Lecture 6: Concurrency & Reactive Programming

Keep the main thread alive — or the user notices

UniEats · Mobile Applications 2026

Syntax You'll Meet Today

Construct	Stack	One-line anchor
`suspend fun`	Kotlin	A Java method that can **pause without blocking the thread** — the JVM (Java Virtual Machine) thread is free while it waits
`viewModelScope.launch { }`	Kotlin	Like Java's `ExecutorService.submit()`, but auto-cancelled when the ViewModel is destroyed
`Flow<T>`	Kotlin	Like RxJava's `Observable<T>` — a lazy sequence of values over time
`async fun` / `await`	Swift & Dart	Same idea as JS `async` / `await` you already know — marks a function that may pause
`actor`	Swift	A class where the compiler guarantees **only one caller runs inside at a time** — no manual locks needed
`Stream<T>`	Dart	Like Kotlin `Flow<T>` / JS `AsyncGenerator` — emits values over time, one subscriber by default

> Swift is **read-only** this semester — you will read and compare it, never write it from scratch.

Hook: The Jank Incident of 2019

> "We shipped. Users rated us 1 star. The app froze for 3 seconds on every restaurant tap."

- The network call was on the **main thread**
- `StrictMode` was disabled for release — no warning in logs
- Fix: move to coroutines → 3 hours of work, 2 stars recovered overnight

Lesson: The main thread is a shared resource. Block it and users feel it right away.

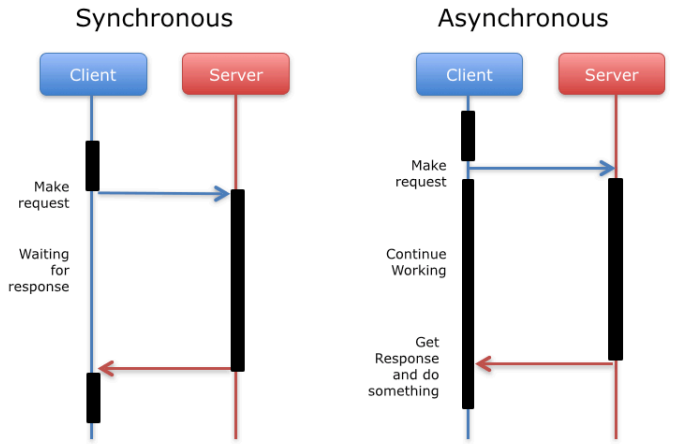
Part 1 — The Sacred Main Thread

Why 16 ms?

- Displays refresh at **60 Hz** → one frame every **16.67 ms**
- ProMotion / high-refresh screens: 120 Hz → **8.33 ms** per frame
- The OS (Operating System) composites your UI (User Interface) on the **main thread**
- If your code holds the thread longer than one frame → **jank** (visible stutter)

```
Frame N    Frame N+1
|——16ms——|——16ms——|——16ms——|
  ✓ draw    ✗ blocked   ✗ missed   ← user sees stutter
```

Rule: Do zero slow work on the main thread. Zero.



Synchronous: the client waits and does nothing.
Asynchronous: the client keeps working while the server responds.

What Counts as "Slow"?

Operation	Typical cost	Safe on main thread?
UI (User Interface) state write	< 1 µs	✓
Simple arithmetic	< 10 µs	✓
SharedPreferences read	1–5 ms	⚠ risky
JSON parse (10 KB)	2–8 ms	⚠ risky
Network request	50 ms–5 s	✕ never
SQLite query (large)	10 ms–500 ms	✕ never
Image decode (4 MP)	20–100 ms	✕ never

Rule of thumb: if it touches I/O (Input/Output) — disk, network, database — it goes off the main thread.

Android: `StrictMode.setThreadPolicy` catches violations in debug builds.
iOS: `@MainActor` enforcer + thread sanitizer in Xcode.

Part 2 — Kotlin Coroutines & Flow (Android)

Coroutines: Sequential Code, No Blocking

From callback hell to readable, sequential-looking async code:

```
// ViewModel
fun loadSpot(id: String) {
    // launch ≈ submit() on a Java ExecutorService, but lifecycle-aware
    viewModelScope.launch {                      // auto-cancelled when ViewModel clears
        // suspend: thread is freed while waiting (≠ Thread.sleep which blocks it)
        val spot = repository.fetch(id)    // suspends, not blocks
        _uiState.value = UiState.Success(spot)  // resumes on Main thread
    }
}
```

- `launch` starts a coroutine in `viewModelScope` — cancelled when ViewModel clears
- `suspend fun fetch()` suspends the coroutine without blocking the thread
- Returns to `Dispatchers.Main` after the suspend — safe to write to `StateFlow`
- **Structured concurrency:** every coroutine has a parent scope that controls its lifetime

Dispatchers: Where Your Code Runs

```
viewModelScope.launch {
    // Dispatchers.Main by default — like runOnUiThread
    // withContext ≈ "switch pool for this block,
    // then come back"
    val list = withContext(Dispatchers.IO) {
        db.spotDao().getAll()   // disk — IO pool
    }
    val processed = withContext(Dispatchers.Default) {
        list.sortedBy { it.rating }   // CPU — Default
    }
    _uiState.value = UiState.Success(processed)
    // back on Main — safe to update the UI
}
```

Dispatcher	Thread pool	Use for
`Main`	UI (User Interface) thread	UI writes
`IO`	64 threads	Network, disk
`Default`	CPU (Central Processing Unit) count	Heavy compute
`Unconfined`	Caller’s	Testing only

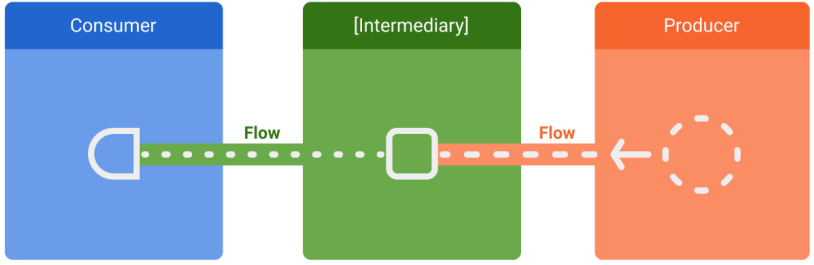
Kotlin Flow: Reactive Streams on Coroutines

Cold flow — produced on demand

```
fun spotFeed(): Flow<List<Spot>> = flow {
    while (true) {
        emit(api.listSpots(1, "", null).spots)
        delay(30_000)
    }
}
```

Hot flow — shared state

```
class SpotListViewModel(…) : ViewModel() {
    private val _uiState =
        MutableStateFlow(SpotListUiState())
    val uiState: StateFlow<SpotListUiState> =
        _uiState.asStateFlow()
}
```



- **Cold:** nothing happens until `collect` is called; each collector gets its own run
- **Hot:** always active; late collectors get the current/latest value

StateFlow vs SharedFlow

	`StateFlow`	`SharedFlow`
Initial value	Required	None
Replay	1 (current value)	Configurable (0…N)
Use case	UI (User Interface) state	Events (navigation, toasts)
Late collector sees	Current value	Only future emissions

```
// One-shot events (navigation) — use SharedFlow, not StateFlow
private val _events = MutableSharedFlow<NavEvent>()
val events = _events.asSharedFlow()

fun navigateToDetail(id: String) {
    viewModelScope.launch { _events.emit(NavEvent.Detail(id)) }
}
```

- A navigation event in a `StateFlow` **re-fires**: every new collector gets the old event again

Lifecycle-Safe Collection (Android)

```
// In Activity / Fragment — the CORRECT pattern (lifecycle-runtime-ktx 2.4+)
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.state.collect { state ->
            render(state)
        }
    }
}
```

- `repeatOnLifecycle(STARTED)` **cancels** the inner block when lifecycle goes below STARTED
- **Re-launches** when lifecycle returns to STARTED
- Stops collecting while the app is in the background — saves battery and avoids crashes
- In Compose: `collectAsStateWithLifecycle()` does the same for you

Part 3 — Swift Structured Concurrency (iOS)

async/await + Actors: Swift's Answer

```swift
// ViewModel (iOS 17+, @Observable)
@Observable
final class SpotViewModel {
    var spots: [Spot] = []
    var error: Error?

    func load() async {
        do {
            spots = try await api.listSpots()
        } catch {
            self.error = error
        }
    }
}

// View
struct SpotsView: View {
    @State private var vm = SpotViewModel()
    var body: some View {
        List(vm.spots) { SpotRow(spot: $0) }
            .task { await vm.load() }
            // cancelled when the view disappears
    }
}
```

```swift
// actor — compiler guarantees one caller at a time
actor SpotCache {
    private var cache: [String: Spot] = [:]
    func get(_ id: String) -> Spot? { cache[id] }
    func set(_ id: String, _ s: Spot) { cache[id] = s }
}
let cached = await cache.get("123")

// @MainActor pins a class to the main thread
@MainActor
final class FeedViewModel: ObservableObject {
    @Published var items: [Spot] = []
    func refresh() async {
        let data = await api.listSpots()
        items = data  // back on MainActor automatically
    }
}
```

`@Observable` (iOS 17) replaces `ObservableObject` + `@Published` · `@MainActor` ≈ Kotlin's `Dispatchers.Main`

Parallel Work: All Four Stacks

Kotlin — `async` + `awaitAll`

```kotlin
// coroutineScope keeps both tasks structured —
// cancelling the caller cancels both children
suspend fun loadPair(): Pair<Menu, Reviews> =
    coroutineScope {
        val menu    = async { api.fetchMenu(spotId) }
        val reviews = async { api.fetchReviews(spotId) }
        Pair(menu.await(), reviews.await())
    }
```

React Native — `Promise.all`

```typescript
async function loadPair(
  spotId: string,
): Promise<[Menu, Reviews]> {
  const [menu, reviews] = await Promise.all([
    fetchMenu(spotId), fetchReviews(spotId),
  ])
  return [menu, reviews]
}
```

Swift — `withTaskGroup`

```swift
func loadAll(ids: [String]) async -> [Spot] {
    await withTaskGroup(of: Spot?.self) { group in
        for id in ids {
            group.addTask { try? await api.fetch(id) }
        }
        return await group.compactMap { $0 }
            .reduce(into: []) { $0.append($1) }
    }
}
```

Dart — `Future.wait`

```dart
final results = await Future.wait([
  fetchMenu(spotId), fetchReviews(spotId)
]);
```

Key difference: Kotlin `coroutineScope` and Swift `withTaskGroup` propagate cancellation to children automatically. `Promise.all` and `Future.wait` have no cancellation — rejected results are discarded but other calls keep running.

Cancellation Is Cooperative — You Have to Check

Structured concurrency *requests* cancellation; a coroutine only stops at a suspension point — or when it **checks**. A tight CPU loop with no `suspend` call ignores cancellation:

```
// ❌ Runs to the end even after the screen (and its scope) is gone
suspend fun rank(items: List<Spot>): List<Spot> =
    items.map { heavyScore(it) }   // no suspension point → never notices cancellation

// ✅ Cooperates — bails out fast when the scope is cancelled
suspend fun rank(items: List<Spot>): List<Spot> = coroutineScope {
    items.map { ensureActive(); heavyScore(it) }  // throws CancellationException
}
```

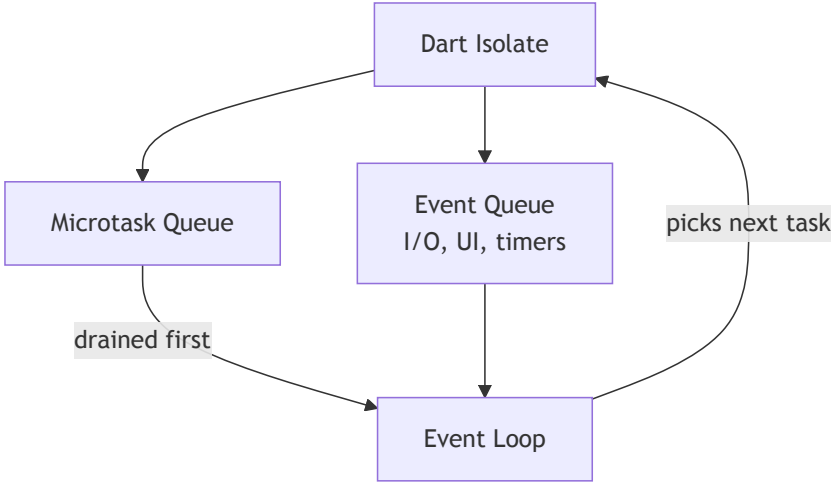
And never swallow the cancellation signal:

```
try { repo.refresh() }
catch (e: CancellationException) { throw e }  // rethrow — HOW cancellation propagates
catch (e: Exception) { showError(e) }
```

- `ensureActive()` / `isActive` let long CPU loops notice the cancellation flag.
- A blanket `catch (e: Exception)` that eats `CancellationException` keeps cancelled work alive — a classic leak. `CancellationException` **is sacred: always rethrow it.**

Part 4 — Dart: Event Loop, Futures, Streams, Isolates (Flutter)

Dart’s Single-Threaded Event Loop



- Everything in one isolate runs on **one thread**
- Async I/O (Input/Output) is non-blocking: the event loop picks up completions
- **Microtask queue** (Futures) drains before the next event
- No shared memory between isolates — message passing only
- Heavy CPU (Central Processing Unit) work **does** block the loop → move it to an Isolate

Futures, Streams & Isolates in Dart

Future — a value that arrives later

```
final spotDetailProvider =
    FutureProvider.autoDispose.family<Spot, String>(
  (ref, id) =>
      ref.watch(apiClientProvider).fetchSpot(id),
);

ref.watch(spotDetailProvider(id)).when(
  data: (spot) => Text(spot.name),
  loading: () => const CircularProgressIndicator(),
  error: (e, _) => Text('$e'),
)
```

Isolate — true parallelism for CPU work

```
Future<SpotsPage> parseInBackground(String json) =>
    Isolate.run(() => SpotsPage.fromJson(
        jsonDecode(json) as Map<String, dynamic>));
```

Stream — reactive sequence of values

```
// Cold stream — async* + yield
Stream<List<Spot>> spotsStream() async* {
  while (true) {
    yield (await api.fetchSpots()).spots;
    await Future.delayed(const Duration(seconds: 30));
  }
}

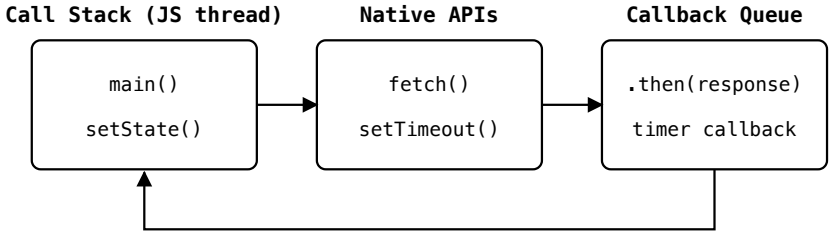
// Hot broadcast stream
final _ctrl = StreamController<AppEvent>.broadcast();
Stream<AppEvent> get events => _ctrl.stream;

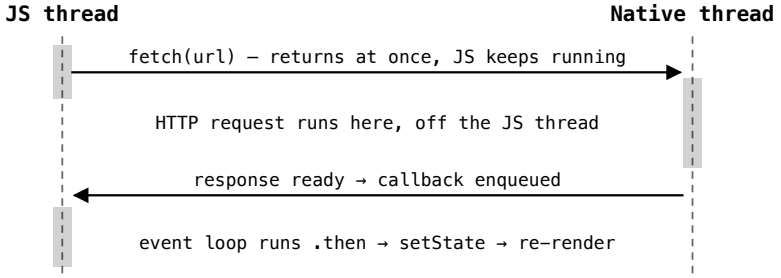
// Always cancel subscriptions in dispose()
late StreamSubscription<List<Spot>> _sub;
@override
void initState() {
  super.initState();
  _sub = spotsStream()
      .listen((list) => setState(() => _spots = list));
}
@override
void dispose() { _sub.cancel(); super.dispose(); }
```

> `Isolate.run` is for CPU work only (e.g. a big JSON parse) — network I/O is already async.

Part 5 — JavaScript Event Loop & React Native

The JavaScript Event Loop & React Native





TanStack Query — the data-fetching standard · `src/hooks/useSpots.ts` Caches, tracks loading/error, retries · `signal` aborts requests nobody needs

```
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),
  });
}
```

New Architecture (JSI + Worklets):

- **JSI (JavaScript Interface)** lets JS call native code synchronously via C++
- **Worklets** (Reanimated 4) run animation logic on the UI thread at 120 Hz
- **TurboModules** lazy-load native modules

Part 6 — Cold vs Hot Streams Across All Four Stacks

Cold vs Hot: The Mental Model

```
COLD: like a Netflix show — starts from episode 1 for every viewer
HOT:  like live TV — you join mid-stream; you miss what already aired
```

Concept	Kotlin	Swift	Dart	React Native
Cold	`Flow`	`AsyncStream`	single-sub `Stream`	Observable (RxJS)
Hot (state)	`StateFlow`	`@Observable` / actor	`StreamController.broadcast`	Zustand store
Hot (events)	`SharedFlow`	`AsyncStream` + `AsyncChannel`	broadcast `Stream`	EventEmitter
Operator chain	`map`, `filter`, `flatMapLatest`	`map`, `filter`, `compactMap`	`map`, `where`, `asyncExpand`	`pipe`, `map`, `switchMap`

Converting Cold to Hot — Android: `stateIn`

`ui/spotlist/SpotListViewModel.kt` @ `l06-async`

```
private val liveConnection: Flow<Boolean> =
    liveUpdates.events()
        .onEach(::applyLiveEvent)
        .map { it != LiveEvent.Disconnected }
        .onStart { emit(false) }
        .distinctUntilChanged()

val uiState: StateFlow<SpotListUiState> =
    combine(_uiState, liveConnection) { state, live ->
        state.copy(isLive = live)
    }.stateIn(
        viewModelScope,
        SharingStarted.WhileSubscribed(5_000),
        _uiState.value,
    )
```

What to notice

- `liveUpdates.events()` is a **cold** flow: nothing connects until someone collects it
- The socket is upstream of `uiState`, so it is open only while the screen collects `uiState`
- Without `stateIn`, every collector would open its own socket. With it, all collectors share one
- `WhileSubscribed(5_000)` keeps the socket through a rotation (< 1 s) and closes it 5 s after the user leaves
- `onEach(::applyLiveEvent)` writes each event into `_uiState`; the flow itself only reports *connected or not*
- Emulator host: `ws://10.0.2.2:3000/live`

Converting Cold to Hot — iOS: `.task` + `for await`

`Views/SpotListView.swift` @ `l06-async`

```
.task(id: scenePhase) {
    guard scenePhase == .active else { return }
    for await update in live.updates() {
        viewModel.apply(update)
    }
    viewModel.apply(.disconnected)
}
```

What to notice

- `live.updates()` is an `AsyncStream`: **cold**, it connects when the `for await` loop starts
- `.task` ties the loop to the view: when the view disappears the task is cancelled and the socket closes
- `id: scenePhase` restarts the task on every phase change; the `guard` returns at once in the background, so backgrounding also closes the socket
- Returning from the loop runs `apply(.disconnected)`, so the badge never shows a stale "live"
- No `stateIn` here: SwiftUI's view lifetime *is* the subscription

Converting Cold to Hot — Flutter: `autoDispose`

`lib/providers/providers.dart` @ `l06-async`

```
final liveEventsProvider =
    StreamProvider.autoDispose<LiveEvent>((ref) {
  final foreground = ref.watch(
    appLifecycleProvider.select(
      (s) => s == AppLifecycleState.resumed ||
          s == AppLifecycleState.inactive,
    ),
  );
  if (!foreground) return const Stream.empty();
  return liveSpotEvents(liveUrl);
});
```

What to notice

- `liveSpotEvents` is a **cold** `Stream`: it connects when the provider is first listened to
- `autoDispose` ≈ `WhileSubscribed`: the provider lives while a widget watches it and is torn down when the last one stops
- `ref.watch(appLifecycleProvider.select(...))` rebuilds the provider on foreground changes only, not on every lifecycle tick
- In the background it returns `Stream.empty()`, so the socket is closed and reopened on resume
- `AsyncValue` from `ref.watch(liveEventsProvider)` gives loading / data / error for free

Converting Cold to Hot — React Native: Query cache

`src/hooks/useLiveSpots.ts` @ `l06-async`

```ts
export function useLiveSpots(): LiveStatus {
  const queryClient = useQueryClient();
  const appActive = useAppActive();

  useFocusEffect(useCallback(() => {
    if (!appActive) return;
    const socket = new WebSocket(LIVE_URL);
    socket.onmessage = (m) =>
      applyLiveEvent(queryClient,
        LiveEventSchema.parse(JSON.parse(m.data)));
    return () => socket.close();
  }, [appActive, queryClient]));
}

function applyLiveEvent(qc: QueryClient, e: LiveEvent) {
  qc.invalidateQueries({ queryKey: ["spots"] });
  if (e.type === "spot.deleted")
    qc.invalidateQueries({ queryKey: ["spot", e.id] });
  else qc.setQueryData(["spot", e.spot.id], e.spot);
}
```

What to notice

- There is no stream type: the **TanStack Query cache** is the hot store every screen already reads
- `useFocusEffect` + `useAppActive` open the socket only while the list is focused and the app is active; the cleanup closes it
- Each event invalidates the `["spots"]` lists (they refetch) and writes the changed spot straight into its detail query
- Deleted spot: invalidate the detail instead of writing to it
- The real hook also reconnects with exponential back-off and resyncs after a reconnect (trimmed here)

A Cold, Reconnecting Stream — Android: `channelFlow`

Requirement: open `ws://<host>:3000/live` only while someone listens, reconnect with back-off, close on cancel

`data/remote/LiveUpdates.kt` @ `l06-async`

```
fun events(): Flow<LiveEvent> = channelFlow {
    var attempt = 0
    while (true) {
        try {
            http.webSocket(urlString = liveUrl, …) {
                attempt = 0
                send(LiveEvent.Connected)
                for (frame in incoming) {
                    if (frame is Frame.Text)
                        handleLiveMessage(frame.readText())
                }
            }
        } catch (e: CancellationException) {
            throw e
        } catch (e: Exception) {
            Log.w(TAG, "live connection failed: …")
        }
        send(LiveEvent.Disconnected)
        delay(minOf(1_000L shl attempt.coerceAtMost(5),
            MAX_BACKOFF_MS))
        attempt++
    }
}
```

What to notice

- **Cold:** the `channelFlow` body runs only when someone collects; no collector, no socket
- `channelFlow` lets the Ktor callback `send` into the flow from inside `webSocket { }`
- **Cancel closes the socket:** cancelling the collector cancels the block, and Ktor closes the connection on the way out
- **Rethrow** `CancellationException` **first.** Otherwise leaving the screen looks like a failure and the loop reconnects — the rule from the cancellation slide
- **Back-off:** `1_000L shl attempt` gives 1 s, 2 s, 4 s … capped by `MAX_BACKOFF_MS` at 30 s; `attempt = 0` resets it on a good connection
- `Connected` / `Disconnected` events drive the `● Live` badge

A Cold, Reconnecting Stream — Flutter: `StreamController`

Requirement: open `ws://<host>:3000/live` only while someone listens, reconnect with back-off, close on cancel

`lib/data/network/live_updates.dart` @ `l06-async`

```
Stream<LiveEvent> liveSpotEvents(Uri url) {
  late final StreamController<LiveEvent> controller;
  WebSocket? socket;
  var cancelled = false;

  Future<void> connectLoop() async {
    var backoff = const Duration(seconds: 1);
    while (!cancelled) {
      try {
        socket = await WebSocket.connect(…);
        controller.add(const LiveConnected());
        await for (final message in socket!) { … }
      } on Exception { … }
      if (cancelled) break;
      controller.add(LiveDisconnected(backoff));
      await Future<void>.delayed(backoff);
      backoff = …; // doubles, capped at 30 s
    }
    await socket?.close();
  }

  controller = StreamController<LiveEvent>(
    onListen: connectLoop,
    onCancel: () {
      cancelled = true;
      return socket?.close();
    },
  );
  return controller.stream;
}
```

What to notice

- **Cold:** `onListen` starts `connectLoop` ; the function only builds the controller, nothing connects until a listener arrives
- **Cancel closes the socket:** `onCancel` sets the `cancelled` flag and closes the socket, which ends the `await for` and the loop
- No `CancellationException` in Dart: cancellation is the **manual flag** checked at the top of the loop and after each failure
- **Back-off:** 1 s, doubling, capped at 30 s — the same curve as Android, written by hand
- `LiveConnected` / `LiveDisconnected` events drive the `● Live` badge

A Cold, Reconnecting Stream — iOS: `AsyncStream` + `Task`

Requirement: open `ws://<host>:3000/live` only while someone listens, reconnect with back-off, close on cancel

`Network/LiveUpdateService.swift` @ `l06-async`

```
func updates() -> AsyncStream<LiveUpdate> {
    AsyncStream { continuation in
        let worker = Task {
            var attempt = 0
            while !Task.isCancelled {
                let socket = session.webSocketTask(with: url)
                socket.resume()
                await withTaskCancellationHandler {
                    do {
                        try await Self.ping(socket)
                        attempt = 0
                        continuation.yield(.connected)
                        while true {
                            let message = try await socket.receive()
                            if let update = Self.decode(message) {
                                continuation.yield(update)
                            }
                        }
                    } catch {
                        continuation.yield(.disconnected)
                    }
                } onCancel: {
                    socket.cancel(with: .goingAway, reason: nil)
                }
                let delay = min(30, 1 << min(attempt, 5))
                attempt += 1
                try? await Task.sleep(for: .seconds(delay))
            }
            continuation.finish()
        }
        continuation.onTermination = { _ in worker.cancel() }
    }
}
```

What to notice

- **Cold:** the `AsyncStream` closure runs when the first `for await` starts; the `worker` Task is created then, not when `updates()` is called
- `continuation.yield` is the Swift twin of Kotlin's `send` and Dart's `controller.add`
- **Cancel closes the socket:** `.task` cancels the consumer → `onTermination` cancels the worker → `withTaskCancellationHandler`'s `onCancel` closes the socket while `receive()` is still waiting
- `while !Task.isCancelled` is the cooperative check — the same idea as `ensureActive()` on Android
- **Back-off:** `1 << min(attempt, 5)` gives 1 s, 2 s, 4 s … capped at 30 s; `attempt = 0` after a successful ping
- `.connected` / `.disconnected` drive the `● Live` badge

A Cold, Reconnecting Stream — React Native: `WebSocket` + `useFocusEffect`

Requirement: open `ws://<host>:3000/live` only while someone listens, reconnect with back-off, close on cancel

`src/hooks/useLiveSpots.ts` @ `l06-async`

```
useFocusEffect(useCallback(() => {
  if (!appActive) return;
  let socket: WebSocket | null = null;
  let retryTimer: ReturnType<typeof setTimeout> | undefined;
  let attempt = 0;
  let disposed = false;

  const connect = () => {
    setStatus("connecting");
    socket = new WebSocket(LIVE_URL);
    socket.onopen = () => {
      attempt = 0;
      setStatus("live");
      // events sent while disconnected are lost: resync once
      if (connectedBefore.current)
        queryClient.invalidateQueries({ queryKey: ["spots"] });
      connectedBefore.current = true;
    };
    socket.onmessage = (m) => { … applyLiveEvent(…) … };
    socket.onclose = () => {
      if (disposed) return;
      setStatus("connecting");
      const delay = Math.min(MAX_BACKOFF_MS, 1000 * 2 ** attempt);
      attempt += 1;
      retryTimer = setTimeout(connect, delay);
    };
  };

  connect();
  return () => {
    disposed = true;
    clearTimeout(retryTimer);
    socket?.close();
  };
}, [appActive, queryClient]));
```

What to notice

- **No stream type at all:** the "subscription" is the effect's lifetime — it runs when the screen gains focus and the app is active
- **Cancel closes the socket:** the returned cleanup sets `disposed`, clears the pending retry and closes the socket; `onclose` sees `disposed` and does *not* reconnect
- `disposed` plays the role of Dart's `cancelled` flag — JS has no cancellation signal for a callback either
- **Back-off:** `1000 * 2 ** attempt` gives 1 s, 2 s, 4 s … capped by `MAX_BACKOFF_MS` at 30 s; `onopen` resets `attempt`
- **Resync on reconnect:** events sent while offline are lost, so `onopen` invalidates `["spots"]` once — the other stacks rely on the next list refresh
- `status` drives the `● Live` badge
- **No database yet at l06:** reactive DB streams (Room Flows, Drift watch queries, expo-sqlite, SwiftData) arrive in L07

Spot the AI Bug 🐞 — Android

Spot the AI Bug 🐞 — Can You Find It?

A student asked an LLM (Large Language Model) to "collect the spot stream safely in a Fragment":

```
class SpotFragment : Fragment() {
    private val viewModel: SpotViewModel by viewModels()

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        lifecycleScope.launch {   // <-- is this correct?
            viewModel.spots.collect { list ->
                adapter.submitList(list)
            }
        }
    }
}
```

Question: What is wrong with this code? Discuss for 60 seconds.

Spot the AI Bug 🐞 — Revealed

```
// WRONG — collects even in background
lifecycleScope.launch {
    viewModel.spots.collect { list ->
        adapter.submitList(list)
    }
}

// CORRECT — cancels collection when Fragment stops
viewLifecycleOwner.lifecycleScope.launch {
    viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.spots.collect { list ->
            adapter.submitList(list)
        }
    }
}
```

Two bugs actually:

1. Missing `repeatOnLifecycle` — collects while in background
2. Using `this.lifecycleScope` instead of `viewLifecycleOwner.lifecycleScope` in a Fragment — the Fragment lifecycle outlives the View

Spot the AI Bug 🐞 — Swift Edition

Spot the AI Bug 🐞 — Swift: Can You Find It?

```swift
@MainActor
final class SpotViewModel: ObservableObject {
    @Published var spots: [Spot] = []

    func refresh() {
        Task {
            let data = await api.listSpots()  // network
            spots = data  // is this safe?
        }
    }
}
```

Question: Is there a thread-safety issue? Why or why not?

Spot the AI Bug 🐞 — Swift: Revealed

```swift
// This is actually CORRECT — Task inherits @MainActor context
@MainActor
final class SpotViewModel: ObservableObject {
    @Published var spots: [Spot] = []

    func refresh() {
        Task {  // inherits @MainActor → body runs on main thread
            let data = await api.listSpots()
            spots = data  // safe: back on MainActor after await
        }
    }
}

// WRONG version — explicit Task.detached breaks the MainActor link
func refresh() {
    Task.detached {  // NOT on MainActor!
        let data = await self.api.listSpots()
        await MainActor.run { self.spots = data }  // must fix explicitly
    }
}
```

> Swift 6 strict concurrency mode flags unsafe `@MainActor` writes at compile time.

Part 7 — Lifecycle-Safe Patterns Across Stacks

Lifecycle Safety: All Four Stacks

Android

```
// repeatOnLifecycle — the gold standard
viewLifecycleOwner.lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        vm.flow.collect { render(it) }
    }
}
// OR in Compose:
val state by vm.flow
    .collectAsStateWithLifecycle()
```

iOS

```
// .task modifier cancels on view disappear
.task {
    for await state in vm.stateStream {
        // cancelled automatically when view leaves
    }
}
```

Flutter

```
// Riverpod handles lifecycle —
// no manual subscribe needed
ref.watch(spotPagesProvider).when(...)

// Manual StreamSubscription: always cancel in dispose()
@override
void dispose() {
  _subscription?.cancel();
  super.dispose();
}
```

React Native

```
useEffect(() => {
  const sub = observable.subscribe(setState)
  return () => sub.unsubscribe()  // cleanup function
}, [])
```

Test Async Code — `runTest` + Turbine

Now that you have coroutines and Flow, you can test the L03 ViewModel — with **no real clock and no real network**. **Exercise:** at l06 the real constructor is `SpotListViewModel(spotRepository: SpotRepository, liveUpdates: LiveUpdates)` — two concrete classes. Extract an interface for each (`fetchSpots(page, query, category): Result<SpotsPage>` and `events(): Flow<LiveEvent>`), let the ViewModel take the interfaces, and this test compiles (`canteen` = any `Spot` , e.g. `spot-1`):

```
class FakeSpotSource(private val spots: List<Spot>) : SpotSource {
    override suspend fun fetchSpots(page: Int, query: String, category: Category?) =
        Result.success(SpotsPage(spots, page = 1, hasNextPage = false))
}
class FakeLiveEvents : LiveEventSource {
    override fun events(): Flow<LiveEvent> = flowOf(LiveEvent.Connected)
}

@Test
fun `first page lands in uiState`() = runTest {
    val vm = SpotListViewModel(FakeSpotSource(listOf(canteen)), FakeLiveEvents())
    vm.uiState.test {
        assertEquals(0, awaitItem().spots.size)
        assertEquals(1, awaitItem().spots.size)
        cancelAndIgnoreRemainingEvents()
    }
}
```

- `runTest { }` runs coroutines on a **virtual clock** — no real `delay` , so tests are fast and deterministic.
- A hand-written **fake** (a plain class implementing the interface) beats a mock: no framework, easy to read.
- **Turbine** (`uiState.test { awaitItem() }`) waits for each emission — asserting synchronously on async state is the #1 testing bug (we name it again in L14).
- Real ViewModels also set `Dispatchers.setMain(testDispatcher)` so `viewModelScope` uses the virtual clock — full setup in L14.

Test Async Code — Reading the Test

- 1st `awaitItem()` → the `stateIn` initial value: an empty list
- 2nd `awaitItem()` → the state after page 1 arrives: one spot
- `cancelAndIgnoreRemainingEvents()` drops the later `isLive` emission — Turbine **fails** a test that leaves events unconsumed
- `SpotSource` and `LiveEventSource` are names for this exercise — at `l06-async` the repo has no unit tests yet
- UniEats does this refactor for real in L12: `EatsRepository`, `FakeEatsRepository`, `SpotListViewModelTest` @ tag `l12-kmp`

🖥️ Demo — UniEats Live Updates (Streams + WebSocket)

> Open the Android list, change a spot from a terminal, and watch the row update with no refresh — the ● `Live` badge shows the WebSocket is open. Then background the app for more than 5 s to show `WhileSubscribed(5_000)` closing the socket.

Repo: `github.com/dancojocar/ma` **Code:** `unieats/android` @ tag `l06-async` · key files `data/remote/LiveUpdates.kt`, `ui/spotlist/SpotListViewModel.kt` **Server:** `cd unieats/server && npm ci && npm start` → REST on `:3000/api`, WebSocket `ws://localhost:3000/live` (`ws://10.0.2.2:3000/live` from the emulator) **Run (repo root):** `./unieats/android/gradlew -p unieats/android :app:installDebug` **Change a spot (no token needed — mutations stay open until L08):** `curl -X PATCH localhost:3000/api/spots/spot-3 -H 'Content-Type: application/json' -d '{"name":"Pizza Stop (new oven)"}'` **What this lecture adds:** `git diff l05-rest l06-async -- unieats/android/`

Interview Corner 🎤

Interview Corner 🎤 — Questions You Will Be Asked

Q1 (Android). What is the difference between `StateFlow` and `SharedFlow`? When would you choose each?

Q2 (Android). Why is `lifecycleScope.launch { flow.collect { } }` potentially dangerous in a Fragment? What is the correct pattern?

Q3 (Swift). What is the difference between `Task` and `Task.detached`? How does actor isolation affect each?

Interview Corner 🎤 — Model Answers

StateFlow vs SharedFlow

- `StateFlow` : always has a value, replays 1 to late collectors → UI (User Interface) state
- `SharedFlow(replay=0)` : no replay → one-shot events (navigation, toasts)
- Both are hot; both are thread-safe

Fragment lifecycle scope

- `lifecycleScope` outlives the Fragment's View
- Must use `viewLifecycleOwner.lifecycleScope`
- AND wrap collect in `repeatOnLifecycle(STARTED)`

Task vs Task.detached (Swift)

- `Task` inherits actor context — `@MainActor` state write after `await` is safe
- `Task.detached` breaks actor link — must use `await MainActor.run { }` to write UI state
- Swift 6 strict concurrency mode catches violations at compile time

Appendix — Where We Came From: Callback Hell (2013–2017)

```
// Pre-RxJava (Reactive Extensions for Java) Android: the pyramid of doom
userManager.login("alice", "pw", object : Callback<User> {
    override fun onSuccess(user: User) {
        apiClient.fetchProfile(user.id, object : Callback<Profile> {
            override fun onSuccess(profile: Profile) {
                imageLoader.load(profile.avatarUrl, object : Callback<Bitmap> {
                    override fun onSuccess(bitmap: Bitmap) {
                        runOnUiThread { avatarView.setImageBitmap(bitmap) }
                    }
                    override fun onError(e: Exception) { showError(e) }
                })
            }
            override fun onError(e: Exception) { showError(e) }
        })
    }
    override fun onError(e: Exception) { showError(e) }
})
```

Problems: error handling repeated at every level, threading manual, lifecycle ignored, impossible to test

RxJava (Reactive Extensions for Java): Legacy Context + Migration

2018-era RxJava 2 (you will see this in legacy code)

```
// Observable<User>: upstream on IO, downstream on UI
val disposable = userManager.getUser()
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribeWith(object : DisposableObserver<User>() {
        override fun onNext(user: User) {
            textView.text = user.name
        }
        override fun onError(e: Throwable) {
            showError(e)
        }
        override fun onComplete() { }
    })
compositeDisposable.add(disposable)
// onDestroy: compositeDisposable.clear()
```

Modern equivalent (Kotlin Coroutines)

```
suspend fun getUser(id: String): User =
    withContext(Dispatchers.IO) { api.fetchSync(id) }

// Bridge when migrating RxJava → coroutines
suspend fun getUserBridge(id: String): User =
    rxSingle.await()   // kotlinx-coroutines-rx3
```

- `io.reactivex.rxjava3:rxjava` is still maintained
- Add `kotlinx-coroutines-rx3` for bridge APIs
- Migrate layer by layer: data → domain → UI
- `subscribeOn` ≈ Flow's `flowOn` · bridges: `Single.await()`, `Observable.asFlow()`
- Forgetting `compositeDisposable.clear()` leaks memory

Interview Corner 🎤 — Q4 (Cross-platform, Self-Study)

Q4 (Cross-platform). A Flutter app freezes when parsing a large JSON response. How do you fix it? What is the Dart equivalent of Kotlin's `Dispatchers.Default`?

Answer:

```
// Move heavy parse to an Isolate
final page = await Isolate.run(() =>
  SpotsPage.fromJson(
      jsonDecode(raw) as Map<String, dynamic>)
);
```

- Dart has no thread pool for CPU work within a single isolate — a separate `Isolate` is the only way to get true parallelism
- `Isolate.run()` handles spawn-compute-terminate automatically; use for any JSON parse over ~50 KB
- The Dart equivalent of `Dispatchers.Default` is spawning an `Isolate`; `Dispatchers.IO` maps to Dart's native async I/O which is already off the event loop

Summary — What We Covered Today

- **The 16 ms rule** — main thread is sacred; jank is visible and measurable
- **Kotlin coroutines + Flow** — suspend functions, structured scopes, cold vs hot, `stateIn`, `repeatOnLifecycle`
- **Swift structured concurrency** — `async` / `await`, `@MainActor`, actors, `AsyncSequence`, `.task`, task groups
- **Dart event loop** — `Future`s, `Stream`s, Stream operators, `Isolate.run()` for CPU (Central Processing Unit) work
- **React Native** — JS (JavaScript) single-thread model, JSI (JavaScript Interface) + worklets for the New Architecture
- **Cold vs hot** — universal concept, platform-specific vocabulary
- **Lifecycle-safe collection** — the number one source of subtle bugs on all four platforms

If You Remember 3 Things From Today

1. **Never block the main thread.** Every platform gives you one frame budget (~16 ms). Network, disk, and heavy CPU (Central Processing Unit) work go on a background thread / coroutine / isolate — always.

2. **Structured concurrency = automatic cleanup.** `viewModelScope`, Swift's `.task`, Riverpod providers, and `useEffect` cleanup all mean the same thing: when the screen is gone, the work stops. Missing this is the number one source of background battery drain and crashes.

3. **Cold = lazy (one subscriber, one execution); Hot = shared (many subscribers, one execution).** Use a cold stream for on-demand fetches; convert to hot (`stateIn` / `StateFlow` / broadcast) when multiple UI (User Interface) components need the same data.

Next Up: L07 — Local Persistence & Offline-First

- Room (Android) ↔ SwiftData (iOS) ↔ Drift (Flutter) ↔ expo-sqlite (React Native)
- The offline-first sync pattern: local DB (Database) as source of truth, server as eventual truth
- Conflict resolution strategies (last-write-wins on `updatedAt`, the server's 409)
- UniEats: spots stored locally, offline edits queued in an outbox and replayed on reconnect

Lab this week: Add `StateFlow`-backed search filtering to UniEats — a query `StateFlow` + `combine` with the spot list

Questions?