Ausformulierte Roadmap für Tap'Em

Phase 1: Projekt-Setup und Basisarchitektur

In Phase 1 richten wir die gesamte technische Basis für unsere App ein. Wir beginnen damit, ein sauberes Flutter-Projekt aufzusetzen, die Verbindung zu Firebase herzustellen und eine modulare, feature-orientierte Ordnerstruktur zu etablieren. Gleichzeitig bereiten wir Flavors (Entwicklungs- und Produktionsumgebung) vor, integrieren grundlegende Analyse- und Crash-Reporting-Tools und verifizieren die Funktionalität durch Testzugriffe auf Firestore.

Meilenstein 1.1: Initialisierung des Flutter-Projekts und Firebase-Anbindung Zunächst prüfen wir, ob das Verzeichnis bereits ein gültiges Flutter-Projekt enthält. Wir erwarten Ordner `android/`, `ios/`, `lib/` sowie eine `pubspec.yaml`. Falls etwas fehlt, führen wir `flutter create .` aus, um das Projekt zu initialisieren.

Anschließend erstellen wir unter `lib/` unsere feature-basierte Struktur. Ein Ordner `core/` enthält allgemeine Dienste (z. B. Theme-Loader, Utility-Klassen), `features/` beherbergt pro Funktionalität (Auth, Gym-Onboarding, Gerätelogging, Historie, Admin-Dashboard) je drei Unterschichten (`data/`, `domain/`, `presentation/`). Im Root von `lib/` liegen `main.dart` und `app_router.dart`.

Um später zwischen Entwicklungs- und Produktions-Firebase-Projekten zu wechseln, legen wir zwei Umgebungsdateien `.env.dev` und `.env.prod` an. Diese enthalten Variablennamen wie `APP_NAME` und `FIREBASE_PROJECT_ID`. In unseren Gradle- und Xcode-Einstellungen fügen wir entsprechende Flavor-Konfigurationen ein, sodass wir die App mit `flutter run -flavor dev` oder `flutter run -flavor prod` starten können.

Als nächstes fügen wir in `pubspec.yaml` die Pakete `firebase_core`, `firebase_auth`, `cloud_firestore`, `firebase_analytics` und `firebase_crashlytics` hinzu. Die nativen Konfigurationsdateien `google-services.json` (Android) und `GoogleService-Info.plist` (iOS) werden in die jeweiligen Plattform-Verzeichnisse kopiert. In `main.dart` stellen wir sicher, dass Widgets-Binding initialisiert ist, laden die Umgebungsvariablen mit `dotenv`, initialisieren Firebase und binden Crashlytics/Analytics direkt vor `runApp()` ein.

Zum Abschluss dieses Meilensteins implementieren wir einen einfachen Firestore-Smoke-Test: Die App schreibt in Code ein Dummy-Dokument in eine Test-Collection und liest es sofort wieder aus. Gelingt das, wissen wir, dass die Anbindung korrekt funktioniert.

Meilenstein 1.2: Grundlegende App-Struktur und Navigation Im zweiten Schritt übernehmen wir die erstellte Ordnerstruktur und fügen das globale State-Management hinzu. Wir entscheiden uns für `Provider`, da es für einen Prototyp leichtgewichtig und flexibel ist, aber abstrahieren unsere Geschäftslogik so, dass später eine Migration auf BLoC möglich bleibt.

Dazu definieren wir in `core/providers/` drei ChangeNotifier-Klassen: `AppProvider` (überwacht App-Wechsel, Themes, Ladezustand), `AuthProvider` (verwaltet Anmeldung, Registrierung, Nutzerzustand) und `GymProvider` (speichert aktuell ausgewähltes Gym, lädt Maschinendaten). Diese binden wir in `main.dart` mittels `MultiProvider` ein.

Für die Navigation erstellen wir `app_router.dart` mit benannten Routen. Die `SplashScreen` prüft anhand von `AuthProvider.isLoggedIn`, ob der Nutzer direkt zum `HomeScreen` oder zur `AuthScreen` weitergeleitet wird. Alle weiteren Screens (Login/Registration, Dashboard, Gerätedetail, Historie, Admin-Dashboard) hängen an entsprechenden Routen, die wir im Router konfigurieren.

Als primäre Navigation wählen wir eine BottomNavigationBar mit vier Hauptbereichen: „Dashboard“, „Historie“, „Profil“ und „Admin“. Dies erlaubt schnellen Zugriff auf die Kernfunktionen. Einen Drawer behält das Layout später als Option, falls weitere Menüpunkte hinzukommen.

Zudem implementieren wir den ersten Onboarding-Flow: Auf dem Auth-Onboarding-Screen kann der Nutzer einen Gym-Code eingeben. Wir validieren diesen in Firestore gegen die Collection `gyms` und zeigen bei ungültigem Code direkt unter dem Eingabefeld eine Fehlermeldung an. Nach drei aufeinanderfolgenden Fehlversuchen sperrt die App für 30 Sekunden den Button und informiert per Snackbar. Ein kleiner „Hilfe“-Link verweist auf Kontaktinformationen des Studios.

Meilenstein 1.3: Internationalisierung (DE/EN) Um in Zukunft problemlos weitere Sprachen zu unterstützen, binden wir bereits jetzt das Flutter-Internationalisierungspaket ein. Wir installieren `flutter_localizations` und `intl`, erzeugen unter `lib/l10n/` zwei ARB-Dateien (`app_de.arb`, `app_en.arb`) mit initialen deutschen Texten und englischen Platzhaltern. In `MaterialApp` konfigurieren wir `supportedLocales` und `localizationsDelegates`. Sämtliche in Widgets genutzten Strings ersetzen wir durch Aufrufe von `AppLocalizations.of(context)<key` sodass die App bei Bedarf automatisch zwischen Deutsch und Englisch wechselt.

Meilenstein 1.4: Grundlegendes Datenmodell (Firestore) Zum Abschluss von Phase 1 definieren wir in Firestore die Collections und Subcol-

lections, die unser Kern-Datenmodell bilden:

- `gyms/{gymId}` enthält unter anderem den Onboarding-Code, Studio-Name und optionale Metadaten.
- `users/{userId}` speichert E-Mail, zugehöriges `gymId`, Rolle („member“, „admin“, später „coach“) und einen Timestamp.
- `machines/{machineId}` enthält Maschinenname, `gymId` und optional `nfcTagId`.
- Unter jeder Maschine legen wir eine Sub-Collection `logs` an: Jedes Dokument darin speichert `userId`, `timestamp`, `weight`, `reps` und optional eine `sessionId`, um Trainingssätze zu gruppieren.

In der Code-Basis spiegeln wir dieses Modell durch Dart-Klassen in `features/<feature>/domain/m` und DTOs in `features/<feature>/data/dtos` wider. Repository-Interfaces und Use-Case-Klassen in `domain/usecases` kapseln den Datenzugriff, damit unsere UI-Schicht davon entkoppelt bleibt.

Am Ende von Phase 1 haben wir ein lauffähiges Android-Projekt, das auf iOS kompilierbar ist, mit funktionierendem Firebase-Zugriff, internationalisierten Texten, Flavors für Dev/Prod, Analyse- und Crash-Reporting sowie einer soliden feature-basierten Struktur. Die Grundlage für das Onboarding und alle folgenden Funktionen ist gelegt.

Phase 2: Onboarding und Benutzerkonto

Meilenstein 2.1: Gym-Code Eingabe und Verifizierung Zu Beginn von Phase 2 präsentiert die App nach der Splash-Logik den Onboarding-Screen, auf dem der neue Nutzer einen exakt vom Fitnessstudio bereitgestellten Gym-Code eingibt. Sobald der Nutzer den Code bestätigt, führt die App eine Firestore-Abfrage in der Collection `gyms` durch und sucht nach einem Dokument, dessen Feld `code` dem Input entspricht. Ist kein Dokument vorhanden, wird unmittelbar unter dem Eingabefeld ein roter Inline-Text „Ungültiger Code“ angezeigt und der Nutzer erhält zu jedem Fehlversuch eine kurze Erklärung. Nach drei fehlerhaften Versuchen deaktiviert sich der Bestätigungsbutton für 30 Sekunden, begleitet von einer Snackbar „Bitte warten…“. Zusätzlich bietet ein „Hilfe“-Link Kontaktinformationen für das Studio. Wird der Code hingegen gefunden, speichert `GymProvider` die `gymId` lokal und leitet nahtlos zum Auth-Screen weiter.

Meilenstein 2.2: Nutzerregistrierung und Login Im Auth-Screen implementieren wir eine TabBar mit zwei Tabs: „Login“ und „Registrieren“. Im Registrierungsformular fordert die App E-Mail, Passwort und bestätigt intern per `FirebaseAuth.createUserWithEmailAndPassword()` die Erstellung. Parallel dazu legt sie in Firestore unter `users/{}` ein Dokument an, das neben der `uid` auch `gymId` und defaultmäßig die Rolle „member“ enthält. Bei erfolgreicher Registrierung wird der Nutzer automatisch eingeloggt und in die Home-Navigation geführt.

Im Login-Tab ruft die App `signInWithEmailAndPassword()` auf. Nach erfolgreicher Authentifizierung lädt sie das zugehörige User-Dokument aus Firestore, speichert Rolle und Profil im `AuthProvider` und navigiert in die Hauptansicht. Fehler wie „E-Mail nicht gefunden“, „Falsches Passwort“ oder Netzwerkprobleme fangen wir in `catch`-Blöcken ab und zeigen den Nutzer:innen leicht verständliche Snackbars an. Ein Link „Passwort vergessen?“ öffnet einen Dialog zur Eingabe der E-Mail und löst `sendPasswordResetEmail()` aus.

Meilenstein 2.3: Post-Login Navigation und Rollenstruktur Sobald die Anmeldung abgeschlossen ist, landet der Nutzer auf dem `HomeScreen`. Dort wird über eine BottomNavigationBar der Zugang zu Dashboard (Geräteübersicht/Scan), Historie, Profil und Admin-Dashboard ermöglicht. Im `AuthProvider` halten wir das Feld `role`; nur wenn dieses „admin“ ist, erkennt die App bei Klick auf den Admin-Tab, dass Schreibfunktionen freigeschaltet werden dürfen, andernfalls bleibt die Ansicht nur lesend. So bereiten wir die spätere Feinsteuerung der Berechtigungen vor, während der Prototyp bereits alle Bereiche testbar bietet.

Phase 3: Geräte-Integration und Trainingserfassung

Meilenstein 3.1: Geräte-Stammdaten und NFC-Vorbereitung Im Hintergrund lädt `GymProvider` beim Start alle Maschinen des Studios aus `machines` und cacht sie. Parallel fügen wir das Plugin `nfc_manager` hinzu, das Android ab API 19 und iOS ab Version 13 unterstützt. In `AndroidManifest.xml` deklarieren wir die NFC-Permission, in der iOS-`Info.plist` ergänzen wir `NFCReaderUsageDescription` und aktivieren das NFC-Entitlement. So ist die Infrastruktur für das spätere Scannen gelegt.

Meilenstein 3.2: Geräte-Detailseite und Satz-Eingabe Wenn eine Maschine ausgewählt wird (manuell oder per NFC), öffnet die App den `DeviceScreen`. Oben werden Name und optional ein Bild angezeigt, dar-

unter eine Liste bereits erfasster Sätze aus Firestore. Zwei Eingabefelder erlauben die Erfassung von Gewicht und Wiederholungen. Ein Button „Satz hinzufügen“ speichert den Eintrag sofort in der Sub-Collection `logs` der Maschine und fügt ihn in die lokale Liste ein. Diese Rückmeldung sorgt für eine flüssige Nutzererfahrung.

Meilenstein 3.3: NFC-Scan Flow Der `nfc_service` startet per `startSession()`, liest das Tag aus und extrahiert eine ID. Anschließend fragt `GymProvider.findByTagId()` die lokale Cache-Liste (bzw. Firestore) nach der zugehörigen Maschine. Wird eine Übereinstimmung gefunden, navigiert die App automatisch zum `DeviceScreen`. Andernfalls zeigt eine Snackbar „Unbekanntes NFC-Tag – bitte manuell auswählen“ an.

Meilenstein 3.4: Manuelle Geräteauswahl Für den Fall, dass kein NFC-Tag genutzt wird oder ein unbekanntes Tag gescannt wird, bietet der `HomeScreen` einen Button „Gerät manuell auswählen“. Dieser öffnet ein Modal mit einer Liste aller Maschinen im Studio. Ein Tap auf eine Zeile navigiert ebenfalls zum `DeviceScreen`, sodass jeder Flow zum selben Ziel führt.

Phase 4: Trainingshistorie und Verlauf

Meilenstein 4.1: Datenabruf der Workout-Historie In `HistoryProvider` implementieren wir eine Firestore-Abfrage auf die Sub-Collections `machines/{mid}/logs`, gefiltert nach der aktuellen `userId` und sortiert nach Zeitstempel. Bei Bedarf legen wir in Firestore einen Composite-Index an, damit die Abfrage performant bleibt.

Meilenstein 4.2: UI-Implementierung der History-Ansicht Der `HistoryScreen` gruppiert Einträge sektional nach Datum. Jeder Eintrag zeigt „Gewicht × Wdh. @ Uhrzeit“. Mithilfe eines `StreamBuilder` binden wir die Logs-Streams ein, sodass die Ansicht automatisch aktualisiert wird, sobald ein neuer Satz gespeichert wurde.

Phase 5: Admin-Auswertung

Meilenstein 5.1: Admin-Zugriff vorbereiten Auf Basis von `AuthProvider.role` aktivieren wir im BottomNavigation-Bar den Tab „Admin“. Mitglieder sehen ihn auch, können im Prototyp aber nur lesend darauf zugreifen – schreibende Funktionen sind ausgegraut.

Meilenstein 5.2: Geräte-Nutzungsübersicht Im `AdminDashboardScreen` aggregieren wir alle Logs der Maschinen des aktuellen Gyms mittels `collectionGroup('logs')` und Zählung nach `machineId`. Die so ermittelten Nutzungszahlen füllen wir in eine Tabelle oder einfache Balkendiagramme, damit Studio-Betreiber auf einen Blick die Auslastung erkennen.

Phase 6: Feinschliff, Tests und Ausblick

Meilenstein 6.1: Offline-Unterstützung Wir aktivieren Firestore-Offline-Persistenz mit `settings(persistenceEnabled:true)`. Die geladenen Maschinen bleiben im Cache, und neue Workoutsätze werden lokal gespeichert, falls das Gerät offline ist. Bei Netzwerkverlust informiert eine Snackbar die Nutzer:innen, dass Daten synchronisiert werden, sobald wieder Verbindung besteht.

Meilenstein 6.2: Code-Refactoring und Performance Im gesamten Projekt entfernen wir Redundanzen, verlagern Logik aus Widgets in Use-Case-Klassen und optimieren Firestore-Abfragen (z. B. Lazy-Loading, Caching). Einheitliche Fehlerbehandlung erfolgt über `error_mapper.dart`.

Meilenstein 6.3: Testing und Bugfixing Wir testen manuell alle Flows (Onboarding, Auth, NFC, Logging, History, Admin) auf Emulatoren und realen Geräten verschiedener Android- und iOS-Versionen. Edge-Cases wie ungültige Eingaben (0 kg, 0 Wdh), mehrfaches Scannen oder wechselnde Verbindungen prüfen wir gezielt. Gefundene Bugs werden priorisiert behoben. Optional schreiben wir erste Widget- und Integrationstests mit `flutter_test`.

Meilenstein 6.4: Cross-Platform Verifikation (iOS) Wir erstellen und testen den iOS-Build auf Simulator und echtem Gerät (iOS 13+). Hier prüfen wir NFC-Scan, SafeArea-Handling und Tastatur-Verhalten. Kleinere Anpassungen stellen sicher, dass die App auf iOS genauso flüssig läuft wie auf Android.

Meilenstein 6.5: Dokumentation und Ausblick Zum Abschluss fertigen wir eine Projektdokumentation an, die Setup-Anleitungen für Dev/Prod, Demo-Accounts und bekannte Einschränkungen enthält. Als Ausblick notieren wir Erweiterungen wie feinere Rollenrechte, Coach-Feature, detaillierte Zeitfilter in der Auswertung und vollständige Offline-Registrierung.

Diese Roadmap dient als lebendiges Dokument. Sie wird im Verlauf der Entwicklung fortlaufend angepasst, sobald neue Erkenntnisse oder Anforderungen auftreten.