 외부 서비스에 붙는 어댑터"에 가깝다.

7. CoordinateRepository와 in-memory cache

```
interface CoordinateRepository {
    suspend fun fetchCoordinate(roadAddress: String): Result<Coordinate>
}
```

```
@Singleton
class CoordinateRepositoryImpl @Inject constructor(
    private val api: KakaoLocalApi,
) : CoordinateRepository {
    private val cache = ConcurrentHashMap<String, Coordinate>()

    override suspend fun fetchCoordinate(roadAddress: String): Result<Coordinate> {
        val cached = cache[roadAddress]
        return if (cached != null) {
            Result.Success(cached)
        } else {
            runCatching { api.geocode(roadAddress) }
                .fold(
                    onSuccess = { response -> handleSuccess(roadAddress, response.documents.firstOrNull()) },
                    onFailure = { throwable -> handleFailure(throwable) },
                )
        }
    }
}
```

이 구현에서 배워야 할 것

- 같은 주소를 반복 조회할 때 네트워크를 다시 때리지 않도록 in-memory cache를 둔다.
- network exception은 `AppError.Network`, 그 외는 `AppError.Unknown`로 감싼다.
- 카카오가 200 OK를 줘도 documents가 비어 있을 수 있으므로 이것도 도메인 실패로 본다.

왜 좋은가

- 상세 시트를 여러 번 열 때 체감 속도를 높인다.
- 외부 API의 애매한 응답을 도메인 규칙으로 바꾼다.
- UI는 "성공/실패"만 알면 되고, 카카오 응답 구조를 몰라도 된다.

8. SearchHistoryRepository와 DataStore

이 저장소는 "검색 히스토리/즐겨찾기"라는 앱 내부 기능을 위한 영속 계층이다.

```
interface SearchHistoryRepository {
    fun observeRecent(): Flow<List<Address>>
    fun observeFavorites(): Flow<List<Address>>
    suspend fun addRecent(address: Address)
```

```
    suspend fun toggleFavorite(address: Address)
    suspend fun removeRecent(address: Address)
    suspend fun removeFavorite(address: Address)
}
```

왜 Flow를 쓰나

- 저장소 내용이 바뀌면 UI가 자동으로 새 리스트를 받을 수 있다.
- Compose + ViewModel + StateFlow 구조와 잘 맞는다.
- 최신 Android 아키텍처 권장 사항도 계층 간 통신에 coroutines와 flows를 권장한다.

`DataStoreSearchHistoryRepository.kt`

```
@Singleton
class DataStoreSearchHistoryRepository @Inject constructor(
    private val dataStore: DataStore<Preferences>,
) : SearchHistoryRepository {
    private val json = Json { ignoreUnknownKeys = true }
    private val listSerializer = ListSerializer(Address.serializer())

    override fun observeRecent(): Flow<List<Address>> = dataStore.data.map { it.readList(KEY_RECENT) }
    override fun observeFavorites(): Flow<List<Address>> = dataStore.data.map { it.readList(KEY_FAVORITES) }
    ...
}
```

왜 DataStore인가

공식 DataStore 문서는 Preferences DataStore를 key-value 저장용 modern storage로 소개하고, coroutine/Flow 기반이며 transactional 업데이트를 제공한다고 설명한다. 이 기능은 "최근/즐겨찾기 리스트" 같은 비교적 작은 데이터에 잘 맞는다.

- Room까지 갈 정도로 복잡한 구조는 아니다.
- SharedPreferences보다 비동기/Flow 친화적이다.
- 관찰 가능한 저장소를 만들기 쉽다.

9. 이 구현이 구체적으로 하는 일

`addRecent(address)`

- 기존 리스트에서 같은 `stableKey`를 제거한다.
- 새 항목을 맨 앞에 넣는다.
- `RECENT_LIMIT`까지만 유지한다.

`toggleFavorite(address)`

- 이미 즐겨찾기면 제거
- 없으면 추가

`readList(key)`

- Preferences의 raw JSON string을 읽는다.
- `Address.serializer()`로 리스트를 역직렬화한다.
- 파싱 실패 시 빈 리스트로 복구한다.

> 이 설계는 "모델 구조 변경이나 저장 데이터 손상"이 있어도 앱 전체를 막지 않도록 되어 있다. 파싱 실패 시 crash보다 graceful degradation을 택했다.

장단점

구분	설명
장점	구현이 가볍고, 작은 앱에서 충분하며, Flow 기반 UI 갱신이 쉽다.
단점	리스트 전체를 JSON string으로 저장하므로 항목이 많아질수록 비효율적일 수 있다. 복잡한 조회/정렬에는 적합하지 않다.

10. 테스트는 무엇을 보장하나

`AddressRepositoryImplTest`

- repository가 provider 결과를 그대로 위임하는지 검증
- page, pageSize 인자가 그대로 전달되는지 검증

`CoordinateRepositoryImplTest`

- 정상 응답 시 첫 document를 좌표로 변환하는지 검증
- documents가 비면 `AppError.ApiBadResponse`인지 검증
- IOException이 `AppError.Network`로 감싸지는지 검증
- 같은 주소 두 번 요청 시 캐시 hit으로 API가 한 번만 호출되는지 검증

`AppErrorTest`

- 행안부 errorCode 헬퍼 매핑이 정확한지 검증
- 잘못된 에러 분류 회귀를 막는다

즉, 이 테스트들은 "데이터 계층이 어떤 의미를 가진 값을 내보내는가"를 보호한다. UI 테스트가 아니라 **앱 내부 언어의 정확성**을 고정하는 테스트다.

11. 최신 공식 흐름과 현재 코드의 관계

주제	현재 zipkr	2026-05 공식 흐름과 해석
레이어 분리	UI와 data 패키지가 나뉘어 있음	공식 layered architecture 권장과 맞는다.
Repository	single provider여도 repository 계층 존재	공식 권장 사항이 "single data source여도 repository를 만들라"고 말한다. 잘 맞는다.
coroutines/flows	repository/history 계층에서 Flow와 suspend 사용	계층 간 통신에 coroutines/flows를 쓰라는 권장과 맞는다.
DataStore	Preferences DataStore + Flow	공식 modern storage 흐름과 맞는다. 작은 key-value 영속에 적합하다.
구현체 이름	`*Impl` 사용	공식 naming recommendations는 가능하면 의미 있는 이름을 권장한다. 현재 이름은 흔하지만, 더 도메인적인 이름으로 바꿀 여지도 있다.

12. 직접 적어보기

Q1. 왜 `Address`를 API DTO 그대로 쓰지 않고 별도 도메인 모델로 두는가?

Q2. 왜 `Result`와 `AppError`가 둘 다 필요한가?

Q3. 왜 repository와 provider를 분리했는가?

Q4. 왜 검색 히스토리에는 Room이 아니라 DataStore를 썼는가?

Q5. 현재 구현에서 의미 있는 캐시 전략은 어디에 들어가 있는가?

13. 다음 단계 예고

여기까지 이해했다면 다음은 진짜 외부 API 연결부를 볼 차례다.

1. `JusoApi.kt`, `JusoModels.kt`, `JusoApiProvider.kt`
2. `KakaoLocalApi.kt`, `KakaoModels.kt`
3. `NetworkModule.kt`와 Hilt 연결

> **다음 문서 후보**
> `Phase 5 네트워크 & Provider 워크북`
> 대상: Retrofit, OkHttp, serialization, API key 주입, DTO 매핑

14. 공식 참고 자료

- Guide to app architecture
- Recommendations for Android architecture
- State holders and UI state
- DataStore overview
- Preferences DataStore API reference
- Kotlin coroutines on Android

data 계층은 눈에 덜 보이지만 앱의 품질을 가장 많이 좌우한다. 이 계층이 정리돼 있으면 UI 변경과 API 변경 모두 훨씬 덜 아프다.

PART 6

Android Phase 5 Network Provider Workbook

PHASE 5 DETAILED WORKBOOK

네트워크 & Provider 워크북
Retrofit, OkHttp, DTO, NetworkModule

`zipkr`의 주소 검색/좌표 조회 네트워크 계층을 기준으로 Retrofit 선언 방식, OkHttp interceptor, JSON 직렬화, DTO와 도메인 모델 변환, Provider의 책임, Hilt 모듈에서의 네트워크 객체 생성까지 함께 배우는 인쇄용 상세 자료.

학습 목표
왜 UI가 직접 HTTP를 다루지 않고, DTO/Provider/Repository를 거쳐야 하는지 설명할 수 있어야 한다.

대상 파일
`JusoApi.kt`, `JusoModels.kt`, `JusoApiProvider.kt`, `KakaoLocalApi.kt`, `KakaoModels.kt`, `NetworkModule.kt`

핵심 개념
Retrofit interface, OkHttp interceptor, converter factory, DTO, baseUrl, API key injection

읽는 방식
API 선언 -> 응답 모델 -> Provider -> Hilt 모듈 순으로 본다.

작성일: 2026-05-06. Android app architecture data layer, Retrofit 공식 문서, OkHttp interceptor 문서, Kotlin serialization 문서를 반영했다.

1. 네트워크 계층을 어떻게 봐야 하나

앱이 외부 API를 호출할 때는 사실 네 가지 층이 함께 움직인다.

```
UI / ViewModel
  ↓
Repository
  ↓
Provider
  ↓
Retrofit API + OkHttp + Converter
  ↓
External HTTP API
```

이 구조의 핵심은 "HTTP 세부 구현"이 위쪽 계층으로 새지 않게 만드는 것이다. UI는 주소 검색 결과가 필요할 뿐, 헤더가 어떻게 붙는지, JSON이 어떻게 파싱되는지, base URL이 무엇인지는 몰라도 된다.

> **핵심 문장**
> 네트워크 계층의 목적은 "API를 부르는 것"이 아니라, HTTP 세계를 앱 내부 언어로 번역하는 것이다.

2. 공식 흐름에서 먼저 잡아야 할 것

- Android Developers 데이터 계층 가이드는 repository가 data source를 추상화하라고 말한다.
- Retrofit 공식 문서는 HTTP API를 Kotlin/Java interface로 바꾸는 도구라고 설명한다.
- Retrofit declarations 문서는 메서드/파라미터 annotation이 요청 방식을 결정한다고 설명한다.
- OkHttp 공식 문서는 interceptor가 요청/응답을 감시, 수정, 재시도할 수 있다고 설명한다.
- Kotlin serialization 문서는 JSON 직렬화/역직렬화에 필요한 `@Serializable`, `Json` 설정을 제공한다.

3. JusoApi.kt: Retrofit interface란 무엇인가

```
interface JusoApi {
    @GET("addrlink/addrLinkApi.do?resultType=json")
    suspend fun search(
        @Query("confmKey") apiKey: String,
        @Query("keyword") keyword: String,
        @Query("currentPage") currentPage: Int = DEFAULT_PAGE,
        @Query("countPerPage") countPerPage: Int = DEFAULT_PAGE_SIZE,
    ): JusoSearchResponse
}
```

Retrofit interface의 본질

Retrofit 공식 문서는 interface 메서드와 annotation으로 HTTP 요청을 선언한다고 설명한다. 즉, 이 파일은 네트워크 호출 "코드"라기보다 "요청 명세"에 가깝다.

코드	의미
`@GET(...)`	이 메서드가 GET 요청임을 선언한다. 상대 경로와 고정 query도 함께 적는다.
`@Query("confmKey")`	행안부 API 인증 키를 query parameter로 붙인다.
`@Query("keyword")`	사용자 검색어를 query parameter로 보낸다.
`suspend fun`	Retrofit Kotlin 지원 기능으로, 비동기 HTTP 호출을 coroutine suspension으로 연결한다.
반환 타입 `JusoSearchResponse`	응답 JSON을 이 DTO로 역직렬화한다.

왜 기본값이 있나

`currentPage`, `countPerPage` 기본값은 API interface를 더 안전하고 간단하게 쓰기 위해 둔다. 다만 실제 현재 앱에서는 ViewModel이 항상 명시적으로 page/pageSize를 넘긴다.

4. JusoModels.kt: DTO는 왜 따로 있나

```
@Serializable
data class JusoSearchResponse(val results: JusoResults)

@Serializable
data class JusoResults(
    val common: JusoCommon,
    val juso: List<JusoAddressDto> = emptyList(),
)

@Serializable
data class JusoCommon(
    val errorCode: String,
    val errorMessage: String,
    val totalCount: String = "0",
)
```

DTO(Data Transfer Object)는 외부 응답 형식을 그대로 받기 위한 모델이다. 여기서 중요한 건 `Address`와 분리돼 있다는 점이다.

왜 분리해야 하나

- 외부 API 필드명이 바뀌어도 UI 모델과 분리돼 영향 범위를 줄인다.
- 외부 응답 형식의 이상한 점을 이 층에서 흡수할 수 있다.
- 예: `totalCount`가 숫자가 아니라 문자열로 온다.
- 예: 일부 필드는 누락될 수 있어 기본값을 둔다.

`@SerialName`가 하는 일

```
@SerialName("roadAddr") val roadAddr: String
@SerialName("zipNo") val zipNo: String
```

JSON 필드명과 Kotlin 프로퍼티 이름을 연결한다. DTO 층에서는 외부 API 명세를 따라야 하므로 이런 annotation이 중요하다.

toDomain()이 중요한 이유

```
fun toDomain(): Address =
    Address(
        zipCode = zipNo,
        roadAddress = roadAddr,
        ...
    )
```

이 함수가 바로 "외부 응답을 앱 내부 언어로 번역하는 지점"이다.

5. JusoApiProvider.kt: Provider의 진짜 책임

```
class JusoApiProvider @Inject constructor(
    private val api: JusoApi,
    @Named(DiQualifiers.JUSO_API_KEY) private val apiKey: String,
) : AddressProvider {
    override suspend fun search(query: String, page: Int, pageSize: Int): Result<AddressPage> =
        try {
            val response = api.search(...)
            val common = response.results.common
            val errorCode = common.errorCode
            val totalCount = common.totalCount.toIntOrNull() ?: 0
            if (errorCode == SUCCESS_CODE) {
                Result.Success(AddressPage(...))
            } else {
                Result.Failure(AppError.ApiBadResponse(code = errorCode))
            }
        } catch (io: IOException) {
            Result.Failure(AppError.Network(io))
        } catch (t: Throwable) {
            Result.Failure(AppError.Unknown(t))
        }
}
```

이 파일이 하는 세 가지

1. Retrofit API를 호출한다.
2. DTO를 도메인 모델로 바꾼다.
3. 예외와 API 에러를 `AppError`로 바꾼다.

특히 중요한 부분

- `errorCode == "0"`일 때만 성공으로 본다.

- `errorMessage`는 도메인 타입에 담지 않고 Timber 로그에만 남긴다.
- `IOException`은 네트워크 실패, 그 외는 Unknown으로 분류한다.
- `totalCount.toIntOrNull() ?: 0`로 API의 문자열 숫자를 안전하게 파싱한다.

> **Provider를 따로 두는 가치**
> 이 파일 덕분에 Repository와 ViewModel은 행안부 API의 이상한 규칙, errorCode 체계, raw JSON 구조를 모른다.

6. KakaoLocalApi와 KakaoModels

```
interface KakaoLocalApi {
    @GET("v2/local/search/address.json")
    suspend fun geocode(
        @Query("query") query: String,
    ): KakaoGeocodeResponse
}
```

```
@Serializable
data class KakaoGeocodeResponse(
    val documents: List<KakaoDocument> = emptyList(),
)

@Serializable
data class KakaoDocument(
    val x: String,
    val y: String,
) {
    fun toCoordinate(): Coordinate = Coordinate(longitude = x.toDouble(), latitude = y.toDouble())
}
```

카카오 쪽은 구조가 더 단순하다. 좌표 변환이 목적이기 때문에 응답 중 필요한 필드만 추렸다.

- `x`는 longitude
- `y`는 latitude
- 둘 다 문자열이므로 도메인 변환 시 `toDouble()` 한다

7. NetworkModule.kt: 객체 생성 공장

실제 HTTP 호출은 API interface나 Provider에서 보이지만, 객체 생성은 Hilt module에서 한다. 이 파일은 네트워크 객체들의 "공장"이다.

```
@Module
@InstallIn(SingletonComponent::class)
object ㅌNetworkModule {
    @Provides @Singleton fun provideJson(): Json = ...
    @Provides @Singleton fun provideLoggingInterceptor(): HttpLoggingInterceptor = ...

    @Provides @Singleton @Named(DiQualifiers.JUSO_OKHTTP)
    fun provideJusoOkHttpClient(...): OkHttpClient = ...

    @Provides @Singleton fun provideJusoRetrofit(...): Retrofit = ...
    @Provides @Singleton fun provideJusoApi(retrofit: Retrofit): JusoApi = ...
    @Provides @Singleton @Named(DiQualifiers.JUSO_API_KEY)
    fun provideJusoApiKey(): String = BuildConfig.JUSO_API_KEY

    @Provides @Singleton @Named(DiQualifiers.KAKAO_REST_API_KEY)
    fun provideKakaoRestApiKey(): String = BuildConfig.KAKAO_REST_API_KEY

    @Provides @Singleton @Named(DiQualifiers.KAKAO_OKHTTP)
    fun provideKakaoOkHttpClient(...): OkHttpClient = ...

    @Provides @Singleton fun provideKakaoLocalApi(...): KakaoLocalApi = ...
}
```

눈에 띄는 점 1: 이름

`object ㅌNetworkModule`처럼 보이는데, 이건 Kotlin이 유니코드 식별자를 허용하기 때문에 컴파일은 된다. 다만 일반적인 팀 코드라면 거의 확실히 오타 또는 비표준 네이밍으로 본다. 학습 관점에서는 "Kotlin 식별자는 넓게 허용되지만, 읽기 좋은 이름을 쓰는 것이 훨씬 중요하다"는 포인트다.

눈에 띄는 점 2: Json 설정

```
Json {
    ignoreUnknownKeys = true
    coerceInputValues = true
}
```

- `ignoreUnknownKeys = true`: JSON에 우리가 모르는 필드가 있어도 무시
- `coerceInputValues = true`: 일부 잘못된 값/누락 값을 더 유연하게 다룸

Kotlin serialization 문서의 `Json {}` 설정 포인트를 실제 앱에 적용한 예다.

눈에 띄는 점 3: Logging interceptor

```
HttpLoggingInterceptor().apply {
    level = if (BuildConfig.DEBUG) BODY else NONE
}
```

Debug에서는 요청/응답을 자세히 보고, Release에서는 끈다. 운영 환경에서 민감 데이터/성능 문제를 줄이기 위한 일반적인 전략이다.

8. OkHttp interceptor는 왜 중요한가

OkHttp 공식 문서는 interceptor가 요청을 감시하고 수정하고 재시도할 수 있다고 설명한다. 현재 코드에서는 카카오 REST 인증 헤더를 붙이는 데 쓴다.

```
val authInterceptor =
    Interceptor { chain ->
        val req =
            chain.request().newBuilder()
                .addHeader("Authorization", "KakaoAK $apiKey")
                .build()
        chain.proceed(req)
    }
```

왜 좋은가

- API interface마다 매번 `@Header`를 적지 않아도 된다.
- "이 클라이언트는 항상 이 헤더가 붙는다"는 정책을 한 곳에 모을 수 있다.
- 요청 정책을 interface 선언에서 분리할 수 있다.

> 헤더를 붙이는 방식은 강력하지만, 어떤 client에 어떤 interceptor가 붙는지 추적하지 못하면 금방 헷갈린다. 그래서 현재 코드처럼 `@Named(DiQualifiers.KAKAO_OKHTTP)` 같은 구분자가 중요하다.

9. Retrofit Builder와 Converter Factory

```
Retrofit.Builder()
    .baseUrl(JUSO_BASE_URL)
    .client(client)
    .addConverterFactory(json.asConverterFactory("application/json".toMediaType()))
    .build()
```

각 요소의 역할

구성 요소	의미
`baseUrl(...)`	상대 경로 앞에 붙을 공통 URL이다. Retrofit은 base URL 마지막에 slash가 있어야 한다.
`client(client)`	어떤 OkHttpClient를 쓸지 지정한다. 즉, timeout/interceptor/logging 정책을 가진 실제 HTTP 엔진을 연결한다.
`addConverterFactory(...)`	응답 body를 어떤 방식으로 Kotlin 객체로 바꿀지 지정한다.
`retrofit.create(Api::class.java)`	interface 선언으로부터 실제 호출 가능한 구현체를 생성한다.

Retrofit configuration 문서가 강조하는 핵심도 이 부분이다. 기본 Retrofit은 `ResponseBody`만 다룰 수 있고, 객체 변환은 converter를 추가해야 한다. 현재 프로젝트는 Kotlin serialization converter를 쓰고 있다.

10. BuildConfig와 API 키 주입

네트워크 계층에서 중요한 것은 코드만이 아니다. API 키를 어디서 읽고 어떻게 주입하느냐도 구조의 일부다.

- `app/build.gradle.kts`에서 `local.properties` 값을 `BuildConfig` 필드로 주입한다.
- `NetworkModule`은 그 `BuildConfig.JUSO_API_KEY`, `BuildConfig.KAKAO_REST_API_KEY`를 다시 제공한다.
- Provider나 OkHttp client는 Hilt로 주입받는다.

```
@Provides
@Singleton
@Named(DiQualifiers.JUSO_API_KEY)
fun provideJusoApiKey(): String = BuildConfig.JUSO_API_KEY
```

즉, 키가 "문자열 상수로 직접 박혀" 돌아다니지 않는다. 이건 보안 때문이기도 하고, 테스트/환경 분리 때문이기도 하다.

11. 현재 코드와 최신 흐름 비교

주제	현재 zipkr	최신 흐름과 해석
네트워크 추상화	Repository → Provider → API interface	Android data layer 권장 구조와 잘 맞는다.
Retrofit 선언	suspend function + DTO 반환	Retrofit 공식 Kotlin support 방식과 맞는다.
헤더 주입	OkHttp interceptor로 카카오 Authorization 추가	OkHttp 공식 interceptor 문서가 설명하는 대표적인 사용 방식이다.

주제	현재 zipkr	최신 흐름과 해석
JSON 처리	kotlinx.serialization + `Json { ignoreUnknownKeys ... }`	현대 Kotlin 앱에서 매우 자연스러운 선택이다.
네이밍	`ㅌNetworkModule` 같은 유니코드 식별자	컴파일은 되지만 팀 코드/장기 유지보수 관점에서는 비표준적이다. 학습상 주의 포인트다.

12. 직접 적어보기

Q1. 왜 `JusoApi`는 바로 UI에서 호출하지 않고 Provider를 거쳐야 하는가?

Q2. 왜 DTO와 도메인 모델을 분리하는가?

Q3. 왜 카카오 인증은 `@Header`가 아니라 interceptor로 넣었는가?

Q4. 왜 `errorMessage`를 `AppError`에 직접 넣지 않았는가?

Q5. 현재 네트워크 계층에서 "HTTP 세계를 앱 언어로 번역하는 지점"은 정확히 어디인가?

13. 다음 단계 예고

다음은 DI 자체를 집중적으로 볼 차례다.

1. `@HiltAndroidApp`, `@AndroidEntryPoint`, `@HiltViewModel`
2. `@Inject`, `@Binds`, `@Provides`, `@Named`, `@Singleton`
3. `DataModule`, `PersistenceModule`, `NetworkModule`, `DiQualifiers`

14. 공식 참고 자료

- Data layer
- Recommendations for Android architecture
- Retrofit introduction
- Retrofit declarations
- Retrofit configuration
- OkHttp interceptors
- Kotlin serialization

네트워크 계층을 제대로 이해하면 이후 어떤 외부 API가 와도 대응이 쉬워진다. 핵심은 라이브러리 암기가 아니라 경계 설계다.

PART 7

Android Phase 6 Hilt DI Workbook

PHASE 6 DETAILED WORKBOOK

Hilt & DI 워크북
객체 그래프 추적하기

`zipkr`의 Hilt 구성을 기준으로 `@HiltAndroidApp`, `@AndroidEntryPoint`, `@HiltViewModel`, `@Inject`, `@Binds`, `@Provides`, `@Named`, `@Singleton`, 그리고 최신 Hilt 흐름까지 함께 정리한 인쇄용 상세 자료.

학습 목표

SearchViewModel 안의 repository가 누가 만들어서 어떻게 들어오는지 끝까지 설명할 수 있어야 한다.

대상 파일

`ZipkrApp.kt`, `MainActivity.kt`, `DataModule.kt`, `PersistenceModule.kt`, `NetworkModule.kt`, `DiQualifiers.kt`

핵심 개념

DI, Hilt component, module, binding, qualifier, singleton scope

읽는 방식

어노테이션 하나씩이 무엇을 생성하고 연결하는지 객체 흐름으로 추적한다.

작성일: 2026-05-06. Android Developers Hilt 문서와 androidx Hilt release notes를 반영했다.

1. DI란 무엇인가

DI(Dependency Injection)는 "객체가 필요한 의존성을 자기 안에서 직접 만들지 않고, 바깥에서 주입받는 방식"이다. Android Developers의 Hilt 문서는 수동 DI는 모든 클래스와 그 의존성을 손으로 만들고 관리해야 해 boilerplate가 커진다고 설명한다.

> **핵심 문장**
> DI의 목표는 편하게 만드는 것이 아니라, 생성 책임을 분리해 변경과 테스트를 쉽게 만드는 것이다.

수동 생성과 DI의 차이

방식	특징
수동 생성	`SearchViewModel(repository = AddressRepositoryImpl(...))` 처럼 직접 다 만들고 넘긴다. 작은 샘플에선 쉽지만 앱이 커질수록 복잡하다.
DI	객체 생성/연결을 Hilt 같은 컨테이너가 맡고, 코드는 "무엇이 필요한지"만 선언한다.

2. Hilt가 현재 앱에서 하는 일

현재 `zipkr`에서 Hilt는 최소한 아래 연결을 자동으로 한다.

```
ZipkrApp (@HiltAndroidApp)
  ↓ application-level component 생성
MainActivity (@AndroidEntryPoint)
  ↓
SearchScreen() inside MainActivity
  ↓ hiltViewModel()
SearchViewModel (@HiltViewModel)
  ↓ constructor injection
AddressRepository / SearchHistoryRepository
  ↓ modules
AddressRepositoryImpl / DataStoreSearchHistoryRepository
  ↓ further deps
AddressProvider / DataStore<Preferences>
```

즉, 이 앱에서 Hilt는 "객체 생성 공장"이자 "객체 그래프 연결기" 역할을 한다.

3. ZipkrApp과 `@HiltAndroidApp`

```
@HiltAndroidApp
class ZipkrApp : Application() {
    override fun onCreate() {
        super.onCreate()
        ...
    }
}
```

Android Developers Hilt 문서는 Hilt를 쓰는 모든 앱은 `@HiltAndroidApp`가 붙은 `Application` 클래스를 가져야 한다고 설명한다.

이 어노테이션이 하는 일

- Hilt code generation을 트리거한다.
- application-level dependency container를 만든다.
- 다른 컴포넌트들의 부모가 되는 root component를 제공한다.

현재 코드에서 이 의미

- `SingletonComponent`에 설치된 모듈들이 여기부터 살아난다.
- `DataModule`, `PersistenceModule`, `NetworkModule`의 binding들이 application graph에 들어간다.

4. MainActivity와 `@AndroidEntryPoint`

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() { ... }
```

Hilt 공식 문서는 `@AndroidEntryPoint`가 붙은 Android 클래스마다 개별 Hilt component가 생성된다고 설명한다. 그리고 Hilt는 `ComponentActivity`를 상속한 Activity를 지원한다.

왜 Activity에 이게 필요한가

- 이 Activity가 Hilt graph와 연결되는 진입점이 된다.
- 이 안에서 사용하는 Fragment/View/Compose/HiltViewModel 흐름이 자연스럽게 이어진다.
- 만약 Activity가 Hilt graph 바깥이면 그 안에서 Hilt 기반 화면/객체 주입이 끊긴다.

5. SearchViewModel과 `@HiltViewModel`

```
@HiltViewModel
class SearchViewModel @Inject constructor(
    private val repository: AddressRepository,
    private val historyRepository: SearchHistoryRepository,
) : ViewModel()
```

여기서 일어나는 일

- `@HiltViewModel`는 이 ViewModel을 Hilt가 생성할 수 있게 한다.
- 생성자에 `@Inject`가 붙어 있으므로, Hilt는 필요한 인자를 graph에서 찾는다.
- `AddressRepository`와 `SearchHistoryRepository`를 어디서 구할지는 module bindings가 결정한다.

> **이해 포인트**
> ViewModel이 의존성을 "요구"만 하고, 실제 생성과 연결은 Hilt가 한다. 이게 생성자 주입의 핵심이다.

6. `@Inject`, `@Binds`, `@Provides` 차이

도구	언제 쓰는가
`@Inject constructor`	구현체를 Hilt가 직접 생성할 수 있을 때. 예: `AddressRepositoryImpl`, `CoordinateRepositoryImpl`, `DataStoreSearchHistoryRepository`
`@Binds`	인터페이스와 구현체를 연결할 때. "이 인터페이스가 필요하면 이 구현체를 써라"를 선언한다.
`@Provides`	생성 코드가 직접 필요할 때. 예: Retrofit.Builder, OkHttpClient.Builder, DataStore delegate 접근

일반적으로 **직접 생성 가능한 구현체는 `@Inject`, 인터페이스 연결은 `@Binds`, 복잡한 팩토리/외부 라이브러리 생성은 `@Provides`**라고 생각하면 된다.

7. DataModule.kt 읽기

```
@Module
@InstallIn(SingletonComponent::class)
abstract class DataModule {
    @Binds @Singleton
    abstract fun bindAddressProvider(impl: JusoApiProvider): AddressProvider

    @Binds @Singleton
    abstract fun bindAddressRepository(impl: AddressRepositoryImpl): AddressRepository

    @Binds @Singleton
    abstract fun bindCoordinateRepository(impl: CoordinateRepositoryImpl): CoordinateRepository

    @Binds @Singleton
    abstract fun bindSearchHistoryRepository(impl: DataStoreSearchHistoryRepository): SearchHistoryRepository
}
```

이 파일이 말하는 것

- `AddressProvider`가 필요하면 `JusoApiProvider`를 써라
- `AddressRepository`가 필요하면 `AddressRepositoryImpl`를 써라
- `CoordinateRepository`가 필요하면 `CoordinateRepositoryImpl`를 써라
- `SearchHistoryRepository`가 필요하면 `DataStoreSearchHistoryRepository`를 써라

왜 abstract class + @Binds인가

`@Binds`는 구현체를 새로 만드는 코드가 아니라 "연결 정보"만 선언한다. 그래서 메서드 본문이 필요 없고 abstract로 선언된다.

8. PersistenceModule.kt와 DataStore 제공

```
private val Context.searchHistoryStore: DataStore<Preferences> by preferencesDataStore(name = SEARCH_HISTORY_STORE_NAME)

@Module
@InstallIn(SingletonComponent::class)
object PersistenceModule {
    @Provides
    @Singleton
    fun provideSearchHistoryDataStore(
        @ApplicationContext context: Context,
    ): DataStore<Preferences> = context.searchHistoryStore
}
```

여기서 배워야 할 것

- `@ApplicationContext`는 Hilt가 기본으로 제공하는 qualifier다.
- DataStore는 `Context` extension delegate로 생성되므로, 단순 생성자 주입만으로는 만들기 어렵다.
- 그래서 `@Provides`가 필요하다.

이 파일은 "외부 라이브러리/플랫폼 객체를 Hilt graph에 넣는 방법"을 보여주는 좋은 예다.

9. NetworkModule.kt와 Qualifier

네트워크 계층에서 같은 타입의 객체가 여러 개 생긴다. 대표적으로 `OkHttpClient`가 두 개다.

```
const val JUSO_OKHTTP = "juso_okhttp"
const val KAKAO_OKHTTP = "kakao_okhttp"
```

```
@Provides
@Singleton
@Named(DiQualifiers.JUSO_OKHTTP)
fun provideJusoOkHttpClient(...): OkHttpClient = ...

@Provides
@Singleton
@Named(DiQualifiers.KAKAO_OKHTTP)
fun provideKakaoOkHttpClient(...): OkHttpClient = ...
```

왜 qualifier가 필요한가

- 둘 다 타입은 `OkHttpClient`지만, 설정이 다르다.
- 하나는 Juso용, 하나는 Kakao용이다.
- Hilt는 타입만으로 구분이 안 되므로 이름표가 필요하다.

> Qualifier가 없으면 같은 타입 binding이 여러 개 있을 때 주입 지점을 결정할 수 없어 graph가 깨진다.

`DiQualifiers.kt`를 따로 둔 이유

```
object DiQualifiers {
    const val JUSO_API_KEY = "juso_api_key"
    const val KAKAO_REST_API_KEY = "kakao_rest_api_key"
    const val JUSO_OKHTTP = "juso_okhttp"
    const val KAKAO_OKHTTP = "kakao_okhttp"
}
```

매직 스트링이 여러 파일에 흩어지면 오타로 DI 미스매치가 날 수 있다. 그래서 qualifier 키를 한 곳에 모은다.

10. SingletonComponent는 무엇인가

Hilt 공식 문서는 generated component hierarchy를 제공한다. 현재 모듈들은 전부 `@InstallIn(SingletonComponent::class)`다.

의미

- 앱 전역 생명주기(application lifecycle)에 묶인다.
- 이 component에 들어간 binding은 앱 전체에서 재사용 가능하다.
- child component(Activity, ViewModel 등)에서 접근 가능하다.

현재 코드에서 왜 적절한가

- Repository
- DataStore
- Retrofit / OkHttp / Json
- API keys

이런 것들은 보통 화면마다 새로 만들 필요가 없다.

11. 객체가 실제로 어떻게 주입되는가

```
SearchScreen()
  ↓ hiltViewModel()
SearchViewModel
  ↓ constructor params
AddressRepository
  ↓ DataModule.bindAddressRepository
AddressRepositoryImpl
  ↓ constructor param
AddressProvider
  ↓ DataModule.bindAddressProvider
JusoApiProvider
  ↓ constructor params
JusoApi + @Named(JUSO_API_KEY) String
  ↓ NetworkModule provides
Retrofit-created JusoApi + BuildConfig JUSO key
```

이 흐름을 말로 설명할 수 있으면 Hilt를 제대로 이해한 것이다.

12. 최신 Hilt 흐름과 현재 코드의 관계

주제	현재 zipkr	최신 흐름과 해석
Hilt app setup	`@HiltAndroidApp`, `@AndroidEntryPoint`, `@HiltViewModel` 사용	공식 Hilt Android 문서와 잘 맞는다.
hiltViewModel import	`androidx.hilt.navigation.compose.hiltViewModel`	AndroidX Hilt 1.3.0 release notes에 따르면 Compose용 `hiltViewModel()` API는 새 artifact/package `androidx.hilt.lifecycle.viewmodel.compose`로 이동했다. 현재 프로젝트는 이전 경로를 쓰고 있다.
bindings	`@Binds`, `@Provides`를 상황에 맞게 분리	좋은 Hilt usage 패턴이다.
scope	대부분 `@Singleton`	앱 전역 객체에 적절하다. 다만 무조건 Singleton으로 가는 습관은 경계해야 한다.

> **최신 포인트**
> 지금 코드는 Hilt 구조 자체는 현대적이다. 다만 Compose용 `hiltViewModel()` artifact/package 변화는 나중에 라이브러리 업그레이드 때 반드시 다시 확인할 가치가 있다.

13. 직접 적어보기

Q1. 왜 `SearchViewModel`은 repository를 직접 생성하지 않는가?

Q2. `@Binds`와 `@Provides` 차이를 현재 코드 예시로 설명해보라.

Q3. 왜 같은 `OkHttpClient` 타입에 qualifier가 필요한가?

Q4. `SingletonComponent`에 설치된 binding이 왜 Activity나 ViewModel에서 보이는가?

Q5. 현재 프로젝트가 최신 Hilt 흐름과 다른 지점은 무엇인가?

14. 다음 단계 예고

이제 네트워크와 DI를 모두 봤으므로, 다음은 상세 화면과 WebView/지도/딥링크 같은 실전 UX 묶음으로 가면 된다.

1. `DetailSheet.kt`
2. `DetailViewModel.kt`
3. `KakaoMapWebView.kt`
4. `MapDeepLinks.kt`

15. 공식 참고 자료

- Dependency injection with Hilt
- AndroidX Hilt release notes
- Guide to app architecture
- Recommendations for Android architecture

Hilt를 이해한다는 것은 어노테이션 이름을 외우는 것이 아니라, 객체 생성 책임이 어디 있는지 추적할 수 있다는 뜻이다.

PART 8

Android Phase 7 Detail Sheet Workbook

PHASE 7 DETAILED WORKBOOK

상세 시트 워크북
DetailSheet, Header, Copy UX, DetailViewModel

`zipkr`의 검색 결과 상세 모달을 기준으로 `ModalBottomSheet`, `LaunchedEffect`, 작은 상태 훅, 애니메이션 하이라이트, 즐겨찾기 토글, ViewModel 역할 분리까지 한 번에 익히는 인쇄용 상세 자료.

학습 목표
검색 결과 카드 하나를 누른 뒤 상세 시트가 닫힐 때까지 어떤 상태와 효과가 움직이는지 설명할 수 있어야 한다.

대상 파일
`DetailSheet.kt`, `DetailHeader.kt`, `DetailUiState.kt`, `DetailViewModel.kt`, `DetailViewModelTest.kt`, `CopyBar.kt`

핵심 개념
ModalBottomSheet, state hoisting, side effects, composable-local state, Flow, animation, sealed state

읽는 방식
UI 껍데기 -> side effect -> 작은 UX 훅 -> header 시각 계층 -> ViewModel 상태 흐름 순으로 본다.

작성일: 2026-05-06. Android Developers Compose side-effects, bottom sheet, lifecycle-aware state collection, Hilt ViewModel 문서를 반영했다.

1. 왜 상세 시트가 중요한가

`zipkr`에서 상세 시트는 단순한 팝업이 아니다. 사용자가 주소를 더 읽고, 좌표를 받아 지도를 보고, 외부 지도 앱으로 넘어가고, 최근 기록을 남기고, 즐겨찾기를 토글하고, 여러 형태의 주소를 복사하는 핵심 작업장이 된다.

```
검색 결과 카드 탭
  ↓
DetailSheet(address, query, onDismiss)
  ↓
LaunchedEffect(address.roadAddress)
  ├─ fetchCoordinate(roadAddress)
  └─ rememberRecent(address)
  ↓
ModalBottomSheet
  ├─ DetailHeader
  ├─ MapArea
  ├─ MapDeepLinkButtons
  └─ CopyBar
```

> **핵심 문장**
> 상세 시트는 "결과를 보여주는 UI"가 아니라 "주소를 실제로 써먹게 만드는 작업 공간"이다.

2. DetailSheet 시그니처에서 바로 읽어야 할 것

```
@Composable
fun DetailSheet(
    address: Address,
    onDismiss: () -> Unit,
    query: String = "",
    viewModel: DetailViewModel = hiltViewModel(),
)
```

요소	뜻
`address: Address`	이 시트가 그릴 대상 데이터다. 상세 시트는 자체적으로 주소를 로드하지 않는다.
`onDismiss`	시트 닫힘의 최종 권한은 부모 화면이 가진다. 이것이 state hoisting이다.
`query`	검색어 하이라이트에만 쓰인다. 즉, 상세 화면도 검색 맥락을 일부 이어받는다.
`viewModel = hiltViewModel()`	상세 시트는 UI 함수지만, 실제 좌표 조회와 최근/즐겨찾기 로직은 ViewModel로 보낸다.

Android Developers의 상태 끌어올리기 가이드 관점에서 보면, 이 함수는 "부모가 소유한 데이터와 이벤트를 받아서 UI와 효과를 조합하는 leaf container"에 가깝다.

3. 시트 상태와 Flow 구독

```
val sheetState = rememberModalBottomSheetState(skipPartiallyExpanded = true)
val scope = rememberCoroutineScope()
val coordinatePhase by viewModel.coordinate.collectAsState()
```

Compose bottom sheet 가이드 기준으로 `SheetState`는 시트를 프로그래밍적으로 제어하는 손잡이다. 여기서는 `skipPartiallyExpanded = true`로 반쯤 열린 상태를 건너뛰고 바로 크게 연다.

왜 `rememberCoroutineScope()`가 필요한가

`sheetState.hide()`는 suspend 함수이므로 클릭 이벤트에서 바로 호출할 수 없다. 그래서 composable 생명주기에 묶인 coroutine scope가 필요하다.

왜 여기서는 `collectAsState()` 인가

루트 화면에서는 보통 `collectAsStateWithLifecycle()` 가 더 권장된다. 하지만 이 시트는 화면 내부의 짧은 수명 composable이고, 소유 Activity가 살아있는 동안만 노출된다. 현재 구조에서도 동작은 안전하다. 다만 "모든 Flow 수집을 lifecycle-aware로 통일한다"는 팀 규칙이 있다면 맞춰 바꿔볼 만하다.

4. LaunchedEffect: 시트 진입을 이벤트로 바꾸기

```
LaunchedEffect(address.roadAddress) {
    viewModel.fetchCoordinate(address.roadAddress)
    viewModel.rememberRecent(address)
}
```

Android Developers side-effects 문서는 `LaunchedEffect` 를 "Composable 수명과 함께 suspend 작업을 실행하는 곳"이라고 설명한다. 이 코드는 상세 시트가 열리는 순간을 UI 이벤트가 아니라 effect로 표현한다.

여기서 동시에 일어나는 두 작업

1. 도로명 주소로 좌표를 조회한다.
2. 사용자가 이 주소를 열어봤다는 사실을 최근 목록에 기록한다.

왜 key가 `address.roadAddress` 인가

검색 결과 A를 보다가 검색 결과 B를 같은 시트 위치에서 연 경우, effect는 새 주소 기준으로 다시 시작되어야 한다. 그래서 key를 주소 식별자에 묶는다.

> **공부 포인트**
> `LaunchedEffect(Unit)` 로 바꾸면 "처음 한 번만" 실행될 수 있어 다른 주소로 갈아탈 때 좌표 조회/최근 기록이 갱신되지 않는다.

5. ModalBottomSheet와 dismiss 흐름

```
ModalBottomSheet(
    onDismissRequest = onDismiss,
    sheetState = sheetState,
    shape = RoundedCornerShape(...),
    dragHandle = {
        ClickableDragHandle(
            scope = scope,
            sheetState = sheetState,
            onDismiss = onDismiss,
        )
    },
) { ... }
```

Compose bottom sheet 문서 기준으로 `ModalBottomSheet` 는 화면 앞에 뜨는 모달 형태의 시트다. 이 앱은 기본 drag handle을 그대로 두지 않고, 그 주변 전체를 클릭 가능한 dismiss 영역으로 다시 감싼다.

```
scope.launch { sheetState.hide() }.invokeOnCompletion {
    if (!sheetState.isVisible) onDismiss()
}
```

이 설계가 좋은 이유

- 바로 부모 상태를 꺼버리지 않고, 먼저 시트 hide 애니메이션을 수행한다.
- 애니메이션 완료 후 실제로 보이지 않을 때만 `onDismiss()` 를 호출한다.
- 즉, UI 상태와 애니메이션 상태의 순서를 맞춘다.

왜 풀 폭 클릭 영역인가

작은 막대만 누르게 하지 않고 그 주변까지 탭 타깃을 넓혀서 사용성이 좋아진다. 이런 부분이 "코드 구조"보다 "제품 감각"에 가까운 결정이다.

6. 작은 훅처럼 만든 복사 UX

```
@Composable
private fun rememberCopyHandler(): CopyHandler {
    val context = LocalContext.current
    val view = LocalView.current
    var lastCopied by remember { mutableStateOf<CopyField?>(null) }

    LaunchedEffect(lastCopied) {
        if (lastCopied != null) {
            delay(HIGHLIGHT_DURATION_MS)
            lastCopied = null
        }
    }
    ...
}
```

이 함수는 React 스타일 훅과 비슷한 역할을 한다. 외부로는 `lastCopied` 와 `onCopy` 만 내보내고, 내부에서는 클립보드, 햅틱, 토스트, 하이라이트 타이머를 묶어 처리한다.

요소	배울 점
`LocalContext.current`	안드로이드 시스템 API 접근용 컨텍스트를 composable 안에서 꺼낸다.
`LocalView.current`	햅틱처럼 View 기반 API가 필요할 때 사용한다.
`mutableStateOf<CopyField?>`	짧게 살아야 하는 UI 로컬 상태는 굳이 ViewModel까지 올리지 않는다.
`LaunchedEffect(lastCopied)`	복사 직후 800ms 동안만 강조하고 자동 해제한다.

중요한 설계 감각
모든 상태를 ViewModel로 올리는 것이 정답은 아니다. 화면 안에서만 쓰고 금방 사라지는 상태는 composable local state가 더 자연스럽다.

7. DetailSheetContent: 조립 전담 함수

```
Column {
    DetailHeader(...)
    MapArea(...)
    MapDeepLinkButtons(...)
    CopyBar(...)
}
```

이 함수에는 비즈니스 로직이 거의 없다. 이미 계산된 상태와 이벤트 핸들러를 받아 화면 조각들을 순서대로 조립하는 데 집중한다.

왜 이런 구조가 좋은가

- `DetailSheet`는 effect와 sheet shell을 담당한다.
- `DetailSheetContent`는 화면 조립을 담당한다.
- `MapArea`는 좌표 상태별 분기를 담당한다.
- `DetailHeader`는 주소 텍스트와 즐겨찾기 시각 계층을 담당한다.

즉, 한 파일 안에서도 책임이 다시 잘게 나뉘어 있다. 이 프로젝트가 계속 반복하는 습관은 "긴 함수 하나"보다 "작은 역할 함수 여러 개"다.

8. MapArea: sealed state를 UI로 번역하기

```
when (phase) {
    CoordinatePhase.Loading -> MapPlaceholder { CircularProgressIndicator() }
    is CoordinatePhase.Success -> KakaoMapWebView(...)
    is CoordinatePhase.Failure -> MapFailure(onRetry)
}
```

여기서는 별도의 if 묶음 대신 `CoordinatePhase`라는 sealed interface를 직접 그린다. 이 패턴은 "상태 머신을 UI에 그대로 매핑한다"는 Compose/MVVM 스타일에 가깝다.

이 설계의 장점

- 로딩, 성공, 실패가 동시에 참이 되는 애매한 상태를 막는다.
- 새 상태가 추가되면 `when` 분기에서 즉시 누락을 발견하기 쉽다.
- 디자인적으로도 placeholder shell이 공통이어서 전환이 깔끔하다.

9. DetailHeader: 시각 계층과 micro animation

```
ZipRow(...)
HighlightableLine(highlighted = lastCopied == CopyField.Road, brand = brand) {
    Text(text = highlightQuery(address.roadAddress, query, highlightStyle), ...)
}
```

이 헤더는 단순 텍스트 나열이 아니다. 점보 우편번호, 도로명 우선순위, 복사 직후 잠깐 강조, 검색어 매칭 강조, 즐겨찾기 별 토글까지 한 줄씩 의미를 가진다.

배워야 할 Compose 포인트

- `animateColorAsState`로 텍스트 색을 자연스럽게 바꾼다.
- `animateFloatAsState`로 살짝 확대해 "방금 복사됨"을 시각적으로 알려준다.
- `highlightQuery(...)`는 검색어 맥락을 상세 시트까지 이어준다.
- `IconButton` 안의 별 아이콘은 boolean 상태를 즉시 시각화한다.

왜 영어 주소는 query highlight를 하지 않나

코드 주석대로 현재 행안부 검색은 한글 매칭 기준이므로, 영어 주소에서 같은 하이라이트 경험을 약속할 수 없다. 이건 "기능적으로 불완전한 강조"를 억지로 하지 않겠다는 제품 결정이다.

10. DetailViewModel: 무엇을 하고 무엇을 하지 않는가

```
@HiltViewModel
class DetailViewModel @Inject constructor(
    private val coordinateRepository: CoordinateRepository,
    private val searchHistoryRepository: SearchHistoryRepository,
) : ViewModel() {
    private val _coordinate = MutableStateFlow<CoordinatePhase>(CoordinatePhase.Loading)
```

```
    val coordinate: StateFlow<CoordinatePhase> = _coordinate.asStateFlow()
}
```

이 ViewModel은 크지 않다. 하지만 역할은 명확하다.

1. 도로명 주소를 좌표 상태로 변환한다.
2. 상세 시트 진입을 최근 목록 누적으로 바꾼다.
3. 즐겨찾기 토글과 현재 주소의 즐겨찾기 여부를 관리한다.

여기서 하지 않는 것

- 주소 데이터 자체를 다시 fetch하지 않는다.
- 지도 UI 렌더링을 직접 알지 않는다.
- 복사 토스트나 하이라이트 상태를 관리하지 않는다.

즉, "UI에서만 의미 있는 짧은 상태"와 "비즈니스/영속 상태"를 잘 나누고 있다.

11. CoordinatePhase와 테스트

```
sealed interface CoordinatePhase {
    data object Loading : CoordinatePhase
    data class Success(val coordinate: Coordinate) : CoordinatePhase
    data class Failure(val error: AppError) : CoordinatePhase
}
```

이 타입은 상세 시트의 좌표 상태 전체를 설명한다. 성공 시 필요한 값은 `Coordinate`, 실패 시 필요한 값은 `AppError`만 담는다.

테스트가 보여주는 것

```
초기 상태는 Loading
fetchCoordinate 성공 시 Success
fetchCoordinate 실패 시 Failure
```

`DetailViewModelTest`는 ViewModel이 "어떤 상태 전이를 해야 하는지"를 검증한다. 아직 UI 스냅샷 테스트는 없지만, 최소한 핵심 흐름은 문서화돼 있다.

> **학습 포인트**
> 좋은 테스트는 구현 세부보다 상태 전이를 고정한다. 이 파일은 그 기본 형태를 보여준다.

12. 이 구조를 바꾸면 어디가 깨질까

단순화 시도	깨질 가능성
좌표 fetch를 composable 안에서 직접 Retrofit 호출	테스트가 어려워지고, 화면 재구성/수명과 네트워크 호출이 뒤엉킨다.
복사 상태를 ViewModel로 올림	짧은 UI 상태가 영속 상태처럼 부풀어 올라 오히려 복잡해진다.
`CoordinatePhase` 대신 boolean 여러 개 사용	로딩/실패/성공 조합이 모호해지고 분기 실수가 늘어난다.
drag handle 클릭 시 곧바로 `onDismiss()`	hide 애니메이션과 부모 상태 제거 순서가 어긋날 수 있다.

13. 최신 흐름과 비교해 볼 포인트

- Compose side-effect 관점에서는 `LaunchedEffect(address.roadAddress)` 사용이 적절하다.
- Flow 수집은 가능하면 lifecycle-aware API로 일관화하는 팀 규칙을 세울 수 있다.
- 지금처럼 작은 UI 상태는 composable local state에 두는 흐름이 현대 Compose 감각과 잘 맞다.
- Header 애니메이션은 "과한 motion"이 아니라 정보 피드백에 붙은 motion이라 좋은 예다.

현재 코드에서 추가로 생각해볼 개선

- 상세 시트 UI에 대한 Compose UI 테스트가 아직 없다.
- 좌표 재조회 중 이전 성공 상태를 잠깐 유지할지, 항상 spinner로 돌릴지 UX 정책을 명시할 수 있다.
- `rememberRecent(address)`를 sheet open signal로 보는 정책은 명확하지만, "실수 탭"까지 최근에 남는다는 trade-off가 있다.

14. 체크리스트

- `DetailSheet`가 왜 주소를 다시 fetch하지 않는지 설명할 수 있는가.
- `LaunchedEffect`의 key를 왜 `address.roadAddress`로 잡았는지 설명할 수 있는가.
- `rememberCopyHandler()` 안의 local state가 왜 ViewModel에 없고 composable에 있는지 설명할 수 있는가.
- `CoordinatePhase`가 boolean 3개보다 나은 이유를 설명할 수 있는가.
- `ClickableDragHandle`가 왜 바로 `onDismiss()`하지 않고 `sheetState.hide()`를 거치는지 설명할 수 있는가.

메모

출처

- Side-effects in Compose
- Bottom sheets
- ModalBottomSheet API reference
- Where to hoist state
- State in Compose
- Hilt and other Jetpack integrations

PART 9

Android Phase 8 Map Deep Link Workbook

PHASE 8 DETAILED WORKBOOK

지도 & 딥링크 워크북
WebView, AndroidView, Kakao HTML, 외부 지도 앱 연동

`zipkr`의 지도 기능을 기준으로 geocoding repository, HTML builder, `AndroidView` 기반 WebView 임베드, 카카오/네이버 외부 앱 deep link, web fallback까지 하나의 기능 체인으로 이해하는 인쇄용 상세 자료.

학습 목표
주소 텍스트가 어떻게 좌표가 되고, 그 좌표가 다시 WebView 지도와 외부 지도 앱 링크로 바뀌는지 설명할 수 있어야 한다.

대상 파일
`CoordinateRepositoryImpl.kt`, `KakaoMapHtmlBuilder.kt`, `KakaoMapWebView.kt`, `MapDeepLinks.kt`, `MapDeepLinkButtons.kt`, 관련 테스트

핵심 개념
AndroidView interop, WebView, custom URI scheme, web fallback, in-memory cache, graceful degradation

읽는 방식
좌표 생성 -> HTML 조립 -> WebView 렌더 -> 외부 앱 이동 순으로 따라간다.

작성일: 2026-05-06. Android Developers WebView, Views in Compose, deep links/App Links 문서를 반영했다.

1. 전체 지도 기능 흐름

```
roadAddress
  ↓
CoordinateRepository.fetchCoordinate()
  ↓
Coordinate(longitude, latitude)
  ├─ KakaoMapWebView(coord, jsKey)
  │     ↓
  │  buildKakaoMapHtml()
  │     ↓
  │  WebView.loadDataWithBaseURL(...)
  └─ MapDeepLinkButtons(coord, address)
        ├─ kakaomap://... or nmap://...
        └─ https://map.kakao.com / https://map.naver.com fallback
```

즉, 이 기능은 "지도 화면 하나"가 아니라 네 계층의 협업이다. 주소를 좌표로 바꾸는 것과, 좌표를 화면/외부 앱 이동으로 바꾸는 것은 분리되어 있다.

> **핵심 문장**
> 지도 기능의 본질은 "맵 SDK 붙이기"가 아니라, 주소를 다양한 소비 형태로 변환하는 것이다.

2. CoordinateRepositoryImpl: 주소를 좌표로 번역하는 계층

```
@Singleton
class CoordinateRepositoryImpl @Inject constructor(
    private val api: KakaoLocalApi,
) : CoordinateRepository {
    private val cache = ConcurrentHashMap<String, Coordinate>()
}
```

이 구현체는 카카오 로컬 API 응답을 앱 내부의 `Coordinate` 모델로 바꾸는 책임을 가진다. 그리고 매우 중요한 부가 기능 하나가 더 있다. 바로 in-memory 캐시다.

왜 캐시가 필요한가

- 같은 주소 상세 시트를 다시 열 때 불필요한 geocoding 재호출을 줄인다.
- 지도 시트 열고 닫기 동작이 체감상 빨라진다.
- Repository 계층이 성능 최적화를 숨겨 주므로 UI는 몰라도 된다.

```
val cached = cache[roadAddress]
return if (cached != null) {
    Result.Success(cached)
```

```
} else {
    runCatching { api.geocode(roadAddress) } ...
}
```

> **설계 감각**
> 이 캐시는 영구 저장이 아니라 세션 최적화용 메모리 캐시다. "정확한 장기 저장"이 아니라 "같은 주소 반복 조회 흡수"가 목표다.

3. API 성공인데도 실패일 수 있다

```
private fun handleSuccess(
    roadAddress: String,
    first: KakaoDocument?,
): Result<Coordinate> {
    if (first == null) return Result.Failure(AppError.ApiBadResponse(EMPTY_DOCUMENTS_CODE))
    val coord = first.toCoordinate()
    cache[roadAddress] = coord
    return Result.Success(coord)
}
```

여기서 배우기 좋은 포인트는 "HTTP 200 OK"가 곧 앱 입장에서 성공이 아니라는 점이다. 카카오 응답이 와도 `documents`가 비어 있으면 사용자 입장에서는 좌표를 못 받은 것이므로 실패다.

상황	앱의 판단
캐시에 주소가 있음	즉시 `Result.Success`
HTTP 성공 + documents 첫 원소 있음	`Coordinate`로 변환 후 성공
HTTP 성공 + documents 비어 있음	`AppError.ApiBadResponse("KAKAO_EMPTY")`
`IOException`	네트워크 실패
그 외 예외	알 수 없는 실패

4. buildKakaoMapHtml: HTML 문자열을 코드로 조립하는 이유

```
fun buildKakaoMapHtml(
    coord: Coordinate,
    jsKey: String,
    level: Int = 3,
): String = """
<!DOCTYPE html>
<html>
  ...
  <script src="//dapi.kakao.com/v2/maps/sdk.js?appkey=$jsKey&autoload=false"></script>
  ...
  var center = new kakao.maps.LatLng(${coord.latitude}, ${coord.longitude});
  var map = new kakao.maps.Map(... { center: center, level: $level });
  new kakao.maps.Marker({ position: center, map: map });
"""
```

이 함수는 카카오맵 Web SDK를 위한 최소 HTML 문서를 문자열로 만든다. 파일로 따로 두지 않고 함수로 두는 이유는 좌표와 JS 키를 매번 안전하게 주입해야 하기 때문이다.

여기서 읽어야 할 포인트

- `100vw / 100vh`를 써서 WebView 내부 map div가 꽉 차게 만든다.
- `autoload=false` 후 `kakao.maps.load(...)`로 SDK 준비 시점을 제어한다.
- 좌표는 문자열 보간으로 HTML 안에 직접 들어간다.
- `level`은 카카오맵 줌 레벨이다. 숫자가 작을수록 더 가깝다.

5. KakaoMapWebView: Compose 안에 View 넣기

```
@Composable
fun KakaoMapWebView(
    coord: Coordinate,
    jsKey: String,
    modifier: Modifier = Modifier,
    level: Int = 3,
) {
    val html = remember(coord, jsKey, level) { buildKakaoMapHtml(coord, jsKey, level) }
    AndroidView(
        factory = { context -> WebView(context).apply { ... } },
        update = { webView -> ... },
    )
}
```

Android Developers의 Views in Compose 가이드는 아직 Compose에 없는 View 계층을 넣고 싶을 때 `AndroidView`를 쓰라고 설명한다. WebView는 대표적인 사례다.

왜 `remember(coord, jsKey, level)` 인가

HTML 문자열 자체도 생성 비용이 있고, 무엇보다 동일 파라미터에서는 같은 문서를 재사용하는 게 의미가 있다. 좌표나 줌 레벨이 바뀔 때만 새 HTML을 만든다.

6. WebView 설정에서 꼭 짚어야 하는 줄

```
layoutParams = ViewGroup.LayoutParams(MATCH_PARENT, MATCH_PARENT)
settings.javaScriptEnabled = true
settings.domStorageEnabled = true
tag = html
setOnTouchListener { view, event ->
    if (event.action == MotionEvent.ACTION_DOWN) {
        view.parent.requestDisallowInterceptTouchEvent(true)
    }
    false
}
loadDataWithBaseURL(MAP_BASE_URL, html, MAP_MIME_TYPE, MAP_ENCODING, null)
```

코드	왜 필요한가
`MATCH_PARENT`	AndroidView 자식이 잘못 측정되어 html viewport 높이가 0이 되는 것을 막는다.
`javaScriptEnabled = true`	카카오 Web SDK가 자바스크립트 기반이라 필수다.
`domStorageEnabled = true`	웹 SDK 동작 호환성을 높이는 설정이다.
`tag = html`	현재 문서를 식별하는 간단한 캐시 키처럼 쓴다.
`requestDisallowInterceptTouchEvent(true)`	BottomSheet의 드래그 제스처가 지도 드래그를 가로채지 못하게 한다.
`loadDataWithBaseURL(...)`	HTML 문자열을 직접 로드하면서도 base URL 맥락을 준다.

> **중요한 주석의 의미**
> 코드 주석대로 `baseUrl = https://localhost` 는 카카오 web 플랫폼 도메인 검사 통과를 위한 장치다. 즉, 임의의 값이 아니라 SDK 제약을 우회하지 않고 맞춰 주는 값이다.

7. update 블록과 불필요한 reload 방지

```
update = { webView ->
    if (webView.tag != html) {
        webView.tag = html
        webView.loadDataWithBaseURL(MAP_BASE_URL, html, MAP_MIME_TYPE, MAP_ENCODING, null)
    }
}
```

`AndroidView` 의 `update` 는 recomposition마다 다시 호출될 수 있다. 그래서 같은 HTML인데도 매번 reload하면 지도가 깜빡이거나 SDK 로딩이 반복될 수 있다.

이 앱은 아주 단순한 방식으로 "현재 WebView에 실린 HTML이 같은지"를 비교해서 reload를 막는다. 이건 작은 코드지만 실사용 UX를 크게 지켜 준다.

8. MapDeepLinks: 외부 앱 이동 URL 생성기

```
fun kakaoMapDeepLink(coord: Coordinate?, fallbackAddress: String): String =
    if (coord != null) {
        "kakaomap://look?p=${coord.latitude},${coord.longitude}"
    } else {
        "kakaomap://search?q=${encode(fallbackAddress)}"
    }
```

```
fun naverMapDeepLink(coord: Coordinate?, name: String): String =
    if (coord != null) {
        "nmap://place?lat=${coord.latitude}&lng=${coord.longitude}&name=${encode(name)}"
    } else {
        "nmap://search?query=${encode(name)}"
    }
```

이 유틸은 좌표가 있을 때와 없을 때를 깔끔하게 분기한다. 좌표가 있으면 정확한 핀 진입, 없으면 주소 검색 진입이다.

왜 `encode()` 가 필요한가

주소는 공백과 한글을 포함하므로 URI에 바로 넣으면 깨질 수 있다. 그래서 UTF-8 URL 인코딩을 거친다.

9. 앱 미설치 대비: web fallback

```
private fun openOrFallback(
    context: Context,
    deepLink: String,
    webFallback: String,
) {
    val deepIntent = Intent(Intent.ACTION_VIEW, Uri.parse(deepLink))
    try {
        context.startActivity(deepIntent)
```

```
    } catch (_: ActivityNotFoundException) {
        context.startActivity(Intent(Intent.ACTION_VIEW, Uri.parse(webFallback)))
    }
}
```

이 함수가 보여주는 핵심은 graceful degradation이다. 외부 지도 앱이 설치되어 있지 않아도 기능 전체가 막히지 않는다.

상황	동작
카카오/네이버 앱 설치됨	custom URI scheme으로 해당 앱을 연다.
앱 미설치	`ActivityNotFoundException`을 잡고 웹 지도로 보낸다.
좌표 조회 실패	주소 검색 기반 fallback URL을 쓴다.

> **제품 관점**
> 사용자는 "정확한 핀"이 가장 좋지만, 최소한 "해당 주소를 지도에서 열어보는 것"만 되면 목적을 달성한다. 이 코드가 그 하한선을 보장한다.

10. MapDeepLinkButtons: 브랜드를 의미로 쓰기

```
private val KAKAO_YELLOW = Color(0xFFFEE500)
private val NAVER_GREEN = Color(0xFF03C75A)
```

이 버튼들은 단순한 액션 버튼이 아니라 "어느 서비스로 이동하는지"를 즉시 인지시키는 역할도 한다. 그래서 텍스트보다 색이 먼저 의미를 전달한다.

```
BrandButton(
    label = stringResource(R.string.map_open_kakao),
    container = KAKAO_YELLOW,
    content = KAKAO_TEXT,
    onClick = { openOrFallback(...) },
)
```

기술적으로는 작은 코드지만, UX적으로는 서비스 전환을 명확히 알려주는 중요한 장치다.

11. 테스트가 무엇을 보장하나

```
MapDeepLinksTest
  - 좌표가 있으면 카카오/네이버 deep link가 lat lng를 사용한다
  - 좌표가 없으면 search query fallback을 쓴다
  - web fallback URL 형식이 맞다

KakaoMapHtmlBuilderTest
  - SDK script tag와 jsKey가 포함된다
  - 위도/경도와 Marker 코드가 포함된다
  - level 값이 HTML에 반영된다
```

이 테스트들은 WebView를 직접 그리진 않지만, 그 앞 단계인 "문자열 조립"과 "URI 조립"을 고정한다. 즉, 가장 깨지기 쉬운 경계값 로직을 빠르게 검증한다.

12. 최신 흐름과 비교해 볼 포인트

- Android Developers는 WebView보다 표준 브라우저 사용을 우선 권하지만, 앱 안에 tightly embedded map preview가 필요할 때 WebView는 합리적 선택이다.
- Compose에서 View 재사용이 필요할 때 `AndroidView`를 쓰는 흐름은 여전히 정석이다.
- Android의 own-domain 딥링크에는 App Links가 권장되지만, 이 코드는 외부 서비스 앱을 여는 outbound deep link라 성격이 다르다.
- WebView state 가이드를 보면 구성 변경 시 상태 보존이 중요할 수 있다. 현재 시트는 짧은 세션 preview라 단순 재생성 전략을 택한 셈이다.

추가로 생각해볼 개선

- WebView 생성/파괴 비용이 커진다면 state preservation 전략을 검토할 수 있다.
- 외부 앱 호출 전에 패키지 존재 여부를 미리 확인하는 방식도 가능하지만, 현재 try/catch fallback이 더 단순하다.
- in-memory 캐시가 커질 경우 `LruCache` 같은 정책형 캐시로 바꿀 수 있다.

13. 체크리스트

- `CoordinateRepositoryImpl`의 캐시가 왜 UI가 아니라 repository 안에 있는지 설명할 수 있는가.
- `documents`가 비면 왜 HTTP 성공이어도 앱은 실패로 처리하는지 설명할 수 있는가.
- `KakaoMapWebView`가 왜 `AndroidView`를 써야 하는지 설명할 수 있는가.
- `baseUrl = https://localhost`가 왜 필요한지 설명할 수 있는가.
- deep link와 web fallback을 왜 둘 다 제공하는지 설명할 수 있는가.
- App Links와 현재 코드의 custom URI outbound link가 어떻게 다른지 설명할 수 있는가.

메모

출처

- Build web apps in WebView
- Using Views in Compose
- WebView API reference
- Manage WebView state
- About deep links
- About App Links

PART 10

Android Phase 9 Build Release Ops Workbook

PHASE 9 DETAILED WORKBOOK

빌드 · 배포 · 운영 워크북
Gradle, Version Catalog, Signing, ProGuard, 최신 운영 방향

`zipkr`의 `build.gradle.kts`, `libs.versions.toml`, `AndroidManifest.xml`, `ZipkrApp.kt`, `proguard-rules.pro`, `gradle.properties`를 기준으로 안드로이드 앱의 실제 운영층을 공부하는 인쇄용 상세 자료.

학습 목표
앱 기능 코드만이 아니라, 빌드/서명/시크릿/릴리스 최적화/로그 수집까지 포함한 "출시 가능한 앱"의 구조를 설명할 수 있어야 한다.

대상 파일
`app/build.gradle.kts`, `gradle/libs.versions.toml`, `gradle.properties`, `AndroidManifest.xml`, `ZipkrApp.kt`, `proguard-rules.pro`, `themes.xml`

핵심 개념
Gradle Kotlin DSL, version catalog, build types, BuildConfig, manifest placeholders, signing, R8/ProGuard, startup, performance

읽는 방식
플러그인 -> android 블록 -> 시크릿/빌드 타입 -> 릴리스 최적화 -> 최신 운영 트렌드 순으로 본다.

작성일: 2026-05-06. Android Developers latest updates, build speed, startup, security checklist, Baseline Profiles, Gradle version catalog/configuration cache 문서를 반영했다.

1. 왜 빌드/운영 편을 따로 공부해야 하나

앱은 화면 코드만으로 출시되지 않는다. 실제 제품은 "어떤 라이브러리 버전을 쓰는가", "릴리스 APK를 어떻게 서명하는가", "API 키와 광고 ID를 어디서 주입하는가", "운영 로그는 어디로 보내는가", "릴리스 빌드에서 무엇을 줄이고 숨기는가"까지 포함해 완성된다.

```
plugins
  ↓
android { ... }
  ├─ defaultConfig
  ├─ signingConfigs
  ├─ buildTypes
  ├─ buildFeatures / composeOptions
  └─ packaging
  ↓
dependencies
  ↓
manifest / BuildConfig / placeholders
  ↓
release APK / AAB
  ↓
운영 로그 / 광고 / Crashlytics / startup behavior
```

핵심 문장
기능 코드는 "앱이 무엇을 하는가"를 정하고, 빌드/운영 코드는 "그 앱이 실제로 배포되고 유지될 수 있는가"를 정한다.

2. plugins 블록: 어떤 도구 체인을 올렸는가

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    id("kotlin-parcelize")
    alias(libs.plugins.kotlin.serialization)
    alias(libs.plugins.ksp)
    alias(libs.plugins.hilt)
    alias(libs.plugins.detekt)
    alias(libs.plugins.ktlint)
    alias(libs.plugins.google.services)
    alias(libs.plugins.firebase.crashlytics)
}
```

플러그인	역할
`com.android.application`	안드로이드 앱 모듈로 빌드하게 한다.
`org.jetbrains.kotlin.android`	안드로이드에서 Kotlin 코드 컴파일을 담당한다.
`kotlin-parcelize`	`@Parcelize` 지원용이다. 현재 핵심 흐름엔 많이 안 보이지만 앱 도구 상자에 포함돼 있다.
`kotlin-serialization`	JSON DTO를 `@Serializable`로 처리한다.
`ksp`	코드 생성 기반 라이브러리 처리. 현재 Hilt가 이를 사용한다.
`hilt`	DI graph 코드 생성과 통합.
`detekt`, `ktlint`	정적 분석과 코드 스타일 자동 검증.
`google-services`, `firebase-crashlytics`	Firebase 설정과 Crashlytics 업로드 연동.

Android Developers의 빌드 속도 가이드는 annotation processing에서 가능하면 `kapt` 보다 `KSP`를 권장한다. 현재 프로젝트가 이미 KSP로 가 있는 것은 좋은 방향이다.

3. android 블록: 앱의 빌드 신분증

```
android {
    namespace = "com.jewan.zipkr"
    compileSdk = 34
    defaultConfig {
        applicationId = "com.jewan.zipkr"
        minSdk = 24
        targetSdk = 34
        versionCode = 1
        versionName = "1.0.0"
    }
}
```

안드로이드 빌드에서 이 블록은 매우 중요하다. 앱의 패키지 정체성과 지원 범위, 버전, 빌드 기능이 모두 여기에 모인다.

속성	뜻
`namespace`	R 클래스와 generated code가 속할 패키지 네임스페이스다.
`applicationId`	스토어/기기에 설치되는 앱의 고유 식별자다.
`compileSdk`	컴파일할 때 참조하는 Android API 레벨이다.
`targetSdk`	최신 플랫폼 정책과 동작 변화에 얼마나 맞춰져 있는지의 기준이다.
`minSdk`	지원 가능한 최소 기기 버전이다.
`versionCode`, `versionName`	업데이트 판별용 정수 버전과 사용자 표시용 문자열 버전이다.

현재 코드 해석
이 프로젝트는 `compileSdk = 34`, `targetSdk = 34`다. 2026-05-06 기준 Android latest updates 페이지에는 Android 16이 안정 버전으로 표시되므로, 향후 플랫폼 대응 작업이 남아 있다고 보는 게 맞다.

4. local.properties, BuildConfig, manifestPlaceholders

```
val localProps = Properties().apply {
    val file = rootProject.file("local.properties")
    if (file.exists()) load(file.inputStream())
}
val jusoKey = localProps.getProperty("juso.api.key", "")
buildConfigField("String", "JUSO_API_KEY", "\"$jusoKey\"")
...
manifestPlaceholders["admobAppId"] = "ca-app-pub-3940256099942544~3347511713"
```

이 부분은 공부 가치가 높다. 시크릿이나 환경값을 코드에 하드코딩하지 않고, 로컬 파일에서 읽어 `BuildConfig` 또는 Manifest placeholder로 주입한다.

두 방식의 차이

- `BuildConfig` : Kotlin 코드에서 `BuildConfig.KAKAO_JS_KEY`처럼 읽는다.
- `manifestPlaceholders` : Manifest XML 안의 `${admobAppId}` 같은 자리에 치환된다.

중요한 현실
Android Developers 보안 체크리스트는 앱에 포함된 API 키는 역공학으로 노출될 수 있으므로, 진짜 비밀은 소스 밖에 두고 환경별로 분리하며 misuse 방지 정책을 같이 가져가라고 권장한다.

5. signingConfigs와 release guardrail

```
signingConfigs {
    create("release") {
        val storeFilePath = keystoreProps.getProperty("storeFile", "")
        if (storeFilePath.isNotBlank()) {
            storeFile = file(storeFilePath)
            ...
        } else {
            println("⚠ ... release 빌드는 서명 실패한다.")
        }
    }
}
```

릴리스 빌드는 반드시 서명돼야 한다. 이 코드는 `keystore.properties`가 없으면 조용히 잘못된 빌드를 만들기보다 경고를 내고 릴리스 실패 쪽으로 유도한다.

즉, "실수로 unsigned release를 만들지 않게 하는 안전장치"다.

왜 `keystore.properties`를 분리하나

- 키스토어 경로와 비밀번호를 저장소에 넣지 않기 위해서다.
- 개발자마다 다른 로컬 환경을 허용하기 위해서다.
- CI에서는 별도 secret 주입 전략으로 대체하기 쉽다.

6. buildTypes: debug와 release를 다르게 대우하기

```
debug {
    isMinifyEnabled = false
    applicationIdSuffix = ".debug"
}
release {
    isMinifyEnabled = true
    isShrinkResources = true
    signingConfig = signingConfigs.getByName("release")
    ...
}
```

빌드 타입	의도
`debug`	개발 편의 우선. 난독화 없이 빠르게 확인하고, 패키지 suffix로 릴리스 앱과 공존 가능하게 한다.
`release`	배포용. 코드 축소, 리소스 축소, 서명, 실제 광고 ID override 같은 운영 설정을 적용한다.

이 차이를 이해하면 "왜 디버그에선 잘 되는데 릴리스에서만 깨질 수 있는가"도 이해하기 쉬워진다.

7. R8 / ProGuard 규칙은 왜 필요한가

```
release {
    isMinifyEnabled = true
    isShrinkResources = true
    proguardFiles(
        getDefaultProguardFile("proguard-android-optimize.txt"),
        "proguard-rules.pro",
    )
}
```

```
# kotlinx-serialization
-keep,includedescriptorclasses class com.jewan.zipkr.**$$serializer { *; }

# Retrofit
-keepattributes RuntimeVisibleAnnotations, RuntimeVisibleParameterAnnotations
-keepclassmembers,allowshrinking,allowobfuscation interface * {
    @retrofit2.http.* <methods>;
}

# Hilt
-keep class dagger.hilt.android.** { *; }
-keep class * extends androidx.lifecycle.ViewModel { *; }
```

R8는 릴리스에서 코드를 줄이고 최적화한다. 그런데 reflection이나 code generation에 의존하는 라이브러리는 필요 클래스를 너무 공격적으로 지우면 런타임에서 죽을 수 있다. 그래서 keep 규칙이 필요하다.

현재 규칙이 보호하는 것

- Kotlin serialization serializer 객체
- Retrofit interface annotation metadata
- Hilt generated graph와 ViewModel 관련 타입

8. Version Catalog: 왜 libs.versions.toml을 쓰나

```
[versions]
kotlin = "1.9.24"
agp = "8.5.2"
...

[libraries]
androidx-core-ktx = { module = "androidx.core:core-ktx", version.ref = "core-ktx" }

[plugins]
android-application = { id = "com.android.application", version.ref = "agp" }
```

Gradle 공식 문서는 version catalog를 "의존성 목록을 중앙 관리하고 type-safe accessor로 참조하는 방식"이라고 설명한다. 이 프로젝트는 그 흐름을 이미 잘 따르고 있다.

장점

- 버전이 한곳에 모여 업그레이드 검토가 쉽다.
- `libs.compose.material3`, `libs.plugins.hilt` 같은 타입 안전 접근자를 쓴다.
- 문자열 하드코딩보다 오타와 중복을 줄인다.

9. gradle.properties와 현재 성능 설정

```
org.gradle.jvmargs=-Xmx2048m -Dfile.encoding=UTF-8
android.useAndroidX=true
android.nonTransitiveRClass=true
```

현재 프로젝트는 최소한의 성능/현대 설정을 이미 켜 두고 있다. 특히 `android.nonTransitiveRClass=true`는 multi-module 확장성을 고려한 좋은 기본값이다.

지금 보이는 상태

- Gradle wrapper는 `9.3.1`이다.
- configuration cache는 아직 명시적으로 켜져 있지 않다.
- parallel build도 주석 처리된 상태다.

Gradle 공식 문서는 Configuration Cache가 구성 단계 결과를 재사용해 빌드 성능을 높인다고 설명하고, Gradle 9 이후 선호 실행 모드라고 소개한다. 다만 프로젝트/플러그인 호환성을 먼저 확인해야 한다.

10. Manifest, Splash, Startup 초기화

```
<application
    android:name=".ZipkrApp"
    android:theme="@style/Theme.Zipkr">

    <meta-data
        android:name="com.google.android.gms.ads.APPLICATION_ID"
        android:value="${admobAppId}" />

    <activity
        android:name=".MainActivity"
        android:theme="@style/Theme.Zipkr.Splash">
```

```
@HiltAndroidApp
class ZipkrApp : Application() {
    override fun onCreate() {
        ...
        MobileAds.initialize(this) {}
    }
}
```

스플래시 테마, Application 초기화, 광고 SDK 앱 ID 주입, Crashlytics logging tree는 운영층의 일부다. Android Developers의 App Startup 문서는 초기화 순서를 명시적으로 관리하는 라이브러리를 제안하지만, 현재 앱은 초기화 수가 많지 않아 Application.onCreate에서 직접 처리하고 있다.

11. 현재 zipkr와 최신 공식 흐름 비교

항목	현재 상태	2026-05-06 기준 해석
Gradle / build script	Kotlin DSL + version catalog 사용	현대적인 구성이다. 유지해도 된다.
Annotation processing	KSP 사용	Android build speed 가이드 방향과 맞다.
Release minify	켜짐	좋다. Baseline Profile 도입 시 release 최적화와 궁합이 더 좋아진다.
Startup 최적화	직접 초기화	아직 단순하므로 괜찮다. 초기화가 늘면 App Startup 검토.
Build performance	Configuration Cache 미사용	Gradle 9 흐름상 도입 검토 가치가 있다.
Performance delivery	Baseline Profile 없음	최신 Android 품질 관점에서 가장 아쉬운 부분 중 하나다.

항목	현재 상태	2026-05-06 기준 해석
Platform target	SDK 34	Android 16 안정화 이후 단계적 상향 검토가 필요하다.

> **한 줄 진단**
> 구조는 현대적이지만, "운영 최적화" 관점에서는 Baseline Profile, 최신 target/compileSdk 점검, configuration cache 검토가 다음 단계다.

12. 지금 공부하면서 바로 가져가야 할 운영 습관

- API 키는 절대 소스에 하드코딩하지 말고, 환경 분리와 사용 제한 정책을 같이 가져간다.
- 릴리스 빌드는 항상 `isMinifyEnabled = true` 상태에서 실제 동작을 확인한다.
- 광고 SDK, Crashlytics, 앱 시작 초기화는 "언제 한 번만" 실행돼야 하는지 의식하고 배치한다.
- 의존성 버전 업은 `libs.versions.toml`를 중심으로 한 번에 검토한다.
- 플랫폼 최신 동향은 `latest-updates`와 각 라이브러리 release notes를 함께 본다.

13. 릴리스 체크리스트

- `keystore.properties`가 올바른가.
- `admob.app.id`, `admob.banner.unit.id`가 release 환경값으로 들어가는가.
- 행안부/카카오 키가 환경별로 분리돼 있는가.
- `release` 빌드에서 난독화 후 네트워크/DI/serialization이 정상 동작하는가.
- Crashlytics가 릴리스에서만 노이즈 없이 작동하는가.
- Manifest placeholder가 실제 배포 값으로 치환되는가.
- target/compileSdk 상향 시 동작 변경을 검토했는가.
- 성능 개선이 필요하면 Baseline Profile 도입을 검토했는가.

메모

출처

- Android latest updates
- Optimize your build speed
- App Startup
- Security checklist
- Baseline Profiles overview
- Gradle Version Catalogs
- Gradle Configuration Cache
- Android 16 features and changes list