Lecture 4: Navigation & App Structure

Stacks · Tabs · Drawers · Modals · Deep Links · Adaptive Layouts

UniEats — connecting screens, one route at a time

Syntax You'll Meet Today

Construct	What it is	Anchor
`@Serializable data class SpotDetail(val spotId: String) : NavKey` (Kotlin)	A **key**: a value that names a screen plus its argument; serializable so the back stack can be saved	Like a Java record used as a map key — but it survives process death
`rememberNavBackStack(SpotList)` (Kotlin)	The back stack as a snapshot-state list you own; Compose redraws when it changes	Like a `Deque<Screen>` in Java that fires change events
`entryProvider { entry<SpotDetail> { key -> … } }` (Kotlin)	A function from key to screen content; `entry<T>` registers one key type	Like a `switch` on the key's class
`NavigationStack(path: $path)` (Swift)	The back stack as an array you own (`NavigationPath` , or UniEats' plain `[String]` of spot ids)	Same idea as `rememberNavBackStack` — append to push, removeLast to pop
`builder: (context, state) =>` (Dart)	Named parameters in a lambda	Like `(ctx, st) => ...` in JavaScript (JS) arrow functions
`GoRoute` / `ShellRoute` (Flutter)	Nested route objects that build a URL (Uniform Resource Locator) tree	Like Express `router.use('/spots', spotsRouter)` in Node.js

Hook: The Infamous Back-Button Crash

> *"We shipped to 10 000 users. The back button on Android crashed the app 97% of the time. We had only tested on iOS."* — anonymous developer, Reddit r/androiddev, 2023

- Android has a **system back gesture** — it is not optional
- iOS (iPhone Operating System) has no system back; SwiftUI manages its own stack
- Flutter and React Native must also respect Android's back contract
- Navigation is **platform-specific behaviour** wrapped in cross-platform APIs (Application Programming Interfaces)

The fix was two lines — once you understand back-stack semantics.

Agenda

1. **Navigation as a data structure** — stack, tabs, drawer, modal, deep link; destinations and keys
2. **The back stack** — push, pop, and who owns the back gesture
3. **UniEats in four stacks** — Navigation 3 · NavigationStack · go_router · Expo Router
4. **Deep links** — cold start straight into a screen, with back still working
5. **Scenes** — how Navigation 3 turns a back stack into a layout: single pane, list-detail, dialog
6. **Beyond one stack** — tabs, modals, drawers, adaptive layouts, state restoration
7. **Spot the AI Bug** · checklist · demo · interview corner

Part 1: Navigation as a Data Structure

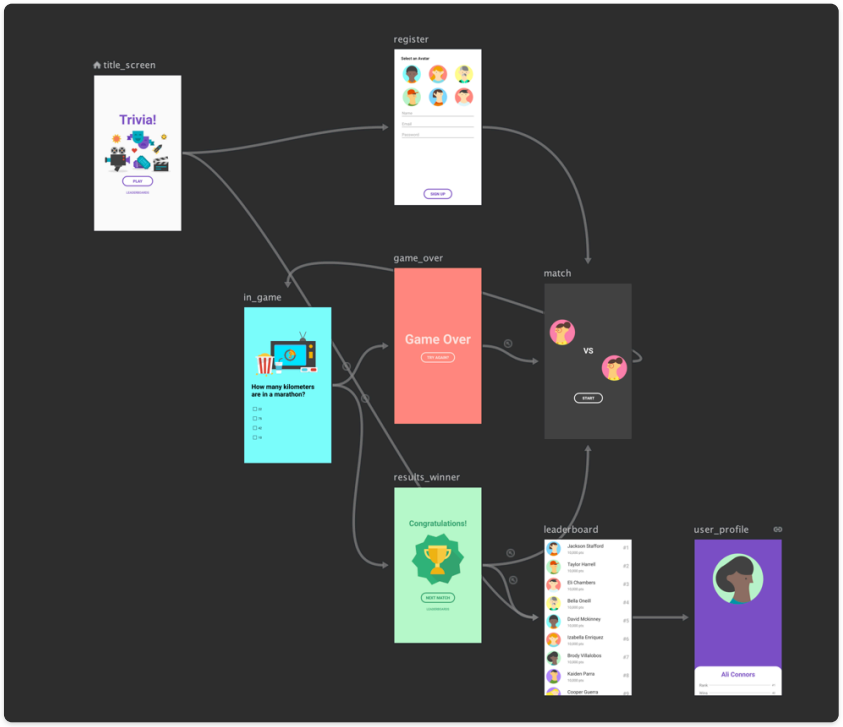
Shapes · destinations · keys

Five Shapes, One Idea

Shape	Data structure	Typical use
Stack	LIFO (Last In, First Out) stack of screens	Drill-down: list → detail → edit
Tab bar / Bottom nav	N parallel stacks	Peer sections: Feed, Search, Profile
Drawer	N parallel stacks + slide-out menu	Admin or secondary navigation
Modal	Ephemeral screen above current stack	Confirmations, forms, pickers
Deep link	Jump to any node, then build the stack underneath	Push notifications, share links

- Every platform names these differently, but **the semantics are the same**
- The OS owns the back gesture — your job is to keep the stack correct

Destinations and Routes



Each card is a **destination**; each arrow is a way to reach it. The app shows one destination at a time; the ones behind it are the back stack.

Navigation 3: The Back Stack Is a List You Own

Three building blocks

```
// 1. Keys name destinations (+ arguments)
@Serializable data object SpotList : NavKey
@Serializable data class SpotDetail(
    val spotId: String) : NavKey

// 2. The back stack: a saved, observable list
val backStack = rememberNavBackStack(SpotList)
backStack.add(SpotDetail("spot-3"))   // push
backStack.removeLastOrNull()          // pop

// 3. NavDisplay renders the list
NavDisplay(
    backStack = backStack,
    onBack = { backStack.removeLastOrNull() },
    entryProvider = entryProvider {
        entry<SpotList> { SpotListScreen(/*…*/) }
        entry<SpotDetail> { key ->
            SpotDetailScreen(spotId = key.spotId) }
    }
)
```

Piece	Role
`NavKey`	A serializable value that names a screen — a `data object` for a screen without arguments, a `data class` for one with arguments
`NavBackStack`	`List<NavKey>` backed by snapshot state — Compose redraws when it changes; `rememberNavBackStack` also saves it across process death
`entryProvider`	key → `NavEntry` (the content for that key, plus optional **metadata**)
`NavDisplay`	Shows the stack: last entry on top, animates changes, wires the system back gesture to `onBack`
entry decorators	Per-entry `rememberSaveable` state and one ViewModel per entry

Navigation 3 is stable since 1.0 (2026-02); UniEats uses 1.2.0. The older `NavHost` + `NavController` is in the appendix — you will meet it in existing code.

Navigation API Mapping

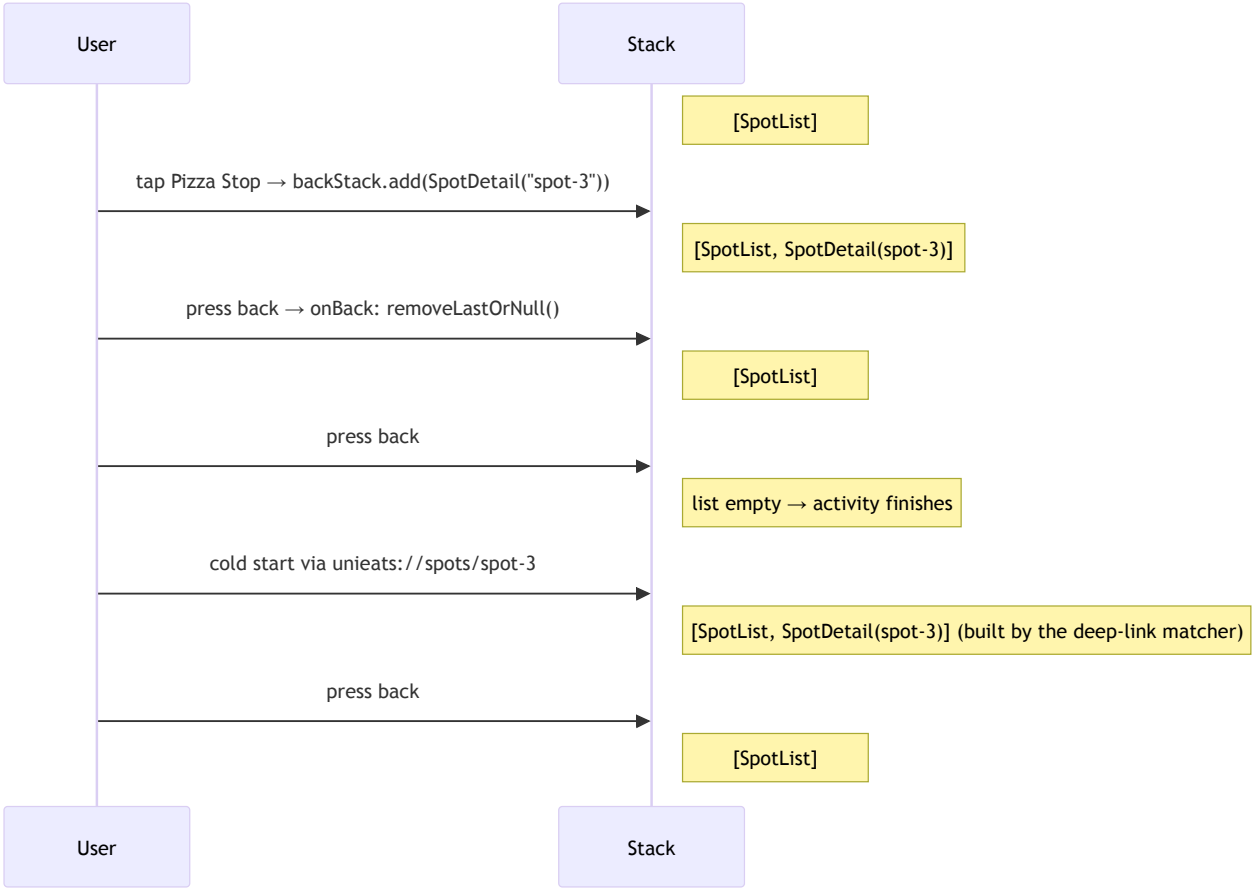
Concept	Android (Navigation 3)	iOS (SwiftUI)	Flutter	React Native
Host / container	`NavDisplay`	`NavigationStack`	`GoRouter` / `Navigator`	`NavigationContainer`
The back stack	`NavBackStack` (a list you own)	`path` array you own	inside `GoRouter`	inside the navigator
Destination name	`@Serializable … : NavKey`	`Hashable` value	path string	file name
Push	`backStack.add(SpotDetail(id))`	`path.append(id)`	`context.push(...)`	`router.push(...)`
Pop	`backStack.removeLastOrNull()`	`path.removeLast()`	`context.pop()`	`router.back()`
Replace stack	`backStack.clear(); add(...)`	`path = [...]`	`context.go(...)`	`router.replace(...)`
Deep link	`UriDeepLinkMatcher` → list of keys	`.onOpenURL` → `path = [id]`	URL path in `GoRoute`	file path is the URL

Memorize the row for your primary platform; recognize the others.

Part 2: The Back Stack

Push, pop, and who owns "back"

Back-Stack Push / Pop in UniEats



The Back Gesture Contract

Platform	Who controls back?	Default behaviour
Android	OS (Operating System) gesture / hardware → `NavDisplay.onBack`	Pops the last key; finishes the activity on an empty list; predictive back previews the pop
iOS	SwiftUI / UIKit (User Interface Kit)	Swipe-left gesture; no hardware button
Flutter	`PopScope` widget	Android gesture intercepted via `PopScope`
React Native	`BackHandler` API	Android only; iOS relies on navigator

Practical rules:

- Android: `onBack` must mutate the same list `NavDisplay` shows; keep it `removeLastOrNull()` unless you have a reason
- Flutter: wrap screens in `PopScope` to intercept back when you need a confirmation dialog
- React Native: `useEffect` + `BackHandler.addEventListener('hardwareBackPress', ...)` on Android
- iOS: customize with `.navigationBarBackButtonHidden(true)` + your own button when needed

Part 3: UniEats in Four Stacks

Same two screens, one stack, ids not objects

UniEats Navigation Map @ `l04-nav`

```
SpotList  (start: search · category filter · favourites from L03)
  └── push SpotDetail(spotId) ──► SpotDetail   (unknown id → "Spot not found")

Deep links, all four stacks ──► SpotDetail, with SpotList underneath   (Part 4)
  ├── unieats://spots/<id>             custom scheme — works everywhere
  └── https://unieats.app/spots/<id>   needs a hosted assetlinks.json / AASA

Android  NavDisplay over rememberNavBackStack · keys SpotList, SpotDetail(spotId)
         + ListDetailSceneStrategy: list | detail side by side on a wide window   (Part 5)
iOS      NavigationStack(path: [String]) · .onOpenURL → path = [spotId]
Flutter  GoRouter /spots → /spots/:id · redirect folds the unieats:// host
RN       app/(tabs): one "Spots" tab → Stack: spots/index, spots/[id]
```

Android: Navigation 3 in UniEats

`ui/navigation/NavGraph.kt` @ `l04-nav` (trimmed):

```
@Serializable data object SpotList : NavKey
@Serializable data class SpotDetail(val spotId: String) : NavKey

@Composable
fun UniEatsNavDisplay(startKeys: List<NavKey>, /* … */
    listViewModel: SpotListViewModel = hiltViewModel()) {
    val backStack = rememberNavBackStack(*startKeys.toTypedArray())
    val listState by listViewModel.uiState.collectAsStateWithLifecycle()

    NavDisplay(
        backStack = backStack,
        onBack = { backStack.removeLastOrNull() },
        entryDecorators = listOf(
            rememberSaveableStateHolderNavEntryDecorator(),
            rememberViewModelStoreNavEntryDecorator()),
        sceneStrategies = listOf(rememberListDetailSceneStrategy()),
        entryProvider = entryProvider {
            entry<SpotList>(metadata = ListDetailSceneStrategy.listPane(
                detailPlaceholder = { SelectASpotPlaceholder() })) {
                SpotListScreen(
                    onSpotClick = { id -> backStack.add(SpotDetail(id)) },
                    viewModel = listViewModel)
            }
            entry<SpotDetail>(metadata = ListDetailSceneStrategy.detailPane()) { key -
                SpotDetailScreen(
                    isFavourite = key.spotId in listState.favouriteIds,
                    onToggleFavourite = { listViewModel.toggleFavourite(key.spotId) },
                    onBack = { backStack.removeLastOrNull() },
                    viewModel = hiltViewModel<SpotDetailViewModel,
                        SpotDetailViewModel.Factory> { it.create(key.spotId) })
            }
        })
}
```

`ui/spotdetail/SpotDetailViewModel.kt` — the key's id goes straight into the ViewModel:

```
@HiltViewModel(assistedFactory = SpotDetailViewModel.Factory::class)
class SpotDetailViewModel @AssistedInject constructor(
    spotRepository: SpotRepository,
    @Assisted private val spotId: String
) : ViewModel() {
    @AssistedFactory
    interface Factory { fun create(spotId: String): SpotDetailViewModel }
    /* uiState = spotRepository.getSpot(spotId) + reviews */
}
```

`MainActivity.kt` — the first list comes from the launch Intent:

```
setContent { UniEatsTheme {
    UniEatsNavDisplay(startKeys = backStackFor(intent))   // Part 4
} }
```

- `backStack.add(SpotDetail(id))` pushes; `removeLastOrNull()` pops — the list **is** the navigation
- `entry<SpotDetail> { key -> … }` receives the typed key; no string parsing
- `rememberViewModelStoreNavEntryDecorator()` → one `SpotDetailViewModel` per entry, cleared on pop
- `listPane()` / `detailPane()` metadata and `sceneStrategies` → Part 5

iOS: SwiftUI NavigationStack + Path

`Views/SpotListView.swift` @ `l04-nav` (trimmed):

```
struct SpotListView: View {
    @State private var viewModel = SpotListViewModel()
    @State private var path: [String] = []

    var body: some View {
        NavigationStack(path: $path) {
            List {
                ForEach(viewModel.filteredSpots) { spot in
                    NavigationLink(value: spot.id) {
                        SpotRow(spot: spot, /* … */)
                    }
                }
            }
            .navigationDestination(for: String.self) {
                spotId in
                SpotDetailView(spotId: spotId, /* … */)
            }
        }
        .onOpenURL { url in
            if let spotId = DeepLink.spotId(from: url) {
                path = [spotId]
            }
        }
    }
}
```

The path **is** the back stack — plain data you own:

```
path.append("spot-3")   // push
path.removeLast()       // pop
path.removeAll()        // back to root
```

- `NavigationStack` is the host; the `path` array is the back stack — exactly Navigation 3's `NavDisplay` + `NavBackStack`
- `NavigationLink(value: spot.id)` pushes the **id**; `navigationDestination(for: String.self)` is the `entryProvider`
- `SpotDetailView(spotId:)` looks the spot up itself; the heart reads the list's `favouriteIds`
- Several destination types? Use an `enum Route: Hashable` or `NavigationPath`
- Swipe-back is free

Flutter: go_router

`lib/app_router.dart` @ `l04-nav`:

```dart
final routerProvider = Provider<GoRouter>((ref) {
  final router = GoRouter(
    initialLocation: '/spots',
    redirect: (context, state) =>
        deepLinkToPath(state.uri),
    routes: [
      GoRoute(path: '/',
          redirect: (_, _) => '/spots'),
      GoRoute(
        path: '/spots',
        builder: (context, state) =>
            const SpotListScreen(),
        routes: [
          GoRoute(
            path: ':id',
            builder: (context, state) =>
                SpotDetailScreen(
                  spotId: state.pathParameters['id']!),
          ),
        ],
      ),
    ],
    errorBuilder: (context, state) => /* … */,
  );
  ref.onDispose(router.dispose);
  return router;
});
```

```dart
context.push('/spots/${spot.id}');   // push
context.pop();                       // pop
context.go('/spots');                // replace the stack
```

- Routes are URL paths; `GoRouter` owns the stack, wired in with `MaterialApp.router`
- `context.push` adds to the stack; `context.go` replaces it
- `:id` is nested under `/spots` → the list sits under the detail, even after a deep link
- The detail gets the **id** and reads the spot from `spotByIdProvider(spotId)`
- `redirect` runs on every route change — deep-link fix-ups (Part 4) and auth guards (L08) go here

React Native: Expo Router

File-system routing — `unieats/rn/app/` **@** `l04-nav`

```
app/
  _layout.tsx       ← root Stack, hosts (tabs)
  index.tsx         ← <Redirect href="/(tabs)/spots" />
  (tabs)/
    _layout.tsx     ← Tabs: one tab, "Spots"
    spots/
      _layout.tsx   ← Stack, initialRouteName "index"
      index.tsx     ← /spots      (list)
      [id].tsx      ← /spots/:id  (detail)
```

```tsx
export default function TabsLayout() {
  return (
    <Tabs screenOptions={{ /* colours */ }}>
      <Tabs.Screen name="spots" options={{
        title: "Spots",
        headerShown: false,
        tabBarIcon: ({ focused }) => /* … */,
      }} />
    </Tabs>
  );
}
```

`app/(tabs)/spots/[id].tsx` :

```tsx
export default function SpotDetailScreen() {
  const { id } =
    useLocalSearchParams<{ id: string }>();
  const spot = SEED_SPOTS.find((s) => s.id === id);
  if (!spot) { /* "No spot with id …" */ }
  /* <SpotDetail spot={spot} … /> */
}
```

```tsx
router.push({
  pathname: "/spots/[id]",
  params: { id: item.id },
});                          // push
router.back();               // pop
```

- The file tree **is** the set of destinations; `_layout.tsx` files are the hosts; `router` mutates the stack
- Every file-system route is a URL → deep links **for free**
- `(tabs)` is a route group: no URL segment
- Built on React Navigation (appendix); default for Expo SDK (Software Development Kit) 52+

Passing Data Between Screens

Rule: pass IDs (identifiers), not full objects

Platform	Forward	Back (result)
Android	`backStack.add(SpotDetail(spotId))` — the key carries the id	result bus (`ResultEventBus` , Nav3 1.2) or shared ViewModel
iOS	`NavigationLink(value: spot.id)` / `path.append(id)`	closure callback
Flutter	`context.push('/spots/${spot.id}')`	`await context.push<bool>('/review')`
Expo Router	`router.push({ pathname: "/spots/[id]", params: { id } })`	URL query param or shared state

Why IDs only?

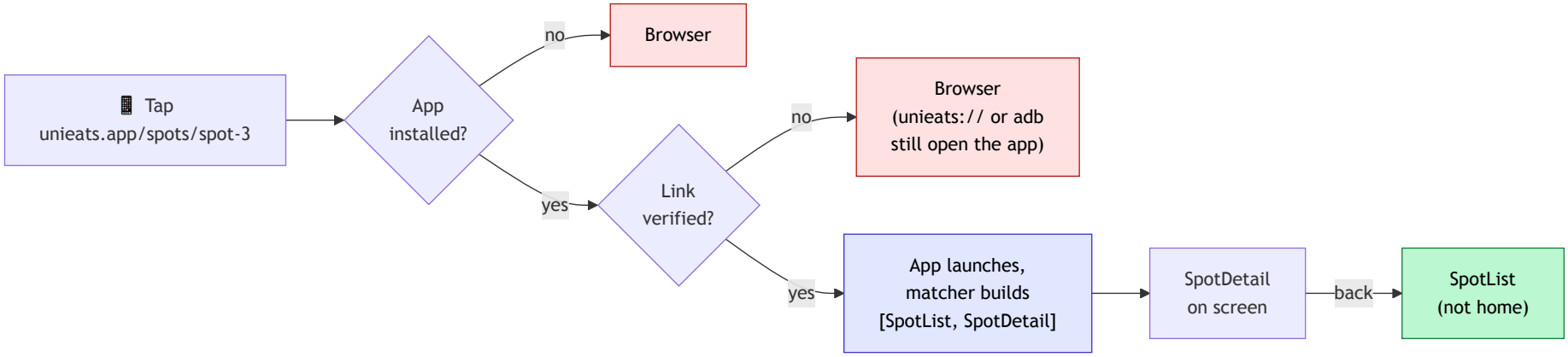
- Full objects break deep links (URLs are strings)
- The saved back stack is serialized on Android — a key with a whole `Spot` inside bloats it and goes stale
- Violates single source of truth (SSOT)

Prefer re-fetching from the data layer over passing results back through navigation. UniEats does this in all four stacks: the detail receives only the spot id and looks the spot up.

Part 4: Deep Links

Cold start straight into a screen — and back still works

Deep-Link Resolution Flow



Verified = Android `autoVerify` with `assetlinks.json`, or iOS Associated Domains with `apple-app-site-association` — unieats.app hosts neither, so the demo uses `unieats://spots/spot-3` or `adb … com.unieats.app`. Same flow in all four stacks.

Deep Links: Android & iOS

Android — `AndroidManifest.xml` + `UriDeepLinkMatcher`

```
<activity android:name=".MainActivity" android:launchMode="singleTop">
  <intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <data android:scheme="unieats" android:host="spots"/>
  </intent-filter>
  <intent-filter android:autoVerify="true">
    <!-- same action + categories -->
    <data android:scheme="https" android:host="unieats.app"
          android:pathPrefix="/spots/"/>
  </intent-filter>
</activity>
```

`ui/navigation/DeepLinks.kt`:

```
private val spotDetailMatchers = listOf(
    "unieats://spots", "https://unieats.app/spots").map { base ->
    UriDeepLinkMatcher(DeepLinkUri("$base/{spotId}"), serializer<SpotDetail>())
        .withBackStack { match -> listOf(SpotList, match.key) }
}

fun backStackFor(intent: Intent?): List<NavKey> {
    if (intent?.data == null) return listOf(SpotList)
    val best = spotDetailMatchers
        .mapNotNull { it.match(DeepLinkRequest(intent)) }.maxOrNull()
    return (best as? BackStackMatchResult<*, *>)?.backStack as? List<NavKey>
        ?: listOf(SpotList)
}
```

`MainActivity`: `startKeys = backStackFor(intent)`; `onNewIntent` → replace the list while running.

iOS — `.onOpenURL` sets the path

```
.onOpenURL { url in
    if let spotId = DeepLink.spotId(from: url) {
        path = [spotId]
    }
}
```

`DeepLink.spotId(from:)` accepts `unieats://spots/<id>` (host `spots` + one path segment) and `https://unieats.app/spots/<id>`, otherwise `nil`.

- `unieats` scheme: `CFBundleURLTypes` in `UniEats/Info.plist`
- `https`: `applinks:unieats.app` in `UniEats.entitlements` **plus** an `apple-app-site-association` file on unieats.app — not hosted, so demo with the scheme
- The list is the root of the `NavigationStack`, so back from the deep-linked detail lands on the list

Same idea on both: the URL becomes a **list of keys**, and the list starts with the parent.

Deep Links: Flutter & React Native

Flutter — the router's `redirect` folds the scheme host

```
String? deepLinkToPath(Uri uri) =>
    switch (uri.scheme) {
  'unieats' => '/${uri.host}${uri.path}',
  'http' || 'https' => uri.path,
  _ => null,
};
```

`unieats://spots/spot-3` parses as host `spots` + path `/spot-3` → `/spots/spot-3`.

`android/app/src/main/AndroidManifest.xml`:

```
<intent-filter>
  <action android:name=
      "android.intent.action.VIEW"/>
  <category … DEFAULT/> <category … BROWSABLE/>
  <data android:scheme="unieats"
        android:host="spots"/>
</intent-filter>
<!-- + https / unieats.app / pathPrefix /spots -->
<meta-data
  android:name="flutter_deeplinking_enabled"
  android:value="true"/>
```

iOS: `unieats` in `CFBundleURLTypes` + `FlutterDeepLinkingEnabled` in `ios/Runner/Info.plist`.

Expo Router — `app/(tabs)/spots/[id].tsx` is the route

```
{
  "expo": {
    "scheme": "unieats",
    "ios": {
      "associatedDomains": ["applinks:unieats.app"]
    },
    "android": {
      "intentFilters": [{
        "action": "VIEW",
        "autoVerify": true,
        "data": [{
          "scheme": "https",
          "host": "unieats.app",
          "pathPrefix": "/spots"
        }],
        "category": ["BROWSABLE", "DEFAULT"]
      }]
    }
  }
}
```

`unieats://spots/spot-3` → `/spots/spot-3` → `[id].tsx` with `id = "spot-3"`. `spots/_layout.tsx` sets `initialRouteName: "index"`, so the list is underneath.

Needs a dev build (`npx expo run:android`), not Expo Go.

Part 5: Scenes

Navigation 3 renders a Scene, not a screen

Scene and SceneStrategy

The question `NavDisplay` asks on every change of the list:

> *Given these entries, which of them do I show, and how?*

```
interface SceneStrategy<T : Any> {
    fun SceneStrategyScope<T>.calculateScene(
        entries: List<NavEntry<T>>
    ): Scene<T>?            // null = "not mine"
}

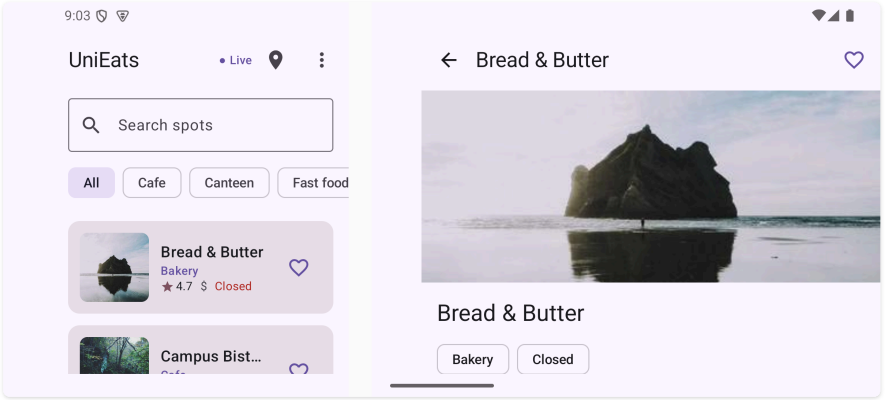
interface Scene<T : Any> {
    val key: Any                          // identity for animations
    val entries: List<NavEntry<T>>        // what this scene shows
    val previousEntries: List<NavEntry<T>>  // what "back" would show
    val content: @Composable () -> Unit // how it lays them out
}
```

How the pieces fit

```
backStack  [SpotList, SpotDetail(spot-3)]
    | entryProvider
    ▼
entries    [NavEntry(SpotList, metadata=listPane),
            NavEntry(SpotDetail, metadata=detailPane)]
    | sceneStrategies, first non-null wins
    ▼
scene      ListDetailScene(list | detail)   on a wide window
           SinglePaneScene(detail)          on a phone (fallback)
    |
    ▼
NavDisplay animates scene → scene, back = previousEntries
```

- Strategies are tried **in order**; `SinglePaneSceneStrategy` is the always-on fallback
- A strategy decides from the **entries' metadata** and the **window** — not from the keys
- Built in: single pane, dialog (`DialogSceneStrategy`), list-detail and supporting pane (Material adaptive)

ListDetailSceneStrategy in UniEats



Pixel 7 emulator in landscape (a later tag's build; the navigation code is identical from `l04-nav` on): the same `[SpotList, SpotDetail]` list, rendered as two panes.

`NavGraph.kt` @ `l04-nav` — three lines make it adaptive:

```
sceneStrategies = listOf(rememberListDetailSceneStrategy()),

entry<SpotList>(metadata = ListDetailSceneStrategy.listPane(
    detailPlaceholder = { SelectASpotPlaceholder() })) { /* … */ }

entry<SpotDetail>(metadata = ListDetailSceneStrategy.detailPane()) { /* … *
```

- Dependency: `androidx.compose.material3.adaptive:adaptive-navigation3` (1.3.0)
- Phone portrait: strategy returns `null` → single pane, exactly as before
- Wide window (tablet, foldable, landscape ≥ medium width): list **and** detail; the placeholder fills the detail pane until a spot is selected
- Back on the two-pane scene removes `SpotDetail` → list + placeholder
- Nothing else changed: screens, ViewModels, deep links are the same code

More Strategies: Dialog, and Your Own

Dialog — a key that renders above the previous entry

```
@Serializable data object ConfirmDelete : NavKey

NavDisplay(
    sceneStrategies = listOf(
        DialogSceneStrategy(),            // tried first
        rememberListDetailSceneStrategy()
    ),
    entryProvider = entryProvider {
        entry<ConfirmDelete>(
            metadata = DialogSceneStrategy.dialog()
        ) {
            AlertDialog(/* … onConfirm → backStack.removeLastOrNull() */)
        }
        /* … */
    }
)
backStack.add(ConfirmDelete)   // open; back or dismiss pops it
```

A modal becomes **an entry on the list** — it survives rotation and process death because the key does.

Custom — two panes when wide, the rule is yours

```
class TwoPaneStrategy<T : Any>(
    private val isWide: Boolean
) : SceneStrategy<T> {
    override fun SceneStrategyScope<T>.calculateScene(
        entries: List<NavEntry<T>>
    ): Scene<T>? {
        if (!isWide || entries.size < 2) return null
        val (left, right) = entries.takeLast(2)
        return TwoPaneScene(
            key = left.contentKey,
            entries = listOf(left, right),
            previousEntries = entries.dropLast(1)
        )          // content = Row { left.Content(); right.Content() }
    }
}
```

- Return `null` when the rule does not apply → the next strategy gets a chance
- `previousEntries` tells `NavDisplay` what back means for this scene
- Scenes must have stable `equals` / `hashCode` — use a data class

Part 6: Beyond One Stack

Tabs · Modals · Drawers · Adaptive · Restoration — concepts for your own app

Tabs: One Stack per Tab — concept (UniEats has one stack)

Android (Navigation 3) — one `NavBackStack` per tab

```
val stacks = remember { tabs.associateWith { tab ->
    rememberNavBackStack(tab.rootKey) } }   // one list each
var selected by rememberSaveable { mutableStateOf(0) }

Scaffold(bottomBar = {
    NavigationBar {
        tabs.forEachIndexed { i, tab ->
            NavigationBarItem(selected = selected == i,
                onClick = { selected = i },      // switch lists
                icon = { Icon(tab.icon, null) }, label = { Text(tab.label) })
        }
    }
}) { padding ->
    val backStack = stacks.getValue(tabs[selected])
    NavDisplay(backStack = backStack,
        onBack = { backStack.removeLastOrNull() },
        modifier = Modifier.padding(padding), entryProvider = /* … */)
}
```

The wrong version — `backStack.add(tab.key)` on a single list — grows `[Feed, Search, Profile, Feed, …]` and back walks through old tabs. In Navigation 2 the same bug was "missing the four NavOptions".

iOS — each tab owns a `NavigationStack`

```
TabView(selection: $selectedTab) {
    NavigationStack { FeedView() }
        .tabItem { Label("Feed",
            systemImage: "fork.knife") }.tag(0)
    NavigationStack { SearchView() }
        .tabItem { Label("Search",
            systemImage: "magnifyingglass") }.tag(1)
}
```

- **Flutter:** `ShellRoute` renders the bar once and swaps only the body; `context.go` switches tabs
- **Expo Router:** the `(tabs)/_layout.tsx` you saw — one more file = one more tab
- Same rule everywhere: **independent back history per tab**, the active tab's stack is the one back pops

Modals Across Platforms

Android Compose — state-driven boolean flag

```
if (showDeleteDialog) {
    AlertDialog(
        onDismissRequest = {
            showDeleteDialog = false
        },
        title = { Text("Delete review?") },
        confirmButton = {
            TextButton(onClick = {
                viewModel.deleteReview()
                showDeleteDialog = false
            }) { Text("Delete") }
        },
        dismissButton = {
            TextButton(onClick = {
                showDeleteDialog = false
            }) { Text("Cancel") }
        }
    )
}
```

Or a key + `DialogSceneStrategy` (Part 5) when the dialog must survive process death or sit in the back stack.

iOS SwiftUI — state-driven sheet

```
.sheet(isPresented: $showAddReview) {
    AddReviewView(placeId: placeId)
        .presentationDetents([.medium, .large])
}
.confirmationDialog(
    "Delete this review?",
    isPresented: $showDeleteConfirm) {
    Button("Delete", role: .destructive) {
        viewModel.deleteReview()
    }
}
```

Flutter — imperative call

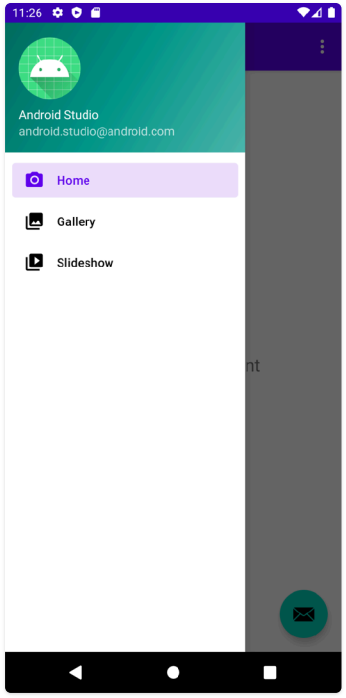
```
showModalBottomSheet(
    context: context,
    builder: (_) => FilterPanel(
        onApply: () =>
            Navigator.pop(context)));
```

React Native — same imperative pattern

```
// @gorhom/bottom-sheet for production
bottomSheetRef.current?.expand();
```

State-driven = easy to restore (restore the bool). Imperative = must re-trigger in `useEffect` / `initState`.

Drawer Navigation — concept (not in UniEats)



A **slide-out menu** for secondary destinations: settings, admin, less-used sections.

- Opened by the hamburger icon (top-left) or a swipe from the left edge
- Same shape as tabs underneath: **N parallel stacks**, one per entry
- **Close the drawer after navigating** — the classic beginner mistake

Platform	API
Android	`ModalNavigationDrawer` + `NavigationDrawerItem`
iOS	sidebar of `NavigationSplitView`
Flutter	`Drawer` in `Scaffold.drawer`
React Native	`createDrawerNavigator()`

Code for all four → appendix. Runnable: `lectures/3/NavigationDrawer`, `lectures/3/navigation_drawer`.

Adaptive Navigation — concept (UniEats is phone-first, except the two-pane scene)

Android — `WindowSizeClass` drives the chrome

```
@Composable
fun AdaptiveApp() {
    val wsc = currentWindowAdaptiveInfo()
        .windowSizeClass
    when (wsc.windowWidthSizeClass) {
        WindowWidthSizeClass.COMPACT ->
            AppWithBottomNav()
        WindowWidthSizeClass.MEDIUM ->
            AppWithNavRail()
        else ->
            AppWithDrawer()
    }
}
```

- `COMPACT` → bottom nav (phone portrait)
- `MEDIUM` → nav rail (foldable inner, landscape)
- `EXPANDED` → drawer; content panes come from `ListDetailSceneStrategy` (Part 5)

Never hardcode by screen size in inches — foldables add a third state.

iOS — `NavigationSplitView` adapts for free

```
NavigationSplitView {
    SpotListView(selection: $selectedSpotId)
} detail: {
    if let id = selectedSpotId {
        SpotDetailView(spotId: id).id(id)
    } else {
        ContentUnavailableView(
            "No Spot Selected",
            systemImage: "fork.knife")
    }
}
```

iPhone → stack · iPad → sidebar + detail · Mac → full sidebar. Zero extra code per form factor.

Flutter	React Native
`flutter_adaptive_scaffold` handles breakpoints	`useWindowDimensions()` — manual one-pane vs two-pane

Same back stack, different chrome. Code → appendix.

State Restoration

Android kills background processes. iOS suspends them. This drives different restoration strategies.

Platform	UI state	ViewModel state	Nav back stack
Android	`rememberSaveable` (per entry via `rememberSaveableStateHolderNavEntryDecorator`)	`SavedStateHandle.saveable { }`	`rememberNavBackStack` saves the key list — that is why keys are `@Serializable`
iOS	`@State` (survives suspension)	`@Observable` (survives suspension)	`@SceneStorage` for `NavigationPath`
Flutter	`RestorationMixin` + `RestorableX`	Provider/Riverpod with persistence	go_router `initialLocation` from storage
React Native	`app.json: statePersistence: true`	Zustand + AsyncStorage persist	Expo Router URL or manual `NavigationState`

UniEats Android already passes: the key list is restored by `rememberNavBackStack`, and the detail ViewModel is rebuilt from the restored key's `spotId`.

Part 7: Spot the AI Bug 

Spot the AI Bug 🐞

Study this LLM (Large Language Model)-generated Navigation 3 code for 60 seconds. Why does tapping a spot do nothing?

Spot the AI Bug 🐞 — The Snippet

```
@Serializable data object SpotList : NavKey
@Serializable data class SpotDetail(val spotId: String) : NavKey

@Composable
fun UniEatsApp(listViewModel: SpotListViewModel = hiltViewModel()) {
    val backStack = remember { mutableListOf<NavKey>(SpotList) }

    NavDisplay(
        backStack = backStack,
        onBack = { backStack.removeLastOrNull() },
        entryProvider = entryProvider {
            entry<SpotList> {
                SpotListScreen(
                    onSpotClick = { spotId -> backStack.add(SpotDetail(spotId)) },
                    viewModel = listViewModel
                )
            }
            entry<SpotDetail> { key ->
                SpotDetailScreen(spotId = key.spotId, onBack = { backStack.removeLastOrNull() })
            }
        }
    )
}
```

Spot the AI Bug 🐞 — Flaw Revealed

Bug: `remember { mutableListOf(...) }` is a plain Kotlin list — **not snapshot state**. `backStack.add(...)` changes it, but nothing tells Compose, so `NavDisplay` never recomposes: the tap "does nothing". And a plain `remember` is lost on process death.

The fix — the list UniEats uses:

```
val backStack = rememberNavBackStack(SpotList)   // snapshot-state + saved
```

(`mutableStateListOf(SpotList)` would redraw but not survive process death.)

Review checklist for AI-generated Navigation 3 code:

1. Is the back stack `rememberNavBackStack` (or at least snapshot state)?
2. Are keys `@Serializable` and `NavKey`, carrying **ids** not objects?
3. Does `onBack` mutate the **same** list `NavDisplay` shows?
4. Tabs: one list per tab, or does every tab tap `add()` to one list?

Concept Check: Choosing the Right Navigation

Given these requirements, which navigation shape fits?

Scenario	Shape	Why
Login → Home (never go back to login)	Replace the stack: `backStack.clear(); add(Home)`	Login must not stay underneath
Feed / Search / Profile sections	Tabs: one back stack per tab	Peer navigation, state preserved per tab
"Are you sure?" before deleting	Modal / dialog (boolean, or a key + `DialogSceneStrategy`)	Ephemeral, blocks interaction
Campaign URL (Uniform Resource Locator) in a push notification	Deep link → `[SpotList, SpotDetail]`	Opens specific content from cold start
Review detail on tablet vs phone	Same list + `ListDetailSceneStrategy` / `NavigationSplitView`	Same data, different scene

These are the five most common navigation design decisions in a real app.

Navigation: Pre-Ship Checklist

1. Back from every screen → correct parent?
2. Tabs A → B → C, then press back → exits task cleanly (not traverse tabs)?
3. Cold deep link → list starts with the parent underneath?
4. Rotate on every screen → state preserved? On a wide window, does the two-pane scene still make sense?
5. Background app 30 min on low-RAM (Random Access Memory) device → resume lands on same screen?
6. Log out → back button cannot reach a protected screen?

```
adb shell am start -W -a android.intent.action.VIEW \
    -d "unieats://spots/spot-3" com.unieats.app
```

Simulate process death (step 5): Android Studio → App Inspection → stop the process while the app is in the background → reopen from recents.

🖥️ Demo — UniEats Navigation

> Tap a spot → detail (only the **id** travels) → back. Then, with the app closed, fire a deep link from the terminal: Pizza Stop opens directly, and **back lands on the list**, not the home screen. Rotate the Android emulator: list and detail side by side. Search, category filter and favourites from L03 still work. Tabs and drawers are concepts today — UniEats has none.

Repo: `github.com/dancojocar/ma` @ tag `l04-nav` · **What this lecture adds:** `git diff l03-state l04-nav -- unieats/<stack>/`

Stack	New / changed at `l04-nav`
Android	`ui/navigation/NavGraph.kt` (new: keys, `rememberNavBackStack`, `NavDisplay`, `ListDetailSceneStrategy`) · `ui/navigation/DeepLinks.kt` (new: `UriDeepLinkMatcher` + `backStackFor(intent)`) · `ui/spotdetail/SpotDetailViewModel.kt` (Hilt assisted, takes `spotId`) · `SpotDetailScreen.kt` · `MainActivity.kt` (`startKeys`, `onNewIntent`) · `AndroidManifest.xml` (2 intent-filters, `singleTop`) · `navigation3-runtime/ui`, `lifecycle-viewmodel-navigation3`, `adaptive-navigation3` in `build.gradle.kts`
iOS	`SpotListView.swift` (`NavigationStack(path:)`, `.onOpenURL`) · `SpotDetailView.swift` (`spotId:` + favourite callbacks) · `Navigation/DeepLink.swift` (new) · `Info.plist` (new, scheme) · `UniEats.entitlements` (new)
Flutter	`lib/app_router.dart` (new) · `main.dart` (`MaterialApp.router`) · `providers.dart` (`spotByIdProvider`) · list + detail screens · `go_router` · Android manifest + iOS `Info.plist`
React Native	`app/(tabs)/spots/[id].tsx` (new) · `spots/index.tsx` (`router.push`) · `spots/_layout.tsx` (`initialRouteName`) · `app.json` (https intent filter, associated domain)

```
./unieats/android/gradlew -p unieats/android :app:installDebug
adb shell am start -W -a android.intent.action.VIEW \
    -d "unieats://spots/spot-3" com.unieats.app   # or https://unieats.app/spots/spot-3
cd unieats/flutter && flutter run                 # then adb … com.unieats.flutter
cd unieats/rn && npm ci && npx expo run:android   # then adb … com.unieats.rn
xcrun simctl openurl booted "unieats://spots/spot-3"   # iOS, after Cmd+R in Xcode
```

Interview Corner 🎤

Interview Corner 🎤

Q1. *"In Navigation 3, what is the back stack and what goes wrong with bottom-nav tabs?"*

> A snapshot-state list of serializable keys you own (`rememberNavBackStack`); `NavDisplay` renders it. One shared list with `add()` per tab tap grows forever and back walks through old tabs — give each tab its own list and switch which one `NavDisplay` shows.

Q2. *"Why pass an ID instead of a full object through navigation arguments?"*

> The key list is serialized and saved — objects bloat it and go stale; deep links can only carry strings; single source of truth (SSOT) belongs in the data layer.

Q3. *"Your app opens via a push-notification deep link to SpotDetail. The user presses back. Where do they go?"*

> The deep-link matcher (`UriDeepLinkMatcher.withBackStack`) started the list as `[SpotList, SpotDetail]` . Back removes SpotDetail and lands on SpotList, not the home screen.

Q4. *"What is a SceneStrategy, and how would you get a list-detail layout on tablets?"*

> `NavDisplay` asks each strategy in order to turn the entries into a Scene; `SinglePaneSceneStrategy` is the fallback. Tag the entries with `listPane()` / `detailPane()` metadata and add `rememberListDetailSceneStrategy()` — the back stack does not change. On iOS, `NavigationSplitView` does the same with zero extra code.

Q5. *"What is Android process death and how does it affect navigation code?"*

> The OS kills a background app to free RAM (Random Access Memory). `rememberNavBackStack` restores the key list (keys are `@Serializable NavKey` s), the entry decorators restore per-entry `rememberSaveable` state and recreate ViewModels from the restored keys.

Key Takeaways

The 3 things you must remember:

1. **The back stack is a data structure you own** — on Android (Navigation 3) and iOS it is literally a list; Android's system back is not optional, and `onBack` must pop that list
2. **Every tab owns its own stack** — one `NavBackStack` per tab on Android; a `NavigationStack` per tab on iOS
3. **Pass IDs, not objects** — the key list is saved and serialized; full objects break deep links and violate SSOT (Single Source Of Truth)

Also:

- **Name the shape first** — stack / tabs / drawer / modal / deep link — then the API follows on any platform
- **A deep link must start the list with the parent underneath** — otherwise back leaves the app
- **Layout is a scene, not a screen** — the same list renders as one pane on a phone and two on a tablet

What's Next — L05: Networking & REST

- HTTP (Hypertext Transfer Protocol) on mobile: timeouts, retries, cancellation
- JSON (JavaScript Object Notation): `kotlinx.serialization` ↔ `Codable` ↔ hand-written `fromJson` ↔ `Zod`
- Clients: `Ktor` ↔ `URLSession` ↔ `dio` ↔ `fetch` + `TanStack Query`
- UniEats REST (Representational State Transfer) server — connect to a real API (Application Programming Interface)
- LLM pitfall of the week: hallucinated endpoints, missing error paths

Practice this week: in your own project, list → detail by **id**, one working deep link, and back that never skips the list. Bonus: a bottom bar with one back stack per tab, or the list-detail scene on a tablet emulator.

Questions?

Appendix — Self-Study (not presented live)

Navigation 2 (legacy): NavHost + NavController

What most existing Android codebases still use (`androidx.navigation:navigation-compose`):

```
@Serializable object SpotList
@Serializable data class SpotDetail(val spotId: String)

val navController = rememberNavController()
NavHost(navController, startDestination = SpotList) {
    composable<SpotList> {
        SpotListScreen(onSpotClick = { id ->
            navController.navigate(SpotDetail(id)) })
    }
    composable<SpotDetail>(deepLinks = listOf(
        navDeepLink<SpotDetail>(basePath = "unieats://spots"))
    ) { entry ->
        val id = entry.toRoute<SpotDetail>().spotId
        SpotDetailScreen(onBack = { navController.popBackStack() })
    }
}
// ViewModel: savedStateHandle.toRoute<SpotDetail>().spotId
```

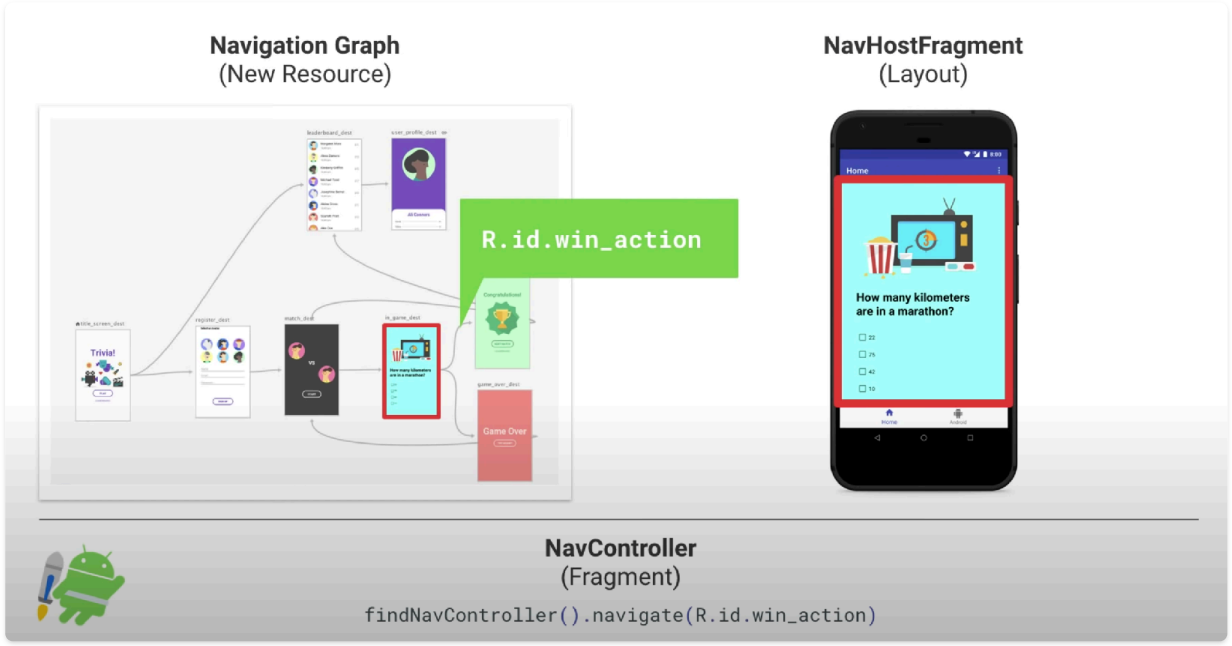
Navigation 2	Navigation 3
`NavHost` + `NavController` own the stack	you own a `NavBackStack` list
`navigate(route)` / `popBackStack()`	`backStack.add(key)` / `removeLastOrNull()`
`composable<T> { }` in a graph	`entry<T> { key -> }` in an `entryProvider`
`navDeepLink<T>()` on the destination	`UriDeepLinkMatcher` → list of keys
one destination visible	a **Scene** of one or more entries
tabs: `navigate(route) { popUpTo(start) { saveState = true }; launchSingleTop = true; restoreState = true }`	one list per tab

The four NavOptions above were the classic LLM bug in Navigation 2 — missing any one grew the stack or lost tab state. Google's migration guide: *developer.android.com/guide/navigation/navigation-3/migrate*.

Android Navigation Concepts — Reference

Concept	Purpose	Type
Host	A UI element that contains the current navigation destination. That is, when a user navigates through an app, the app essentially swaps destinations in and out of the navigation host.	• Compose: `NavHost` • Fragments: `NavHostFragment`
Graph	A data structure that defines all the navigation destinations within the app and how they connect together.	`NavGraph`
Controller	The central coordinator for managing navigation between destinations. The controller offers methods for navigating between destinations, handling deep links, managing the back stack, and more.	`NavController`
Destination	A node in the navigation graph. When the user navigates to this node, the host displays its content.	`NavDestination` Typically created when constructing the navigation graph.
Route	Uniquely identifies a destination and any data required by it. You can navigate using routes. Routes take you to destinations.	Any serializable data type.

Graph · Host · Controller (Navigation 2 picture)



NavGraph = the map · **NavHost** = the container showing one destination · **NavController** = the object you call `navigate()` on. In Navigation 3 the map and the controller are both just your list.

React Native: React Navigation (bare)

Explicit type-safe params

```
type RootStackParams = {
  SpotList: undefined;
  SpotDetail: { spotId: string };
};

const Stack =
    createNativeStackNavigator<RootStackParams>();

function SpotsStack() {
  return (
    <Stack.Navigator>
      <Stack.Screen name="SpotList"
          component={SpotListScreen} />
      <Stack.Screen name="SpotDetail"
          component={SpotDetailScreen} />
    </Stack.Navigator>
  );
}

type Props = NativeStackScreenProps<
    RootStackParams, 'SpotDetail'>;

function SpotDetailScreen({ route }: Props) {
  const { spotId } = route.params;
}
```

Tabs — each tab wraps its own stack

```
const Tab = createBottomTabNavigator();

function AppTabs() {
  return (
    <Tab.Navigator
      screenOptions={{ headerShown: false }}>
      <Tab.Screen
        name="Spots"
        component={SpotsStack}
        options={{ tabBarIcon: ... }} />
      <Tab.Screen
        name="Profile"
        component={ProfileStack}
        options={{ tabBarIcon: ... }} />
    </Tab.Navigator>
  );
}
```

```
navigation.navigate('SpotDetail', { spotId: id });
navigation.goBack();
navigation.replace('Home');
```

Expo Router (used by UniEats) is built on top of this library, so everything here still applies underneath.

Tabs: Flutter & Expo Router — code

Flutter — ShellRoute with persistent bar

```
ShellRoute(
  builder: (context, state, child) => Scaffold(
    body: child,
    bottomNavigationBar: NavigationBar(
      selectedIndex: _selectedIndex(state),
      onDestinationSelected: (i) {
        switch (i) {
          case 0: context.go('/spots'); break;
          case 1: context.go('/search'); break;
          case 2: context.go('/profile'); break;
        }
      },
      destinations: const [
        NavigationDestination(
            icon: Icon(Icons.home),
            label: 'Feed'),
        NavigationDestination(
            icon: Icon(Icons.search),
            label: 'Search'),
        NavigationDestination(
            icon: Icon(Icons.person),
            label: 'Profile'),
      ],
    ),
  ),
  routes: [ /* GoRoutes here */ ],
)
```

Expo Router — `app/(tabs)/_layout.tsx` with two tabs

```
import { Tabs } from 'expo-router';

export default function TabLayout() {
  return (
    <Tabs screenOptions={{
        tabBarActiveTintColor: '#14b8a6',
    }}>
      <Tabs.Screen
        name="spots"
        options={{
          title: 'Spots',
          tabBarIcon: ({ color }) =>
            <Ionicons name="home" color={color} />,
        }} />
      <Tabs.Screen
        name="search"
        options={{
          title: 'Search',
          tabBarIcon: ({ color }) =>
            <Ionicons name="search" color={color} />,
        }} />
    </Tabs>
  );
}
```

`context.go()` vs `context.push()` in Flutter: go replaces the stack, push adds to it — same as `backStack.clear(); add()` vs `backStack.add()` on Android.

Drawer: Android & iOS — code

Android — ModalNavigationDrawer in Compose

```
val drawerState = rememberDrawerState(
    DrawerValue.Closed)
val scope = rememberCoroutineScope()

ModalNavigationDrawer(
    drawerState = drawerState,
    drawerContent = {
        ModalDrawerSheet {
            NavigationDrawerItem(
                label = { Text("Feed") },
                selected = selectedSection == Section.Feed,
                onClick = {
                    scope.launch {
                        drawerState.close()
                    }
                    selectedSection = Section.Feed  // switches which
                }                                   // NavBackStack is shown
            )
        }
    }
) {
    Scaffold { /* NavDisplay(backStack = stacks[selectedSection]) */ }
}
```

iOS SwiftUI — sidebar in NavigationSplitView

```
NavigationSplitView {
    List(AppSection.allCases,
         selection: $selectedSection) { section in
        Label(section.title,
              systemImage: section.icon)
    }
    .navigationTitle("UniEats")
} detail: {
    switch selectedSection {
    case .feed:   FeedView()
    case .search: SearchView()
    case .profile: ProfileView()
    }
}
```

A drawer item does what a tab does: it selects which back stack is on screen. The drawer itself is chrome, not navigation data.

Drawer: Flutter & React Native — code

Flutter — Drawer in Scaffold

```
Scaffold(
  appBar: AppBar(title: const Text('UniEats')),
  drawer: Drawer(
    child: ListView(
      children: [
        const DrawerHeader(
          decoration: BoxDecoration(
              color: Colors.teal),
          child: Text('UniEats',
              style: TextStyle(
                  color: Colors.white,
                  fontSize: 24)),
        ),
        ListTile(
          leading: const Icon(Icons.home),
          title: const Text('Feed'),
          onTap: () {
            context.go('/spots');
            Navigator.pop(context);
          },
        ),
        ListTile(
          leading: const Icon(Icons.search),
          title: const Text('Search'),
          onTap: () {
            context.go('/search');
            Navigator.pop(context);
          },
        ),
      ],
    ),
  ),
  body: child,
)
```

React Native — createDrawerNavigator

```
import { createDrawerNavigator }
    from '@react-navigation/drawer';

const Drawer = createDrawerNavigator();

function AppDrawer() {
  return (
    <Drawer.Navigator
      screenOptions={{
        drawerType: 'front',
        drawerStyle: { width: 280 },
      }}>
      <Drawer.Screen
        name="Feed"
        component={FeedStack}
        options={{
          drawerIcon: ({ color }) => (
            <Icon name="home" color={color} />
          ),
        }} />
      <Drawer.Screen
        name="Settings"
        component={SettingsScreen} />
    </Drawer.Navigator>
  );
}

// Open programmatically:
navigation.openDrawer();
navigation.closeDrawer();
```

Both platforms: swipe from left edge to open.

Adaptive: Flutter & React Native — code

Flutter — flutter_adaptive_scaffold

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

AdaptiveScaffold(
  destinations: const [
    NavigationDestination(
        icon: Icon(Icons.home), label: 'Feed'),
    NavigationDestination(
        icon: Icon(Icons.search), label: 'Search'),
    NavigationDestination(
        icon: Icon(Icons.person), label: 'Profile'),
  ],
  body: (_) => const SpotListScreen(),
  secondaryBody: (_) =>
      selectedSpotId != null
          ? SpotDetailScreen(spotId: selectedSpotId!)
          : const SizedBox(),
)
// Automatically shows bottom nav on small screens,
// nav rail on medium, side nav on large.
```

React Native — useWindowDimensions hook

```jsx
import { useWindowDimensions, View } from 'react-native';

function AdaptiveLayout() {
  const { width } = useWindowDimensions();
  const isTablet = width >= 768;

  if (isTablet) {
    return (
      <View style={{ flexDirection: 'row', flex: 1 }}>
        <SpotSidebar
            style={{ width: 320 }}
            onSelect={setSelectedSpotId} />
        <View style={{ flex: 1 }}>
          {selectedSpotId
            ? <SpotDetailScreen spotId={selectedSpotId} />
            : <EmptyDetail />}
        </View>
      </View>
    );
  }

  return <BottomTabNavigator />;
}
```

React Native has no built-in adaptive scaffold — dimension checks are manual.

Adaptive Master-Detail — API reference

Platform	API	How it adapts
Android (Navigation 3)	`ListDetailSceneStrategy` + `listPane()` / `detailPane()` metadata (`adaptive-navigation3`)	Strategy reads the window size class; one or two panes with animated transition; the back stack is unchanged
Android (Navigation 2 era)	`ListDetailPaneScaffold` (`material3.adaptive`)	Same panes, driven by its own `scaffoldNavigator` instead of the back stack
iOS	`NavigationSplitView`	Collapses to stack on iPhone, splits on iPad, sidebar on Mac — zero extra code
Flutter	`flutter_adaptive_scaffold` package	`AdaptiveScaffold` handles breakpoints
React Native	`useWindowDimensions()` hook	Manual: show list or detail based on `width`

```
// Navigation 3 — UniEats l04-nav
sceneStrategies = listOf(rememberListDetailSceneStrategy())
entry<SpotList>(metadata = ListDetailSceneStrategy.listPane()) { /* … */ }
entry<SpotDetail>(metadata = ListDetailSceneStrategy.detailPane()) { /* … */ }
```

Common Navigation Pitfalls (1/2) — Android, Navigation 3

Pitfall	Symptom	Fix
Back stack is `remember { mutableListOf() }`	Taps "do nothing"; stack lost on process death	`rememberNavBackStack(...)`
One shared list, `add()` on every tab tap	Back traverses old tabs instead of exiting	One `NavBackStack` per tab; switch which one `NavDisplay` shows
Key carries the whole object	Stale detail, bloated saved state, no deep link	Keys carry **ids**; look the object up
Missing `rememberViewModelStoreNavEntryDecorator`	One ViewModel shared by every detail entry	Add both default decorators to `entryDecorators`
Deep link produces `[SpotDetail]` only	Back after a deep link exits the app	`UriDeepLinkMatcher(...).withBackStack { listOf(SpotList, it.key) }`
Navigation 2: missing the 4 NavOptions on a tab switch	Back traverses tabs	`popUpTo(start) { saveState = true }` + `launchSingleTop` + `restoreState`

LLMs generate the happy-path code correctly; pitfalls appear at the edges.

Common Navigation Pitfalls (2/2) — iOS, Flutter, React Native

Pitfall	Symptom	Fix
Sharing one `NavigationStack` across iOS tabs	Back from detail goes to wrong tab	Give each tab its own `NavigationStack`
`context.go()` instead of `context.push()` in Flutter detail	Back stack empty after navigation	Use `push` for drill-down, `go` for tab switches
`WillPopScope` in Flutter	Deprecated warning, may break on Flutter 4	Replace with `PopScope`
Replacing the whole `path` on every push (iOS)	Back skips screens	`path.append(id)` to push; assign the array only for deep links
Passing objects in Expo Router params	`[object Object]` in the URL, no deep link	Params are strings: pass the id, look the object up
Testing deep links in Expo Go	Scheme never fires	Dev build: `npx expo run:android`