State & Architecture Foundations

One mental model. Four dialects.

Lecture 3 — Mobile Applications

Syntax You Will Meet Today

New syntax used in this deck — each one anchored to something you already know:

Construct	Stack	One-line anchor
`sealed interface` / `data object`	Kotlin	Like a Java `enum`, but each variant can carry different fields
`StateFlow<T>` / `Flow<T>`	Kotlin	Like Java's `Observable` or a Promise that keeps sending values
`suspend fun` / `viewModelScope.launch { }`	Kotlin	Marks work that can pause for I/O without freezing the UI; `launch` runs it. **Recognition only today — full model in L06.**
`LaunchedEffect(key) { }`	Kotlin	Runs a coroutine when the composable appears or its `key` changes — Compose's "run this side-effect" hook
`by` delegation (`val x by remember { … }`)	Kotlin	Sugar for `val x = remember { … }.value`; Kotlin handles the getter/setter
`@State` / `@Binding` property wrappers	Swift	Like a Java annotation that auto-generates a getter/setter backed by SwiftUI storage
`AsyncValue<T>` (Riverpod)	Dart	A wrapper that is either `loading`, `error(e)`, or `data(value)` — like a typed Result

Everything else reuses Java/JS/Dart syntax from L02. If something looks unfamiliar, ask!

War Story: The Phantom Rating Bug

> "The app showed the old restaurant rating for 30 seconds after the user submitted a new one — but only on Android, only after rotating the phone."

- Root cause: state stored in a local variable, not in a ViewModel (View + Model combined)
- Fix: 4 lines
- Debug time: 6 hours

Question: Before we dive in — where do *you* think state should live?

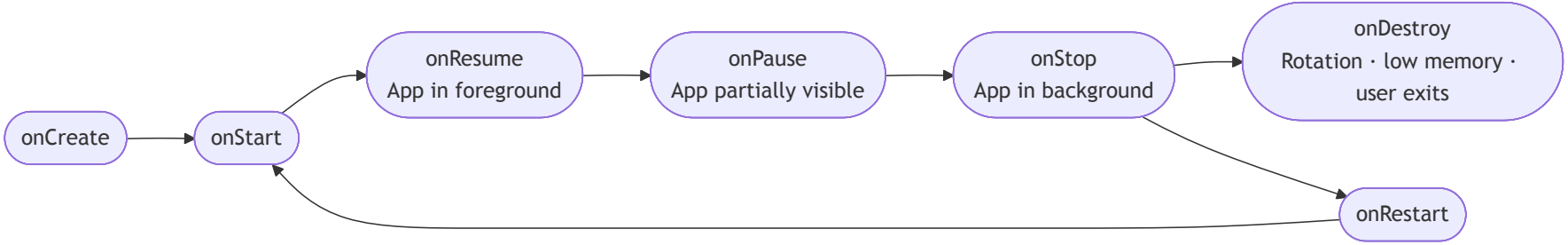
Agenda

1. **Why State Management Exists** — the lifecycle problem
2. **The Concept** — Ephemeral vs App State; UDF (Unidirectional Data Flow); Single Source of Truth
3. **Four Dialects** — remember/StateFlow · @State/@Observable · setState/Riverpod · useState/Zustand
4. **State Hoisting** — the universal pattern
5. **Architecture Layer** — ViewModel, Repository, UiState (User Interface State)
6. **Spot the AI Bug** — stale closures, recomposition loops, retained ViewModels
7. **Interview Corner**
8. **Demo time**

Part 1: Why State Management Exists

The Lifecycle Problem

Android Activity Lifecycle



The danger zone: Rotating the phone calls `onDestroy` then `onCreate` — your local variables are gone.

The Rotation / Process-Death Story

Scenario 1 — Rotation (configuration change):

- Activity destroyed and recreated
- ViewModel **survives** (Jetpack holds it)
- `remember { }` state in Compose **does NOT survive** (local to composition)
- `rememberSaveable { }` **does survive** (saved to Bundle)

Scenario 2 — Process death (OS (Operating System) kills app under memory pressure):

- Everything in memory is gone, including the ViewModel
- `SavedStateHandle` in the ViewModel **survives** (written to Bundle before kill)
- Room / SQLite / UserDefaults / SharedPreferences / DataStore **survive** (on disk)

```
// SavedStateHandle: survives BOTH rotation AND process death
class PlaceDetailViewModel @Inject constructor(
    savedStateHandle: SavedStateHandle,  // injected by Hilt/Jetpack
) : ViewModel() {
    private val placeId = savedStateHandle.get<String>("placeId")!!
    //                     ↑ Like a Map<String, Any> backed by the OS Bundle
}
```

iOS ScenePhase & App Suspension

```swift
// SwiftUI — observe app foreground/background transitions
struct UniEatsApp: App {
    @Environment(\.scenePhase) private var scenePhase
    //            ↑ reads a value from SwiftUI's environment

    var body: some Scene {
        WindowGroup { ContentView() }
            .onChange(of: scenePhase) { _, newPhase in
                switch newPhase {
                case .active:
                    // App is in the foreground — resume timers, refresh data
                    print("Active")
                case .inactive:
                    // Transitioning (e.g. Control Center open) — pause work
                    print("Inactive")
                case .background:
                    // App may be suspended soon — save state NOW
                    print("Background — persist unsaved changes")
                @unknown default: break
                }
            }
    }
}
```

iOS model: apps are suspended (frozen in RAM (Random Access Memory)), not destroyed when you leave them. But the OS can silently end a suspended app — **state in memory is not guaranteed**.

Flutter & React Native: App Lifecycle

```dart
// Flutter — WidgetsBindingObserver
class _PlaceListState extends State<PlaceListScreen>
    with WidgetsBindingObserver {   // mixin = interface with default impl

  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addObserver(this);
  }

  @override
  void dispose() {
    WidgetsBinding.instance.removeObserver(this);
    super.dispose();
  }

  @override
  void didChangeAppLifecycleState(AppLifecycleState state) {
    // states: resumed | inactive | paused | detached
    if (state == AppLifecycleState.paused) {
      // save unsaved user input
    }
  }
}
```

```tsx
// React Native — AppState API
import { AppState, AppStateStatus } from 'react-native';
import { useEffect, useRef } from 'react';

function useSaveOnBackground(save: () => void) {
  const appState = useRef<AppStateStatus>(
    AppState.currentState   // 'active' | 'background' | 'inactive'
  );

  useEffect(() => {
    const sub = AppState.addEventListener(
      'change',
      (nextState) => {
        if (
          appState.current === 'active' &&
          nextState.match(/inactive|background/)
        ) {
          save();
        }
        appState.current = nextState;
      },
    );
    return () => sub.remove();   // cleanup = like removeObserver
  }, [save]);
}
```

Lifecycle Summary: All Four Platforms

Event	Android	iOS	Flutter	React Native
App starts	`onCreate`	`scene(.active)`	`resumed`	`active`
App to background	`onStop`	`scene(.background)`	`paused`	`background`
App killed silently	Process death	Silent termination	`detached`	Process death
Save state here	`onStop` / `SavedStateHandle`	`scene(.background)`	`paused`	`background`
State that survives rotation	`ViewModel`	`@State` VM	Riverpod / Provider	Zustand store
State that survives process death	`SavedStateHandle` + disk	`UserDefaults` / disk	`SharedPreferences` / disk	`AsyncStorage` / disk

Why This Leads to State Management

The lifecycle problem creates three hard questions:

1. Where does state live so it survives **rotation**?
2. Where does state live so it survives **process death**?
3. How do we avoid **duplicating** state across the screen and its data holder?

```
Answer:
  ephemeral state  →  component-local (dies with component — that is fine)
  session state    →  ViewModel (survives rotation, in-process)
  persistent state →  SavedStateHandle + on-disk storage (survives everything)
```

UDF (Unidirectional Data Flow) + single source of truth make these three layers work together.

This is the "why" behind every pattern in this lecture.

Part 2: The State Management Patterns

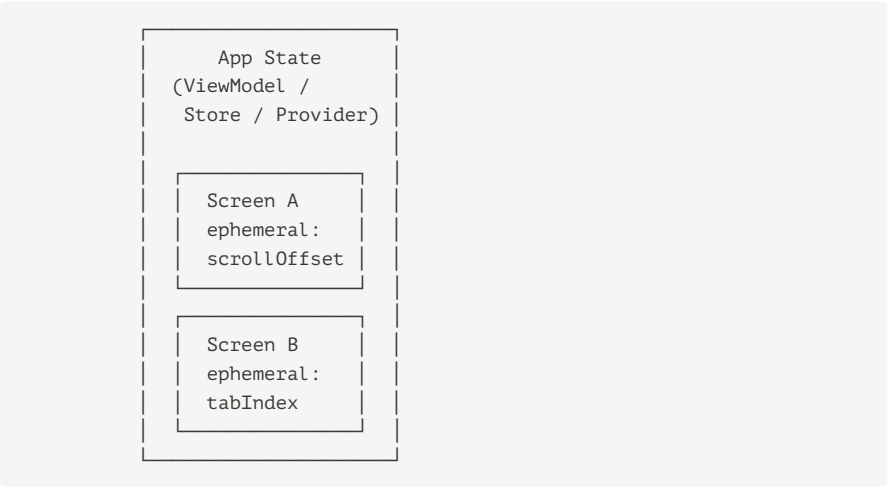
Ephemeral vs App State

Ephemeral (local) state

- Lives inside one widget or composable
- Nobody else cares about it
- Examples: animation toggle, text field focus, scroll offset, tab selection within a screen

App (shared) state

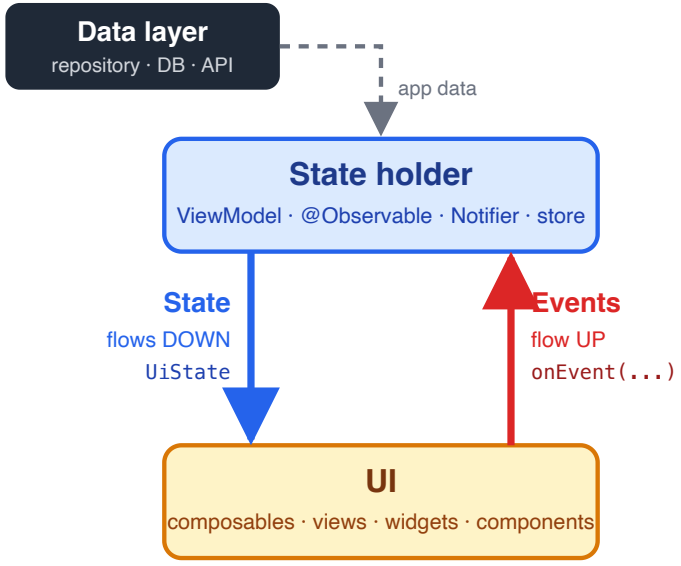
- Survives navigation, configuration changes, or process death
- Shared across multiple screens
- Examples: logged-in user, cart items, fetched feed, theme preference



UDF (Unidirectional Data Flow)

The loop

one direction, every stack



UI never mutates state. It only sends events and re-renders.

One tap in UniEats

favourite a spot, step by step

1. **User taps ♥ on Pizza Stop**
   UI emits an event, nothing else changes yet
2. **Event goes up**
   `viewModel.toggleFavourite("pizza-stop")`
3. **State holder builds a new state**
   `state.copy(favouriteIds = ids + "pizza-stop")`
4. **New state flows down**
   list card and detail screen read the same UiState
5. **UI re-renders: ♥ is filled everywhere**
   no screen told another screen, they share one source

Same state + same event ⇒ same result
bugs reproduce, tests are just events in and states out

State flows DOWN. Events flow UP. Never the reverse.

Single Source of Truth

> One place owns each piece of data. Everyone else reads from it.

Why it matters:

- No sync problems between two copies of the same fact
- Consistent UI across screens — no ghost stale values
- Offline-first becomes straightforward: the local DB (Database) is the source; the network refreshes it

Common violations that AI tools generate:

```
// WRONG — two sources of truth
var localRating by remember { mutableStateOf(place.rating) }  // copy 1
// "by remember { mutableStateOf(...) }" is Kotlin delegation sugar —
// identical to: val _s = remember { mutableStateOf(...) }; var localRating
//               get() = _s.value; set(v) { _s.value = v }
val viewModelRating by viewModel.rating.collectAsState()       // copy 2
// They drift the moment the user submits a new rating
```

The Four Dialects at a Glance

Concept	Android (Compose)	iOS (SwiftUI)	Flutter	React Native
Ephemeral state	`remember { mutableStateOf() }`	`@State`	`setState()` in `StatefulWidget`	`useState()`
Observed state	`StateFlow.collectAsState()`	`@Observable` class	`StreamBuilder` / Riverpod	`useReducer` / Zustand
State holder	`ViewModel`	`@Observable` ViewModel-equivalent	Riverpod `Notifier` / `ChangeNotifier`	Zustand store / Redux
Lifecycle-safe collection	`repeatOnLifecycle`	automatic (View body)	`StreamBuilder` disposes itself	`useEffect` cleanup
State hoisting pattern	value + onValueChange lambda	`@Binding`	callback prop	callback prop

What is a Modifier? (Compose)

> A `Modifier` is a chain of instructions you attach to a composable to change how it looks or behaves — without changing the composable's own code.

```
// Think of a Modifier like a Java Builder that decorates a view:
Text("Hello")                          // bare text
Text("Hello", modifier = Modifier
    .padding(16.dp)                    // add spacing around it
    .background(Color.Yellow)          // paint the background
    .clickable { println("tapped") }   // make it respond to taps
)
```

- Order matters: `.padding(8.dp).background(Color.Red)` ≠ `.background(Color.Red).padding(8.dp)`
- Every composable accepts an optional `modifier` parameter — it is the standard extension point
- L02 showed layout composables (`Column`, `Row`); Modifiers are how you customize their size, spacing, click behavior, and more

We will see Modifiers in every code example from here on. You do not need to memorize them all — read the name and you will usually guess what it does.

Android: Ephemeral State

```
@Composable
fun RatingBar(placeId: String) {
    // "by remember { mutableStateOf(-1) }" is delegation sugar:
    //   Kotlin calls mutableStateOf(-1).getValue/setValue for us,
    //   so we can write "hoverIndex = 2" instead of "hoverIndex.value = 2"
    var hoverIndex by remember { mutableStateOf(-1) }

    Row {
        (1..5).forEach { star ->
            Icon(
                imageVector = if (star <= hoverIndex + 1)
                    Icons.Filled.Star else Icons.Outlined.Star,
                contentDescription = "Star $star",
                modifier = Modifier          // Modifier: see "What is a Modifier?" slide
                    .pointerInput(Unit) {    // pointerInput: intercepts touch gestures
                        detectTapGestures(
                            onPress = { hoverIndex = star - 1 },
                        )
                    }
            )
        }
    }
}
```

`hoverIndex` only affects this composable's visual feedback — no other screen needs it.

Android: App State — ViewModel + StateFlow

```
// data class: like a Java POJO (Plain Old Java Object) with equals/hashCode/copy auto-generated
data class PlaceDetailUiState(
    val place: Place? = null,
    val userRating: Int = 0,
    val isLoading: Boolean = false,
    val errorMessage: String? = null,
)

@HiltViewModel  // Hilt = Jetpack DI (Dependency Injection); covered in L12
class PlaceDetailViewModel @Inject constructor(
    private val placesRepo: PlacesRepository,
    savedStateHandle: SavedStateHandle,   // survives process death
) : ViewModel() {

    private val placeId = savedStateHandle.get<String>("placeId")!!

    // MutableStateFlow is private (writable); public surface is read-only StateFlow
    private val _uiState = MutableStateFlow(PlaceDetailUiState(isLoading = true))
    val uiState: StateFlow<PlaceDetailUiState> = _uiState.asStateFlow()
    //           ↑ StateFlow<T> = like a JS Observable that always has a current value

    init {
        viewModelScope.launch {  // coroutine scoped to ViewModel lifetime
            placesRepo.getPlace(placeId)
                .catch { e -> _uiState.update { it.copy(errorMessage = e.message) } }
                .collect { place -> _uiState.update { it.copy(place = place, isLoading = false) } }
        }
    }

    fun submitRating(stars: Int) {
        viewModelScope.launch {
            _uiState.update { it.copy(userRating = stars) }
            placesRepo.submitRating(placeId, stars)
        }
    }
}
```

Android: Collecting State in the UI

```
@Composable
fun PlaceDetailScreen(viewModel: PlaceDetailViewModel = hiltViewModel()) {
    // collectAsStateWithLifecycle: pauses collection when app is backgrounded
    // (saves battery vs plain collectAsState which runs even when invisible)
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    when {
        uiState.isLoading -> CircularProgressIndicator()
        uiState.errorMessage != null -> ErrorView(uiState.errorMessage!!)
        uiState.place != null -> PlaceContent(
            place = uiState.place!!,
            userRating = uiState.userRating,
            onRatingSelected = viewModel::submitRating,
            // "viewModel::submitRating" is a method reference — like Java's
        )
    }
}
```

`collectAsStateWithLifecycle()` — stops collecting when the app is in the background, saving battery.

iOS: Ephemeral State with @State

```
struct RatingBar: View {
    let onCommit: (Int) -> Void
    // @State = property wrapper that allocates storage OUTSIDE the struct
    // (structs are value types — they would be re-created on every render otherwise)
    // Equivalent mental model: Kotlin's remember { mutableStateOf(-1) }
    @State private var hoverIndex: Int = -1

    var body: some View {
        HStack {
            ForEach(1...5, id: \.self) { star in
                Image(systemName: star <= hoverIndex + 1 ? "star.fill" : "star")
                    .foregroundColor(.yellow)
                    .onTapGesture { hoverIndex = star - 1 }
            }
            .simultaneousGesture(
                DragGesture(minimumDistance: 0)
                    .onEnded { _ in onCommit(hoverIndex + 1) }
            )
        }
    }
}
```

`@State` is private to this view. SwiftUI re-renders when it changes — same idea as `remember`.

iOS: App State with @Observable

```swift
// @Observable is a Swift macro (like a Kotlin annotation processor)
// It instruments every property so SwiftUI knows WHICH ones a view reads,
// and only re-renders that view when those specific properties change.
@Observable
final class PlaceDetailViewModel {
    private(set) var place: Place?        // private(set) = Kotlin's "val" on the outside
    private(set) var userRating: Int = 0
    private(set) var isLoading: Bool = false
    private(set) var errorMessage: String?

    private let repo: PlacesRepository
    private let placeId: String

    init(placeId: String, repo: PlacesRepository) {
        self.placeId = placeId
        self.repo = repo
        Task { await loadPlace() }  // Task = Swift's coroutine launch
    }

    private func loadPlace() async {
        isLoading = true
        do {
            place = try await repo.getPlace(id: placeId)
        } catch {
            errorMessage = error.localizedDescription
        }
        isLoading = false
    }

    func submitRating(_ stars: Int) async {
        userRating = stars
        try? await repo.submitRating(placeId: placeId, stars: stars)
    }
}
```

Flutter: Ephemeral State — setState

```dart
class RatingBar extends StatefulWidget {
  final void Function(int) onCommit;
  const RatingBar({super.key, required this.onCommit});

  @override
  State<RatingBar> createState() => _RatingBarState();
}

class _RatingBarState extends State<RatingBar> {
  int _hoverIndex = -1;
  // Leading underscore = private in Dart (like "private" in Java)

  @override
  Widget build(BuildContext context) {
    return Row(
      children: List.generate(5, (i) {
        return GestureDetector(
          onTap: () {
            setState(() => _hoverIndex = i);
            // setState() is REQUIRED — mutating _hoverIndex directly
            // won't trigger a rebuild (same idea as notifyListeners)
            widget.onCommit(i + 1);
            // "widget." accesses the parent StatefulWidget's fields
          },
          child: Icon(
            i <= _hoverIndex
                ? Icons.star
                : Icons.star_border,
            color: Colors.amber,
          ),
        );
      }),
    );
  }
}
```

Key points

- `StatefulWidget` + `State<T>` is Flutter's ephemeral state primitive
- `setState()` schedules a rebuild of **this widget only**
- State is private to `_RatingBarState`
- Pass events up via callback (`onCommit`)
- `widget.onCommit` accesses the parent's callback

`setState` is like `remember { mutableStateOf() }` + assignment in Compose.

Flutter: App State — Riverpod (2026 standard)

```
// Immutable state with a hand-written copyWith (Dart has no data class)
class PlaceDetailState {
  const PlaceDetailState({required this.place, this.userRating = 0});
  final Place place;
  final int userRating;

  PlaceDetailState copyWith({int? userRating}) => PlaceDetailState(
        place: place,
        userRating: userRating ?? this.userRating,
      );
}

// AsyncNotifier = Riverpod's ViewModel equivalent for async data
class PlaceDetailNotifier extends AsyncNotifier<PlaceDetailState> {
  @override
  Future<PlaceDetailState> build() async {
    final placeId = ref.watch(placeIdProvider);
    // ref.watch = reactive read: rebuilds if placeIdProvider changes
    final repo = ref.watch(placesRepositoryProvider);
    return PlaceDetailState(place: await repo.getPlace(placeId));
  }

  Future<void> submitRating(int stars) async {
    // AsyncData wraps the value in Riverpod's loading/error/data union
    state = AsyncData(state.requireValue.copyWith(userRating: stars));
    await ref.read(placesRepositoryProvider)
        .submitRating(state.requireValue.place.id, stars);
  }
}

final placeDetailProvider =
    AsyncNotifierProvider<PlaceDetailNotifier, PlaceDetailState>(
      PlaceDetailNotifier.new,
    );
```

UniEats at `l03-state` uses the synchronous sibling `Notifier<SpotListUiState>` (seed data, nothing to await, no code generation); `AsyncNotifier` arrives with REST in L05.

React Native: Ephemeral State — useState

```tsx
import { useState } from 'react';
import { View, Pressable } from 'react-native';
import { Ionicons } from '@expo/vector-icons';

function RatingBar({ onCommit }: { onCommit: (n: number) => void }) {
  // useState returns [currentValue, setter] — you already know this from React/JS
  const [hoverIndex, setHoverIndex] = useState(-1);

  return (
    <View style={{ flexDirection: 'row' }}>
      {[0, 1, 2, 3, 4].map((i) => (
        <Pressable
          key={i}
          onPress={() => {
            setHoverIndex(i);
            onCommit(i + 1);
          }}
        >
          <Ionicons
            name={i <= hoverIndex ? 'star' : 'star-outline'}
            size={24}
            color="#F59E0B"
          />
        </Pressable>
      ))}
    </View>
  );
}
```

Key points

- `useState` returns `[value, setter]`
- Setter triggers a re-render of this component
- State is private to this component instance
- Events bubble up via the `onCommit` prop callback
- The pattern is identical in concept to `@State` and `remember`

React Native: App State — Zustand

```
import { create } from 'zustand';

interface PlaceDetailState {
  place: Place | null;
  userRating: number;
  isLoading: boolean;
  errorMessage: string | null;
  loadPlace: (placeId: string) => Promise<void>;
  submitRating: (stars: number) => Promise<void>;
}

const usePlaceDetailStore = create<PlaceDetailState>()((set, get) => ({
  place: null,
  userRating: 0,
  isLoading: false,
  errorMessage: null,

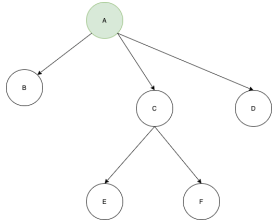
  loadPlace: async (placeId) => {
    set({ isLoading: true });
    try {
      const place = await placesApi.getPlace(placeId);
      set({ place, isLoading: false });
    } catch (e: any) {
      set({ errorMessage: e.message, isLoading: false });
    }
  },

  submitRating: async (stars) => {
    set({ userRating: stars });
    await placesApi.submitRating(get().place!.id, stars);
  },
}));
```

`set({ … })` shallow-merges into the current state — no spreads, no middleware. UniEats' `useSpotsStore` (`src/store/spotsStore.ts`) is written the same way.

State Hoisting — The Universal Pattern

Rule: Move state to the *lowest common ancestor* of all consumers.




```
// BEFORE — state trapped inside child
@Composable
fun SearchBar() {
    var query by remember { mutableStateOf("") }
    TextField(value = query,
              onValueChange = { query = it })
    // Parent can never read query!
}

// AFTER — state hoisted to caller
@Composable
fun SearchBar(
    query: String,
    onQueryChange: (String) -> Unit,
    // "(String) -> Unit" = lambda type, like Java's Consumer<String>
) {
    TextField(value = query,
              onValueChange = onQueryChange)
}

@Composable
fun PlaceListScreen() {
    var query by remember { mutableStateOf("") }
    SearchBar(query = query, onQueryChange = { query = it })
    PlaceList(filter = query) // can now read query!
}
```

```
// BEFORE — state trapped
struct SearchBar: View {
    @State private var query = ""
    var body: some View {
        TextField("Search", text: $query)
    }
    // parent cannot see query
}

// AFTER — state hoisted via @Binding
// @Binding = a reference to @State owned by the parent
// equivalent to Kotlin's "value + onValueChange" pair
struct SearchBar: View {
    @Binding var query: String
    var body: some View {
        TextField("Search", text: $query)
    }
}

struct PlaceListScreen: View {
    @State private var query = ""
    var body: some View {
        SearchBar(query: $query)
        // "$query" passes a Binding (reference), not the value
        PlaceList(filter: query)
    }
}
```

State Hoisting — Flutter & React Native

```dart
// Flutter — callback prop pattern
class SearchBar extends StatelessWidget {
  // StatelessWidget: no internal state — parent controls it
  final String query;
  final ValueChanged<String> onQueryChanged;
  // ValueChanged<String> = Dart shorthand for void Function(String)

  const SearchBar({
    super.key,
    required this.query,
    required this.onQueryChanged,
  });

  @override
  Widget build(BuildContext context) {
    return TextField(
      controller: TextEditingController(text: query),
      onChanged: onQueryChanged,
    );
  }
}

// Parent owns state
class PlaceListScreen extends StatefulWidget { ... }
class _PlaceListScreenState
    extends State<PlaceListScreen> {
  String _query = '';

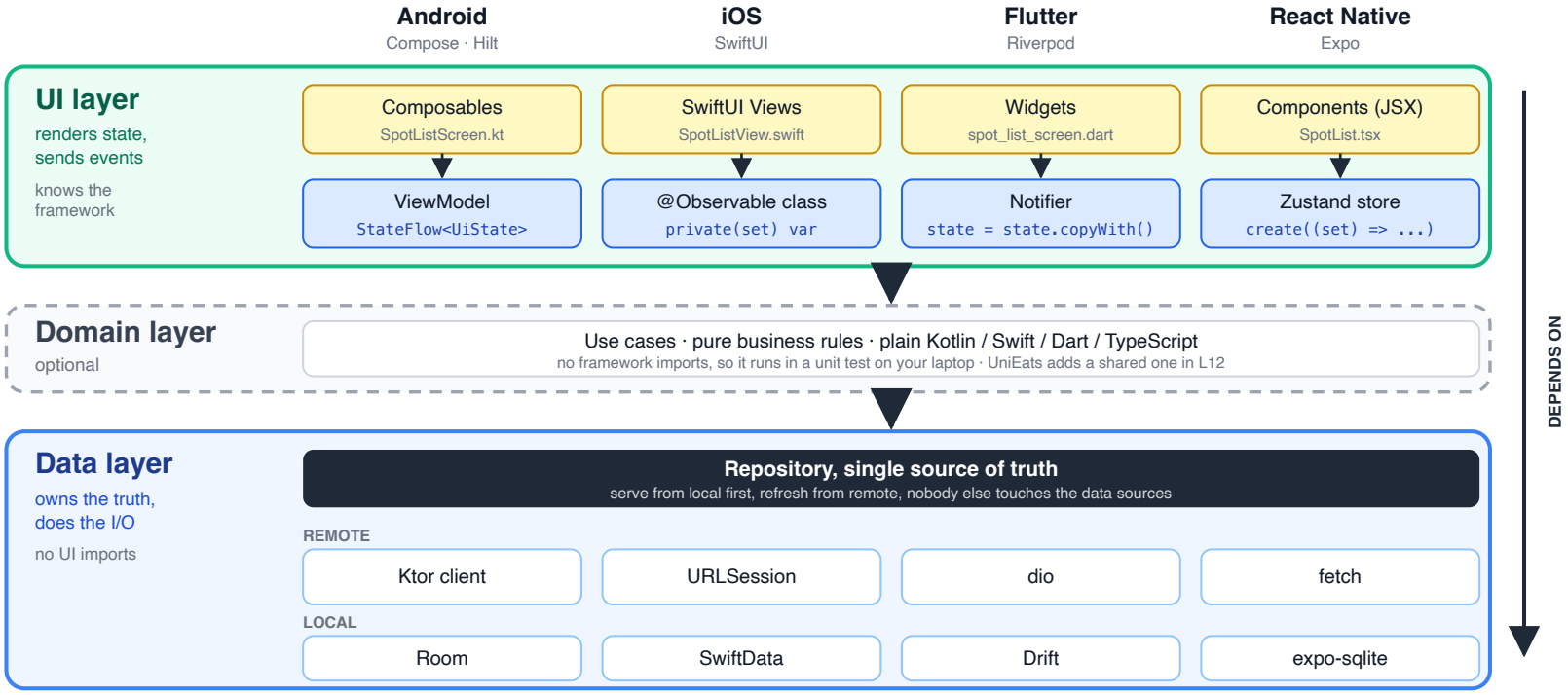
  @override
  Widget build(BuildContext context) => Column(children: [
    SearchBar(query: _query,
              onQueryChanged: (v) =>
                  setState(() => _query = v)),
    PlaceList(filter: _query),
  ]);
}
```

```tsx
// React Native — props pattern
function SearchBar({
  query,
  onQueryChange,
}: {
  query: string;
  onQueryChange: (v: string) => void;
}) {
  return (
    <TextInput
      value={query}
      onChangeText={onQueryChange}
      placeholder="Search places..."
    />
  );
}

// Parent owns state
function PlaceListScreen() {
  const [query, setQuery] = useState('');

  return (
    <View>
      <SearchBar
        query={query}
        onQueryChange={setQuery}
      />
      <PlaceList filter={query} />
    </View>
  );
}
```

Architecture: The Three Layers



Dependencies flow **strictly downward**: UI → Domain → Data. Never upward.

ViewModel: How Each Platform Implements It

Android — Jetpack ViewModel

- Survives rotation automatically
- Scoped to the screen or nav graph
- Owned by `ViewModelStore` in Jetpack

```
@HiltViewModel
class PlaceListViewModel @Inject constructor(
    private val repo: PlacesRepository,
) : ViewModel() {
    val uiState = MutableStateFlow(PlaceListUiState())
        .asStateFlow()
}
```

iOS — @Observable class

- No built-in "rotation survival" (iOS suspends, not destroys)
- Created with `@State` in the owning View
- Lifetime matches the View's lifetime

```
@Observable final class PlaceListViewModel {
    private(set) var uiState = PlaceListUiState()
    private let repo: PlacesRepository
    init(repo: PlacesRepository) { self.repo = repo }
}
```

ViewModel: Flutter & React Native

Flutter — Riverpod AsyncNotifier

- Provider survives widget rebuilds
- `ref.invalidate()` to reset
- Scoped to the provider tree

```
class PlaceListNotifier
    extends AsyncNotifier<PlaceListState> {
  @override
  Future<PlaceListState> build() async {
    final repo = ref.watch(
      placesRepositoryProvider);
    final places = await repo.getPlaces();
    return PlaceListState(places: places);
  }
}

final placeListProvider = AsyncNotifierProvider<
    PlaceListNotifier, PlaceListState>(
  PlaceListNotifier.new,
);
```

React Native — Zustand store

- Store is a singleton (global by default)
- Survives re-renders automatically
- Can be scoped with React context

```
const usePlaceListStore =
  create<PlaceListState & Actions>()((set) => ({
    places: [],
    isLoading: false,
    load: async () => {
      set({ isLoading: true });
      const places = await api.getPlaces();
      set({ places, isLoading: false });
    },
  }));
```

The UiState Pattern

Model everything a screen needs in one immutable data class:

```
// Android — data class (auto-generates equals/hashCode/copy/toString)
data class PlaceListUiState(
    val places: List<Place> = emptyList(),
    val searchQuery: String = "",
    val isLoading: Boolean = false,
    val errorMessage: String? = null,
)
```

```
// iOS — struct (Swift structs are value types, like Kotlin data classes)
struct PlaceListUiState {
    var places: [Place] = []
    var searchQuery: String = ""
    var isLoading: Bool = false
    var errorMessage: String? = nil
}
```

Benefits: single observable, no flag soup, trivial to snapshot-test, easy for AI to extend correctly.

Repository Pattern

```
interface PlacesRepository {
    fun getPlacesStream(query: String): Flow<List<Place>>
    // Flow<T> = a stream that emits values over time, like RxJava Observable or JS AsyncIterable
    suspend fun refreshPlaces()
    suspend fun submitRating(placeId: String, stars: Int)
}

class DefaultPlacesRepository @Inject constructor(
    private val remote: PlacesRemoteDataSource,
    private val local: PlacesLocalDataSource,
) : PlacesRepository {

    override fun getPlacesStream(query: String): Flow<List<Place>> =
        local.getPlaces(query)  // Room emits on every DB change automatically

    override suspend fun refreshPlaces() {
        val fetched = remote.fetchPlaces()
        local.upsertPlaces(fetched)  // Room notifies active collectors
    }

    override suspend fun submitRating(placeId: String, stars: Int) {
        remote.submitRating(placeId, stars)
        local.updateRating(placeId, stars)  // immediate local optimistic update
    }
}
```

Part 3: Spot the AI Bug

Spot the AI Bug — Can You Find It?

```
@Composable
fun PlaceListScreen() {
    val viewModel: PlaceListViewModel = viewModel()
    val places by viewModel.places.collectAsState()
    var searchQuery by remember { mutableStateOf("") }

    // LaunchedEffect: runs a coroutine when its key (searchQuery) changes
    // Think of it like a "run this block whenever this value changes"
    LaunchedEffect(searchQuery) {
        viewModel.search(searchQuery)
    }

    Column {
        TextField(
            value = searchQuery,
            onValueChange = { searchQuery = it },
            label = { Text("Search") }
        )
        LazyColumn {
            items(places) { place -> PlaceCard(place) }
        }
    }
}
```

What is wrong with this code?

Spot the AI Bug — Revealed

The bug: two sources of truth for the search query

```
// In the ViewModel
private val _places = MutableStateFlow<List<Place>>(emptyList())
val places: StateFlow<List<Place>> = _places.asStateFlow()

fun search(query: String) {
    viewModelScope.launch {
        _places.value = repository.search(query)
    }
}
```

The ViewModel's `places` reflects the *last search*, but `searchQuery` in the composable is the *current input*. On rotation:

1. ViewModel is kept with its last search result
2. Composable resets `searchQuery` to `""`
3. `LaunchedEffect("")` fires, clearing results
4. User sees their previous results disappear

Fix: Move `searchQuery` into the ViewModel as part of `UiState`.

Stale Closures in React Native

```
// BUGGY — classic AI-generated stale closure
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setCount(count + 1); // 'count' captured at creation time = 0
      // This is standard JS closure scoping — the function "closes over"
      // the value of count that existed when useEffect ran (0)
    }, 1000);
    return () => clearInterval(interval);
  }, []); // empty deps — effect runs once, closes over count=0 forever

  return <Text>{count}</Text>;
}
// Result: count jumps to 1 and stays there
```

Fix: Use the functional updater form — no closure dependency on `count`:

```
useEffect(() => {
  const interval = setInterval(() => {
    setCount((prev) => prev + 1); // reads latest state at call time
  }, 1000);
  return () => clearInterval(interval);
}, []);
```

Why AI State Code Breaks — Summary

Pattern	Root cause	Symptom
Two sources of truth	AI copies state into local `var` for "convenience"	Stale data after rotation
`SideEffect` with state mutation	AI misuses Compose effect APIs	Infinite recomposition, freeze
Stale closure in `useEffect`	JS closure scoping + `[]` deps	Counter stuck, events lost
Activity reference in ViewModel	AI confuses ViewModel scope	Memory leak and crash on rotation
Missing `collectAsStateWithLifecycle`	AI uses `collectAsState` from older docs	Background battery drain
`ref.read` instead of `ref.watch`	AI confuses Riverpod read/watch	UI (User Interface) never updates

The pattern: AI tools reproduce code from older tutorials. State management APIs changed a lot between 2020 and 2026.

🖥️ Demo — UniEats State: Spot → ViewModel → UI

> Search "**pizza**" → only Pizza Stop is left. **Rotate** the emulator: the Activity really is recreated (the manifest has no `configChanges`), yet the query and the filtered list survive — they live in one `StateFlow<SpotListUiState>` behind a private `MutableStateFlow`. Pizza Stop exists in all four stacks, so the same search works everywhere.

Repo: `github.com/dancojocar/ma` @ tag `l03-state` · from the repo root: `./unieats/android/gradlew -p unieats/android :app:installDebug`

What this lecture adds: `git diff l02-ui l03-state -- unieats/<stack>/`

Stack	New / changed at `l03-state`
Android	`ui/spotlist/SpotListViewModel.kt` (new: `SpotListUiState`, `filterBy`) · `SpotSearchBar.kt` (new, stateless) · `SpotListScreen.kt` · `SpotCard.kt` + `SpotDetailScreen.kt` (heart) · `data/repository/SpotRepository.kt` (new) · Hilt in `UniEatsApplication` / `MainActivity` (one `SpotListViewModel` feeds list and detail)
iOS	`ViewModels/SpotListViewModel.swift` (new: `@Observable`, `private(set)`) · `Views/CategoryFilterView.swift` (new) · `SpotListView.swift` (`.searchable`) · `SpotRow.swift` + `SpotDetailView.swift` (heart)
Flutter	`lib/providers/providers.dart` (new: `SpotListNotifier extends Notifier`) · `spot_list_widgets.dart` (new) · `spot_list_screen.dart` · `spot_detail_screen.dart` + `spot_badges.dart` (heart) · `main.dart` (`ProviderScope`) · `flutter_riverpod`
React Native	`src/store/spotsStore.ts` (new: Zustand `create`, `filterSpots`) · `SearchBar.tsx`, `CategoryChips.tsx`, `FavouriteButton.tsx` (new) · `SpotCard.tsx` + `SpotDetail.tsx` (heart) · `app/(tabs)/spots/index.tsx` · `zustand`

Interview Corner

Q1: Difference between ephemeral and app state?

> Ephemeral: dropdown open, hover index — dies with the component. App: logged-in user, search results — lives in the ViewModel and survives rotation.

Q2: Junior dev makes `MutableStateFlow` public. Problem?

> Any caller can push arbitrary state — breaks single source of truth.

```
private val _uiState = MutableStateFlow(UiState())
val uiState: StateFlow<UiState> = _uiState.asStateFlow()
```

Q3: What is a stale closure? React Native example?

> `useEffect(() => { setInterval(() => setCount(count + 1), 1000); }, []) — count` captured at 0, forever. Fix: `setCount(prev => prev + 1)`.

Quick Quiz — Which Container?

For each piece of state, choose the right container:

State	Container
Whether a dropdown menu is open	?
The currently logged-in user	?
A list of restaurants fetched from the API	?
The scroll offset of a list	?
Whether a network request is in flight	?
The text typed into a search field (shared with results)	?

Summary — The Mental Model

```
UI = f(state)

state lives in:
  ├── ephemeral container  (component/composable/widget)
  │     └── for: animation, focus, hover, local UI toggle
  └── ViewModel / Store / Observable (survives lifecycle)
        └── for: screen data, user input that crosses components

state changes via:
  └── events flowing UP → state holder processes → emits new state DOWN

state travels as:
  └── single immutable UiState object
        ├── no public mutable surfaces
        └── one authoritative source per fact

state survives via:
  ├── ViewModel     — rotation (in-process)
  └── SavedState + disk  — process death
```

You now have the architecture. Next lecture: we add Navigation.

Make State Testable — Push Logic Into Pure Functions

The search bug earlier lived in the UI. Move the *decision* into a plain function and you can test it in milliseconds — no device, no rotation. UniEats already does (`SpotListViewModel.kt` @ `l03-state`):

```
fun List<Spot>.filterBy(query: String, category: Category?): List<Spot> = filter { spot ->
    spot.name.contains(query.trim(), ignoreCase = true) &&
        (category == null || spot.category == category)
}
```

Exercise — UniEats has no unit tests yet; add `testImplementation(kotlin("test"))` and write:

```
class FilterByTest {
    @Test fun `blank query returns every spot`() =
        assertEquals(8, seedSpots.filterBy("", null).size)

    @Test fun `query is trimmed and case-insensitive`() {
        val ids = seedSpots.filterBy(" PIZZA ", null).map { it.id }
        assertEquals(listOf("spot-3"), ids)
    }

    @Test fun `category narrows the result`() {
        val ids = seedSpots.filterBy("", Category.CAFE).map { it.id }
        assertEquals(listOf("spot-2", "spot-7"), ids)
    }
}
```

- This is L14's #1 testing rule, early: **push logic down into pure functions** so fast unit tests cover it.
- `SpotListViewModel` just calls `allSpots.filterBy(…)` inside `_uiState.update { }` . Same idea elsewhere: iOS `filteredSpots` , Flutter `visibleSpots` , RN `filterSpots(…)` .
- Testing the **async** part of a ViewModel (waiting for `viewModelScope.launch` to finish) needs coroutine test tools — **that's L06.**

If You Only Remember 3 Things

1. State in the wrong place = bugs after rotation

Put session state in the **ViewModel**, not in a local composable variable.

2. One source of truth per fact

Never copy state into a local variable "for convenience." Observe the original.

3. Events up, state down — never the reverse

The UI (User Interface) sends events. The ViewModel/store emits new state. The UI renders it.

Practice: UniEats SpotList with State

Ungraded exercise on the demo app — the same patterns your project's UI assignments are graded on

Starting from tag `l02-ui`, rebuild the SpotList screen in **your choice of two stacks** (compare with `l03-state` afterwards) with:

1. Live search query stored in the ViewModel / store (not in a local variable)
2. Offline-first (stretch): show the 8 seed spots instantly, then refresh from the UniEats server's `GET /api/spots` in the background (the app itself only adds networking in L05)
3. Loading and error states reflected in `UiState`
4. A stateless search bar component that receives `query` and `onQueryChange` as parameters
5. Unit test: write a unit test for a **pure** state helper — e.g. `filterBy(query, category)` from the testability slide (blank query returns all 8; a real query filters case-insensitively). No coroutines yet — the async ViewModel test comes after L06.

Bonus: Implement a `SharedFlow`-based one-time snackbar event ("Added to favourites") when a heart is tapped.

Bonus 2: Add an `AppState` / `WidgetsBindingObserver` / `ScenePhase` hook that saves the current search query when the app goes to the background.

Apply it: the rubric for the UI assignments scores the list for *not* rebuilding on every change and the forms for handling errors in place — this exercise is the rehearsal.

Appendix — Self-Study (not presented live)

API Migration Quick Reference

Both iOS and Flutter have a new and an old state API you will encounter in AI-generated code:

	Old (still common)	New (2026 standard)	Key difference
iOS	`ObservableObject` + `@Published`, `@StateObject`	`@Observable` macro, `@State` for VM	Old: re-renders on any `@Published` change. New: fine-grained per-property tracking
Flutter	`ChangeNotifier` + `notifyListeners()`, `context.watch<T>()`	Riverpod `Notifier` / `AsyncNotifier`, `ref.watch(provider)`	Old: manual trigger. New: compile-safe, async-first, no `BuildContext` needed

When AI tools generate iOS SwiftUI code targeting iOS 17+ using `@StateObject` — flag it; `@State` + `@Observable` is correct. When AI generates Flutter state code using `Provider` for a new project — prefer Riverpod.

Key APIs Reference — All Four Stacks

Concept	Android / Kotlin	iOS / Swift	Flutter / Dart	React Native / TS
Ephemeral state	`remember { mutableStateOf(x) }`	`@State`	`setState(() {})`	`useState<T>()`
Survives rotation	`rememberSaveable`	`@State` (iOS suspends)	`RestorableProp`	`AsyncStorage`
ViewModel state	`MutableStateFlow` / `StateFlow`	`@Observable` class	`Notifier<T>` / `AsyncNotifier<T>`	Zustand `create()`
One-time events	`MutableSharedFlow` / `SharedFlow`	closure/callback	`Stream`	callback prop
Lifecycle-safe collect	`collectAsStateWithLifecycle()`	automatic	`StreamBuilder`	`useEffect` cleanup
State hoisting	value + lambda	`@Binding`	callback param	callback prop
Reactive read	`collectAsStateWithLifecycle`	body reads property	`ref.watch(provider)`	`useStore(selector)`
Non-reactive read	—	—	`ref.read(provider)`	`useStore.getState()`
Process death	`SavedStateHandle`	`@SceneStorage`	`SharedPreferences`	`AsyncStorage`
App lifecycle	`repeatOnLifecycle`	`ScenePhase`	`WidgetsBindingObserver`	`AppState.addEventListener`

Items marked "legacy" in AI-generated code: iOS `@StateObject` / `@Published` → use `@Observable`; Flutter `ChangeNotifier` → use Riverpod `Notifier` / `AsyncNotifier` for new projects.

Deep Dive: StateFlow vs SharedFlow

```
// StateFlow — has a current value, replays last item to new collectors
val uiState: StateFlow<UiState>  // always has a value
// Use for: screen state (loading, content, error)

// SharedFlow — configurable replay, no required initial value
val events: SharedFlow<UiEvent>  // fire-and-forget events
// Use for: one-time events (navigation, snackbar, toast)
```

```
// sealed interface: like a Java sealed class / TypeScript discriminated union
// Each variant is a separate type but they all belong to UiEvent
sealed interface UiEvent {
    data object NavigateToDetail : UiEvent  // data object = singleton with equals/toString
    data class ShowSnackbar(val message: String) : UiEvent
}

private val _events = MutableSharedFlow<UiEvent>()
val events: SharedFlow<UiEvent> = _events.asSharedFlow()

fun onPlaceClicked(id: String) {
    viewModelScope.launch {
        _events.emit(UiEvent.NavigateToDetail)
    }
}
```

iOS: Using the ViewModel in a View

```
struct PlaceDetailScreen: View {
    // @State stores the ViewModel outside the struct (same reason as for @State values)
    // @Observable VMs use @State, NOT @StateObject (that is the old pre-iOS-17 pattern)
    @State private var viewModel: PlaceDetailViewModel

    init(placeId: String, repo: PlacesRepository) {
        _viewModel = State(initialValue: PlaceDetailViewModel(
            placeId: placeId, repo: repo
        ))
    }

    var body: some View {
        Group {
            if viewModel.isLoading {
                ProgressView()
            } else if let msg = viewModel.errorMessage {
                Text(msg).foregroundColor(.red)
            } else if let place = viewModel.place {
                PlaceContent(place: place,
                             userRating: viewModel.userRating,
                             onRating: { stars in
                                 Task { await viewModel.submitRating(stars) }
                             })
            }
        }
        .navigationTitle(viewModel.place?.name ?? "")
    }
}
```

Flutter: Consuming Riverpod State

```
// ConsumerWidget = StatelessWidget + WidgetRef (access to providers)
class PlaceDetailScreen extends ConsumerWidget {
  final String placeId;
  const PlaceDetailScreen({super.key, required this.placeId});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final state = ref.watch(placeDetailProvider);
    // ref.watch subscribes this widget — any state change triggers rebuild

    return state.when(
      // AsyncValue.when: like a switch on loading/error/data — from the syntax slide
      loading: () => const CircularProgressIndicator(),
      error: (err, _) => Text('Error: $err'),
      data: (s) => PlaceContent(
        place: s.place,
        userRating: s.userRating,
        onRating: (stars) => ref
            .read(placeDetailProvider.notifier)
            // ref.read in a callback (NOT ref.watch) — we want the action,
            // not to subscribe the button to state changes
            .submitRating(stars),
      ),
    );
  }
}
```

React Native: Consuming Zustand State

```
import { useEffect } from 'react';

function PlaceDetailScreen({ placeId }: { placeId: string }) {
  const { place, userRating, isLoading, errorMessage,
          loadPlace, submitRating } = usePlaceDetailStore();

  useEffect(() => {
    loadPlace(placeId);
    // useEffect with [placeId] = "run this when placeId changes"
    // equivalent to Kotlin ViewModel's init{} for the initial load,
    // or SwiftUI's .task modifier
  }, [placeId]);

  if (isLoading) return <ActivityIndicator />;
  if (errorMessage) return <Text style={{ color: 'red' }}>{errorMessage}</Text>;
  if (!place) return null;

  return (
    <PlaceContent
      place={place}
      userRating={userRating}
      onRating={submitRating}
    />
  );
}
```

UiState: Flutter & React Native

The real UniEats code at `l03-state`:

```dart
// lib/providers/providers.dart
class SpotListUiState {
  const SpotListUiState({
    required this.spots,
    this.searchQuery = '',
    this.categoryFilter,
    this.favouriteIds = const {},
  });

  final List<Spot> spots;
  final String searchQuery;
  final SpotCategory? categoryFilter;
  final Set<String> favouriteIds;

  List<Spot> get visibleSpots { /* … */ }

  SpotListUiState copyWith({
    String? searchQuery,
    Set<String>? favouriteIds,
  }) => /* … */;
}
```

```ts
// src/store/spotsStore.ts
interface SpotsState {
  searchQuery: string;
  categoryFilter: CategoryFilter;
  favouriteIds: Set<string>;
  setSearchQuery: (query: string) => void;
  setCategoryFilter: (c: CategoryFilter) => void;
  toggleFavourite: (id: string) => void;
}

export const useSpotsStore =
  create<SpotsState>()((set) => ({
    searchQuery: "",
    categoryFilter: "all",
    favouriteIds: new Set<string>(),
    setSearchQuery: (searchQuery) =>
      set({ searchQuery }),
    setCategoryFilter: (categoryFilter) =>
      set({ categoryFilter }),
    toggleFavourite: (id) =>
      set((state) => { /* new Set, add/delete */ }),
  }));
```

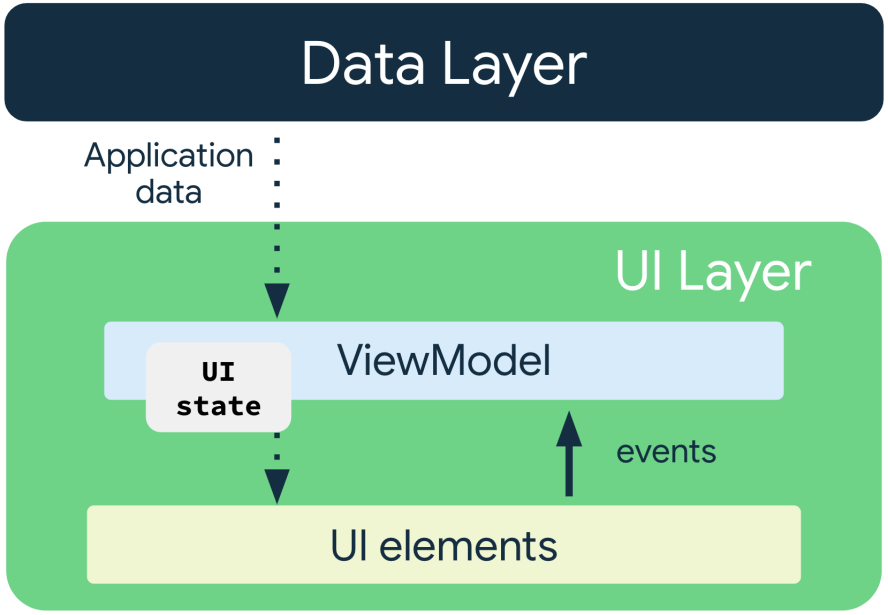
Repository: iOS, Flutter, React Native

```swift
// iOS — async/await protocol
protocol PlacesRepository {
    func getPlaces(query: String) async throws -> [Place]
    func submitRating(placeId: String, stars: Int) async throws
}
class DefaultPlacesRepository: PlacesRepository {
    private let remote: PlacesRemoteSource
    private let local: SwiftDataStore
    func getPlaces(query: String) async throws -> [Place] {
        let cached = try local.search(query)
        if !cached.isEmpty { return cached }
        let fresh = try await remote.fetch(query)
        try local.upsert(fresh)
        return fresh
    }
}
```

```dart
// Flutter — abstract class (Dart equivalent of interface)
abstract class PlacesRepository {
  Stream<List<Place>> watchPlaces(String query);
  Future<void> refreshPlaces();
}
class DefaultPlacesRepository implements PlacesRepository {
  @override
  Stream<List<Place>> watchPlaces(String q) =>
    _local.watchPlaces(q); // Drift Stream — like Room Flow
  @override
  Future<void> refreshPlaces() async {
    final fresh = await _remote.fetchPlaces();
    await _local.upsertPlaces(fresh);
  }
}
```

```typescript
// React Native — TypeScript interface + AsyncStorage cache
interface PlacesRepository {
  getPlaces(): Promise<Place[]>;
  submitRating(placeId: string, stars: number): Promise<void>;
}
class DefaultPlacesRepository implements PlacesRepository {
  async getPlaces(): Promise<Place[]> {
    const raw = await AsyncStorage.getItem('places_cache');
    const cached: Place[] = raw ? JSON.parse(raw) : [];
    // Refresh from network in background — same offline-first pattern
    fetch('https://api.example.com/places')
      .then(r => r.json())
      .then(fresh => AsyncStorage.setItem(
        'places_cache', JSON.stringify(fresh)))
      .catch(() => {});
    return cached;
  }
}
```

Architecture Diagram: UI Layer + ViewModel



State flows from the Data Layer → ViewModel → UI elements. Events flow from UI elements → ViewModel → Data Layer.

Recomposition Loops in Compose

```
// BUGGY — recomposition loop
@Composable
fun AutoSearchScreen(viewModel: SearchViewModel = viewModel()) {
    val results by viewModel.results.collectAsState()

    // SideEffect runs on EVERY successful recomposition
    // (unlike LaunchedEffect which can be keyed to run once)
    SideEffect {
        viewModel.search("uniEats") // triggers StateFlow update
    }                               // which triggers recomposition
                                    // which runs SideEffect again → infinite loop

    LazyColumn {
        items(results) { PlaceCard(it) }
    }
}
```

Fix: Use `LaunchedEffect(Unit)` — runs once per composition entry point:

```
LaunchedEffect(Unit) {
    viewModel.search("uniEats") // runs exactly once
    // Unit as key = "run when this composable enters composition, never again"
}
```

Retained ViewModel Anti-Pattern

```
// BUGGY — ViewModel holding an Activity reference
class PlaceDetailViewModel(
    private val activity: Activity // NEVER do this
) : ViewModel() {
    fun showError(message: String) {
        // Activity is destroyed on rotation — this reference is stale
        // Like keeping a Java reference to an object that was garbage collected,
        // except here the OS destroys it without warning
        Toast.makeText(activity, message, Toast.LENGTH_SHORT).show()
    }
}
```

Why it breaks:

- ViewModel survives configuration change; Activity does not
- The Activity reference becomes stale after rotation
- Memory leak: Activity cannot be garbage collected

Fix: Emit error events as part of `UiState`:

```
_uiState.update { it.copy(errorMessage = "Network error") }
// Let the composable observe and show the Toast/Snackbar
```

Further Reading

- Android Architecture Guide — the definitive reference
- Compose State documentation
- Swift @Observable migration guide
- Riverpod documentation
- Zustand GitHub
- Now In Android sample — real-world UiState + Repository
- Android Activity Lifecycle
- Flutter AppLifecycleState

Video: "Managing State in Jetpack Compose" — Google I/O 2023

Paper: "Unidirectional User Interface Architectures" — Andre Staltz, 2015 (still worth reading)