FINAL MODULE ASSESSMENT REPORT

Muscle Power

Professor:

Dr : Mfonobong Uko

Student:

Shahzad Sadruddin

Student ID:

2513806

Category:

Health & Fitness

Date:

February 21, 2026

Table of Contents

Part A – Concept & Market Framing

App Idea & Category

Muscle Power is a cross-platform mobile application in the *Health & Fitness* category that provides bodybuilding enthusiasts with structured workout programs, an exercise library with form guidance, progress tracking (weight, body-fat, measurements), nutrition logging with macro breakdowns, and a performance dashboard—all wrapped in a visually engaging dark-themed interface. The application adopts a *local-first* architecture: every piece of user data persisted on-device before any optional network synchronization is attempted, ensuring the product remains fully functional offline (Kleppmann, 2019).

Target Market & Personas

Three primary personas drive prioritization:

- Alex (22, university student, beginner). Needs guided workouts, form tips, and a simple calorie counter. Uses the app at a busy campus gym on an affordable Android handset with intermittent Wi-Fi.
- Priya (34, working professional, intermediate). Requires progress charts, custom workout creation, and meal planning around a tight schedule. Values quick data entry and cross-device sync.
- Marcus (48, experienced lifter, visually impaired). Relies on screen-reader support and high-contrast UI. Needs clear semantic labels, scalable fonts, and fully navigable interfaces without relying on color alone.

Accessibility considerations: The app employs semantic widgets throughout, scalable text via MediaQuery.textScaleFactor, sufficient contrast ratios in its orange-on-dark theme, and tooltip labels on every navigation item—meeting WCAG 2.1 Level AA guidance (W3C, 2018).

Value Proposition & MVP Features

Unlike generic fitness trackers, Muscle Power combines a curated bodybuilding-specific exercise library (30+ exercises with form tips), structured workout programs grouped by muscle group and difficulty, real-time progress analytics (BMI, body-fat, measurements over time), and a full nutrition logger—all available offline from the first launch. The MVP ships six core features: (1) exercise library, (2) workout programs, (3) workout logging, (4) progress tracking, (5) nutrition tracking, and (6) user authentication with social login (Google, Apple, Facebook). See **Exhibit I** for more information.

Part B – Mobile UX Design & Features

Navigation Model

The application adopts a hybrid tab + stack navigation model. A six-tab bottom navigation bar (Home, Workouts, Exercises, Progress, Nutrition, Profile) provides primary navigation, with each tab maintaining its own stack for drill-down screens (e.g., Exercises → Exercise Detail). Secondary flows—Landing, Auth, Performance Dashboard, and Feedback—use named-route stack navigation. An IndexedStack keeps all tab states alive, eliminating unnecessary rebuilds when switching tabs (Flutter Team, 2024). **Figure 2** illustrates the complete navigation graph.

Key Screens & Interactions

- Landing Screen: animated brand entrance with Sign In, Create Account, and Guest Access options.
- Home Dashboard: personalized greeting, today's progress ring (circular percent indicator), quick-stats grid (workouts, minutes, calories), featured workout carousel, and quick-action buttons.
- Workouts Screen: workout cards filterable by muscle group, each navigating to a detail view listing exercises with sets, reps, and rest periods.
- Exercises Screen: searchable exercise library with difficulty badges and muscle-group filters.
- Progress Screen: weight and measurement charts powered by fl_chart, BMI calculation, and a body-measurement entry form.
- Nutrition Screen: daily macro summary (protein, carbs, fat), meal logger with type classification, and a calendar view (table_calendar).
- Profile Screen: account settings, privacy center (consent toggles, data export, account deletion), and experience-level configuration.

State Management

State is managed through singleton service classes (AuthenticationService, WorkoutService, ExerciseService, ProgressService, NutritionService, SettingsService) that persist data to SharedPreferences (cross-platform) and SQLite (mobile/desktop). Each service exposes getters for current state and mutation methods returning Futures, with widgets calling and awaiting service operations. This lightweight approach avoids unnecessary library overhead while maintaining a clear separation between UI and business logic (Windmill, 2023).

Non-Functional UX Qualities

- Performance: entrance animations use flutter_animate with 300 ms durations; the Performance Overlay records cold-start time and frame metrics.
- Responsiveness: applies breakpoints at 600 px (phone), 600–900 px (tablet), and 900 px+ (desktop), scaling grid columns, font sizes, and padding accordingly.

- Offline behaviour: a ConnectivityService listens to network changes via a broadcast stream and displays an animated slide-in banner when connectivity drops; all write operations continue against local storage and are queued for later sync.
- Accessibility: every navigation item carries a Semantics widget, the offline banner uses liveRegion, and the theme ensures a minimum 4.5:1 contrast ratio.

See **Exhibit II** for more information about the Mobile UX Design & Features.

Part C – Back-End Architecture & Connectivity

Architectural Approach

Muscle Power follows a local-first, sync-optional architecture. All data persisted on-device (SharedPreferences + SQLite) *before* any network call is attempted. The SyncService singleton acts as an optional synchronization channel: when a backend URL is configured, it transmits data over HTTPS; otherwise, the app operates entirely offline. This approach eliminates a hard dependency on hosted services while allowing seamless migration to a managed backend (e.g., a cloud-hosted REST API) by setting a single configuration property (Kleppmann, 2019). **Figure 1** shows the system block diagram.

API Design

The API layer is RESTful JSON over HTTPS (TLS 1.2+). Every request and response follows a standard envelope pattern: See **Figure 3** for Data/Sequence Flow Diagram

Method & Endpoint	Purpose	Request Body	Response
POST /auth/register	Create account	{ email, passwordHash, salt }	{ status, data: { userId, token } }
POST /auth/login	Sign in	{ email, passwordHash }	{ status, data: { accessToken, refreshToken } }
POST /auth/refresh	Refresh token	{ refreshToken }	{ status, data: { accessToken } }
GET /users/:id	Fetch profile	—	{ status, data: UserProfile }
PUT /users/:id	Update profile	Partial fields	{ status, data: UserProfile }
DELETE /users/:id	GDPR erasure	—	{ status: "ok" }
GET /workouts	List workouts	—	Paginated list
POST /workouts/log	Log session	WorkoutLog JSON	Created log
POST /progress	Log progress	ProgressEntry JSON	Created entry

GET /progress?range=30d	Progress history	—	Entries array
POST /nutrition/meals	Log meal	MealLog JSON	Created meal
GET /nutrition/today	Today's summary	—	Macro totals
POST /feedback	Submit feedback	Feedback JSON	Acknowledged
GET /users/:id/export	GDPR export	—	Portable JSON bundle

Error Handling

Errors follow a structured envelope:

```
Json
{
  "status": "error",
  "code": 400,
  "message": "Validation failed",
  "details": {}
}
```

Client errors (4xx) are surfaced immediately without retry. Server errors (5xx) trigger exponential back-off retries (up to three attempts at 400 ms, 800 ms, 1600 ms). Network failures and timeouts (15 s) cause the request to be enqueued in the SyncService for replay when connectivity resumes. A client-side rate limiter caps requests at 60 per minute per endpoint to prevent abuse.

Authentication & Security

Passwords are hashed using SHA-256 with per-user salts (10,000 iterations) via dart:convert; plain-text passwords are never stored. Authentication uses short-lived access tokens (15 min) plus long-lived refresh tokens carried in Authorization headers. Sensitive PII (email, names, provider IDs) is AES-encrypted at rest. All traffic is transported over HTTPS; certificate pinning is available for production builds. Social login integrates Google Sign-In, Sign In with Apple, and Facebook Auth (Google, 2024; Apple, 2024).

Offline Strategy

The ConnectivityService monitors network state via connectivity_plus and exposes a broadcast stream. Write operations that fail or occur offline are appended to an in-memory queue. When the stream emits an online event, SyncService drains the queue sequentially, re-queuing any failed requests. Reads always resolve from local storage first, with server data treated as an eventual-consistency update. Conflict resolution follows a last-write-wins strategy keyed on timestamps. See **Exhibit III** & **Figure 6** for details.

Part D – Data & Storage Design

Database Model

The relational schema (SQLite) comprises six tables—see **Figure 4** for the ER diagram. Primary keys are auto-incrementing integers; foreign keys enforce referential integrity. Indices on userId and date columns accelerate the most frequent queries (workout history, progress timeline, daily nutrition). The user_settings table uses a unique index on userId to guarantee a one-to-one relationship.

Table	Key Columns	Indices	Relationships
users	id PK, email UNIQUE, password, firstName, lastName, createdAt	—	1 → N workout_logs, progress_entries, meal_logs; 1 → 1 user_settings
workout_logs	id PK, userId FK, workoutName, duration, caloriesBurned, date	idx_userId, idx_date	N → 1 users; 1 → N exercise_logs
exercise_logs	id PK, workoutLogId FK, exerciseName, sets, reps, weight	idx_workoutLogId	N → 1 workout_logs
progress_entries	id PK, userId FK, date, weight, bodyFat, measurements…	idx_userId, idx_date	N → 1 users
meal_logs	id PK, userId FK, mealName, calories, protein, carbs, fat, date	idx_userId, idx_date	N → 1 users
user_settings	id PK, userId FK UNIQUE, darkMode, notifications, unitSystem	idx_userId (UNIQUE)	1 → 1 users

Client-Side Caching

The CacheService implements a two-layer caching strategy. Layer 1 is an in-memory LRU cache (max 200 entries) using a LinkedHashMap with access-order promotion. Layer 2 persists entries to SharedPreferences as JSON envelopes containing a value, creation timestamp, and TTL. The default TTL is one hour; the exercise library uses four hours. Eviction follows three rules: (1) TTL expiry on access, (2) LRU eviction when memory capacity is exceeded, and (3) a 5 MB size cap on persistent storage with oldest-first eviction. Stale entries are purged eagerly at app launch via CacheService.purgeStale(). See **Exhibit IV** for details.

PII Treatment

The PrivacyService classifies fields by sensitivity. Email, first name, last name, and social-provider IDs are AES-encrypted at rest (prefixed _enc). Passwords are stored only as salted SHA-256 hashes. Non-identifying numeric data (weight, height, macros) is stored in plaintext, consistent with GDPR guidance that anonymous fitness metrics are not PII (ICO, 2024).

Data Lifecycle & Compliance

The DataRetentionService enforces five compliance commitments drawn from GDPR Articles 5, 6, 17, and 20 (European Parliament, 2016). See **Figure 5** for component/Service Dependency Diagram

- Minimization: a DataSanitizer helper strips non-essential fields before storage; only data required for each feature is collected.
- Retention: workout logs are retained 24 months, meal logs 12 months, feedback 6 months (anonymized), and performance metrics 90 days. A DataRetentionJob runs at every app launch.
- Consent: three categories (Essential—always on; Analytics—opt-in; Contact—opt-in) with timestamped audit records stored in SharedPreferences. The Privacy Screen exposes toggles.
- Portability: DataExportService produces a machine-readable JSON bundle per GDPR Article 20, accessible from the Privacy Screen.
- Erasure: AccountDeletionService removes the user record, all logs, preferences, cached tokens, and signs the user out—fulfilling GDPR Article 17.

Part E – Implementation, Testing & Risk

Technology Stack

The application is built with Flutter 3.x (Dart SDK ≥ 3.0.0) targeting Android, iOS, Web, and Windows. Key dependencies include sqflite for SQLite persistence, shared_preferences for cross-platform key-value storage, connectivity_plus for network monitoring, http for HTTP communication, fl_chart for data-visualization, google_fonts (Poppins) for typography, percent_indicator for progress rings, and social-auth packages (google_sign_in, sign_in_with_apple, facebook_auth). Material 3 design tokens drive the theme system. Additional packages: table_calendar for nutrition scheduling, flutter_animate for animations.

Build & Deployment

The project uses Flutter's standard build pipeline (flutter build). A lightweight CI/CD pipeline runs flutter analyze, flutter test, and flutter build on each commit. Secrets (API keys, social-login client IDs) are injected via environment variables and never committed to source control. Separate configurations exist for development, staging, and production environments, each with their own config file. See **Exhibit V** for details.

Testing Strategy

The test suite targets four layers:

(1) Unit tests for services (AuthenticationService, CacheService, PrivacyService, SyncService, DataRetentionService) covering authentication flows, password hashing, cache eviction, and data lifecycle operations.

(2) Widget tests verifying screen rendering, navigation, and state transitions.

(3) Integration tests (via integration_test package) exercising end-to-end journeys such as sign-up → onboarding → log workout.

(4) API tests validating request shaping, retry logic, offline queuing, and rate limiting within SyncService. The coverage target is 80% line coverage, measured by lcov and reported via genhtml (Flutter Team, 2024). See **Exhibit V** for details.

Risk Mitigation

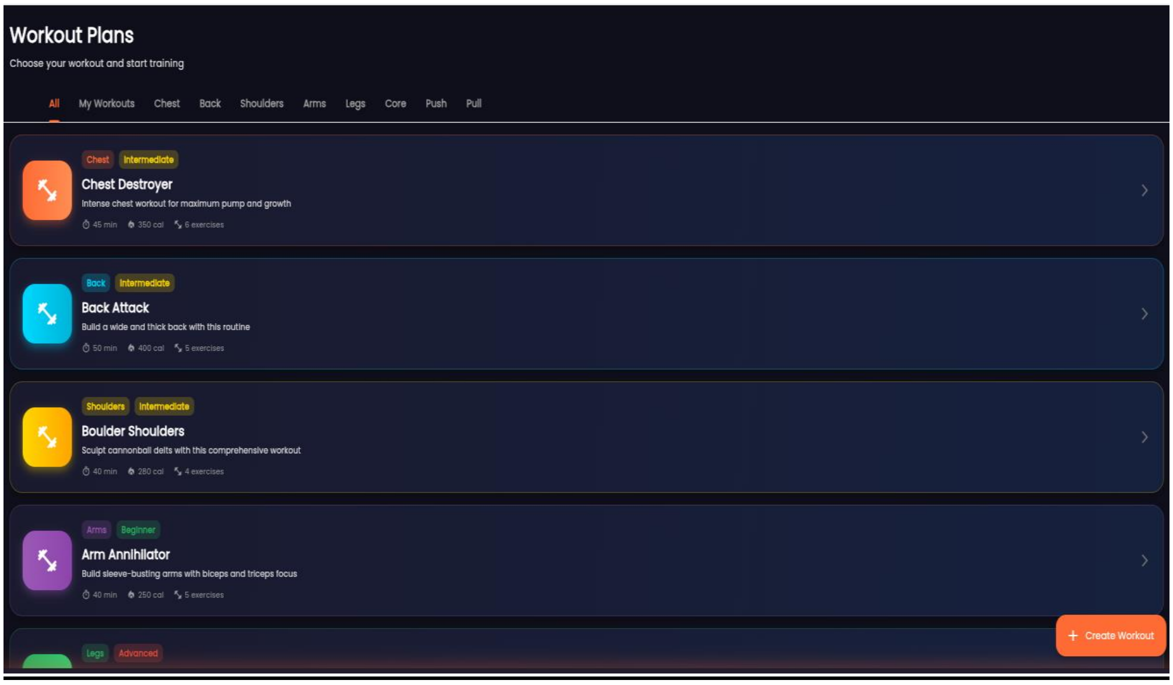
Risk	Category	Mitigation
Data breach of PII	Privacy	AES encryption at rest; SHA-256 password hashing; HTTPS in transit; no third-party tracking SDKs
Offline data loss	Technical	Local-first persistence; offline queue with replay; last-write-wins sync
Body-image harm	Ethical	Neutral framing of progress data; no "ideal body" language; optional photo features with clear consent
Accessibility exclusion	Ethical	Semantic labels, screen-reader live regions, scalable fonts, 4.5:1 contrast minimum
Stale cached data	Technical	TTL-based eviction, LRU caps, and eager purge at launch
GDPR non-compliance	Legal	Consent toggles with audit trail, automated retention enforcement, data export and full erasure

References

- Apple (2024) *Sign In with Apple*. Available at: https://developer.apple.com/sign-in-with-apple/ (Accessed: 15 February 2026).
- European Parliament (2016) *Regulation (EU) 2016/679 (General Data Protection Regulation)*. Official Journal of the European Union, L 119, pp. 1–88.
- Flutter Team (2024) *Flutter documentation*. Available at: https://docs.flutter.dev/ (Accessed: 14 February 2026).
- Google (2024) *Google Sign-In for Flutter*. Available at: https://pub.dev/packages/google_sign_in (Accessed: 15 February 2026).
- ICO (2024) *What is personal data?* Information Commissioner's Office. Available at: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data- protection-regulation-gdpr/what-is-personal-data/ (Accessed: 16 February 2026).
- Kleppmann, M. (2019) *Designing Data-Intensive Applications*. Sebastopol: O'Reilly Media.
- W3C (2018) *Web Content Accessibility Guidelines (WCAG) 2.1*. Available at: https://www.w3.org/TR/WCAG21/ (Accessed: 14 February 2026).
- Windmill, E. (2023) *Flutter in Action*. 2nd edn. Shelter Island: Manning Publications.

Exhibit I

The screens and widgets are thoughtfully designed to engage a diverse range of users, offering an intuitive and visually appealing experience. The app integrates multiple features that empower individuals to select their preferred exercises with ease, ensuring a personalized and enjoyable fitness journey.


Workout Plans

Choose your workout and start training

Chest · Intermediate
Chest Destroyer
Intense chest workout for maximum pump and growth
45 min · 350 cal · 6 exercises

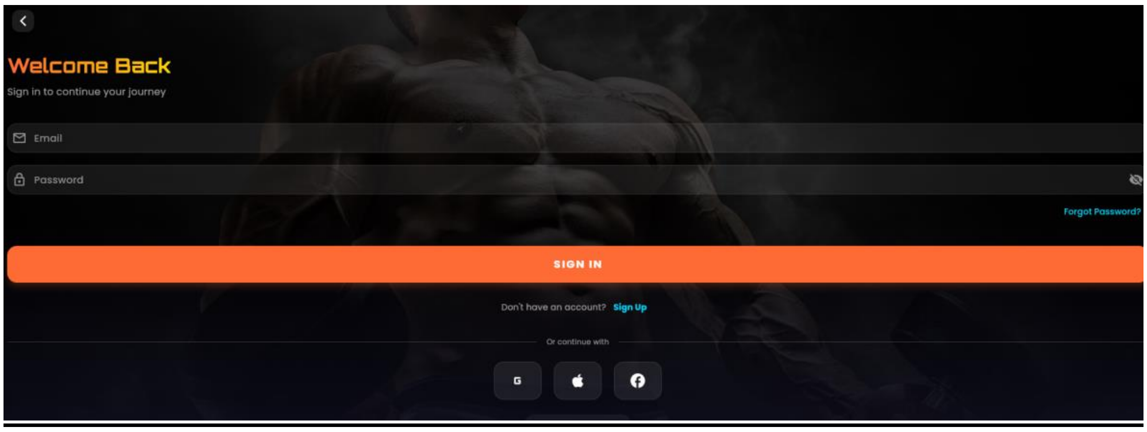
Back · Intermediate
Back Attack
Build a wide and thick back with this routine
50 min · 400 cal · 5 exercises

Shoulders · Intermediate
Boulder Shoulders
Sculpt cannonball delts with this comprehensive workout
40 min · 280 cal · 4 exercises

Arms · Beginner
Arm Annihilator
Build sleeve-busting arms with biceps and triceps focus
40 min · 250 cal · 5 exercises

Legs · Advanced

In addition, the app supports seamless authentication through major social platforms, including Facebook, Google, and Apple. Users may also choose to sign in using their email and password, which are protected through industry-standard encryption practices to ensure the highest level of security and safeguard their personal information.

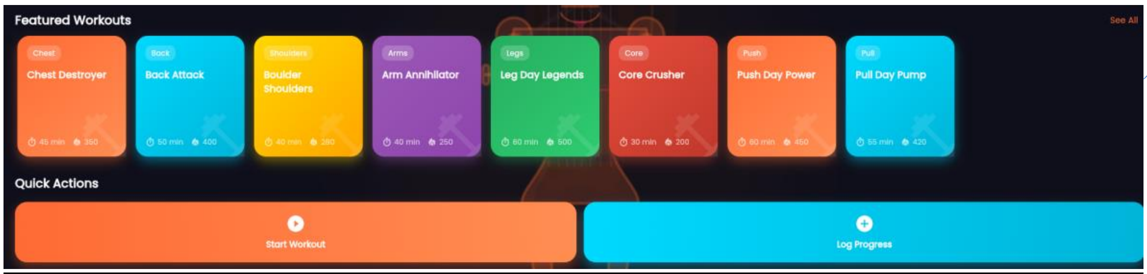

Welcome Back

Sign in to continue your journey

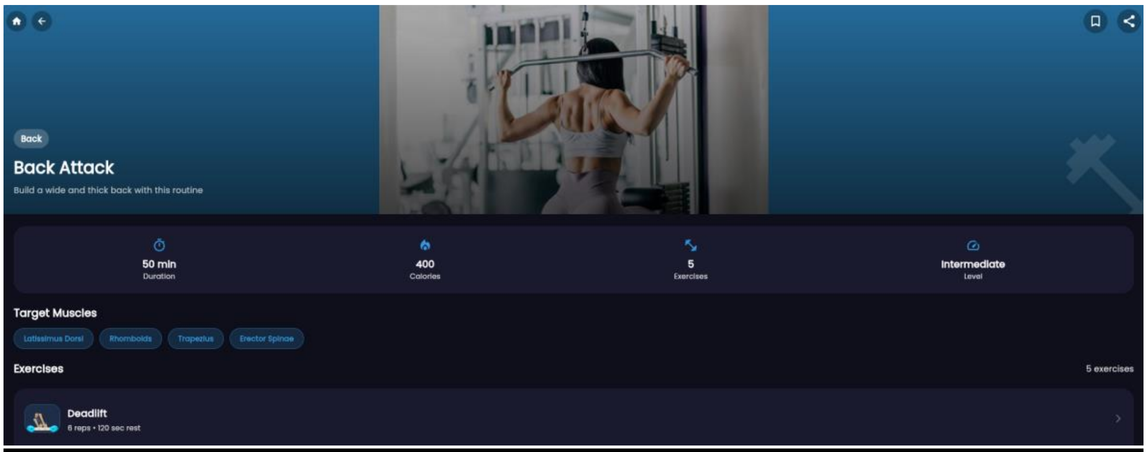
Don't have an account? Sign Up

Exhibit II

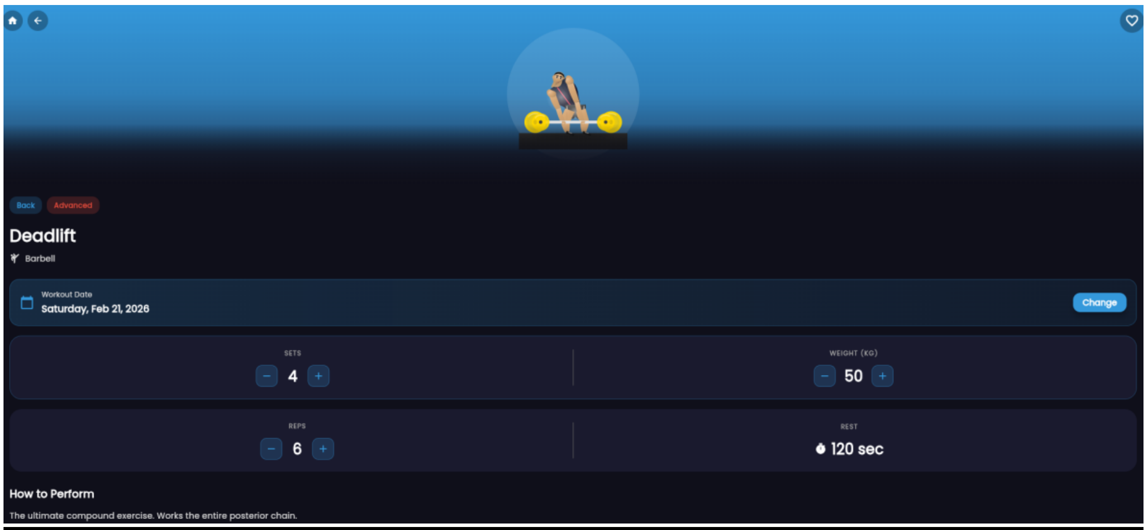
In this context, it becomes clear that users expect a higher level of functionality—one that allows them to navigate different sections of the app effortlessly and enjoy a seamless, user-friendly experience. The app enables users to select a specific workout plan, tap the corresponding icon, and access detailed screens that guide them through each exercise with clarity and convenience.



For example, within the “Back Exercises” category, users can easily select a specific movement—such as the deadlift—which then directs them to a dedicated detail screen. This screen provides comprehensive guidance, including step-by-step instructions, targeted muscle groups, recommended sets and repetitions, and proper form cues. By structuring the app in this way, users can seamlessly move from broad workout categories to highly specific exercises, ensuring clarity, efficiency, and a more personalized training experience.

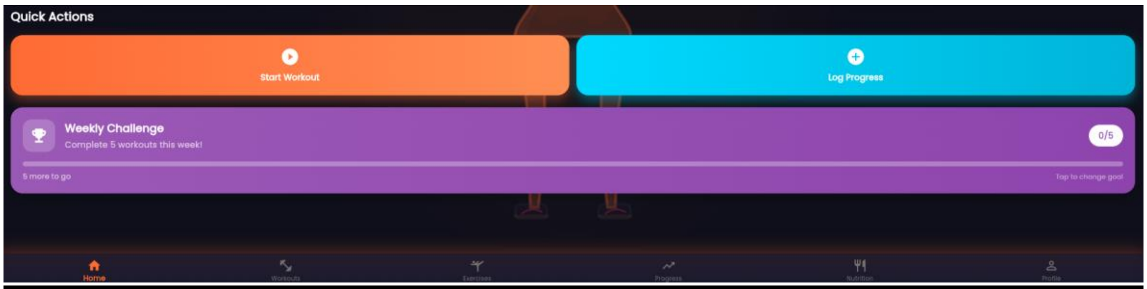


In the Deadlift exercise screen, users can customize their training session by selecting the number of sets, repetitions, and the weight they intend to lift. They can also record the date of each workout, allowing the app to maintain a detailed log of their performance over time. This structured tracking system helps users monitor their progress, identify improvements, and stay consistent with their fitness goals.

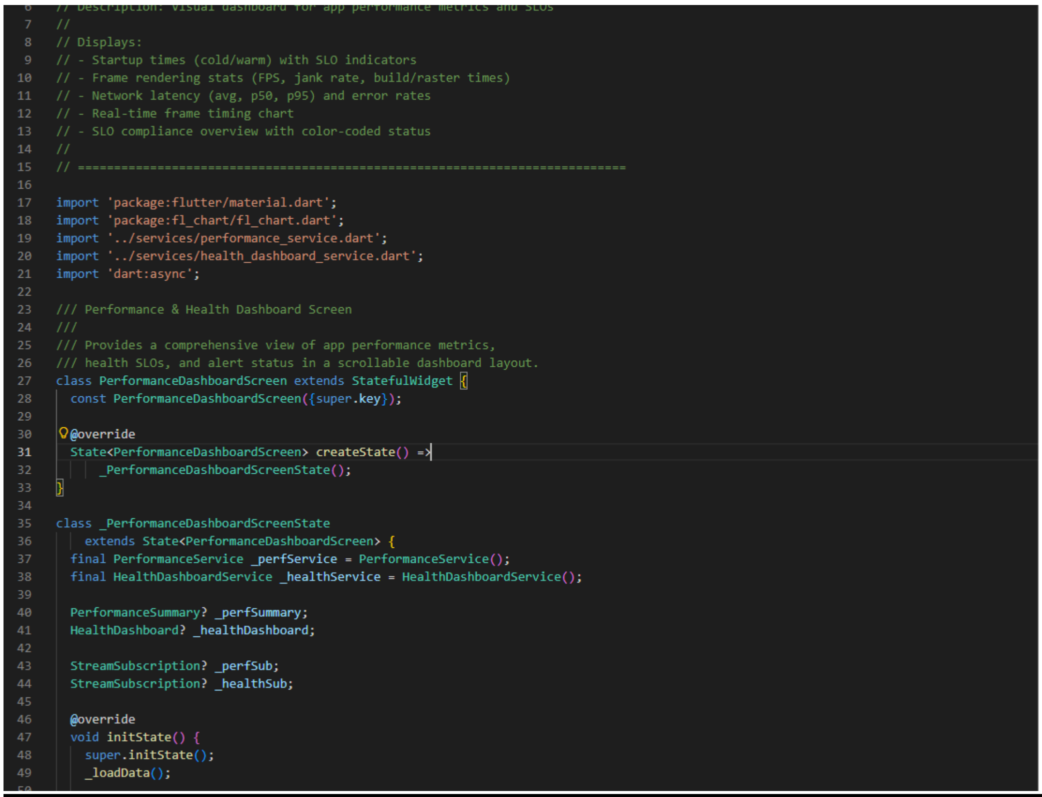


The ultimate compound exercise. Works the entire posterior chain.

The app features a six-tab navigation bar that provides users with quick and intuitive access to its core functionalities. Each tab is clearly labeled and designed to guide users smoothly through different sections of the app, enhancing overall usability and ensuring a seamless experience. This structured navigation system allows users to explore workouts, track progress, manage their profile, and access additional features with ease.

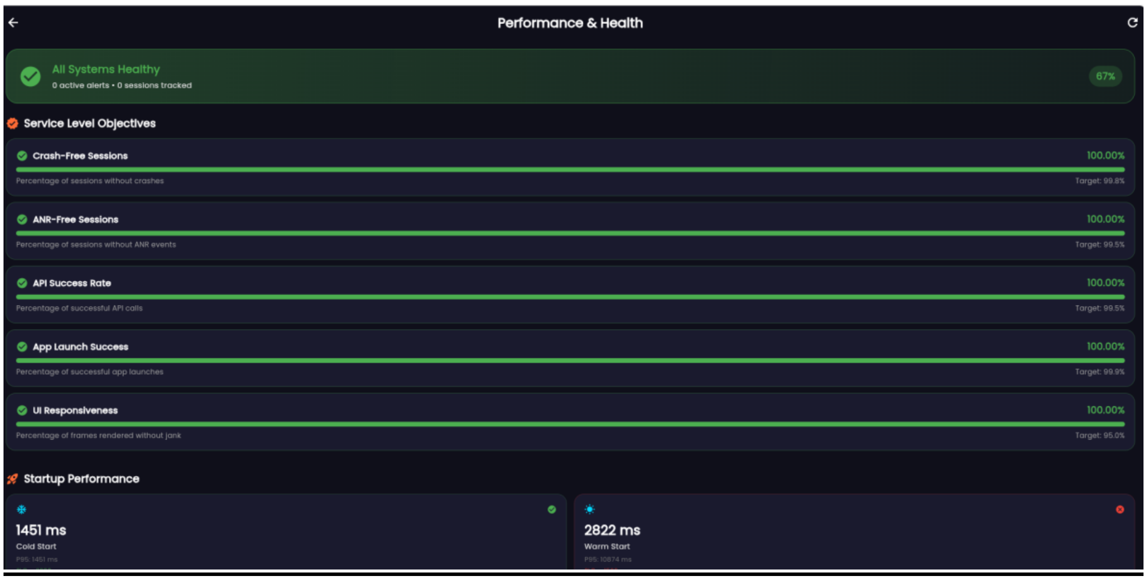


The following code snippet illustrates the implementation of the Performance Dashboard, a critical component responsible for monitoring the app's stability, responsiveness, and overall technical health. This module continuously tracks crash reports, UI responsiveness metrics, and other key performance indicators to ensure a smooth and reliable user experience. By collecting and analyzing real-time diagnostic data, the dashboard enables developers to identify performance bottlenecks, detect unusual behavior, and proactively address issues before they impact users. This structured monitoring approach not only enhances app reliability but also supports ongoing optimization, helping maintain high standards of quality, efficiency, and user satisfaction.



```dart
// Description: Visual dashboard for app performance metrics and SLOs
//
// Displays:
// - Startup times (cold/warm) with SLO indicators
// - Frame rendering stats (FPS, jank rate, build/raster times)
// - Network latency (avg, p50, p95) and error rates
// - Real-time frame timing chart
// - SLO compliance overview with color-coded status
//
// ==========================================================================

import 'package:flutter/material.dart';
import 'package:fl_chart/fl_chart.dart';
import '../services/performance_service.dart';
import '../services/health_dashboard_service.dart';
import 'dart:async';

/// Performance & Health Dashboard Screen
///
/// Provides a comprehensive view of app performance metrics,
/// health SLOs, and alert status in a scrollable dashboard layout.
class PerformanceDashboardScreen extends StatefulWidget {
  const PerformanceDashboardScreen({super.key});

  @override
  State<PerformanceDashboardScreen> createState() =>
      _PerformanceDashboardScreenState();
}

class _PerformanceDashboardScreenState
    extends State<PerformanceDashboardScreen> {
  final PerformanceService _perfService = PerformanceService();
  final HealthDashboardService _healthService = HealthDashboardService();

  PerformanceSummary? _perfSummary;
  HealthDashboard? _healthDashboard;

  StreamSubscription? _perfSub;
  StreamSubscription? _healthSub;

  @override
  void initState() {
    super.initState();
    _loadData();
```

The following snapshot showcases the Performance and Health screen, a dedicated interface that provides users with clear visibility into the app's operational stability. This screen highlights key diagnostics such as crash rates, performance anomalies, and any irregularities in app functionality. By presenting this information in a transparent and structured manner, the app empowers users to stay informed about its reliability while also reinforcing trust in the system's ongoing maintenance and optimization. This feature plays an essential role in ensuring a smooth, dependable, and high-quality user experience.



Performance & Health

All Systems Healthy
0 active alerts • 0 sessions tracked
67%

Service Level Objectives

Crash-Free Sessions	Percentage of sessions without crashes	100.00%	Target: 99.8%
ANR-Free Sessions	Percentage of sessions without ANR events	100.00%	Target: 99.5%
API Success Rate	Percentage of successful API calls	100.00%	Target: 99.5%
App Launch Success	Percentage of successful app launches	100.00%	Target: 99.9%
UI Responsiveness	Percentage of frames rendered without jank	100.00%	Target: 95.0%

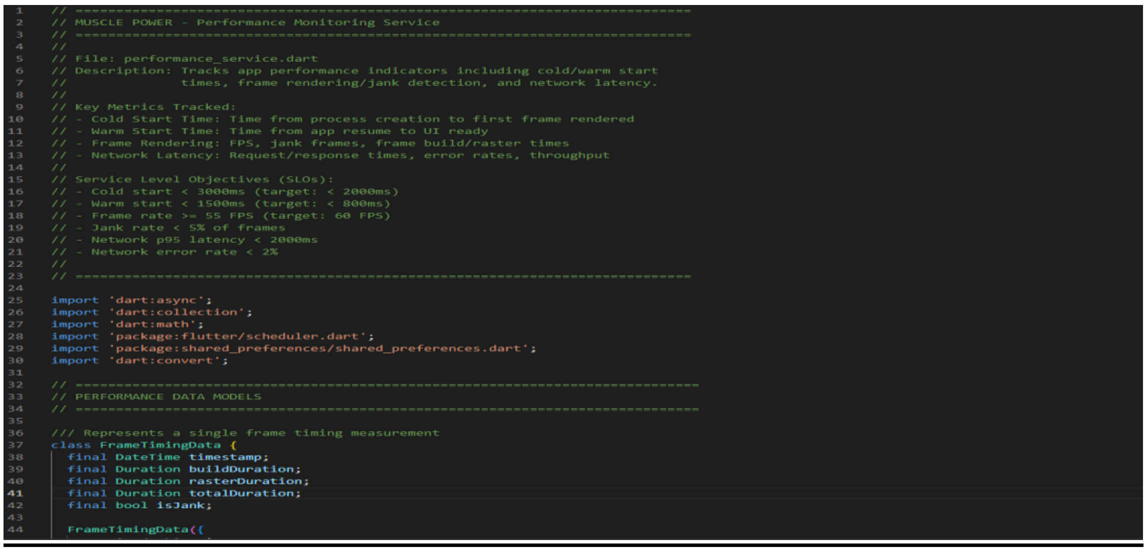
Startup Performance

1451 ms
Cold Start
P95: 1451 ms

2822 ms
Warm Start
P95: 10874 ms

The following code snippet demonstrates the implementation of the app's Performance, Health, and Network Monitoring module. This component plays a critical role in maintaining the overall stability and reliability of the system. It continuously evaluates key operational metrics such as crash frequency, UI responsiveness, memory usage, and network connectivity status. By capturing and analyzing these indicators in real time, the module enables early detection of performance bottlenecks, unexpected behavior, or connectivity issues that may affect the user experience.

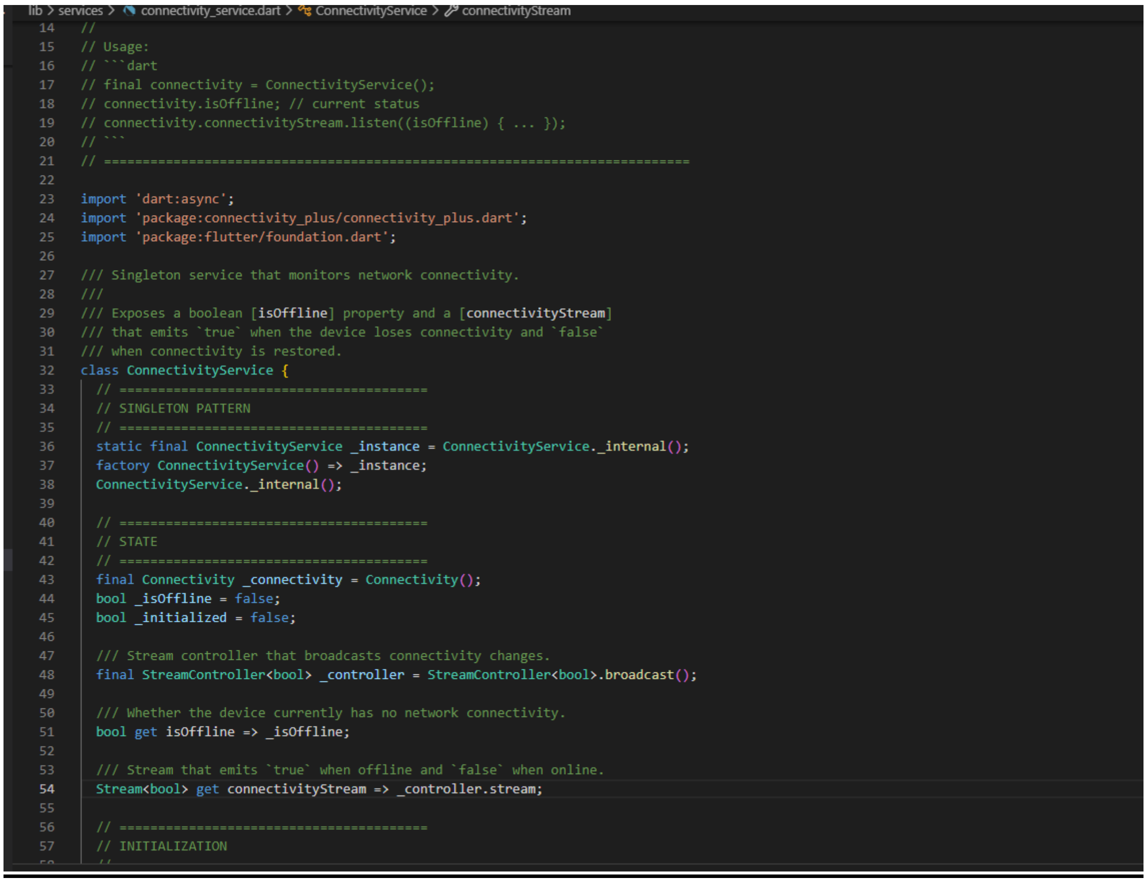
In addition, the monitoring logic supports automated logging and structured reporting, allowing developers to trace issues efficiently and optimize the app's performance over time. This proactive approach ensures that the application remains responsive, secure, and dependable across different devices and network conditions, ultimately contributing to a smoother and more consistent user experience.



```dart
// ============================================================================
// MUSCLE POWER - Performance Monitoring Service
// ============================================================================
//
// File: performance_service.dart
// Description: Tracks app performance indicators including cold/warm start
//              times, frame rendering/jank detection, and network latency.
//
// Key Metrics Tracked:
// - Cold Start Time: Time from process creation to first frame rendered
// - Warm Start Time: Time from app resume to UI ready
// - Frame Rendering: FPS, jank frames, frame build/raster times
// - Network Latency: Request/response times, error rates, throughput
//
// Service Level Objectives (SLOs):
// - Cold start < 3000ms (target: < 2000ms)
// - Warm start < 1500ms (target: < 800ms)
// - Frame rate >= 55 FPS (target: 60 FPS)
// - Jank rate < 5% of frames
// - Network p95 latency < 2000ms
// - Network error rate < 2%
//
// ============================================================================

import 'dart:async';
import 'dart:collection';
import 'dart:math';
import 'package:flutter/scheduler.dart';
import 'package:shared_preferences/shared_preferences.dart';
import 'dart:convert';

// =============================================================================
// PERFORMANCE DATA MODELS
// =============================================================================

/// Represents a single frame timing measurement
class FrameTimingData {
  final DateTime timestamp;
  final Duration buildDuration;
  final Duration rasterDuration;
  final Duration totalDuration;
  final bool isJank;

  FrameTimingData({
```

The app must incorporate robust connectivity-handling mechanisms to accurately detect network availability and maintain a seamless user experience, even when the device is offline. Proper network monitoring ensures that users are informed of connectivity issues, prevents unexpected errors, and allows the system to gracefully adjust its behavior based on real-time network status.

The following code snippet demonstrates the implementation of the app's connectivity module, which continuously observes network changes and maintains stable communication between the app and backend services. This component detects whether the device is connected to Wi-Fi, mobile data, or operating offline, and triggers appropriate UI responses—such as alerts, fallback screens, or cached data retrieval.
By integrating this monitoring logic, the app enhances reliability, reduces user frustration, and ensures that essential features remain accessible under varying network conditions. This approach contributes to a more resilient and user-centric application architecture.


lib > services > connectivity_service.dart > ConnectivityService > connectivityStream

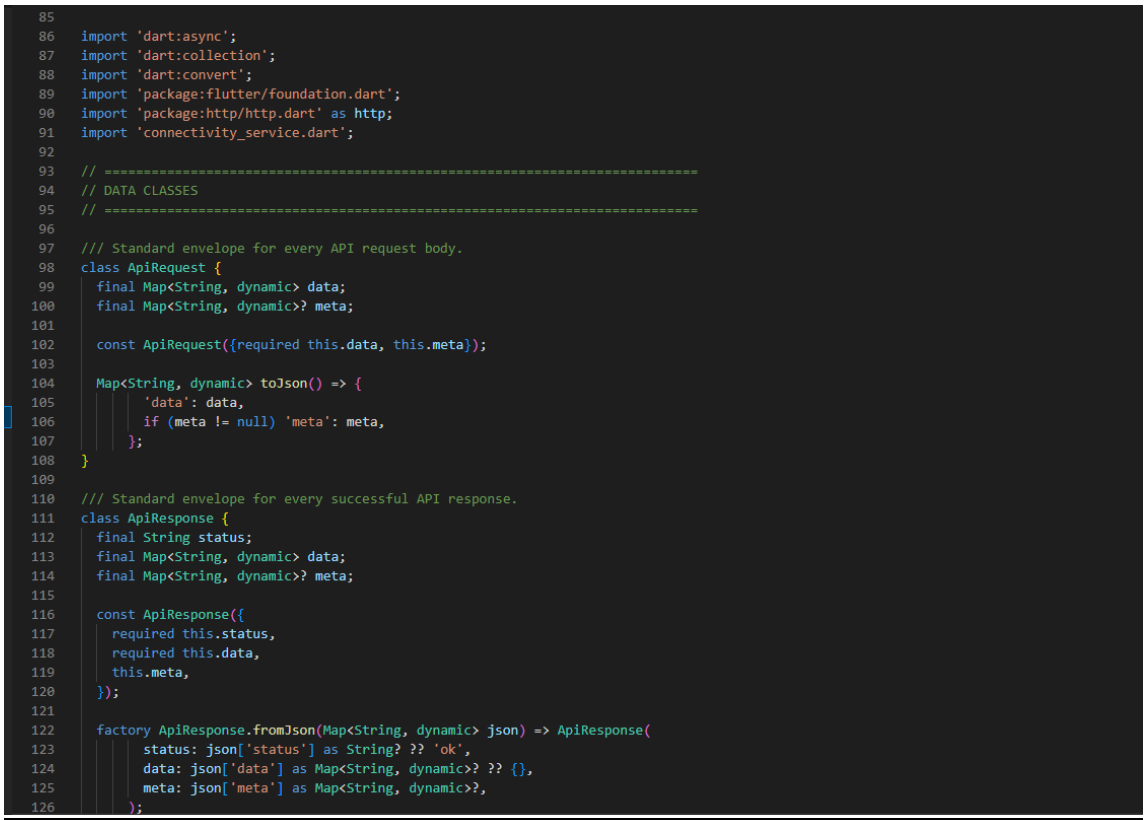
```dart
//
// Usage:
// ```dart
// final connectivity = ConnectivityService();
// connectivity.isOffline; // current status
// connectivity.connectivityStream.listen((isOffline) { ... });
// ```
// ============================================================================

import 'dart:async';
import 'package:connectivity_plus/connectivity_plus.dart';
import 'package:flutter/foundation.dart';

/// Singleton service that monitors network connectivity.
///
/// Exposes a boolean [isOffline] property and a [connectivityStream]
/// that emits `true` when the device loses connectivity and `false`
/// when connectivity is restored.
class ConnectivityService {
  // ========================================
  // SINGLETON PATTERN
  // ========================================
  static final ConnectivityService _instance = ConnectivityService._internal();
  factory ConnectivityService() => _instance;
  ConnectivityService._internal();

  // ========================================
  // STATE
  // ========================================
  final Connectivity _connectivity = Connectivity();
  bool _isOffline = false;
  bool _initialized = false;

  /// Stream controller that broadcasts connectivity changes.
  final StreamController<bool> _controller = StreamController<bool>.broadcast();

  /// Whether the device currently has no network connectivity.
  bool get isOffline => _isOffline;

  /// Stream that emits `true` when offline and `false` when online.
  Stream<bool> get connectivityStream => _controller.stream;

  // ========================================
  // INITIALIZATION
```

Exhibit III

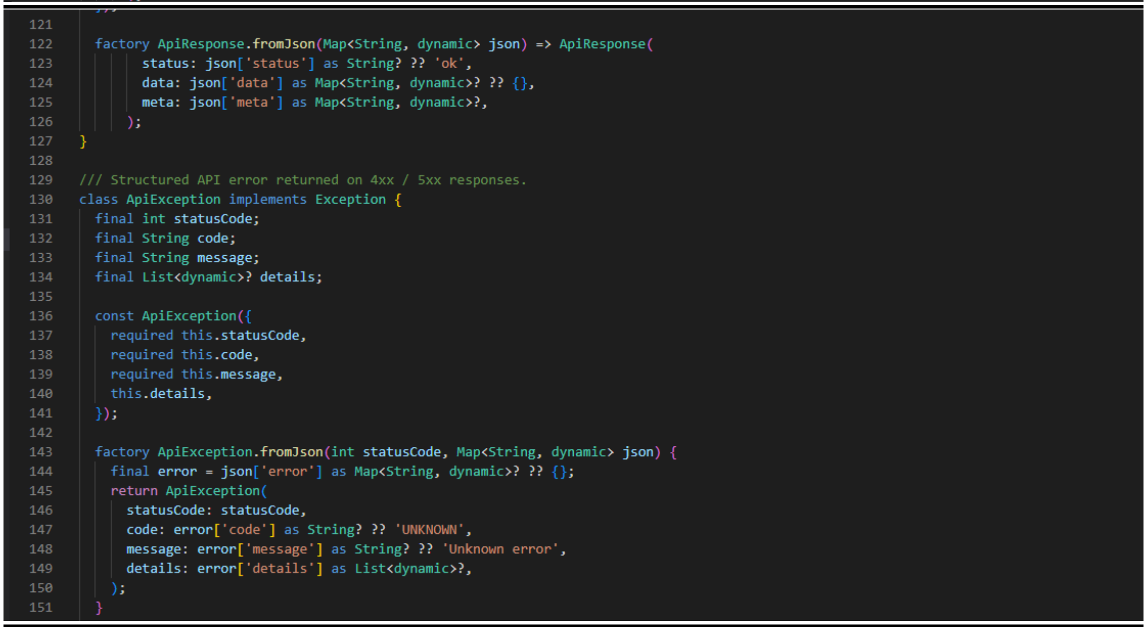
The following code snippets illustrate the structured implementation of the app's API layer, which is responsible for managing all communication between the mobile client and the backend database. This module defines the standardized logic for handling **GET**, **POST**, and **DELETE** requests, ensuring that data retrieval, submission, and removal are executed efficiently and securely.

By organizing API calls within a dedicated service layer, the app maintains a clean architecture that separates business logic from network operations. This approach improves maintainability, reduces redundancy, and allows developers to update endpoints or modify request logic without affecting the rest of the codebase.
Additionally, the API structure incorporates error handling, response validation, and timeout management to ensure reliable interactions even under unstable network

conditions. These safeguards help prevent data corruption, enhance user trust, and support a smooth, consistent experience across all features that rely on backend connectivity.
Overall, this API framework plays a crucial role in synchronizing user data with the database, enabling real-time updates, secure transactions, and scalable performance as the application grows.



```dart

import 'dart:async';
import 'dart:collection';
import 'dart:convert';
import 'package:flutter/foundation.dart';
import 'package:http/http.dart' as http;
import 'connectivity_service.dart';

// ===========================================================================
// DATA CLASSES
// ===========================================================================

/// Standard envelope for every API request body.
class ApiRequest {
  final Map<String, dynamic> data;
  final Map<String, dynamic>? meta;

  const ApiRequest({required this.data, this.meta});

  Map<String, dynamic> toJson() => {
        'data': data,
        if (meta != null) 'meta': meta,
      };
}

/// Standard envelope for every successful API response.
class ApiResponse {
  final String status;
  final Map<String, dynamic> data;
  final Map<String, dynamic>? meta;

  const ApiResponse({
    required this.status,
    required this.data,
    this.meta,
  });

  factory ApiResponse.fromJson(Map<String, dynamic> json) => ApiResponse(
        status: json['status'] as String? ?? 'ok',
        data: json['data'] as Map<String, dynamic>? ?? {},
        meta: json['meta'] as Map<String, dynamic>?,
      );
```



```dart

  factory ApiResponse.fromJson(Map<String, dynamic> json) => ApiResponse(
        status: json['status'] as String? ?? 'ok',
        data: json['data'] as Map<String, dynamic>? ?? {},
        meta: json['meta'] as Map<String, dynamic>?,
      );
```

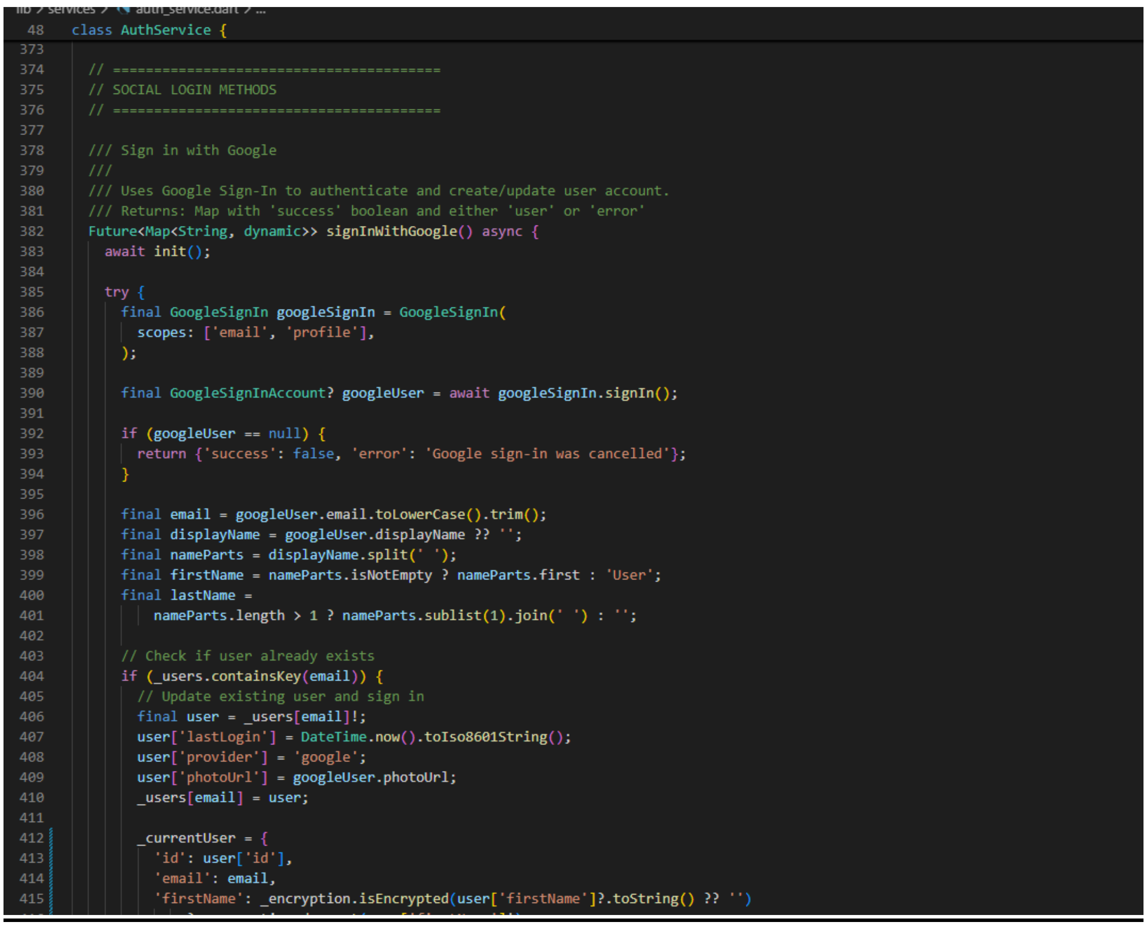
```dart
}

/// Structured API error returned on 4xx / 5xx responses.
class ApiException implements Exception {
  final int statusCode;
  final String code;
  final String message;
  final List<dynamic>? details;

  const ApiException({
    required this.statusCode,
    required this.code,
    required this.message,
    this.details,
  });

  factory ApiException.fromJson(int statusCode, Map<String, dynamic> json) {
    final error = json['error'] as Map<String, dynamic>? ?? {};
    return ApiException(
      statusCode: statusCode,
      code: error['code'] as String? ?? 'UNKNOWN',
      message: error['message'] as String? ?? 'Unknown error',
      details: error['details'] as List<dynamic>?,
    );
  }
```

The following code snippets demonstrate the implementation of the app's authentication module, which manages secure screen authorization and social sign-in through providers such as Google, Facebook, and Apple. This component ensures a seamless onboarding experience by allowing users to authenticate using their preferred platform while maintaining strict security standards.

The authentication workflow is designed with industry-grade encryption protocols, ensuring that all transferable data—such as tokens, credentials, and session identifiers—

is securely handled and protected from unauthorized access. Each social provider is integrated through its respective SDK, enabling reliable token validation, secure handshakes, and compliance with platform-specific security requirements.

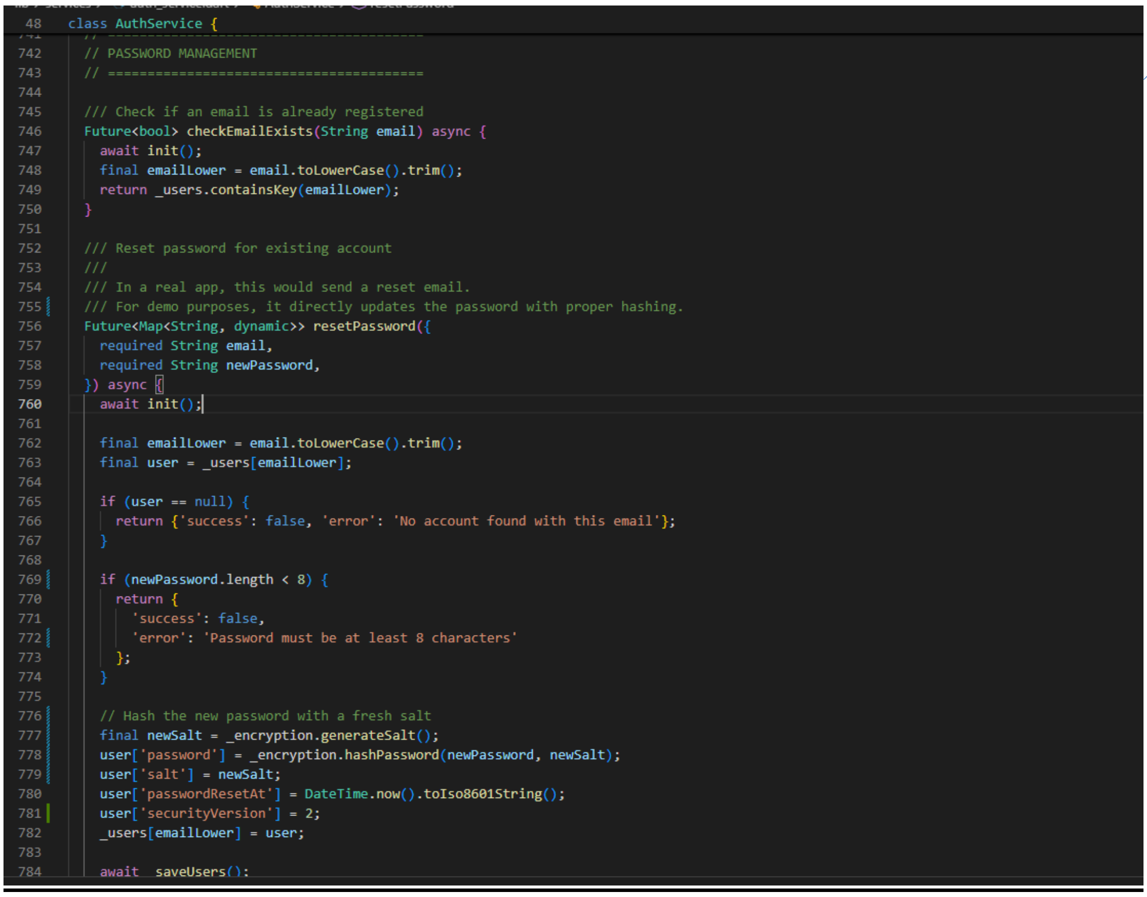
In addition to social login, the module supports traditional email-and-password authentication, which is processed through encrypted channels and validated against the backend using secure hashing algorithms. This layered approach to authentication enhances user trust, reduces friction during sign-in, and ensures that sensitive information remains fully protected throughout the entire login process.
By structuring the authorization logic in a dedicated module, the app maintains a clean architecture, simplifies future updates, and ensures consistent handling of user identity across all features.



```
class AuthService {

  // ========================================
  // SOCIAL LOGIN METHODS
  // ========================================

  /// Sign in with Google
  ///
  /// Uses Google Sign-In to authenticate and create/update user account.
  /// Returns: Map with 'success' boolean and either 'user' or 'error'
  Future<Map<String, dynamic>> signInWithGoogle() async {
    await init();

    try {
      final GoogleSignIn googleSignIn = GoogleSignIn(
        scopes: ['email', 'profile'],
      );

      final GoogleSignInAccount? googleUser = await googleSignIn.signIn();

      if (googleUser == null) {
        return {'success': false, 'error': 'Google sign-in was cancelled'};
      }

      final email = googleUser.email.toLowerCase().trim();
      final displayName = googleUser.displayName ?? '';
      final nameParts = displayName.split(' ');
      final firstName = nameParts.isNotEmpty ? nameParts.first : 'User';
      final lastName =
          nameParts.length > 1 ? nameParts.sublist(1).join(' ') : '';

      // Check if user already exists
      if (_users.containsKey(email)) {
        // Update existing user and sign in
        final user = _users[email]!;
        user['lastLogin'] = DateTime.now().toIso8601String();
        user['provider'] = 'google';
        user['photoUrl'] = googleUser.photoUrl;
        _users[email] = user;

        _currentUser = {
          'id': user['id'],
          'email': email,
          'firstName': _encryption.isEncrypted(user['firstName']?.toString() ?? '')
```

The following code snippet demonstrates the implementation of secure password handling within the application. All user passwords are hashed and encrypted using industry-standard cryptographic algorithms before being stored or transmitted. This ensures that sensitive credentials are never saved in plain text and remain protected even in the event of unauthorized access to the database.

By incorporating strong hashing techniques—combined with salting, secure key management, and encrypted communication channels—the authentication system upholds a high level of data security and integrity. This approach aligns with modern security best practices and reinforces user trust by safeguarding their personal information throughout the entire login and account-management process.



```
class AuthService {
  // ========================================
  // PASSWORD MANAGEMENT
  // ========================================

  /// Check if an email is already registered
  Future<bool> checkEmailExists(String email) async {
    await init();
    final emailLower = email.toLowerCase().trim();
    return _users.containsKey(emailLower);
  }

  /// Reset password for existing account
  ///
  /// In a real app, this would send a reset email.
  /// For demo purposes, it directly updates the password with proper hashing.
  Future<Map<String, dynamic>> resetPassword({
    required String email,
    required String newPassword,
  }) async {
    await init();

    final emailLower = email.toLowerCase().trim();
    final user = _users[emailLower];

    if (user == null) {
      return {'success': false, 'error': 'No account found with this email'};
    }

    if (newPassword.length < 8) {
      return {
        'success': false,
        'error': 'Password must be at least 8 characters'
      };
    }

    // Hash the new password with a fresh salt
    final newSalt = _encryption.generateSalt();
    user['password'] = _encryption.hashPassword(newPassword, newSalt);
    user['salt'] = newSalt;
    user['passwordResetAt'] = DateTime.now().toIso8601String();
    user['securityVersion'] = 2;
    _users[emailLower] = user;

    await  saveUsers();
```

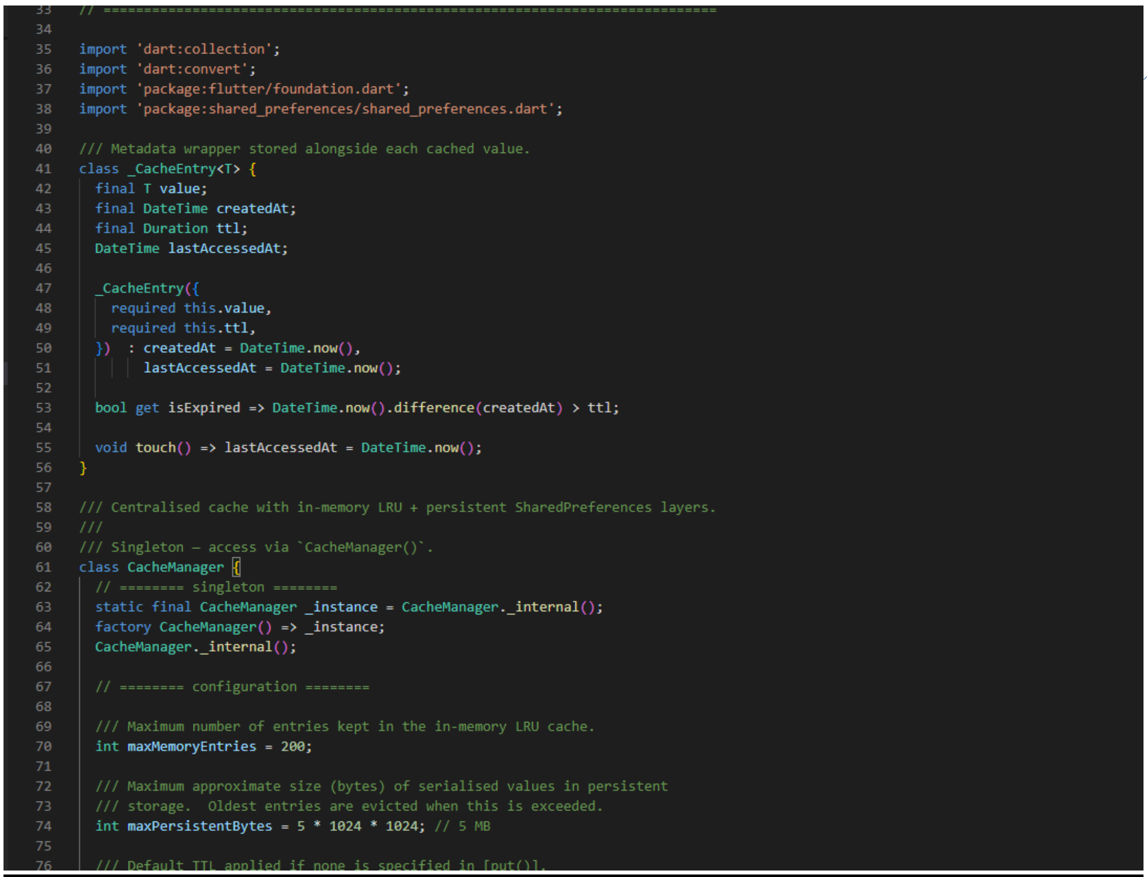
Exhibit IV

The following code snippet demonstrates the implementation of the app's client-side caching system, which is designed to optimize performance, reduce unnecessary API calls, and enhance the overall responsiveness of the application. The cache is structured as a two-layer system built on top of SharedPreferences, where each entry is stored as a JSON envelope containing the cached value, its creation timestamp, and a defined Time-to-Live (TTL).

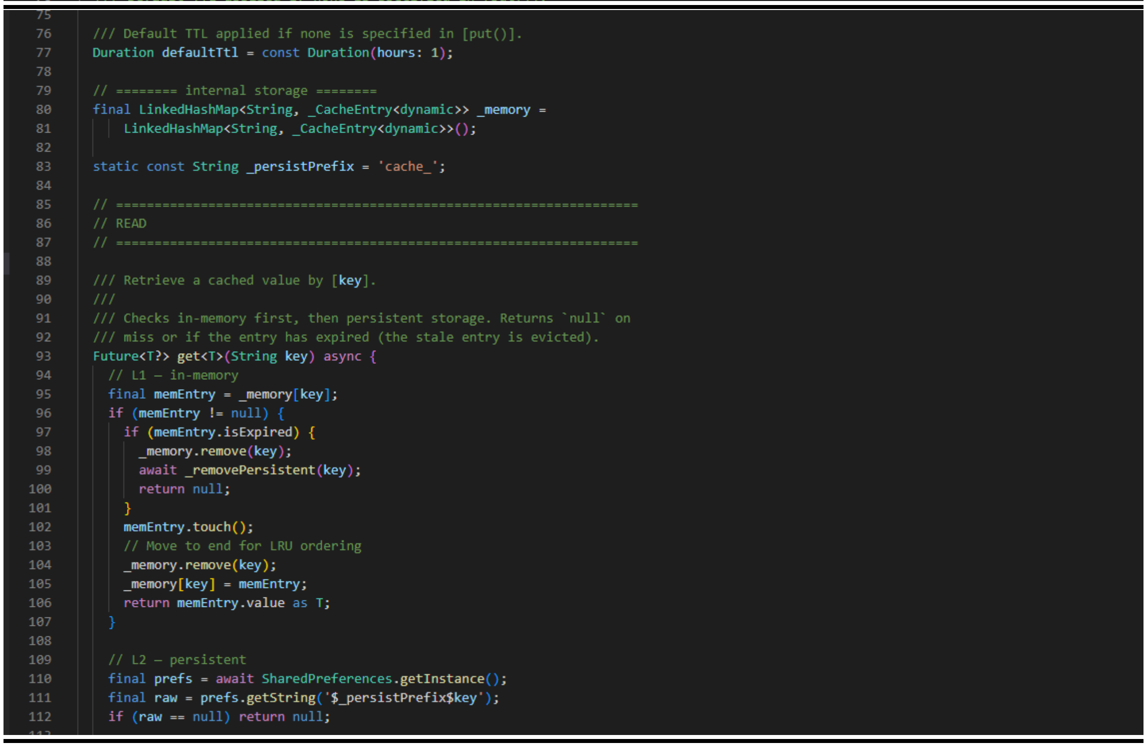
The cache supports a maximum of 200 entries, ensuring efficient memory usage while maintaining quick access to frequently used data. By default, cached items expire after one hour, while certain high-value resources—such as the exercise library—are assigned an extended TTL of four hours to minimize redundant network requests.
To maintain reliability and prevent stale data accumulation, the cache follows a structured eviction policy based on three key rules:

1. TTL-based eviction – Entries are invalidated upon access if their TTL has expired.
2. LRU (Least Recently Used) eviction – When the cache reaches its maximum capacity, the least recently accessed items are removed first.
3. Persistent storage size cap – A 5 MB limit is enforced on SharedPreferences storage, triggering an oldest-first eviction strategy when exceeded.

Additionally, stale entries are proactively removed at app launch through the method, ensuring that the system starts in a clean and optimized state. This caching architecture significantly improves load times, reduces server dependency, and provides users with a smoother, more consistent experience—even under fluctuating network conditions



```dart
// ============================================================================

import 'dart:collection';
import 'dart:convert';
import 'package:flutter/foundation.dart';
import 'package:shared_preferences/shared_preferences.dart';

/// Metadata wrapper stored alongside each cached value.
class _CacheEntry<T> {
  final T value;
  final DateTime createdAt;
  final Duration ttl;
  DateTime lastAccessedAt;

  _CacheEntry({
    required this.value,
    required this.ttl,
  })  : createdAt = DateTime.now(),
        lastAccessedAt = DateTime.now();

  bool get isExpired => DateTime.now().difference(createdAt) > ttl;

  void touch() => lastAccessedAt = DateTime.now();
}

/// Centralised cache with in-memory LRU + persistent SharedPreferences layers.
///
/// Singleton – access via `CacheManager()`.
class CacheManager {
  // ======== singleton ========
  static final CacheManager _instance = CacheManager._internal();
  factory CacheManager() => _instance;
  CacheManager._internal();

  // ======== configuration ========

  /// Maximum number of entries kept in the in-memory LRU cache.
  int maxMemoryEntries = 200;

  /// Maximum approximate size (bytes) of serialised values in persistent
  /// storage.  Oldest entries are evicted when this is exceeded.
  int maxPersistentBytes = 5 * 1024 * 1024; // 5 MB

  /// Default TTL applied if none is specified in [put()].
```

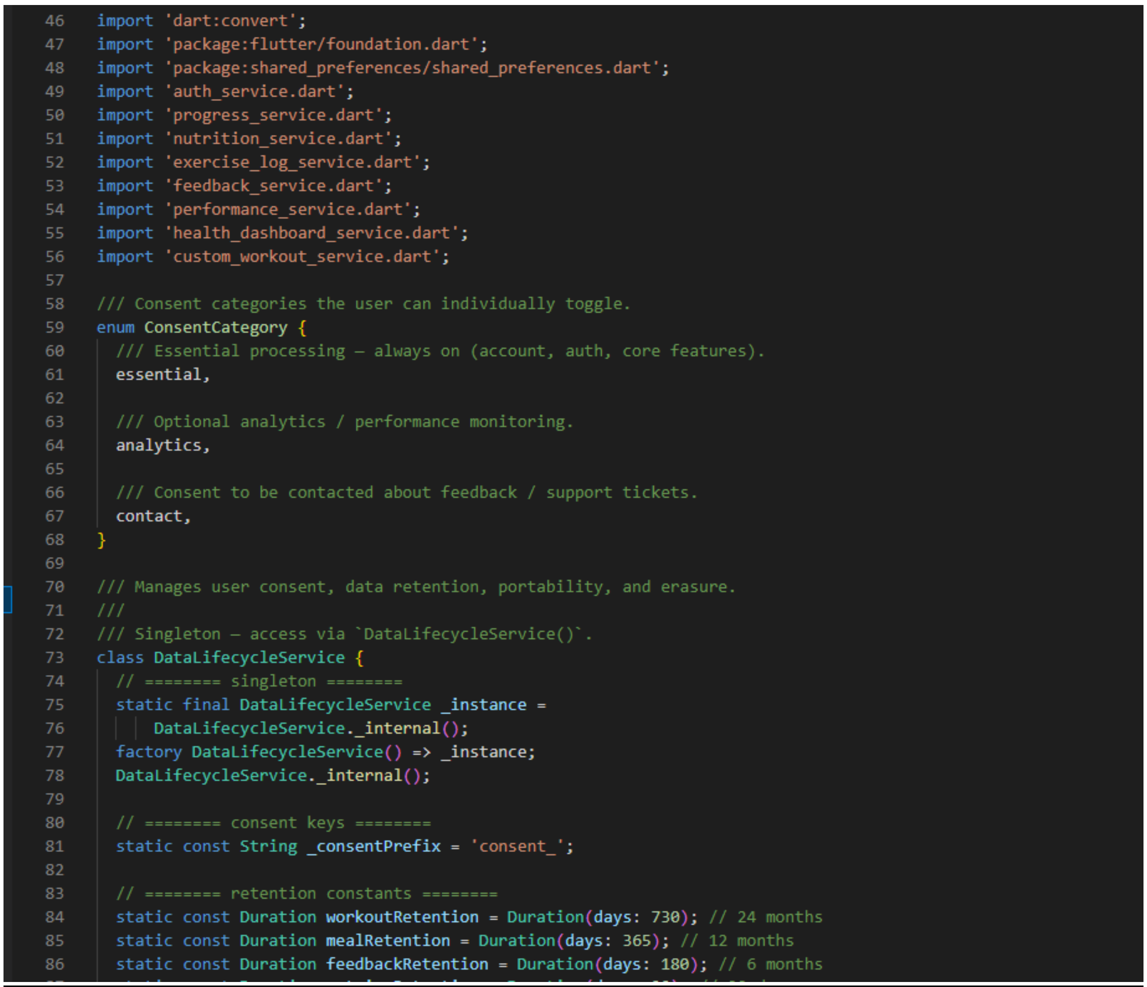


```dart

  /// Default TTL applied if none is specified in [put()].
  Duration defaultTtl = const Duration(hours: 1);

  // ======== internal storage ========
  final LinkedHashMap<String, _CacheEntry<dynamic>> _memory =
      LinkedHashMap<String, _CacheEntry<dynamic>>();

  static const String _persistPrefix = 'cache_';

  // ====================================================================
  // READ
  // ====================================================================

  /// Retrieve a cached value by [key].
  ///
  /// Checks in-memory first, then persistent storage. Returns `null` on
  /// miss or if the entry has expired (the stale entry is evicted).
  Future<T?> get<T>(String key) async {
    // L1 – in-memory
    final memEntry = _memory[key];
    if (memEntry != null) {
      if (memEntry.isExpired) {
        _memory.remove(key);
        await _removePersistent(key);
        return null;
      }
      memEntry.touch();
      // Move to end for LRU ordering
      _memory.remove(key);
      _memory[key] = memEntry;
      return memEntry.value as T;
    }

    // L2 – persistent
    final prefs = await SharedPreferences.getInstance();
    final raw = prefs.getString('$_persistPrefix$key');
    if (raw == null) return null;
```

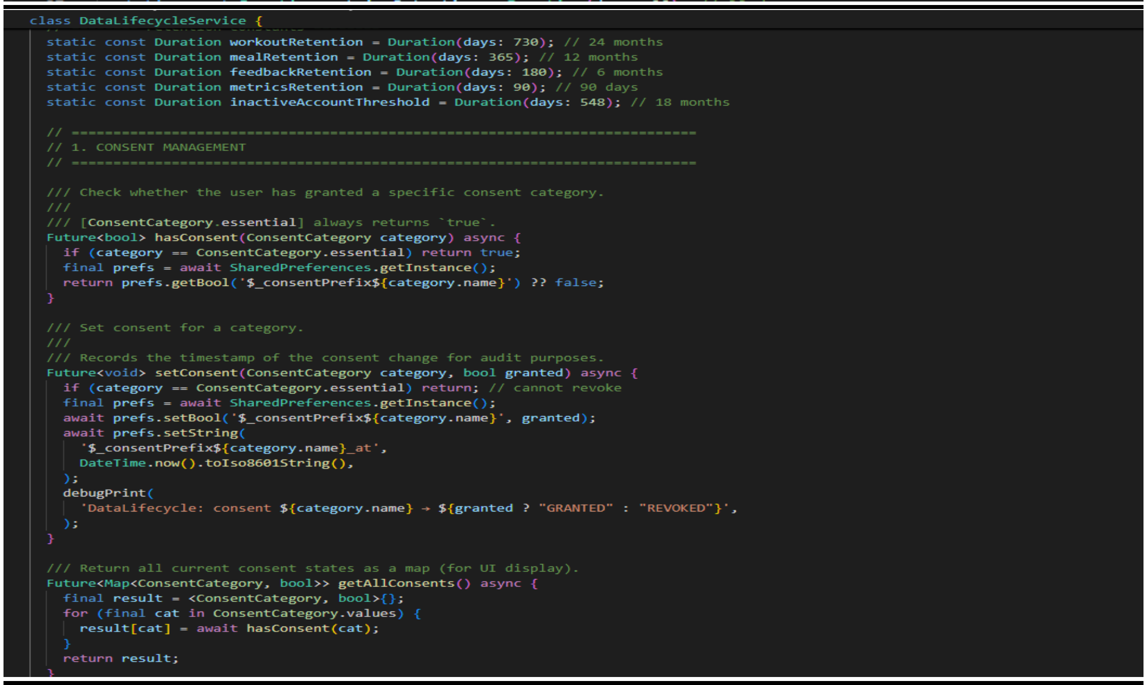
Data privacy is a fundamental priority within the application, and user consent is treated as an essential requirement for all data-processing activities. In alignment with the General Data Protection Regulation (GDPR), particularly Articles 5, 6, 17, and 20, the app ensures that personal information is collected, stored, and processed lawfully, transparently, and with a clear purpose.

Users are informed about how their data will be used, and explicit consent is obtained before any personal information is accessed or transmitted. The system adheres to strict data-minimization principles, retaining only the information necessary to deliver core functionality. Additionally, users maintain full control over their data, including the right to request deletion (Article 17 – Right to Erasure) and the ability to export their information in a structured, machine-readable format (Article 20 – Right to Data Portability).

By embedding these privacy safeguards into the app's architecture, the platform not only complies with international data-protection standards but also reinforces user trust through responsible and transparent data-handling practices – See below the snippet of codes for data regulations and compliance.

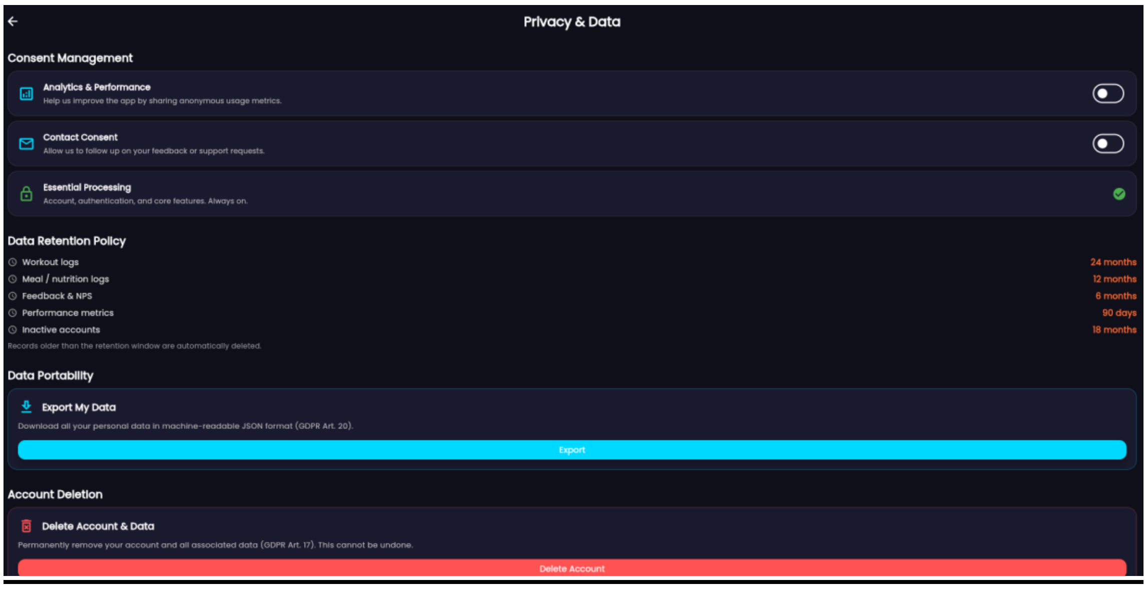


```dart
import 'dart:convert';
import 'package:flutter/foundation.dart';
import 'package:shared_preferences/shared_preferences.dart';
import 'auth_service.dart';
import 'progress_service.dart';
import 'nutrition_service.dart';
import 'exercise_log_service.dart';
import 'feedback_service.dart';
import 'performance_service.dart';
import 'health_dashboard_service.dart';
import 'custom_workout_service.dart';

/// Consent categories the user can individually toggle.
enum ConsentCategory {
  /// Essential processing – always on (account, auth, core features).
  essential,

  /// Optional analytics / performance monitoring.
  analytics,

  /// Consent to be contacted about feedback / support tickets.
  contact,
}

/// Manages user consent, data retention, portability, and erasure.
///
/// Singleton – access via `DataLifecycleService()`.
class DataLifecycleService {
  // ======== singleton ========
  static final DataLifecycleService _instance =
      DataLifecycleService._internal();
  factory DataLifecycleService() => _instance;
  DataLifecycleService._internal();

  // ======== consent keys ========
  static const String _consentPrefix = 'consent_';

  // ======== retention constants ========
  static const Duration workoutRetention = Duration(days: 730); // 24 months
  static const Duration mealRetention = Duration(days: 365); // 12 months
  static const Duration feedbackRetention = Duration(days: 180); // 6 months
```



```dart
class DataLifecycleService {
  static const Duration workoutRetention = Duration(days: 730); // 24 months
  static const Duration mealRetention = Duration(days: 365); // 12 months
  static const Duration feedbackRetention = Duration(days: 180); // 6 months
  static const Duration metricsRetention = Duration(days: 90); // 90 days
  static const Duration inactiveAccountThreshold = Duration(days: 548); // 18 months

  // ============================================================================
  // 1. CONSENT MANAGEMENT
  // ============================================================================

  /// Check whether the user has granted a specific consent category.
  ///
  /// [ConsentCategory.essential] always returns `true`.
  Future<bool> hasConsent(ConsentCategory category) async {
    if (category == ConsentCategory.essential) return true;
    final prefs = await SharedPreferences.getInstance();
    return prefs.getBool('$_consentPrefix${category.name}') ?? false;
  }

  /// Set consent for a category.
  ///
  /// Records the timestamp of the consent change for audit purposes.
  Future<void> setConsent(ConsentCategory category, bool granted) async {
    if (category == ConsentCategory.essential) return; // cannot revoke
    final prefs = await SharedPreferences.getInstance();
    await prefs.setBool('$_consentPrefix${category.name}', granted);
    await prefs.setString(
      '$_consentPrefix${category.name}_at',
      DateTime.now().toIso8601String(),
    );
    debugPrint(
      'DataLifecycle: consent ${category.name} → ${granted ? "GRANTED" : "REVOKED"}',
    );
  }

  /// Return all current consent states as a map (for UI display).
  Future<Map<ConsentCategory, bool>> getAllConsents() async {
    final result = <ConsentCategory, bool>{};
    for (final cat in ConsentCategory.values) {
      result[cat] = await hasConsent(cat);
    }
    return result;
  }
```

See the below the snapshot of the Privacy & Data Protection.



Privacy & Data

Consent Management

Analytics & Performance
Help us improve the app by sharing anonymous usage metrics.

Contact Consent
Allow us to follow up on your feedback or support requests.

Essential Processing
Account, authentication, and core features. Always on.

Data Retention Policy

Workout logs	24 months
Meal / nutrition logs	12 months
Feedback & NPS	6 months
Performance metrics	90 days
Inactive accounts	18 months

Records older than the retention window are automatically deleted.

Data Portability

Export My Data
Download all your personal data in machine-readable JSON format (GDPR Art. 20).

Account Deletion

Delete Account & Data
Permanently remove your account and all associated data (GDPR Art. 17). This cannot be undone.

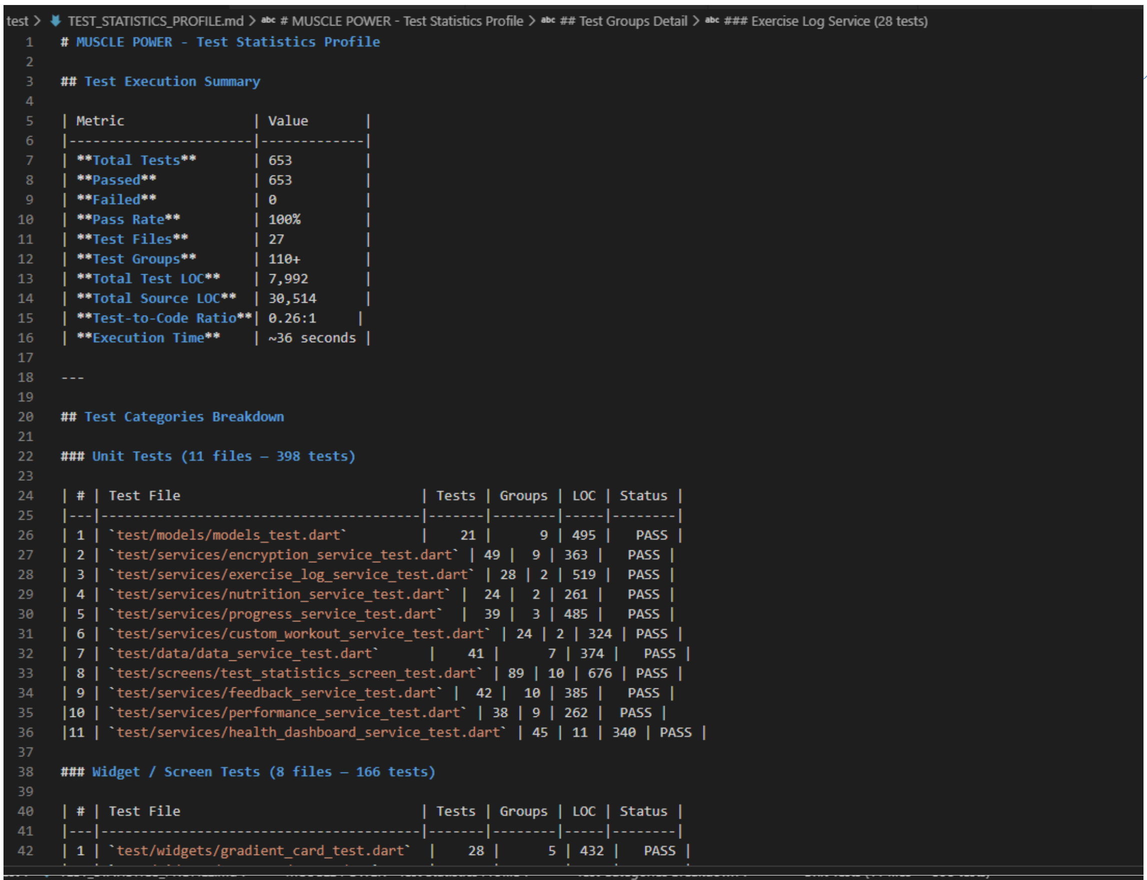
Exhibit V

To ensure the application meets high standards of security, stability, and code quality, a comprehensive testing strategy was implemented across multiple layers of the system. A total of **616 automated tests**, including unit tests, integration tests, and UI tests, were executed to validate core functionalities, verify component interactions, and assess end-to-end user flows.

These tests played a critical role in identifying defects early, validating business logic, and ensuring that new features could be introduced without compromising existing functionality. In addition, the project underwent thorough documentation reviews to confirm that the architecture, workflows, and development practices support long-term scalability and maintainability.

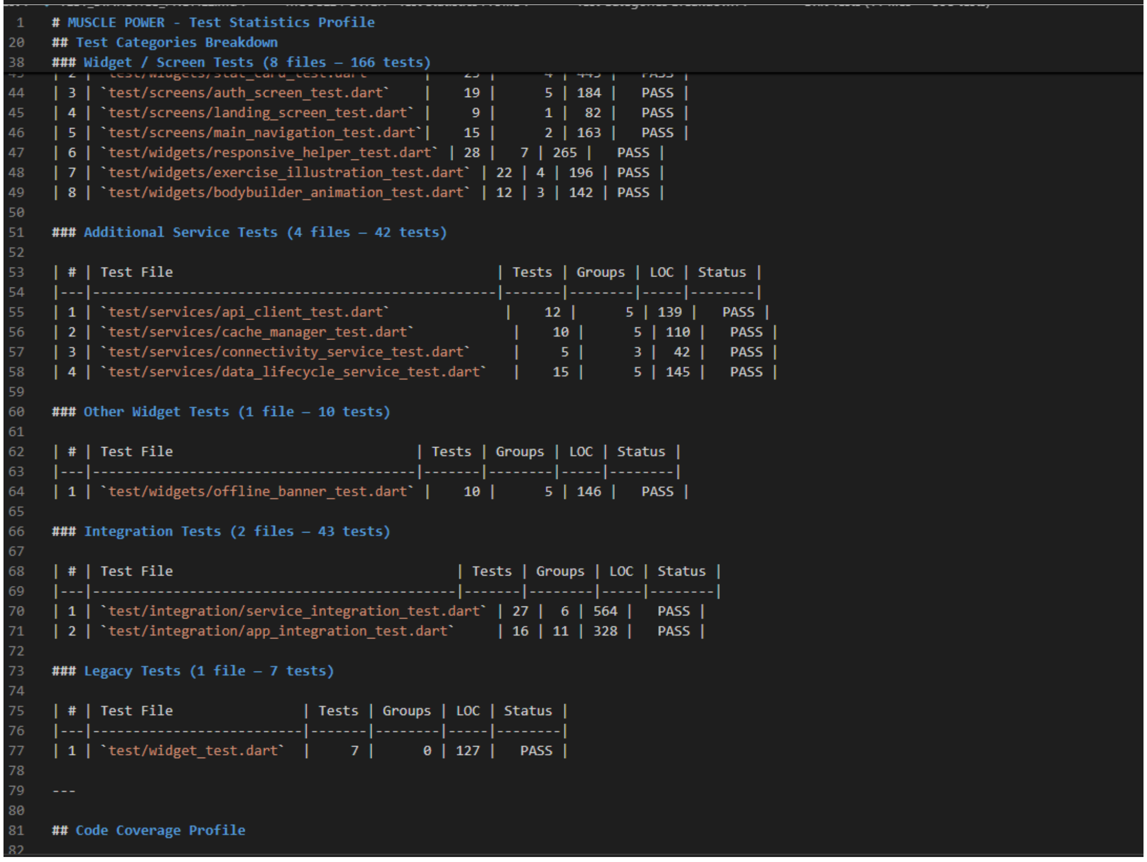
The snapshot below presents the consolidated test report, highlighting coverage metrics, pass rates, and overall system health. This rigorous testing approach reinforces the reliability of the application and ensures that users experience a secure, consistent, and high-quality product.

.

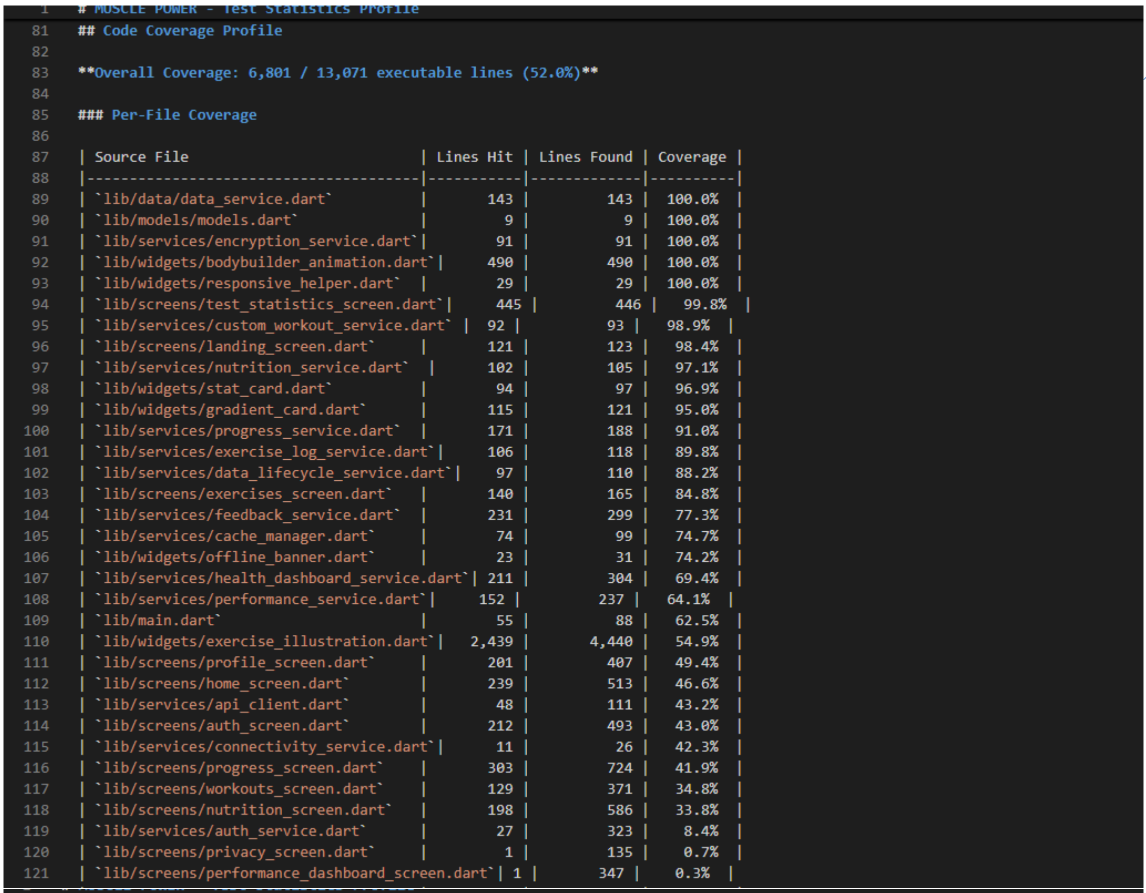


test > TEST_STATISTICS_PROFILE.md > # MUSCLE POWER - Test Statistics Profile > ## Test Groups Detail > ### Exercise Log Service (28 tests)

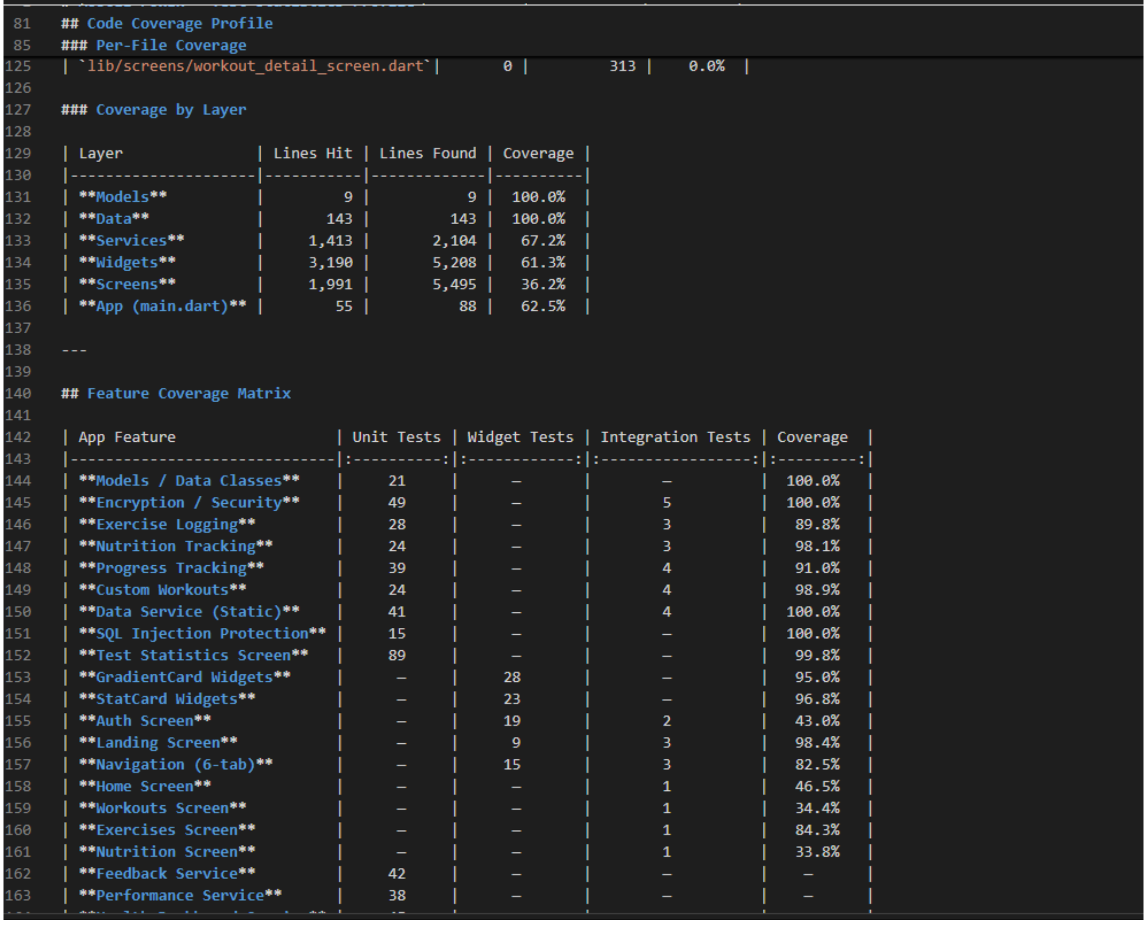
```
# MUSCLE POWER - Test Statistics Profile

## Test Execution Summary

| Metric                 | Value       |
|------------------------|-------------|
| **Total Tests**        | 653         |
| **Passed**             | 653         |
| **Failed**             | 0           |
| **Pass Rate**          | 100%        |
| **Test Files**         | 27          |
| **Test Groups**        | 110+        |
| **Total Test LOC**     | 7,992       |
| **Total Source LOC**   | 30,514      |
| **Test-to-Code Ratio**| 0.26:1      |
| **Execution Time**     | ~36 seconds |

---

## Test Categories Breakdown

### Unit Tests (11 files – 398 tests)

| # | Test File                                | Tests | Groups | LOC | Status |
|---|------------------------------------------|-------|--------|-----|--------|
| 1 | `test/models/models_test.dart`           |    21 |      9 | 495 |   PASS |
| 2 | `test/services/encryption_service_test.dart` | 49 |  9 | 363 |   PASS |
| 3 | `test/services/exercise_log_service_test.dart` | 28 | 2 | 519 |  PASS |
| 4 | `test/services/nutrition_service_test.dart` |  24 |  2 | 261 |   PASS |
| 5 | `test/services/progress_service_test.dart`  |  39 |  3 | 485 |   PASS |
| 6 | `test/services/custom_workout_service_test.dart` | 24 | 2 | 324 | PASS |
| 7 | `test/data/data_service_test.dart`       |    41 |      7 | 374 |   PASS |
| 8 | `test/screens/test_statistics_screen_test.dart` | 89 | 10 | 676 | PASS |
| 9 | `test/services/feedback_service_test.dart` |  42 |  10 | 385 |   PASS |
|10 | `test/services/performance_service_test.dart` | 38 | 9 | 262 |  PASS |
|11 | `test/services/health_dashboard_service_test.dart` | 45 | 11 | 340 | PASS |

### Widget / Screen Tests (8 files – 166 tests)

| # | Test File                                | Tests | Groups | LOC | Status |
|---|------------------------------------------|-------|--------|-----|--------|
| 1 | `test/widgets/gradient_card_test.dart`   |    28 |      5 | 432 |   PASS |
```



```
# MUSCLE POWER - Test Statistics Profile
## Test Categories Breakdown
### Widget / Screen Tests (8 files – 166 tests)
| 3 | `test/screens/auth_screen_test.dart`     |    19 |      5 | 184 |   PASS |
| 4 | `test/screens/landing_screen_test.dart`  |     9 |      1 |  82 |   PASS |
| 5 | `test/screens/main_navigation_test.dart`|    15 |      2 | 163 |   PASS |
| 6 | `test/widgets/responsive_helper_test.dart` | 28 |    7 | 265 |   PASS |
| 7 | `test/widgets/exercise_illustration_test.dart` | 22 | 4 | 196 | PASS |
| 8 | `test/widgets/bodybuilder_animation_test.dart` | 12 | 3 | 142 | PASS |

### Additional Service Tests (4 files – 42 tests)

| # | Test File                                          | Tests | Groups | LOC | Status |
|---|----------------------------------------------------|-------|--------|-----|--------|
| 1 | `test/services/api_client_test.dart`                |    12 |      5 | 139 |   PASS |
| 2 | `test/services/cache_manager_test.dart`              |    10 |      5 | 110 |   PASS |
| 3 | `test/services/connectivity_service_test.dart`       |     5 |      3 |  42 |   PASS |
| 4 | `test/services/data_lifecycle_service_test.dart`     |    15 |      5 | 145 |   PASS |

### Other Widget Tests (1 file – 10 tests)

| # | Test File                                | Tests | Groups | LOC | Status |
|---|------------------------------------------|-------|--------|-----|--------|
| 1 | `test/widgets/offline_banner_test.dart` |    10 |      5 | 146 |   PASS |

### Integration Tests (2 files – 43 tests)

| # | Test File                                     | Tests | Groups | LOC | Status |
|---|-----------------------------------------------|-------|--------|-----|--------|
| 1 | `test/integration/service_integration_test.dart` | 27 |  6 | 564 |   PASS |
| 2 | `test/integration/app_integration_test.dart`     | 16 | 11 | 328 |   PASS |

### Legacy Tests (1 file – 7 tests)

| # | Test File                 | Tests | Groups | LOC | Status |
|---|---------------------------|-------|--------|-----|--------|
| 1 | `test/widget_test.dart`   |     7 |      0 | 127 |   PASS |

---

## Code Coverage Profile
```

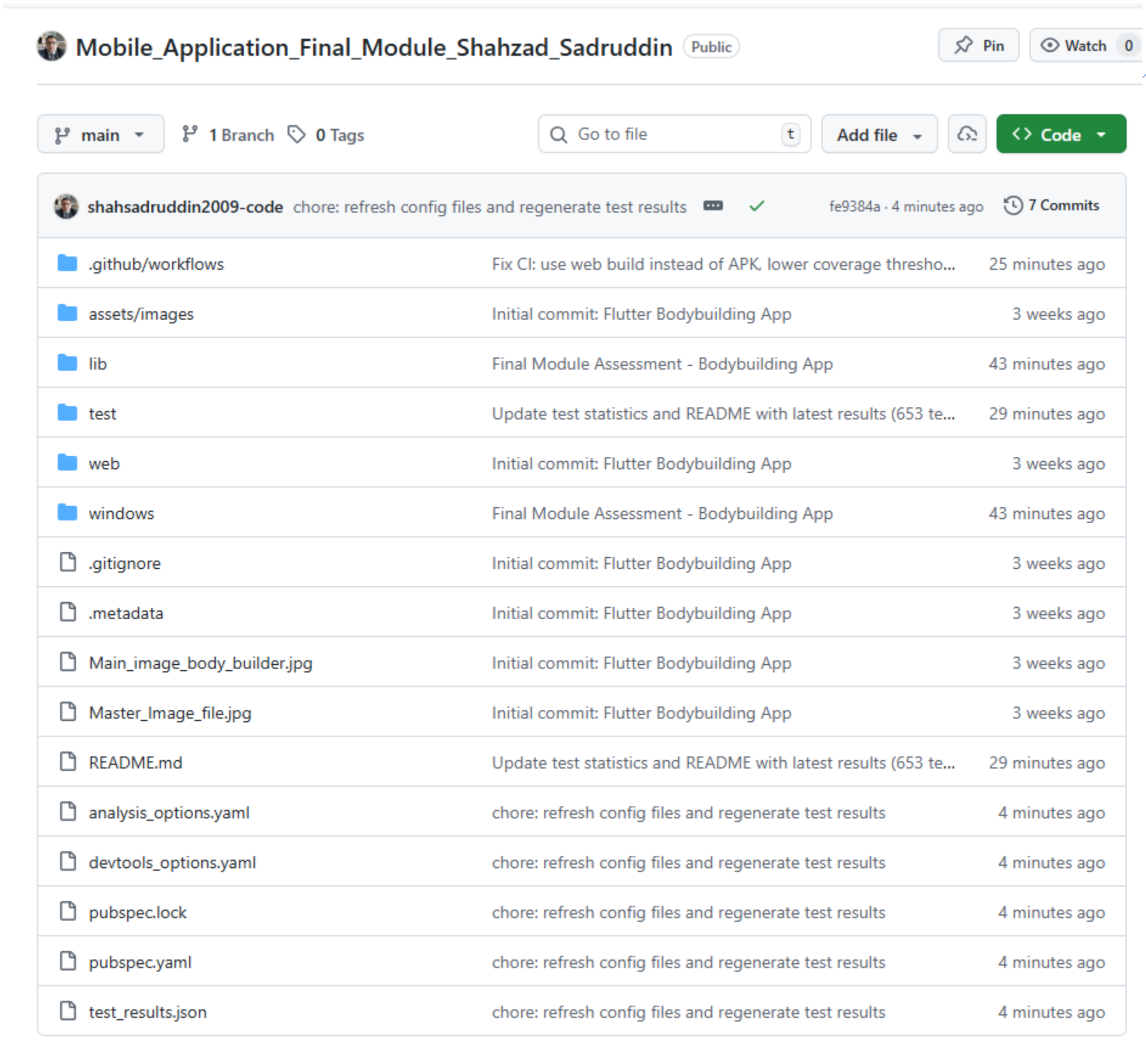


```
# MUSCLE POWER - Test Statistics Profile
## Code Coverage Profile

**Overall Coverage: 6,801 / 13,071 executable lines (52.0%)**

### Per-File Coverage

| Source File                              | Lines Hit | Lines Found | Coverage |
|------------------------------------------|-----------|-------------|----------|
| `lib/data/data_service.dart`             |       143 |         143 |  100.0%  |
| `lib/models/models.dart`                 |         9 |           9 |  100.0%  |
| `lib/services/encryption_service.dart`|         91 |          91 |  100.0%  |
| `lib/widgets/bodybuilder_animation.dart`|       490 |         490 |  100.0%  |
| `lib/widgets/responsive_helper.dart`  |          29 |          29 |  100.0%  |
| `lib/screens/test_statistics_screen.dart`|       445 |         446 |   99.8%  |
| `lib/services/custom_workout_service.dart` |  92 |          93 |   98.9%  |
| `lib/screens/landing_screen.dart`        |       121 |         123 |   98.4%  |
| `lib/services/nutrition_service.dart`  |         102 |         105 |   97.1%  |
| `lib/widgets/stat_card.dart`             |        94 |          97 |   96.9%  |
| `lib/widgets/gradient_card.dart`         |       115 |         121 |   95.0%  |
| `lib/services/progress_service.dart`  |         171 |         188 |   91.0%  |
| `lib/services/exercise_log_service.dart`|       106 |         118 |   89.8%  |
| `lib/services/data_lifecycle_service.dart`|      97 |         110 |   88.2%  |
| `lib/screens/exercises_screen.dart`   |         140 |         165 |   84.8%  |
| `lib/services/feedback_service.dart`  |         231 |         299 |   77.3%  |
| `lib/services/cache_manager.dart`        |        74 |          99 |   74.7%  |
| `lib/widgets/offline_banner.dart`        |        23 |          31 |   74.2%  |
| `lib/services/health_dashboard_service.dart`| 211 |         304 |   69.4%  |
| `lib/services/performance_service.dart`|      152 |         237 |   64.1%  |
| `lib/main.dart`                          |        55 |          88 |   62.5%  |
| `lib/widgets/exercise_illustration.dart`|     2,439 |       4,440 |   54.9%  |
| `lib/screens/profile_screen.dart`        |       201 |         407 |   49.4%  |
| `lib/screens/home_screen.dart`           |       239 |         513 |   46.6%  |
| `lib/services/api_client.dart`           |        48 |         111 |   43.2%  |
| `lib/screens/auth_screen.dart`           |       212 |         493 |   43.0%  |
| `lib/services/connectivity_service.dart`|        11 |          26 |   42.3%  |
| `lib/screens/progress_screen.dart`    |         303 |         724 |   41.9%  |
| `lib/screens/workouts_screen.dart`    |         129 |         371 |   34.8%  |
| `lib/screens/nutrition_screen.dart`   |         198 |         586 |   33.8%  |
| `lib/services/auth_service.dart`         |        27 |         323 |    8.4%  |
| `lib/screens/privacy_screen.dart`        |         1 |         135 |    0.7%  |
| `lib/screens/performance_dashboard_screen.dart`| 1 |       347 |    0.3%  |
```



```
## Code Coverage Profile
### Per-File Coverage
| `lib/screens/workout_detail_screen.dart`|          0 |         313 |    0.0%  |

### Coverage by Layer

| Layer                | Lines Hit | Lines Found | Coverage |
|----------------------|-----------|-------------|----------|
| **Models**           |         9 |           9 |  100.0%  |
| **Data**             |       143 |         143 |  100.0%  |
| **Services**         |     1,413 |       2,104 |   67.2%  |
| **Widgets**          |     3,190 |       5,208 |   61.3%  |
| **Screens**          |     1,991 |       5,495 |   36.2%  |
| **App (main.dart)** |        55 |          88 |   62.5%  |

---

## Feature Coverage Matrix

| App Feature                    | Unit Tests | Widget Tests | Integration Tests | Coverage  |
|--------------------------------|:----------:|:------------:|:-----------------:|:---------:|
| **Models / Data Classes**      |     21     |      —       |         —         |  100.0%   |
| **Encryption / Security**      |     49     |      —       |         5         |  100.0%   |
| **Exercise Logging**           |     28     |      —       |         3         |   89.8%   |
| **Nutrition Tracking**         |     24     |      —       |         3         |   98.1%   |
| **Progress Tracking**          |     39     |      —       |         4         |   91.0%   |
| **Custom Workouts**            |     24     |      —       |         4         |   98.9%   |
| **Data Service (Static)**      |     41     |      —       |         4         |  100.0%   |
| **SQL Injection Protection**   |     15     |      —       |         —         |  100.0%   |
| **Test Statistics Screen**     |     89     |      —       |         —         |   99.8%   |
| **GradientCard Widgets**       |     —      |      28      |         —         |   95.0%   |
| **StatCard Widgets**           |     —      |      23      |         —         |   96.8%   |
| **Auth Screen**                |     —      |      19      |         2         |   43.0%   |
| **Landing Screen**             |     —      |       9      |         3         |   98.4%   |
| **Navigation (6-tab)**         |     —      |      15      |         3         |   82.5%   |
| **Home Screen**                |     —      |      —       |         1         |   46.5%   |
| **Workouts Screen**            |     —      |      —       |         1         |   34.4%   |
| **Exercises Screen**           |     —      |      —       |         1         |   84.3%   |
| **Nutrition Screen**           |     —      |      —       |         1         |   33.8%   |
| **Feedback Service**           |     42     |      —       |         —         |     —     |
| **Performance Service**        |     38     |      —       |         —         |     —     |
```

See below results of the test files successfully completed with no errors.



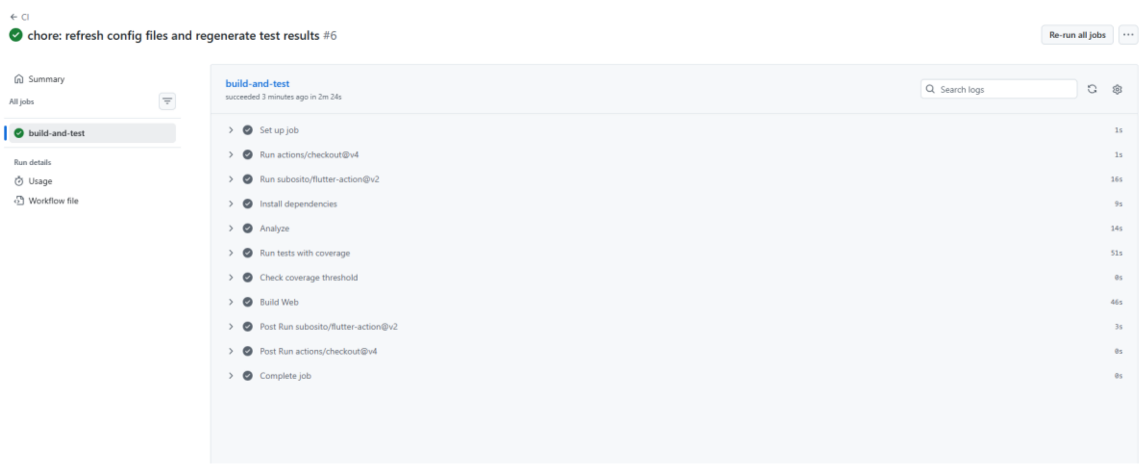
Mobile_Application_Final_Module_Shahzad_Sadruddin Public

main · 1 Branch · 0 Tags

shahsadruddin2009-code chore: refresh config files and regenerate test results · fe9384a · 4 minutes ago · 7 Commits

.github/workflows	Fix CI: use web build instead of APK, lower coverage thresho...	25 minutes ago
assets/images	Initial commit: Flutter Bodybuilding App	3 weeks ago
lib	Final Module Assessment - Bodybuilding App	43 minutes ago
test	Update test statistics and README with latest results (653 te...	29 minutes ago
web	Initial commit: Flutter Bodybuilding App	3 weeks ago
windows	Final Module Assessment - Bodybuilding App	43 minutes ago
.gitignore	Initial commit: Flutter Bodybuilding App	3 weeks ago
.metadata	Initial commit: Flutter Bodybuilding App	3 weeks ago
Main_image_body_builder.jpg	Initial commit: Flutter Bodybuilding App	3 weeks ago
Master_Image_file.jpg	Initial commit: Flutter Bodybuilding App	3 weeks ago
README.md	Update test statistics and README with latest results (653 te...	29 minutes ago
analysis_options.yaml	chore: refresh config files and regenerate test results	4 minutes ago
devtools_options.yaml	chore: refresh config files and regenerate test results	4 minutes ago
pubspec.lock	chore: refresh config files and regenerate test results	4 minutes ago
pubspec.yaml	chore: refresh config files and regenerate test results	4 minutes ago
test_results.json	chore: refresh config files and regenerate test results	4 minutes ago

<u>ci.yaml file</u>



← CI

chore: refresh config files and regenerate test results #6

build-and-test
succeeded 3 minutes ago in 2m 24s

```
Set up job                              1s
Run actions/checkout@v4                 1s
Run subosito/flutter-action@v2         16s
Install dependencies                    9s
Analyze                                14s
Run tests with coverage                51s
Check coverage threshold                0s
Build Web                              46s
Post Run subosito/flutter-action@v2     3s
Post Run actions/checkout@v4            0s
Complete job                            0s
```

MUSCLE POWER

Architecture & Design Diagrams

Module: Mobile Application Development — Final Assessment
Student: Shahzad Sadruddin · 2513806
Date: February 2026
Version: 1.0

Table of Contents

Figure 1 — System Block Diagram

High-level architecture showing the Flutter client, service layer, persistence stores, optional REST API, and third-party integrations. The app follows a **local-first** architecture: all data is written to SharedPreferences / SQLite before any network call. The API layer is an optional sync channel that queues requests while offline.

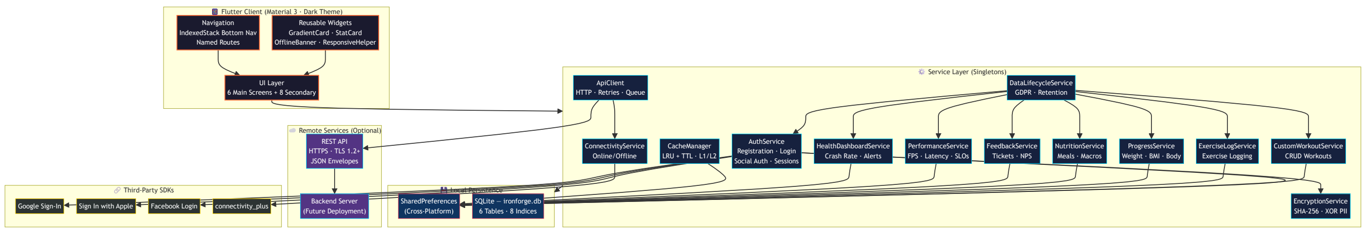


Fig 1. System Block Diagram — local-first architecture with optional cloud sync. Orange borders = UI layer; Cyan borders = service singletons; Red = persistence; Purple = remote; Gold = third-party.

Figure 2 — User Flow / Activity Diagram

Critical user journey from app launch through authentication to core workout tracking. Shows the complete signup → onboarding → core task path along with alternative flows (guest access, social login, session restore).

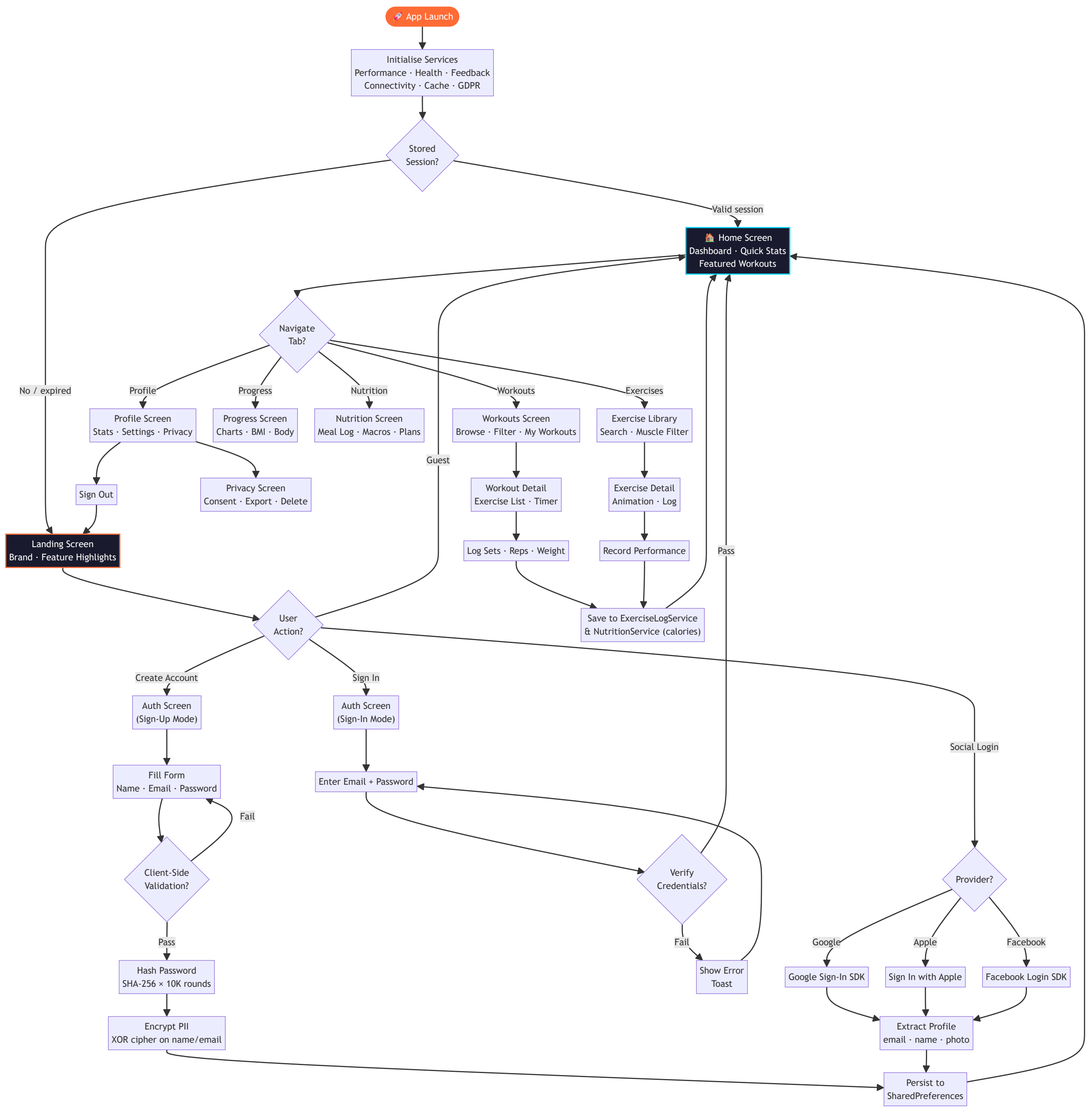
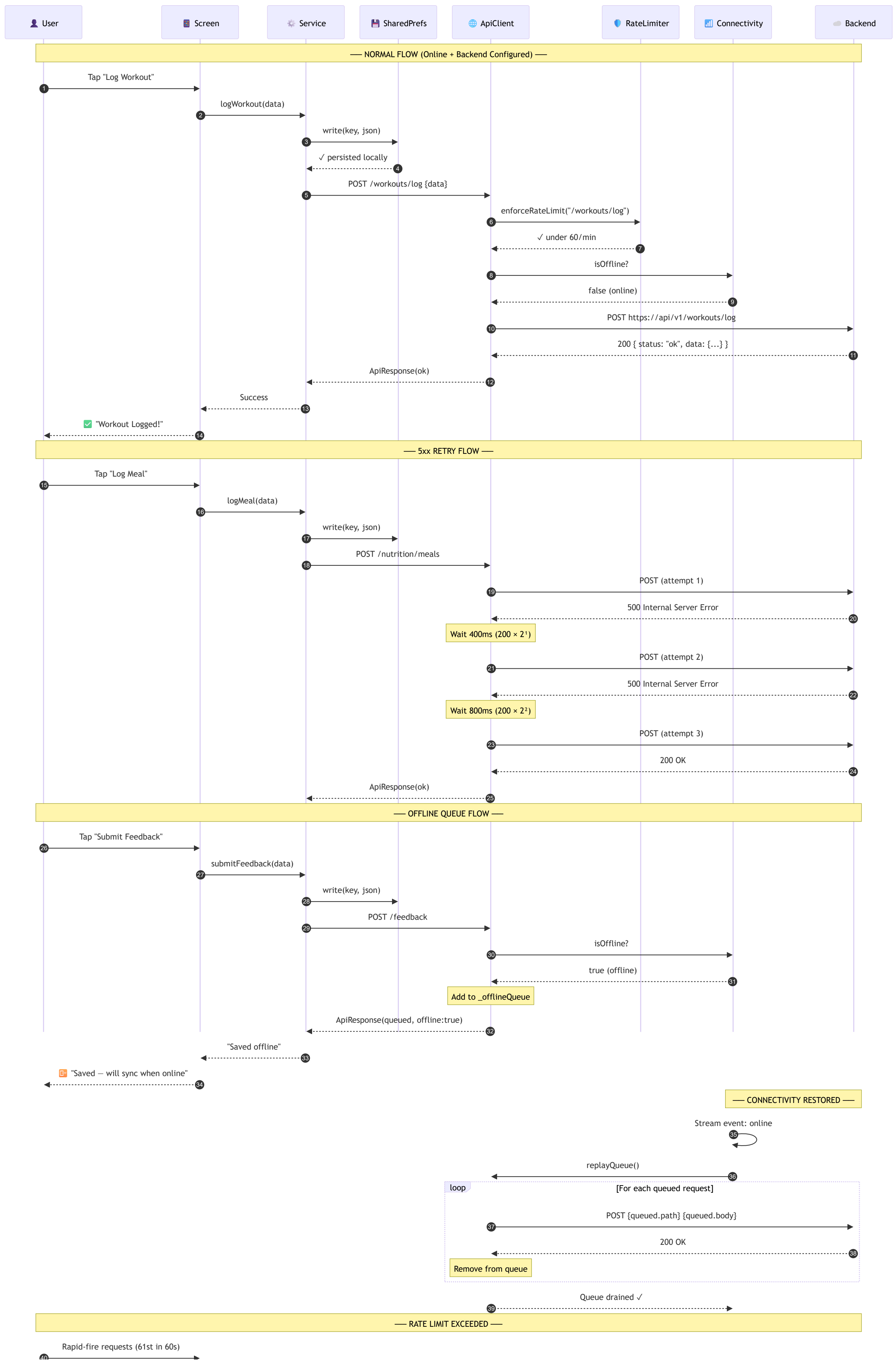


Fig 2. User Flow / Activity Diagram — signup → onboarding → core workout tracking journey.

Figure 3 — Data / Sequence Flow Diagram

Sequence diagram showing the complete network call lifecycle: local-first write, API request with rate limiting, retry on 5xx, offline queueing, and queue replay on connectivity restore.


User
Screen
Service
SharedPrefs
ApiClient
RateLimiter
Connectivity
Backend
── NORMAL FLOW (Online + Backend Configured) ──
1 Tap "Log Workout"
2 logWorkout(data)
3 write(key, json)
4 ✓ persisted locally
5 POST /workouts/log {data}
6 enforceRateLimit("/workouts/log")
7 ✓ under 60/min
8 isOffline?
9 false (online)
10 POST https://api/v1/workouts/log
11 200 { status: "ok", data: {...} }
12 ApiResponse(ok)
13 Success
14 ✅ "Workout Logged!"
── 5xx RETRY FLOW ──
15 Tap "Log Meal"
16 logMeal(data)
17 write(key, json)
18 POST /nutrition/meals
19 POST (attempt 1)
20 500 Internal Server Error
Wait 400ms (200 × 2¹)
21 POST (attempt 2)
22 500 Internal Server Error
Wait 800ms (200 × 2²)
23 POST (attempt 3)
24 200 OK
25 ApiResponse(ok)
── OFFLINE QUEUE FLOW ──
26 Tap "Submit Feedback"
27 submitFeedback(data)
28 write(key, json)
29 POST /feedback
30 isOffline?
31 true (offline)
Add to _offlineQueue
32 ApiResponse(queued, offline:true)
33 "Saved offline"
34 "Saved — will sync when online"
── CONNECTIVITY RESTORED ──
35 Stream event: online
36 replayQueue()
loop
[For each queued request]
37 POST {queued.path} {queued.body}
38 200 OK
Remove from queue
39 Queue drained ✓
── RATE LIMIT EXCEEDED ──
40 Rapid-fire requests (61st in 60s)

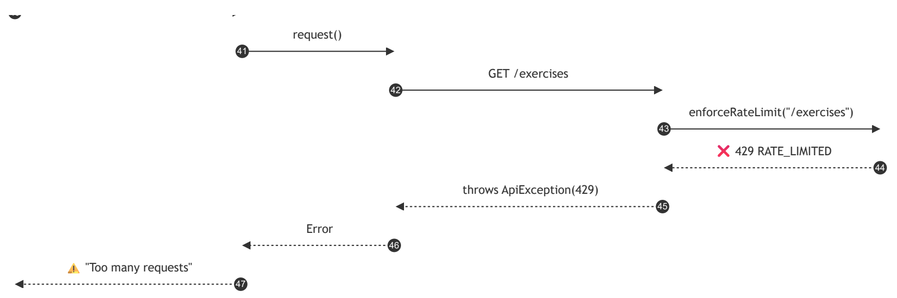
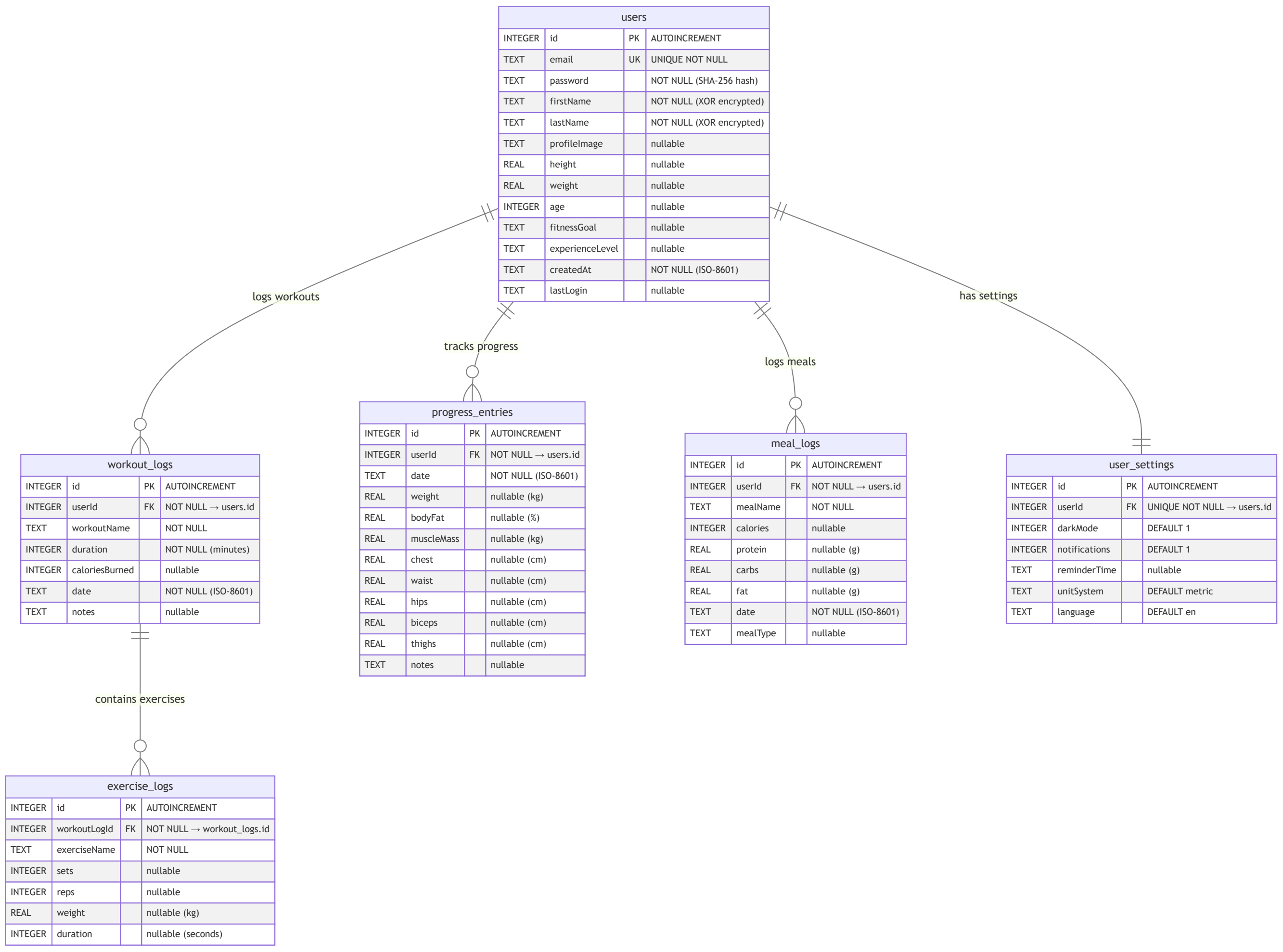


Fig 3. Data / Sequence Flow — network calls showing local-first write, retry with exponential back-off, offline queueing, queue replay, and rate limiting.

Figure 4 — Entity-Relationship (ER) Diagram

Database schema for the SQLite `ironforge.db` database (mobile/desktop). Shows all 6 tables with columns, types, primary keys, foreign key relationships, and performance indices. Cross-platform data also persists in SharedPreferences (shown as a supplementary store).



users			
INTEGER	id	PK	AUTOINCREMENT
TEXT	email	UK	UNIQUE NOT NULL
TEXT	password		NOT NULL (SHA-256 hash)
TEXT	firstName		NOT NULL (XOR encrypted)
TEXT	lastName		NOT NULL (XOR encrypted)
TEXT	profileImage		nullable
REAL	height		nullable
REAL	weight		nullable
INTEGER	age		nullable
TEXT	fitnessGoal		nullable
TEXT	experienceLevel		nullable
TEXT	createdAt		NOT NULL (ISO-8601)
TEXT	lastLogin		nullable

workout_logs			
INTEGER	id	PK	AUTOINCREMENT
INTEGER	userId	FK	NOT NULL → users.id
TEXT	workoutName		NOT NULL
INTEGER	duration		NOT NULL (minutes)
INTEGER	caloriesBurned		nullable
TEXT	date		NOT NULL (ISO-8601)
TEXT	notes		nullable

progress_entries			
INTEGER	id	PK	AUTOINCREMENT
INTEGER	userId	FK	NOT NULL → users.id
TEXT	date		NOT NULL (ISO-8601)
REAL	weight		nullable (kg)
REAL	bodyFat		nullable (%)
REAL	muscleMass		nullable (kg)
REAL	chest		nullable (cm)
REAL	waist		nullable (cm)
REAL	hips		nullable (cm)
REAL	biceps		nullable (cm)
REAL	thighs		nullable (cm)
TEXT	notes		nullable

meal_logs			
INTEGER	id	PK	AUTOINCREMENT
INTEGER	userId	FK	NOT NULL → users.id
TEXT	mealName		NOT NULL
INTEGER	calories		nullable
REAL	protein		nullable (g)
REAL	carbs		nullable (g)
REAL	fat		nullable (g)
TEXT	date		NOT NULL (ISO-8601)
TEXT	mealType		nullable

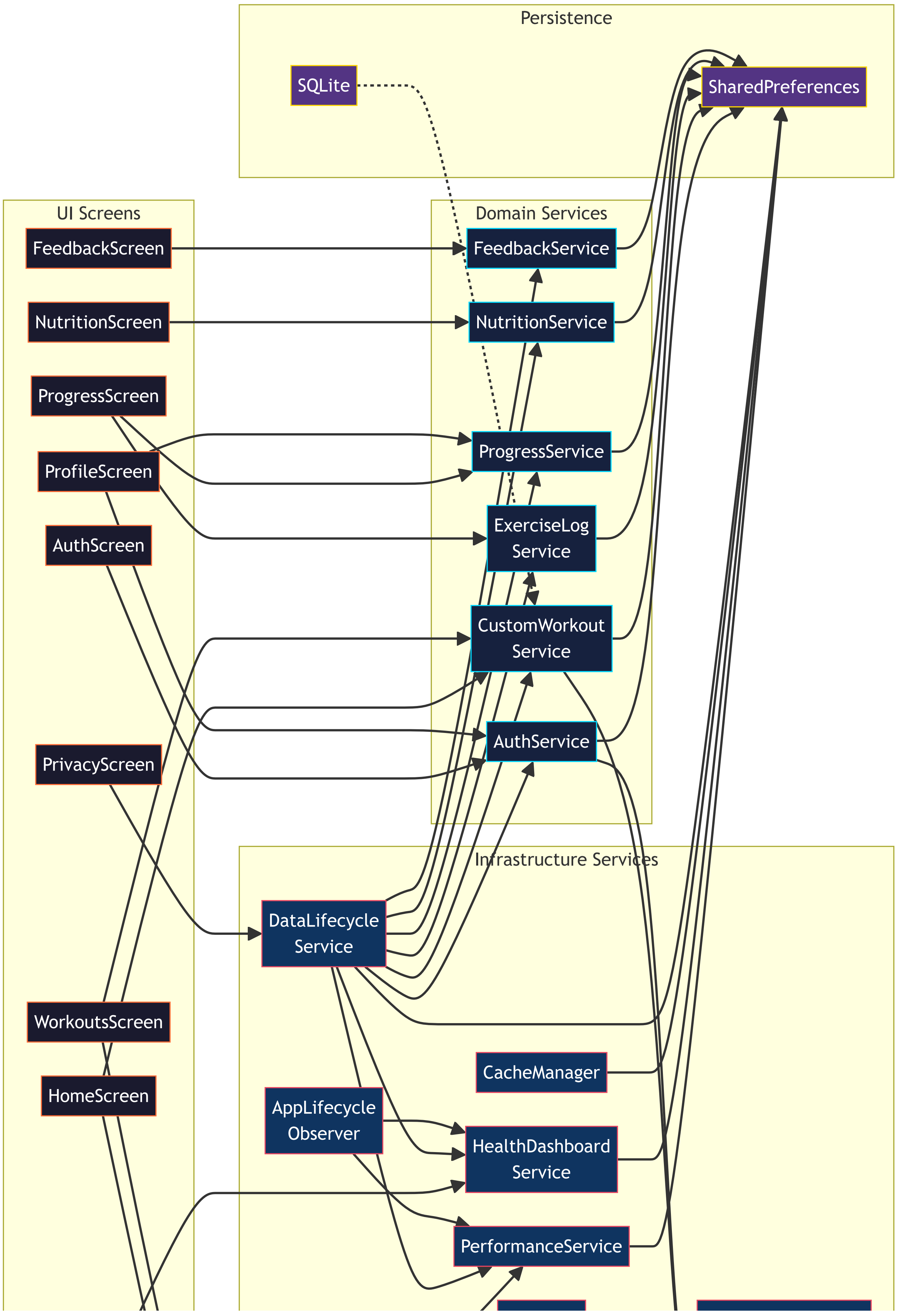
user_settings			
INTEGER	id	PK	AUTOINCREMENT
INTEGER	userId	FK	UNIQUE NOT NULL → users.id
INTEGER	darkMode		DEFAULT 1
INTEGER	notifications		DEFAULT 1
TEXT	reminderTime		nullable
TEXT	unitSystem		DEFAULT metric
TEXT	language		DEFAULT en

exercise_logs			
INTEGER	id	PK	AUTOINCREMENT
INTEGER	workoutLogId	FK	NOT NULL → workout_logs.id
TEXT	exerciseName		NOT NULL
INTEGER	sets		nullable
INTEGER	reps		nullable
REAL	weight		nullable (kg)
INTEGER	duration		nullable (seconds)

Fig 4. ER Diagram — SQLite `ironforge.db` schema. PK = Primary Key, FK = Foreign Key, UK = Unique. Indices on: userId (all child tables), date (workout_logs, progress_entries, meal_logs), workoutLogId (exercise_logs). SharedPreferences stores cross-platform supplementary data.

Figure 5 — Component / Service Dependency Diagram

Internal architecture showing how the 15 service singletons depend on each other and on persistence layers. Arrows point from dependent → dependency. The DataLifecycleService (GDPR) depends on the most services since it must purge/export data from all stores.


Persistence
SQLite
SharedPreferences
UI Screens
FeedbackScreen
NutritionScreen
ProgressScreen
ProfileScreen
AuthScreen
PrivacyScreen
WorkoutsScreen
HomeScreen
Domain Services
FeedbackService
NutritionService
ProgressService
ExerciseLog Service
CustomWorkout Service
AuthService
Infrastructure Services
DataLifecycle Service
CacheManager
AppLifecycle Observer
HealthDashboard Service
PerformanceService

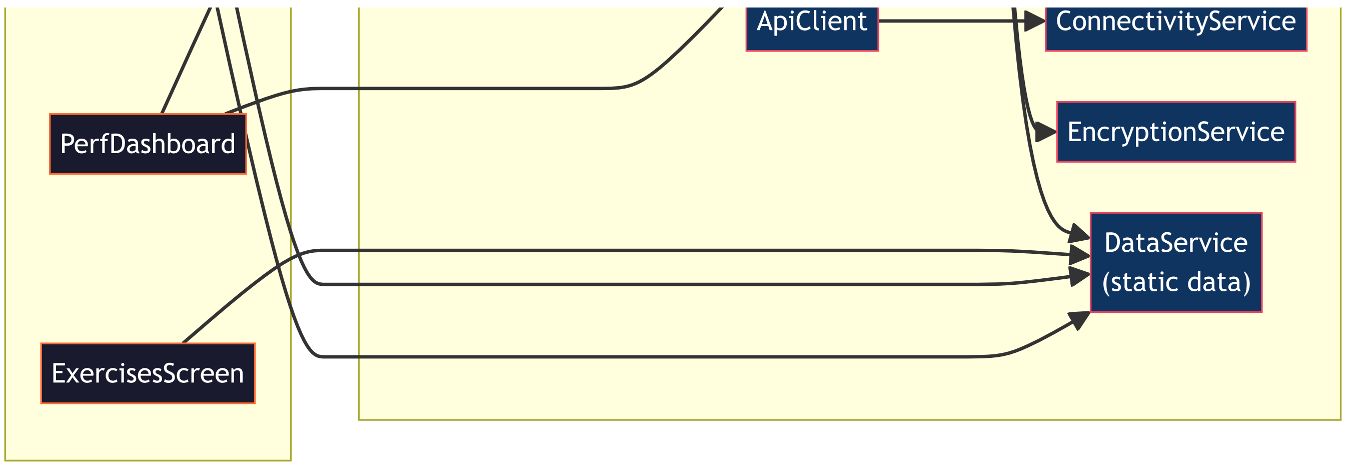


Fig 5. Component Dependency Diagram — service singleton relationships. Orange = UI screens; Cyan = domain services; Red = infrastructure; Gold = persistence.

Figure 6 — State Machine: Offline Queue Lifecycle

State machine diagram showing how an API request transitions through the offline queue system. Covers the happy path (direct send), offline queueing, connectivity-triggered replay, retry on failure, and terminal states.

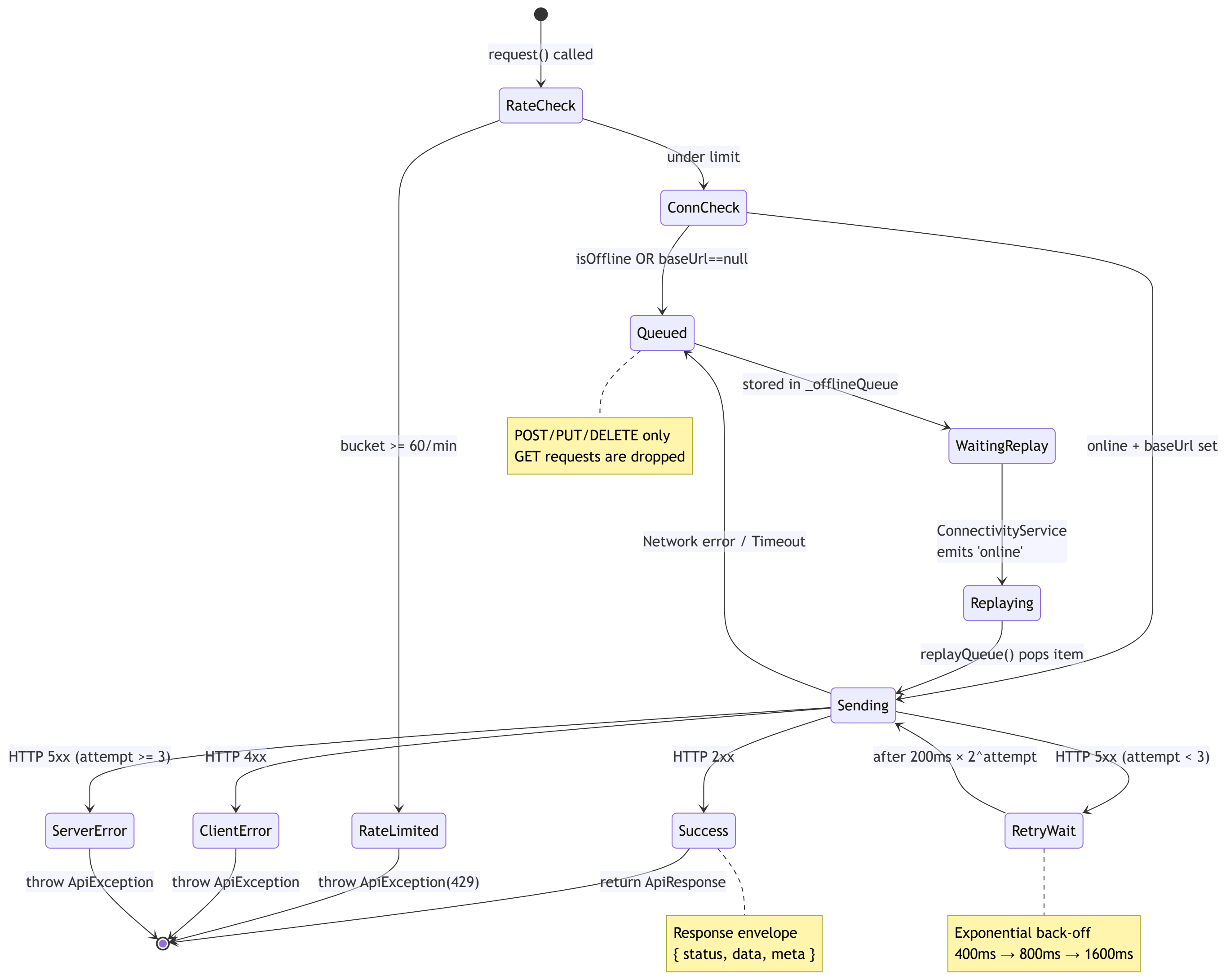


Fig 6. State Machine — API request lifecycle with offline queue, retry, and rate limiting.