Lecture 5: Networking & REST (Representational State Transfer)

HTTP (HyperText Transfer Protocol), JSON (JavaScript Object Notation), Clients, and the LLM Hallucination Trap

UniEats — fetching campus food spots from a real server

Syntax You'll Meet Today

Construct	Where you'll see it	Anchor
`suspend fun` (Kotlin)	Ktor `ApiClient` methods, Retrofit `SpotsApi`	like `async` in JS — marks a function that can pause without blocking a thread; full details in L06
`interface` + `@GET` / `@Query` annotations (Kotlin)	Retrofit `SpotsApi`	like Spring's `@GetMapping` / `@RequestParam`, but on the client side — Retrofit implements the interface for you
`sealed class` (Dart) / `enum` with payloads (Swift)	`AppError`, `ApiError`	like a Java `enum` but each case can carry its own data
`Future<T>` (Dart)	`dio.get(...)` return type	like a JS `Promise<T>` — a value that arrives later
`Result<T>` (Kotlin)	`apiCall` / `SpotRepository` return type	like a checked exception you can pass around as a value
`z.infer<typeof Schema>` (TypeScript)	Zod schema to type	reads the schema shape at compile time and creates a TypeScript type

The Hook — A True Production Story

A team shipped a food-delivery app. The LLM (AI code generator) wrote the API (Application Programming Interface) client. Tests passed.

What could go wrong?

- The AI invented an endpoint: `GET /restaurants/nearby/v2`
- The real server had: `GET /api/spots?q=&category=`
- The error path was empty — the app silently showed a blank list
- Users saw **nothing**. Support tickets flooded in on launch day

> "The code compiled. The mock tests passed. The field names were just… made up."

The Full Networking Pipeline — What We're Building Today

Everything in this lecture is one step of this pipeline. Learn it once; it works on all four stacks.

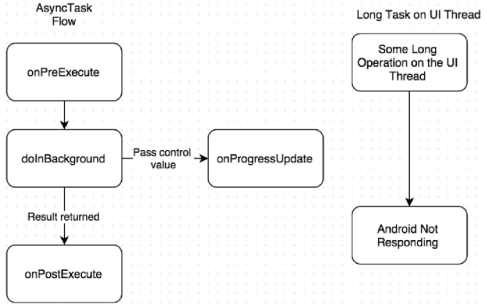
```
User taps "Refresh"
        ↓
ViewModel sets isLoading = true       ← UiState from L03
        ↓
Repository calls HTTP Client          ← Part 1 & 3 today
        ↓
HTTP Client sends GET /api/spots      ← HTTP verbs & status codes
        ↓
Server responds: 200 OK + JSON body   ← { "spots": [...] }
        ↓
Serializer parses JSON → Spot objects ← Part 2 today
        ↓
Repository returns Result<SpotsPage>
        ↓
ViewModel sets spots or errorMessage
        ↓
Composable / View / Widget re-renders ← the UI = f(state) from L02
```

> **If any step fails** (no network, 404, wrong field name) → the error path handles it → Part 4 today.

Part 1 — HTTP (HyperText Transfer Protocol) on Mobile

HTTP (HyperText Transfer Protocol) on Mobile — The Mental Model

```
App → HTTP Client → TLS Handshake → TCP → Server
            ↓
       Request:  GET /api/spots?page=1&limit=20
                 Accept: application/json
            ↓
       Response: 200 OK
                 Content-Type: application/json
                 { "spots": [...], "page": 1,
                   "hasNextPage": true }
```



Mobile constraints that do not exist on the web:

- Network can disappear mid-request (tunnel, elevator, subway)
- Radio wake-up cost — batching requests saves battery
- The OS can pause your app during a long request
- Cellular vs Wi-Fi — different speed and reliability
- Never call the network on the **main thread** — Android shows an **ANR** (Application Not Responding), as in the picture

HTTP (HyperText Transfer Protocol) Verbs & Status Codes — The Ones That Matter

Verbs for REST (Representational State Transfer)

Verb	Use
`GET`	Read, idempotent, cacheable
`POST`	Create, not idempotent
`PUT`	Full replace
`PATCH`	Partial update
`DELETE`	Remove

Status codes to handle in every client

Code	Meaning
200–204	Success (parse body)
400	Bad request — your bug
401	Unauthenticated — refresh token
403	Forbidden — do not retry
404	Not found
429	Rate limited — back off
5xx	Server error — retry with backoff

The UniEats REST (Representational State Transfer) Server

Node.js + Express in `unieats/server` · `cd unieats/server && npm ci && npm start` → `http://localhost:3000`

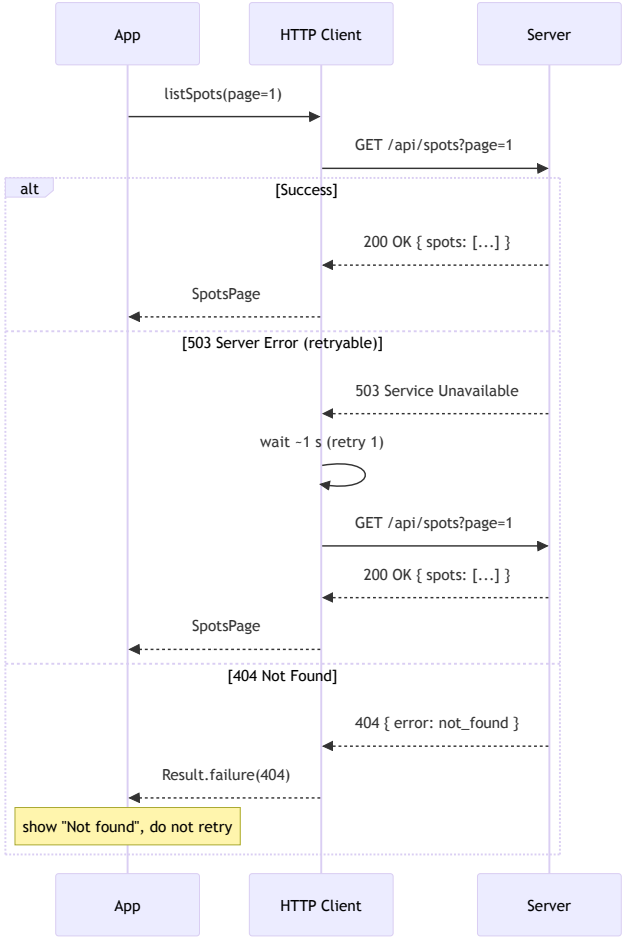
```
GET    /api/health             → { ok: true }
GET    /api/spots?page=1&limit=20&q=&category=
       → { spots, page, hasNextPage }
GET    /api/spots/:id          → Spot (404 if unknown)
POST   /api/spots              → create (201)
PATCH  /api/spots/:id          → partial update (200)
DELETE /api/spots/:id          → remove (204)
GET    /api/spots/:id/reviews  → Review[]
POST   /api/spots/:id/reviews  → { stars, text } (201)
```

- 25 seeded spots `spot-1` … `spot-25` → page 1 has 20, page 2 has 5
- Data lives in memory: restarting the server resets the 25 seeded spots
- Mutations are **open** at this tag — login + JWT arrive in L08
- Errors: always `{ "error": { "code", "message" } }`

Response shape (`GET /api/spots/spot-1`):


```
{
  "id": "spot-1",
  "name": "Central Canteen",
  "category": "canteen",
  "rating": 3.8,
  "priceLevel": 1,
  "lat": 44.4268,
  "lng": 26.1025,
  "openNow": true,
  "photoUrl":
    "https://picsum.photos/seed/spot-1/400/300",
  "description":
    "The main campus canteen with daily hot meals.",
  "updatedAt": 1700000000000
}
```

REST (Representational State Transfer) Request/Response Flow with Retry



Part 2 — JSON (JavaScript Object Notation) Serialization

Four ways to turn bytes into objects

JSON Serialization — Android: kotlinx.serialization

`data/model/Models.kt` @ `l05-rest`

```
@Serializable
data class Spot(
    val id: String,
    val name: String,
    val category: Category,
    val rating: Double,
    val priceLevel: Int,
    val lat: Double, val lng: Double,
    val openNow: Boolean,
    val photoUrl: String = "",
    val description: String = "",
    val updatedAt: Long
)

@Serializable
data class SpotsPage(
    val spots: List<Spot>,
    val page: Int,
    val hasNextPage: Boolean
)
```

What to notice

- `@Serializable` makes the Kotlin compiler plugin generate the parser — no reflection, nothing to maintain
- The property name **is** the JSON key: `photoUrl`, not `photo_url` — rename it and the parse breaks silently
- `= ""` defaults: a missing key falls back to the default instead of crashing
- The client's `Json { ignoreUnknownKeys = true }` skips keys the class does not declare
- `SpotsPage` mirrors the wrapper `{ spots, page, hasNextPage }` — never decode straight into `List<Spot>`
- Key differs from the name? `@SerialName("photo_url") val photoUrl: String`

JSON Serialization — iOS: Codable

`Models/Spot.swift` @ `l05-rest`

```
struct SpotDTO: Codable, Sendable {
    let id: String
    let name: String
    let category: String
    let rating: Double
    let priceLevel: Int
    let lat: Double; let lng: Double
    let openNow: Bool
    let photoUrl: String
    let description: String
    let updatedAt: Double
}

struct SpotsPage: Codable, Sendable {
    let spots: [SpotDTO]
    let page: Int
    let hasNextPage: Bool
}

let page = try JSONDecoder()
    .decode(SpotsPage.self, from: data)
```

What to notice

- `Codable` = `Encodable` + `Decodable`; the compiler synthesises the decoder when property names match the keys
- No defaults here: a missing `photoUrl` key throws a `DecodingError` — louder than Android, which is good
- `SpotDTO` is the wire shape; the app maps it to its own `Spot` model, so the UI never depends on the server's naming
- `Sendable` means the value can cross threads safely — Swift concurrency, see L06
- Key differs from the name? Add a `CodingKeys` enum, or set `decoder.keyDecodingStrategy = .convertFromSnakeCase`

JSON Serialization — Flutter: hand-written `fromJson`

`lib/domain/models.dart` @ `l05-rest`

```
factory Spot.fromJson(Map<String, dynamic> json) =>
    Spot(
  id: json['id'] as String,
  name: json['name'] as String,
  category: SpotCategory.values
      .byName(json['category'] as String),
  rating: (json['rating'] as num).toDouble(),
  priceLevel: (json['priceLevel'] as num).toInt(),
  lat: (json['lat'] as num).toDouble(),
  lng: (json['lng'] as num).toDouble(),
  openNow: json['openNow'] as bool,
  photoUrl: json['photoUrl'] as String? ?? '',
  description: json['description'] as String? ?? '',
  updatedAt: (json['updatedAt'] as num).toInt(),
);
```

What to notice

- `jsonDecode` gives you a `Map<String, dynamic>`; every field is a cast you write by hand
- `as num` then `.toDouble()`: JSON `3` arrives as an `int`, JSON `3.8` as a `double` — casting straight to `double` can throw
- `as String? ?? ''` is the Dart spelling of Android's default value
- A typo in `json['photoUrl']` compiles fine and fails at runtime — write a test with a real response
- Industry alternative: `json_serializable` / `freezed` generate this code (`build_runner`); UniEats stays hand-written so there is no code-generation step to learn today

JSON Serialization — React Native: Zod

`src/domain/schemas.ts` @ `l05-rest`

```ts
export const SpotSchema: z.ZodType<Spot> = z.object({
  id: z.string(),
  name: z.string(),
  category: SpotCategorySchema,
  rating: z.number(),
  priceLevel: z.number().int(),
  lat: z.number(),
  lng: z.number(),
  openNow: z.boolean(),
  photoUrl: z.string(),
  description: z.string(),
  updatedAt: z.number().int(),
});

export const SpotPageSchema = z.object({
  spots: z.array(SpotSchema),
  page: z.number().int(),
  hasNextPage: z.boolean(),
});
export type SpotPage = z.infer<typeof SpotPageSchema>;
```

What to notice

- TypeScript types are **erased** at runtime — `response.json() as Spot` checks nothing
- Zod is a runtime schema: `SpotPageSchema.safeParse(body)` returns `{ success, data | error }` and never throws
- `z.infer<typeof SpotPageSchema>` derives the TypeScript type from the schema, so the two cannot drift apart
- A renamed key is a parse error with a readable path, e.g. `spots[0].photoUrl: Required`
- The schema **is** the contract: one place to look when the server changes

Part 3 — HTTP (HyperText Transfer Protocol) Clients

The Four Dialects side by side

HTTP Clients — Android: Ktor (1/2) — one client for the app

`di/AppModule.kt` @ `l05-rest`

```
HttpClient(OkHttp) {
    expectSuccess = true
    defaultRequest { url(baseUrl) }
    install(ContentNegotiation) { json(json) }
    install(HttpTimeout) {
        connectTimeoutMillis = 10_000
        requestTimeoutMillis = 15_000
    }
    install(HttpRequestRetry) {
        retryOnServerErrors(maxRetries = 2)
        retryOnException(
            maxRetries = 2, retryOnTimeout = true)
        exponentialDelay(baseDelayMs = 500)
    }
}
```

`baseUrl` = `http://10.0.2.2:3000/api/` — the emulator's name for your laptop; a real phone needs the laptop's LAN IP

What to notice

- Built **once** in `AppModule` and injected by Hilt — never create a client per request
- `HttpClient(OkHttp)` : Ktor is the API, OkHttp is only the engine underneath
- Every cross-cutting concern is a plugin you `install` : JSON, timeouts, retry
- `expectSuccess = true` : any non-2xx answer throws a `ResponseException` instead of silently decoding an error body
- `HttpRequestRetry` : 1 try + 2 retries on 5xx and on transport errors, about 1 s then 2 s back-off — never on 4xx
- `defaultRequest` sets the base URL, so endpoints are relative: `"spots"`

HTTP Clients — Android: Ktor (2/2) — endpoint and repository

`data/remote/ApiClient.kt`

```
suspend fun listSpots(
    page: Int, query: String, category: Category?
): SpotsPage =
    http.get("spots") {
        parameter("page", page)
        parameter("limit", PAGE_SIZE)
        if (query.isNotBlank())
            parameter("q", query.trim())
        if (category != null)
            parameter("category", category.apiValue)
    }.body()
```

`data/repository/SpotRepository.kt`

```
suspend fun fetchSpots(
    page: Int, query: String, category: Category?
): Result<SpotsPage> =
    apiCall { api.listSpots(page, query, category) }
```

What to notice

- `http.get("spots") { ... }` is a **DSL**: you write the request imperatively, parameter by parameter
- `suspend fun` : the call can pause without blocking the UI thread — full story in L06
- `.body()` picks the type from the declared return type `SpotsPage` and asks kotlinx.serialization to decode it
- The repository wraps every call in `apiCall { }` and returns `Result<SpotsPage>` — the ViewModel never sees an exception
- Three layers, three jobs: client = HTTP, `ApiClient` = endpoints, repository = errors as values

HTTP Clients — Android: Retrofit (1/3) — the API as an interface

UniEats uses Ktor, but most Android codebases and job listings say **Retrofit** (Square, 2013). The same UniEats contract, written the Retrofit way:

```
interface SpotsApi {

    @GET("spots")
    suspend fun listSpots(
        @Query("page") page: Int,
        @Query("limit") limit: Int = PAGE_SIZE,
        @Query("q") query: String? = null,
        @Query("category") category: String? = null
    ): SpotsPage

    @GET("spots/{id}")
    suspend fun getSpot(@Path("id") id: String): Spot

    @POST("spots/{id}/reviews")
    suspend fun addReview(
        @Path("id") id: String,
        @Body review: NewReview
    ): Review
}
```

What to notice

- You **describe** the endpoint, you never implement it: Retrofit generates the implementation at runtime from the annotations
- `@GET("spots")` + `@Query` ↔ `http.get("spots") { parameter(...) }` in Ktor
- A `null` `@Query` is simply left out of the URL — same effect as Ktor's `if` guards
- `@Path("id")` fills the `{id}` placeholder; `@Body` serialises the request body
- `suspend fun` + a plain return type: Retrofit runs the call off the main thread and decodes `SpotsPage` for you
- Same models, same wrapper rule: `SpotsPage`, not `List<Spot>`
- Older code returns `Call<T>`: `enqueue()` runs async, `execute()` blocks — never call it on the main thread

HTTP Clients — Android: Retrofit (2/3) — build it once, handle errors

```
val okHttp = OkHttpClient.Builder()
    .connectTimeout(10, TimeUnit.SECONDS)
    .readTimeout(15, TimeUnit.SECONDS)
    .addInterceptor(HttpLoggingInterceptor()
        .setLevel(HttpLoggingInterceptor.Level.BASIC))
    .build()

val retrofit = Retrofit.Builder()
    .baseUrl("http://10.0.2.2:3000/api/")  // must end in /
    .client(okHttp)
    .addConverterFactory(
        json.asConverterFactory(
            "application/json".toMediaType()))
    .build()

val api: SpotsApi = retrofit.create()
```

Real Retrofit code in the course repo: `lectures/14/ComposeMovieApp` — `movies/service/MoviesService.kt`, `di/AppModule.kt`, `movies/repo/HandleErrors.kt` (Gson converter + Hilt)

What to notice

- Retrofit sits **on top of OkHttp**: timeouts, logging, auth headers and retries are OkHttp interceptors, not Retrofit plugins
- `baseUrl` **must end in** `/` and the paths must **not** start with one, or the builder throws / the path replaces the `/api` segment
- The converter decides the JSON library: `converter-kotlinx-serialization` here; Gson or Moshi in older code
- Errors are exceptions, like Ktor: a non-2xx answer throws `retrofit2.HttpException` (`code()`, `response()?.errorBody()`), a network failure throws `IOException`
- No built-in retry plugin — put back-off in an OkHttp interceptor or in the repository
- Wrap calls in the same `apiCall { }` → `Result<T>` as the Ktor version

Ktor vs Retrofit (3/3) — same contract, two dialects

	Ktor (UniEats)	**Retrofit** (most codebases)
Style	imperative DSL: `http.get("spots") { … }`	declarative: `interface` + annotations
Who writes the call	you, in `ApiClient`	Retrofit, from the interface
Timeouts / JSON / retry	Ktor plugins (`install(...)`)	OkHttp + interceptors; converter; no retry plugin
Non-2xx answer	`ResponseException` (`expectSuccess = true`)	`HttpException` , or check `Response<T>.isSuccessful`
Network failure	`IOException`	`IOException`
Platforms	Kotlin Multiplatform (Android, iOS, desktop, web)	JVM / Android only
Engine	OkHttp on Android (Darwin on iOS, CIO…)	OkHttp, always
Pick it when	you share code with iOS (L12)	you join an existing Android codebase

> **Both hit the same server.** A hallucinated `/api/places` is a 404 in both. The `curl` rule does not care which client you chose.

HTTP Clients — iOS: URLSession + async/await

`Network/ApiClient.swift` @ `l05-rest`

```
func listSpots(page: Int, query: String,
    category: String?) async throws -> SpotsPage {
  var items = [
    URLQueryItem(name: "page", value: String(page)),
    URLQueryItem(name: "limit",
                 value: String(Self.pageSize)),
  ]
  if !query.isEmpty { items.append(
    URLQueryItem(name: "q", value: query)) }
  if let category { items.append(
    URLQueryItem(name: "category", value: category)) }
  return try await get("spots", query: items)
}
```

Own `URLSession`: 15 s request / 30 s resource timeout; base `http://localhost:3000/api`

What to notice

- `URLSession` ships with the OS — no third-party client needed on iOS
- `async throws` + `try await`: Swift's spelling of `suspend fun`; errors are thrown, not returned
- `URLQueryItem` builds the query string safely — no hand-concatenated `?q=` strings
- The shared `get` / `perform` helpers check the status code **before** decoding — otherwise a 404 body becomes a confusing `DecodingError`
- Timeouts are set on the session, not per call; the simulator shares the Mac's network, so `localhost` works; a real iPhone needs the Mac's LAN IP
- No automatic retry: the error views offer a **Retry** button instead

HTTP Clients — Flutter: Dio

`lib/data/network/api_client.dart` @ `l05-rest`

```dart
Dio createDio(String baseUrl) {
  final dio = Dio(
    BaseOptions(
      baseUrl: baseUrl,
      connectTimeout: const Duration(seconds: 10),
      receiveTimeout: const Duration(seconds: 15),
      sendTimeout: const Duration(seconds: 15),
      contentType: Headers.jsonContentType,
    ),
  );
  dio.interceptors.add(RetryInterceptor(dio));
  return dio;
}
```

`ApiClient.fetchSpots(page:, limit:, query:, category:)` → `SpotsPage`; every `DioException` becomes an `AppError`

What to notice

- Dart ships a bare `http` package; **Dio** adds base URL, timeouts, interceptors and typed exceptions — it is the Retrofit/Ktor of Flutter
- Dio has **no default timeout**: without `BaseOptions` a request can hang forever (pitfall 6 later today)
- `interceptors` are the plugin point: logging, auth headers, retry — like OkHttp interceptors
- `RetryInterceptor`: 3 retries with exponential back-off, GET only, on timeouts, dropped connections and 5xx
- `dio.get(...)` returns a `Future<Response>`; the client hands `response.data` to `Spot.fromJson`
- Base URL is `10.0.2.2` on the Android emulator and `localhost` elsewhere

HTTP Clients — React Native: `fetch` + Zod + TanStack Query

`src/api/client.ts` @ `l05-rest`

```ts
export function listSpots(params: {
  page: number; q: string;
  category: SpotCategory | "all"; signal?: AbortSignal;
}): Promise<SpotPage> {
  const search = new URLSearchParams({
    page: String(params.page),
    limit: String(PAGE_SIZE),
  });
  if (params.q) search.set("q", params.q);
  if (params.category !== "all")
    search.set("category", params.category);
  return getJson(`/spots?${search.toString()}`,
    SpotPageSchema, params.signal);
}
```

Key pattern across all stacks: typed async call → serialization step → state container (StateFlow / @Observable / Riverpod `AsyncNotifier` / TanStack Query)

What to notice

- Plain `fetch` — the same API as in the browser; `axios` is the common third-party alternative
- `fetch` has **no timeout**: `getJson` adds a 10 s `AbortController` and passes the `signal` through
- `URLSearchParams` builds the query string safely, like `URLQueryItem` on iOS
- Every body goes through a Zod schema in `getJson` — the serialization step is not optional
- **TanStack Query** sits on top (`useSpotsInfinite` , on the pagination slide): cache, refetch, retry up to twice, never on a 4xx
- Returns a `Promise<SpotPage>` — the JS spelling of `suspend` / `async throws` / `Future`

Spot the AI Bug 🐞

Can you find the bug in this LLM (AI)-generated Ktor client?

Spot the AI Bug 🐞 — The Code

An AI generated this Ktor client for UniEats. There are two bugs — both detectable from the server contract shown earlier in this lecture.

```
// AI-generated ApiClient.kt
class ApiClient(private val httpClient: HttpClient) {

    // Bug A: look at the path — is this the real UniEats endpoint?
    suspend fun listSpots(page: Int = 1): List<Spot> =
        httpClient.get("http://localhost:3000/api/places") {
            parameter("page", page)
        }.body()

    suspend fun getSpot(id: String): Spot =
        httpClient.get("http://localhost:3000/api/places/$id")
            .body()
}
```

Question: What is wrong? Look at the server contract on the earlier slide — think for 60 seconds.

Spot the AI Bug 🐞 — Revealed

Bug A: Hallucinated endpoint path

```
// WRONG — AI invented /api/places; real server has /api/spots
httpClient.get("http://localhost:3000/api/places") { ... }

// RIGHT — match the server contract exactly
httpClient.get("http://localhost:3000/api/spots") { ... }
```

The request returns **404 Not Found**. Without error handling the app shows a blank list — no crash, no log warning.

Bug B: Wrong return type — bare list vs. wrapped object

The server returns `{ "spots": [...], "page": 1, "hasNextPage": true }`, not a bare array.

```
// WRONG — Ktor cannot deserialize an object into List<Spot>
suspend fun listSpots(): List<Spot> = httpClient.get(...).body()

// RIGHT — wrap it to match the real JSON shape (data/model/Models.kt)
@Serializable
data class SpotsPage(val spots: List<Spot>, val page: Int, val hasNextPage: Boolean)

suspend fun listSpots(page: Int, query: String, category: Category?): SpotsPage =
    http.get("spots") { parameter("page", page) /* + limit, q, category */ }.body()
```

Rule: Always `curl` the real endpoint and check the JSON shape before writing the data class.

Part 4 — Error Handling

Structured error handling across the four stacks

Typed Errors — Android: `Result` + `toUserMessage()`

`data/remote/ApiResult.kt` @ `l05-rest`

```
suspend fun <T> apiCall(
    block: suspend () -> T
): Result<T> =
    try {
        Result.success(block())
    } catch (e: CancellationException) {
        throw e
    } catch (e: Exception) {
        Result.failure(e)
    }

fun Throwable.toUserMessage(): String =
    when (this) {
        is HttpRequestTimeoutException ->
            "The server took too long to answer."
        is ResponseException ->
            if (response.status.value == 404)
                "Not found on the server."
            else "The server answered " +
                "${response.status.value}. Try again."
        is SerializationException -> "The server " +
            "sent a response the app could not read."
        is IOException -> "Can't reach the " +
            "server. Is it running on port 3000?"
        else -> "Something went wrong " +
            "(${this::class.simpleName})."
    }
```

What to notice

- Errors as **values**: `apiCall` turns any exception into `Result.failure`, so the ViewModel handles success and failure explicitly
- `CancellationException` is rethrown — coroutine cancellation is not an error (L06)
- Ktor already throws one class per failure category: `ResponseException` (server said no), `IOException` (could not reach it), `HttpRequestTimeoutException`, `SerializationException`
- `when` maps the class to a message — **order matters**: the timeout exception *is* an `IOException`, so it must come first
- With Retrofit, swap `ResponseException` for `HttpException` — the rest is identical

Typed Errors — iOS: an `ApiError` enum

`Network/ApiError.swift` , `ApiClient.swift` @ `l05-rest`

```
enum ApiError: Error, LocalizedError {
    case http(statusCode: Int, message: String)
    case noConnectivity
    case timeout
    case decodingFailed(Error)
}

do {
    (data, response) =
        try await session.data(for: request)
} catch let error as URLError
            where error.code == .cancelled {
    throw CancellationError()
} catch let error as URLError
            where error.code == .timedOut {
    throw ApiError.timeout
} catch is URLError {
    throw ApiError.noConnectivity
}
guard let http = response as? HTTPURLResponse
    else { throw ApiError.noConnectivity }
guard (200..<300).contains(http.statusCode) else {
    let message = (try? JSONDecoder().decode(
        ErrorEnvelope.self, from: data))?.error.message
    throw ApiError.http(statusCode: http.statusCode,
        message: message ?? HTTPURLResponse
            .localizedString(
                forStatusCode: http.statusCode))
}
```

What to notice

- An `enum` with payloads: each case carries its own data (`statusCode` , the wrapped `Error`) — the Swift sealed type
- `URLSession` only throws `URLError` ; the client **translates** it into the app's own categories by `error.code`
- Cancellation is passed through as `CancellationError` , exactly like Android's rethrow
- A non-2xx status is not an exception in `URLSession` — the `guard` makes it one, and reads the server's `{ error: { message } }` envelope first
- The view model catches `ApiError` and shows `localizedDescription` ; the four cases drive four different UIs

Typed Errors — Flutter: a `sealed class AppError`

`lib/data/network/app_error.dart` @ `l05-rest`

```
sealed class AppError implements Exception {
  const AppError();
  String get message;
}
class NoConnectivity extends AppError { … }
class TimeoutError extends AppError { … }
class HttpError extends AppError {
  const HttpError(this.statusCode,
      [this.serverMessage]);
  final int statusCode;
  final String? serverMessage;
  …
}
class BadResponse extends AppError { … }

AppError mapDioError(DioException e) =>
    switch (e.type) {
  DioExceptionType.connectionTimeout ||
  DioExceptionType.sendTimeout ||
  DioExceptionType.receiveTimeout =>
    const TimeoutError(),
  DioExceptionType.connectionError =>
    const NoConnectivity(),
  DioExceptionType.badResponse => HttpError(
    e.response?.statusCode ?? -1,
    _serverMessage(e.response?.data),
  ),
  DioExceptionType.unknown
      when e.error is FormatException =>
    const BadResponse(),
  _ => const NoConnectivity(),
};
```

What to notice

- Dart 3 `sealed class` : the compiler knows every subclass, so a `switch` over `AppError` must cover all cases — miss one and you get a warning
- Dio already classifies failures by `DioExceptionType` ; `mapDioError` folds them into the app's four categories
- `badResponse` = the server answered with an error; the status code and the server's message travel inside `HttpError`
- Each `AppError` carries the `message` the UI shows — the widget never inspects a `DioException`
- Same four buckets as Android and iOS: unreachable, timed out, server said no, unreadable body

Typed Errors — React Native: `ApiError` with a `kind`

`src/api/errors.ts`, `client.ts` @ `l05-rest`

```
export type ApiErrorKind =
  "http" | "network" | "timeout" | "parse";

export class ApiError extends Error {
  constructor(
    readonly kind: ApiErrorKind,
    message: string,
    readonly status?: number,
    readonly code?: string,
  ) { super(message); … }
}

// inside getJson(path, schema, signal)
try {
  response = await fetch(`${API_URL}${path}`, …);
} catch (e) {
  if (controller.signal.aborted && !signal?.aborted)
    throw new ApiError("timeout",
      `GET ${path} timed out`);
  throw new ApiError("network", …);
}
if (!response.ok)
  throw new ApiError("http",
    `GET ${path} → ${response.status}`,
    response.status, code);
const parsed = schema.safeParse(body);
if (!parsed.success)
  throw new ApiError("parse",
    parsed.error.message, response.status);
```

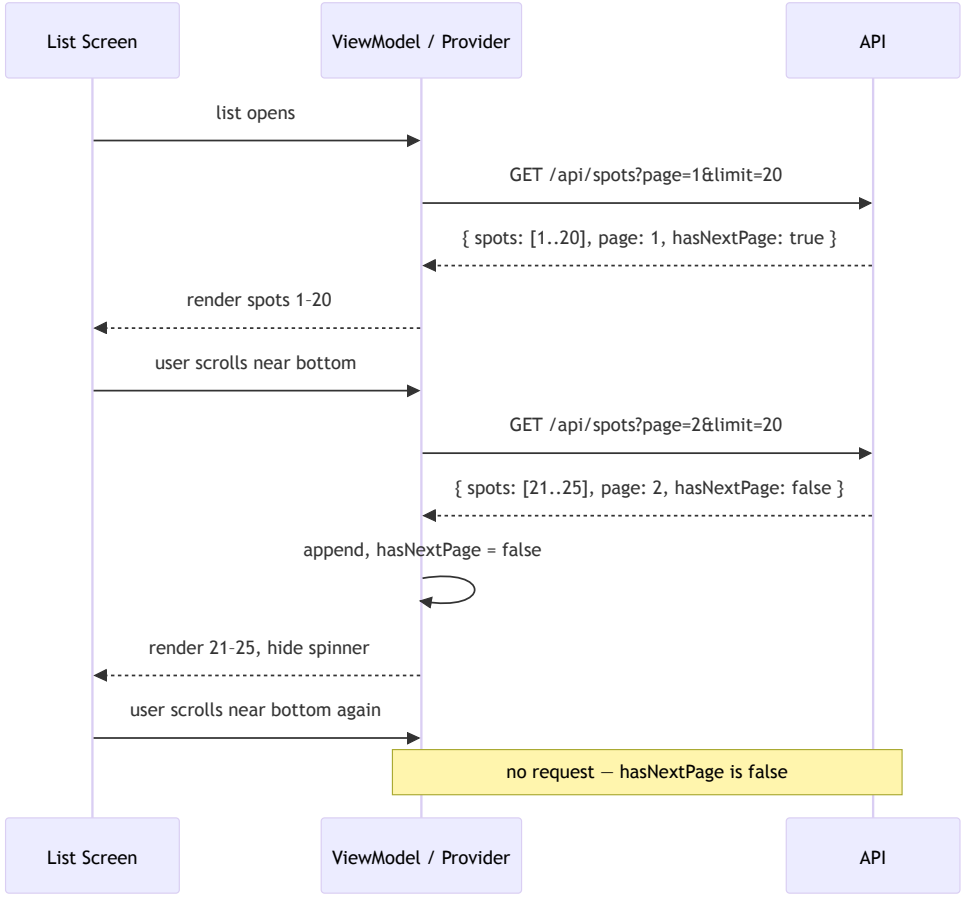
What to notice

- `kind` is a union of string literals — a `switch (error.kind)` is checked by the compiler, like a sealed class
- `fetch` rejects only on network failure; `response.ok` is false for 4xx/5xx and **nobody throws** unless you do
- Two different aborts: our own 10 s timer → `"timeout"`; the caller's signal → a cancellation, not an error
- The Zod `safeParse` failure becomes a `"parse"` error — a renamed field is reported, not swallowed
- Thrown, not returned: TanStack Query expects a rejected promise and exposes it as `query.error`

Part 5 — Pagination

Offset-based pagination across all four stacks

Pagination Flow



Pagination — Start on the Server: `GET /api/spots?page=`

`unieats/server/src/routes/spots.js`

```
router.get('/', (req, res) => {
  const page  = Math.max(1, parseInt(req.query.page, 10) || 1);
  const limit = Math.max(1, parseInt(req.query.limit, 10) || 20);
  const q = String(req.query.q || '').toLowerCase().trim();
  const category = String(req.query.category || '')
    .toLowerCase().trim();

  let results = store.getSpots();
  if (q) results = results.filter((s) =>
    s.name.toLowerCase().includes(q) ||
    s.description.toLowerCase().includes(q));
  if (category) results = results.filter(
    (s) => s.category === category);

  const start = (page - 1) * limit;
  return res.json({
    spots: results.slice(start, start + limit),
    page,
    hasNextPage: start + limit < results.length,
  });
});
```

What to notice

- **Offset pagination**: page `n` is the slice `[(n-1)·limit, n·limit)` of the filtered list
- Filters run **before** slicing — so a new search or category means a new sequence that starts again at page 1
- `hasNextPage` is computed by the server; the client never guesses from `spots.length`
- 25 seeded spots → page 1 has 20, page 2 has 5, page 3 is `[]` with `hasNextPage: false`
- Missing or bad `page` defaults to 1, bad `limit` to 20 — the client still always sends both

```
curl -s 'localhost:3000/api/spots?page=2' \
  | jq '{page, hasNextPage, n: (.spots | length)}'
# { "page": 2, "hasNextPage": false, "n": 5 }
```

Pagination — The State One Screen Needs

`ui/spotlist/SpotListViewModel.kt` @ `l05-rest`

```
data class SpotListUiState(
    val spots: List<Spot> = emptyList(),
    val searchQuery: String = "",
    val categoryFilter: Category? = null,
    val favouriteIds: Set<String> = emptySet(),
    val isLoading: Boolean = false,
    val isRefreshing: Boolean = false,
    val isLoadingMore: Boolean = false,
    val errorMessage: String? = null,
    val currentPage: Int = 0,
    val hasNextPage: Boolean = false
)
```

Same fields, different spelling, on the other stacks: iOS `@Observable SpotListViewModel` properties, Flutter `SpotPages`, React Native `query.data.pages` + `query.hasNextPage` / `isFetchingNextPage`

What to notice

- `spots` is the **accumulated** list — every page is appended, never replaced, until a filter changes
- `currentPage` + `hasNextPage` are the cursor: the next request is `currentPage + 1`, and only while `hasNextPage`
- Three loading flags because the UI shows three different things: full-screen spinner (`isLoading`), pull-to-refresh indicator (`isRefreshing`), footer spinner under the list (`isLoadingMore`)
- `errorMessage` with `spots` already on screen → snackbar with Retry, not a blank error screen
- `currentPage = 0` at start: nothing loaded yet, so the first load is "page 1", not "next page"

Pagination — Android: `loadFirstPage()` restarts the sequence

```
fun onSearchQueryChange(query: String) {
    _uiState.update { it.copy(searchQuery = query) }
    loadFirstPage(debounceMs = SEARCH_DEBOUNCE_MS)   // 300 ms
}
fun onCategorySelected(category: Category?) {
    _uiState.update { it.copy(categoryFilter = category) }
    loadFirstPage()
}
fun refresh() = loadFirstPage(refreshing = true)

private fun loadFirstPage(debounceMs: Long = 0,
                          refreshing: Boolean = false) {
    loadJob?.cancel()
    loadJob = viewModelScope.launch {
        delay(debounceMs)
        _uiState.update { it.copy(isLoading = !refreshing,
            isRefreshing = refreshing, errorMessage = null) }
        val state = _uiState.value
        spotRepository.fetchSpots(1, state.searchQuery, state.categoryFilter)
            .onSuccess { page ->
                _uiState.update { it.copy(
                    spots = page.spots,            // replace
                    currentPage = 1,
                    hasNextPage = page.hasNextPage,
                    isLoading = false, isRefreshing = false) }
            }
            .onFailure { error ->
                retryFailedLoad = { loadFirstPage() }
                _uiState.update { it.copy(isLoading = false,
                    isRefreshing = false,
                    errorMessage = error.toUserMessage()) }
            }
    }
}
```

What to notice

- Every filter change goes through **one** function that always asks for page 1 and **replaces** `spots`
- `loadJob?.cancel()` : typing "piz" then "pizz" cancels the in-flight request for "piz" — otherwise an older, slower response could overwrite a newer one
- `delay(300)` inside the cancelled job **is** the search debounce: no timer, no extra state
- `refreshing` only chooses which spinner to show; the request is identical
- The retry action is remembered as a lambda, so the snackbar's Retry repeats exactly the load that failed

Pagination — Android: `loadNextPage()` — append and advance

```
private var loadJob: Job? = null

fun loadNextPage() {
    val state = _uiState.value
    if (!state.hasNextPage || loadJob?.isActive == true) return
    val nextPage = state.currentPage + 1
    loadJob = viewModelScope.launch {
        _uiState.update { it.copy(isLoadingMore = true,
                                  errorMessage = null) }
        spotRepository.fetchSpots(
            nextPage, state.searchQuery, state.categoryFilter)
            .onSuccess { page ->
                _uiState.update { it.copy(
                    spots = (it.spots + page.spots)
                        .distinctBy(Spot::id),       // append
                    currentPage = nextPage,
                    hasNextPage = page.hasNextPage,
                    isLoadingMore = false) }
            }
            .onFailure { error ->
                retryFailedLoad = ::loadNextPage
                _uiState.update { it.copy(isLoadingMore = false,
                    errorMessage = error.toUserMessage()) }
            }
    }
}
```

What to notice

- The **guard** on line one is the whole protocol: stop when the server said `hasNextPage = false`, and never run two loads at once
- `nextPage` is computed from the state **before** launching, then written back only on success — a failed page 2 leaves `currentPage` at 1, so Retry asks for page 2 again
- `it.spots + page.spots`: append to the list already on screen; `distinctBy(Spot::id)` protects against an item that moved between pages — a duplicate `key` would crash `LazyColumn`
- `isLoadingMore` drives the footer spinner; `errorMessage` with content on screen is a snackbar — the list never disappears
- Same `fetchSpots` as the first page, same filters, different page number

Pagination — Android: The Trigger — `InfiniteScrollEffect`

`ui/spotlist/SpotListScreen.kt` @ `l05-rest`

```
@Composable
private fun InfiniteScrollEffect(
    listState: LazyListState,
    state: SpotListUiState,
    onLoadMore: () -> Unit
) {
    val nearEnd by remember {
        derivedStateOf {
            val total = listState.layoutInfo.totalItemsCount
            val lastVisible = listState.layoutInfo
                .visibleItemsInfo.lastOrNull()?.index ?: -1
            total > 0 && lastVisible >= total - 3
        }
    }
    LaunchedEffect(nearEnd, state.hasNextPage, state.spots.size) {
        if (nearEnd && state.hasNextPage &&
            state.errorMessage == null) onLoadMore()
    }
}

// in the LazyColumn, after the items:
if (state.isLoadingMore || state.isLoading) {
    item(key = "loading-more") { CircularProgressIndicator() }
}
```

What to notice

- `rememberLazyListState()` knows which items are on screen; `layoutInfo` is read **inside** `derivedStateOf`, so `nearEnd` only changes when the boolean flips, not on every pixel of scroll
- "Near the end" = the last visible row is within 3 of the end — the next page loads **before** the user hits the bottom, so the spinner is rarely seen
- `LaunchedEffect` re-runs when one of its keys changes: `nearEnd` flips, a page arrives (`spots.size`), or `hasNextPage` turns false
- The `errorMessage == null` guard stops an error from re-triggering a load forever while the snackbar is still up
- The spinner is a normal list `item` with its own `key` — it scrolls with the content

Pagination — iOS: `@Observable` view model + `.task` on each row

`ViewModels/SpotListViewModel.swift`, `Views/SpotListView.swift` @ `l05-rest`

```
func loadMoreIfNeeded(after spot: Spot) async {
    guard spot.id == spots.last?.id else { return }
    await loadMore()
}

func loadMore() async {
    guard hasNextPage, !isLoading, !isLoadingMore else { return }
    isLoadingMore = true
    defer { isLoadingMore = false }
    errorMessage = nil
    do {
        let page = try await api.listSpots(
            page: currentPage + 1,
            query: searchQuery, category: categoryFilter)
        let known = Set(spots.map(\.id))
        spots += page.spots.map(Spot.init(dto:))
            .filter { !known.contains($0.id) }
        currentPage = page.page
        hasNextPage = page.hasNextPage && !page.spots.isEmpty
    } catch is CancellationError {
    } catch {
        errorMessage = error.localizedDescription
    }
}

// SpotListView — inside ForEach(viewModel.spots):
SpotRow(spot: spot)
    .task { await viewModel.loadMoreIfNeeded(after: spot) }
```

What to notice

- The trigger is SwiftUI's `.task` on every row: it runs when the row **appears**, so the last row appearing is "the user reached the end"
- `loadMoreIfNeeded` checks `spot.id == spots.last?.id`, so only the last row's task actually loads
- `guard hasNextPage, !isLoading, !isLoadingMore` is the same guard as Android's first line
- `defer { isLoadingMore = false }` resets the flag on **every** exit — success, error, or cancellation
- Dedupe with a `Set` of known ids, like `distinctBy`; `currentPage = page.page` trusts the server's page number
- `catch is CancellationError {}` swallows cancellation, so a view disappearing never shows an error

Pagination — Flutter (1/2): `build()` is the first page

`lib/providers/providers.dart` @ `l05-rest`

```dart
class SpotPagesNotifier extends AsyncNotifier<SpotPages> {
  static const pageSize = 20;
  int _generation = 0;

  @override
  Future<SpotPages> build() async {
    _generation++;
    final (query, category) = ref.watch(
      spotListProvider.select(
        (s) => (s.searchQuery.trim(), s.categoryFilter)));
    if (query.isNotEmpty) {
      await Future<void>.delayed(
        const Duration(milliseconds: 300));
    }
    final first = await ref.read(apiClientProvider).fetchSpots(
      limit: pageSize, query: query,
      category: category?.name);
    return SpotPages(
      query: query, category: category,
      spots: first.spots, page: first.page,
      hasNextPage: first.hasNextPage);
  }
}
```

What to notice

- `AsyncNotifier.build()` **is** the first-page load — there is no `loadFirstPage()` to write
- `ref.watch(... .select(...))` subscribes to the search text and category: when either changes, Riverpod **re-runs** `build()` from page 1 and disposes the old one — Flutter's spelling of `loadJob.cancel()`
- The 300 ms debounce is a plain `delayed`; a newer `build()` replaces the older one before its request lands
- `SpotPages` is an immutable value: the accumulated `spots`, the cursor (`page`, `hasNextPage`) and the filters it was loaded for
- `_generation` counts rebuilds — the next slide uses it to drop stale page-2 results

Pagination — Flutter (2/2): `loadNextPage()` + a `ScrollController`

```dart
Future<void> loadNextPage() async {
  final current = state.valueOrNull;
  if (current == null || state.isLoading ||
      !current.hasNextPage || current.isLoadingMore) return;
  final generation = _generation;
  state = AsyncData(_copy(current, isLoadingMore: true));
  try {
    final next = await ref.read(apiClientProvider).fetchSpots(
      page: current.page + 1, limit: pageSize,
      query: current.query, category: current.category?.name);
    if (generation != _generation) return;
    state = AsyncData(SpotPages(
      query: current.query, category: current.category,
      spots: [...current.spots, ...next.spots],
      page: next.page, hasNextPage: next.hasNextPage));
  } on Exception catch (e) {
    if (generation != _generation) return;
    state = AsyncData(_copy(current, loadMoreError: e));
  }
}
```

`lib/ui/spot_list/spot_list_screen.dart`

```dart
void _onScroll() {
  final position = _scrollController.position;
  if (position.pixels >= position.maxScrollExtent - 300) {
    ref.read(spotPagesProvider.notifier).loadNextPage();
  }
}
```

What to notice

- Same guard as Android: no next page, or already loading → return
- The state stays `AsyncData` while loading more, so the list **stays on screen**; `isLoadingMore` and `loadMoreError` travel inside the value and the footer widget renders them
- `[...current.spots, ...next.spots]` appends; `page` and `hasNextPage` come from the server's answer
- `generation != _generation` : if the filters changed while page 2 was in flight, the result is dropped instead of being appended to the wrong list
- Trigger: a `ScrollController` listener in the screen — within 300 px of `maxScrollExtent` , ask for more; the guard makes it safe to call on every scroll event
- Retry in the footer calls `loadNextPage()` again

Pagination — React Native (1/2): `useInfiniteQuery`

`src/hooks/useSpots.ts` @ `l05-rest`

```
export function useSpotsInfinite(
  q: string, category: CategoryFilter
) {
  return useInfiniteQuery({
    queryKey: ["spots", q, category],
    queryFn: ({ pageParam, signal }) =>
      listSpots({ page: pageParam, q, category, signal }),
    initialPageParam: 1,
    getNextPageParam: (lastPage) =>
      lastPage.hasNextPage ? lastPage.page + 1 : undefined,
  });
}
```

What the hook gives back:

```
query.data?.pages         // SpotPage[] — one entry per page
query.hasNextPage         // false once getNextPageParam → undefine
query.isFetchingNextPage // footer spinner
query.isFetchNextPageError
query.fetchNextPage()     // deduped: a second call while loading i
```

What to notice

- TanStack Query owns the state: the array of pages, the cursor, the loading and error flags you wrote by hand on the other stacks
- `getNextPageParam` is the cursor rule: given the last page, return the next page number, or `undefined` to stop — the server's `hasNextPage` again
- `initialPageParam: 1` is "page 1 first"; `pageParam` is whatever the previous call returned
- The `queryKey` includes `q` and `category`: a new search is a **new cache entry** that starts at page 1, and switching back to an old search shows its cached pages instantly
- `signal` lets the query cancel an in-flight request when the key changes

Pagination — React Native (2/2): the screen — `flatMap`, `onEndReached`, footer

`app/(tabs)/spots/index.tsx` @ `l05-rest`

```
const q = useDebouncedValue(searchQuery.trim(), 300);
const query = useSpotsInfinite(q, categoryFilter);
const spots = useMemo(
  () => query.data?.pages.flatMap((p) => p.spots) ?? [],
  [query.data]);

const onEndReached = () => {
  if (query.hasNextPage && !query.isFetchingNextPage &&
      !query.isFetchNextPageError) query.fetchNextPage();
};

<FlatList
  data={spots}
  onEndReached={onEndReached}
  onEndReachedThreshold={0.5}
  ListFooterComponent={
    query.isFetchingNextPage
      ? <ActivityIndicator />
      : query.isFetchNextPageError
        ? <Pressable onPress={() => query.fetchNextPage()}>
            <Text>Couldn't load more. Tap to retry.</Text>
          </Pressable>
        : null
  }
/>
```

What to notice

- Debounce is a hook on the **input** (`useDebouncedValue`), so the query key only changes 300 ms after the user stops typing
- `flatMap` turns the array of pages into the one flat list `FlatList` needs; `useMemo` keeps the same array between renders
- `onEndReachedThreshold={0.5}` : fire when the end is within half a visible screen — the same idea as Android's three rows and Flutter's 300 px
- The guard in `onEndReached` mirrors the other stacks; the `isFetchNextPageError` check stops an automatic retry loop after a failure
- The footer renders straight from the query flags; Retry is just `fetchNextPage()` again

Pagination — The Same Six Decisions on Every Stack

Decision	Android	iOS	Flutter	React Native
Trigger	last visible row ≥ total − 3	`.task` on the last row	within 300 px of the end	`onEndReached` , 0.5 screens
Guard	`hasNextPage` and no job active	`guard hasNextPage, !isLoadingMore`	`hasNextPage` and not loading	`hasNextPage && !isFetchingNextPage`
Append	`spots + page.spots` , `distinctBy`	`spots +=` , filter known ids	`[...spots, ...next.spots]`	`data.pages` + `flatMap`
Stop	server's `hasNextPage`	`hasNextPage && !isEmpty`	server's `hasNextPage`	`getNextPageParam` → `undefined`
Filter change	cancel job, page 1, replace	cancel task, page 1	`build()` re-runs	new `queryKey`
Error loading more	snackbar + Retry, list stays	`errorMessage` , list stays	footer `loadMoreError`	footer `isFetchNextPageError`

> Offset pagination is the simplest form. Production APIs with large or fast-changing lists use a **cursor** (`?after=<id>`) instead of a page number — same client code, different `getNextPageParam` .

Part 6 — LLM Pitfalls in Networking Code

The hallucination patterns to watch for

LLM Networking Pitfalls — The Hit List

1. Hallucinated endpoints

> AI generates `GET /api/places/nearby` — the real server has `GET /api/spots?q=&category=`

2. Wrong field names

> Model class has `photoURL` — server sends `photoUrl` — silently empty at runtime

3. Empty error paths

> Generated code has no catch block — app crashes on any network failure

4. Bare arrays vs wrapped objects

> Expects `List<Spot>` — server returns `{ "spots": [...] }` — `JsonConvertException` (Ktor)

5. Missing authentication headers

> AI forgets the `Authorization: Bearer` header on POST/PATCH/DELETE (UniEats requires it from L08)

6. No timeout configuration

> Dio has no default timeout, URLSession waits 60 s — users wait forever

7. Synchronous network call on main thread

> AI calls `api.getSpots().execute()` without coroutine — NetworkOnMainThreadException

Verifying LLM-Generated API (Application Programming Interface) Code — Checklist

```
# Always do this before writing a single line of client code
curl -s http://localhost:3000/api/spots | jq .

# Check field names exactly
curl -s http://localhost:3000/api/spots/spot-1 | jq 'keys'

# Check the wrapper and the error envelope
curl -s 'http://localhost:3000/api/spots?page=2' | jq '{page, hasNextPage}'
curl -s http://localhost:3000/api/spots/nope | jq .
# → 404 { "error": { "code": "not_found", "message": … } }
```

In code: validate the contract with a test

```
@Test
fun `spot response matches schema`() = runTest {
    val response = apiClient.listSpots(page = 1, query = "", category = null)
    assertTrue(response.spots.isNotEmpty())
    assertTrue(response.hasNextPage)
    val first = response.spots.first()
    assertEquals("spot-1", first.id)
    assertTrue(first.photoUrl.isNotEmpty())   // the "" default would hide a renamed field
}
```

Interview Corner 🎤

Three questions mobile interviewers actually ask about networking

Interview Corner 🎤

Q1. What is the difference between an `IOException` and an HTTP error (`HttpException` in Retrofit, `ResponseException` in Ktor)? When would you retry each?

> IOException is a network-level failure — no connectivity, DNS failure, socket reset. These are retryable. HttpException / ResponseException is an HTTP error response from the server. 5xx may be retryable; 4xx should not be.

Q2. Your API (Application Programming Interface) returns snake_case field names but your Kotlin data class uses camelCase. Name two ways to handle this and the trade-offs.

> Option 1: `@SerialName("photo_url")` on each property — explicit but verbose. Option 2: configure `Json { namingStrategy = JsonNamingStrategy.SnakeCase }` globally — automatic but hides the mapping. Moshi supports `@Json(name=)` the same way. Pick `@SerialName` when only a few keys differ; the strategy when every key follows one convention.

Q3. The UniEats server returns `{ "spots": [...], "page": 1, "hasNextPage": true }` rather than a bare JSON (JavaScript Object Notation) array. Why do real APIs wrap list responses in an object? What breaks in your client if you ignore the wrapper and decode directly to `List<Spot>`?

> Wrapping lets the server add metadata (page number, total count, `hasNextPage`) without a breaking change to the response shape. If you decode directly to `List<Spot>`, the serializer tries to read an object `{}` as a list `[]` — Ktor throws a `JsonConvertException`, Codable throws a `DecodingError`, and the app shows an error or blank screen even though the server returned 200 OK. The fix is a `SpotsPage` wrapper data class whose field names match the JSON exactly.

🖥️ Demo — UniEats REST Fetch

> Start the server and the Android client, watch real HTTP requests hit `/api/spots`, and render the spot list with photos. Then rename `photoUrl` to `photoURL` in `Models.kt` and watch the photos vanish silently — fix it with `@SerialName("photoUrl")`. Finally stop the server: error state + **Retry**.

Repo: `github.com/dancojocar/ma` **Code:** `unieats/android` @ tag `l05-rest` · key files `data/remote/ApiClient.kt`, `di/AppModule.kt`, `data/model/Models.kt` **Server:** `cd unieats/server && npm ci && npm start` (listens on `http://localhost:3000`) **Run (repo root):** `./unieats/android/gradlew -p unieats/android :app:installDebug` **What this lecture adds:** `git diff l04-nav l05-rest -- unieats/android/`

🖥️ Demo — Pagination Live

> Scroll past spot 20 and watch page 2 load automatically: 5 more spots, then `hasNextPage` turns `false` and the app stops asking. Show it in the Android Studio debugger and the Network Inspector.

Repo: `github.com/dancojocar/ma` **Code:** `unieats/android` @ tag `l05-rest` · key file `unieats/android/app/src/main/java/com/unieats/app/ui/spotlist/SpotListViewModel.kt` **What to observe:** `loadNextPage()` moves `currentPage` from 1 to 2; the server answers `hasNextPage: false`; scrolling further sends no request

Comparing the Four Clients — Quick Reference

Concern	Android	iOS	Flutter	React Native
HTTP client	Ktor (OkHttp engine); Retrofit in most codebases	URLSession	Dio	fetch
Serialization	kotlinx.serialization	Codable	hand-written `fromJson`	Zod + TS
Async model	coroutine / suspend	async/await	async/await	Promise / async
State management	StateFlow + ViewModel	@Observable	Riverpod `AsyncNotifier`	TanStack Query
Pagination	manual `currentPage` (Paging 3 in larger apps)	manual `currentPage`	manual, in `SpotPagesNotifier`	useInfiniteQuery
Retry logic	Ktor `HttpRequestRetry`	Retry button	Dio `RetryInterceptor`	TanStack Query retry

Summary — What We Covered

- **HTTP (HyperText Transfer Protocol) on mobile**: request-response, status codes, mobile-specific constraints
- **JSON (JavaScript Object Notation) serialization**: kotlinx.serialization / Codable / hand-written `fromJson` / Zod
- **HTTP clients**: Ktor (UniEats) and Retrofit (the industry default) / URLSession / Dio / fetch + TanStack Query
- **Error handling**: typed errors, catch categories, client-level retry (Ktor plugin, Dio interceptor, TanStack Query) — all four stacks
- **Pagination**: server slice + `hasNextPage`; trigger / guard / append / stop on every stack — manual page counter (Android, iOS, Flutter `AsyncNotifier`), useInfiniteQuery (React Native)
- **LLM pitfalls**: 7 hallucination patterns, how to verify with curl
- **Architecture**: repository pattern, separating data and UI layers

Key takeaway:

> Always run `curl` against the real API (Application Programming Interface) before generating any client code with an AI tool.

If You Only Remember 3 Things

1. **Run `curl` first.** Before writing any client code or data class, look at the real JSON (JavaScript Object Notation) response. Field names, wrapper objects, and endpoint paths are the top source of silent bugs — and AI gets all three wrong.

2. **Separate your error types.** `IOException` (no network) and an HTTP error (`ResponseException` in Ktor — the server said no) need different handling. Putting them in one catch block hides information you need for retries and user messages. This rule applies on all four platforms.

3. **The platform clients all share the same shape.** Ktor or Retrofit / URLSession / Dio / fetch+TanStack Query are dialects of the same pattern: a typed async call, a serialization step, and a state container. Learn one column deeply — the others are translations.

Next Lecture

L06 — Concurrency & Reactive Programming

- Why the main thread is sacred (the 16ms budget)
- Kotlin Coroutines + Flow — cold vs hot
- Swift structured concurrency — actors, AsyncSequence
- Dart event loop, Futures, Streams, Isolates
- The JavaScript event loop — why RN has no threads (and what JSI changes)
- How viewModelScope, .task, and staleTime relate to lifecycle

Lab task: add pull-to-refresh and a loading skeleton to your UniEats spot list, using proper coroutine/async error handling.

Questions?