13 · Routing

Flutter Practice Notes

Contents

13 · Routing

Introduction

Routing is production-grade navigation: URL-driven, deep-link-capable, guarded, and nested. This module focuses on `go_router` (the officially-recommended package built on Navigator 2.0) — declarative routes, path/query params, deep linking, redirects/guards, shell routes for tabs, and type-safe routes.

Why this module exists

Module 12 covered the mechanics; real apps need URL sync (web), deep links, auth redirects, and tabbed shells that survive links. `go_router` provides these ergonomically. Getting routing right is critical for UX (shareable links, back behavior), web SEO, and maintainability.

Module map

#	File	Topic	Level
1	01_go_router_fundamentals.md	Setup, routes, path/query params, `go` / `push`	🔵
2	02_deep_linking_and_url_strategy.md	Deep links (mobile/web), URL strategy	🔴
3	03_guards_and_redirects.md	`redirect`, auth guards, `refreshListenable`	🔴
4	04_shell_routes_and_nested.md	`ShellRoute` / `StatefulShellRoute` (tabs + deep links)	🔴
5	05_type_safe_and_error_handling.md	Typed routes (codegen), `errorBuilder`, `extra`	🔵

Cross-references: Navigator/stack + declarative basics: Module 12. Auth: Module 17. State (auth listenable): Module 11. Web URL/SEO: Module 53. App entry/ `MaterialApp.router`: 06 · app_entry_point.

Prerequisites

12 Navigation — especially declarative navigation and nested navigators/tabs.

What you'll be able to do after this module

- Configure `go_router` with path/query params and navigate declaratively.
- Handle deep links on mobile and web with the right URL strategy.
- Implement auth guards/redirects that react to auth-state changes.
- Build tabbed shells (`StatefulShellRoute`) that preserve state and support deep links.

- Use type-safe routes and handle route errors.

Capstone

Routed app with auth + tabs: A `go_router` app with a login guard (redirect), deep-linkable detail pages (`/product/:id`), and a bottom-nav `StatefulShellRoute` where each tab keeps its stack and is deep-linkable — the routing backbone of a real app.

Summary

`go_router` turns Navigator 2.0 into a declarative, URL-first router with params, deep links, guards, and shells. It's the production default; master it and your navigation is shareable, guarded, testable, and web-ready.

`go_router` Fundamentals

> `go_router` is a declarative, URL-based router (built on Navigator 2.0) where you define routes by path with builders, navigate with `context.go` / `push`, and read `:params` and `?query` from the matched location.

Introduction

`go_router` (package: `go_router`) lets you declare a route table by **path** (`/product/:id`), wire it with `MaterialApp.router`, and navigate declaratively. It handles the Navigator 2.0 boilerplate (delegate/parser) for you. This file covers setup, params, and the navigation API.

Why this concept exists

Raw Navigator 2.0 is powerful but verbose (12 · declarative_navigation); imperative named routes don't sync URLs or handle deep links well. `go_router` gives a concise, URL-first API that's the officially-recommended production choice.

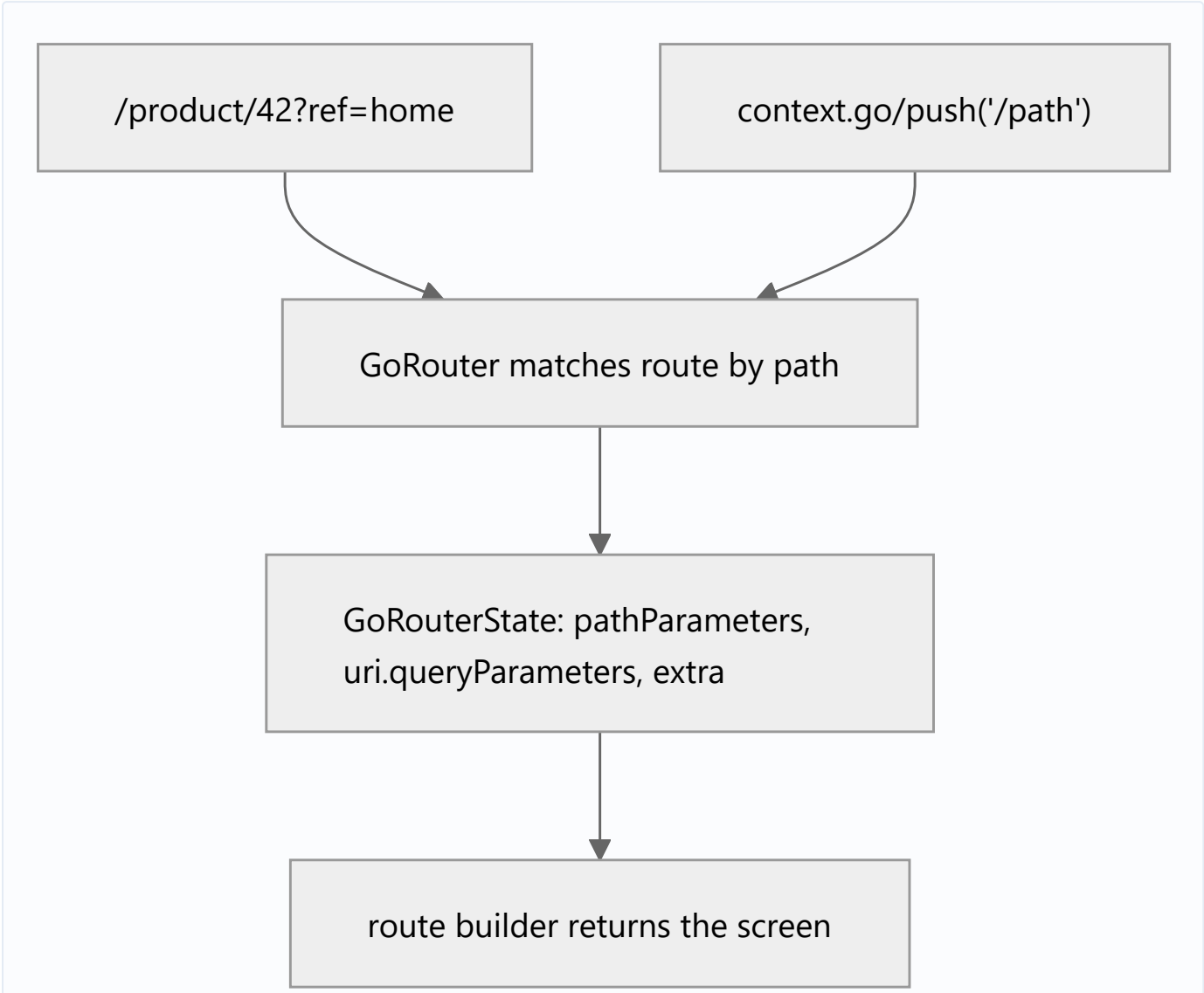
Real-world analogy

`go_router` is a **street-address system** for your app: every screen has an address (`/product/42`), you navigate by address (`go('/product/42')`), and anyone can share/bookmark an address to land exactly there (deep link).

Problem Statement

You need `/`, `/products`, and `/product/:id?ref=...`, navigable by URL and deep-linkable. You'll configure `GoRouter`, read the `id` path param and `ref` query param, and navigate with `go` / `push`.

Internal Working



- **Config**: `GoRouter(routes: [GoRoute(path: '/', builder: ...), GoRoute(path: '/product/:id', builder: ...)])` ; attach via `MaterialApp.router(routerConfig: router)` .
- **Params**: `:id` is a **path parameter** (`state.pathParameters['id']`); `?ref=` is a **query parameter** (`state.uri.queryParameters['ref']`).
- **Navigation**:
  - `context.go('/path')` — set the location (replaces the stack per the route tree; URL-style).
  - `context.push('/path')` — push on top (adds to stack, returns a result future).
  - `context.goNamed('name', pathParameters: {...}, queryParameters: {...})` — by route name.
  - `context.pop()` — pop.
- **Sub-routes**: nest `routes:` inside a `GoRoute` to build hierarchical paths/stacks.
- `extra` : pass a non-URL object alongside navigation (not shareable/deep-linkable — for in-app objects).

Memory Representation

`go_router` manages the Navigator 2.0 delegate/pages; route state is derived from the current location (12 · declarative_navigation).

Compiler Behavior

Path strings are untyped by default (typos = runtime 404) → use **type-safe routes** for compile safety (05_type_safe_and_error_handling.md).

Runtime Behavior

Navigating changes the location; `go_router` matches it to a route (or the error builder), builds the page stack, and syncs the URL (on web). `go` vs `push` differ in stack behavior.

Flutter Engine Behavior / Dart VM Behavior

Not applicable beyond rendering.

Examples

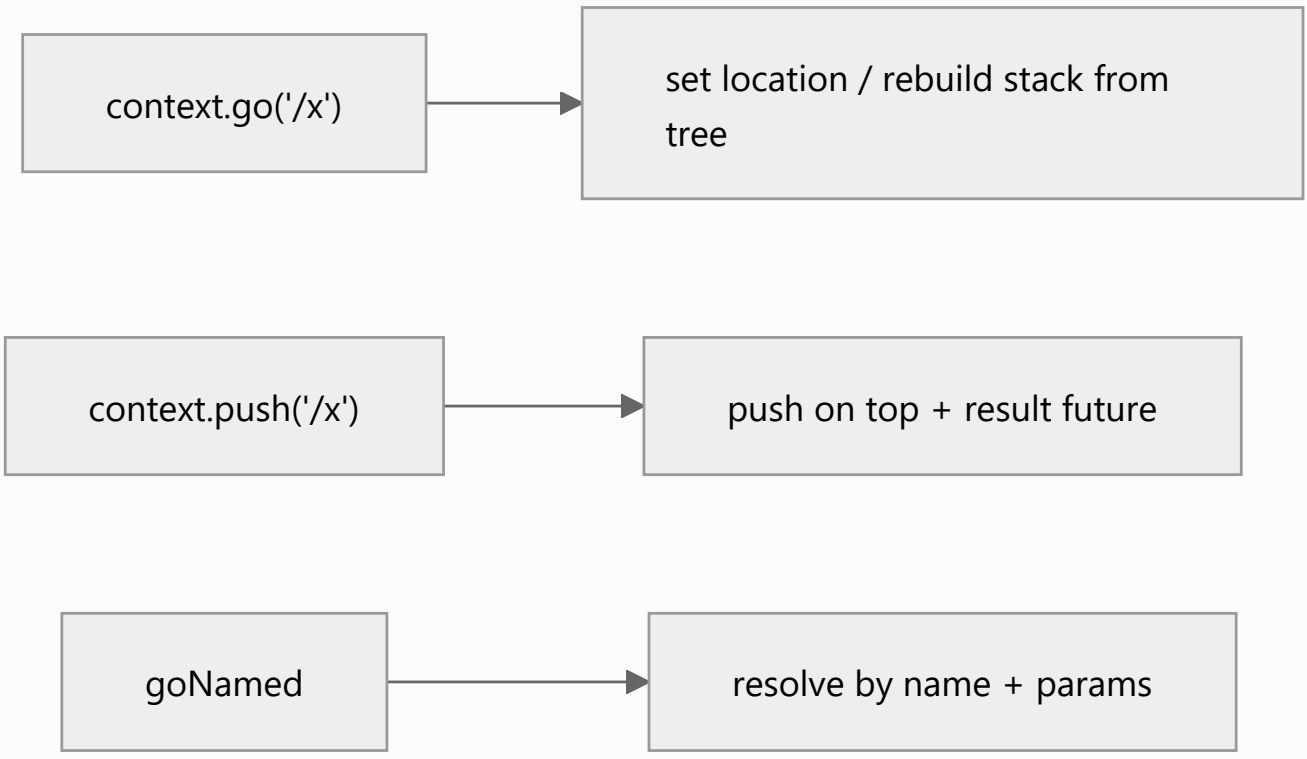
```yaml
# pubspec.yaml
dependencies:
  go_router: ^14.0.0
```

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

final router = GoRouter(
  initialLocation: '/',
  routes: [
    GoRoute(
      path: '/',
      name: 'home',
      builder: (context, state) => const HomeScreen(),
      routes: [ // sub-route -> /products
        GoRoute(
          path: 'products',
          name: 'products',
          builder: (context, state) => const ProductsScreen(),
        ),
```

```
      ],
    ),
    GoRoute(
      path: '/product/:id', // path param
      name: 'product',
      builder: (context, state) {
        final id = state.pathParameters['id']!;            // path param
        final ref = state.uri.queryParameters['ref'];      // query param
        return ProductScreen(id: id, ref: ref);
      },
    ),
  ],
);

void main() => runApp(MaterialApp.router(routerConfig: router));

class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});
  @override
  Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(title: const Text('Home')),
        body: Center(
          child: Column(mainAxisSize: MainAxisSize.min, children: [
            ElevatedButton(
              onPressed: () => context.go('/products'),
              child: const Text('Go to products'),
            ),
            ElevatedButton(
              // navigate by name with params + query
              onPressed: () => context.pushNamed('product',
                  pathParameters: {'id': '42'}, queryParameters: {'ref': 'home'}),
              child: const Text('Open product 42'),
            ),
          ]),
        ),
      );
}
class ProductsScreen extends StatelessWidget {
  const ProductsScreen({super.key});
  @override
```

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

Diagrams



Common Mistakes

Mistake	Why	Fix
Confusing `go` vs `push`	Different stack behavior	`go` = set location; `push` = add on top
Untyped path typos	Runtime 404	Use type-safe routes (05_type_safe_and_error_handling.md)
Passing objects via query params	URLs can't hold objects	Use `extra` (non-shareable) or fetch by id
Forgetting `MaterialApp.router`	Router not wired	Use `.router(routerConfig:)`
Reading query from `pathParameters`	Wrong map	Use `state.uri.queryParameters`

Best Practices

- Design **clean, hierarchical paths** (`/product/:id`); use sub-routes for hierarchy.
- Use **path params** for identity, **query params** for options/filters, `extra` only for in-app objects (not deep-linkable).
- Prefer **named routes** + params for refactor-safety; adopt **type-safe routes** for compile safety.
- Wire via `MaterialApp.router(routerConfig:)` ; keep the router config centralized.
- Understand `go` (URL-set) vs `push` (stack-add) and pick intentionally.

Performance

Comparable to Navigator; route matching is cheap. Deep-link restoration builds the matched stack once (12 · navigator_stack).

Advantages / Disadvantages

- **+** Declarative URL-based routing, deep links, params, sub-routes, guards/shells, official recommendation, web-ready.
- **–** Untyped paths by default (typo-prone), `go` / `push` nuance, learning curve vs simple `Navigator.push` .

Interview Questions

1. 🟢 **What is `go_router` ?** — A declarative, URL-based routing package built on Navigator 2.0; you define routes by path and navigate with `context.go` / `push` .
2. 🟢 **Path param vs query param?** — Path (`:id`) identifies the resource (`state.pathParameters`); query (`?ref=`) carries options (`state.uri.queryParameters`).

3. 🟡 `go` **vs** `push` **?** — `go` sets the location (rebuilds the stack from the route tree; URL-style); `push` adds a route on top (and returns a result future).
4. 🟡 **How do you pass an in-app object that shouldn't be in the URL?** — Via `extra` (not shareable/deep-linkable); for shareable, use an id param and fetch.
5. 🟡 **How is the router attached?** — `MaterialApp.router(routerConfig: goRouter)`.
6. 🔴 **Why prefer type-safe routes?** — Untyped path strings cause runtime 404 on typos; codegen'd typed routes catch errors at compile time (05_type_safe_and_error_handling.md).
7. 🔴 **How do sub-routes work?** — Nesting `routes:` inside a `GoRoute` composes paths/stacks hierarchically (`/` → `/products`).

Senior Engineer Tips

- Centralize the router config; name routes and navigate by name/typed routes to survive path refactors.
- Reserve `extra` for genuinely non-URL data; anything shareable/deep-linkable must be expressible in the path/query.
- Be deliberate about `go` vs `push`; misuse causes surprising back-stack behavior.

Architect Perspective

`go_router` makes navigation a declarative projection of URLs/state — the routing backbone for shareable, deep-linkable, web-capable apps. Centralized, typed, guarded route config is a maintainability and UX asset that scales across teams and platforms (Modules 17, 53).

Summary

- `go_router`: declarative URL routes with `:path` / `?query` params, navigated via `go` / `push` / `goNamed`.
- Path params = identity, query = options, `extra` = in-app objects; wire with `MaterialApp.router`.
- Prefer named/type-safe routes; understand `go` vs `push`.

Revision Notes

- `GoRouter(routes: [GoRoute(path, builder, routes:)])` + `MaterialApp.router(routerConfig:)`.
- Params: `state.pathParameters` (`:id`), `state.uri.queryParameters` (`?q=`), `state.extra` (in-app obj).
- Navigate: `context.go` (set location) / `push` (add + result) / `goNamed` / `pushNamed`.
- Untyped paths → runtime 404; prefer type-safe routes.

Practice Questions

1. When use `go` vs `push`?

2. Where do you read `:id` vs `?ref=`?
3. Why not pass an object via query params?

Coding Questions

1. Configure `/`, `/products`, `/product/:id?ref=` and navigate to each.
2. Add a sub-route and navigate by name with params.
3. Pass an in-app object via `extra` and read it.

Mini Project

Product routing (Flutter): Build a `go_router` app with home, products list, and `/product/:id?ref=` (path + query), navigable by name with params, plus one sub-route. Verify a pasted URL opens the right screen. Acceptance: params read correctly; `go` / `push` used intentionally; URL-addressable; app runs.

Deep Linking & URL Strategy

> Deep links let an external URL (or app link/universal link) open a specific screen with its full state; `go_router` maps the link's path to a route and restores the stack — but mobile requires platform config, and web requires choosing a URL strategy (path vs hash).

Introduction

A deep link is a URL that opens your app at a specific location (`myapp://product/42` or `https://app.com/product/42`). This file covers how `go_router` handles them, the mobile platform setup (App Links / Universal Links / custom schemes), and the **web URL strategy** (path vs hash) plus SEO implications.

Why this concept exists

Users share links, click notifications, and (on web) bookmark/refresh URLs. The app must land them on the exact screen with correct state. `go_router`'s path-based routing makes links map to routes; platform/web config makes the OS/browser deliver those links to your app.

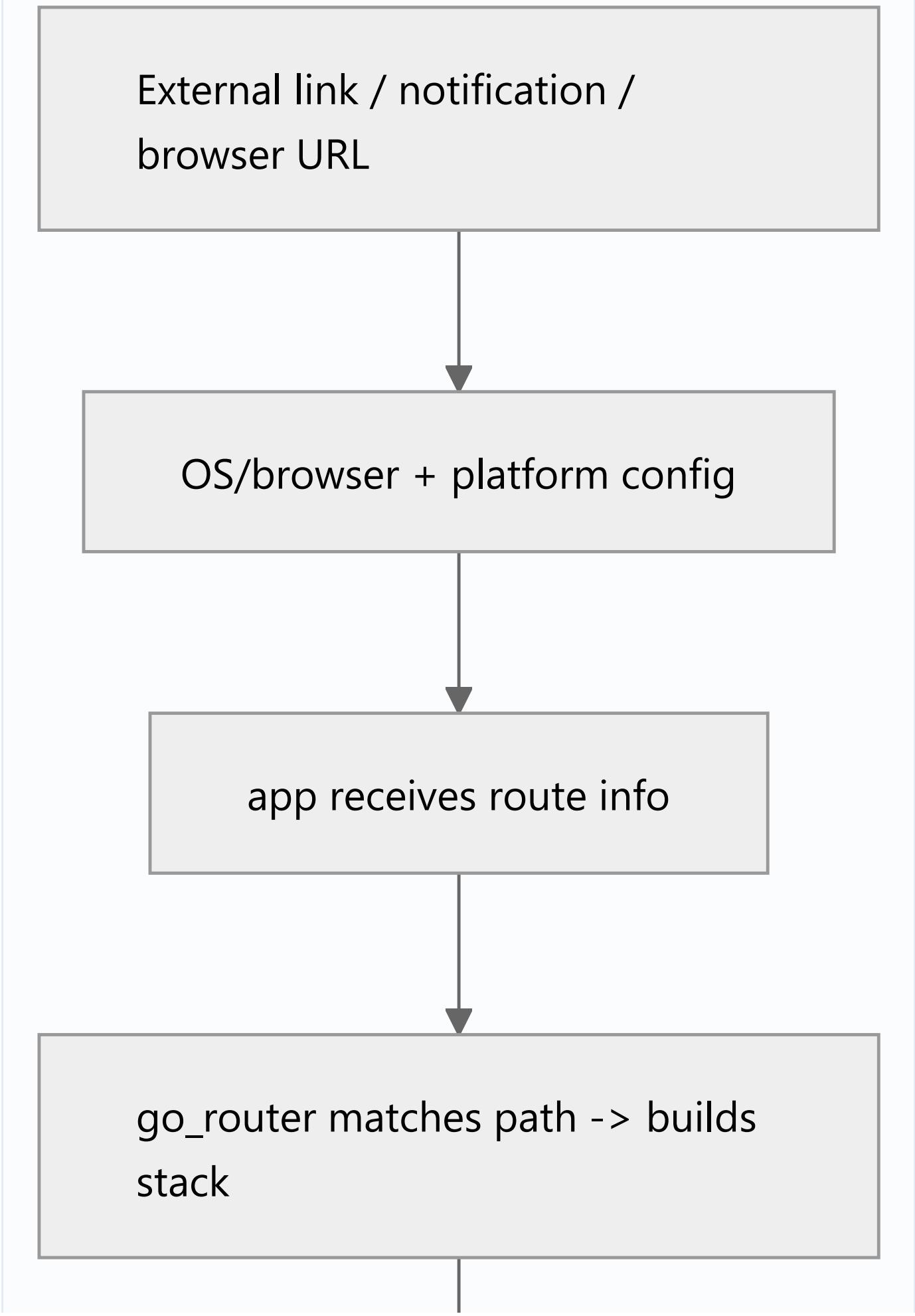
Real-world analogy

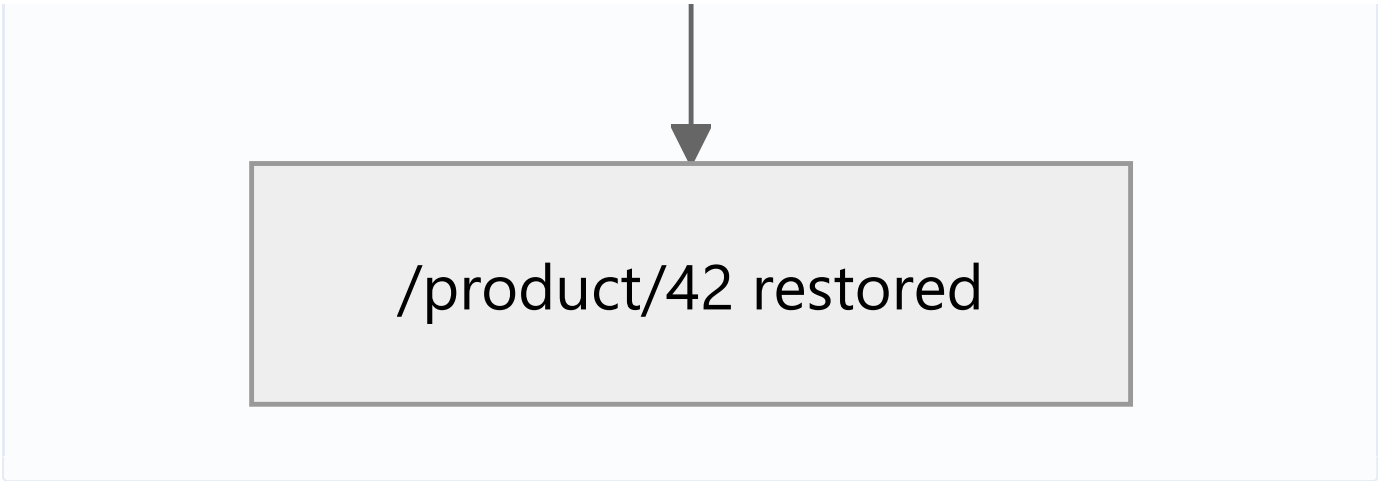
A deep link is a **direct-dial phone extension**: instead of calling the front desk (app home) and being transferred, you dial the exact extension (screen) and reach it immediately. The phone system (OS/browser + platform config) must be set up to route that extension to you.

Problem Statement

Tapping `https://app.com/product/42` (or a push notification) should open the product screen directly; on web, refreshing that URL should work and links should be clean (no `#`). You'll configure deep links and choose a URL strategy.

Internal Working


External link / notification / browser URL
OS/browser + platform config
app receives route info
go_router matches path -> builds stack



- `go_router` **side**: because routes are path-based, an incoming path (`/product/42`) matches a `GoRoute` and `go_router` builds the corresponding (possibly nested) stack — deep-link restoration is automatic if your route tree models the hierarchy.
- **Mobile platform config** (required for OS to deliver links):
  - **Android App Links** (verified `https://` via `assetlinks.json`) and/or **custom scheme** (`myapp://`) in `AndroidManifest.xml` intent filters.
  - **iOS Universal Links** (associated domains + `apple-app-site-association`) and/or custom URL scheme in `Info.plist` .
- **Web URL strategy**:
  - **Path strategy** (clean URLs, `app.com/product/42`) — requires server config to serve `index.html` for all paths; better for SEO/sharing. Enabled via `usePathUrlStrategy()` .
  - **Hash strategy** (default historically, `app.com/#/product/42`) — no server config needed, but ugly and worse for SEO.
- **Notifications**: a push payload carrying a route → on tap, `context.go(path)` (or set initial location) restores the screen (Module 32).

Memory Representation

Deep-link restoration builds the matched stack once; no special memory beyond the resulting routes (12 · navigator_stack).

Compiler Behavior

Not applicable (typed routes still help correctness).

Runtime Behavior

On cold start via a link, the app's initial location is the link's path; on warm start, the OS delivers the new location and `go_router` navigates. Web refresh re-parses the URL → same route.

Flutter Engine Behavior

The embedder delivers platform route info (deep links) to the framework (10 · embedder_and_startup).

Dart VM Behavior

Not applicable.

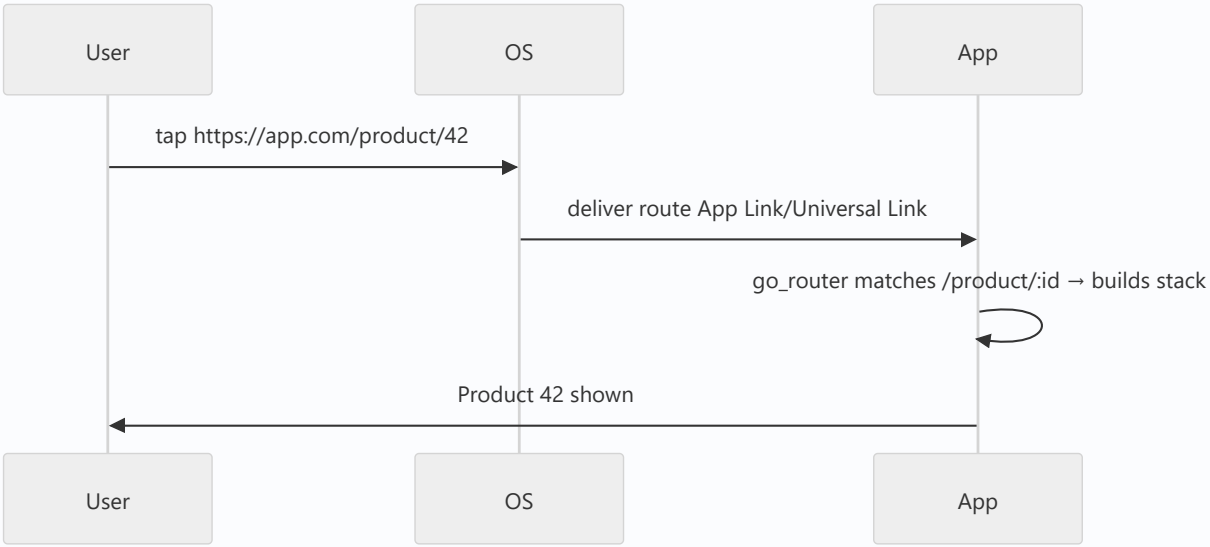
Examples

```
import 'package:flutter/material.dart';
import 'package:flutter_web_plugins/url_strategy.dart'; // for web path strategy
import 'package:go_router/go_router.dart';

void main() {
  usePathUrlStrategy(); // web: clean URLs (app.com/product/42) instead of /#/product/42
  runApp(MaterialApp.router(routerConfig: _router));
}

final _router = GoRouter(
  // Nested route tree so /product/42 restores home -> product stack on deep link:
  routes: [
    GoRoute(
      path: '/',
      builder: (_, __) => const HomeScreen(),
      routes: [
        GoRoute(
          path: 'product/:id',
          builder: (_, state) => ProductScreen(id: state.pathParameters['id']!),
        ),
      ],
    ),
  ],
);

// Handling a notification payload that carries a route:
void onNotificationTap(BuildContext context, String? route) {
  if (route != null) context.go(route); // e.g., '/product/42'
}
```

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

```
Android (AndroidManifest.xml) intent filter for https App Links:
  <intent-filter android:autoVerify="true">
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <data android:scheme="https" android:host="app.com"/>
  </intent-filter>
  + host assetlinks.json for verification.

iOS: add Associated Domains (applinks:app.com) + apple-app-site-association file.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Expecting deep links without platform config	OS won't deliver links	Configure App Links/Universal Links/schemes
Flat routes but expecting stack restoration	No parent to restore	Model hierarchy with sub-routes
Hash URLs on web for SEO/sharing	Ugly, poor SEO	`usePathUrlStrategy()` + server rewrite to `index.html`
Path strategy without server rewrite	404 on refresh/direct link	Configure hosting to serve `index.html` for all paths
Objects in deep-link URLs	Not serializable	Use ids in the path; fetch data

Best Practices

- Model your **route tree hierarchically** so deep links restore the correct (nested) stack.
- Configure platform deep links: **App Links (Android)** + **Universal Links (iOS)**; add custom schemes if needed.
- On web, prefer **path URL strategy** (clean URLs, SEO) and configure hosting to serve `index.html` for all routes.
- Encode only **ids/params** in URLs; fetch data by id on arrival.
- Route notification taps through `go_router` (`context.go(path)`).

Performance

Deep-link restoration is a one-time stack build; no ongoing cost. Path strategy has SEO/shareability benefits at the cost of server config (Module 53).

Advantages / Disadvantages

- **+** Direct-to-screen links, shareable/bookmarkable URLs, notification routing, web refresh works, SEO (path strategy).
- **−** Platform setup required (Android/iOS), web needs server rewrite for path strategy, careful route-tree modeling.

Interview Questions

1. 🟢 **What is a deep link?** — A URL that opens the app at a specific screen with its state (e.g., `https://app.com/product/42`).

2. 🟢 **How does `go_router` support deep links?** — Its path-based routes match incoming URLs and build the corresponding (nested) stack automatically.
3. 🟡 **What platform setup is required?** — Android App Links (verified https + `assetlinks.json`) / custom scheme; iOS Universal Links (associated domains + AASA) / URL scheme.
4. 🟡 **Path vs hash URL strategy on web?** — Path (`/product/42`, clean, SEO-friendly, needs server rewrite) vs hash (`/#/product/42`, no server config, ugly/poor SEO).
5. 🟡 **Why must route trees be hierarchical for deep links?** — So restoring a deep path recreates its parent stack (home→product), enabling correct back behavior.
6. 🔴 **Why does path strategy 404 on refresh without server config?** — The server must serve `index.html` for all paths; otherwise it looks for a real file at that path. Configure hosting rewrites.
7. 🔴 **How do you route a notification tap to a screen?** — Carry the route/path in the payload and call `context.go(path)` (or set initial location) on tap.

Senior Engineer Tips

- Test deep links in all three states: cold start, warm/background, and web refresh — behavior differs.
- Keep URLs id-based and canonical; avoid embedding transient/objects.
- Set up path strategy + hosting rewrites early for web; retrofitting hash→path breaks existing links.

Architect Perspective

Deep linking + URL strategy are foundational for shareability, growth (marketing links), notifications, and web/SEO. A hierarchical, id-based route tree with platform + hosting config makes the whole app addressable and link-restorable — a cross-cutting concern spanning routing, notifications, and web (Modules 32, 53, 17).

Summary

- Deep links map external URLs to routes; `go_router` restores the (nested) stack from the path.
- Configure mobile App/Universal Links; on web choose path strategy (+ server rewrite) for clean, SEO-friendly URLs.
- Use hierarchical, id-based routes; route notification taps through the router.

Revision Notes

- Deep link = URL → specific screen+state; `go_router` matches path → builds stack (model hierarchy!).

- Mobile: App Links (Android) / Universal Links (iOS) + optional custom schemes.
- Web: `usePathUrlStrategy()` (clean/SEO, needs `index.html` rewrite) vs hash (default, no config).
- URLs carry ids/params, not objects; route notifications via `context.go`.

Practice Questions

1. What must you configure for OS to deliver deep links?
2. Path vs hash strategy tradeoffs?
3. Why must routes be hierarchical for correct deep-link back behavior?

Coding Questions

1. Configure a nested route so `/product/42` restores home→product.
2. Enable path URL strategy and note the hosting requirement.
3. Route a fake notification payload to a screen via `context.go`.

Mini Project

Deep-linkable app (Flutter + docs): Build a `go_router` app with a nested `/product/:id` route, enable path URL strategy, and document the Android/iOS deep-link config + web hosting rewrite. Verify a pasted/deep URL restores the correct stack. Acceptance: deep link opens the right nested screen; clean web URLs; config documented; app runs.

Route Guards & Redirects

> `go_router`'s `redirect` is a function that inspects the target location + app state and optionally returns a different location — the mechanism for auth guards, onboarding gates, and conditional routing that also react to state changes via `refreshListenable`.

Introduction

Guards protect routes: unauthenticated users go to `/login`, unonboarded users to `/welcome`, etc. `go_router` implements this with **`redirect`** (global and/or per-route) plus **`refreshListenable`** so the router re-evaluates when auth/app state changes. This file covers safe, loop-free guard design.

Why this concept exists

Apps must enforce access rules centrally and consistently — not sprinkle `if (loggedIn)` checks in every screen. A redirect function centralizes routing policy and reacts to auth changes, so login/logout instantly moves the user to the right place, including for deep links.

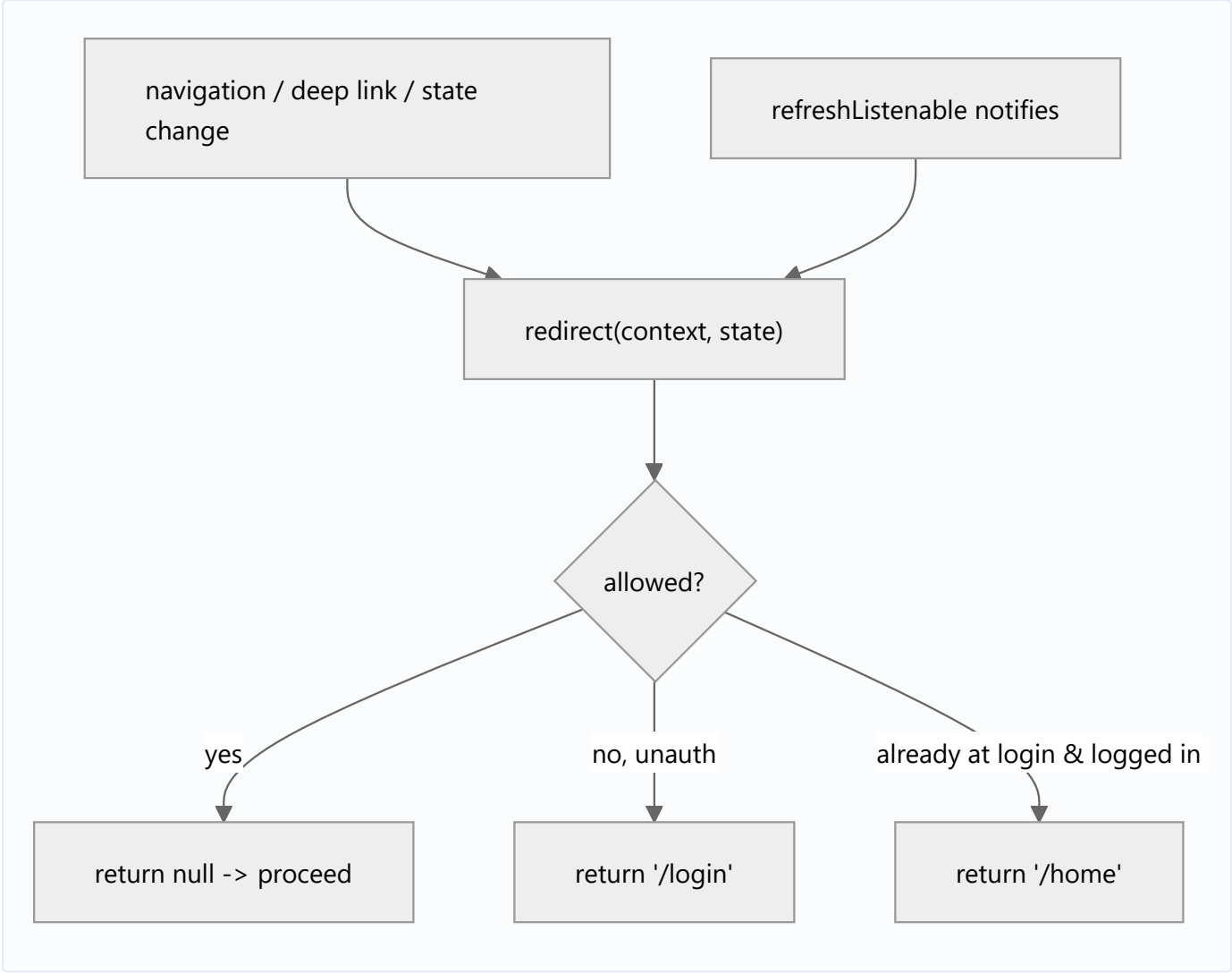
Real-world analogy

A **security checkpoint** at a building: before you reach any floor (route), the guard checks your badge (auth state) and either lets you through or redirects you to reception (`/login`). When your access changes (badge activated/revoked), the checkpoint re-evaluates.

Problem Statement

Unauthenticated users hitting any protected route (even via deep link) must be sent to `/login`; after login they proceed to their original target; logout kicks them back to `/login`. You'll use `redirect` + `refreshListenable` without redirect loops.

Internal Working



- `redirect` : `String? Function(BuildContext, GoRouterState)` on `GoRouter` (global) or a `GoRoute` (local). Return `null` to proceed, or a location string to redirect. Runs on every navigation **and** whenever `refreshListenable` notifies.
- `refreshListenable` : a `Listenable` (e.g., your auth `ChangeNotifier`/a stream adapter) that tells the router to re-run redirects when auth/app state changes — so login/logout re-routes immediately.
- **Loop safety**: guard against infinite redirects — allow the destination you redirect *to* (e.g., don't redirect `/login` to `/login`); compare current location.
- **Preserve intended destination**: capture `state.uri` (the attempted location) so post-login you can `go` back to it (via a `from` / `redirect` query or stored value).
- **Per-route vs global**: global for app-wide auth; per-route `redirect` for route-specific rules.

Memory Representation

The `refreshListenable` (auth notifier) lives in your state layer (Module 11); redirects are pure functions of state + target.

Compiler Behavior

Not applicable (typed routes help correctness).

Runtime Behavior

Every navigation triggers `redirect`; a returned location causes an immediate re-navigation (which re-runs redirect — hence loop guards). `refreshListenable` notifications re-evaluate the current location.

Flutter Engine Behavior / Dart VM Behavior

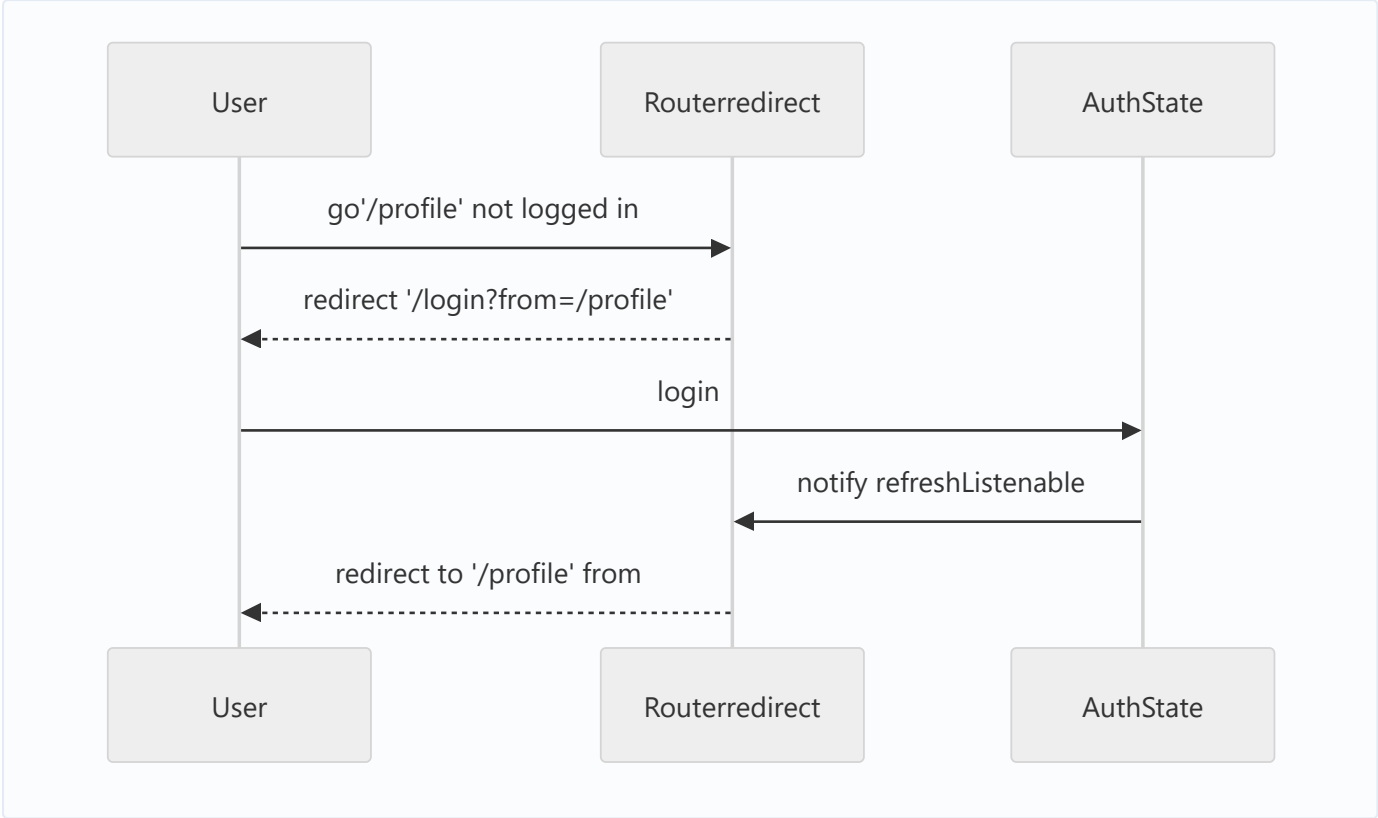
Not applicable.

Examples

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

// Auth state as a Listenable (drives router refresh)
class AuthState extends ChangeNotifier {
  bool loggedIn = false;
  void login() { loggedIn = true; notifyListeners(); }   // triggers redirect re-eval
  void logout() { loggedIn = false; notifyListeners(); }
}
final auth = AuthState();

final router = GoRouter(
  refreshListenable: auth, // re-run redirects when auth changes
  redirect: (context, state) {
    final loggingIn = state.matchedLocation == '/login';
    if (!auth.loggedIn) {
      // not logged in -> go to login, remembering where we wanted to go
      return loggingIn ? null : '/login?from=${state.uri}';
    }
    if (loggingIn) {
      // logged in but on login -> send onward (to 'from' or home)
      final from = state.uri.queryParameters['from'];
      return (from != null && from.isNotEmpty) ? from : '/home';
    }
    return null; // allow
```

```
  },
  routes: [
    GoRoute(path: '/login', builder: (_, __) => const LoginScreen()),
    GoRoute(path: '/home', builder: (_, __) => const HomeScreen()),
    GoRoute(path: '/profile', builder: (_, __) => const ProfileScreen()), // protected
  ],
);

void main() => runApp(MaterialApp.router(routerConfig: router));

class LoginScreen extends StatelessWidget {
  const LoginScreen({super.key});
  @override
  Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(title: const Text('Login')),
        body: Center(
          child: ElevatedButton(onPressed: auth.login, child: const Text('Log in')),
        ),
      );
}
class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});
  @override
  Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(title: const Text('Home'), actions: [
          IconButton(onPressed: auth.logout, icon: const Icon(Icons.logout)),
        ]),
      );
}
class ProfileScreen extends StatelessWidget {
  const ProfileScreen({super.key});
  @override
  Widget build(BuildContext context) => Scaffold(appBar: AppBar(title: const Text('Profile')));
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Redirect loop	Redirecting the destination to itself	Allow the target (`if at /login return null`)
No `refreshListenable`	Router won't react to login/logout	Provide an auth `Listenable`
Per-screen `if (loggedIn)` checks	Scattered, inconsistent, deep-link holes	Centralize in `redirect`
Losing the intended destination	Post-login lands on home, not target	Carry `from` / `redirect` query param
Heavy work in `redirect`	Runs on every navigation	Keep it a fast pure check

Best Practices

- Centralize access policy in `redirect`; keep screens free of auth branching.
- Provide a `refreshListenable` (auth notifier / stream adapter) so login/logout re-routes instantly.
- **Guard against loops**: always allow the location you redirect to; compare `matchedLocation`.
- **Preserve intended destination** (`?from=`) for post-login continuation — including deep links.
- Keep `redirect` a fast, pure function of state + target.

Performance

`redirect` runs per navigation and on every `refreshListenable` notification — keep it O(1)/pure (no I/O). Negligible if disciplined (09 · build_phase).

Advantages / Disadvantages

- **+** Centralized, consistent, deep-link-safe access control that reacts to state; supports post-login continuation.
- **−** Loop pitfalls; requires an auth `Listenable`; redirect logic can grow complex (keep it structured).

Interview Questions

1. 🟢 **How do you guard routes in `go_router`?** — With `redirect`: inspect target + app state, return `null` to allow or a location to redirect (e.g., to `/login`).
2. 🟢 **What is `refreshListenable`?** — A `Listenable` (e.g., auth `ChangeNotifier`) that makes the router re-run `redirect` when it notifies — so login/logout re-routes immediately.
3. 🟡 **How do you avoid redirect loops?** — Allow the destination you redirect to (return `null` when already at `/login`); compare the current/matched location.
4. 🟡 **How do you send the user back to their intended page after login?** — Capture the attempted location (`state.uri`) in a `from`/`redirect` query and `go` there after auth.
5. 🟡 **Why centralize guards in `redirect` vs per-screen checks?** — Consistency, no deep-link bypass, single source of routing policy, easier to maintain/test.
6. 🔴 **When does `redirect` run?** — On every navigation and whenever `refreshListenable` notifies (re-evaluating the current location).
7. 🔴 **Why keep `redirect` pure/fast?** — It runs frequently; async/heavy work would stall navigation — do I/O elsewhere and expose results as state.

Senior Engineer Tips

- Model auth as a `Listenable`/stream in your state layer; the router just reads it in `redirect` (Module 11, Module 17).
- Structure complex guard logic as small helpers (isProtected, needsOnboarding) for readability/testability.
- Test guards for cold-start deep links (unauth → login → back to target) — a common gap.

Architect Perspective

Redirect-based guards centralize routing policy, making auth/onboarding rules consistent and deep-link-safe — a key security/UX concern. Coupling the router to auth state via `refreshListenable` yields reactive, testable access control that integrates cleanly with the auth module (Module 17) and state layer.

Summary

- `redirect` centralizes access control (return `null` or a location); `refreshListenable` re-evaluates on auth/state changes.
- Guard against loops, preserve the intended destination, keep `redirect` pure/fast.
- Centralized guards beat per-screen checks for consistency and deep-link safety.

Revision Notes

- `redirect(context, state)` → `null` (allow) or location (redirect); global or per-route.
- `refreshListenable: auth` → re-run redirects on auth change.
- Loop-safe: allow the redirect target; preserve `?from=` for post-login.
- Keep `redirect` pure/fast; centralize policy (no per-screen auth ifs).

Practice Questions

1. How do you prevent an infinite redirect loop?
2. Why is `refreshListenable` necessary?
3. How do you return the user to a deep link after login?

Coding Questions

1. Add an auth guard redirecting protected routes to `/login`.
2. Wire `refreshListenable` so logout instantly routes to `/login`.
3. Preserve and honor a `from` destination after login.

Mini Project

Auth-guarded router (Flutter): Build a `go_router` app with `/login`, `/home`, and protected `/profile`; guard via `redirect` + an auth `ChangeNotifier` `refreshListenable`; preserve the intended destination for post-login continuation (test with a deep link to `/profile` while logged out). Acceptance: no loops; deep-link continuation works; logout re-routes; app runs.

Shell Routes & Nested Routing (`ShellRoute`, `StatefulShellRoute`)

> `ShellRoute` wraps child routes in a persistent shell (e.g., a scaffold with a bottom nav); `StatefulShellRoute` gives each branch its **own navigator + preserved state**, delivering deep-linkable tabs that keep their stacks — the `go_router` answer to Module 12's nested-navigator pattern.

Introduction

Real apps have a persistent chrome (bottom nav, side rail) around swapping content, where each tab keeps its own stack. Hand-wiring nested navigators (12 · nested_navigators_and_tabs) is fiddly and not deep-linkable. `go_router`'s `StatefulShellRoute` provides this as first-class, URL-addressable routing.

Why this concept exists

Tabs need: persistent shell UI, independent per-tab back stacks, preserved state on switch, **and** deep-linkability (each tab's screens have URLs). `StatefulShellRoute` combines nested navigators + state preservation + URL routing in one construct.

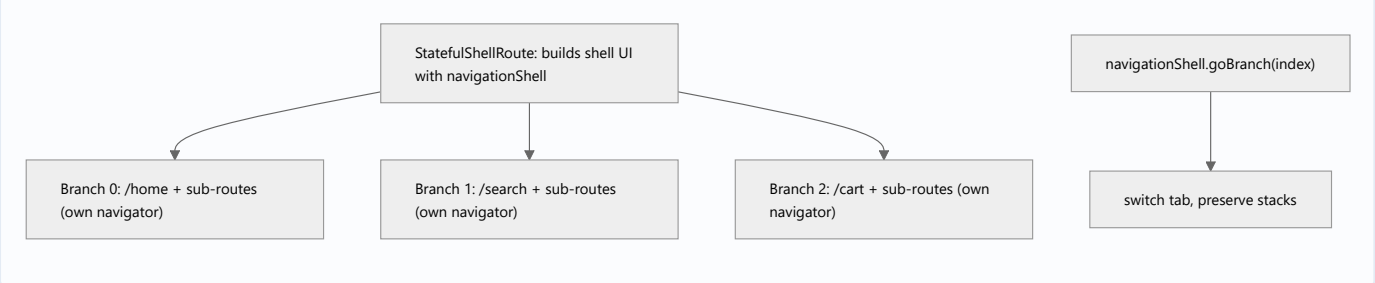
Real-world analogy

A **shopping mall with a fixed directory/escalators (the shell)** and independent **stores (branches)**: each store remembers where you were browsing (per-branch stack/state), the mall frame stays put as you switch stores, and every store/aisle has an address (deep link).

Problem Statement

A bottom-nav app (Home/Search/Cart) where each tab can push details, switching preserves each stack, back pops within the active tab, and `/search/product/42` deep-links into the Search tab. You'll use `StatefulShellRoute.indexedStack`.

Internal Working



- `ShellRoute`: a route with a `builder` that wraps its child routes in shared UI (one navigator for the children). Good for a common frame with a **single** stack.
- `StatefulShellRoute.indexedStack`: multiple **branches**, each a `StatefulShellBranch` with its own routes and **its own** `Navigator` (backed by an `IndexedStack` for state preservation). The shell `builder` receives a `navigationShell` you use to render the tab bar and switch branches (`navigationShell.goBranch(index)`), reading `navigationShell.currentIndex`.
- **Deep links**: because each branch's routes have real paths (`/search/product/:id`), deep links land in the correct branch with its stack restored.
- **Back handling**: managed per-branch by the shell; back pops within the active branch.

Memory Representation

`indexedStack` keeps all branches alive (state/stacks preserved) — memory ∝ number/heft of branches, like Module 12's `IndexedStack` (12 · nested_navigators_and_tabs, 08 · dispose_and_leaks).

Compiler Behavior

Not applicable (typed routes still help).

Runtime Behavior

Switching branches via `goBranch` shows the preserved branch stack; navigating within a branch pushes onto that branch's navigator; deep links route to the matching branch.

Flutter Engine Behavior / Dart VM Behavior

Not applicable.

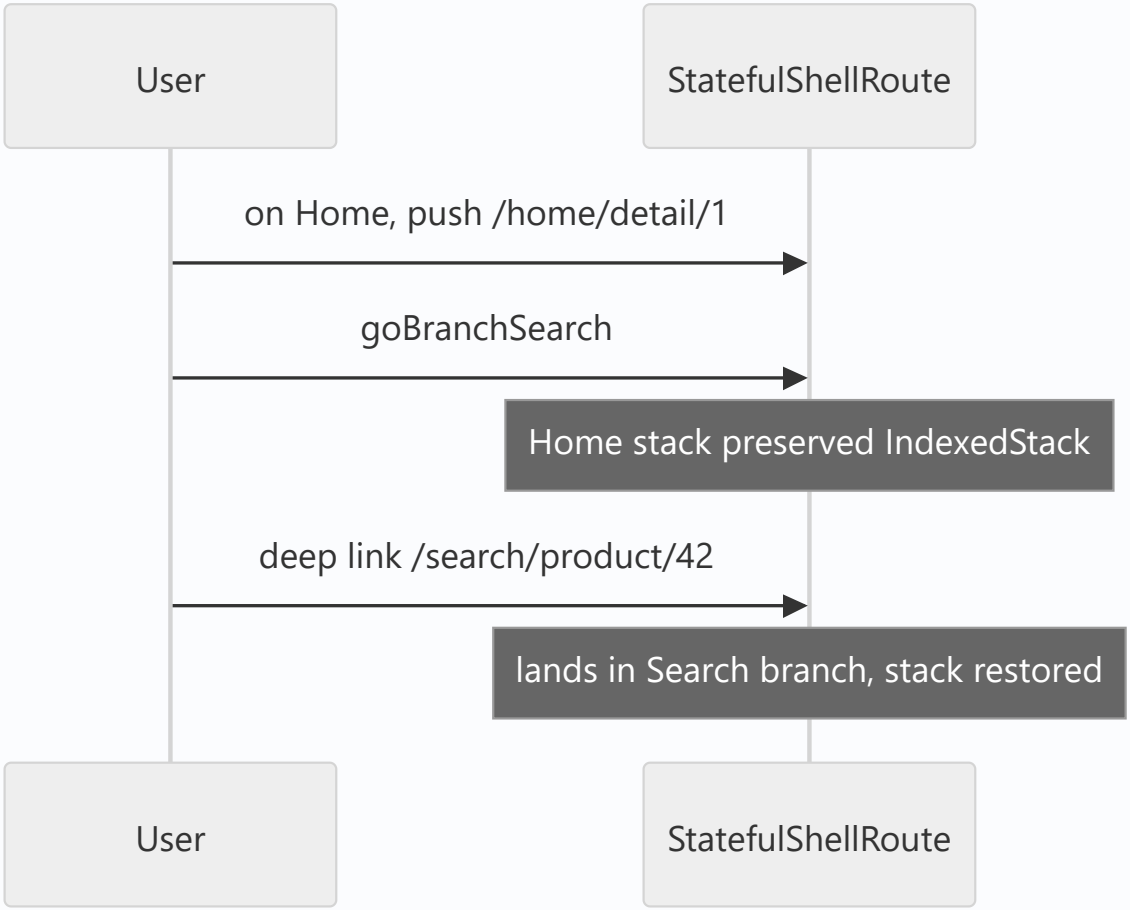
Examples

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


final router = GoRouter(
  initialLocation: '/home',
  routes: [
    StatefulShellRoute.indexedStack(
      builder: (context, state, navigationShell) =>
          ScaffoldWithNav(navigationShell: navigationShell), // persistent shell
      branches: [
```

```
        StatefulShellBranch(routes: [
          GoRoute(
            path: '/home',
            builder: (_, __) => const TabScreen('Home'),
            routes: [ // deep-linkable within the Home branch
              GoRoute(path: 'detail/:id',
                   builder: (_, s) => DetailScreen('Home', s.pathParameters['id']!)),
            ],
          ),
        ]),
        StatefulShellBranch(routes: [
          GoRoute(
            path: '/search',
            builder: (_, __) => const TabScreen('Search'),
            routes: [
              GoRoute(path: 'product/:id',
                   builder: (_, s) => DetailScreen('Search', s.pathParameters['id']!)),
            ],
          ),
        ]),
        StatefulShellBranch(routes: [
          GoRoute(path: '/cart', builder: (_, __) => const TabScreen('Cart')),
        ]),
      ],
    ),
  ],
);

class ScaffoldWithNav extends StatelessWidget {
  final StatefulNavigationShell navigationShell;
  const ScaffoldWithNav({super.key, required this.navigationShell});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: navigationShell, // shows the active branch's navigator (IndexedStack)
      bottomNavigationBar: BottomNavigationBar(
        currentIndex: navigationShell.currentIndex,
        onTap: (i) => navigationShell.goBranch(
          i,
          initialLocation: i == navigationShell.currentIndex, // re-tap -> reset branch
```

```
        ),
        items: const [
          BottomNavigationBarItem(icon: Icon(Icons.home), label: 'Home'),
          BottomNavigationBarItem(icon: Icon(Icons.search), label: 'Search'),
          BottomNavigationBarItem(icon: Icon(Icons.shopping_cart), label: 'Cart'),
        ],
      ),
    );
  }
}

class TabScreen extends StatelessWidget {
  final String name;
  const TabScreen(this.name, {super.key});
  @override
  Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(title: Text(name)),
        body: Center(
          child: ElevatedButton(
            onPressed: () => context.go(
                '/${name.toLowerCase()}${name == 'Search' ? '/product/42' : '/detail/1'}'),
            child: const Text('Push detail in this tab'),
          ),
        ),
      );
}
class DetailScreen extends StatelessWidget {
  final String tab, id;
  const DetailScreen(this.tab, this.id, {super.key});
  @override
  Widget build(BuildContext context) =>
      Scaffold(appBar: AppBar(title: Text('$tab detail $id')));
}

void main() => runApp(MaterialApp.router(routerConfig: router));
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using `ShellRoute` for independent tab stacks	Single navigator, no per-branch state	Use `StatefulShellRoute.indexedStack`
Hand-wiring nested navigators + `IndexedStack`	Reinvents the wheel, not deep-linkable	Use `StatefulShellRoute`
No re-tap reset behavior	Poor tab UX	`goBranch(i, initialLocation: i == current)`
Many heavy branches kept alive	Memory	Keep branches lean; lazy-load heavy content
Deep links not landing in the right tab	Flat routes	Nest branch routes with real paths

Best Practices

- Use `StatefulShellRoute.indexedStack` for tabbed shells needing per-branch stacks + state + deep links.

- Use plain `ShellRoute` only when children share one stack/frame (no independent tabs).
- Give each branch **real nested paths** so deep links land correctly.
- Implement **re-tap resets** via `goBranch(initialLocation:)`.
- Mind memory (all branches alive); keep branch content lean (Module 21).

Performance

`indexedStack` preserves all branches (memory cost) for instant, state-preserving switches — same tradeoff as manual `IndexedStack`, now with deep-link support (12 · nested_navigators_and_tabs).

Advantages / Disadvantages

- **+** Deep-linkable tabs with preserved per-branch stacks/state, persistent shell UI, first-class in `go_router`, far less boilerplate than manual nesting.
- **−** Memory (branches kept alive), conceptually dense, `ShellRoute` vs `StatefulShellRoute` confusion.

Interview Questions

1. 🟢 **`ShellRoute` vs `StatefulShellRoute`?** — `ShellRoute` wraps children in shared UI with one navigator; `StatefulShellRoute` gives each branch its own navigator + preserved state (for tabs).
2. 🟢 **What does `StatefulShellRoute.indexedStack` give you?** — Deep-linkable tabs where each branch keeps its own stack/state (via an `IndexedStack`) inside a persistent shell.
3. 🟡 **How do you switch tabs and get re-tap reset?** — `navigationShell.goBranch(index, initialLocation: index == currentIndex)`.
4. 🟡 **How do deep links land in the right tab?** — Each branch's routes have real nested paths (`/search/product/:id`), so the URL matches that branch and restores its stack.
5. 🟡 **How is this different from Module 12's manual approach?** — Same nested-navigator + `IndexedStack` idea, but first-class and **deep-linkable** with far less wiring.
6. 🔴 **What's the memory implication?** — All branches are kept alive (state preserved) → memory grows with branch count/heft; keep branches lean.
7. 🔴 **When would you use plain `ShellRoute`?** — When children share a single stack/frame (e.g., a common layout) and don't need independent per-tab stacks.

Senior Engineer Tips

- Default to `StatefulShellRoute.indexedStack` for bottom-nav/side-rail apps — it's the modern, deep-linkable tab solution.
- Structure branch routes so every screen is URL-addressable; that's what makes tabs shareable/linkable.

- Watch memory with many/heavy branches; lazy-load expensive tab content.

Architect Perspective

`StatefulShellRoute` makes tabbed shells first-class, URL-addressable, and state-preserving — reconciling the "native tab UX" requirement with deep linking and web. It's the routing structure for most multi-section apps and integrates guards/deep links cleanly, a key scalability/UX decision (Modules 17, 53).

Summary

- `ShellRoute` = shared frame + one stack; `StatefulShellRoute.indexedStack` = per-branch navigators with preserved state + deep links.
- Give branches real nested paths; implement re-tap reset; mind memory.
- The modern, deep-linkable replacement for manual nested navigators.

Revision Notes

- `ShellRoute` (shared UI, single stack) vs `StatefulShellRoute.indexedStack` (per-branch navigator + preserved state + deep links).
- Shell `builder` gets `navigationShell`; `goBranch(i, initialLocation: i==current)` (re-tap reset).
- Branches need real nested paths for deep links; all branches kept alive (memory).
- Modern replacement for manual nested navigators (Module 12).

Practice Questions

1. Why `StatefulShellRoute` over manual nested navigators?
2. How do deep links reach a specific tab?
3. What's the memory tradeoff and how do you mitigate it?

Coding Questions

1. Build a 3-branch `StatefulShellRoute.indexedStack` bottom-nav shell.
2. Add deep-linkable detail sub-routes per branch and test a deep link.
3. Implement re-tap-to-reset on the active tab.

Mini Project

Deep-linkable tab shell (Flutter): Build a `go_router` bottom-nav app with `StatefulShellRoute.indexedStack` (Home/Search/Cart), each branch with a deep-linkable detail route, preserved stacks on switch, re-tap reset, and back popping within the active branch. Verify `/search/product/42` deep-links into Search. Acceptance: preserved per-branch state; deep links land correctly; re-tap resets; app runs.

Type-Safe Routes & Error Handling

> Type-safe routes (via `go_router_builder` codegen) turn stringly-typed paths into compile-checked classes with typed params, and `errorBuilder` / `onException` handle unmatched routes and navigation errors — together making routing refactor-safe and robust.

Introduction

Two production concerns: (1) **type-safe routes** — replacing error-prone path strings (`'/product/${id}'`) with generated route classes (`ProductRoute(id: 42).go(context)`); (2) **error handling** — a fallback UI for unknown/invalid routes (`errorBuilder`) and exception handling. This file covers both.

Why this concept exists

String paths are typo-prone and break silently on refactor (runtime 404). Codegen'd typed routes give compile-time safety and IDE support. And any router needs a graceful "route not found"/error screen instead of a crash or blank page.

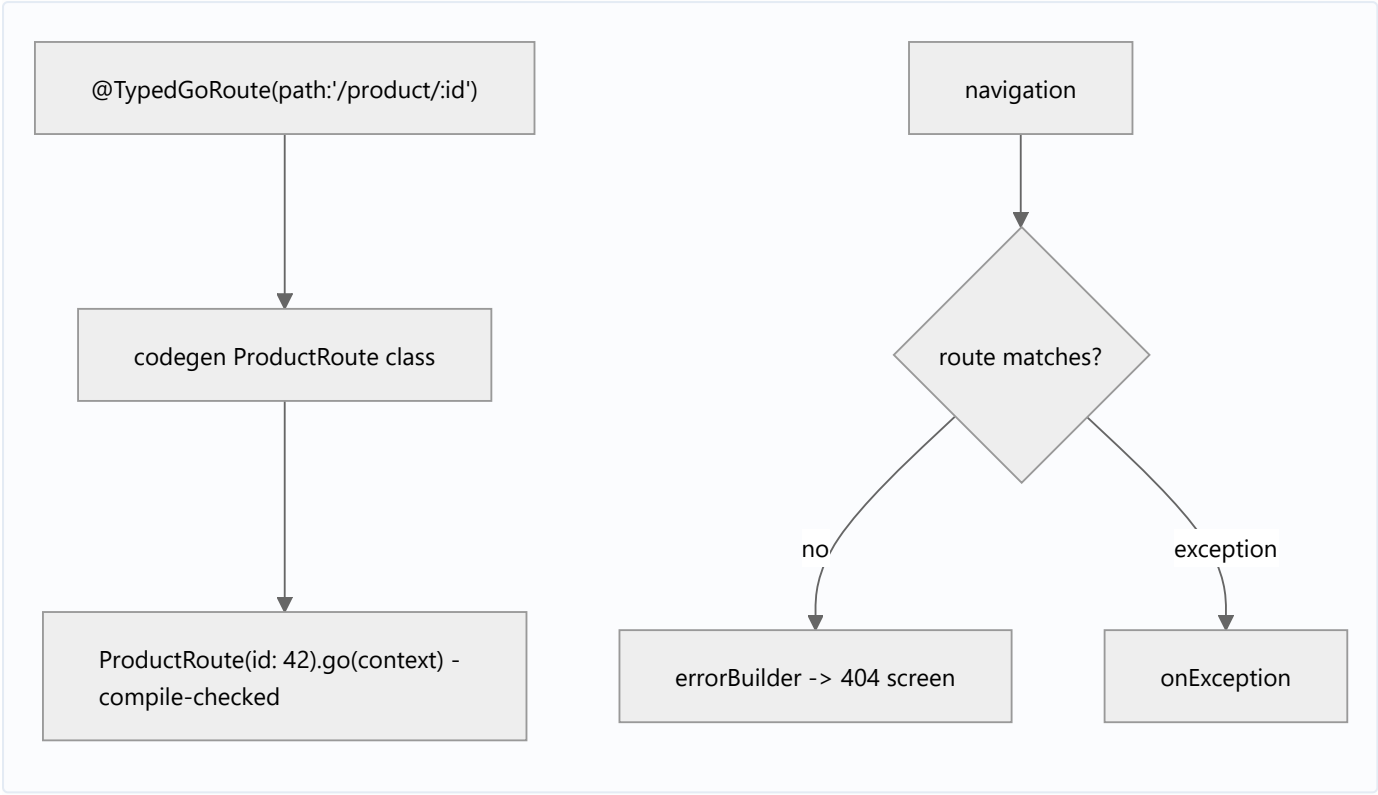
Real-world analogy

Type-safe routes are **pre-addressed, validated envelopes**: you can't mis-address them (the compiler checks), versus hand-writing addresses (typos → lost mail = 404). `errorBuilder` is the **dead-letter office**: mail that can't be delivered goes to a defined handler, not the void.

Problem Statement

You keep breaking `'/product/:id'` links on refactors and want compile-time safety and typed params, plus a proper 404 screen for bad URLs/deep links. You'll use `go_router_builder` typed routes and an `errorBuilder`.

Internal Working



- **Type-safe routes** (`go_router_builder` + `build_runner`): annotate a route class with `@TypedGoRoute<T>(path: '/product/:id')` , declare params as fields; codegen produces `T.go(context)` / `T.push(context)` and a `build` / `GoRouteData` -based builder. Params are typed (`int id`), so typos/wrong types are **compile errors**.
- `errorBuilder` (`GoRouter(errorBuilder: (context, state) => ...)`): renders when no route matches (404) or a build error occurs; `state.error` holds details.
- `onException` (newer API): centralized handling of navigation exceptions (e.g., redirect to a safe location).
- `extra` typing: with typed routes you can carry typed `extra` more safely (still non-URL).
- Typed routes compose with guards/shells from the other files.

Memory Representation

Not applicable beyond normal routing.

Compiler Behavior

The point: typed routes make missing/mistyped paths and params **compile-time errors** rather than runtime 404s — the key safety win. Requires a codegen build step (`dart run build_runner build`).

Runtime Behavior

Navigating a typed route resolves to its path with encoded params; unmatched/invalid locations hit `errorBuilder` / `onException` instead of crashing.

Flutter Engine Behavior / Dart VM Behavior

Not applicable.

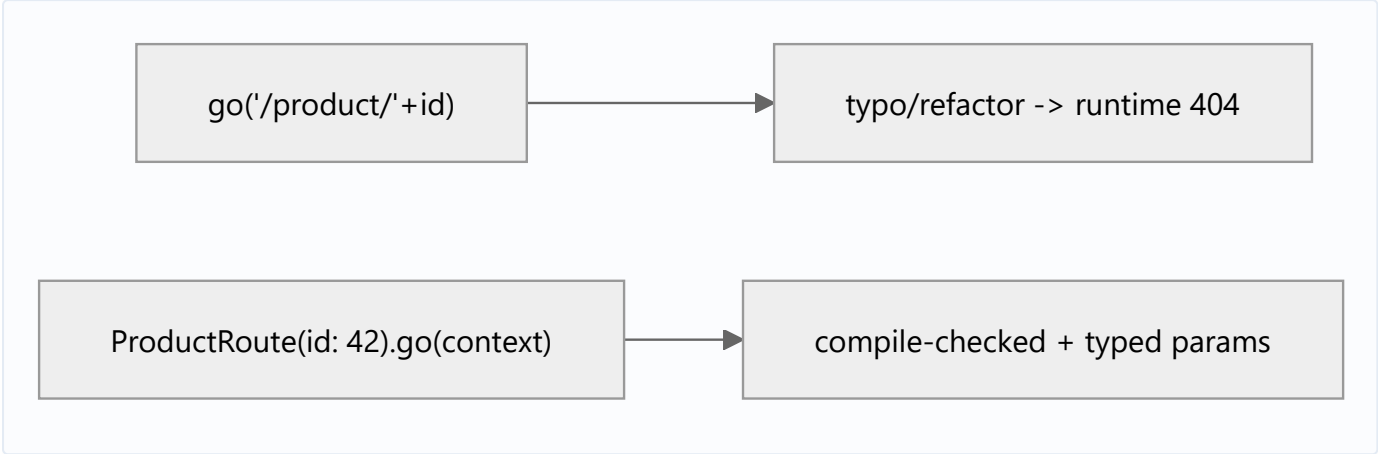
Examples

```yaml
# pubspec.yaml
dependencies:
  go_router: ^14.0.0
dev_dependencies:
  go_router_builder: ^2.0.0
  build_runner: ^2.4.0
```

```dart
import 'package:flutter/material.dart';
import 'package:go_router/go_router.dart';
// part 'routes.g.dart'; // generated by build_runner

// Type-safe route definitions (codegen produces navigation helpers)
@TypedGoRoute<HomeRoute>(path: '/', routes: [
  TypedGoRoute<ProductRoute>(path: 'product/:id'),
])
class HomeRoute extends GoRouteData {
  const HomeRoute();
  @override
  Widget build(BuildContext context, GoRouterState state) => const HomeScreen();
}

class ProductRoute extends GoRouteData {
  final int id;              // typed path param
  final String? ref;         // typed query param
  const ProductRoute({required this.id, this.ref});
  @override
  Widget build(BuildContext context, GoRouterState state) =>
      ProductScreen(id: id, ref: ref);
}
```

```
final router = GoRouter(
  routes: $appRoutes, // generated aggregate of typed routes
  // 404 / error fallback:
  errorBuilder: (context, state) => Scaffold(
    appBar: AppBar(title: const Text('Not found')),
    body: Center(child: Text('No route for "${state.uri}"\n${state.error}')),
  ),
);

// Navigating — compile-checked, typed params (no string typos):
// const ProductRoute(id: 42, ref: 'home').go(context);
// const ProductRoute(id: 42).push(context);

void main() => runApp(MaterialApp.router(routerConfig: router));

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: () => const ProductRoute(id: 42, ref: 'home').go(context),
            child: const Text('Open product 42 (type-safe)'),
          ),
        ),
      );
}
class ProductScreen extends StatelessWidget {
  final int id;
  final String? ref;
  const ProductScreen({super.key, required this.id, this.ref});
  @override
  Widget build(BuildContext context) =>
      Scaffold(appBar: AppBar(title: Text('Product $id (ref: $ref)')));
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Hand-building path strings everywhere	Typos, refactor breakage	Use type-safe routes
No `errorBuilder`	Crash/blank on bad route/deep link	Provide a 404/error screen
Forgetting to run `build_runner`	Generated code stale/missing	Run/watch codegen
Putting objects in typed query/path	URLs can't hold objects	Use ids; `extra` for in-app objects
Ignoring `state.error`	Unhelpful error UI	Surface/log the error appropriately

Best Practices

- Adopt **type-safe routes** for anything beyond a trivial app — compile-time safety + IDE navigation + refactor-friendliness.
- Always provide an `errorBuilder` (404) and handle exceptions (`onException`) gracefully; log `state.error` .
- Keep params **typed and id-based**; reserve `extra` for non-URL data.
- Automate codegen in CI (`build_runner`); commit generated files or generate in CI consistently.
- Combine typed routes with guards/shells for a fully robust router.

Performance

Codegen adds build time, not runtime cost. Error handling is a cheap fallback path. No runtime penalty vs untyped (01_go_router_fundamentals.md).

Advantages / Disadvantages

- + Compile-time route/param safety, IDE autocomplete/refactor, robust 404/error UX, fewer navigation bugs.
- − Codegen setup + build step, more ceremony than raw strings, must keep generated code in sync.

Interview Questions

1. 🟢 **What do type-safe routes give you?** — Compile-checked navigation with typed params (via `go_router_builder` codegen), eliminating string-typo runtime 404s.
2. 🟢 **What is `errorBuilder` for?** — Rendering a fallback (404/error) screen when no route matches or a build error occurs.
3. 🟡 **How are typed routes defined?** — Annotate `GoRouteData` classes with `@TypedGoRoute`, declare params as fields; codegen generates `.go` / `.push` helpers and the route table (`$appRoutes`).
4. 🟡 **What's the tradeoff of type-safe routes?** — Compile-time safety + refactor-friendliness at the cost of a codegen build step and extra ceremony.
5. 🟡 **How do you handle a deep link to a non-existent route?** — `errorBuilder` renders a 404 (log `state.error`); optionally `onException` / `redirect` to a safe page.
6. 🔴 **Where does the compile-time safety come from?** — Params/paths become Dart types/fields, so mistakes are caught by the analyzer/compiler, not at runtime.
7. 🔴 **How do typed routes handle non-URL objects?** — Still via `extra` (not encoded in the URL); URL params stay id/primitive-based.

Senior Engineer Tips

- Turn on type-safe routes early; migrating string call sites later is tedious.
- Wire `build_runner` into CI and pre-commit so generated routes never go stale.
- Pair a good `errorBuilder` with logging/monitoring so bad deep links are visible (Module 52).

Architect Perspective

Type-safe routes + robust error handling make routing a compile-checked, resilient contract — fewer runtime navigation bugs, safe refactors, and graceful failure for bad links. Combined with guards (03_guards_and_redirects.md) and shells (04_shell_routes_and_nested.md), it forms a production routing layer that scales across large teams and platforms.

Summary

- Type-safe routes (codegen) give compile-checked navigation with typed params; `errorBuilder` / `onException` handle 404s/errors.
- Prefer typed, id-based routes; `extra` for in-app objects; always provide an error screen.
- Automate codegen; combine with guards/shells for a robust router.

Revision Notes

- Type-safe: `@TypedGoRoute<T>` on `GoRouteData` + `build_runner` → `T(params).go(context)` (compile-checked). `routes: $appRoutes`.
- `errorBuilder` (404/error, `state.error`) + `onException` (centralized handling).
- Typed/id params; `extra` for non-URL objects; automate codegen in CI.
- Compose with guards + shells for production routing.

Practice Questions

1. How do type-safe routes prevent runtime 404s?
2. What renders on an unmatched deep link?
3. What's the cost of adopting type-safe routes?

Coding Questions

1. Define a typed `ProductRoute(id, ref)` and navigate with it.
2. Add an `errorBuilder` 404 screen surfacing `state.error`.
3. Set up `build_runner` and regenerate after adding a route.

Mini Project — Module capstone

Production router (Flutter): Build a `go_router` app combining all four concerns: type-safe routes (codegen), an auth guard (`redirect` + `refreshListenable`), a deep-linkable `StatefulShellRoute` tab shell, and an `errorBuilder` 404 — with clean web URLs (path strategy). Acceptance: compile-checked navigation; guards + deep links + tabs all work; 404 handled; app runs. (This is the Module 13 routing backbone for a real app.)