

Flutter Foundations Masterclass (Korean Edition)

- [1 Flutter Foundations Masterclass \(Korean Edition\)](#)
 - [1.1 이 교재를 읽는 방법](#)
 - [1.2 프로젝트 구조 스캔 요약 \(실제 레포 기준\)](#)
 - [1.2.1 상위 디렉터리 요약](#)
 - [1.2.2 현재 기술 스택 분석](#)
 - [1.2.3 런타임 흐름 개요](#)
 - [1.2.4 레이어 아키텍처 개요](#)
 - [1.2.5 이 프로젝트에서 초보가 먼저 건드려도 안전한 파일](#)
 - [1.2.6 건드리면 위험한 파일 \(주의\)](#)
 - [1.2.7 기능 추가 권장 작업 순서 \(체크리스트\)](#)
 - [1.3 공통 검증 게이트](#)
- [2 Chapter 1. Flutter와 Dart를 왜 배워야 하는가](#)
 - [2.1 왜 필요한가](#)
 - [2.2 학습 목표](#)
 - [2.3 사전지식](#)
 - [2.4 핵심 개념 설명](#)
 - [2.4.1 1\) 개념 정의](#)
 - [2.4.2 2\) 동작 원리](#)
 - [2.4.3 3\) 실무 적용 절차](#)
 - [2.4.4 4\) 프로젝트 적용 포인트](#)
 - [2.5 실무 예제 코드](#)
 - [2.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [2.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [2.6 나쁜 예 vs 좋은 예](#)
 - [2.7 디버깅 포인트](#)
 - [2.8 성능/보안/유지보수 체크포인트](#)
 - [2.9 장 마무리 요약](#)
 - [2.10 퀴즈 5문항](#)
 - [2.10.1 정답 및 해설](#)
 - [2.11 실습 과제](#)
 - [2.12 실습 정답 코드 또는 해설](#)
 - [2.13 실무에서 바로 쓰는 체크리스트](#)
- [3 Chapter 2. Dart 핵심 문법: 변수, null-safety, class, async/await](#)
 - [3.1 왜 필요한가](#)
 - [3.2 학습 목표](#)
 - [3.3 사전지식](#)
 - [3.4 핵심 개념 설명](#)
 - [3.4.1 1\) 개념 정의](#)
 - [3.4.2 2\) 동작 원리](#)
 - [3.4.3 3\) 실무 적용 절차](#)
 - [3.4.4 4\) 프로젝트 적용 포인트](#)
 - [3.5 실무 예제 코드](#)
 - [3.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [3.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [3.6 나쁜 예 vs 좋은 예](#)
 - [3.7 디버깅 포인트](#)
 - [3.8 성능/보안/유지보수 체크포인트](#)
 - [3.9 장 마무리 요약](#)

- [3.10 퀴즈 5문항](#)
 - [3.10.1 정답 및 해설](#)
- [3.11 실습 과제](#)
- [3.12 실습 정답 코드 또는 해설](#)
- [3.13 실무에서 바로 쓰는 체크리스트](#)
- [4 Chapter 3. Widget 사고방식: Everything is Widget](#)
 - [4.1 왜 필요한가](#)
 - [4.2 학습 목표](#)
 - [4.3 사전지식](#)
 - [4.4 핵심 개념 설명](#)
 - [4.4.1\) 개념 정의](#)
 - [4.4.2\) 동작 원리](#)
 - [4.4.3\) 실무 적용 절차](#)
 - [4.4.4\) 프로젝트 적용 포인트](#)
 - [4.5 실무 예제 코드](#)
 - [4.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [4.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [4.6 나쁜 예 vs 좋은 예](#)
 - [4.7 디버깅 포인트](#)
 - [4.8 성능/보안/유지보수 체크포인트](#)
 - [4.9 장 마무리 요약](#)
 - [4.10 퀴즈 5문항](#)
 - [4.10.1 정답 및 해설](#)
 - [4.11 실습 과제](#)
 - [4.12 실습 정답 코드 또는 해설](#)
 - [4.13 실무에서 바로 쓰는 체크리스트](#)
- [5 Chapter 4. 레이아웃 기초: Row/Column/Stack/Flex/Expanded/Padding](#)
 - [5.1 왜 필요한가](#)
 - [5.2 학습 목표](#)
 - [5.3 사전지식](#)
 - [5.4 핵심 개념 설명](#)
 - [5.4.1\) 개념 정의](#)
 - [5.4.2\) 동작 원리](#)
 - [5.4.3\) 실무 적용 절차](#)
 - [5.4.4\) 프로젝트 적용 포인트](#)
 - [5.5 실무 예제 코드](#)
 - [5.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [5.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [5.6 나쁜 예 vs 좋은 예](#)
 - [5.7 디버깅 포인트](#)
 - [5.8 성능/보안/유지보수 체크포인트](#)
 - [5.9 장 마무리 요약](#)
 - [5.10 퀴즈 5문항](#)
 - [5.10.1 정답 및 해설](#)
 - [5.11 실습 과제](#)
 - [5.12 실습 정답 코드 또는 해설](#)
 - [5.13 실무에서 바로 쓰는 체크리스트](#)
- [6 Chapter 5. 화면 구조 실전: Scaffold/AppBar/BottomNavigation/SafeArea](#)
 - [6.1 왜 필요한가](#)
 - [6.2 학습 목표](#)
 - [6.3 사전지식](#)
 - [6.4 핵심 개념 설명](#)

- [6.4.1 1\) 개념 정의](#)
- [6.4.2 2\) 동작 원리](#)
- [6.4.3 3\) 실무 적용 절차](#)
- [6.4.4 4\) 프로젝트 적용 포인트](#)
- [6.5 실무 예제 코드](#)
 - [6.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [6.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
- [6.6 나쁜 예 vs 좋은 예](#)
- [6.7 디버깅 포인트](#)
- [6.8 성능/보안/유지보수 체크포인트](#)
- [6.9 장 마무리 요약](#)
- [6.10 퀴즈 5문항](#)
 - [6.10.1 정답 및 해설](#)
- [6.11 실습 과제](#)
- [6.12 실습 정답 코드 또는 해설](#)
- [6.13 실무에서 바로 쓰는 체크리스트](#)
- [7 Chapter 6. 상태관리 기초 setState\(\)와 함께](#)
 - [7.1 왜 필요한가](#)
 - [7.2 학습 목표](#)
 - [7.3 사전지식](#)
 - [7.4 핵심 개념 설명](#)
 - [7.4.1 1\) 개념 정의](#)
 - [7.4.2 2\) 동작 원리](#)
 - [7.4.3 3\) 실무 적용 절차](#)
 - [7.4.4 4\) 프로젝트 적용 포인트](#)
 - [7.5 실무 예제 코드](#)
 - [7.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [7.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [7.6 나쁜 예 vs 좋은 예](#)
 - [7.7 디버깅 포인트](#)
 - [7.8 성능/보안/유지보수 체크포인트](#)
 - [7.9 장 마무리 요약](#)
 - [7.10 퀴즈 5문항](#)
 - [7.10.1 정답 및 해설](#)
 - [7.11 실습 과제](#)
 - [7.12 실습 정답 코드 또는 해설](#)
 - [7.13 실무에서 바로 쓰는 체크리스트](#)
- [8 Chapter 7. Riverpod 핵심: Provider/Notifier/DI](#)
 - [8.1 왜 필요한가](#)
 - [8.2 학습 목표](#)
 - [8.3 사전지식](#)
 - [8.4 핵심 개념 설명](#)
 - [8.4.1 1\) 개념 정의](#)
 - [8.4.2 2\) 동작 원리](#)
 - [8.4.3 3\) 실무 적용 절차](#)
 - [8.4.4 4\) 프로젝트 적용 포인트](#)
 - [8.5 실무 예제 코드](#)
 - [8.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [8.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [8.6 나쁜 예 vs 좋은 예](#)
 - [8.7 디버깅 포인트](#)
 - [8.8 성능/보안/유지보수 체크포인트](#)

- [8.9 장 마무리 요약](#)
- [8.10 퀴즈 5문항](#)
 - [8.10.1 정답 및 해설](#)
- [8.11 실습 과제](#)
- [8.12 실습 정답 코드 또는 해설](#)
- [8.13 실무에서 바로 쓰는 체크리스트](#)
- [9 Chapter 8. 라우팅\(go_router\) 구조와 딥링크](#)
 - [9.1 왜 필요한가](#)
 - [9.2 학습 목표](#)
 - [9.3 사전지식](#)
 - [9.4 핵심 개념 설명](#)
 - [9.4.1 1\) 개념 정의](#)
 - [9.4.2 2\) 동작 원리](#)
 - [9.4.3 3\) 실무 적용 절차](#)
 - [9.4.4 4\) 프로젝트 적용 포인트](#)
 - [9.5 실무 예제 코드](#)
 - [9.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [9.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [9.6 나쁜 예 vs 좋은 예](#)
 - [9.7 디버깅 포인트](#)
 - [9.8 성능/보안/유지보수 체크포인트](#)
 - [9.9 장 마무리 요약](#)
 - [9.10 퀴즈 5문항](#)
 - [9.10.1 정답 및 해설](#)
 - [9.11 실습 과제](#)
 - [9.12 실습 정답 코드 또는 해설](#)
 - [9.13 실무에서 바로 쓰는 체크리스트](#)
- [10 Chapter 9. 앱 테마: ThemeData, ColorScheme, TextTheme](#)
 - [10.1 왜 필요한가](#)
 - [10.2 학습 목표](#)
 - [10.3 사전지식](#)
 - [10.4 핵심 개념 설명](#)
 - [10.4.1 1\) 개념 정의](#)
 - [10.4.2 2\) 동작 원리](#)
 - [10.4.3 3\) 실무 적용 절차](#)
 - [10.4.4 4\) 프로젝트 적용 포인트](#)
 - [10.5 실무 예제 코드](#)
 - [10.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [10.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [10.6 나쁜 예 vs 좋은 예](#)
 - [10.7 디버깅 포인트](#)
 - [10.8 성능/보안/유지보수 체크포인트](#)
 - [10.9 장 마무리 요약](#)
 - [10.10 퀴즈 5문항](#)
 - [10.10.1 정답 및 해설](#)
 - [10.11 실습 과제](#)
 - [10.12 실습 정답 코드 또는 해설](#)
 - [10.13 실무에서 바로 쓰는 체크리스트](#)
- [11 Chapter 10. 비동기/네트워크/JSON/애러 처리](#)
 - [11.1 왜 필요한가](#)
 - [11.2 학습 목표](#)
 - [11.3 사전지식](#)

- [11.4 핵심 개념 설명](#)
 - [11.4.1 1\) 개념 정의](#)
 - [11.4.2 2\) 동작 원리](#)
 - [11.4.3 3\) 실무 적용 절차](#)
 - [11.4.4 4\) 프로젝트 적용 포인트](#)
- [11.5 실무 예제 코드](#)
 - [11.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [11.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
- [11.6 나쁜 예 vs 좋은 예](#)
- [11.7 디버깅 포인트](#)
- [11.8 성능/보안/유지보수 체크포인트](#)
- [11.9 장 마무리 요약](#)
- [11.10 퀴즈 5문항](#)
 - [11.10.1 정답 및 해설](#)
- [11.11 실습 과제](#)
- [11.12 실습 정답 코드 또는 해설](#)
- [11.13 실무에서 바로 쓰는 체크리스트](#)
- [12 Chapter 11. 로컬 저장소\(shared_preferences\)와 앱 상태 복원](#)
 - [12.1 왜 필요한가](#)
 - [12.2 학습 목표](#)
 - [12.3 사전지식](#)
 - [12.4 핵심 개념 설명](#)
 - [12.4.1 1\) 개념 정의](#)
 - [12.4.2 2\) 동작 원리](#)
 - [12.4.3 3\) 실무 적용 절차](#)
 - [12.4.4 4\) 프로젝트 적용 포인트](#)
 - [12.5 실무 예제 코드](#)
 - [12.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [12.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [12.6 나쁜 예 vs 좋은 예](#)
 - [12.7 디버깅 포인트](#)
 - [12.8 성능/보안/유지보수 체크포인트](#)
 - [12.9 장 마무리 요약](#)
 - [12.10 퀴즈 5문항](#)
 - [12.10.1 정답 및 해설](#)
 - [12.11 실습 과제](#)
 - [12.12 실습 정답 코드 또는 해설](#)
 - [12.13 실무에서 바로 쓰는 체크리스트](#)
- [13 Chapter 12. 아키텍처 설계: presentation/domain/data 분리](#)
 - [13.1 왜 필요한가](#)
 - [13.2 학습 목표](#)
 - [13.3 사전지식](#)
 - [13.4 핵심 개념 설명](#)
 - [13.4.1 1\) 개념 정의](#)
 - [13.4.2 2\) 동작 원리](#)
 - [13.4.3 3\) 실무 적용 절차](#)
 - [13.4.4 4\) 프로젝트 적용 포인트](#)
 - [13.5 실무 예제 코드](#)
 - [13.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [13.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [13.6 나쁜 예 vs 좋은 예](#)
 - [13.7 디버깅 포인트](#)

- [13.8 성능/보안/유지보수 체크포인트](#)
- [13.9 장 마무리 요약](#)
- [13.10 퀴즈 5문항](#)
 - [13.10.1 정답 및 해설](#)
- [13.11 실습 과제](#)
- [13.12 실습 정답 코드 또는 해설](#)
- [13.13 실무에서 바로 쓰는 체크리스트](#)
- [14 Chapter 13. 광고 기본: AdMob 배너 + UMP 등의](#)
 - [14.1 왜 필요한가](#)
 - [14.2 학습 목표](#)
 - [14.3 사전지식](#)
 - [14.4 핵심 개념 설명](#)
 - [14.4.1\) 개념 정의](#)
 - [14.4.2\) 동작 원리](#)
 - [14.4.3\) 실무 적용 절차](#)
 - [14.4.4\) 프로젝트 적용 포인트](#)
 - [14.5 실무 예제 코드](#)
 - [14.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [14.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [14.6 나쁜 예 vs 좋은 예](#)
 - [14.7 디버깅 포인트](#)
 - [14.8 성능/보안/유지보수 체크포인트](#)
 - [14.9 장 마무리 요약](#)
 - [14.10 퀴즈 5문항](#)
 - [14.10.1 정답 및 해설](#)
 - [14.11 실습 과제](#)
 - [14.12 실습 정답 코드 또는 해설](#)
 - [14.13 실무에서 바로 쓰는 체크리스트](#)
- [15 Chapter 14. 테스트 전략: 단위/위젯/통합과 품질 게이트](#)
 - [15.1 왜 필요한가](#)
 - [15.2 학습 목표](#)
 - [15.3 사전지식](#)
 - [15.4 핵심 개념 설명](#)
 - [15.4.1\) 개념 정의](#)
 - [15.4.2\) 동작 원리](#)
 - [15.4.3\) 실무 적용 절차](#)
 - [15.4.4\) 프로젝트 적용 포인트](#)
 - [15.5 실무 예제 코드](#)
 - [15.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [15.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [15.6 나쁜 예 vs 좋은 예](#)
 - [15.7 디버깅 포인트](#)
 - [15.8 성능/보안/유지보수 체크포인트](#)
 - [15.9 장 마무리 요약](#)
 - [15.10 퀴즈 5문항](#)
 - [15.10.1 정답 및 해설](#)
 - [15.11 실습 과제](#)
 - [15.12 실습 정답 코드 또는 해설](#)
 - [15.13 실무에서 바로 쓰는 체크리스트](#)
- [16 Chapter 15. 디버깅: 로그/스택트레이스/재현 전략](#)
 - [16.1 왜 필요한가](#)
 - [16.2 학습 목표](#)

- [16.3 사전지식](#)
 - [16.4 핵심 개념 설명](#)
 - [16.4.1\) 개념 정의](#)
 - [16.4.2\) 동작 원리](#)
 - [16.4.3\) 실무 적용 절차](#)
 - [16.4.4\) 프로젝트 적용 포인트](#)
 - [16.5 실무 예제 코드](#)
 - [16.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [16.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [16.6 나쁜 예 vs 좋은 예](#)
 - [16.7 디버깅 포인트](#)
 - [16.8 성능/보안/유지보수 체크포인트](#)
 - [16.9 장 마무리 요약](#)
 - [16.10 퀴즈 5문항](#)
 - [16.10.1 정답 및 해설](#)
 - [16.11 실습 과제](#)
 - [16.12 실습 정답 코드 또는 해설](#)
 - [16.13 실무에서 바로 쓰는 체크리스트](#)
- [17 Chapter 16. 빌드/릴리즈: Android AAB/버전/서명](#)
 - [17.1 왜 필요한가](#)
 - [17.2 학습 목표](#)
 - [17.3 사전지식](#)
 - [17.4 핵심 개념 설명](#)
 - [17.4.1\) 개념 정의](#)
 - [17.4.2\) 동작 원리](#)
 - [17.4.3\) 실무 적용 절차](#)
 - [17.4.4\) 프로젝트 적용 포인트](#)
 - [17.5 실무 예제 코드](#)
 - [17.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [17.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [17.6 나쁜 예 vs 좋은 예](#)
 - [17.7 디버깅 포인트](#)
 - [17.8 성능/보안/유지보수 체크포인트](#)
 - [17.9 장 마무리 요약](#)
 - [17.10 퀴즈 5문항](#)
 - [17.10.1 정답 및 해설](#)
 - [17.11 실습 과제](#)
 - [17.12 실습 정답 코드 또는 해설](#)
 - [17.13 실무에서 바로 쓰는 체크리스트](#)
 - [18 Chapter 17. CI/CD 기초: GitHub Actions 파이프라인 읽기](#)
 - [18.1 왜 필요한가](#)
 - [18.2 학습 목표](#)
 - [18.3 사전지식](#)
 - [18.4 핵심 개념 설명](#)
 - [18.4.1\) 개념 정의](#)
 - [18.4.2\) 동작 원리](#)
 - [18.4.3\) 실무 적용 절차](#)
 - [18.4.4\) 프로젝트 적용 포인트](#)
 - [18.5 실무 예제 코드](#)
 - [18.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [18.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [18.6 나쁜 예 vs 좋은 예](#)

- [18.7 디버깅 포인트](#)
- [18.8 성능/보안/유지보수 체크포인트](#)
- [18.9 장 마무리 요약](#)
- [18.10 퀴즈 5문항](#)
 - [18.10.1 정답 및 해설](#)
- [18.11 실습 과제](#)
- [18.12 실습 정답 코드 또는 해설](#)
- [18.13 실무에서 바로 쓰는 체크리스트](#)
- [19 Chapter 18. 성능 최적화: Render/Rebuild/메모리/이미지](#)
 - [19.1 왜 필요한가](#)
 - [19.2 학습 목표](#)
 - [19.3 사전지식](#)
 - [19.4 핵심 개념 설명](#)
 - [19.4.1\) 개념 정의](#)
 - [19.4.2\) 동작 원리](#)
 - [19.4.3 3\) 실무 적용 절차](#)
 - [19.4.4\) 프로젝트 적용 포인트](#)
 - [19.5 실무 예제 코드](#)
 - [19.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [19.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [19.6 나쁜 예 vs 좋은 예](#)
 - [19.7 디버깅 포인트](#)
 - [19.8 성능/보안/유지보수 체크포인트](#)
 - [19.9 장 마무리 요약](#)
 - [19.10 퀴즈 5문항](#)
 - [19.10.1 정답 및 해설](#)
 - [19.11 실습 과제](#)
 - [19.12 실습 정답 코드 또는 해설](#)
 - [19.13 실무에서 바로 쓰는 체크리스트](#)
- [20 Chapter 19. 유지보수 전략: 리팩터링/코드리뷰/기술부채](#)
 - [20.1 왜 필요한가](#)
 - [20.2 학습 목표](#)
 - [20.3 사전지식](#)
 - [20.4 핵심 개념 설명](#)
 - [20.4.1\) 개념 정의](#)
 - [20.4.2\) 동작 원리](#)
 - [20.4.3 3\) 실무 적용 절차](#)
 - [20.4.4\) 프로젝트 적용 포인트](#)
 - [20.5 실무 예제 코드](#)
 - [20.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [20.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
 - [20.6 나쁜 예 vs 좋은 예](#)
 - [20.7 디버깅 포인트](#)
 - [20.8 성능/보안/유지보수 체크포인트](#)
 - [20.9 장 마무리 요약](#)
 - [20.10 퀴즈 5문항](#)
 - [20.10.1 정답 및 해설](#)
 - [20.11 실습 과제](#)
 - [20.12 실습 정답 코드 또는 해설](#)
 - [20.13 실무에서 바로 쓰는 체크리스트](#)
- [21 Chapter 20. 실전 프로젝트 적용: OurMatchWellFlutter 개선 로드맵](#)
 - [21.1 왜 필요한가](#)

- [21.2 학습 목표](#)
- [21.3 사전지식](#)
- [21.4 핵심 개념 설명](#)
 - [21.4.1\) 개념 정의](#)
 - [21.4.2\) 동작 원리](#)
 - [21.4.3\) 실무 적용 절차](#)
 - [21.4.4\) 프로젝트 적용 포인트](#)
- [21.5 실무 예제 코드](#)
 - [21.5.1 예제 1: 현재 장 핵심 패턴](#)
 - [21.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿](#)
- [21.6 나쁜 예 vs 좋은 예](#)
- [21.7 디버깅 포인트](#)
- [21.8 성능/보안/유지보수 체크포인트](#)
- [21.9 장 마무리 요약](#)
- [21.10 퀴즈 5문항](#)
 - [21.10.1 정답 및 해설](#)
- [21.11 실습 과제](#)
- [21.12 실습 정답 코드 또는 해설](#)
- [21.13 실무에서 바로 쓰는 체크리스트](#)
- [22 실습 종합 로드맵 \(12개 이상 요구 충족\)](#)
- [23 용어집 \(Glossary, 90개\)](#)
- [24 부록 A. 프로젝트 맞춤 운영 규칙](#)
- [25 부록 B. 7일 학습 운영표 \(예시\)](#)
- [26 부록 C. 초보가 자주 하는 실수 50선과 즉시 해결법](#)
- [27 마무리](#)
- [28 부록 D. 출시 전/후 실전 운영 시나리오 30선](#)
 - [28.1 D-1. 빌드/배포 시나리오 10선](#)
 - [28.2 D-2. 광고/정책 시나리오 10선](#)
 - [28.3 D-3. 상태/데이터 시나리오 10선](#)
 - [28.4 D-4. 상황별 대응 플레이북 \(상세\)](#)
 - [28.4.1 플레이북 1: “앱은 켜지는데 홀이 비어 있음”](#)
 - [28.4.2 플레이북 2: “탭 전환 시 이전 상태가 이상함”](#)
 - [28.4.3 플레이북 3: “설정 화면에서 동의 상태가 Unknown으로 고정”](#)
 - [28.4.4 플레이북 4: “스토어 심사 직전 최종 점검 순서”](#)
 - [28.4.5 플레이북 5: “핫픽스가 필요한 치명적 버그 발생”](#)
 - [28.5 D-5. 유지보수 체크시트 \(인쇄용\)](#)
 - [28.5.1 주간 체크시트](#)
 - [28.5.2 월간 체크시트](#)
 - [28.5.3 분기 체크시트](#)
 - [28.6 D-6. 1:1 코칭 질문 템플릿](#)

1 Flutter Foundations Masterclass (Korean Edition)

- 작성일: 2026-02-20
- 대상 프로젝트: /Users/jaebinchoi/Desktop/OurMatchWellFlutter
- 대상 독자: Flutter 초심자(바이브코딩 중심)
- 목표: 혼자 유지보수 가능한 Flutter 개발자 수준 도달

1.1 이 교재를 읽는 방법

이 교재는 다음 순서를 엄격하게 따릅니다.

1. 개념(What): 용어와 기본 의미를 이해합니다.
2. 원리(Why): 내부 동작 원리와 설계 이유를 이해합니다.
3. 실무 적용(How): 실제 코드로 적용하는 방법을 학습합니다.
4. 프로젝트 적용(Apply): OurMatchWellFlutter 코드에 직접 매핑합니다.

학습 중 모든 예제는 아래 검증 루틴을 기준으로 확인하세요.

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

1.2 프로젝트 구조 스캔 요약 (실제 레포 기준)

1.2.1 상위 디렉터리 요약

경로	역할	유지보수 중요도
pubspec.yaml	패키지/SDK/자산/버전 정의	매우 높음
lib/app/	앱 진입, 라우팅, 테마, 부트스트랩	매우 높음
lib/core/	광고, 동의, 저장소, 콘텐츠 인프라	매우 높음
lib/domain/	모델/유스케이스	높음
lib/features/	화면/상태/UX	매우 높음
lib/ui_components/	재사용 UI	높음
android/	Android 빌드/권한/서명/Manifest	매우 높음
ios/	iOS 번들/Info.plist/아이콘	높음
.github/workflows/	CI/CD 자동화	높음
docs/	출시 문서/정책 문서	중간

1.2.2 현재 기술 스택 분석

항목	현재 구성
상태관리	Riverpod (flutter_riverpod)
라우팅	go_router
광고	AdMob 배너 (google_mobile_ads)
동의	UMP 네이티브 채널 + ConsentState
로컬 저장소	shared_preferences

항목	현재 구성
테스트	flutter_test + 위젯/도메인 스모크
빌드 배포	GitHub Actions (CI + Play Internal CD)

1.2.3 런타임 흐름 개요

```
flowchart TD
A["main.dart"] --> B["startup bootstrap"]
B --> C["ProviderScope override sharedPreferences"]
C --> D["App"]
D --> E["MaterialApp.router"]
E --> F["GoRouter redirect onboarding"]
E --> G["AdsBootstrapper"]
G --> H["Consent gather"]
H --> I["canRequestAds true"]
I --> J["MobileAds initialize"]
```

1.2.4 레이어 아키텍처 개요

```
flowchart LR
UI["features/ui_components"] --> APP["app"]
UI --> DOMAIN["domain"]
UI --> CORE["core providers"]
DOMAIN --> CORE
CORE --> INFRA["persistence/content/ads"]
INFRA --> PLATFORM["Android iOS Native"]
```

1.2.5 이 프로젝트에서 초보가 먼저 건드려도 안전한 파일

파일	이유	권장 작업
lib/features/today/today_view.dart	UI 텍스트/배치 중심	문구 수정, 카드 라벨 개선
lib/features/explore/explore_view.dart	필터/검색 구조가 비교적 단순	검색 조건 확장
lib/features/my_library/my_library_view.dart	세그먼트 UI와 빈 상태 관리가 명확	빈 상태 문구, 정렬 개선
lib/ui_components/app_surfaces.dart	공용 카드/배지 컴포넌트	스타일 일관성 개선
lib/app/theme.dart	토론 기반 테마 집중	색상/텍스트 테마 조정

1.2.6 건드리면 위험한 파일 (주의)

파일	위험 이유	안전 수칙
android/app/build.gradle.kts	릴리즈 서명/AdMob App ID 검증 로직 포함	변경 전 백업 + 빌드 검증
android/app/src/main/AndroidManifest.xml	광고/네트워크 권한 및 메타데이터	권한 삭제 금지, placeholder 확인
lib/core/ads/consent/ump_consent_channel.dart	네이티브 채널 계약 불일치 시 런타임 오류	메서드명/파라미터 변경 금지

파일	위험 이유	안전 수칙
lib/core/persistence/preferences_store.dart	앱 데이터 포맷 저장소	키 변경 시 마이그레이션 필요
lib/app/router.dart	온보딩/탭 라우팅 핵심	redirect 조건 변경 시 회귀 테스트 필수

1.2.7 기능 추가 권장 작업 순서 (체크리스트)

- 요구사항을 화면/상태/저장소/정책으로 분해한다.
 - domain에 모델/유스케이스 변경 여부를 먼저 결정한다.
 - core에서 데이터 소스/Provider 계약을 정의한다.
 - features에서 UI를 최소 변경으로 연결한다.
 - 하드코딩/매직넘버를 상수나 토큰으로 이동한다.
 - flutter analyze, flutter test -j 1를 통과한다.
 - 광고/정책/권한 회귀 체크를 수행한다.
-

1.3 공통 검증 게이트

모든 장 실습은 아래 품질 게이트를 통과해야 완료로 간주합니다.

게이트	통과 조건
정적 분석	flutter analyze 경고/오류 0
테스트	flutter test -j 1 전부 통과
실행	에뮬레이터에서 크래시 없음
정책	동의 전 광고 요청 없음
유지보수	하드코딩/중복/매직넘버 제거

2 Chapter 1. Flutter와 Dart를 왜 배워야 하는가

2.1 왜 필요한가

왜 Flutter + Dart가 유지보수 비용을 낮추는지는 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(lib/main.dart)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

2.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

2.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: lib/main.dart

2.4 핵심 개념 설명

2.4.1 1) 개념 정의

- **정의(Definition):** Flutter와 Dart를 왜 배워야 하는가는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

2.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

2.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

2.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/main.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

2.5 실무 예제 코드

2.5.1 예제 1: 현재 장 핵심 패턴

```
// 앱 진입점: 외부 의존성(SharedPreferences)을 주입한 뒤 App을 실행한다.
Future<void> main() async {
  final prefs = await bootstrap();
  runApp(
    ProviderScope(
      overrides: [sharedPreferencesProvider.overrideWithValue(prefs)],
      child: const App(),
    ),
  );
}
```

2.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}
```

```
// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}
```

2.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 종복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

2.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 여러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

2.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?

- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

2.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

2.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

2.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

2.11 실습 과제

- 난이도: 초급
- 과제명: Flutter와 Dart를 왜 배워야 하는가 실습
- 요구사항:
 - 대상 파일(lib/main.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

2.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

2.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매진넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

3 Chapter 2. Dart 핵심 문법: 변수, null-safety, class, async/await

3.1 왜 필요한가

문법 실수로 인한 런타임 오류를 줄이는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(lib/domain/models/sentence.dart)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

3.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

3.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: lib/domain/models/sentence.dart

3.4 핵심 개념 설명

3.4.1 1) 개념 정의

- **정의(Definition):** Dart 핵심 문법: 변수, null-safety, class, async/await 는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

3.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

3.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

3.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/domain/models/sentence.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

3.5 실무 예제 코드

3.5.1 예제 1: 현재 장 핵심 패턴

```
// null-safe 모델 예시: 누락 값이 와도 기본값으로 안전하게 복구한다.
class ReviewState {
  const ReviewState({required this.dueAtEpochMs, required this.intervalDays});

  final int dueAtEpochMs;
  final int intervalDays;

  factory ReviewState.fromJson(Map<String, dynamic> json) {
    return ReviewState(
      dueAtEpochMs: json['dueAtEpochMs'] is int ? json['dueAtEpochMs'] as int : 0,
      intervalDays: json['intervalDays'] is int ? json['intervalDays'] as int : 0,
    );
  }
}
```

3.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
```

```

    }
    return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

3.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 종복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

3.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 여러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 넘겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

3.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?

- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

3.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

3.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

3.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

3.11 실습 과제

- 난이도: 초급
- 과제명: Dart 핵심 문법: 변수, null-safety, class, async/await 실습
- 요구사항:
 - 대상 파일(lib/domain/models/sentence.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
```

```
flutter test -j 1
flutter run
```

3.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

3.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

4 Chapter 3. Widget 사고방식: Everything is Widget

4.1 왜 필요한가

위젯 트리 사고방식으로 UI를 분해하는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(lib/features/today/today_view.dart)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

4.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

4.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: lib/features/today/today_view.dart

4.4 핵심 개념 설명

4.4.1 1) 개념 정의

- **정의(Definition):** Widget 사고방식: Everything is Widget는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향 도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

4.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

4.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

4.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/features/today/today_view.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

4.5 실무 예제 코드

4.5.1 예제 1: 현재 장 핵심 패턴

```
// 위젯을 작은 단위로 분해하면 재사용성과 테스트성이 올라간다.
class CircleToolbarButton extends StatelessWidget {
  const CircleToolbarButton({required this.icon, required this.onPressed, super.key});

  final IconData icon;
  final VoidCallback onPressed;

  @override
  Widget build(BuildContext context) {
    return IconButton(onPressed: onPressed, icon: Icon(icon));
  }
}
```

4.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
```

```

}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFFFE2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

4.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

4.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

4.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?

- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

4.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

4.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

4.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 탐색이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

4.11 실습 과제

- 난이도: 초급
- 과제명: Widget 사고방식: Everything is Widget 실습
- 요구사항:
 - 대상 파일(lib/features/today/today_view.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

4.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

4.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매진넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

5 Chapter 4. 레이아웃 기초: Row/Column/Stack/Flex/Expanded/Padding

5.1 왜 필요한가

레이아웃 깨짐 없이 반응형 UI를 만드는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(lib/features/explore/explore_view.dart)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

5.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

5.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: lib/features/explore/explore_view.dart

5.4 핵심 개념 설명

5.4.1 1) 개념 정의

- **정의(Definition):** 레이아웃 기초: Row/Column/Stack/Flex/Expanded/Padding 는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

5.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

5.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	자사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

5.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/features/explore/explore_view.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

5.5 실무 예제 코드

5.5.1 예제 1: 현재 장 핵심 패턴

```
// 반응형 레이아웃: Expanded와 SizedBox로 명시적 여백을 관리한다.
Row(
  children: [
    Expanded(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.start,
        children: const [Text('제목'), SizedBox(height: 6), Text('부제목')],
      ),
    ),
    const SizedBox(width: 10),
    const Icon(Icons.chevron_right_rounded),
  ],
)
```

5.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
}
```

```

return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
UiToken._();
static const Color selectedChip = Color(0xFF0F766E);
static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

5.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

5.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

5.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?

- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

5.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

5.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

5.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 탐색이 고여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

5.11 실습 과제

- 난이도: 초급
- 과제명: 레이아웃 기초: Row/Column/Stack/Flex/Expanded/Padding 실습
- 요구사항:
 - 대상 파일(lib/features/explore/explore_view.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:
 - 템플릿 테스트 작성
 - 템플릿 테스트 실행
 - 템플릿 테스트 결과 확인

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

5.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

5.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매진넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

6 Chapter 5. 화면 구조 실전: Scaffold/AppBar/BottomNavigation/SafeArea

6.1 왜 필요한가

앱 골격을 안정적으로 유지하는 구조는 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(`lib/app/shell.dart`)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

6.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

6.3 사전지식

- Flutter SDK 설치와 `flutter doctor` 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: `lib/app/shell.dart`

6.4 핵심 개념 설명

6.4.1 1) 개념 정의

- **정의(Definition):** 화면 구조 실전: Scaffold/AppBar/BottomNavigation/SafeArea 는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

6.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

6.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

6.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/app/shell.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

6.5 실무 예제 코드

6.5.1 예제 1: 현재 장 핵심 패턴

```
// 화면 골격: Scaffold + SafeArea + ListView 조합은 모바일 기본 패턴이다.
Scaffold(
  body: SafeArea(
    child: ListView(
      padding: const EdgeInsets.all(16),
      children: const [Text('헤더'), SizedBox(height: 12), Text('본문')],
    ),
  ),
)
```

6.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
```

```

class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

6.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

6.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

6.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?

- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

6.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

6.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

6.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 갱신 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

6.11 실습 과제

- 난이도: 초급
- 과제명: 화면 구조 실전: Scaffold/AppBar/BottomNavigation/SafeArea 실습
- 요구사항:
 - 대상 파일(lib/app/shell.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

6.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

6.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매진넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

7 Chapter 6. 상태관리 기초 setState와 한계

7.1 왜 필요한가

setState로 시작해 Riverpod로 확장하는 기준은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(lib/features/explore/explore_view.dart)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

7.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

7.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: lib/features/explore/explore_view.dart

7.4 핵심 개념 설명

7.4.1 1) 개념 정의

- **정의(Definition):** 상태관리 기초 setState와 한계는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이기 위한 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

7.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

7.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

7.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/features/explore/explore_view.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

7.5 실무 예제 코드

7.5.1 예제 1: 현재 장 핵심 패턴

```

class CounterView extends StatefulWidget {
  const CounterView({super.key});

  @override
  State<CounterView> createState() => _CounterViewState();
}

class _CounterViewState extends State<CounterView> {
  int _count = 0;

  /// 버튼 탭 카운트를 증가시킨다.
  void _increment() => setState(() => _count += 1);

  @override
  Widget build(BuildContext context) {
    return TextButton(onPressed: _increment, child: Text('${_count}'));
}
}

```

7.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}
```

7.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

7.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 여러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

7.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

7.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

7.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

7.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 갱신 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

7.11 실습 과제

- 난이도: 초급
- 과제명: 상태관리 기초 setState와 함께 실습
- 요구사항:
 - 대상 파일(lib/features/explore/explore_view.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1통과

- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

7.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

7.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

8 Chapter 7. Riverpod 핵심: Provider/Notifier/DI

8.1 왜 필요한가

의존성 주입과 테스트 가능한 상태관리는 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것 이 아니라, 실제 파일(lib/features/today/today_controller.dart)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

8.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

8.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: lib/features/today/today_controller.dart

8.4 핵심 개념 설명

8.4.1 1) 개념 정의

- **정의(Definition):** Riverpod 핵심: Provider/Notifier/DI 는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

8.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

8.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

8.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/features/today/today_controller.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

```
graph TD
    A[UI Widget] --> B[ref.watch(provider)]
    B --> C[Notifier/State]
    C --> D[Repository]
    D --> E[Store/Cache]
    E --> C
```

8.5 실무 예제 코드

8.5.1 예제 1: 현재 장 핵심 패턴

```
final todayPackUseCaseProvider = Provider<TodayPackUseCase>((ref) {
    final repo = ref.watch(contentRepositoryProvider);
    final prefs = ref.watch(preferencesStoreProvider);
    return TodayPackUseCase(repo: repo, prefs: prefs);
});

// today 화면에서 선택한 포커스 태그 상태를 제공한다.
final todayFocusProvider = StateProvider<String>((ref) => 'date');
```

8.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
```

```

if (selected) {
  return const Color(0xFF123456);
}
return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

8.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

8.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 여러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

8.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 없는가?

- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

8.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

8.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

8.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 탐색이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

8.11 실습 과제

- 난이도: 중급
- 과제명: Riverpod 핵심: Provider/Notifier/DI 실습
- 요구사항:
 - 대상 파일(lib/features/today/today_controller.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
```

```
flutter test -j 1
flutter run
```

8.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

8.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

9 Chapter 8. 라우팅(go_router) 구조와 딥링크

9.1 왜 필요한가

URL 기반 화면 이동과 리다이렉트 설계는 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것 이 아니라, 실제 파일(lib/app/router.dart)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

9.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

9.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: lib/app/router.dart

9.4 핵심 개념 설명

9.4.1 1) 개념 정의

- **정의(Definition):** 라우팅(go_router) 구조와 딥링크는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

9.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

9.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

9.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/app/router.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

```
graph LR
    O[/onboarding/] --> T[/today]
    T --> E[/explore]
    T --> M[/my]
    M --> S[/my/settings]
    T --> SD[/sentence/:id]
```

9.5 실무 예제 코드

9.5.1 예제 1: 현재 장 핵심 패턴

```
final routerProvider = Provider<GoRouter>((ref) {
    final prefs = ref.read(preferencesStoreProvider);
    return GoRouter(
        initialLocation: '/onboarding',
        refreshListenable: prefs,
        redirect: (context, state) {
            final onboardingCompleted = prefs.onboardingCompleted;
            final isOnboarding = state.matchedLocation == '/onboarding';
            if (!onboardingCompleted && !isOnboarding) return '/onboarding';
            if (onboardingCompleted && isOnboarding) return '/today';
            return null;
        },
    );
});
```

9.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}
```

9.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

9.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 여러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

9.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

9.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

9.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

9.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 갱신 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

9.11 실습 과제

- 난이도: 중급
- 과제명: 라우팅(go_router) 구조와 딥링크 실습
- 요구사항:
 - 대상 파일(lib/app/router.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1통과

- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

9.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

9.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

10 Chapter 9. 앱 테마: ThemeData, ColorScheme, TextTheme

10.1 왜 필요한가

디자인 시스템으로 UI 일관성을 만드는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(`lib/app/theme.dart`)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

10.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

10.3 사전지식

- Flutter SDK 설치와 `flutter doctor` 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: `lib/app/theme.dart`

10.4 핵심 개념 설명

10.4.1 1) 개념 정의

- **정의(Definition):** 앱 테마: ThemeData, ColorScheme, TextTheme 는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

10.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

10.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	자사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

10.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/app/theme.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

10.5 실무 예제 코드

10.5.1 예제 1: 현재 장 핵심 패턴

```
ThemeData buildAppTheme() {
  final base = ThemeData(useMaterial3: true);
  return base.copyWith(
    scaffoldBackgroundColor: const Color(0xFFFF8FAFC),
    textTheme: base.textTheme.copyWith(
      headlineLarge: const TextStyle(fontSize: 30, fontWeight: FontWeight.w800),
      bodyMedium: const TextStyle(fontSize: 15, height: 1.4),
    ),
  );
}
```

10.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}
```

```
// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}
```

10.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 종복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

10.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 여러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

10.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?

- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

10.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

10.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

10.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 갱신 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

10.11 실습 과제

- 난이도: 중급
- 과제명: 앱 테마: ThemeData, ColorScheme, TextTheme 실습
- 요구사항:
 - 대상 파일(lib/app/theme.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

10.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

10.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매진넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

11 Chapter 10. 비동기/네트워크/JSON/에러 처리

11.1 왜 필요한가

비동기 상태와 실패 복구를 제어하는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것 이 아니라, 실제 파일(`lib/core/content/content_store.dart`)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

11.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

11.3 사전지식

- Flutter SDK 설치와 `flutter doctor` 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: `lib/core/content/content_store.dart`

11.4 핵심 개념 설명

11.4.1 1) 개념 정의

- **정의(Definition):** 비동기/네트워크/JSON/에러 처리는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

11.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

11.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

11.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/core/content/content_store.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

11.5 실무 예제 코드

11.5.1 예제 1: 현재 장 핵심 패턴

```
// JSON 문자열을 Sentence 리스트로 안전하게 파싱한다.
List<Sentence>? tryDecodeSentences(String? raw) {
  if (raw == null || raw.isEmpty) return null;
  try {
    final list = (jsonDecode(raw) as List<dynamic>);
    return list
      .map((e) => Sentence.fromJson(e as Map<String, dynamic>))
      .toList(growable: false);
  } catch (_) {
    return null;
  }
}
```

11.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
```

```

}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

11.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

11.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

11.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?

- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

11.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

11.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

11.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 탐색이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

11.11 실습 과제

- 난이도: 중급
- 과제명: 비동기/네트워크/JSON/에러 처리 실습
- 요구사항:
 - 대상 파일(lib/core/content/content_store.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

11.12 실습 정답 코드 또는 해설

```

/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}

```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

11.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매진넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

12 Chapter 11. 로컬 저장소(shared_preferences)와 앱 상태 복원

12.1 왜 필요한가

앱 재실행 후에도 상태를 보존하는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(lib/core/persistence/preferences_store.dart)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

12.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

12.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: lib/core/persistence/preferences_store.dart

12.4 핵심 개념 설명

12.4.1 1) 개념 정의

- **정의(Definition):** 로컬 저장소(shared_preferences)와 앱 상태 복원은 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

12.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

12.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

12.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/core/persistence/preferences_store.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

12.5 실무 예제 코드

12.5.1 예제 1: 현재 장 핵심 패턴

```

/// 즐겨찾기 토큰: 저장소 반영 후 notifyListeners로 UI를 갱신한다.
void toggleFavorite(String sentenceId, {DateTime? now}) {
    final timestamp = now ?? DateTime.now();
    if (_favoriteIds.contains(sentenceId)) {
        _favoriteIds.remove(sentenceId);
        _reviewMap.remove(sentenceId);
    } else {
        _favoriteIds.add(sentenceId);
        _reviewMap[sentenceId] = ReviewState(
            dueAtEpochMs: timestamp.millisecondsSinceEpoch,
            intervalDays: 0,
        );
    }
    _persistFavorites();
    _persistReviewState();
    notifyListeners();
}

```

12.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}
```

12.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

12.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 여러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

12.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

12.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

12.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

12.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 갱신 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

12.11 실습 과제

- 난이도: 응용
- 과제명: 로컬 저장소(shared_preferences)와 앱 상태 복원 실습
- 요구사항:
 - 대상 파일(lib/core/persistence/preferences_store.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과

- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

12.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

12.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

13 Chapter 12. 아키텍처 설계: presentation/domain/data 분리

13.1 왜 필요한가

레이어 분리로 변경 영향도를 줄이는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것 이 아니라, 실제 파일(`lib/domain/usecases/today_pack_usecase.dart`)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

13.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

13.3 사전지식

- Flutter SDK 설치와 `flutter doctor` 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: `lib/domain/usecases/today_pack_usecase.dart`

13.4 핵심 개념 설명

13.4.1 1) 개념 정의

- **정의(Definition):** 아키텍처 설계: presentation/domain/data 분리는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

13.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

13.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

13.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/domain/usecases/today_pack_usecase.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

13.5 실무 예제 코드

13.5.1 예제 1: 현재 장 핵심 패턴

```
// 유스케이스는 UI 의존 없이 도메인 조합만 수행해야 테스트가 쉽다.
class TodayPackUseCase {
    TodayPackUseCase({required ContentRepository repo, required PreferencesStore prefs})
        : _repo = repo,
        _prefs = prefs;

    final ContentRepository _repo;
    final PreferencesStore _prefs;
}
```

13.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
    if (selected) {
        return const Color(0xFF123456);
    }
    return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
```

```

class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

13.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

13.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

13.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?

- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

13.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

13.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

13.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 갱신 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

13.11 실습 과제

- 난이도: 응용
- 과제명: 아키텍처 설계: presentation/domain/data 분리 실습
- 요구사항:
 - 대상 파일(lib/domain/usecases/today_pack_usecase.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

13.12 실습 정답 코드 또는 해설

```

/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}

```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

13.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매진넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

14 Chapter 13. 광고 기본: AdMob 배너 + UMP 동의

14.1 왜 필요한가

광고 정책 준수와 사용자 신뢰를 동시에 잡는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(`lib/core/ads/ads_service.dart`)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

14.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

14.3 사전지식

- Flutter SDK 설치와 `flutter doctor` 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: `lib/core/ads/ads_service.dart`

14.4 핵심 개념 설명

14.4.1 1) 개념 정의

- **정의(Definition):** 광고 기본: AdMob 배너 + UMP 동의는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

14.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

14.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

14.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/core/ads/ads_service.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

```

sequenceDiagram
    participant U as User
    participant App as Flutter App
    participant UMP as Consent SDK
    participant ADS as Mobile Ads SDK
    U->>App: App Launch
    App->>UMP: gatherConsent()
    UMP-->>App: canRequestAds=true/false
    alt true
        App->>ADS: initialize()
        ADS-->>App: ready
    else false
        App-->>U: no ad request
    end

```

14.5 실무 예제 코드

14.5.1 예제 1: 현재 장 핵심 패턴

```

/// 광고 SDK 초기화: 동의가 확보된 이후에만 수행한다.
Future<void> ensureInitialized() async {
    await _ref.read(consentControllerProvider.notifier).gatherConsent();
    final consent = _ref.read(consentControllerProvider);
    if (!consent.canRequestAds) return;
}

```

```

    await MobileAds.instance.initialize();
}

```

14.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```

// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

14.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

14.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.

9. 고친 뒤에는 반대 경로(negative path)도 검증한다.

10. “원인-해결-재검증” 3단계로 문서화한다.

14.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

14.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

14.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

14.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 타이밍이 고여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

14.11 실습 과제

- 난이도: 응용
- 과제명: 광고 기본: AdMob 배너 + UMP 동의 실습
- 요구사항:
 - 대상 파일(lib/core/ads/ads_service.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가

- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

14.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

14.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

15 Chapter 14. 테스트 전략: 단위/위젯/통합과 품질 게이트

15.1 왜 필요한가

회귀 버그를 방지하는 테스트 습관은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(test/feature_sanity_test.dart)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

15.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

15.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: test/feature_sanity_test.dart

15.4 핵심 개념 설명

15.4.1 1) 개념 정의

- **정의(Definition):** 테스트 전략: 단위/위젯/통합과 품질 게이트는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

15.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

15.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

15.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
test/feature_sanity_test.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

15.5 실무 예제 코드

15.5.1 예제 1: 현재 장 핵심 패턴

```
testWidgets('onboarding skip routes to today', (tester) async {
  await tester.pumpWidget(const App());
  await tester.tap(find.text('건너뛰기'));
  await tester.pumpAndSettle();
  expect(find.textContaining('오늘 바로 써먹는 문장과 패턴으로'), findsOneWidget);
});
```

15.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
```

```

static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

15.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 종복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

15.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

15.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

15.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

15.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

15.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

15.11 실습 과제

- 난이도: 응용
- 과제명: 테스트 전략: 단위/위젯/통합과 품질 게이트 실습
- 요구사항:
 - 대상 파일(test/feature_sanity_test.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

15.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\';
}
```

```
/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

15.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

16 Chapter 15. 디버깅: 로그/스택트레이스/재현 전략

16.1 왜 필요한가

문제 재현률을 높여 디버깅 시간을 줄이는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(`lib/core/support/app_logger.dart`)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

16.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

16.3 사전지식

- Flutter SDK 설치와 `flutter doctor` 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: `lib/core/support/app_logger.dart`

16.4 핵심 개념 설명

16.4.1 1) 개념 정의

- **정의(Definition):** 디버깅: 로그/스택트레이스/재현 전략은 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

16.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

16.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

16.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/core/support/app_logger.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

16.5 실무 예제 코드

16.5.1 예제 1: 현재 장 핵심 패턴

```
logger.error(
  AppLogCategory.consent,
  'Failed to gather UMP consent state.',
  error: error,
  stackTrace: stackTrace,
);
```

16.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
```

```

static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

16.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 종복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

16.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

16.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

16.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

16.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

16.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

16.11 실습 과제

- 난이도: 응용
- 과제명: 디버깅: 로그/스택트레이스/재현 전략 실습
- 요구사항:
 - 대상 파일(lib/core/support/app_logger.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

16.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\';
}
```

```
/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

16.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

17 Chapter 16. 빌드/릴리즈: Android AAB/버전/서명

17.1 왜 필요한가

릴리즈 실패를 사전에 막는 빌드 파이프라인은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(`android/app/build.gradle.kts`)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

17.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

17.3 사전지식

- Flutter SDK 설치와 `flutter doctor` 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: `android/app/build.gradle.kts`

17.4 핵심 개념 설명

17.4.1 1) 개념 정의

- **정의(Definition):** 빌드/릴리즈: Android AAB/버전/서명은 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

17.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

17.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

17.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
android/app/build.gradle.kts	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

17.5 실무 예제 코드

17.5.1 예제 1: 현재 장 핵심 패턴

```
if (isReleaseTask) {
    if (!hasKeystoreProperties) {
        throw GradleException("Missing android/key.properties")
    }
    if (adMobAppIdProp.isBlank()) {
        throw GradleException("Missing ADMOB_APP_ID")
    }
}
```

17.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
    if (selected) {
        return const Color(0xFF123456);
    }
    return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
```

```

UiToken._();
static const Color selectedChip = Color(0xFF0F766E);
static const Color unselectedChip = Color(0xFFFFE2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

17.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

17.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 여러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

17.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?

- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

17.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

17.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

17.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 갱신 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

17.11 실습 과제

- 난이도: 응용
- 과제명: 빌드/릴리즈: Android AAB/버전/서명 실습
- 요구사항:
 - 대상 파일(android/app/build.gradle.kts)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

17.12 실습 정답 코드 또는 해설

```

/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}

```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

17.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매진넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

18 Chapter 17. CI/CD 기초: GitHub Actions 파이프라인 읽기

18.1 왜 필요한가

자동검사/자동배포를 읽고 운영하는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것 이 아니라, 실제 파일(.github/workflows/ci.yml)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

18.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

18.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: .github/workflows/ci.yml

18.4 핵심 개념 설명

18.4.1 1) 개념 정의

- **정의(Definition):** CI/CD 기초: GitHub Actions 파이프라인 읽기는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

18.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

18.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	자사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

18.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
.github/workflows/ci.yml	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

```

flowchart TD
    PR[Pull Request] --> CI[CI: analyze + test]
    CI --> M{Pass?}
    M -->|Yes| Merge[maintain merge]
    M -->|No| Fix[Fix and re-run]
    Merge --> Tag[v* tag or manual]
    Tag --> CD[CD: build AAB + upload internal]
  
```

18.5 실무 예제 코드

18.5.1 예제 1: 현재 장 핵심 패턴

```

name: CI
on:
  pull_request:
  push:
    branches: [main]
jobs:
  analyze_test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: subosito/flutter-action@v2
      - run: flutter pub get
  
```

```
- run: flutter analyze
- run: flutter test -j 1
```

18.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
    if (selected) {
        return const Color(0xFF123456);
    }
    return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
    UiToken._();
    static const Color selectedChip = Color(0xFF0F766E);
    static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
    return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}
```

18.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

18.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.

9. 고친 뒤에는 반대 경로(negative path)도 검증한다.

10. “원인-해결-재검증” 3단계로 문서화한다.

18.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

18.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

18.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

18.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

18.11 실습 과제

- 난이도: 응용
- 과제명: CI/CD 기초: GitHub Actions 파이프라인 읽기 실습
- 요구사항:
 - 대상 파일(.github/workflows/ci.yml)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가

- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

18.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

18.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

19 Chapter 18. 성능 최적화: Render/Rebuild/메모리/이미지

19.1 왜 필요한가

느린 화면과 메모리 누수를 예방하는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것 이 아니라, 실제 파일(`lib/ui_components/sentence_card.dart`)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

19.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

19.3 사전지식

- Flutter SDK 설치와 `flutter doctor` 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: `lib/ui_components/sentence_card.dart`

19.4 핵심 개념 설명

19.4.1 1) 개념 정의

- **정의(Definition):** 성능 최적화: Render/Rebuild/메모리/이미지 는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

19.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

19.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

19.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/ui_components/sentence_card.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

19.5 실무 예제 코드

19.5.1 예제 1: 현재 장 핵심 패턴

```
// 불필요한 rebuild를 줄이기 위해 const 위젯과 지역 상태 분리를 사용한다.
class HeaderTitle extends StatelessWidget {
  const HeaderTitle({super.key});

  @override
  Widget build(BuildContext context) {
    return const Text('상황 탐색');
  }
}
```

19.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
```

```

class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

19.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

19.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

19.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?

- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

19.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

19.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

19.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 갱신 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

19.11 실습 과제

- 난이도: 응용
- 과제명: 성능 최적화: Render/Rebuild/메모리/이미지 실습
- 요구사항:
 - 대상 파일(lib/ui_components/sentence_card.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

19.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

19.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매진넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

20 Chapter 19. 유지보수 전략: 리팩터링/코드리뷰/기술부채

20.1 왜 필요한가

기능 추가 없이 코드 건강을 유지하는 법은 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(lib/core/content/content_repository.dart)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

20.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

20.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: lib/core/content/content_repository.dart

20.4 핵심 개념 설명

20.4.1 1) 개념 정의

- **정의(Definition):** 유지보수 전략: 리팩터링/코드리뷰/기술부채는 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

20.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

20.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	재사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

20.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
lib/core/content/content_repository.dart	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	애러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

20.5 실무 예제 코드

20.5.1 예제 1: 현재 장 핵심 패턴

```
/// 리팩터링 규칙: 중복 로직은 공용 함수로 추출한다.
List<T> filterByTag<T>(List<T> list, String tag, List<String> Function(T) tagsReader) {
  final filtered = list.where((item) => tagsReader(item).contains(tag)).toList(growable: false);
  return filtered.isEmpty ? list : filtered;
}
```

20.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```
// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
  if (selected) {
    return const Color(0xFF123456);
  }
  return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
```

}

```
/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}
```

20.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

20.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

20.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?
- 상태 객체가 불변(imutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

20.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

20.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

20.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

20.11 실습 과제

- 난이도: 응용
- 과제명: 유지보수 전략: 리팩터링/코드리뷰/기술부채 실습
- 요구사항:
 - 대상 파일(lib/core/content/content_repository.dart)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
flutter test -j 1
flutter run
```

20.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\';
}
```

```
/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

20.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

21 Chapter 20. 실전 프로젝트 적용: OurMatchWellFlutter 개선 로드맵

21.1 왜 필요한가

실제 레포에서 바로 실행 가능한 개선 순서는 이 프로젝트에서 반복적으로 발생하는 유지보수 비용을 줄이기 위한 핵심 역량입니다. 초보 단계에서 가장 흔한 실패는 “일단 동작만 되게” 구현한 뒤, 나중에 고치기 어렵게 되는 구조를 만드는 것입니다. 이 장에서는 개념을 암기하는 것이 아니라, 실제 파일(docs/PLAY_RELEASE.md)에서 어떤 결정이 품질을 좌우하는지를 기준으로 학습합니다.

21.2 학습 목표

- 이 장의 핵심 용어를 한국어/영어 병기로 설명할 수 있다.
- 동작 원리를 코드 흐름으로 그릴 수 있다.
- 안티패턴을 보고 왜 위험한지 설명할 수 있다.
- OurMatchWellFlutter에 맞는 수정 포인트를 고를 수 있다.
- 분석/테스트/실행 검증 루틴을 직접 수행할 수 있다.

21.3 사전지식

- Flutter SDK 설치와 flutter doctor 기본 점검
- Dart 파일 열람과 기본 편집기 사용법
- Git 기본 개념(commit/branch/revert)
- 이 장의 대상 파일: docs/PLAY_RELEASE.md

21.4 핵심 개념 설명

21.4.1 1) 개념 정의

- **정의(Definition):** 실전 프로젝트 적용: OurMatchWellFlutter 개선 로드맵은 화면 구현과 비즈니스 로직 사이의 경계를 명확히 하여, 변경 영향도를 줄이는 방법론입니다.
- **핵심 원칙(Core Principle):** 하드코딩 금지, 매직넘버 금지, 의존성 분리, 테스트 가능한 구조.
- **실무 기준(Production Criteria):** 릴리즈 빌드에서 실패 원인을 빠르게 추적 가능해야 하며, 팀원이 코드를 읽고 안전하게 수정할 수 있어야 합니다.

21.4.2 2) 동작 원리

1. 입력 이벤트(탭/검색/라우팅/초기화)가 발생합니다.
2. 상태 계층(UI 상태 vs 도메인 상태)을 분리합니다.
3. Provider 또는 유스케이스가 데이터 조합을 수행합니다.
4. 실패 가능 지점(네트워크/스토리지/채널)을 예외 처리합니다.
5. UI는 결과 상태/loading/error/data)만 렌더링합니다.
6. 로그는 카테고리 기반으로 기록합니다.
7. 테스트는 단위/위젯/스모크를 분리 실행합니다.
8. 품질 게이트(analyze/test/build)를 통과해야 배포 후보가 됩니다.

21.4.3 3) 실무 적용 절차

단계	작업	산출물
1	요구사항 분해	변경 범위 문서
2	데이터/상태 경계 정의	Provider/UseCase 설계
3	UI 컴포넌트 분해	자사용 위젯
4	실패 시나리오 정의	예외 처리 정책
5	로깅 포인트 지정	카테고리 로그
6	테스트 작성	단위/위젯 테스트
7	정적 분석/실행 검증	품질 리포트

21.4.4 4) 프로젝트 적용 포인트

현재 파일	관찰 포인트	개선 기준
docs/PLAY_RELEASE.md	기능이 몰려 있는지 확인	단일 책임(SRP) 유지
lib/app/theme.dart	스타일 하드코딩 여부	토큰 기반 관리
lib/core/*	에러 처리 누락 여부	실패 복구 경로 명시
test/*	회귀 테스트 부재 여부	최소 스모크 + 핵심 단위 테스트

```

flowchart TD
    R[Requirement] --> D[Design]
    D --> I[Implementation]
    I --> T[Test]
    T --> P[Policy Check]
    P --> B[Build AAB]
    B --> C[Console Submit]
    C --> M[Monitor + Maintain]
  
```

21.5 실무 예제 코드

21.5.1 예제 1: 현재 장 핵심 패턴

```

# 기능 변경 없이 비기능 점검 루틴
flutter pub get
flutter analyze
flutter test -j 1
flutter build appbundle --release
  
```

21.5.2 예제 2: 나쁜 예와 좋은 예를 비교하는 공통 템플릿

```

// [나쁜 예] 나쁜 예: 하드코딩 + 중복 + 목적 불분명 함수명
Color getColor(bool selected) {
    if (selected) {
  
```

```

    return const Color(0xFF123456);
}

return const Color(0xFFFFFFFF);
}

// [좋은 예] 좋은 예: 토큰화 + 함수 목적 주석 + 단일 책임
class UiToken {
  UiToken._();
  static const Color selectedChip = Color(0xFF0F766E);
  static const Color unselectedChip = Color(0xFFEEF2FF);
}

/// 칩 선택 상태에 맞는 색상을 반환한다.
Color resolveChipColor({required bool selected}) {
  return selected ? UiToken.selectedChip : UiToken.unselectedChip;
}

```

21.6 나쁜 예 vs 좋은 예

항목	나쁜 예(안티패턴)	좋은 예(개선 패턴)
상수 관리	숫자/색상 직접 입력	토큰/상수 클래스로 분리
상태관리	위젯 곳곳에서 임시 상태 남발	Provider/Notifier로 책임 분리
라우팅	문자열 하드코딩 중복	라우트 규칙 중앙화
예외처리	try-catch 누락	실패 시 fallback + 로깅
테스트	수동 확인만 수행	analyze/test 자동화
로그	print 남발	카테고리/레벨 기반 로그
재사용성	화면마다 비슷한 위젯 복붙	공용 컴포넌트 추출
정책 준수	동의 전 광고 요청	동의 상태 검증 후 요청

21.7 디버깅 포인트

1. 이슈를 “재현 가능 단계”로 먼저 분해한다.
2. 로그를 카테고리별로 보고 입력/상태/출력을 분리한다.
3. UI 문제인지, 상태 문제인지, 데이터 문제인지 먼저 판단한다.
4. 에러 메시지의 첫 줄보다 스택트레이스 상단 함수명을 우선 확인한다.
5. 재현 조건(기기, 날짜, 동의 상태, 로그인 상태)을 기록한다.
6. 동일 증상을 최소 샘플 코드로 축소해 본다.
7. 변경 전/후 스냅샷을 남겨 회귀를 추적한다.
8. 테스트로 재현 가능한지 먼저 본다.
9. 고친 뒤에는 반대 경로(negative path)도 검증한다.
10. “원인-해결-재검증” 3단계로 문서화한다.

21.8 성능/보안/유지보수 체크포인트

- 불필요한 rebuild를 유발하는 watch 범위를 줄였는가?
- 하드코딩된 비밀값/키가 코드에 있는가?
- 예외 발생 시 사용자에게 의미 있는 피드백이 있는가?

- 상태 객체가 불변(immutable) 원칙을 따르는가?
- 중복 조건문/필터 로직을 공용 함수로 추출했는가?
- 네트워크/스토리지 실패 시 앱이 죽지 않는가?
- 릴리즈 경로에서 디버그 코드가 실행되지 않는가?
- 파일/함수 이름이 역할을 명확히 표현하는가?
- 테스트 코드가 핵심 비즈니스 규칙을 덮고 있는가?
- 변경된 파일 경로를 문서화했는가?

21.9 장 마무리 요약

- 이 장의 핵심은 “동작하는 코드”가 아니라 “유지되는 코드”입니다.
- 구조를 나누는 이유는 협업보다 먼저, 미래의 나 자신을 위한 비용 절감입니다.
- 한 번에 크게 고치지 말고, 작은 단위로 안전하게 수정하고 검증해야 합니다.
- 항상 분석/테스트/실행 3종 세트를 반복하세요.
- 실무 품질은 화려한 기능보다 일관된 기본기에서 나옵니다.

21.10 퀴즈 5문항

1. 하드코딩이 단기적으로 빠른데도 장기적으로 위험한 이유는?
2. 상태관리 계층을 나누지 않으면 어떤 버그가 쉽게 생기는가?
3. 예외 처리에서 “fallback”이 필요한 이유는?
4. 테스트가 없는 리팩터링이 위험한 이유는?
5. 이 장의 내용을 현재 프로젝트 파일에 적용할 때 첫 단계는?

21.10.1 정답 및 해설

1. 변경 지점이 분산되어 수정 누락/회귀 확률이 급증합니다.
2. UI 간접 타이밍이 꼬여 재현 어려운 간헐 버그가 발생합니다.
3. 실패 시 앱 전체 기능 정지 대신, 제한된 기능으로 운영을 계속하기 위해서입니다.
4. 동작 보장이 없어 “개선”이 아닌 “변형”이 될 수 있기 때문입니다.
5. 변경 범위를 문서로 분해하고 영향 파일을 식별하는 것입니다.

21.11 실습 과제

- 난이도: 응용
- 과제명: 실전 프로젝트 적용: OurMatchWellFlutter 개선 로드맵 실습
- 요구사항:
 - 대상 파일(docs/PLAY_RELEASE.md)의 하드코딩 3개 이상 식별
 - 상수 또는 토큰으로 치환
 - 실패 시 fallback 동작 1개 추가
 - 로그 1개 이상 추가
- 완료 조건:
 - 기능 동일 유지
 - flutter analyze 통과
 - flutter test -j 1 통과
- 검증법:

```
flutter pub get
flutter analyze
```

```
flutter test -j 1
flutter run
```

21.12 실습 정답 코드 또는 해설

```
/// 실습 해설: 목적이 분명한 함수로 로직을 분리한다.
String buildDebugLabel({required String module, required String action}) {
  return '\\';
}

/// 실습 해설: 하드코딩 값을 상수로 분리한다.
class PracticeToken {
  PracticeToken._();
  static const int retryLimit = 3;
  static const Duration retryDelay = Duration(milliseconds: 300);
}
```

해설 포인트:

- 함수는 “무엇을 반환하는지”가 이름에서 드러나야 합니다.
- 숫자 상수는 의미를 부여한 이름으로 바꿔야 합니다.
- 실패 시나리오를 먼저 작성하면 안정성이 올라갑니다.
- 로그는 나중에 지우는 게 아니라, 분류 체계를 정해 유지해야 합니다.
- 테스트 가능한 함수 형태(순수 함수)에 가까울수록 유지보수가 쉽습니다.

21.13 실무에서 바로 쓰는 체크리스트

- 이 장에서 배운 개념을 한 문장으로 설명 가능
- 대상 파일의 책임을 2개 이상 나누지 않음(SRP)
- 하드코딩/매직넘버 제거
- 예외 처리와 fallback 존재
- 로그 카테고리 정의
- 테스트 가능한 구조로 분리
- PR 전 analyze/test 수행
- 사용자 노출 문구 QA 완료
- 정책 위반 가능성 점검
- 변경 내용 문서화

22 실습 종합 로드맵 (12개 이상 요구 충족)

번호	실습명	난이도	핵심 역량
1	앱 진입점 의존성 주입 이해	초급	bootstrap + ProviderScope
2	null-safety 모델 파싱	초급	타입 안정성
3	위젯 분해 연습	초급	재사용/가독성
4	레이아웃 안정화	초급	반응형 배치
5	Scaffold 구조 재조립	초급	화면 골격
6	setState 제한적 사용	초급	로컬 상태
7	Riverpod provider 연결	중급	상태관리
8	go_router redirect 테스트	중급	라우팅
9	테마 토큰 정리	중급	디자인 시스템
10	JSON 실패 fallback	중급	에러 처리
11	저장소 상태 복원	응용	persistence
12	레이어 분리 리팩터링	응용	아키텍처
13	UMP 동의 후 광고 요청 검증	응용	정책 준수
14	위젯 테스트 보강	응용	품질 게이트
15	로그 기반 디버깅 리포트 작성	응용	운영 안정성
16	릴리즈 빌드 사전 점검	응용	배포 안정성
17	CI 실패 원인 추적	응용	자동화 운영
18	Rebuild 최적화 측정	응용	성능
19	리팩터링 체크리스트 적용	응용	유지보수
20	Preflight 점검 자동화	응용	출시 품질

23 용어집 (Glossary, 90개)

용어	설명
Flutter	구글의 크로스플랫폼 UI 프레임워크
Dart	Flutter에서 사용하는 프로그래밍 언어
Widget	Flutter UI를 구성하는 최소 단위
StatelessWidget	상태가 없는 위젯

용어	설명
StatefulWidget	상태를 가지는 위젯
BuildContext	위젯 트리 상의 위치/정보 컨텍스트
Scaffold	기본 화면 골격 위젯
AppBar	상단 앱 바
SafeArea	노치/시스템 영역 회피
Row	가로 배치 레이아웃
Column	세로 배치 레이아웃
Stack	겹치기 레이아웃
Expanded	남는 공간 확장
Flexible	유연한 공간 분배
Padding	내부 여백
Margin	외부 여백(보통 Container로 표현)
ThemeData	앱 공통 테마 정의 객체
ColorScheme	색상 체계 톤
TextTheme	텍스트 스타일 체계
Material3	최신 머티리얼 디자인 스펙
ProviderScope	Riverpod 루트 스코프
Provider	읽기 전용 의존성 제공자
StateProvider	단순 상태 제공자
NotifierProvider	상태 + 로직 제공자
Ref	Provider 읽기/구독 인터페이스
watch	상태 구독
read	단발성 읽기
invalidate	캐시 무효화
go_router	선언형 라우팅 패키지
GoRouter	라우터 인스턴스
GoRoute	단일 라우트 정의
redirect	조건부 경로 전환
deep link	URL로 특정 화면 진입
async/await	비동기 제어 문법
Future	비동기 결과 탐색

용어	설명
Stream	비동기 이벤트 연속 타입
jsonDecode	JSON 문자열 파싱
exception	예외 상황 객체
stack trace	호출 스택 정보
fallback	실패 시 대체 동작
shared_preferences	키-값 로컬 저장소
cache	재사용 데이터 임시 저장
repository	데이터 접근 추상 계층
usecase	도메인 동작 유스케이스
model	데이터 구조 클래스
immutable	불변 객체 설계
SRP	단일 책임 원칙
OCP	개방-폐쇄 원칙
LSP	리스코프 치환 원칙
ISP	인터페이스 분리 원칙
DIP	의존성 역전 원칙
DI	의존성 주입
AdMob	구글 모바일 광고 플랫폼
Banner Ad	배너 광고 포맷
UMP	User Messaging Platform(동의 SDK)
consent	개인정보 동의 상태
Privacy Options	동의 설정 화면
AD_ID	Android 광고 ID 권한
AndroidManifest	Android 앱 메타데이터/권한 파일
build.gradle.kts	Android 빌드 설정 파일
applicationId	Android 패키지 식별자
versionCode	Android 내부 버전 코드
versionName	사용자 표시 버전
AAB	Android App Bundle
keystore	릴리즈 서명 키 저장소
key.properties	서명 정보 로컬 파일

용어	설명
CI	지속적 통합
CD	지속적 배포
GitHub Actions	CI/CD 자동화 플랫폼
workflow	자동화 파이프라인 정의
artifact	빌드 산출물
internal track	Play 내부 테스트 트랙
analyze	정적 분석 명령
flutter test	테스트 실행 명령
smoke test	핵심 동선 점검 테스트
widget test	UI 위젯 단위 테스트
unit test	로직 단위 테스트
integration test	통합 시나리오 테스트
lint	코드 규칙 검사
refactor	동작 유지 구조 개선
regression	기존 기능 퇴행 버그
technical debt	기술 부채
code review	변경 코드 검토
observability	관측 가능성(로그/메트릭)
logging	실행 기록
retry	실패 후 재시도
cooldown	재실행 간격 제한
policy compliance	정책 준수
Play Console	구글 플레이 배포 콘솔
Data safety	데이터 안전성 고지
privacy policy	개인정보처리방침
release note	버전 변경 안내

24 부록 A. 프로젝트 맞춤 운영 규칙

1. UI 수정은 우선 `lib/features/*`와 `lib/ui_components/*`에서 시작한다.
2. 앱 전체 스타일은 `lib/app/theme.dart`에서만 조정한다.
3. 광고/동의 로직은 `lib/core/ads/*` 외부에서 직접 호출하지 않는다.

4. 저장소 키 변경 시 `preferences_store.dart` 마이그레이션 계획이 필요하다.
5. 라우트 변경 시 `router_smoke_test.dart` 계열 테스트를 함께 수정한다.
6. Android 릴리즈 설정 변경 시 `build.gradle.kts`와 `AndroidManifest.xml`을 같이 검토한다.
7. 배포 전 반드시 `flutter analyze`, `flutter test -j 1`, `flutter build appbundle --release`을 수행한다.

25 부록 B. 7일 학습 운영표 (예시)

Day	학습 범위	실습	완료 기준
1일차	Chapter 1~3	실습 1~3	위젯 분해 가능
2일차	Chapter 4~5	실습 4~5	레이아웃 안정화
3일차	Chapter 6~7	실습 6~7	상태관리 이해
4일차	Chapter 8~10	실습 8~10	라우팅/비동기 처리
5일차	Chapter 11~13	실습 11~13	저장소/광고 정책
6일차	Chapter 14~17	실습 14~17	테스트/배포 자동화
7일차	Chapter 18~20	실습 18~20	성능/유지보수/프리플라이트

26 부록 C. 초보가 자주 하는 실수 50선과 즉시 해결법

번호	실수	원인	즉시 해결법
1	setState 남발	상태 경계 미정의	로컬 상태 vs 전역 상태 분리
2	문자열 라우트 오타	상수화 미흡	라우트 상수/중앙 선언
3	try-catch 누락	낙관적 구현	실패 경로를 먼저 작성
4	하드코딩 색상 난립	토큰 부재	ThemeData 토큰화
5	빌드 전 테스트 생략	시간 압박	최소 스모크 자동화
6	로그 없이 디버깅	관측성 부재	카테고리 로그 추가
7	테스트 광고 배포	환경 분리 실패	dart-define 확인
8	권한 누락	Manifest 미검토	배포 전 권한 체크리스트
9	리팩터링 후 검증 누락	자신감 과잉	analyze/test/run 필수
10	종복 로직 복불	구조 미설계	유틸/리포지토리 추출
11	JSON null 미처리	타입 가정	기본값/nullable 처리
12	비동기 race condition	await 누락	흐름도 작성 후 정렬
13	provider 순환 참조	설계 누락	레이어 방향 고정
14	UI와 도메인 결합	편의 구현	usecase 계층 도입
15	테스트 flaky	타이밍 의존	pumpUntil/고정 시나리오

번호	실수	원인	즉시 해결법
16	import 혼선	파일 구조 난잡	feature 단위 정리
17	긴 함수	책임 과다	목적별 함수 분리
18	네이밍 불명확	역할 미정	동사+명사 규칙
19	스펙 미확정 개발	요구사항 부족	체크리스트 선작성
20	디버그 코드 유입	release 가드 부재	kDebugMode 가드
21	빈 화면 처리 누락	정상 케이스 편향	loading/empty/error 분리
22	국제화 미고려	문자열 분산	문구 관리 규칙
23	과한 rebuild	watch 범위 과대	Consumer 분할
24	대형 위젯 파일	모듈화 부족	컴포넌트 추출
25	cache 오염	무효화 누락	invalidate 정책
26	구버전 dependency	업데이트 미관리	pub outdated 점검
27	Android/iOS 차이 무시	플랫폼 인식 부족	platform detector 사용
28	release signing 실패	key.properties 누락	로컬/CI 템플릿 준비
29	ci 실패 원인 미기록	운영 절차 부재	실패 로그 템플릿
30	콘솔 입력값 혼동	문서 부재	docs/ 릴리즈 체크 표
31	동의 상태 오해	정책 이해 부족	Required/Obtained 구분
32	광고 요청 타이밍 오류	부트스트랩 순서 오류	consent 이후 init
33	JSON 스키마 변경 대응 실패	계약 문서 없음	모델 버전 관리
34	선택/비선택 색 대비 부족	접근성 미검토	대비 비율 체크
35	컴포넌트 props 과다	추상화 과잉	실제 사용 기준 축소
36	상태 초기화 누락	라이프사이클 이해 부족	init/dispose 점검
37	컨트롤러 dispose 누락	메모리 관리 미흡	dispose 규칙화
38	SnackBar 남용	UX 흐름 미고려	중요 이벤트만 노출
39	오류 메시지 노출 과다	내부 정보 노출	사용자 친화 메시지
40	풀더 규칙 없음	팀 컨벤션 부재	README 구조 명시
41	route push/go 혼용 오해	네비 API 이해 부족	사용 기준 문서화
42	versionCode 충돌	배포 전략 부재	run number 정책
43	스크린샷 관리 난잡	문서 자산 규칙 없음	docs/screenshots 규칙
44	assets 미사용 파일 누적	정리 프로세스 부재	주기적 정리 스크립트
45	로컬 빌드만 신뢰	환경 편차	CI 결과 우선
46	테스트 데이터 혼재	fixture 분리 미흡	test fixture 구성

번호	실수	원인	즉시 해결법
47	코드리뷰 생략	일정 압박	최소 체크리스트 리뷰
48	핫픽스 후 문서 미갱신	운영 습관 부재	수정 후 docs 즉시 반영
49	룰백 경로 미준비	릴리즈 계획 부족	백업/태그 전략
50	학습 중단	목표 불명확	7일 루틴 재시작

27 마무리

이 교재의 목적은 “정답 코드 복사”가 아니라, 당신이 다음 수정을 혼자 안전하게 수행하는 능력을 만드는 것입니다.

학습의 기준은 아래 한 문장으로 요약됩니다.

기능을 추가하기 전에 구조를 이해하고, 구조를 바꾸기 전에 테스트를 먼저 준비하라.

28 부록 D. 출시 전/후 실전 운영 시나리오 30선

이 부록은 “실제로 유지보수할 때 무엇부터 확인해야 하는가”를 빠르게 찾기 위한 운영 핸드북입니다. 각 시나리오는 **현상** → **원인 후보** → **즉시 조치** → **재검증** 순서로 작성했습니다.

28.1 D-1. 빌드/배포 시나리오 10선

시나리오	현상	원인 후보	즉시 조치	재검증
1	AAB 빌드 실패	key.properties 누락	android/key.properties 생성	flutter build appbundle -release
2	릴리즈에서 광고 미노출	배너 유닛 ID 누락	dart-define/gradle 확인	로그 + 배너 로드 확인
3	테스트 광고가 릴리즈에 노출	ADMOB_USE_TEST_ADS true	릴리즈 플래그 false 고정	내부테스트 재빌드
4	Play 업로드 실패	서비스 계정 권한 부족	Play Console 권한 재부여	release workflow 재실행
5	versionCode 총돌	이전 값 재사용	run_number 기반 증가 정책 적용	재빌드 후 업로드
6	Manifest placeholder 오류	ADMOB_APP_ID 미설정	gradle.properties 값 설정	assembleRelease
7	CI 통과/로컬 실패	로컬 캐시 오염	flutter clean + pub get	analyze/test 재실행
8	앱 실행 즉시 크래시	초기화 순서 오류	bootstrap, Provider override 재점검	flutter run
9	라우팅 꼬임	redirect 조건 충돌	onboarding 분기 조건 재정의	router smoke test
10	아이콘 반영 실패	launcher 아이콘 생성 미실행	flutter_launcher_icons 재실행	기기 재설치 확인

28.2 D-2. 광고/정책 시나리오 10선

시나리오	현상	원인 후보	즉시 조치	재검증
11	동의 버튼 무반응	UMP 채널 예외	lastError 로깅 확인	Settings에서 재시도
12	동의 전 광고 요청	가드 누락	canRequestAds 체크 위치 상향	로그 검증
13	Privacy Options 불가	UMP 설정 미완료	AdMob 콘솔 품 설정	디버그 기기 재검증
14	Unknown 상태 지속	동의 gather 미호출	앱 시작 gather 호출 보장	앱 재실행
15	배너 크기 null	adaptive size 실패	width/context 타이밍 점검	회전/재진입 테스트
16	애뮬레이터 광고 미표시	Play 인증/네트워크 이슈	실기기 테스트 병행	test device 등록

시나리오	현상	원인 후보	즉시 조치	재검증
17	권한 관련 경고	AD_ID 누락	Manifest 권한 추가 확인	lint + 런타임 점검
18	사용자 불만 증가	과도한 광고 노출	배너 위치/빈도 재검토	UX 실측
19	정책 심사 거절	고지 문구 미흡	개인정보/광고 고지 업데이트	내부 QA
20	데이터 안전 혼선	콘솔 답변 불명확	수집/공유 항목 문서화	콘솔 점검

28.3 D-3. 상태/데이터 시나리오 10선

시나리오	현상	원인 후보	즉시 조치	재검증
21	즐겨찾기 사라짐	저장 키 변경/초기화	키 상수 복구, マイグ레이션	재로그인 없이 확인
22	복습 큐 비정상	due 계산 로직 오류	submitReviewResult 분기 점검	단위 테스트
23	설치일 계산 오류	installDate 파싱 실패	기본값 처리 보강	dayIndex 테스트
24	검색 결과 0개	태그/쿼리 조건 충돌	필터 순서 확인	explore 수동 테스트
25	리스트 깜빡임	과도한 invalidate	provider 구독 범위 축소	성능 측정
26	JSON 파싱 실패	schema mismatch	fromJson fallback 보강	fixture 테스트
27	캐시 불일치	refresh 누락	cache delete + 메모리 초기화	새로고침 검증
28	언어 혼합 문구	문구 소스 분산	문자열 위치 정리	QA 스크립트
29	리뷰 간격 과도	interval 계산값 하드코딩	상수화 및 클램프	도메인 테스트
30	특정 화면만 느림	rebuild 범위 과대	Consumer 분할/const 최적화	profiler 확인

28.4 D-4. 상황별 대응 플레이북 (상세)

28.4.1 플레이북 1: “앱은 켜는데 흠이 비어 있음”

- `contentSnapshotProvider` 상태를 loading/error/data로 분기해 로그를 찍습니다.
- `assets/data/*.json` 가 pubspec assets에 등록되었는지 확인합니다.
- `ContentStore._loadSentences` 에서 minCount 조건 충족 여부를 봅니다.
- 캐시 파일이 깨졌을 가능성이 있으면 refresh 루틴으로 캐시를 비웁니다.
- 여러 문구가 사용자 친화적인지(내부 stack 노출 금지) 점검합니다.

재검증:

```
flutter pub get
flutter analyze
```

```
flutter test -j 1
flutter run
```

28.4.2 플레이북 2: “탭 전환 시 이전 상태가 이상함”

- IndexedStack 기반인지 확인합니다.
- 상태가 화면 로컬인지 Provider 전역인지 경계를 점검합니다.
- dispose 타이밍과 controller 재생성 여부를 확인합니다.
- 검색창/세그먼트 선택값이 탭별로 분리되어야 하는지 요구사항을 재검토합니다.

28.4.3 플레이북 3: “설정 화면에서 동의 상태가 Unknown으로 고정”

- UMP 콘솔 설정(App ID, privacy form) 완료 여부 확인
- 네트워크/디버그 기기 해시 등록 여부 확인
- `gatherConsent(force: true)` 재호출 시 상태 전이 여부 관찰
- lastError를 사용자용 메시지와 내부 로그로 분리

28.4.4 플레이북 4: “스토어 심사 직전 최종 점검 순서”

1. 앱 이름/아이콘/패키지명 확인
2. 개인정보처리방침 링크 열림 여부 확인
3. 광고 선언/데이터 안전 답변 최종 검수
4. 릴리즈 노트(ko-KR) 최신 반영
5. 내부 테스트 트랙 설치 검증
6. 비정상 종료 없는지 최소 10분 스모크 테스트

28.4.5 플레이북 5: “핫픽스가 필요한 치명적 버그 발생”

- 기준 브랜치에서 hotfix 브랜치를 분기
- 재현 테스트를 먼저 작성(또는 재현 절차 문서화)
- 최소 수정만 적용
- analyze/test 통과 후 버전 증가
- 배포 후 모니터링 로그 체크

28.5 D-5. 유지보수 체크시트 (인쇄용)

28.5.1 주간 체크시트

- 신규 경고/오류(analysis) 0건 유지
- flaky 테스트 여부 확인
- 정책/문구 변경점 반영
- 불필요 assets/code 정리
- release 문서 최신화
- crash 재현 보고서 업데이트

28.5.2 월간 체크시트

- dependency 업데이트 필요성 검토
- 성능 회귀 측정(탭 전환, 초기 진입, 리스트 스크롤)
- UX 개선 후보 5개 작성

- 기술부채 목록 우선순위 재정렬
- Play Console 경고/권고사항 점검

28.5.3 분기 체크시트

- 아키텍처 를 위반 코드 정리
 - 테스트 커버리지 핵심 흐름 점검
 - CI/CD 실패 패턴 회고
 - 사고 대응 플레이북 갱신
 - 운영 문서와 실제 코드 동기화
-

28.6 D-6. 1:1 코칭 질문 템플릿

멘토링 또는 자기점검에 아래 질문을 사용하세요.

1. 이 변경은 어떤 사용자 문제를 해결하는가?
2. 이 변경은 어느 레이어에서 책임져야 하는가?
3. 기존 테스트는 무엇이 깨질 수 있는가?
4. 실패 시 fallback이 준비되어 있는가?
5. 로그로 원인 추적이 가능한가?
6. 한 주 뒤 내가 다시 봐도 이해 가능한가?
7. 릴리즈 노트에 어떤 문장으로 남길 것인가?