16 · Networking

Flutter Practice Notes

Contents

16 · Networking

Introduction

Networking connects your app to backends: REST, GraphQL, WebSockets, gRPC. This module covers HTTP fundamentals, a production REST client (`dio` + interceptors + typed errors), and the other protocols — always **wrapped behind repositories** with retry/timeout/cancellation and DTO↔entity mapping.

Why this module exists

Networking is where most runtime failures live (timeouts, 4xx/5xx, offline, parsing). A disciplined client — typed failures, interceptors for auth/logging/retry, and repository boundaries — is what separates a robust app from one that crashes on a flaky connection.

Module map

#	File	Topic	Level
1	01_http_fundamentals.md	HTTP basics, `http` vs `dio`	🟢
2	02_rest_client_and_interceptors.md	`dio` client, interceptors, typed errors, retry/timeout/cancel, DTO mapping	🔴
3	03_graphql.md	Queries/mutations/subscriptions, caching	🔵
4	04_websockets_and_realtime.md	WebSockets, SSE, Socket.IO	🔵
5	05_grpc.md	Protobuf, gRPC, streaming	🔵

> **Cross-references:** Serialization: 02 · json_and_serialization. Repository/DTO boundary: 05 · repository. Typed failures/ `Result` : Module 38. Caching/offline: 15 · caching_strategies, Module 19. Auth tokens/refresh: Module 17. Isolates for heavy parsing: 02 · isolates.

Prerequisites

02 Advanced Dart (async/streams/isolates/JSON), 05 · repository/DI, 14 DI.

What you'll be able to do after this module

- Make HTTP requests and reason about methods/status/headers.
- Build a production `dio` client with interceptors, typed errors, retry/timeout/cancellation.
- Map DTOs↔entities at the repository boundary.
- Choose and use GraphQL, WebSockets/real-time, and gRPC appropriately.

Capstone

Robust API layer: A `dio`-based client with auth + logging + retry interceptors, timeouts and cancellation, errors converted to a typed `Result`/failure, DTO→entity mapping, and a repository the app depends on — testable with a mocked client.

Summary

Pick the protocol per need (REST/GraphQL/WS/gRPC), build a disciplined client (interceptors, typed errors, resilience), and hide it behind repositories that return domain entities. Networking done right is resilient, testable, and invisible to the UI.

HTTP Fundamentals (`http` vs `dio`)

> HTTP is a request/response protocol: you send a **method** (GET/POST/...) to a URL with headers/body and get back a **status code** + headers + body; Flutter's `http` package is minimal, while `dio` adds interceptors, timeouts, cancellation, and more for production apps.

Introduction

Before frameworks, understand HTTP: methods, status codes, headers, and bodies. Then the two main Dart clients: `http` (simple, official, minimal) and `dio` (feature-rich: interceptors, timeouts, cancel tokens, transformers). This file covers both and when to use which.

Why this concept exists

Every REST call is HTTP under the hood; misreading status codes or ignoring headers causes bugs. And choosing the client matters: `http` is fine for trivial calls, but production apps need `dio`'s cross-cutting features (auth injection, retry, logging) — better to know both.

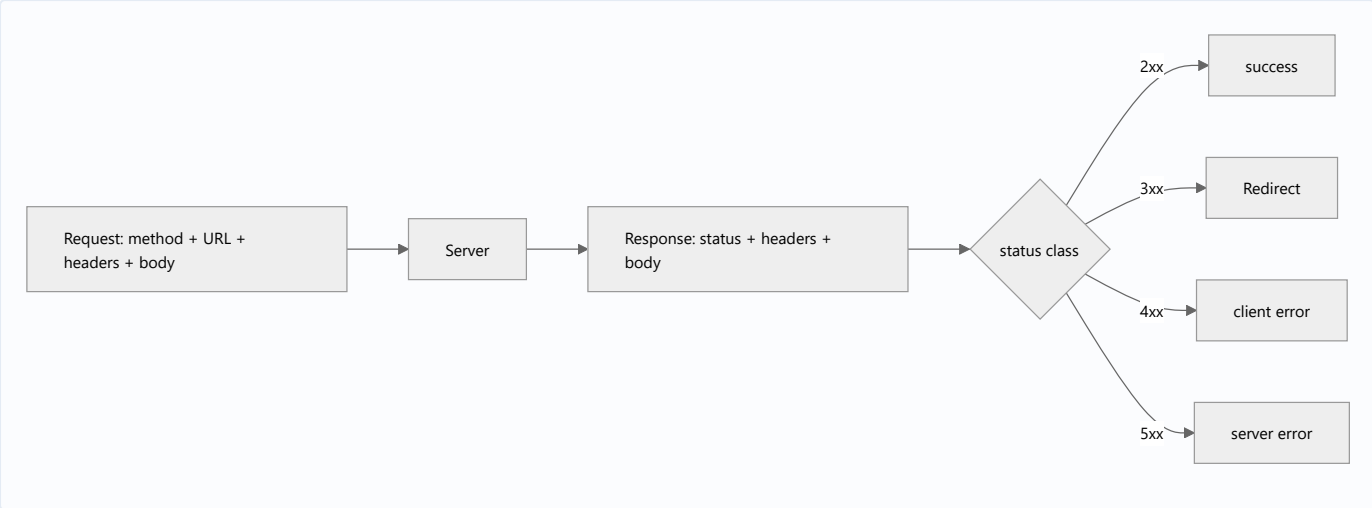
Real-world analogy

HTTP is **ordering by mail**: you send a labeled request (method + address + contents), and get a reply with a status stamp ("delivered 200", "not found 404", "office closed 503"). `http` is a basic postbox; `dio` is a mailroom with tracking, auto-retry, and a clerk stamping every outgoing letter (interceptors).

Problem Statement

Fetch a list of products (GET) and create one (POST with JSON), reading status codes and headers correctly, and decide between `http` and `dio`. You'll do both requests and compare clients.

Internal Working



- **Methods**: GET (read), POST (create), PUT/PATCH (update), DELETE (remove) — plus semantics (idempotency: GET/PUT/DELETE idempotent, POST not).
- **Status classes**: 2xx success, 3xx redirect, 4xx client error (400 bad request, 401 unauthorized, 403 forbidden, 404 not found, 429 too many requests), 5xx server error (500, 502, 503).
- **Headers**: `Content-Type` / `Accept` (usually `application/json`), `Authorization: Bearer <token>`, caching headers (`ETag`, `Cache-Control`).
- **Body**: request payload (JSON) and response body (JSON to parse — 02 · json).
- **Clients**:
  - `http`: `http.get/post(...)`; returns a `Response` (`statusCode`, `body`); you check status + decode manually. Minimal.
  - `dio`: `Dio().get/post(...)`; auto-decodes JSON, throws `DioException` on non-2xx (configurable), supports interceptors/timeouts/cancel tokens/base options.

Memory Representation

Response bodies load into memory (stream large ones); decoded JSON is a heap tree — parse big payloads off-isolate (02 · isolates).

Compiler Behavior

Not applicable.

Runtime Behavior

Requests are async (`Future`). `http` returns a `Response` regardless of status (you check `statusCode`); `dio` throws `DioException` on non-2xx by default. Timeouts/cancellation are manual in `http`, built-in in `dio`.

Flutter Engine Behavior

Networking uses platform sockets via the embedder; on the UI isolate for orchestration, heavy parsing should be offloaded (10 · threading_model).

Dart VM Behavior

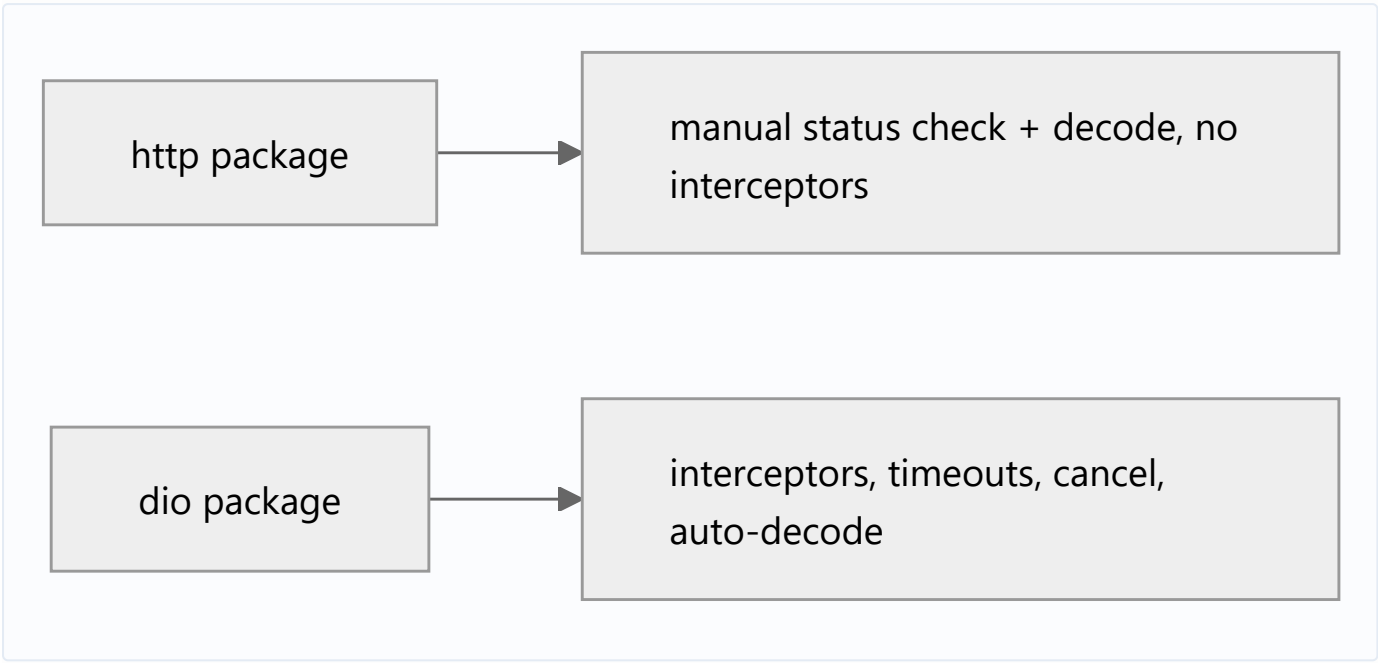
Not applicable beyond async on the event loop (02 · event_loop).

Examples

```yaml
# pubspec.yaml
dependencies:
  http: ^1.2.0      # minimal
  dio: ^5.4.0       # production-featured
```

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'package:dio/dio.dart';

// --- http (minimal): check status + decode manually ---
Future<List<dynamic>> fetchWithHttp() async {
  final res = await http.get(
    Uri.parse('https://api.example.com/products'),
    headers: {'Accept': 'application/json'},
  );
  if (res.statusCode == 200) {                 // must check status yourself
    return jsonDecode(res.body) as List<dynamic>;
  }
  throw Exception('HTTP ${res.statusCode}');   // handle non-2xx
}

// --- dio (production): base options, auto-decode, throws on non-2xx ---
final dio = Dio(BaseOptions(
  baseUrl: 'https://api.example.com',
  connectTimeout: const Duration(seconds: 10),
  receiveTimeout: const Duration(seconds: 10),
  headers: {'Accept': 'application/json'},
));

Future<List<dynamic>> fetchWithDio() async {
  final res = await dio.get('/products');       // decodes JSON, throws DioException on non-2xx
  return res.data as List<dynamic>;
}

Future<void> createProduct(Map<String, dynamic> body) async {
  await dio.post('/products', data: body);      // JSON-encodes body automatically
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Ignoring status codes (`http`)	Treats errors as success	Check `statusCode` ; handle non-2xx
No timeouts	Hangs on bad network	Set connect/receive timeouts (`dio` built-in)
Parsing huge JSON on UI isolate	Jank	Offload to `Isolate.run` / `compute`
Hardcoding base URL/headers everywhere	Duplication	Central `BaseOptions` /client
Leaking raw `Response` / `DioException` to UI	Coupling	Map to domain types at the repository

Best Practices

- Understand and handle **status codes** explicitly; don't assume 2xx.
- Set **timeouts** always; add cancellation for abandoned requests.
- Centralize config (base URL, headers) in one client (`BaseOptions`).
- Use `http` for trivial/one-off; `dio` for real apps (interceptors/timeouts/cancel).
- Offload large parsing off the UI isolate; map responses to domain models at the repo boundary.

Performance

Latency dominates; parallelize independent calls (`Future.wait` — 02 · async_futures), set timeouts, and offload heavy JSON parsing (Module 21).

Advantages / Disadvantages

- **+** `http`**:** simple, official, tiny. **+** `dio`**:** interceptors, timeouts, cancellation, transformers, base options.
- **−** `http`**:** manual status/decoding, no interceptors/timeouts. **−** `dio`**:** heavier, more API to learn.

Interview Questions

1. 🟢 **What are the main HTTP methods and their semantics?** — GET (read), POST (create, non-idempotent), PUT/PATCH (update), DELETE (remove); GET/PUT/DELETE are idempotent.
2. 🟢 **What do status code classes mean?** — 2xx success, 3xx redirect, 4xx client error, 5xx server error.
3. 🟡 `http` **vs** `dio`**?** — `http` is minimal (manual status check/decode); `dio` adds interceptors, timeouts, cancellation, auto-decode, base options — better for production.
4. 🟡 **How does error handling differ between them?** — `http` returns a `Response` for any status (you check `statusCode`); `dio` throws `DioException` on non-2xx by default.
5. 🟡 **Why set timeouts?** — To avoid hanging on slow/dead connections; `dio` has connect/receive timeouts built in.
6. 🔴 **Why offload JSON parsing?** — Large decodes are CPU-bound and block the UI isolate → jank; use `compute` / `Isolate.run`.
7. 🔴 **Where should raw responses be converted to domain models?** — At the repository boundary (DTO→entity), so the UI never sees `Response` / `DioException` (05 · repository).

Senior Engineer Tips

- Default to `dio` for anything real — you'll want interceptors (auth/logging/retry) and timeouts almost immediately.
- Never let transport types (`Response`, `DioException`) escape the data layer; map to typed failures (Module 38).
- Set sane timeouts and parallelize independent requests; treat the network as unreliable by default.

Architect Perspective

The HTTP client is the app's edge to the outside world. A centralized, timeout-aware, interceptor-driven client behind repositories (returning domain entities/typed failures) makes networking resilient, testable, and swappable — the foundation for the REST-client design, auth, caching, and offline layers (02_rest_client_and_interceptors.md, Modules 17, 15, 19).

Summary

- HTTP = method + URL + headers + body → status + headers + body; know the status classes.
- `http` (minimal, manual) vs `dio` (interceptors/timeouts/cancel/auto-decode) — prefer `dio` for production.
- Set timeouts, offload heavy parsing, and map responses to domain types at the repository.

Revision Notes

- Methods: GET/POST/PUT/PATCH/DELETE (idempotency varies); status: 2xx/3xx/4xx/5xx.
- Headers: Content-Type/Accept/Authorization/caching; body = JSON.
- `http` = manual status+decode; `dio` = interceptors/timeouts/cancel/auto-decode/throws on non-2xx.
- Timeouts always; offload big parsing; map to domain at repo.

Practice Questions

1. What does a 401 vs 404 vs 503 indicate?
2. How does error handling differ between `http` and `dio`?
3. Why offload large JSON parsing?

Coding Questions

1. GET + POST with `http`, checking status and decoding.
2. Same with `dio` + `BaseOptions` (timeouts, base URL).
3. Parallelize three independent GETs with `Future.wait`.

Mini Project

Two-client comparison (Flutter/Dart): Implement product fetch/create with both `http` and `dio` (timeouts, base URL), handling status codes and mapping responses to a domain model at a repository boundary; parse a large list off-isolate. Acceptance: status handled; timeouts set; no transport types leak to callers; big parse offloaded; runs.

REST Client & Interceptors (Production `dio` Setup)

> A production REST client centralizes cross-cutting concerns in **interceptors** (auth token injection, logging, retry, error mapping), enforces **timeouts/cancellation**, converts failures to a **typed** `Result`, and maps **DTOs→entities** at the repository — so feature code sees only domain objects, never HTTP.

Introduction

This is the heart of the module: turning raw `dio` into a robust, testable API layer. Interceptors handle auth/logging/retry/errors uniformly; the repository maps DTOs to entities and returns typed results; timeouts and cancel tokens keep it resilient. This applies the Decorator/Interceptor and Repository patterns (05).

Why this concept exists

Without central handling, every call re-implements auth headers, logging, retry, and error parsing — inconsistent and bug-prone. Interceptors and a repository boundary centralize these, so adding auth or retry happens once, and the UI depends on clean domain APIs returning success/failure.

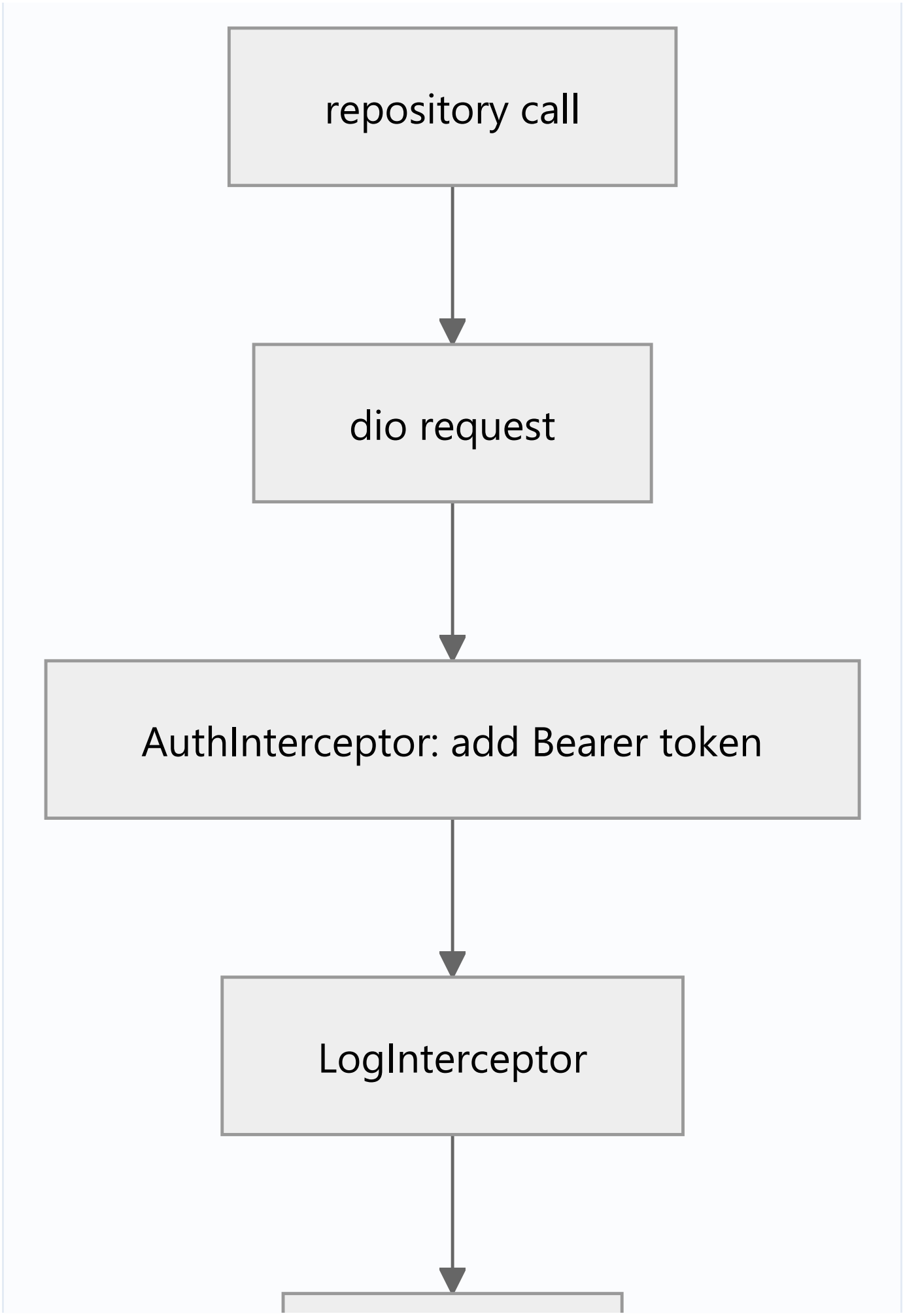
Real-world analogy

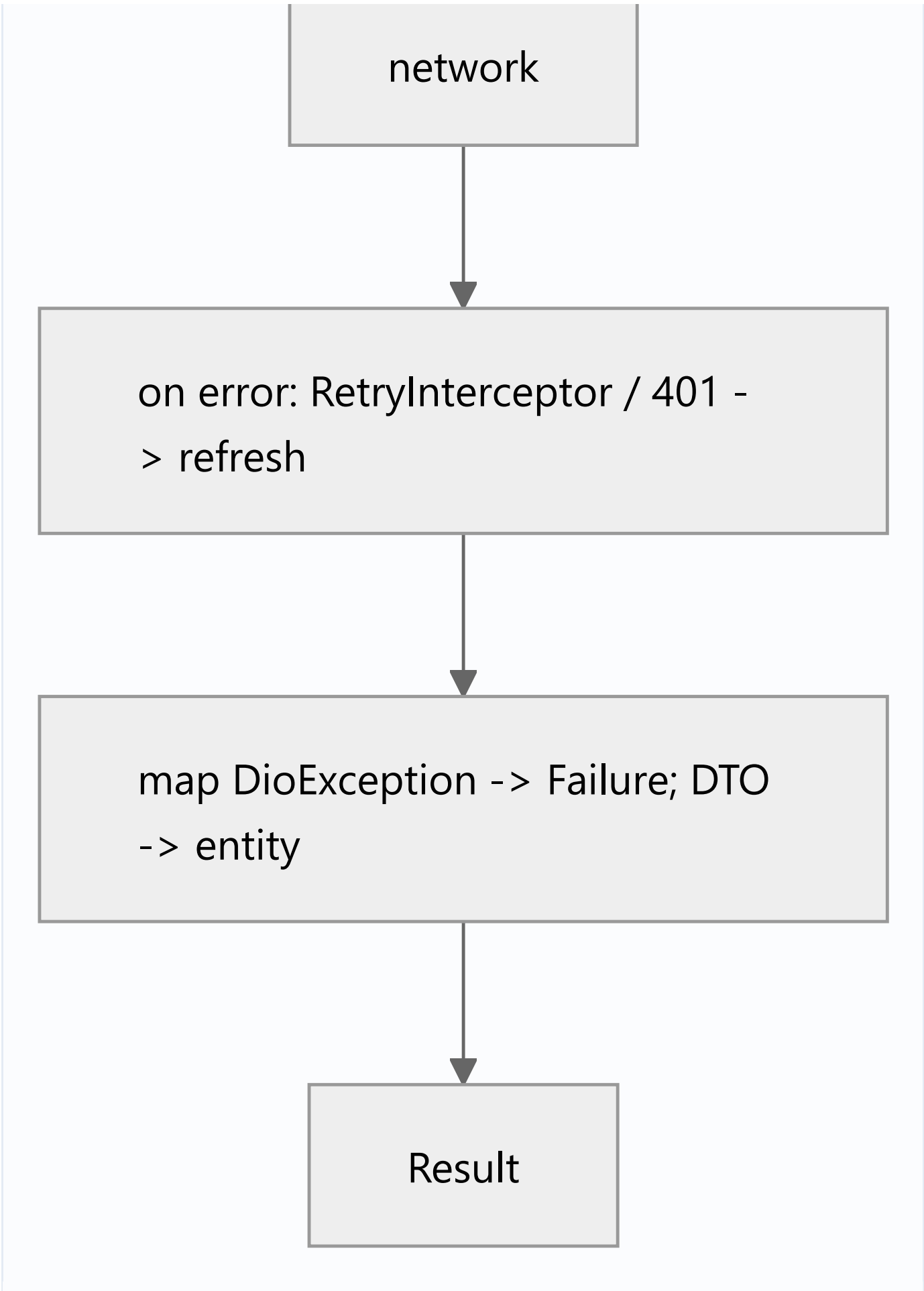
A **corporate mailroom**: every outgoing letter automatically gets a stamp (auth token) and a log entry (logging); failed deliveries are auto-retried; and incoming mail is translated into your language (DTO→entity) before reaching your desk. You never touch postage or translation — the mailroom (interceptors + repository) does.

Problem Statement

Build an API layer where every request carries the auth token, is logged, retries transient failures, times out, can be cancelled, and returns a typed `Result<T, Failure>` of domain entities. You'll wire `dio` interceptors + a repository.

Internal Working


repository call
dio request
AuthInterceptor: add Bearer token
LogInterceptor


network
on error: RetryInterceptor / 401 -> refresh
map DioException -> Failure; DTO -> entity
Result

- **Interceptors** (`dio.interceptors.add(...)`): run on every request/response/error:
  - **Auth**: inject `Authorization: Bearer <token>` (from secure storage — 15 · secure_storage); on 401, refresh token + retry.
  - **Logging**: log method/url/status/timing (redact secrets).
  - **Retry**: retry transient failures (timeouts, 5xx, network) with backoff, bounded attempts.
  - **Error mapping**: translate `DioException` into your domain `Failure` types.
- **Timeouts**: `connectTimeout` / `receiveTimeout` / `sendTimeout` on `BaseOptions` .
- **Cancellation**: `CancelToken` per request; cancel on screen dispose / new search query (08 · dispose_and_leaks).
- **Typed results**: the repository catches errors and returns `Result<T, Failure>` (or `Either`) instead of throwing across layers (Module 38).
- **DTO↔entity**: parse response into DTOs, map to domain entities; the UI never sees DTOs/ `Response` (02 · json, 05 · repository).

Memory Representation

The `dio` instance + interceptors are typically singletons (14 · scopes_and_lifetimes); cancel tokens tie to widget lifetimes. Offload large parses off-isolate (02 · isolates).

Compiler Behavior

Typed `Result` /sealed failures give exhaustive handling (02 · records_and_patterns).

Runtime Behavior

Interceptors execute in order per request/response/error; retry/refresh re-issue requests; cancelled requests throw a cancellation error the repository maps to a "cancelled" failure or ignores.

Flutter Engine Behavior / Dart VM Behavior

Not applicable beyond async/sockets.

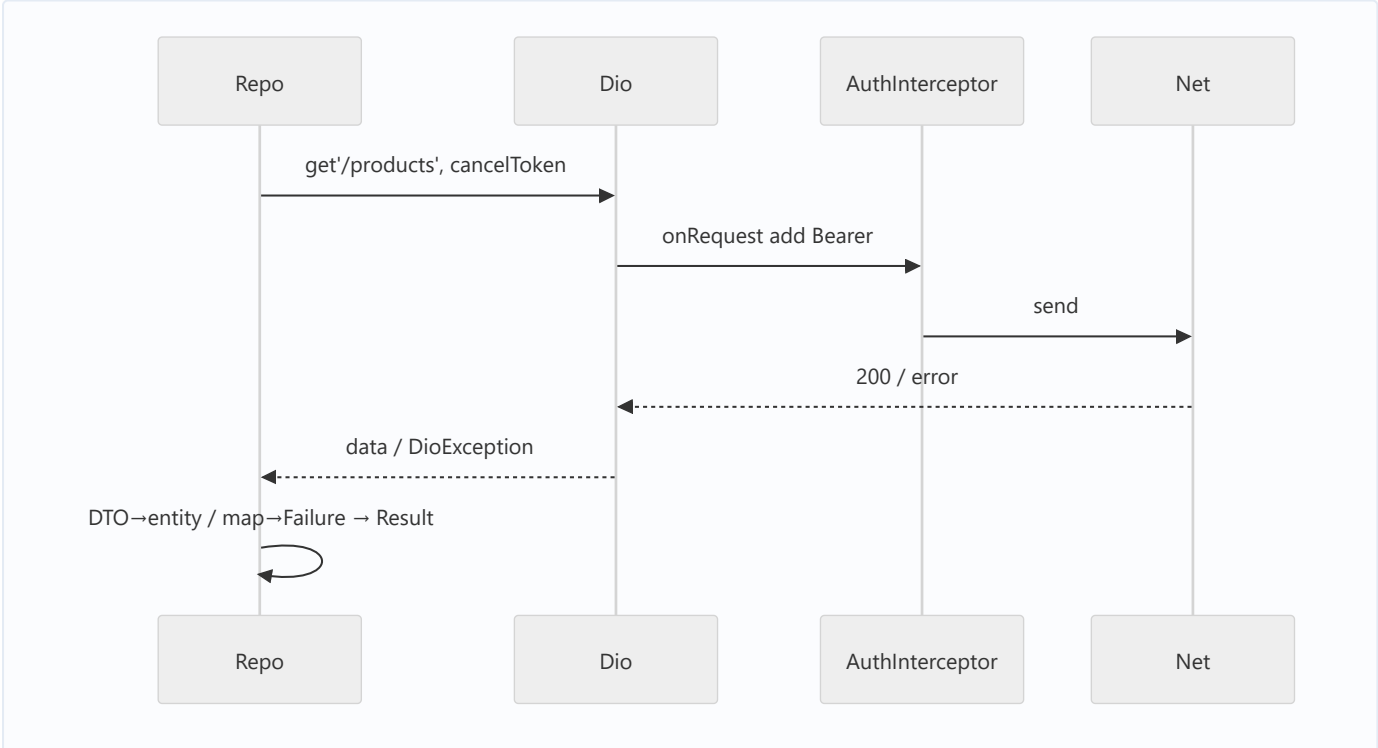
Examples

```
import 'package:dio/dio.dart';


// Domain types
class Product { final String id, name; const Product(this.id, this.name); }
```

```
sealed class Failure { const Failure(); }
class NetworkFailure extends Failure { const NetworkFailure(); }
class ServerFailure extends Failure { final int code; const ServerFailure(this.code); }
class UnauthorizedFailure extends Failure { const UnauthorizedFailure(); }

// A tiny Result type
sealed class Result<T> { const Result(); }
class Ok<T> extends Result<T> { final T value; const Ok(this.value); }
class Err<T> extends Result<T> { final Failure failure; const Err(this.failure); }

// Auth interceptor: inject token, refresh on 401
class AuthInterceptor extends Interceptor {
  final Future<String?> Function() getToken;
  AuthInterceptor(this.getToken);
  @override
  void onRequest(RequestOptions options, RequestInterceptorHandler handler) async {
    final token = await getToken();
    if (token != null) options.headers['Authorization'] = 'Bearer $token';
    handler.next(options);
  }
  // (onError could refresh the token and retry the request)
}

Dio buildDio(Future<String?> Function() getToken) {
  final dio = Dio(BaseOptions(
    baseUrl: 'https://api.example.com',
    connectTimeout: const Duration(seconds: 10),
    receiveTimeout: const Duration(seconds: 10),
  ));
  dio.interceptors.addAll([
    AuthInterceptor(getToken),
    LogInterceptor(requestBody: false, responseBody: false), // redact in real apps
    // RetryInterceptor(...) for transient failures
  ]);
  return dio;
}

// Repository: DTO -> entity, DioException -> Failure, returns Result
class ProductRepository {
  final Dio dio;
```

```
  ProductRepository(this.dio);

  Future<Result<List<Product>>> getProducts({CancelToken? cancel}) async {
    try {
      final res = await dio.get('/products', cancelToken: cancel);
      final list = (res.data as List)
          .map((j) => Product(j['id'] as String, j['name'] as String)) // DTO->entity
          .toList();
      return Ok(list);
    } on DioException catch (e) {
      return Err(_mapError(e)); // typed failure, never leaks DioException
    }
  }

  Failure _mapError(DioException e) => switch (e.response?.statusCode) {
        401 => const UnauthorizedFailure(),
        final int code when code >= 500 => ServerFailure(code),
        _ => const NetworkFailure(),
      };
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Auth/logging/retry per call	Duplicated, inconsistent	Centralize in interceptors
Throwing `DioException` to the UI	Leaks transport, hard to handle	Map to typed `Failure` / `Result`
No timeouts/cancellation	Hangs, wasted work, leaks	Timeouts + `CancelToken`
Logging tokens/PII	Security leak	Redact in log interceptor
Unbounded retry	Hammering/backoff storms	Bounded retries + backoff, only transient errors
Returning DTOs to UI	Couples to wire format	Map DTO→entity in repo

Best Practices

- Centralize **auth/logging/retry/error-mapping** in interceptors; keep one configured `dio` (DI singleton).
- Always set **timeouts**; use `CancelToken` and cancel on dispose/new query.
- Return **typed** `Result` / `Failure` from repositories; never leak `DioException` / `Response`.
- **Retry only transient** errors (timeouts/5xx/network) with **bounded backoff**; refresh token on 401 then retry once.
- Map **DTO→entity** at the boundary; offload large parses off-isolate.
- **Redact secrets** in logging.

Performance

Interceptors add negligible overhead; retry/backoff prevent storms; cancellation avoids wasted work; parallelize independent calls and offload parsing (Module 21).

Advantages / Disadvantages

- **+** Uniform cross-cutting concerns, resilient (retry/timeout/cancel), typed errors, testable (mock `dio`/repo), clean UI.
- **−** More setup; interceptor ordering/refresh logic can get subtle; must avoid retry storms.

Interview Questions

1. 🟢 **What are `dio` interceptors for?** — Running cross-cutting logic on every request/response/error (auth injection, logging, retry, error mapping).
2. 🟢 **Why return a typed `Result` / `Failure` from repositories?** — So callers handle success/failure explicitly without catching transport exceptions; the UI stays decoupled from HTTP.

3. 🟡 **How do you handle a 401?** — In an interceptor: refresh the token and retry the request once; if refresh fails, surface an `UnauthorizedFailure` (→ re-login).
4. 🟡 **What errors should you retry, and how?** — Transient ones (timeouts, network, 5xx) with bounded attempts + exponential backoff — never non-idempotent unsafe retries or 4xx like 400/401.
5. 🟡 **How do you cancel requests?** — With a `CancelToken` per request, cancelled on screen dispose or when a newer request supersedes it.
6. 🔴 **Where do DTOs become entities?** — At the repository boundary (DTO→entity mapping); the UI never sees DTOs or `Response`.
7. 🔴 **How do you keep this testable?** — Inject `dio` (mock it) or the repository (fake it); assert `Result` outcomes for success/error paths (Module 14).

Senior Engineer Tips

- Build the client once (interceptors + timeouts) and inject it; never construct `dio` ad hoc in features.
- Get the **401→refresh→retry-once** flow right early; it's the most error-prone interceptor (avoid infinite refresh loops).
- Make failures a **sealed type** so the UI can exhaustively map each to messaging/actions (Module 38).

Architect Perspective

A disciplined REST client is the data layer's backbone: interceptors centralize cross-cutting policy, the repository boundary enforces domain purity (entities + typed failures), and resilience features (timeout/retry/cancel) make the app robust on real networks. This composes with caching/offline (15, 19), auth (17), and error handling (38) into a production data stack.

Summary

- Centralize auth/logging/retry/error-mapping in interceptors; one configured `dio`.
- Enforce timeouts + cancellation; return typed `Result` / `Failure`; map DTO→entity at the repo.
- Retry only transient errors with backoff; refresh on 401; redact secrets; keep it testable.

Revision Notes

- Interceptors: auth (Bearer/refresh-on-401), logging (redact), retry (transient+backoff), error→Failure.
- `BaseOptions` timeouts; `CancelToken` per request (cancel on dispose/new query).
- Repo returns `Result<T, Failure>` (sealed); DTO→entity at boundary; no `DioException` to UI.

- Inject/mock `dio`/repo for tests; offload big parses.

Practice Questions

1. Design the 401→refresh→retry flow (avoiding loops).
2. Which failures are safe to retry and how?
3. Why never leak `DioException` to the UI?

Coding Questions

1. Write an `AuthInterceptor` that injects a token and refreshes on 401 (retry once).
2. Add a bounded retry-with-backoff interceptor for transient errors.
3. Implement a repository returning `Result<List<Entity>, Failure>` with DTO mapping + cancellation; test with a mocked `dio`.

Mini Project — Module capstone (REST)

Robust API layer (Flutter): Build a configured `dio` (timeouts) with auth + logging + retry interceptors, a `ProductRepository` returning `Result<List<Product>, Failure>` (DTO→entity, `DioException` → `Failure`), and per-request `CancelToken`s cancelled on dispose. Unit-test success/401/timeout/5xx with a mocked `dio`. Acceptance: cross-cutting concerns in interceptors; typed failures; timeouts/cancellation; no transport leakage; tests pass.

GraphQL

> GraphQL is a query language where the **client specifies exactly which fields it wants** in a single request against a typed schema — reducing over/under-fetching — with queries (read), mutations (write), and subscriptions (real-time), plus a normalized client cache.

Introduction

GraphQL replaces many REST endpoints with a single typed endpoint you query precisely. This file covers queries/mutations/subscriptions, variables, the `graphql_flutter` client and its normalized cache, and when GraphQL beats REST.

Why this concept exists

REST often over-fetches (returns fields you don't need) or under-fetches (requires multiple round-trips to assemble a screen). GraphQL lets the client ask for exactly the shape it needs in one request, driven by a server schema — efficient for complex, nested, client-driven data needs.

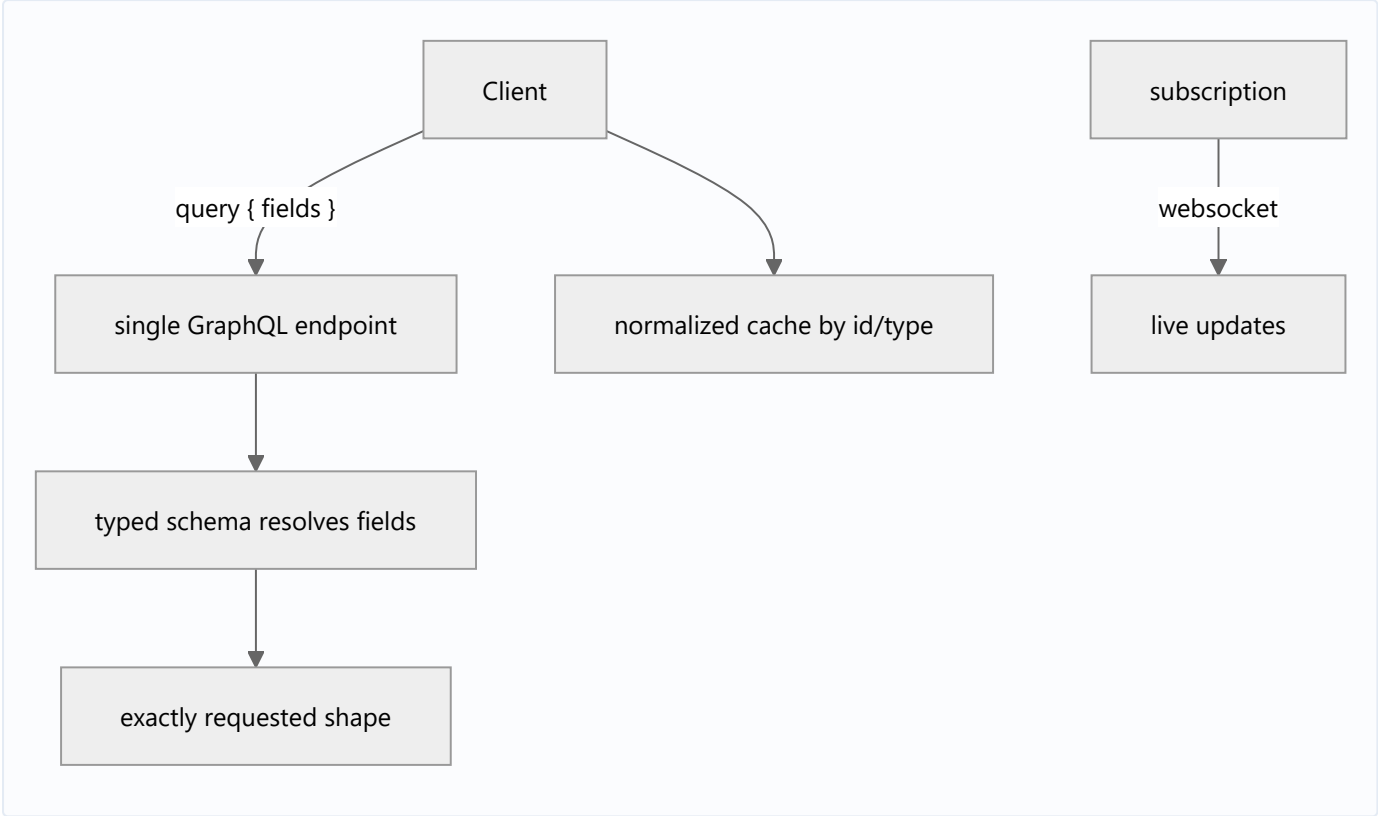
Real-world analogy

REST is a **fixed-menu restaurant** (you order dish #4 and get whatever it includes); GraphQL is a **build-your-own bowl** — you specify exactly the ingredients (fields) you want, in one order, and get precisely that.

Problem Statement

A profile screen needs a user's name, avatar, and their last 5 posts' titles — one request, no extra fields. You'll write a GraphQL query with variables, run a mutation, and note subscriptions + caching.

Internal Working



- **Operations**:
  - **Query** (read): request a field tree; server returns exactly that shape.
  - **Mutation** (write): create/update/delete, returning selected fields of the result.
  - **Subscription** (real-time): push updates over a WebSocket (04_websockets_and_realtime.md).
- **Variables**: parameterize operations (`query($id: ID!)`), passed separately — avoids string interpolation/injection.
- **Schema/types**: the server defines types/fields; clients get typed, validated queries (codegen tools generate Dart types).
- **Client (`graphql_flutter`)**: `GraphQLClient` with a `Link` (HTTP + WebSocket for subs) and a **normalized cache** (stores objects by `id` / `__typename` , so entities update everywhere).
- **Caching**: normalized cache dedupes/updates entities across queries; policies control cache-first vs network.

Memory Representation

The normalized cache holds entities keyed by type+id (in-memory, optionally persisted). Large results still cost memory; request only needed fields (02 · memory_and_gc).

Compiler Behavior

With codegen (`graphql_codegen` / `ferry`), operations become typed Dart classes — compile-time-safe queries/variables/results.

Runtime Behavior

Queries hit the endpoint (or cache per policy); mutations update the cache (auto for entities with ids, or via manual cache writes); subscriptions stream over WebSocket.

Flutter Engine Behavior / Dart VM Behavior

Not applicable beyond async/sockets.

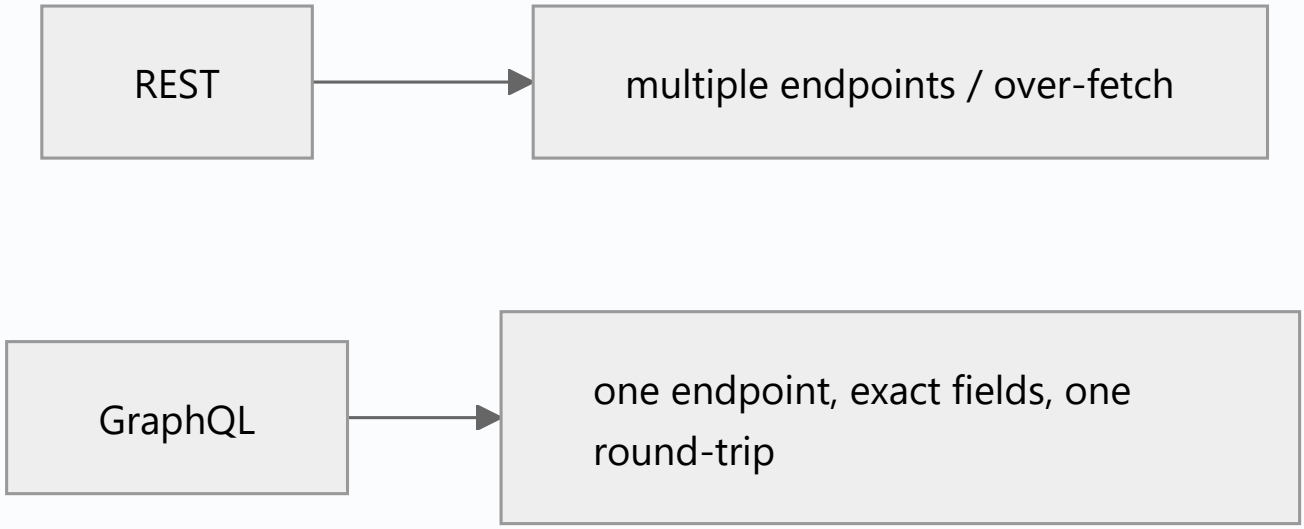
Examples

```
# pubspec.yaml
dependencies:
  graphql_flutter: ^5.1.0
```

```
import 'package:graphql_flutter/graphql_flutter.dart';

final client = GraphQLClient(
  link: HttpLink('https://api.example.com/graphql'),
  cache: GraphQLCache(), // normalized cache
);

// Query with variables — ask for EXACTLY these fields:
const _profileQuery = r'''
  query Profile($id: ID!) {
    user(id: $id) {
      id
      name
      avatarUrl
      posts(last: 5) { title }
    }
  }
''';

Future<Map<String, dynamic>?> fetchProfile(String id) async {
  final result = await client.query(QueryOptions(
```

```
    document: gql(_profileQuery),
    variables: {'id': id},
    fetchPolicy: FetchPolicy.cacheAndNetwork, // cache-first + revalidate
  ));
  if (result.hasException) {
    throw result.exception!; // map to Failure at the repository boundary
  }
  return result.data?['user'] as Map<String, dynamic>?;
}

// Mutation:
const _likeMutation = r'''
  mutation Like($postId: ID!) {
    likePost(id: $postId) { id likes }
  }
''';
Future<void> like(String postId) => client.mutate(
    MutationOptions(document: gql(_likeMutation), variables: {'postId': postId}));
```

Diagrams



Common Mistakes

Mistake	Why	Fix
String-interpolating variables	Injection/errors	Use `variables:` (parameterized)
Over-selecting fields	Defeats GraphQL's benefit	Request only needed fields
Ignoring the normalized cache	Redundant fetches/stale entities	Use `id` / `__typename`; set fetch policies
Leaking client types to UI	Coupling	Map to entities at repo boundary
Assuming GraphQL is always better	Not for simple APIs	Use REST when a simple resource API suffices

Best Practices

- Request **exactly the fields** each screen needs; use **variables** (never interpolate).
- Leverage the **normalized cache** (ids + `__typename`) so entities update consistently; pick fetch policies (`cacheAndNetwork`, `networkOnly`).
- Adopt **codegen** (`graphql_codegen` / `ferry`) for type-safe operations.
- Wrap the client in a **repository** returning entities/typed failures; don't leak client types.
- Use **subscriptions** for real-time; otherwise queries/mutations.

Performance

Fewer round-trips + precise fields reduce payloads/latency; the normalized cache dedupes and enables cache-first. Beware huge nested queries (cost/complexity limits server-side) (Module 21).

Advantages / Disadvantages

- **+** Precise fetching (no over/under-fetch), single endpoint, typed schema, normalized cache, subscriptions, great for complex client-driven data.
- **−** Server/tooling complexity, caching subtleties, overkill for simple CRUD, learning curve, query-cost/security considerations.

Interview Questions

1. 🟢 **What problem does GraphQL solve?** — Over/under-fetching: the client requests exactly the fields it needs in one typed request against a schema.
2. 🟢 **Query vs mutation vs subscription?** — Read; write (create/update/delete); real-time push (over WebSocket).
3. 🟡 **Why use variables instead of interpolation?** — Safety (no injection) and reuse; variables are typed and passed separately.

4. 🟡 **What is the normalized cache?** — A client cache storing entities by type+id (`__typename` / `id`) so the same entity updates consistently across queries.
5. 🟡 **When is REST preferable to GraphQL?** — Simple resource CRUD APIs, or when server/tooling complexity isn't justified.
6. 🔴 **How do you keep GraphQL type-safe in Dart?** — Codegen (`graphql_codegen` / `ferry`) turns operations into typed classes (compile-time-checked queries/results).
7. 🔴 **What are GraphQL's server-side risks?** — Expensive/deeply-nested queries; mitigate with query-cost limits, depth limits, and persisted queries.

Senior Engineer Tips

- Keep queries **field-minimal and screen-scoped**; co-locate queries with the widgets that use them.
- Use fetch policies deliberately (`cacheAndNetwork` for lists = SWR-like UX — 15 · caching_strategies).
- Front the client with a repository returning domain entities/typed failures, exactly like REST.

Architect Perspective

GraphQL shifts data-shaping to the client with a typed contract — powerful for complex, nested, evolving UIs and for reducing round-trips. Architecturally it still sits behind repositories (entities + typed failures) with caching policies; the choice vs REST depends on data complexity, team, and backend capability (05 · repository, Module 40).

Summary

- GraphQL: one typed endpoint; clients request exact fields via queries/mutations/subscriptions with variables.
- Normalized cache (id/ `__typename`) + fetch policies; codegen for type safety.
- Great for complex client-driven data; front with repositories; use REST for simple APIs.

Revision Notes

- Query (read)/mutation (write)/subscription (real-time via WS); variables (typed, not interpolated).
- `graphql_flutter`: `GraphQLClient` + `Link` + normalized cache; fetch policies (`cacheAndNetwork`).
- Codegen for typed ops; repo maps to entities/failures; request minimal fields.
- REST for simple CRUD; watch server query cost.

Practice Questions

1. How does GraphQL avoid over/under-fetching?
2. Why use variables over interpolation?
3. What does the normalized cache do?

Coding Questions

1. Write a parameterized query (variables) selecting minimal fields.
2. Run a mutation and update the cache.
3. Wrap the client in a repository returning entities/ `Result`.

Mini Project

GraphQL profile (Flutter): Build a `ProfileRepository` over `graphql_flutter` fetching a user + last posts with a variabled query (`cacheAndNetwork`), a like mutation updating the cache, mapping results to entities and exceptions to failures. Acceptance: minimal field selection; variables used; cache leveraged; entities/failures at boundary; runs.

WebSockets & Real-Time (WebSockets, SSE, Socket.IO)

> Real-time features (chat, live prices, presence) need a persistent connection: **WebSockets** (bidirectional), **Server-Sent Events** (server→client only), or **Socket.IO** (WebSocket + fallbacks + rooms) — exposed to the app as a `Stream`, with reconnection and lifecycle management.

Introduction

Request/response HTTP can't push updates. This file covers persistent-connection options — **WebSockets** (`web_socket_channel`), **SSE** (one-way server push), and **Socket.IO** (`socket_io_client`) — how to model them as `Stream`s, and the hard parts: reconnection, heartbeats, and lifecycle/leak management.

Why this concept exists

Chat, notifications, live dashboards, collaborative editing, and presence require the server to push data as it happens. Polling is wasteful/laggy; persistent connections deliver low-latency, bidirectional/streamed updates.

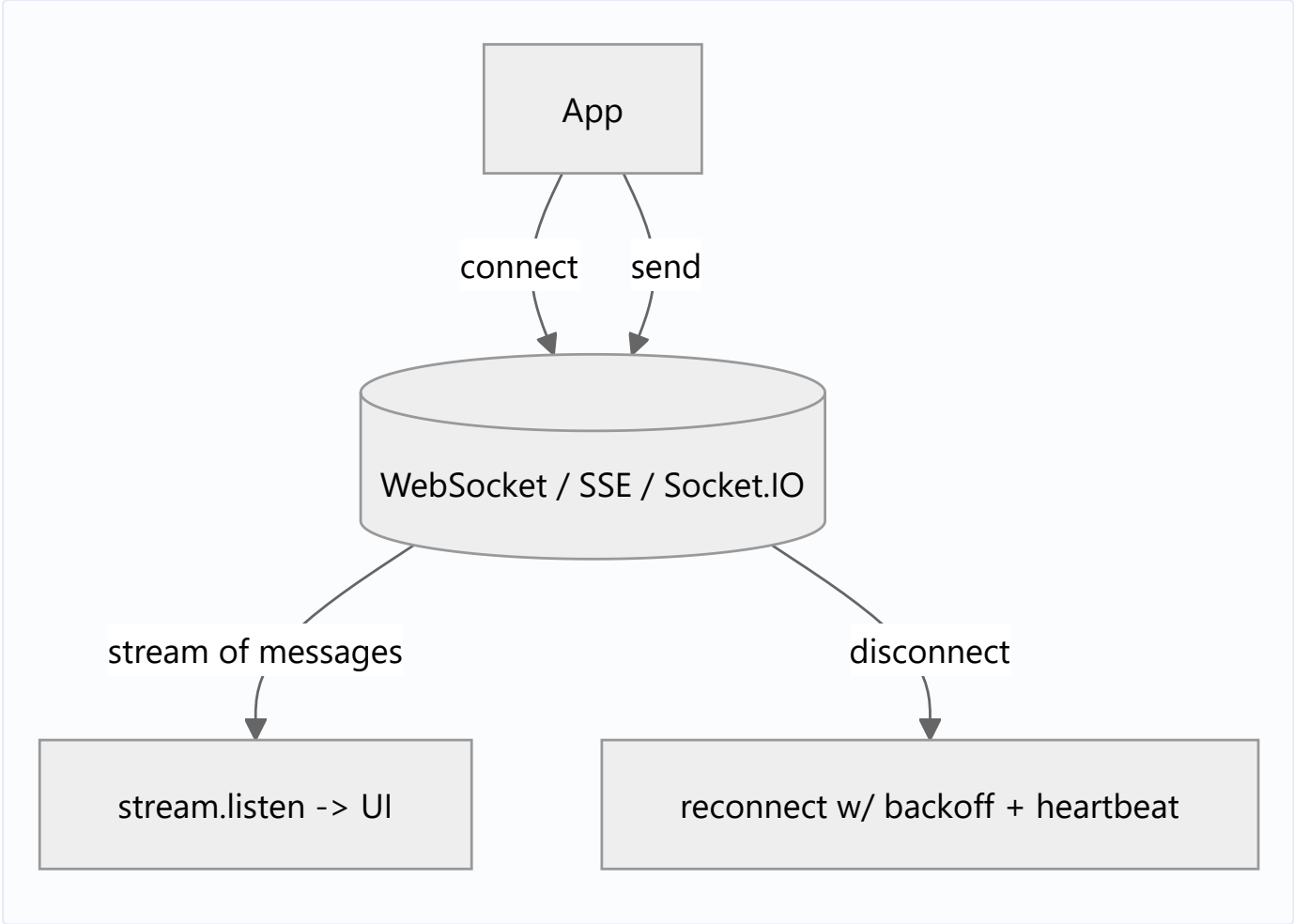
Real-world analogy

HTTP is **sending letters** (one round-trip each). A WebSocket is an **open phone line** — both sides talk anytime. SSE is a **radio broadcast** (you only listen). Socket.IO is a **managed conference call** with auto-redial (reconnect) and breakout rooms (channels).

Problem Statement

A chat screen must receive messages in real time, send messages, reconnect after drops, and clean up on dispose. You'll open a WebSocket, expose it as a `Stream`, handle reconnection, and cancel on dispose.

Internal Working



- **WebSocket** (`web_socket_channel`): full-duplex over one TCP connection. `channel.stream` (incoming) + `channel.sink.add(...)` (outgoing). Use for chat, games, collaborative apps.
- **SSE** (Server-Sent Events): **server→client only**, over HTTP, auto-reconnecting, text/event-stream. Simpler than WS for one-way feeds (notifications, live scores).
- **Socket.IO** (`socket_io_client`): a library/protocol atop WebSocket with **automatic reconnection, fallbacks, rooms/namespaces, and event names** (`socket.on('event')` / `emit`). Requires a Socket.IO server (not plain WS).
- **Model as a `Stream`**: expose incoming messages as a broadcast `Stream` the UI/blocs consume (02 · streams); consumers `listen` and **cancel** subscriptions on dispose.
- **Resilience**: reconnect with backoff, heartbeat/ping-pong to detect dead connections, resubscribe/replay on reconnect, buffer outgoing while disconnected.
- **Lifecycle**: close the channel and cancel subscriptions on dispose; pause on app background (08 · app_lifecycle).

Memory Representation

Open connections + stream subscriptions retain callbacks (and captured state) → leaks if not cancelled/closed (08 · dispose_and_leaks, 02 · streams).

Compiler Behavior

Not applicable.

Runtime Behavior

Messages arrive as stream events; disconnects surface as stream errors/ `onDone` → trigger reconnect. Backpressure/buffering may be needed for fast producers.

Flutter Engine Behavior

Uses platform sockets via the embedder; runs on the isolate's event loop — keep message handlers light (10 · threading_model).

Dart VM Behavior

Not applicable.

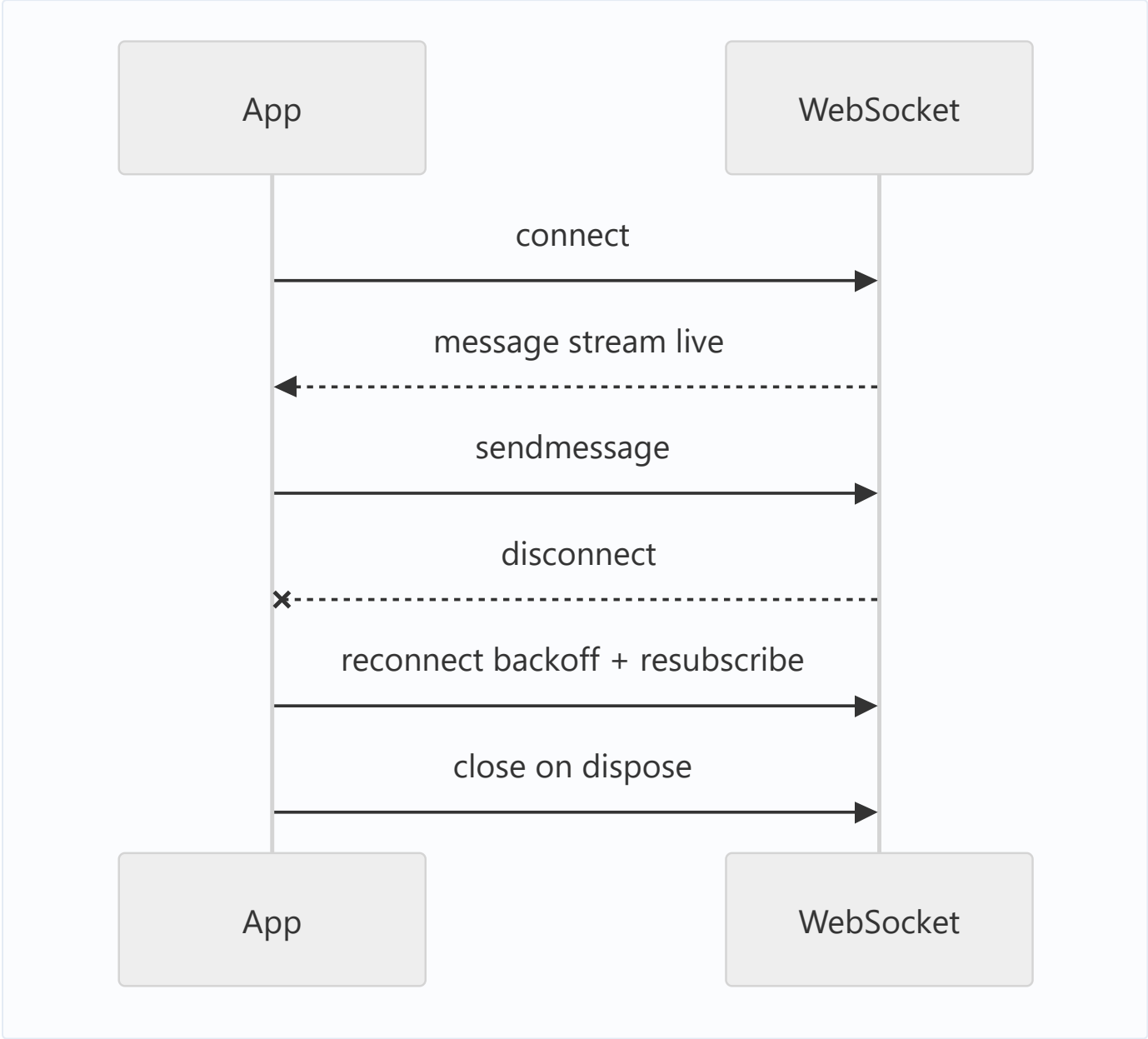
Examples

```
# pubspec.yaml
dependencies:
  web_socket_channel: ^2.4.0
  # socket_io_client: ^2.0.0   # if using a Socket.IO server
```

```
import 'dart:async';
import 'package:web_socket_channel/web_socket_channel.dart';

// Chat service exposing incoming messages as a Stream, with reconnect + cleanup
class ChatService {
  final Uri uri;
  WebSocketChannel? _channel;
  final _controller = StreamController<String>.broadcast();
  StreamSubscription? _sub;
  bool _closed = false;

  ChatService(this.uri) { _connect(); }

  Stream<String> get messages => _controller.stream;

  void _connect() {
```

```
    _channel = WebSocketChannel.connect(uri);
    _sub = _channel!.stream.listen(
      (data) => _controller.add(data as String),   // incoming -> stream
      onError: (_) => _reconnect(),
      onDone: _reconnect,                           // disconnect -> reconnect
    );
  }

  Future<void> _reconnect() async {
    if (_closed) return;
    await _sub?.cancel();
    await Future.delayed(const Duration(seconds: 2)); // backoff (grow with attempts)
    _connect();
  }

  void send(String msg) => _channel?.sink.add(msg);  // outgoing

  Future<void> dispose() async {                     // MUST clean up
    _closed = true;
    await _sub?.cancel();
    await _channel?.sink.close();
    await _controller.close();
  }
}

// UI/bloc consumes ChatService.messages via a StreamBuilder / listen,
// and calls dispose() in State.dispose().
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Not closing channel / cancelling subscription	Leaks, battery drain	Close + cancel in `dispose`
No reconnection logic	Silent dead connection	Reconnect with backoff + heartbeat
Using plain WS with a Socket.IO server (or vice versa)	Protocol mismatch	Match client to server protocol
Heavy work in message handlers	Jank	Keep handlers light; offload
Not pausing on background	Wasted battery/data	Pause/close on app background (08)
Losing outgoing messages while disconnected	Data loss	Buffer + flush on reconnect

Best Practices

- Expose real-time data as a **broadcast** `Stream`; consumers `listen` and **cancel on dispose**; **close** the channel.
- Implement **reconnection with backoff** + **heartbeat/ping** to detect dead links; resubscribe/replay on reconnect.
- **Pause/close** connections on app background; resume on foreground.
- Choose the protocol: **WebSocket** (bidirectional), **SSE** (one-way feeds), **Socket.IO** (managed reconnect/rooms, needs matching server).
- Keep message handlers light; front the service behind a repository/bloc.

Performance

Persistent connections are efficient for frequent updates (vs polling) but consume battery/data when idle — pause on background. Keep handlers light; buffer/batch high-frequency streams (Module 21).

Advantages / Disadvantages

- **+** Low-latency push, bidirectional (WS/Socket.IO), efficient vs polling, natural `Stream` model.
- **−** Connection/lifecycle management (reconnect/heartbeat/cleanup), battery/data cost, protocol-specific servers, leak-prone if undisciplined.

Interview Questions

1. 🟢 **Why not use HTTP polling for real-time?** — It's wasteful and laggy; persistent connections (WebSocket/SSE/Socket.IO) push updates with low latency.
2. 🟢 **WebSocket vs SSE?** — WebSocket is full-duplex (both directions); SSE is server→client only (simpler, HTTP-based, auto-reconnecting).
3. 🟡 **What is Socket.IO and what does it add?** — A library/protocol atop WebSocket adding auto-reconnection, transport fallbacks, rooms/namespaces, and named events — requires a Socket.IO server.
4. 🟡 **How do you model incoming messages in Flutter?** — As a (broadcast) `Stream` consumers `listen` / `StreamBuilder` on, cancelling subscriptions on dispose.
5. 🟡 **Why must you close/cancel connections?** — Open channels + subscriptions leak memory/battery and can fire callbacks after dispose.
6. 🔴 **How do you make a real-time connection resilient?** — Reconnect with exponential backoff, heartbeat/ping to detect dead links, resubscribe/replay and buffer outgoing on reconnect.

7. 🔴 **How should real-time connections behave on app background?** — Pause/close to save battery/data; resume/reconnect on foreground (08 · app_lifecycle).

Senior Engineer Tips

- Wrap the socket in a service exposing a `Stream` + `send` + `dispose`; keep reconnection/heartbeat inside it so the UI is simple.
- Always handle background/foreground (pause/reconnect) and buffer outgoing messages while offline.
- Treat every subscription/connection as an owned resource with explicit cleanup — a top real-time leak source.

Architect Perspective

Real-time connectivity is a stateful, lifecycle-heavy subsystem best encapsulated behind a service/repository exposing streams. Reconnection, heartbeat, background handling, and cleanup are cross-cutting reliability concerns; centralizing them yields robust chat/live features and integrates with app lifecycle, notifications, and offline strategy (Modules 08, 32, 19).

Summary

- Real-time needs persistent connections: WebSocket (bidirectional), SSE (one-way), Socket.IO (managed, rooms).
- Model as a `Stream`; implement reconnection/backoff/heartbeat; pause on background; close/cancel on dispose.
- Encapsulate behind a service/repository; keep handlers light; buffer outgoing while disconnected.

Revision Notes

- WebSocket (full-duplex, `web_socket_channel`) / SSE (server→client, auto-reconnect) / Socket.IO (reconnect+rooms, needs server).
- Expose broadcast `Stream`; cancel subscriptions + close channel on dispose (leak!).
- Reconnect w/ backoff + heartbeat + resubscribe; buffer outgoing; pause on background.
- Light handlers; front behind service/repo.

Practice Questions

1. WebSocket vs SSE vs Socket.IO — pick for chat, live scores, and rooms.
2. How do you detect and recover from a dead connection?

3. Why pause connections on app background?

Coding Questions

1. Build a `ChatService` (WebSocket) exposing a message `Stream` + `send` + `dispose`.
2. Add reconnection with exponential backoff + heartbeat.
3. Pause/resume the connection on app lifecycle changes.

Mini Project

Live chat service (Flutter): Build a `ChatService` over `web_socket_channel` exposing incoming messages as a broadcast `Stream`, with `send`, reconnection (backoff + heartbeat), outgoing buffering while disconnected, background pause/resume, and full cleanup on dispose. Consume it via `StreamBuilder`. Acceptance: real-time updates; resilient reconnect; no leaks (closed/cancelled); background-aware; runs.

gRPC

> gRPC is a high-performance RPC framework using **Protocol Buffers** (binary, schema-defined messages) over HTTP/2, with generated typed client/server stubs and four call types (unary + three streaming) — ideal for efficient, strongly-typed, low-latency service-to-service and mobile-to-backend communication.

Introduction

gRPC lets you call remote methods as if local, with messages defined in `.proto` files (Protocol Buffers) and typed Dart stubs generated by codegen. This file covers protobuf, the four RPC types, the `grpc` Dart package, and when gRPC fits vs REST/GraphQL.

Why this concept exists

For performance-sensitive, strongly-typed APIs (microservices, streaming, low-bandwidth mobile), JSON/REST is verbose and untyped. gRPC's compact binary protobuf + HTTP/2 multiplexing + generated types give smaller payloads, lower latency, streaming, and compile-time contracts.

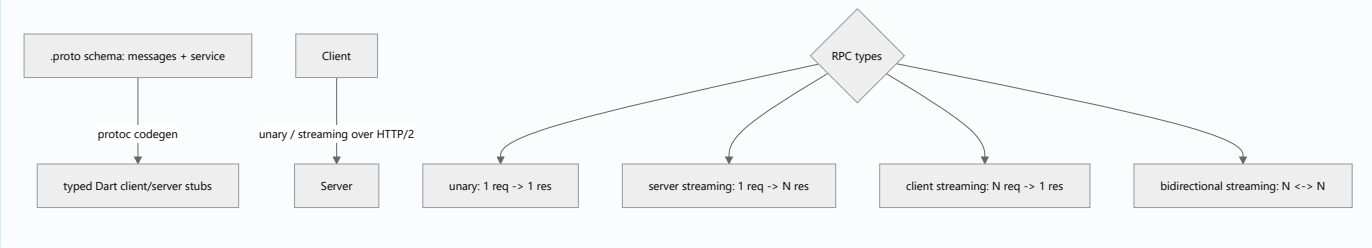
Real-world analogy

REST/JSON is **writing full sentences in a shared language** (verbose but universal); gRPC is a **pre-agreed shorthand code** between two parties (`.proto` schema) — compact, fast, and unambiguous, but both sides must share the codebook (generated stubs).

Problem Statement

You call a backend `GetProduct(id)` and stream live price updates, with typed messages and efficient binary transport. You'll define a `.proto`, generate stubs, and make unary + streaming calls.

Internal Working



- **Protocol Buffers**: define `message`s and a `service` with RPC methods in a `.proto`; `protoc` + the Dart plugin generate typed stubs and message classes (binary serialization).
- **Transport**: HTTP/2 (multiplexed streams, header compression) — efficient, supports streaming.
- **Four RPC types**:
  - **Unary**: one request → one response (like a normal call).
  - **Server streaming**: one request → a stream of responses (e.g., live prices).
  - **Client streaming**: a stream of requests → one response (e.g., upload chunks).
  - **Bidirectional streaming**: both stream concurrently (e.g., chat).
- **Dart (`grpc` package)**: create a `ClientChannel`, instantiate the generated client, call methods (returning `Future` / `Stream`), pass metadata (auth) via `CallOptions`.
- **Errors**: `GrpcError` with status codes (e.g., `unauthenticated`, `notFound`) — map to domain failures.
- **Web caveat**: browsers need **gRPC-Web** (a proxy like Envoy) since raw gRPC isn't browser-supported.

Memory Representation

Protobuf messages are compact binary → smaller than JSON in memory/on the wire. Streaming responses arrive incrementally (bounded memory) (02 · streams).

Compiler Behavior

`.proto` codegen produces **typed** Dart classes/stubs → compile-time-safe messages and method signatures (a major advantage over untyped JSON).

Runtime Behavior

Unary calls return `Future`; streaming calls return `Stream`; the channel manages HTTP/2 connections. Cancel via the call's `cancel()`; handle `GrpcError` status codes.

Flutter Engine Behavior

Uses platform networking via the embedder; web requires gRPC-Web proxying (10 · threading_model).

Dart VM Behavior

Not applicable beyond async/streams.

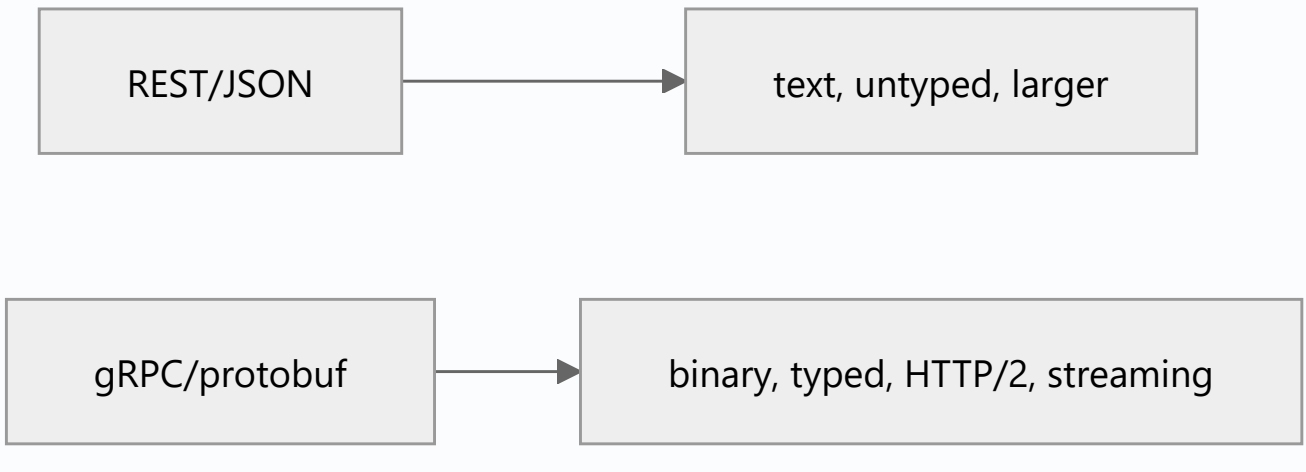
Examples

```
# pubspec.yaml
dependencies:
  grpc: ^3.2.0
  protobuf: ^3.1.0
dev_dependencies:
  protoc_plugin: ^21.0.0   # generates Dart from .proto via protoc
```

```
// product.proto
syntax = "proto3";
message ProductRequest { string id = 1; }
message Product { string id = 1; string name = 2; double price = 3; }
message PriceUpdate { double price = 1; }

service ProductService {
  rpc GetProduct (ProductRequest) returns (Product);                 // unary
  rpc WatchPrice (ProductRequest) returns (stream PriceUpdate);       // server streaming
}
```

```
// After `protoc --dart_out=grpc:lib/generated product.proto`, generated stubs exist.
import 'package:grpc/grpc.dart';
// import 'generated/product.pbgrpc.dart';

Future<void> useGrpc() async {
  final channel = ClientChannel(
    'api.example.com',
    port: 443,
    options: const ChannelOptions(credentials: ChannelCredentials.secure()),
  );
  // final client = ProductServiceClient(channel);

  // Unary call (typed):
  // final product = await client.getProduct(ProductRequest(id: '42'),
  //     options: CallOptions(metadata: {'authorization': 'Bearer $token'}));
  // print('${product.name}: ${product.price}');

  // Server streaming (Stream of typed updates):
  // await for (final update in client.watchPrice(ProductRequest(id: '42'))) {
  //   print('price: ${update.price}');
  // }

  await channel.shutdown(); // clean up the connection
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using gRPC on Flutter Web without gRPC-Web	Browsers don't support raw gRPC	Use gRPC-Web + a proxy (Envoy)
Not shutting down channels	Leaked connections	`channel.shutdown()` when done
Ignoring `GrpcError` status codes	Poor error handling	Map status → domain failures
No auth metadata	Unauthenticated calls	Pass token via `CallOptions.metadata`
Choosing gRPC for simple public APIs	Complexity/tooling overhead	REST/GraphQL may fit better
Leaking generated types to UI	Coupling	Map protobuf → entities at repo

Best Practices

- Define clear `.proto` contracts; commit generated stubs or generate in CI (`protoc`).
- Use the right **RPC type** (unary vs streaming) per use case; consume streams as Dart `Stream`s and cancel/ `shutdown` properly.
- Pass **auth via metadata** (`CallOptions`); map `GrpcError` to domain failures.
- On **web**, plan for **gRPC-Web** (proxy) — raw gRPC won't work in browsers.
- Front gRPC clients behind **repositories** returning entities/typed failures (map protobuf→entity).

Performance

Compact binary + HTTP/2 multiplexing + streaming → lower latency/bandwidth than JSON/REST, ideal for chatty/streaming/low-bandwidth scenarios. Streaming keeps memory bounded (Module 21).

Advantages / Disadvantages

- **+** Compact/fast binary, strongly typed (codegen), streaming (4 types), HTTP/2 multiplexing, great for microservices/mobile.
- **−** Tooling/codegen + `.proto` maintenance, not human-readable, web needs gRPC-Web proxy, overkill for simple public APIs, less ubiquitous than REST.

Interview Questions

1. 🟢 **What is gRPC?** — A high-performance RPC framework using Protocol Buffers (binary, schema-defined) over HTTP/2, with generated typed client/server stubs.

2. 🟢 **What are the four RPC types?** — Unary (1→1), server streaming (1→N), client streaming (N→1), bidirectional streaming (N↔N).
3. 🟡 **Why is gRPC efficient vs REST/JSON?** — Compact binary protobuf, HTTP/2 multiplexing/header compression, and typed contracts — smaller payloads, lower latency, streaming.
4. 🟡 **How are Dart types generated?** — From `.proto` via `protoc` + the Dart plugin, producing typed message/stub classes (compile-time-safe).
5. 🟡 **How do you pass auth in gRPC?** — Via call metadata (`CallOptions.metadata`, e.g., an `authorization` header).
6. 🔴 **Why can't raw gRPC run on Flutter Web?** — Browsers don't support the required HTTP/2 primitives; use **gRPC-Web** with a proxy (e.g., Envoy).
7. 🔴 **When choose gRPC over REST/GraphQL?** — Performance-critical, strongly-typed, streaming, or microservice/mobile-to-backend scenarios; not for simple public/human-facing APIs.

Senior Engineer Tips

- Treat `.proto` as the source-of-truth contract; automate codegen in CI and version schemas carefully (backward compatibility via field numbers).
- Map `GrpcError` status codes to your sealed failures and protobuf messages to entities at the repo boundary.
- For web targets, decide gRPC-Web early — it changes deployment (proxy) and client setup.

Architect Perspective

gRPC is a strong choice for typed, high-performance, streaming communication — common in microservice backends and performance-sensitive mobile clients. Architecturally it still sits behind repositories (entities + typed failures); the tradeoff vs REST/GraphQL is efficiency/typing vs tooling/ubiquity/web friction — a deliberate protocol decision per system needs (05 · repository, Module 48).

Summary

- gRPC = protobuf (binary, typed, schema) + HTTP/2 + generated stubs + four RPC types (unary + streaming).
- Efficient and strongly typed; consume as `Future` / `Stream`; pass auth via metadata; map errors/messages to domain.
- Web needs gRPC-Web (proxy); front behind repositories; choose per performance/typing/streaming needs.

Revision Notes

- Protobuf messages/service in `.proto` → `protoc` Dart stubs (typed); HTTP/2 transport.
- RPC types: unary (1→1), server-stream (1→N), client-stream (N→1), bidi (N↔N).
- `grpc` pkg: `ClientChannel` + generated client; `CallOptions.metadata` for auth; `GrpcError` codes; `shutdown()`.
- Web → gRPC-Web + proxy; map protobuf→entity/failure at repo; not for simple APIs.

Practice Questions

1. Name the four RPC types with a use case each.
2. Why is gRPC more efficient than REST/JSON?
3. Why does Flutter Web need gRPC-Web?

Coding Questions

1. Write a `.proto` with a unary and a server-streaming RPC.
2. Make a unary call with auth metadata and map `GrpcError` to a failure.
3. Consume a server-streaming RPC as a Dart `Stream` and cancel it.

Mini Project

gRPC product service (Flutter/Dart): Define `ProductService` (`GetProduct` unary + `WatchPrice` server-streaming) in `.proto`, generate stubs, and build a repository that returns a `Product` entity (unary) and a price `Stream` (streaming), passing auth metadata, mapping `GrpcError` →failures, and shutting down the channel. Acceptance: typed calls; auth via metadata; streaming consumed as `Stream`; errors mapped; entities at boundary; channel closed. (Note gRPC-Web requirement for web.)