Finance Tracker

Take-Home Assessment — Technical Report

Flutter · Node.js · Riverpod · GoRouter · Dio

1. Project Overview

Finance Tracker is a full-stack personal finance application consisting of a **Flutter mobile frontend** and a **Node.js/Express REST API**. Users can log income and expense transactions, view a live summary of their financial position, and drill into individual transactions to review or delete them.

Platform	Android (Flutter 3.41 · Dart 3.3)
Backend	Node.js 18 · Express 4 · In-memory store
State Mgmt	flutter_riverpod 2.5.1 — StateNotifier + FutureProvider
Navigation	go_router 14.x — declarative URL-based routing
HTTP Client	dio 5.4.x — structured error handling via DioException
Formatting	intl 0.19 — dates and currency
Architecture	Models → Services → Providers → Screens (clean separation)

2. How to Start the App

Step 1 — Start the Backend

Open a terminal and run:

```
cd C:\Users\LENOVO\Desktop\finance-tracker\backend
node server.js
# Output: ■  Finance Tracker API  →  http://localhost:3000
```

The backend runs on port 3000. Data resets on restart (in-memory by design).

Step 2 — Connect your Infinix Phone

Plug in via USB (USB debugging must be enabled), then run:

```
adb reverse tcp:3000 tcp:3000
# Maps phone's localhost:3000 → your PC's port 3000
adb devices
# Should show: 1164870488002542  device  (Infinix X6525)
```

Step 3 — Run the Flutter App

```
export PATH="$PATH:/c/Users/LENOVO/flutter/bin"
export GRADLE_OPTS="-Dorg.gradle.offline=true"
cd C:\Users\LENOVO\Desktop\finance-tracker\finance_tracker
flutter run -d 1164870488002542
```

On first run, Gradle downloads build dependencies (~2 min). Subsequent runs are much faster. The app installs directly on the phone.

Hot Reload during development

r	Hot reload — apply Dart code changes instantly, state preserved
R	Hot restart — full restart, state reset
q	Quit and stop the app on device
d	Detach — leave app running, exit flutter run

3. Features Walkthrough

Screen 1 — Home / Dashboard (/)

- Summary card at top: Net Balance (large), Total Income, Total Expenses
- Deep green gradient card — colour convention for finance apps
- Transaction list below, sorted newest date first
- Each list item shows: direction icon, title, category, date, and amount (green/red)
- Filter chips: All · Income · Expense — instant local filtering, no API call
- Pull-to-refresh: swipe down to reload from API
- Loading spinner, error state with Retry button, empty state message
- Floating Action Button (+) navigates to Add Transaction

Screen 2 — Add Transaction (/add)

- Animated type toggle: Income (green) / Expense (red) with border animation
- Title field — required, validated before submit
- Amount field — numeric keyboard, max 2 decimal places enforced
- Category dropdown — 12 options: Food & Dining, Salary, Transport, Housing, etc.
- Date picker — taps to open calendar, defaults to today
- Note field — optional free-text
- Submit button disabled while request is in-flight (prevents double submit)
- On success: navigates back to Home, new transaction appears at top of list
- On failure: SnackBar error message shown

Screen 3 — Transaction Detail (/transactions/:id)

- Large amount display with gradient header (green=income, red=expense)
- Detail card: title, category, date (formatted), note (if present)
- Delete button with confirmation dialog ('Are you sure?')
- On confirm: DELETE request, navigates back to Home, item removed from list
- If transaction not found: graceful 'Not found' screen shown

4. Architecture & Design Decisions

State Management — Riverpod

Riverpod was chosen over Provider and Bloc. The key reasons:

- **Type safety:** Providers are accessed by variable reference, not by type cast through context. No runtime failures from missing providers.
- **No BuildContext dependency:** Providers are Dart objects. They can be read from anywhere — services, other providers, tests — without needing a widget tree.
- **Derived state:** filteredTransactionsProvider watches the transaction list and the filter chip. It recomputes instantly in memory — zero extra API calls when the user taps a filter chip.
- **FutureProvider for summary:** Loading/error/data states handled automatically. ref.invalidate(summaryProvider) after any mutation triggers a silent background re-fetch.

The 5 providers and their roles:

apiServiceProvider	Provider	Singleton Dio wrapper
filterProvider	StateProvider	Current filter: all/income/expense
transactionNotifierProvider	StateNotifierProvider<…State>	Transaction list + loading + error
filteredTransactionsProvider	Provider>	Derived filtered list (no API call)
summaryProvider	FutureProvider	Totals from /summary endpoint

Navigation — GoRouter

GoRouter is the Flutter team's officially recommended router. URL-based routing makes navigation intent explicit:

/	HomeScreen — transaction list + summary
/add	AddTransactionScreen — create form
/transactions/:id	TransactionDetailScreen — view + delete

context.push() preserves back-stack. context.go('/') replaces the stack (used after deletion so back button doesn't return to a deleted item).

HTTP Client — Dio

Dio's DioException carries a structured type enum and the full response body. The _handleError() method in ApiService maps every failure mode to a user-readable ApiException string:

- 400 response with errors array → joins validation messages
- 404 response with error string → extracts the message
- Connection timeout → 'Connection timed out. Check your network.'
- Connection error → 'Cannot reach the server. Make sure the backend is running.'

5. REST API Reference

Base URL

Physical device (USB + ADB reverse): http://localhost:3000

Android emulator: http://10.0.2.2:3000

Endpoints

GET	/transactions	All transactions, sorted newest date first
POST	/transactions	Create a transaction (201 Created)
GET	/transactions/:id	Single transaction by UUID
PATCH	/transactions/:id	Update any field(s) — partial update
DELETE	/transactions/:id	Delete transaction (204 No Content)
GET	/summary	{ totalIncome, totalExpenses, netBalance }

Transaction Schema

id	string — UUID v4, auto-generated
title	string — required, non-empty
amount	number — required, positive (always > 0)
type	string — 'income' | 'expense'
category	string — required, non-empty
date	string — ISO 8601 datetime
note	string | null — optional
createdAt	string — ISO 8601, auto-set, immutable

Validation Rules

- POST: all fields except note are required
- PATCH: only provided fields are validated (partial=true mode)
- PATCH: id and createdAt are protected — cannot be overwritten
- 400 response: { errors: ['field: reason', ...] }
- 404 response: { error: "Transaction 'uuid' not found" }
- Summary: totals rounded to 2 decimal places to prevent floating-point drift

6. Key Design Decisions — Discussion Guide

This section prepares you for the take-home discussion call. Each decision has a rationale and the trade-off acknowledged.

Why Riverpod over BLoC?

BLoC is excellent for large teams and complex state but requires significant boilerplate (Event class, State class, Bloc class, StreamBuilder). For this scope, Riverpod's StateNotifier achieves the same separation with half the code. I'd consider BLoC if the team was larger or the domain logic became significantly more complex.

Why is amount always positive?

Storing amount as a positive number and using the type field for direction avoids a whole class of bugs: negative number validation, display formatting (when is a minus sign needed?), and arithmetic (should balances add or subtract?). The sign is determined at render time by checking type === 'income'.

Why does the filter not call the API?

filteredTransactionsProvider is a derived Riverpod provider that watches the full transaction list in memory and the filter string. When the user taps Income or Expense, Riverpod recomputes the filtered list instantly — no network latency, no loading spinner. This is only possible because we loaded the full list upfront. For very large datasets, server-side filtering via query params would be the right trade-off.

Why is summary fetched from the API rather than computed locally?

The backend is the source of truth. Computing totals locally risks drift if the app ever has multiple clients or if a background sync updates data. The FutureProvider pattern makes this easy: ref.invalidate(summaryProvider) after any mutation triggers a silent re-fetch. The cost is one extra network round-trip on add/delete, which is acceptable.

Why GoRouter over Navigator 2.0 directly?

Navigator 2.0 is powerful but notoriously verbose. GoRouter wraps it with a declarative route table that maps URL paths to screens — the same mental model as web routing. It's the Flutter team's recommendation and handles edge cases (deep links, back button behaviour on Android) correctly out of the box.

In-memory backend — acceptable?

Yes, for this assessment. The spec asked for a working REST API, not persistence. The architecture is correct — swapping the in-memory array for a SQLite call would be a one-file change to server.js. I'd use better-sqlite3: synchronous API, no daemon, single file, zero configuration.

7. Trade-offs & What I'd Add With More Time

Persistence	In-memory (resets on restart)	SQLite via better-sqlite3
Edit screen	Not implemented (PATCH API exists)	Add /edit/:id screen using existing PATCH endpoint
Charts	Not implemented	fl_chart: category pie + monthly bar chart
Search	Not implemented	Full-text search on title + date range filter
Offline	Not implemented	isar or hive local cache with sync
Tests	Widget test boilerplate only	Provider unit tests, widget tests, API integration tests
Auth	None	JWT auth — multiple users, private ledgers
Pagination	All transactions loaded at once	Cursor-based pagination on GET /transactions
Categories	Hardcoded in Flutter	API-driven category list from backend
CI/CD	None	GitHub Actions: lint → test → build APK on PR

8. Setup Fixes Applied (Physical Device)

The project was originally scaffolded for an Android emulator. The following changes were made to run on the physical Infinix X6525 (Android 13):

Flutter SDK	Cloned from GitHub to C:\Users\LENOVO\flutter (git clone --depth=1 --branch stable)
Android licenses	Accepted via sdkmanager --licenses
Gradle version	gradle-wrapper.properties → gradle-9.0.0-bin.zip (locally cached, offline-compatible)
NDK version	app/build.gradle.kts: ndkVersion = "28.0.12433566" (avoids malformed 28.2 install)
GRADLE_OPTS	GRADLE_OPTS=-Dorg.gradle.offline=true forces use of local Maven/Gradle cache
API base URL	api_service.dart: 10.0.2.2:3000 → localhost:3000 (works with adb reverse)
ADB reverse tunnel	adb reverse tcp:3000 tcp:3000 — phone's localhost:3000 maps to PC port 3000