12 · Navigation

Flutter Practice Notes

Contents

12 · Navigation

Introduction

Navigation is how users move between screens. Flutter models screens as a **stack of routes** managed by a `Navigator`, with two paradigms: **imperative** (`push` / `pop`) and **declarative** (Navigator 2.0 / Router API). This module covers both, named routes, nested navigators/tabs, and transitions.

Why this module exists

Navigation touches every app and is a common source of bugs (lost arguments, back-button issues, deep-link mismatches, nested-navigator confusion). Understanding the stack model and both paradigms lets you build correct, testable navigation — and choose the right approach.

Module map

#	File	Topic	Level
1	01_navigator_stack.md	The route stack model	🟢
2	02_imperative_navigation.md	`push` / `pop`, `MaterialPageRoute`, args & results	🟢
3	03_named_routes.md	Named routes, `onGenerateRoute`, settings	🔵
4	04_declarative_navigation.md	Navigator 2.0 / Router / `Page` API	🔴
5	05_nested_navigators_and_tabs.md	Nested navigators, tabs, `IndexedStack`	🔵
6	06_route_transitions.md	`PageRouteBuilder`, custom transitions, `Hero`	🔵

> **Cross-references:** Deep links, route guards, and `go_router` are in Module 13 Routing. `BuildContext` / `Navigator.of`: 06 · build_context. State across navigation: Module 11. App shell/ `MaterialApp`: 06 · app_entry_point.

Prerequisites

06 Flutter Fundamentals (context, app entry) and 07 Widgets.

What you'll be able to do after this module

- Explain the navigator stack and push/pop semantics.
- Pass arguments and return results between screens (type-safely).
- Use named routes + `onGenerateRoute`.
- Understand Navigator 2.0 / Router and when it's needed.

- Build nested navigators, tabs, and custom transitions/Hero animations.

Capstones

Tier	Build
Beginner	Two-screen push/pop with a returned result.
Intermediate	Named-route app with argument passing + `onGenerateRoute`.
Advanced	Bottom-nav shell with per-tab nested navigators preserving state.
Enterprise	A declarative router mapping app state → pages (bridge to Module 13).

Summary

Screens are a stack of routes. Master imperative push/pop + args/results first, then named routes, then the declarative Router for URL-driven/complex navigation. Nested navigators and transitions round out real-world needs.

The Navigator Stack

> A `Navigator` manages a **stack of routes** (screens): pushing adds one on top, popping removes it — exactly like a stack data structure, and it's what the device back button pops.

Introduction

Flutter navigation is a stack: the visible screen is the top route; navigating deeper **pushes**, going back **pops**. The `Navigator` (provided by `MaterialApp`) owns this stack. Understanding it makes push/pop, back-button behavior, and route results intuitive.

Why this concept exists

Users expect history/back semantics. A stack naturally models "screens I've visited, most recent on top, back returns to the previous." Flutter exposes this via `Navigator`, so navigation state is explicit and predictable.

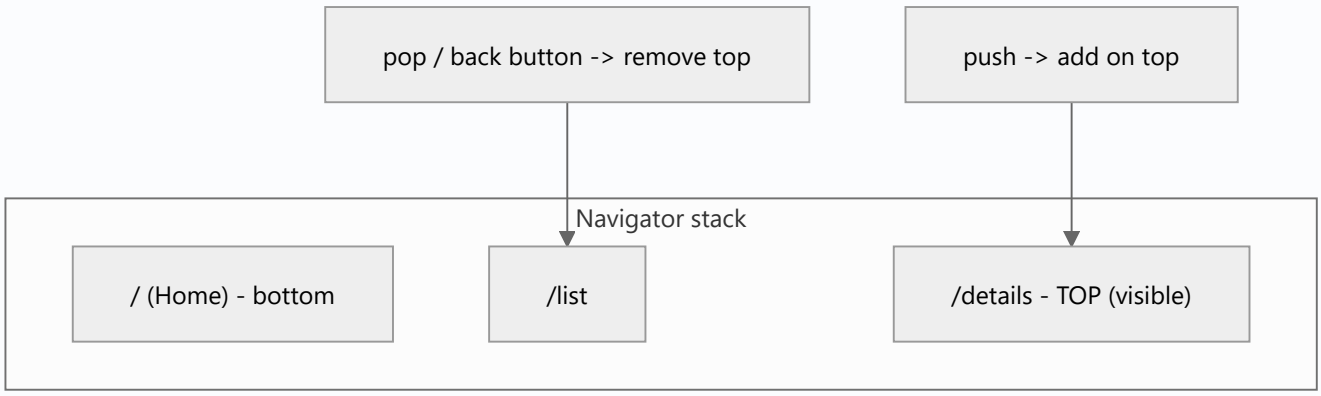
Real-world analogy

A **stack of plates**: you add a plate on top (push a screen), and you can only remove the top plate (pop back). The plate you see is the top of the stack; removing it reveals the one below (the previous screen).

Problem Statement

You tap into a details screen, then back out; the back button returns to the list. Why, and what is the `Navigator` actually doing? You'll model it as a stack and reason about push/pop.

Internal Working



- The `Navigator` holds an ordered list of `Route` objects; the last is on top (visible).
- `push` adds a route; `pop` removes the top (returning an optional result to the pusher).
- The **device/system back** triggers a pop (or `WillPopScope` / `PopScope` interception).
- `Navigator.of(context)` (or `Navigator.push(context, ...)`) finds the nearest `Navigator` up the tree (06 · build_context).
- The root `Navigator` is created by `MaterialApp`; you can nest more (05_nested_navigators_and_tabs.md).

Memory Representation

Each route holds its screen's element/state subtree; routes below the top are kept alive (retained) so back returns to their preserved state. Popped routes are disposed (08 · dispose_and_leaks).

Compiler Behavior

Not applicable.

Runtime Behavior

Pushing animates the new route in and pauses (but retains) the one below; popping animates it out and resumes the previous. An empty stack (popping the last route) can exit the app.

Flutter Engine Behavior / Dart VM Behavior

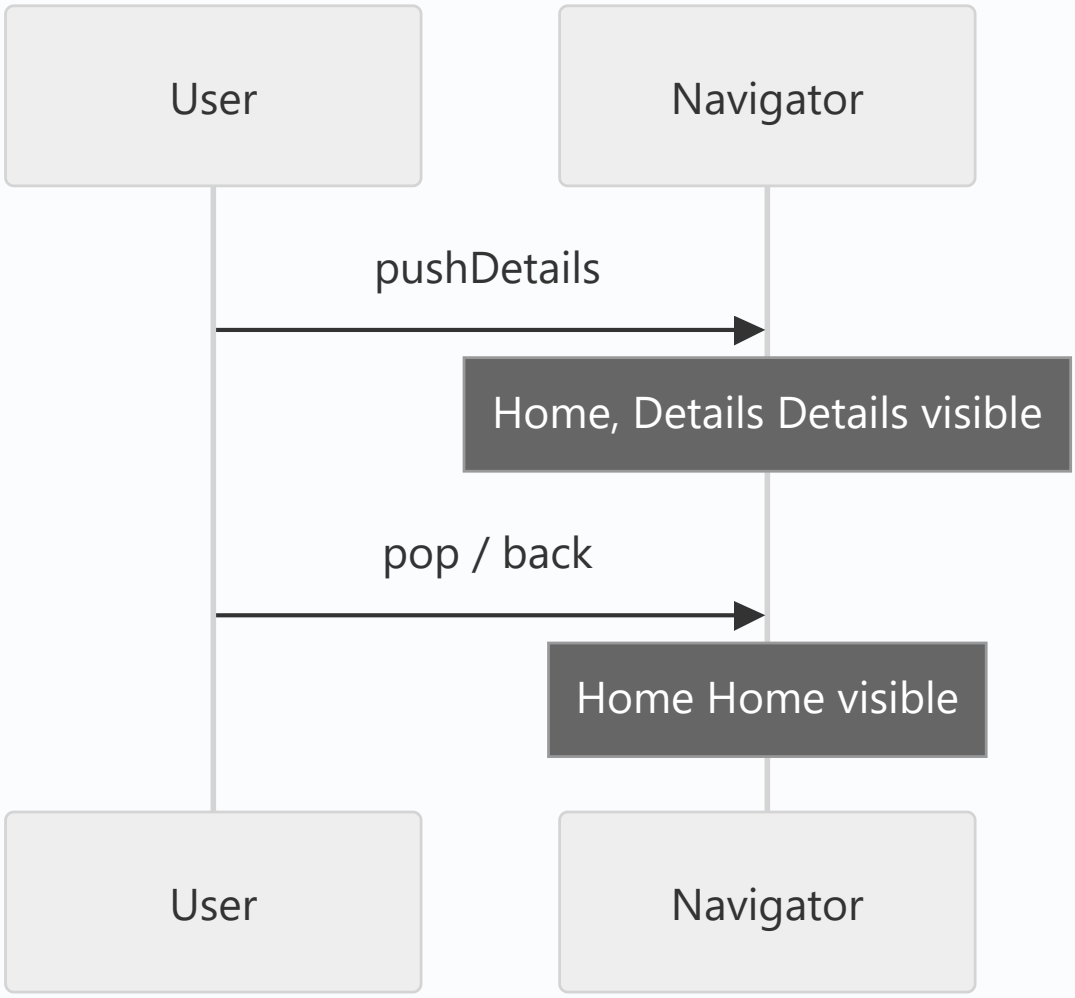
Not applicable beyond rendering the transitions.

Examples

```
import 'package:flutter/material.dart';


void main() => runApp(const MaterialApp(home: HomeScreen()));


class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Home (stack bottom)')),
      body: Center(
        child: ElevatedButton(
```

```
          onPressed: () {
            // push -> adds DetailsScreen on TOP of the stack
            Navigator.of(context).push(
              MaterialPageRoute(builder: (_) => const DetailsScreen()),
            );
          },
          child: const Text('Go to details'),
        ),
      ),
    );
  }
}

class DetailsScreen extends StatelessWidget {
  const DetailsScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Details (top)')), // back button auto-added
      body: Center(
        child: ElevatedButton(
          onPressed: () => Navigator.of(context).pop(), // remove top -> back to Home
          child: const Text('Back'),
        ),
      ),
    );
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`Navigator.of(context)` with a context above the Navigator	Not found	Use a descendant context / `Builder` (06)
Expecting popped screens to retain state	They're disposed	Keep state above the route or in a store (Module 11)
Popping the last route unexpectedly	Exits the app	Guard with `PopScope` /check `canPop`
Using `context` after an async gap post-navigation	Widget may be unmounted	Check `mounted` (08)
Assuming one global Navigator	Nested navigators exist	Target the right Navigator

Best Practices

- Think in **stack operations**: push to go deeper, pop to go back; the top is visible.

- Keep shared/important state **outside** routes (a store) so it survives pops (Module 11).
- Use `PopScope` to intercept back (confirm-exit, unsaved-changes).
- Obtain a **descendant context** for `Navigator.of` when needed.

Performance

Retained lower routes cost memory (their subtrees persist); deep stacks of heavy screens add up. Pop disposes; keep screens disposable and stores external (08 · dispose_and_leaks).

Advantages / Disadvantages

- + Intuitive history/back semantics, preserved back-stack state, matches user expectations.
- − Imperative stack can get tangled for complex flows/deep links (→ declarative/Router — 04_declarative_navigation.md).

Interview Questions

1. 🟢 **How does Flutter model navigation?** — As a stack of routes managed by a `Navigator`; the top route is visible.
2. 🟢 **What do push and pop do?** — Push adds a route on top (new screen); pop removes the top (back), optionally returning a result.
3. 🟡 **What handles the device back button?** — The `Navigator` pops the top route (interceptable via `PopScope` / `WillPopScope`).
4. 🟡 **What happens to a screen below the top?** — It's retained (kept alive with its state) so back returns to it; the popped screen is disposed.
5. 🟡 **How does `Navigator.of(context)` find the navigator?** — It walks up the tree to the nearest `Navigator` (usually the one from `MaterialApp`).
6. 🔴 **Why keep important state outside routes?** — Popped routes are disposed; state living only in a route is lost on back — store it above or in a state solution.
7. 🔴 **When does imperative stack navigation break down?** — Complex flows, deep links, and web URL sync — where declarative Navigator 2.0/Router fits better.

Senior Engineer Tips

- Draw the stack when debugging navigation bugs; most "wrong screen/back" issues are stack-state confusion.
- Keep feature/session state in a store, not in ephemeral route widgets, so navigation doesn't lose it.
- Use `PopScope` for unsaved-changes/exit-confirm rather than hacking the back button.

Architect Perspective

The stack model is the mental foundation for all navigation. Imperative push/pop suffices for simple flows; as apps need URL sync, deep links, and complex flows, you move to declarative Router (04_declarative_navigation.md, Module 13) — but it's still a stack underneath. Keeping navigation-independent state external is a key architectural habit.

Summary

- `Navigator` = a stack of routes; push adds, pop removes; top is visible; back pops.
- Lower routes retain state; popped routes dispose; keep shared state external.
- Foundation for imperative + declarative navigation.

Revision Notes

- Navigation = route stack; push (add top), pop (remove top, returns result).
- Back button = pop (intercept via `PopScope`).
- Lower routes retained; popped disposed; keep state external.
- `Navigator.of(context)` = nearest navigator up the tree.

Practice Questions

1. Why does back return to the previous screen automatically?
2. What happens to a screen's state when it's popped vs when it's below the top?
3. When does the imperative stack become insufficient?

Coding Questions

1. Build a two-screen push/pop app and trace the stack.
2. Add a `PopScope` confirm-exit on the home screen.
3. Demonstrate state loss on pop, then fix by lifting state to a store.

Mini Project

Stack explorer (Flutter): Build a 3-level push/pop flow (Home→List→Details) with a `PopScope` confirm on Home and a shared counter kept in a store so it survives navigation. Log the stack at each step. Acceptance: correct push/pop; state survives pops; back-intercept works; app runs.

Imperative Navigation (`push` / `pop`, Arguments & Results)

> Imperative navigation drives the stack directly with `Navigator.push` / `pop`; pass **arguments** via the destination's constructor and get a **result** back by `await`ing `push` (which completes when the destination `pop`s a value).

Introduction

The everyday API: `Navigator.push(context, MaterialPageRoute(...))` to go forward, `Navigator.pop(context, result)` to go back with data. This file covers argument passing (type-safe via constructors), returning results, and the stack-manipulation variants (`pushReplacement`, `pushAndRemoveUntil`, `popUntil`, `maybePop`).

Why this concept exists

Most navigation is direct and event-driven ("on tap, open details for this item"). Imperative navigation maps intuitively to that: call a method with the data. It's the simplest correct approach for typical flows.

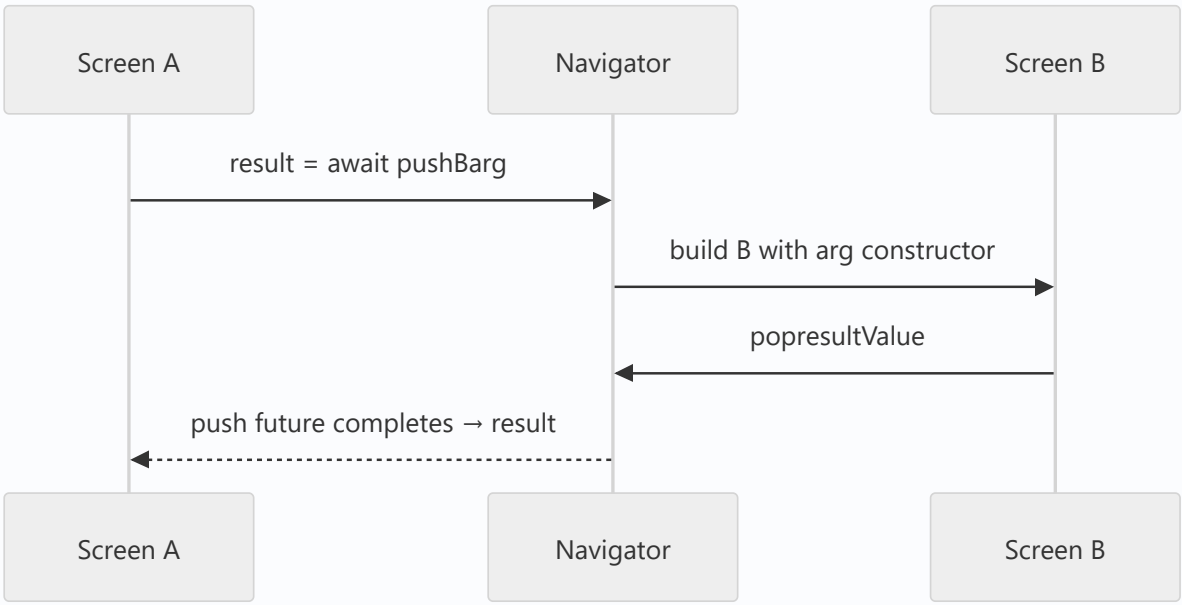
Real-world analogy

Handing someone a **filled-in form** as you send them to another department (arguments), and them handing back a **stamped receipt** when they return (result). You wait at the counter (`await`) until they come back.

Problem Statement

Open a product's detail screen passing the `Product`, let the user edit and return the updated value, and replace the login screen with home after auth (no back to login). You'll use constructor args, `await push`, and `pushReplacement`.

Internal Working



- **Forward**: `Navigator.push(context, MaterialPageRoute(builder: (_) => DetailsScreen(product: p)))` — pass data via the **constructor** (type-safe, preferred over untyped route arguments).
- **Result**: `push` returns a `Future<T?>`; `await` it. The destination calls `pop(value)`; `push` completes with that value (or `null` if popped without one/back-button).
- **Variants**:
  - `pushReplacement` — replace current top (e.g., login → home).
  - `pushAndRemoveUntil(route, predicate)` — push and clear stack up to a condition (e.g., to home).
  - `popUntil(predicate)` — pop repeatedly until a route matches.
  - `maybePop` / `canPop` — pop only if possible.
- **`context.mounted`** must be checked before using `context` after `await push` (08 · setstate_mechanics).

Memory Representation

The pushed route retains its subtree until popped; results are plain returned values (01_navigator_stack.md).

Compiler Behavior

Constructor-based args are **type-checked**; untyped `RouteSettings.arguments` are `Object?` (cast needed) — prefer constructors.

Runtime Behavior

`await push` suspends until the destination pops; the awaiting code resumes with the result. Back-button pop yields `null`.

Flutter Engine Behavior / Dart VM Behavior

Not applicable beyond transition rendering.

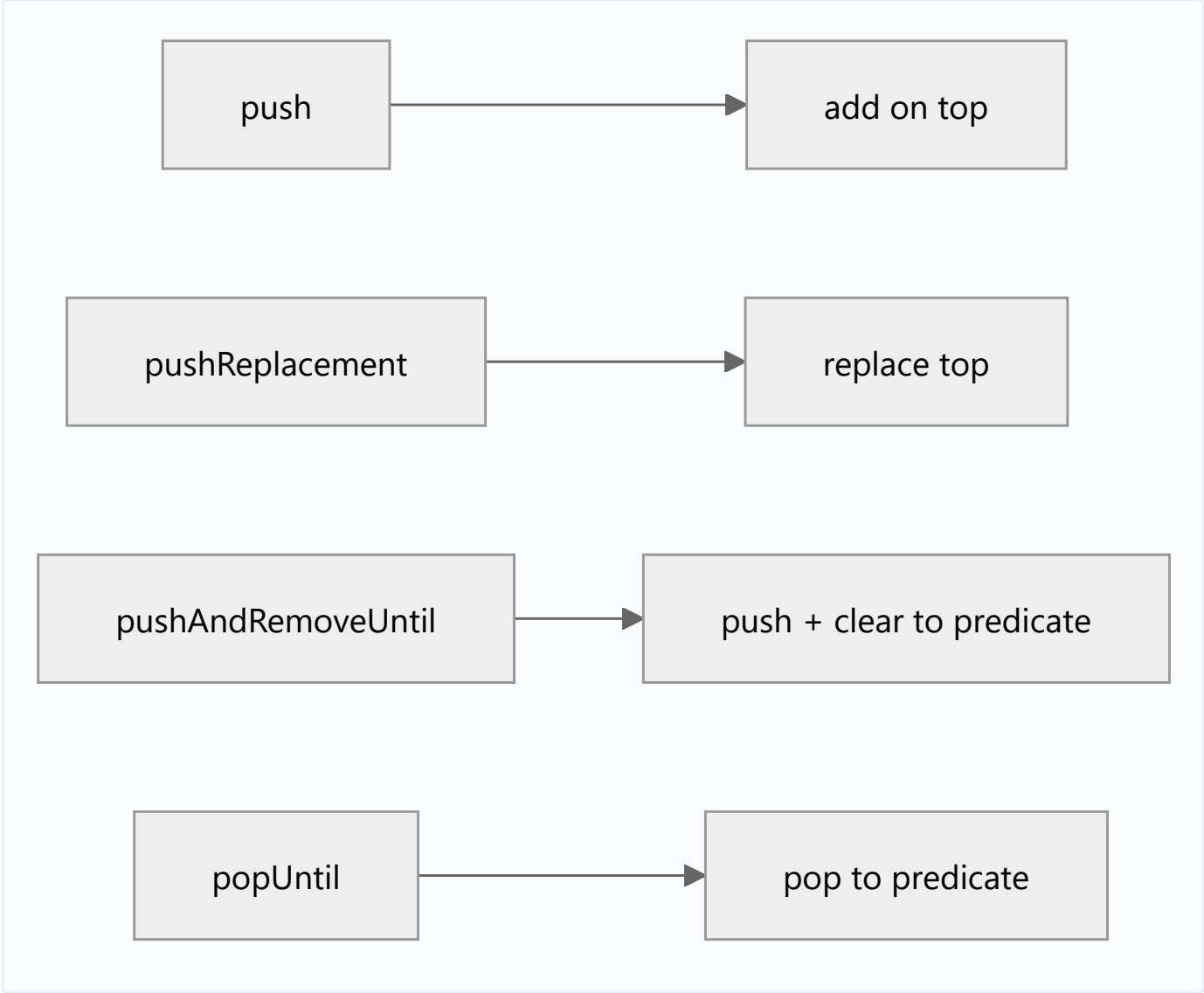
Examples

```
import 'package:flutter/material.dart';


class Product {
  final String name;
  const Product(this.name);
}


class ListScreen extends StatefulWidget {
  const ListScreen({super.key});
  @override
  State<ListScreen> createState() => _ListScreenState();
}
class _ListScreenState extends State<ListScreen> {
  String _lastResult = '';


  Future<void> _openDetails() async {
    // Pass argument via constructor; AWAIT the returned result:
    final result = await Navigator.of(context).push<String>(
      MaterialPageRoute(builder: (_) => const DetailsScreen(product: Product('Widget'))),
    );
    if (!mounted) return;                // guard context after await
    setState(() => _lastResult = result ?? 'cancelled');
  }


  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('List')),
      body: Center(
```

```
        child: Column(mainAxisSize: MainAxisSize.min, children: [
          Text('Result: $_lastResult'),
          ElevatedButton(onPressed: _openDetails, child: const Text('Open details')),
        ]),
      ),
    );
  }
}

class DetailsScreen extends StatelessWidget {
  final Product product;
  const DetailsScreen({super.key, required this.product});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text(product.name)),
      body: Center(
        child: Column(mainAxisSize: MainAxisSize.min, children: [
          ElevatedButton(
            onPressed: () => Navigator.of(context).pop('saved'),   // return a result
            child: const Text('Save & return'),
          ),
          ElevatedButton(
            onPressed: () => Navigator.of(context).pop(),          // returns null
            child: const Text('Cancel'),
          ),
        ]),
      ),
    );
  }
}

// Login -> Home without back to login:
void goHome(BuildContext context) => Navigator.of(context).pushReplacement(
      MaterialPageRoute(builder: (_) => const HomeScreen()),
    );
class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});
  @override
```

```
  Widget build(BuildContext context) => const Scaffold(body: Center(child: Text('Home')));
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using `context` after `await push` without `mounted`	Widget may be unmounted	`if (!mounted) return;`
Untyped route args + casts everywhere	Loses type safety, runtime errors	Pass via constructor
`push` where `pushReplacement` is needed	User can go back to login/splash	Use replacement/removeUntil
Ignoring the returned `Future` result	Miss returned data	`await push<T>()`
Not handling `null` result (back button)	Null crash	Default when result is null

Best Practices

- Pass arguments via the **destination's constructor** (type-safe); reserve untyped route arguments for named routes (03_named_routes.md).
- `await push<T>()` and handle `null` (back/cancel) results.
- Use `pushReplacement` / `pushAndRemoveUntil` for auth/onboarding transitions (no back to those).
- Guard `context` after `await` with `mounted`.
- Keep navigation calls in handlers, not `build`.

Performance

Negligible; each pushed route retains its subtree until popped (01_navigator_stack.md).

Advantages / Disadvantages

- \+ Simple, direct, type-safe (constructor args), easy result passing, fine stack control.
- − Scattered across the codebase; no central URL map; harder for deep links/web (→ declarative/Router).

Interview Questions

1. 🟢 **How do you navigate forward and back imperatively?** — `Navigator.push(context, MaterialPageRoute(...))` and `Navigator.pop(context, [result])`.
2. 🟢 **How do you pass data to a new screen?** — Preferably via the destination widget's constructor (type-safe).
3. 🟡 **How do you get a result back?** — `await Navigator.push<T>(...)`; the destination calls `pop(value)`, completing the future (back button → `null`).
4. 🟡 **`push` vs `pushReplacement` vs `pushAndRemoveUntil`?** — Add on top; replace the current top; push and clear routes up to a predicate (e.g., login→home, splash→home).
5. 🟡 **Why check `mounted` after `await push`?** — The awaiting widget may have been disposed; using its `context` would error.
6. 🔴 **Constructor args vs `RouteSettings.arguments`?** — Constructor args are type-checked; `arguments` is `Object?` (needs casting) — used with named routes where you can't call the constructor directly.
7. 🔴 **How do you return to a specific earlier screen?** — `popUntil((route) => route.settings.name == '/target')` or `pushAndRemoveUntil`.

Senior Engineer Tips

- Prefer constructor injection of arguments for compile-time safety; it also makes destinations independently testable.
- Standardize auth/onboarding transitions on `pushReplacement` / `pushAndRemoveUntil` so users can't back into them.
- Wrap common flows (e.g., `openDetails(context, product)`) in helper functions to keep call sites clean and consistent.

Architect Perspective

Imperative navigation is ideal for simple, event-driven flows and remains widely used. As apps need centralized route maps, deep links, and web URLs, you layer in named routes and ultimately declarative routing (Module 13). Keeping arguments type-safe and results explicit makes navigation testable and refactor-friendly.

Summary

- `push` / `pop` drive the stack; pass args via constructors, get results by `await`ing `push`.
- Use `pushReplacement` / `pushAndRemoveUntil` / `popUntil` for stack control; guard `context` after `await`.
- Simple and type-safe; scale to named/declarative routing for URLs/deep links.

Revision Notes

- `push(MaterialPageRoute)` / `pop([result])`; args via constructor; result via `await push<T>()` (null on back).
- Variants: `pushReplacement`, `pushAndRemoveUntil`, `popUntil`, `maybePop` / `canPop`.
- Guard `context` after `await` (`mounted`); navigate in handlers.
- Constructor args = type-safe (prefer over untyped `arguments`).

Practice Questions

1. How do you get an edited value back from a details screen?
2. Which method sends login→home with no back?
3. Why guard `context` after `await push`?

Coding Questions

1. Push a details screen with a constructor arg and return an edited result.
2. Implement login→home using `pushReplacement`.

3. Use `pushAndRemoveUntil` to return to home from a deep stack.

Mini Project

Edit flow (Flutter): Build List→Detail→Edit: pass a `Product` via constructors, return an edited `Product` up the stack, and after a simulated "save" use `popUntil` to return to the list showing the update. Guard all post-`await` context use. Acceptance: type-safe args; results handled (incl. cancel/null); correct stack control; app runs.

Named Routes & `onGenerateRoute`

> Named routes give each screen a string name (`/details`) registered centrally; `onGenerateRoute` is the programmatic factory that builds a route from its name + arguments — enabling a single source of truth for navigation and dynamic/validated routing.

Introduction

Instead of constructing `MaterialPageRoute`s inline everywhere, you register a **route table** (`routes:`) and/or an **`onGenerateRoute`** callback on `MaterialApp` , then navigate by name: `Navigator.pushNamed(context, '/details', arguments: ...)` . This centralizes routing and supports argument passing and dynamic route logic.

Why this concept exists

Inline `MaterialPageRoute`s scatter navigation logic and duplicate screen wiring. Named routes centralize the map (one place to see all screens), enable dynamic/parameterized routing via `onGenerateRoute` , and are a stepping stone toward URL-based/declarative routing (Module 13).

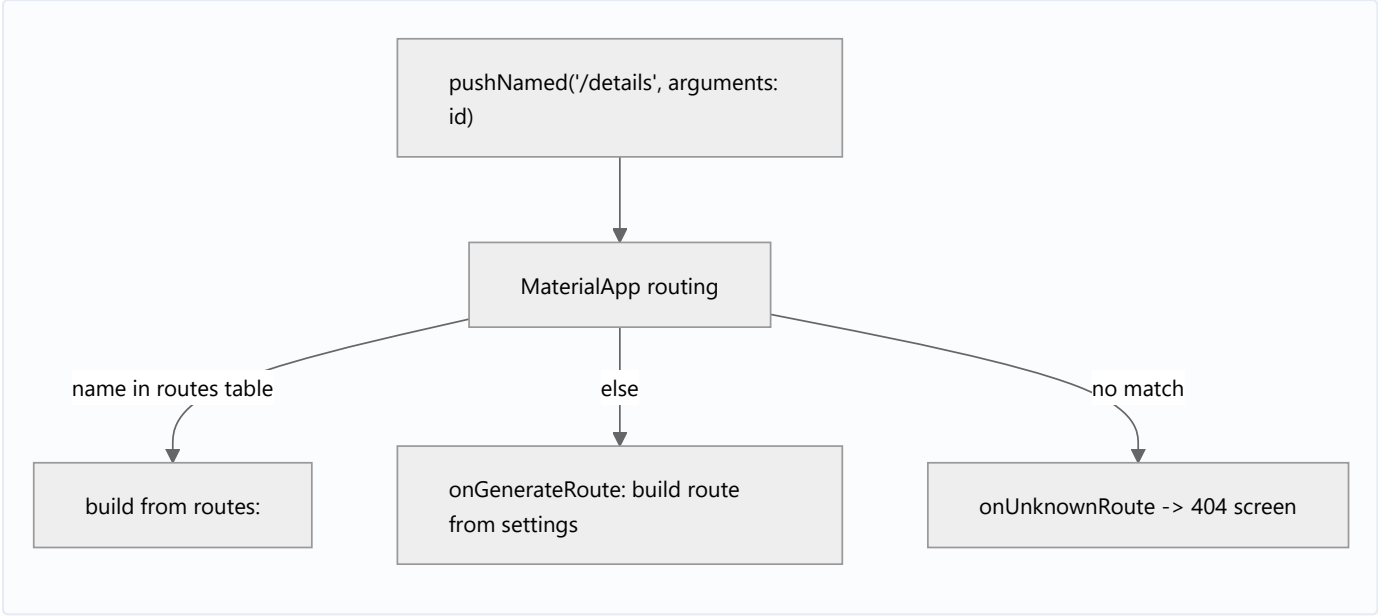
Real-world analogy

A **hotel directory + concierge**: rooms have names ("Suite 12"); the directory (`routes:`) lists fixed ones, and the concierge (`onGenerateRoute`) handles dynamic requests ("a room for this specific guest"), validating and directing appropriately.

Problem Statement

You want a central route map, to pass a product id to `/details` , and to handle unknown routes gracefully. You'll use `routes:` , `onGenerateRoute` , and `onUnknownRoute` .

Internal Working



- `routes: {'/': ..., '/list': ...}`: a static map of name → `WidgetBuilder` for simple, argument-less screens.
- `onGenerateRoute(RouteSettings settings)`: called for names not in `routes` (or for all, if you omit `routes`); read `settings.name` + `settings.arguments` and return a `Route` (e.g., `MaterialPageRoute`). This is where you do **parameterized/validated/dynamic** routing.
- `onUnknownRoute`: fallback for unmatched names (a 404/not-found screen).
- **Arguments**: passed via `pushNamed(name, arguments: obj)` and read from `settings.arguments` (typed via cast — less type-safe than constructors, so validate).
- `initialRoute`: the first route name at startup.

Memory Representation

Same route-stack behavior as imperative (01_navigator_stack.md); the route table/factory is config on `MaterialApp`.

Compiler Behavior

`settings.arguments` is `Object?` → runtime cast; centralize + validate to avoid scattered unsafe casts.

Runtime Behavior

`pushNamed` looks up `routes` first, then `onGenerateRoute`, then `onUnknownRoute`. `onGenerateRoute` runs per navigation, so it can branch on arguments/auth/etc.

Flutter Engine Behavior / Dart VM Behavior

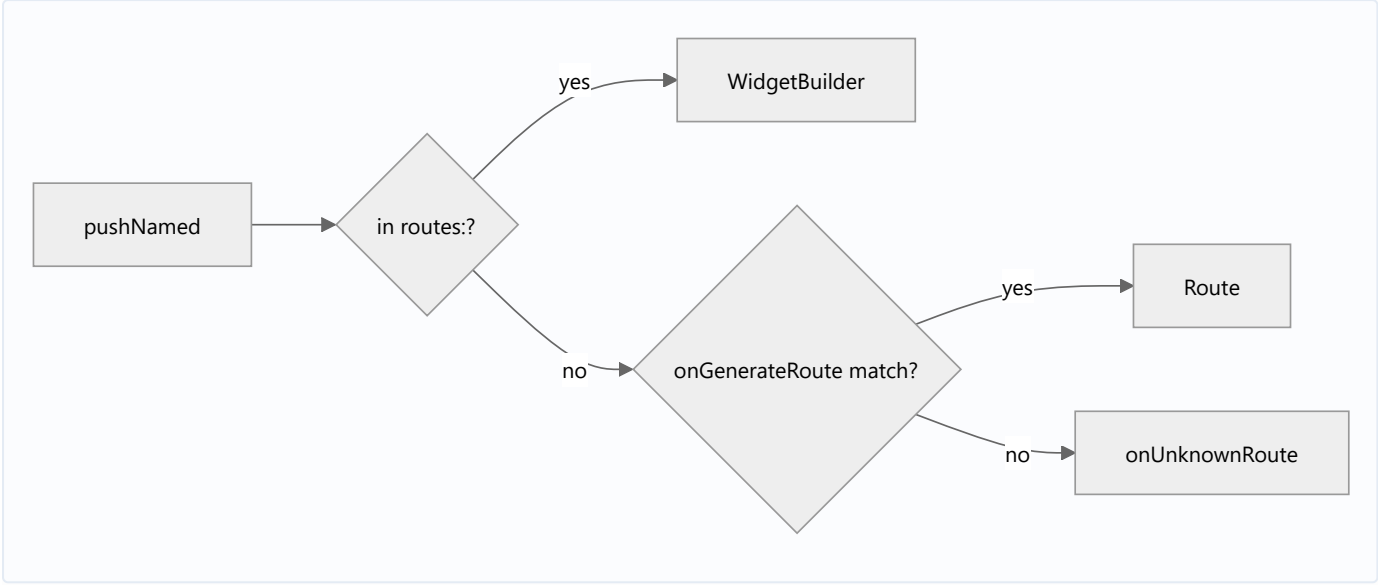
Not applicable.

Examples

```
import 'package:flutter/material.dart';


class DetailsArgs {
  final String id;
  const DetailsArgs(this.id);
}


void main() {
  runApp(MaterialApp(
    initialRoute: '/',
    routes: {
      '/': (_) => const HomeScreen(),        // simple, no args
      '/list': (_) => const ListScreen(),
    },
    // Dynamic/parameterized routes + validation:
    onGenerateRoute: (settings) {
      switch (settings.name) {
        case '/details':
          final args = settings.arguments;
          if (args is DetailsArgs) {
            return MaterialPageRoute(
              builder: (_) => DetailsScreen(id: args.id),
              settings: settings, // keep name for popUntil/analytics
            );
          }
          return _errorRoute('Missing/invalid details args');
        default:
          return null; // fall through to onUnknownRoute
      }
    },
    onUnknownRoute: (settings) => _errorRoute('Unknown route: ${settings.name}'),
  ));
}
```

```
MaterialPageRoute _errorRoute(String message) => MaterialPageRoute(
      builder: (_) => Scaffold(body: Center(child: Text(message))),
    );

class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});
  @override
  Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(title: const Text('Home')),
        body: Center(
          child: ElevatedButton(
            onPressed: () => Navigator.pushNamed(
              context, '/details', arguments: const DetailsArgs('42')),
            child: const Text('Open details 42'),
          ),
        ),
      );
}
class ListScreen extends StatelessWidget {
  const ListScreen({super.key});
  @override
  Widget build(BuildContext context) => const Scaffold(body: Center(child: Text('List')));
}
class DetailsScreen extends StatelessWidget {
  final String id;
  const DetailsScreen({super.key, required this.id});
  @override
  Widget build(BuildContext context) =>
      Scaffold(appBar: AppBar(title: Text('Details $id')));
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Unvalidated `settings.arguments` cast	Runtime crash on wrong/absent args	Type-check + fallback in `onGenerateRoute`
No `onUnknownRoute`	Crash/blank on bad name	Provide a 404 route
Putting arg-heavy screens in static `routes:`	Can't pass typed args cleanly	Use `onGenerateRoute`
Losing `settings` on the built route	`popUntil(name)` /analytics break	Pass `settings: settings`
Duplicating route names/strings	Typos, drift	Centralize route-name constants

Best Practices

- Keep a **central route map** (constants for names) as a single source of truth.
- Use static `routes:` for simple screens; `onGenerateRoute` for parameterized/validated/guarded routing.
- **Validate** `arguments` and provide `onUnknownRoute` (404).
- Preserve `settings` on generated routes (for `popUntil` /analytics/deep links).
- Consider graduating to `go_router` for URL/deep-link-heavy apps (Module 13).

Performance

Negligible; `onGenerateRoute` runs per navigation (keep it light). Same stack cost as imperative (01_navigator_stack.md).

Advantages / Disadvantages

- + Centralized route map, dynamic/parameterized/validated routing, unknown-route handling, easier analytics/deep-link hookup.
- − `arguments` is untyped (validate), string names can drift, still imperative under the hood; `go_router` is often better for complex URL routing.

Interview Questions

1. 🟢 **What are named routes?** — Screens registered by string name (`routes:` map) and navigated via `Navigator.pushNamed` .
2. 🟢 **What is `onGenerateRoute` ?** — A callback that builds a `Route` from `RouteSettings` (name + arguments) for names not in the static table — enabling dynamic/parameterized routing.
3. 🟡 **How do you pass arguments to a named route?** — `pushNamed(name, arguments: obj)` , read via `settings.arguments` (cast/validate).
4. 🟡 **What is `onUnknownRoute` for?** — Handling unmatched route names (a 404/not-found screen).
5. 🟡 **Static `routes:` vs `onGenerateRoute` ?** — `routes:` for simple, argument-less screens; `onGenerateRoute` for parameterized/validated/guarded routing.
6. 🔴 **Why validate `settings.arguments` ?** — It's `Object?` ; a wrong/missing type crashes on cast — validate and fall back gracefully.
7. 🔴 **When move beyond named routes?** — For URL sync, deep links, nested/guarded routing at scale — use `go_router` /declarative routing (Module 13).

Senior Engineer Tips

- Centralize route names as constants (or an enum) to avoid typos and enable refactoring.
- Do route **guards/validation** in `onGenerateRoute` (auth checks, arg validation) as an early step toward guarded routing.
- Always pass `settings:` to generated routes so `popUntil(name)` , analytics, and deep links work.

Architect Perspective

Named routes + `onGenerateRoute` centralize navigation into a single, testable map and introduce a place for guards/validation — the conceptual bridge to declarative, URL-driven routing. For anything beyond simple apps (deep links, web, nested/guarded flows), this evolves into `go_router` /Navigator 2.0 (Module 13).

Summary

- Register routes by name (`routes:`) and/or build them dynamically in `onGenerateRoute` ; handle unmatched names with `onUnknownRoute` .
- Pass/validate arguments via `settings.arguments` ; preserve `settings` ; centralize names.
- Great for a central map + dynamic routing; graduate to `go_router` for URLs/deep links.

Revision Notes

- `routes:` (static, no-arg) + `onGenerateRoute` (dynamic/params/validate) + `onUnknownRoute` (404).
- `pushNamed(name, arguments:)` ; read `settings.arguments` (cast+validate).
- Pass `settings:` to generated routes; centralize name constants.
- Beyond simple apps → `go_router` (Module 13).

Practice Questions

1. When use `onGenerateRoute` over the static `routes:` map?
2. Why validate `settings.arguments` ?
3. What breaks if you don't pass `settings` to a generated route?

Coding Questions

1. Build a named-route app with `/` , `/list` , and a parameterized `/details` .
2. Add `onUnknownRoute` + argument validation with a fallback.
3. Centralize route names as constants and refactor call sites.

Mini Project

Central route map (Flutter): Build an app with a route-name constants file, static `routes:` for simple screens, `onGenerateRoute` for a validated parameterized `/details` , and `onUnknownRoute` 404. Add a simple auth guard in `onGenerateRoute` . Acceptance: centralized names; validated args; 404 handled; guard works; app runs.

Declarative Navigation (Navigator 2.0 / Router / `Page` API)

> Declarative navigation makes the route stack a **function of app state**: you provide a list of `Page`s and the framework reconciles the `Navigator` to match — enabling URL-sync, deep links, and complex flows, at the cost of more boilerplate (usually hidden behind `go_router`).

Introduction

Navigator 2.0 (the Router API) flips navigation from imperative ("push this") to declarative ("the stack *is* `[A, B, C]` given this state"). You build `Page`s from state; the `Navigator` diffs and updates. This file explains the model, its pieces (`Router` / `RouterDelegate` / `RouteInformationParser` / `Page`), and when you actually need it.

Why this concept exists

Imperative push/pop can't cleanly express: browser URL ↔ stack sync, deep links restoring a whole stack, or "stack derived from auth/onboarding state." Declarative navigation makes the stack a pure projection of state (like the rest of Flutter's UI = f(state)), which handles these correctly.

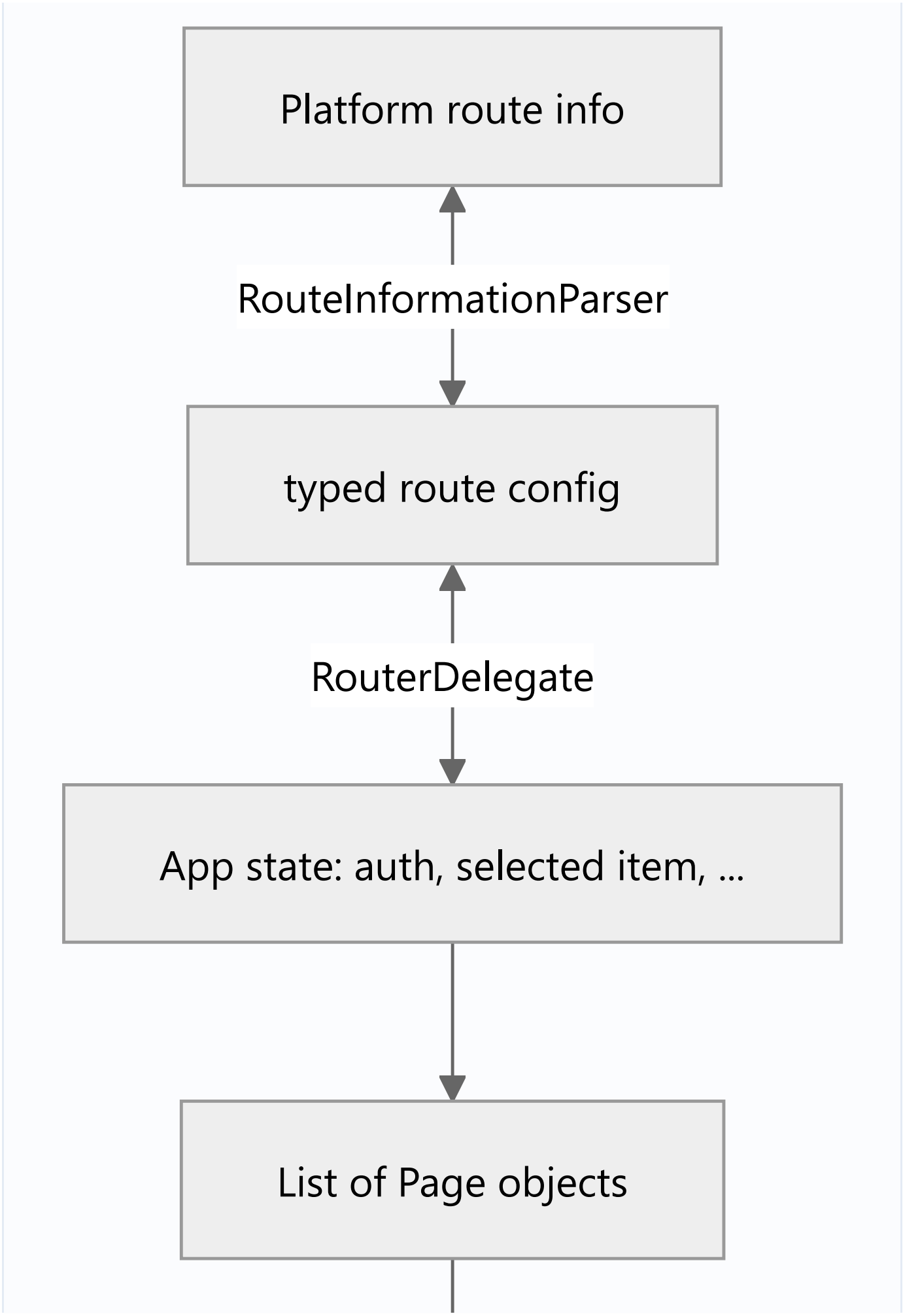
Real-world analogy

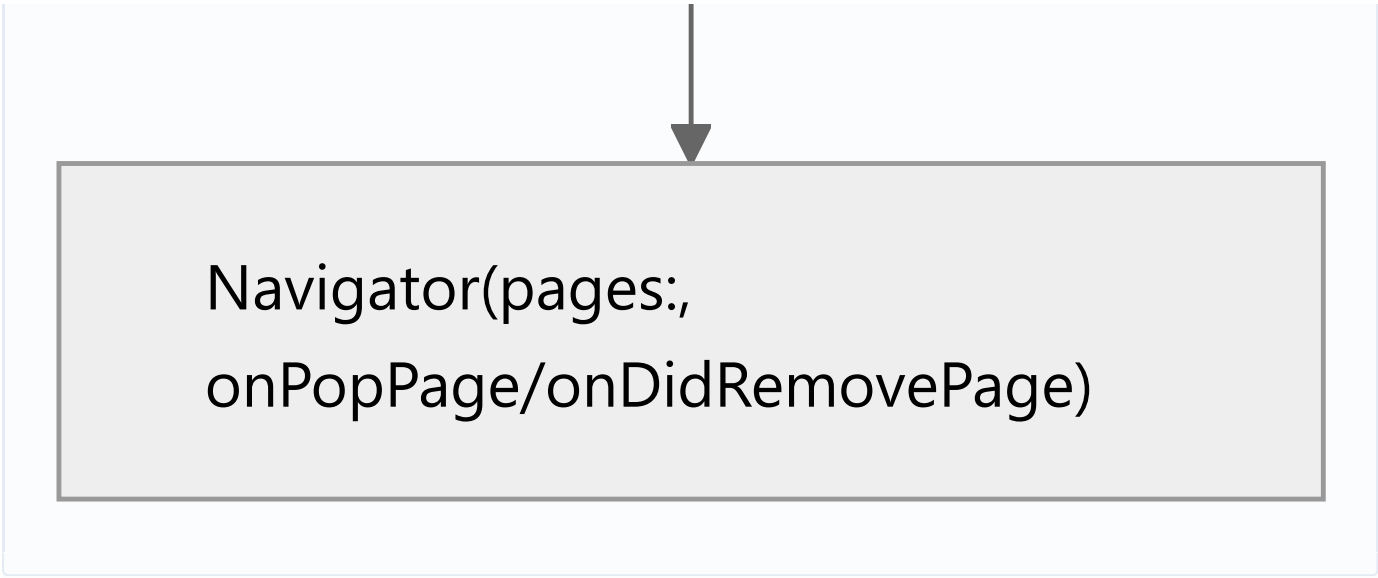
Imperative is **giving turn-by-turn driving directions**; declarative is **stating your destination address** and letting the GPS compute the whole route from current position. You describe the desired end-state (the page list); the framework figures out the pushes/pops.

Problem Statement

A web app must sync the URL with the screen and restore the right stack from a deep link; navigation should reflect auth state (logged-out → login stack). You'll model pages from state (conceptually), and see why `go_router` is the practical tool.

Internal Working


Platform route info
RouteInformationParser
typed route config
RouterDelegate
App state: auth, selected item, ...
List of Page objects



- `Navigator(pages: [...], onDidRemovePage: ...)`: the declarative form — you pass the **current page list**; the framework reconciles (pushes new, pops removed).
- `Page`: an immutable description of a route (like a widget for a screen); `MaterialPage` / `CupertinoPage` wrap a child.
- **Router pieces** (full 2.0):
  - `RouteInformationParser`: platform route info (URL) ↔ a typed configuration.
  - `RouterDelegate`: holds app navigation state, builds the `Navigator` (pages) from it, and handles pops/new routes.
  - `RouteInformationProvider` / `BackButtonDispatcher`: platform integration.
- **Reconciliation**: change the state → rebuild the page list → `Navigator` diffs it (add/remove routes), like widget reconciliation (06 · widgets_elements_render_objects).
- **Reality**: raw Navigator 2.0 is verbose; most teams use `go_router` (built on it) — covered in Module 13.

Memory Representation

Pages are immutable descriptions; the `Navigator` maintains the underlying route stack/state (01_navigator_stack.md).

Compiler Behavior

Not applicable (typed route configs improve safety in `go_router`/typed routes).

Runtime Behavior

State change → new page list → diff → route add/remove with transitions. URL changes drive the parser → delegate → state; state changes update the URL (two-way sync on web).

Flutter Engine Behavior / Dart VM Behavior

Not applicable beyond transition rendering.

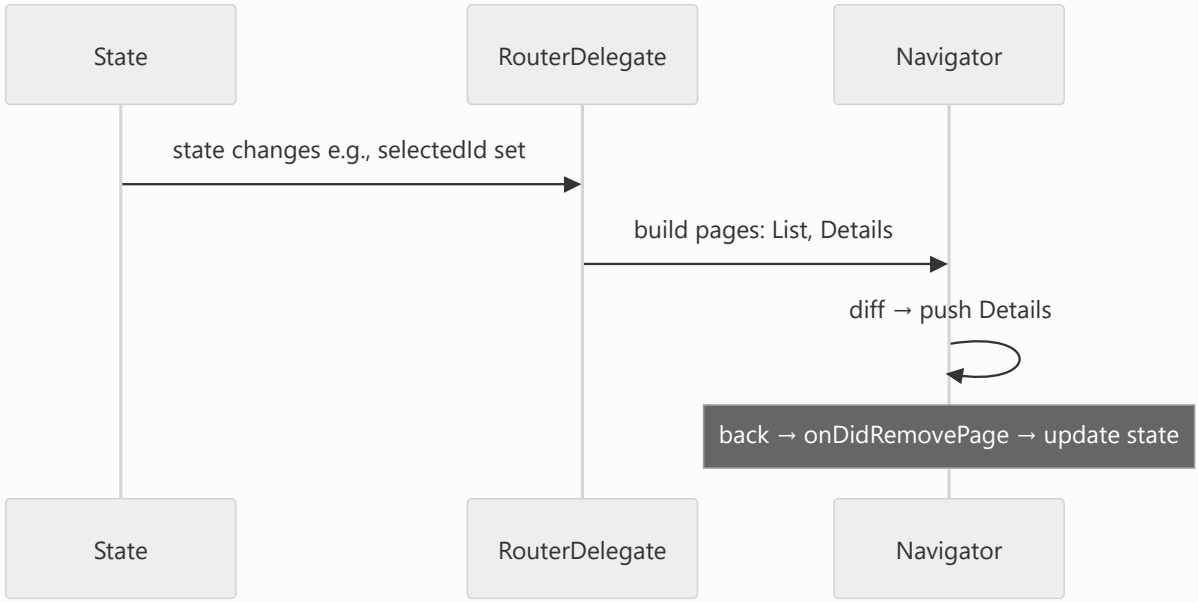
Examples

```
import 'package:flutter/material.dart';

// Declarative stack from state using the pages API (simplified, no URL parser):
class AppShell extends StatefulWidget {
  const AppShell({super.key});
  @override
  State<AppShell> createState() => _AppShellState();
}
class _AppShellState extends State<AppShell> {
  String? selectedId; // app state drives the stack

  @override
  Widget build(BuildContext context) {
    return Navigator(
      // The stack IS a function of state:
      pages: [
        const MaterialPage(child: ListPage(), name: '/list'),
        if (selectedId != null)
          MaterialPage(child: DetailsPage(id: selectedId!), name: '/details'),
      ],
      onDidRemovePage: (page) {
        // reconcile state when a page is popped (e.g., back on details)
        if (page.name == '/details') setState(() => selectedId = null);
      },
    );
  }
}
// (In a real app, ListPage would setState(selectedId=...) to "navigate".)

class ListPage extends StatelessWidget {
  const ListPage({super.key});
  @override
  Widget build(BuildContext context) =>
      Scaffold(appBar: AppBar(title: const Text('List (declarative)')));
```

```
}
class DetailsPage extends StatelessWidget {
  final String id;
  const DetailsPage({super.key, required this.id});
  @override
  Widget build(BuildContext context) =>
      Scaffold(appBar: AppBar(title: Text('Details $id')));
}
```

```
// In practice, use go_router (built on Navigator 2.0) — see Module 13:
// final router = GoRouter(routes: [ GoRoute(path: '/', ...), GoRoute(path: '/details/:id', ...) ]);
// MaterialApp.router(routerConfig: router);
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Hand-rolling full Navigator 2.0	Very verbose/error-prone	Use `go_router` (Module 13)
Not syncing state on pop	Stack/state diverge	Handle `onDidRemovePage` /pop callbacks
Using declarative for a simple app	Unneeded complexity	Stick to imperative/named routes
Duplicating navigation state	Two sources of truth	State drives pages (single source)
Ignoring URL/deep-link parsing	Broken web/deep links	Implement parser or use `go_router`

Best Practices

- Use declarative routing when you need **URL sync, deep links, or state-driven stacks**; otherwise imperative/named is simpler.
- Make the **page list a pure function of app state** (single source of truth); reconcile state on pops.
- In practice, adopt `go_router` (built on Navigator 2.0) rather than hand-writing delegates/parsers.
- Keep navigation state in your state layer (Module 11) so pages derive from it.

Performance

Reconciliation is efficient (diff pages like widgets); no notable overhead vs imperative. The cost is conceptual/boilerplate, mitigated by `go_router` (09 · build_phase).

Advantages / Disadvantages

- **+** Stack = f(state), correct URL/deep-link handling, restores full stacks, fits Flutter's declarative model, testable navigation state.
- **−** Verbose in raw form (delegates/parsers), steeper concepts, overkill for simple apps — hence `go_router`.

Interview Questions

1. 🟢 **Imperative vs declarative navigation?** — Imperative: you call push/pop. Declarative: you describe the stack as a list of `Page`s derived from state and the framework reconciles it.
2. 🟢 **What is a `Page`?** — An immutable description of a route (like a widget for a screen); e.g., `MaterialPage`.
3. 🟡 **What are the Router API pieces?** — `RouteInformationParser` (URL ↔ config), `RouterDelegate` (state → pages, handle pops/new routes), plus provider/back-button dispatcher.
4. 🟡 **When do you need declarative navigation?** — URL sync (web), deep links restoring a stack, and state-driven navigation (e.g., auth) — cases imperative handles poorly.
5. 🟡 **Why is raw Navigator 2.0 rarely hand-written?** — It's verbose/error-prone; `go_router` (built on it) provides the ergonomics most apps want.
6. 🔴 **How does declarative navigation mirror Flutter's UI model?** — Stack = f(state), just like UI = f(state); state changes rebuild the page list, which the Navigator diffs.
7. 🔴 **How do deep links restore a stack?** — The parser turns the URL into a typed config; the delegate builds the corresponding page list, so opening a deep link recreates the full stack.

Senior Engineer Tips

- Reach for declarative routing when URLs/deep links/state-driven stacks are requirements; don't adopt it for simple push/pop apps.
- Use `go_router` in practice; understand Navigator 2.0 so you can reason about and debug it.
- Keep navigation a projection of state — it makes deep links and testing straightforward.

Architect Perspective

Declarative navigation aligns navigation with the app's state model, which is essential for web, deep linking, and complex/guarded flows. It's the foundation of `go_router` and modern routing architecture (Module 13); treating the stack as derived state (single source of truth) yields robust, testable, link-friendly navigation at scale.

Summary

- Declarative navigation: the route stack is a list of `Page`s derived from state, reconciled by the `Navigator` (Router API: parser + delegate).
- Use it for URL sync/deep links/state-driven stacks; use `go_router` rather than hand-rolling.
- Stack = f(state), mirroring Flutter's UI model; keep navigation state in your state layer.

Revision Notes

- Declarative = `Navigator(pages: fromState)`; framework diffs/reconciles.
- Router 2.0: `RouteInformationParser` (URL↔config) + `RouterDelegate` (state→pages, pops).
- Needed for URL/deep links/state-driven stacks; use `go_router` in practice.
- Reconcile state on pop (`onDidRemovePage`); single source of truth.

Practice Questions

1. Why is declarative navigation better for deep links/web?
2. What do the parser and delegate each do?
3. When is imperative navigation still the right choice?

Coding Questions

1. Build a state-driven `Navigator(pages:)` stack (list→details via a state field).
2. Handle back via `onDidRemovePage` to keep state in sync.
3. Sketch how a `/details/:id` URL would map to a page list.

Mini Project

State-driven stack (Flutter): Build a declarative `Navigator(pages:)` whose stack derives from an app-state field (e.g., `selectedId`), with back reconciled to state. Then note (docs) how `go_router` would express the same with URL/deep-link support. Acceptance: stack = f(state); back updates state; single source of truth; app runs. (Continues in Module 13 Routing.)

Nested Navigators & Tabs

> A bottom-nav/tab app usually wants **each tab to keep its own navigation stack and state**; achieve this with a `Navigator` per tab (nested) inside an `IndexedStack`, so switching tabs preserves each tab's stack and scroll position.

Introduction

Real apps have tabs where each tab is its own mini-app with its own back stack (Instagram-style). This file covers nesting `Navigator`s, preserving tab state with `IndexedStack`, handling the back button across nested navigators, and the tradeoffs.

Why this concept exists

A single global navigator can't independently preserve each tab's stack/scroll/state. Nested navigators give each tab an isolated stack; `IndexedStack` keeps inactive tabs alive so switching back restores exactly where you were.

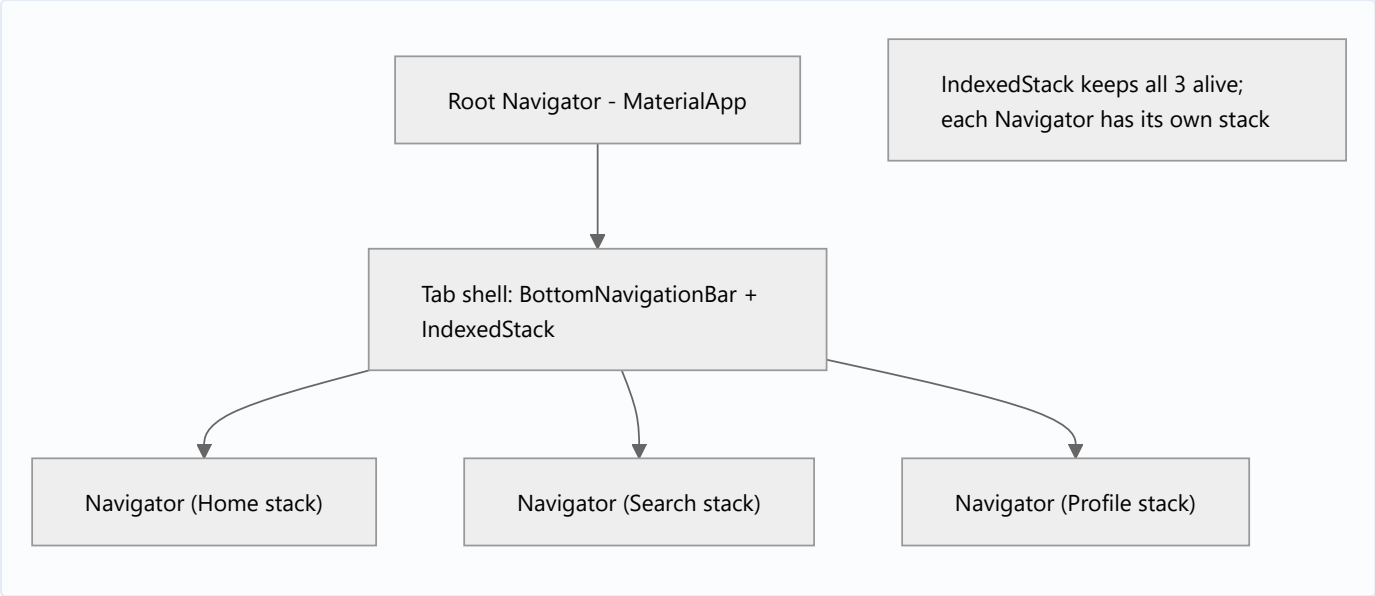
Real-world analogy

A **building with multiple elevators**, one per wing (tab). Each elevator remembers its own current floor (stack position). Switching wings doesn't reset the other elevators — they stay where you left them.

Problem Statement

A 3-tab app (Home/Search/Profile) where each tab can push detail screens and, when you switch away and back, resumes its own stack + scroll. You'll nest a `Navigator` per tab inside an `IndexedStack` and handle back.

Internal Working



- `IndexedStack`: renders all children but shows one (by `index`); inactive children stay in the tree (state/stack preserved). (Alternative: rebuild per tab to save memory, losing state.)
- **A `Navigator` per tab**: each tab's content is wrapped in its own `Navigator` with its own routes → independent back stacks.
- **Back button across nested navigators**: use `PopScope` / `NavigatorState.maybePop` so back first pops the **active tab's** nested navigator, and only exits/handles at the root when that tab's stack is at its root.
- **Keys**: give each nested `Navigator` a `GlobalKey<NavigatorState>` to control it (pop to root on re-tap, etc.).
- **State preservation** also needs `AutomaticKeepAliveClientMixin` or `IndexedStack` for scroll/list state (06 · widgets_elements_render_objects).

Memory Representation

`IndexedStack` keeps all tab subtrees (and their nested stacks) alive → higher memory but preserved state; rebuilding tabs saves memory but loses state (08 · dispose_and_leaks).

Compiler Behavior

Not applicable.

Runtime Behavior

Switching `IndexedStack.index` shows a different, already-built subtree (instant, preserved). Back pops the active nested navigator first; re-tapping a tab commonly pops that tab to its root.

Flutter Engine Behavior / Dart VM Behavior

Not applicable beyond keeping subtrees alive.

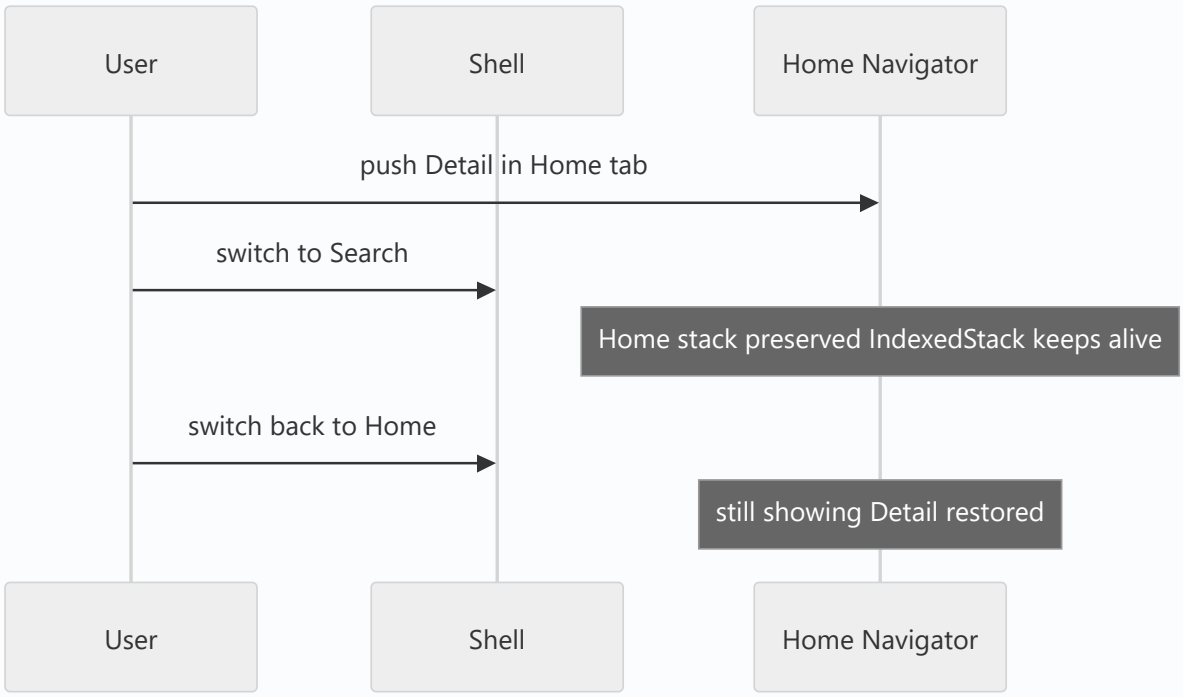
Examples

```
import 'package:flutter/material.dart';


class TabShell extends StatefulWidget {
  const TabShell({super.key});
  @override
  State<TabShell> createState() => _TabShellState();
}
class _TabShellState extends State<TabShell> {
  int _index = 0;
  // One Navigator key per tab (control + pop-to-root):
  final _navKeys = List.generate(3, (_) => GlobalKey<NavigatorState>());


  @override
  Widget build(BuildContext context) {
    return PopScope(
      canPop: false, // intercept root back
      onPopInvokedWithResult: (didPop, _) async {
        if (didPop) return;
        // Back first pops the ACTIVE tab's nested navigator:
        final popped = await _navKeys[_index].currentState!.maybePop();
        if (!popped && mounted) {
          // active tab at its root -> could exit or go to first tab
          if (_index != 0) setState(() => _index = 0);
        }
      },
      child: Scaffold(
        body: IndexedStack(               // keeps all tabs alive (state preserved)
          index: _index,
          children: [
            _tabNavigator(0, const HomeTab()),
            _tabNavigator(1, const SearchTab()),
            _tabNavigator(2, const ProfileTab()),
          ],
        ),
```

```
        bottomNavigationBar: BottomNavigationBar(
          currentIndex: _index,
          onTap: (i) {
            if (i == _index) {
              _navKeys[i].currentState!.popUntil((r) => r.isFirst); // re-tap: pop to root
            } else {
              setState(() => _index = i);
            }
          },
          items: const [
            BottomNavigationBarItem(icon: Icon(Icons.home), label: 'Home'),
            BottomNavigationBarItem(icon: Icon(Icons.search), label: 'Search'),
            BottomNavigationBarItem(icon: Icon(Icons.person), label: 'Profile'),
          ],
        ),
      ),
    );
  }

  Widget _tabNavigator(int i, Widget root) => Navigator(
        key: _navKeys[i],
        onGenerateRoute: (_) => MaterialPageRoute(builder: (_) => root),
      );
}

class HomeTab extends StatelessWidget {
  const HomeTab({super.key});
  @override
  Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(title: const Text('Home')),
        body: Center(
          child: ElevatedButton(
            onPressed: () => Navigator.of(context).push(
                MaterialPageRoute(builder: (_) => const _Detail('Home detail'))),
            child: const Text('Push detail (stays in tab)'),
          ),
        ),
      );
}
class SearchTab extends StatelessWidget {
```

```
  const SearchTab({super.key});
  @override
  Widget build(BuildContext context) => Scaffold(appBar: AppBar(title: const Text('Search')));
}
class ProfileTab extends StatelessWidget {
  const ProfileTab({super.key});
  @override
  Widget build(BuildContext context) => Scaffold(appBar: AppBar(title: const Text('Profile')));
}
class _Detail extends StatelessWidget {
  final String title;
  const _Detail(this.title);
  @override
  Widget build(BuildContext context) => Scaffold(appBar: AppBar(title: Text(title)));
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
One global navigator for tabs	Tabs lose independent stacks/state	Nested `Navigator` per tab
Rebuilding tabs on switch	Loses scroll/stack state	Use `IndexedStack` (or keep-alive)
Ignoring back-button across nesting	Back exits app instead of popping tab	`PopScope` + active nested `maybePop`
Keeping all tabs alive with heavy content	High memory	Balance: lazy tabs / dispose heavy ones
No pop-to-root on re-tap	Poor UX	`popUntil((r)=>r.isFirst)` on re-tap

Best Practices

- Give **each tab its own** `Navigator` (with a `GlobalKey`) for independent stacks.
- Use `IndexedStack` to preserve tab state/scroll (or `PageStorage`/keep-alive selectively).
- Handle **back** to pop the active tab's nested navigator first (`PopScope` + `maybePop`).
- Implement **pop-to-root on re-tap** for expected UX.
- Balance memory: keep-alive is great for a few tabs; for many/heavy tabs consider lazy build/dispose.

Performance

`IndexedStack` trades memory (all tabs alive) for instant, state-preserving switches. For many heavy tabs, that memory adds up — consider lazy instantiation (Module 21).

Advantages / Disadvantages

- **+** Independent per-tab stacks + preserved state/scroll, native-feeling tab UX, controllable via keys.
- **−** More complex back handling, higher memory with `IndexedStack`, careful wiring needed.

Interview Questions

1. 🟢 **How do you give each tab its own back stack?** — Wrap each tab's content in its own nested `Navigator`.
2. 🟢 **How do you preserve each tab's state when switching?** — Use `IndexedStack` (keeps all tab subtrees alive) or keep-alive mechanisms.
3. 🟡 **How do you handle the back button with nested navigators?** — Intercept with `PopScope` and first `maybePop` the active tab's nested navigator; only handle at the root when that stack is at its root.

4. 🟡 **Why give nested navigators a `GlobalKey`?** — To control them (pop-to-root on re-tap, programmatic navigation) from the shell.
5. 🟡 **`IndexedStack` vs rebuilding tabs?** — `IndexedStack` preserves state (more memory); rebuilding saves memory but loses state/scroll.
6. 🔴 **What's the memory tradeoff of keep-alive tabs?** — All tab subtrees (and their stacks) stay in memory; fine for a few tabs, costly for many/heavy ones.
7. 🔴 **How do you implement "tap active tab → go to its root"?** — `popUntil((r) => r.isFirst)` on that tab's nested `NavigatorState`.

Senior Engineer Tips

- Standardize the tab-shell pattern (per-tab Navigator + `IndexedStack` + back handling) as a reusable widget; it's fiddly to get right each time.
- For URL/deep-link support with tabs, use `go_router`'s `StatefulShellRoute` (nested navigators + deep links) — Module 13.
- Watch memory with many heavy tabs; lazy-init or dispose the least-used.

Architect Perspective

Nested navigators model multi-stack apps (tabbed shells) correctly, preserving per-section state — a common product requirement. Modern routing (`go_router` `StatefulShellRoute`) generalizes this with deep-link support, making the tab-shell a first-class, URL-addressable structure (Module 13) — important for web and shareable links at scale.

Summary

- Give each tab its own nested `Navigator` (with a `GlobalKey`); use `IndexedStack` to preserve state.
- Handle back by popping the active tab's navigator first (`PopScope` + `maybePop`); pop-to-root on re-tap.
- Balance memory; for deep links/URLs use `go_router`'s stateful shell.

Revision Notes

- Per-tab nested `Navigator` (GlobalKey) + `IndexedStack` (keep-alive) = independent, preserved stacks.
- Back: `PopScope` → active tab `maybePop` → root.
- Re-tap active tab → `popUntil(isFirst)`.
- Memory: keep-alive costs; deep links/URLs → `go_router` `StatefulShellRoute`.

Practice Questions

1. Why does a single navigator fail for independent tab stacks?
2. How does `IndexedStack` preserve tab state?
3. How do you make back pop the active tab, not exit?

Coding Questions

1. Build a 3-tab shell with per-tab nested navigators + `IndexedStack`.
2. Implement correct back handling and pop-to-root on re-tap.
3. Measure memory difference between `IndexedStack` and rebuild-per-tab.

Mini Project

Tabbed shell (Flutter): Build a bottom-nav app where each tab has its own `Navigator` and can push details; switching tabs preserves each stack (`IndexedStack`); back pops the active tab first; re-tapping a tab pops it to root. Acceptance: independent preserved stacks; correct back behavior; pop-to-root works; app runs.

Route Transitions & `Hero` Animations

> Customize how screens animate in/out with `PageRouteBuilder` + a `transitionsBuilder`, and animate a shared element between screens with `Hero` — polished motion that reinforces spatial continuity.

Introduction

By default `MaterialPageRoute` gives platform-appropriate transitions. This file shows how to build **custom route transitions** (fade/slide/scale) via `PageRouteBuilder`, and how `Hero` animates a shared widget (e.g., a thumbnail growing into a detail image) across a route change.

Why this concept exists

Motion communicates relationships and hierarchy. Custom transitions match brand/UX; `Hero` provides spatial continuity ("this thumbnail *became* this detail"), which reduces cognitive load and feels polished. Flutter builds these on its animation system (Module 22).

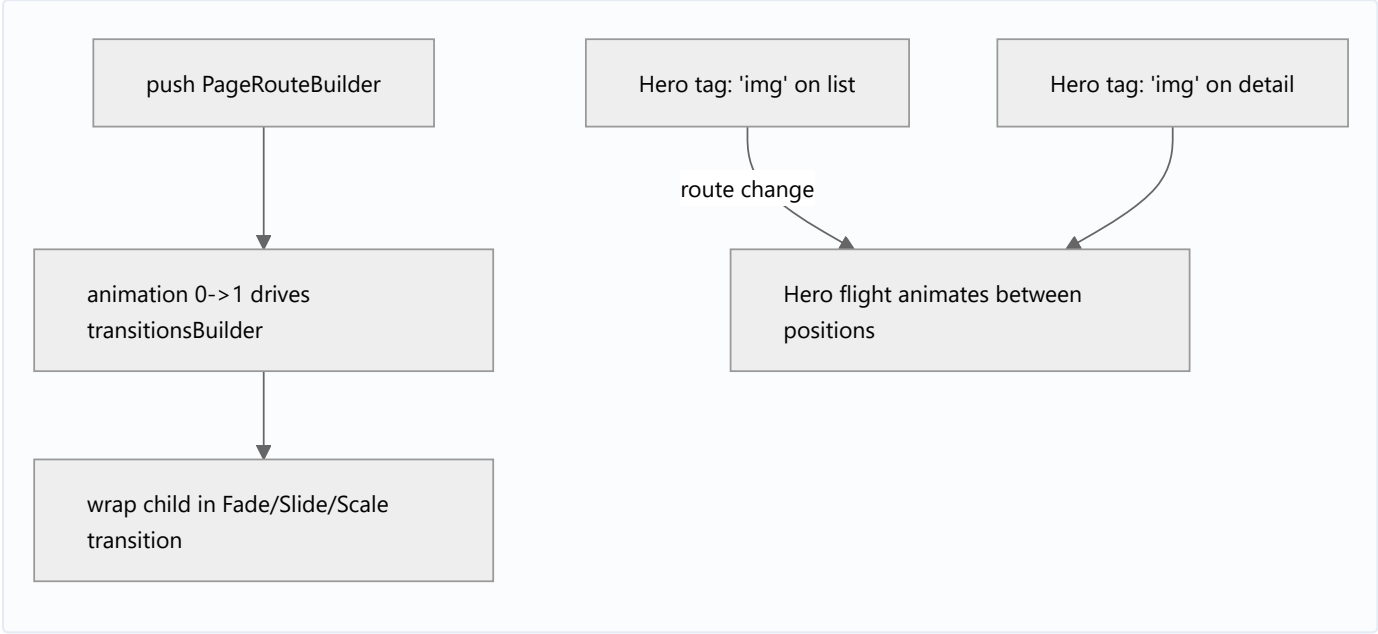
Real-world analogy

A **stage scene change**: default is a standard curtain (Material transition); a custom transition is choosing how the set slides/fades in; `Hero` is a **prop carried by an actor** from one scene to the next so the audience tracks it across the cut.

Problem Statement

You want a fade+slide transition to a detail screen, and the list thumbnail to smoothly expand into the detail's header image. You'll use `PageRouteBuilder` and `Hero`.

Internal Working



- `PageRouteBuilder`: replaces `MaterialPageRoute`; you provide `pageBuilder` (the screen) and `transitionsBuilder(context, animation, secondaryAnimation, child)` that wraps `child` in transition widgets (`FadeTransition`, `SlideTransition`, `ScaleTransition`) driven by `animation` (0→1 on push).
- `transitionDuration` / `reverseTransitionDuration`: control timing.
- `Hero`: wrap the source and destination widgets with the **same** `tag`; on route change, Flutter runs a "flight" animating the widget from its source rect to its destination rect (via an overlay).
- `secondaryAnimation`: lets the outgoing route react as a new one covers it (e.g., fade out).
- Built on the animation framework (`Animation`, `CurvedAnimation`, `Tween`) — Module 22.

Memory Representation

Transitions/Hero flights are transient animation objects; the Hero temporarily moves the widget into an overlay during the flight (09 · compositing).

Compiler Behavior

Not applicable.

Runtime Behavior

On push, `animation` runs 0→1 driving the transition; on pop it reverses. Hero matches tags between the outgoing and incoming routes and animates the flight; mismatched/duplicate tags cause errors/no animation.

Flutter Engine Behavior

Transitions are per-frame repaints; wrap expensive content in `RepaintBoundary` if needed (09 · compositing). Hero flights composite over an overlay layer.

Dart VM Behavior

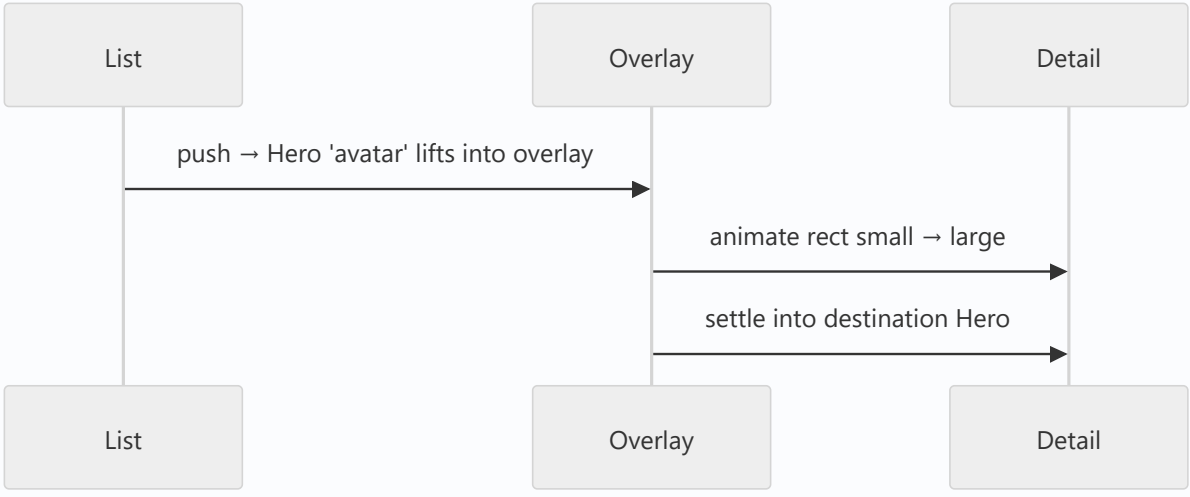
Not applicable.

Examples

```
import 'package:flutter/material.dart';


// Custom fade + slide transition
Route _fadeSlideRoute(Widget page) => PageRouteBuilder(
      transitionDuration: const Duration(milliseconds: 350),
      pageBuilder: (_, __, ___) => page,
      transitionsBuilder: (_, animation, __, child) {
        final curved = CurvedAnimation(parent: animation, curve: Curves.easeOut);
        return FadeTransition(
          opacity: curved,
          child: SlideTransition(
            position: Tween(begin: const Offset(0, 0.1), end: Offset.zero).animate(curved),
            child: child,
          ),
        );
      },
    );


class ListScreen extends StatelessWidget {
  const ListScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('List')),
      body: GestureDetector(
        onTap: () => Navigator.of(context).push(_fadeSlideRoute(const DetailScreen())),
        child: const Padding(
          padding: EdgeInsets.all(16),
          // Shared element: same tag on both screens
```

```
          child: Hero(tag: 'avatar', child: CircleAvatar(radius: 30, child: Icon(Icons.person))),
        ),
      ),
    );
  }
}

class DetailScreen extends StatelessWidget {
  const DetailScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Detail')),
      body: const Center(
        // Same Hero tag -> the avatar "flies" from list to here, growing
        child: Hero(tag: 'avatar', child: CircleAvatar(radius: 80, child: Icon(Icons.person, size: 64))),
      ),
    );
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Duplicate `Hero` tags on one screen	Ambiguous flight → error	Unique tags per screen (use ids)
Mismatched tags across screens	No Hero animation	Same tag on source + destination
Heavy widget in transition without boundary	Jank	`RepaintBoundary` / simpler content
Overlong/overwrought transitions	Feels sluggish	Keep durations ~200–350ms; standard curves
Hero on differently-shaped widgets	Visual jump	Match shapes or use `flightShuttleBuilder`

Best Practices

- Keep transitions **short and standard** (~200–350ms, `Curves.easeInOut` / `easeOut`); match brand without slowing UX.
- Use **unique, stable** `Hero` **tags** (e.g., item id) per screen; identical between source/destination.
- For complex Hero morphs, customize with `flightShuttleBuilder`.
- Wrap heavy transitioning content in `RepaintBoundary` (09).
- Prefer built-in transitions unless a custom one adds real value.

Performance

Transitions/Hero repaint per frame; isolate expensive subtrees with `RepaintBoundary`, keep transitions simple, and watch raster cost for image Heroes (Module 21).

Advantages / Disadvantages

- **+** Polished, brand-consistent motion; `Hero` gives spatial continuity; built on a robust animation system.
- **−** Overdone transitions hurt UX; Hero tag pitfalls; per-frame cost for heavy content.

Interview Questions

1. 🟢 **How do you make a custom route transition?** — Use `PageRouteBuilder` with a `transitionsBuilder` that wraps the child in transition widgets driven by the route's `animation`.
2. 🟢 **What is a** `Hero` **animation?** — A shared-element transition: widgets with the same `tag` on two screens animate ("fly") between their positions across the route change.
3. 🟡 **What drives a route transition?** — The route's `animation` (0→1 on push, reversed on pop), typically curved and mapped via `Tween`s.
4. 🟡 **What causes Hero to not animate or to error?** — Mismatched tags (no animation) or duplicate tags on one screen (ambiguous → error).

5. 🟡 **What is `secondaryAnimation` for?** — Letting the current route react as a new route covers it (e.g., fade/scale out).
6. 🔴 **How does Hero perform the flight?** — It lifts the widget into an `Overlay` and animates its rect from source to destination, then settles into the destination Hero.
7. 🔴 **How do you optimize heavy transitions?** — Simplify content, keep durations short, and wrap expensive subtrees in `RepaintBoundary`; watch raster cost for images.

Senior Engineer Tips

- Use item ids as Hero tags in lists to guarantee uniqueness and correct matching.
- Default to built-in transitions; add custom ones only where they improve comprehension/brand.
- For image Heroes, ensure both ends use similarly-sized/decoded images to avoid flicker and raster spikes (07 · images).

Architect Perspective

Transitions and Hero animations are UX polish grounded in the animation and rendering systems (Modules 22, 09). Standardizing a small set of transitions and a Hero-tagging convention across the app yields consistent, professional motion without ad-hoc, janky one-offs — part of a coherent design system.

Summary

- Custom transitions: `PageRouteBuilder` + `transitionsBuilder` driven by `animation`.
- `Hero` animates shared elements between screens via matching `tag`s (overlay flight).
- Keep motion short/standard; unique stable tags; isolate heavy content with `RepaintBoundary`.

Revision Notes

- `PageRouteBuilder(pageBuilder, transitionsBuilder(anim))` → wrap child in Fade/Slide/Scale.
- `Hero(tag: sameOnBoth)` → shared-element flight via Overlay; unique per screen, matching across.
- `secondaryAnimation` = outgoing route reaction; `flightShuttleBuilder` for custom morphs.
- ~200–350ms; `RepaintBoundary` for heavy content; built on Module 22 animations.

Practice Questions

1. What object drives a custom route transition?
2. Why do Hero tags need to match across screens but be unique within one?
3. How does Hero physically animate the widget?

Coding Questions

1. Build a fade+scale `PageRouteBuilder` transition.
2. Add a `Hero` thumbnail→detail image animation with id-based tags.
3. Customize a Hero flight with `flightShuttleBuilder`.

Mini Project

Gallery Hero flow (Flutter): Build a grid of thumbnails that push a detail screen via a custom fade+slide transition, with each thumbnail Hero-animating (id-based tag) into the detail's large image. Isolate the transition content with `RepaintBoundary`. Acceptance: smooth custom transition; correct Hero flights (unique/matching tags); no jank on images; app runs.