Author: @xster, @gaaclarke
Go Link: flutter.dev/go/multiple-flutters
Created: 01/20 / **Last updated:** 01/20

Context

Using multiple instances of lightweight, encapsulated Flutter containers has been an area of exploration since our very early customers started pioneering the add-to-app concept in mid 2018. This topic continues to be a frequent ask by customers like xxxxxx yyyyyyy (go/multiple-flutters).

flutter/flutter/issues/37644

In the absence of any investments from the Flutter team, the community has created a whole slew of different guides using a variety of approaches to solve this problem:

- ByteDance: view-level hybrid development (original)
- Weidian: hybrid navigation stack management (original)
- Mafengwo: cross-platform development with Flutter (original)
- Alibaba: hybrid development 1 (original)
- Alibaba: start hybrid development with flutter_boost (original)

- 360: reuse principle with hybrid stack (original)
- flutter_boost page changing and channel data passing (original)
- mixed navigation stack management (original)

The work-around flutter_boost project has twice as many GitHub stars as the sqlite plugin.

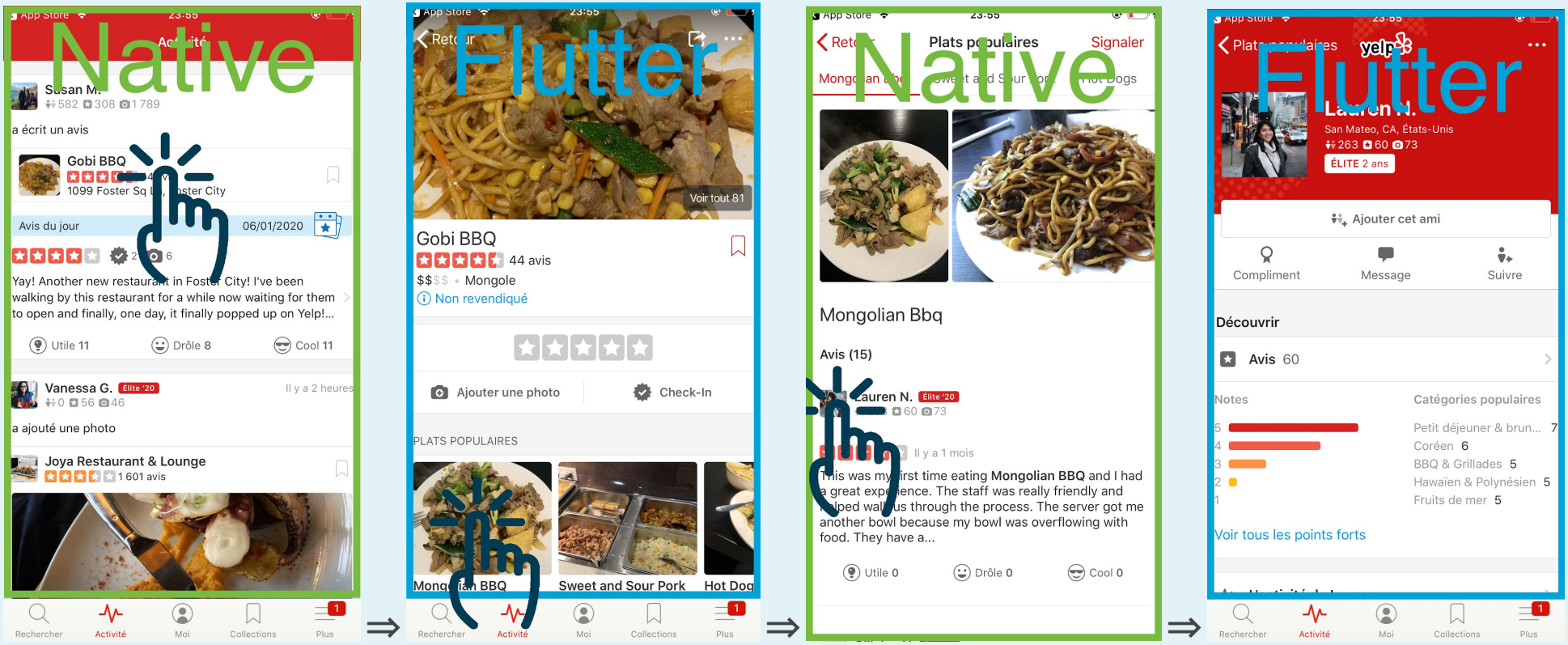
But the proposed work-arounds are complex for the torso market.

Test use cases

We'll use the examples below to test any proposed approaches.

We'll use Yelp as the example app.

Case 1 - navigation stack mixing



In this sequence of actions, the home feed (1) is implemented in native iOS. Clicking on a restaurant opens a restaurant details page (2) implemented in Flutter. Clicking on a menu item image then opens a food details page (3) in native iOS. Clicking on a reviewer's profile icon opens a user details page (4) implemented in Flutter.

Here, pages (2) and (4) don't share a lot of UI (e.g. no RenderObject instances can be sensibly shared between the 2 pages). They may be running off of different source Dart code which may be in non-interdependent Dart packages developed by different teams in the company.

From page (4), it may be possible to click some UI that navigates yet back to another instance of page (2).

Each page should have independent UI state. For instance, leaving page (2) in some scroll position and navigation to pages (3) and (4) shouldn't unload all State instance from page (2)'s StatefulElement tree or destroy any Dart objects constructed from page (2). Popping the conceptual navigation stack back to page (2) should return to the screen with the previous scroll position. This applies to other UI

states as well such as partially entered text etc.

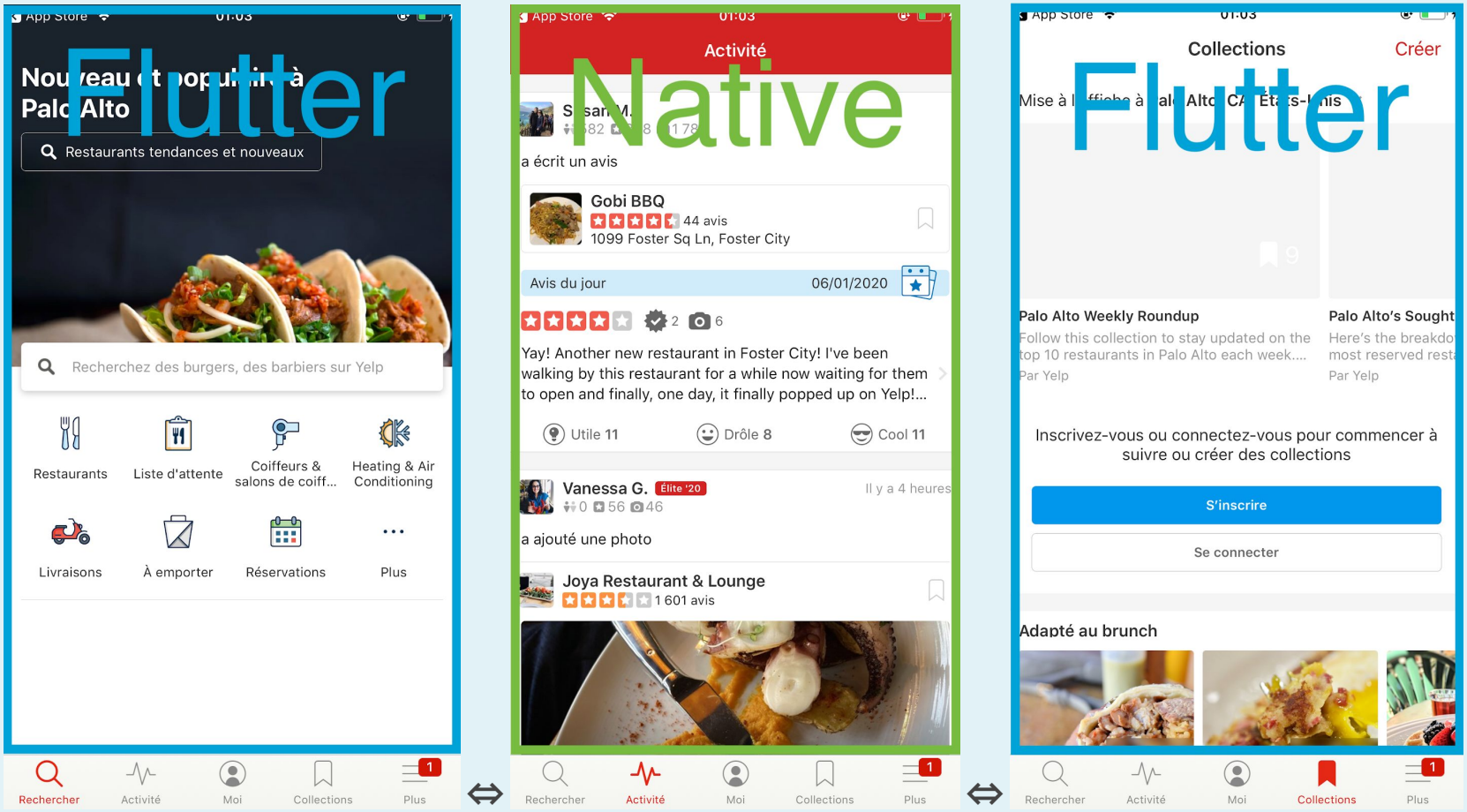
If page (4) navigated to yet another instance of page (2), that second page (2) should have its own UI state, such as the scroll position, independently from the first instance of page (2).

The 2 Flutter containers may, however, also share some Dart state such as a common Dart object representing the currently signed in user, or data in an image cache.

In this case, the navigation stack's history itself is most pragmatically held on the platform side rather than inside Flutter's Navigator (which is unaware of platform routes).

As a slight edge case, it may be also possible for page (1) to navigate to page (3). In that case, 2 Flutter views will be concurrently visible and rendering.

Case 2 - tabs

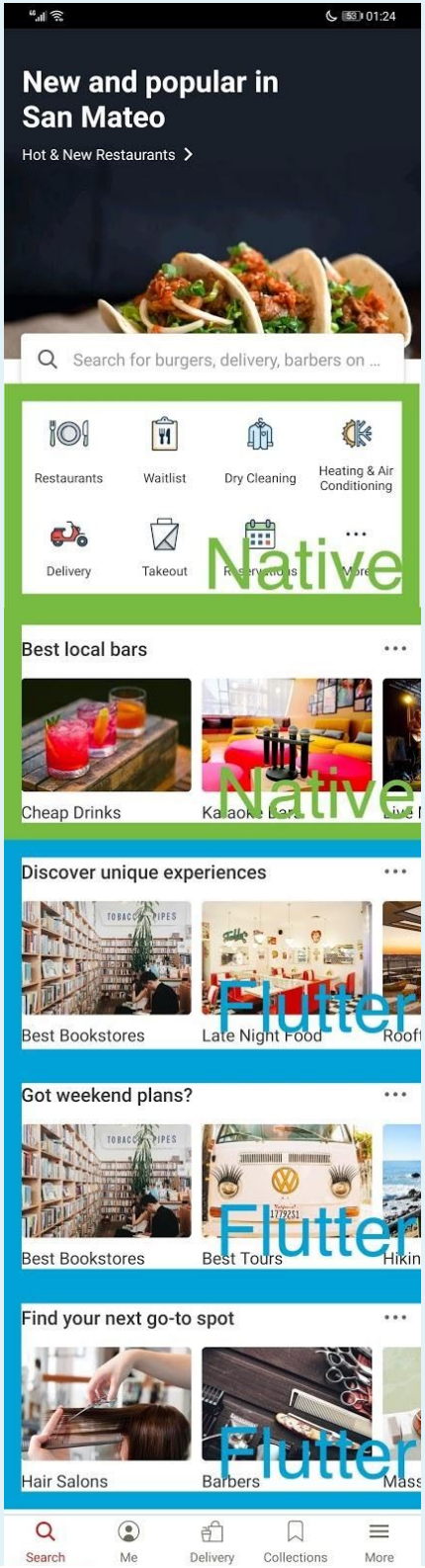


Here, the Flutter screens can be accessed in parallel via tabs rather than sequentially via the navigation stack, but the root concepts are similar. The first 'search' tab and the 4th 'collections' tab are mostly independent and mostly don't share UI or any same instances in the Elements tree.

Here too, each tab should keep its own UI state.

Since Yelp here follows the iOS navigation model where each parallel tab has its own concurrent navigation stacks, the second 'activity' tab may either be a native screen or another Flutter screen if the user already navigated into a restaurant details page implemented in Flutter (such as in case 1).

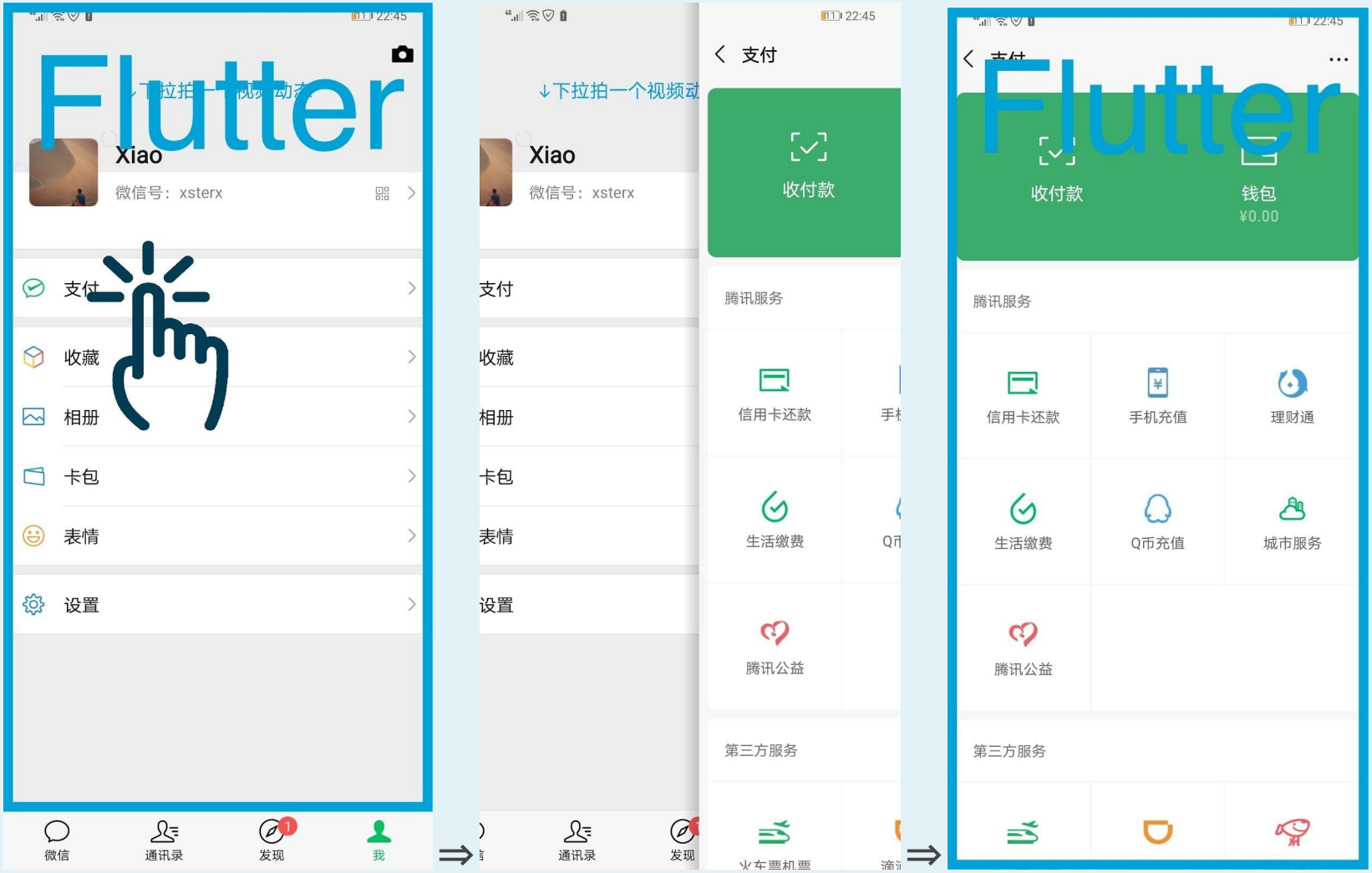
Case 3 - concurrent partial views



In this example, assume that the feed list of the first ‘search’ tab is an native iOS UITableView but the cell contents are delegated to various feature teams to implement. Each team is responsible of delivering an opaque UITableViewCell implementation that may internally be done in native UIKit views or with Flutter and may toggle on and off remotely via A/B testing.

Each feature team is independent and does not know whether the other cells are implemented in native or Flutter.

Case 4 - reparenting



This is a special casing of case 1 that's common in Android-patterned bottom bar apps. Where the bottom bar is inside one of the pages rather than on top of the entire app.

Using WeChat as an example to illustrate.

The bottom tab toggles between 4 content tabs, one of which is Flutter. The bottom tab itself and the other tabs are done in native. Upon clicking an item in tab 4 (such as payment in this case), a new page is brought modally on top of the entire screen including all the tabs. Once the transition is done, the bottom tab is no longer visible and the new page is fullscreen.

One might argue for case 1 that Flutter can intercept all navigations and in case it's a Flutter->Flutter navigation, the navigation can happen internally inside of the same Flutter container rather than as a separate Flutter container. This isn't feasible for this case 4 (or for case 3 for that matter) since the 2 Flutter views have different parent containers (one as a child to the bottom tab and one as a sibling to the bottom tab).

Considerations

Factors for consideration when considering approaches.

Ergonomics

Developers of the UI modules should be able to write normal Flutter code without any routing code that toggles between different screens. Accidental leaks between UI modules of different business concerns should be difficult.

The platform side code should have clear handles to each of the Flutter containers to be able to send platform channel messages to specific Flutter instances.

The navigation stack history's ergonomics should be similar to webviews. For instance, if 2 tabs have 2 webviews, the developer shouldn't have to worry about tab 1's navigation stack mixing with tab 2's navigation stack.

There are also no CSS rules one can apply to the webview on one tab that would accidentally affect the UI of the other tab's webview. Similarly, no Flutter widgets in one Flutter container should accidentally affect the UI in another container (unless via explicit platform channel or isolate port messages).

Performance

These containers should have the right degree of isolation for ergonomics but otherwise be lightweight.

One stress test is in case 1. If the native->Flutter->native->Flutter chain continues ad-infinitum, the memory consumption curve shouldn't be drastically higher than a native->native->native->native equivalent chain (beyond the one time fixed cost of going from 0 to 1 Flutter instance).

Considering the numbers tested for phase 1 in https://flutter.dev/docs/development/add-to-app/performance, it costs about 38mb of dirty memory to render to 1 Flutter view (on iOS for instance).

Of which, 4mb for creating pthreads, 10mb for GPU drivers, 1mb for Dart VM managed memory, 5mb for Flutter loaded font maps, 16mb for graphics buffers.

For cases where only 1 Flutter container is visible concurrently, having 2 containers co-existing (such as in the native->Flutter->native->Flutter navigation stack case) should certainly not cost 38mb * 2. A more reasonable number would be closer to 38mb + 1mb (for VM memory of the new isolate) + 1mb (for a new pthread hosting the Dart/UI thread).

Simultaneous rendering

As shown in case 3 and case 4, multiple Flutter containers may be simultaneously rendering UI. While it may be of lower priority vs case 1 and 2 which are more commonly asked, we shouldn't architecturally exclude ourselves from it.

Sharing data

While most of the UI state is not shared between different Flutter containers, it should be possible to share some Dart data between the containers. An example would be an image cache in Dart that could be used by all the containers, or a Dart "User" class instance which represents the business logic of the currently logged-in user behind all the Flutter UI containers.

Having all the isolates call SendPort.send(dynamic message) in a daisy chain is not an acceptable solution.

The Dart team is also pursuing Lightweight Isolates & Faster isolate communication.

Formalizing the IPCs into autogenerated, strongly typed, schema'ed APIs could help too via https://pub.dev/packages/pigeon.

Sharing plugins

In the native->Flutter->native->Flutter ad infinitum case, it may not be sensible to have as many instances of Flutter plugins instantiated and registered.

Lifecycle states

The Dart-side UI should have awareness of the lifecycle states of its respective platform-side container (https://github.com/flutter/flutter/issues/52456). i.e. the Dart logic populating the UI content for a FlutterActivity or FlutterViewController should be aware of when the FlutterActivity or FlutterViewController is becoming visible and invisible.

Possible approaches

========== WIP ==========

This is just a back of the napkin quick guess with no empirical testing so far:

Approach		**1**	**2**	**3**	**4**
Component	VM	1	1	1	1
	Engine (embedding)	1	many	many	many
	Isolate	1	1	many	many + 1 shared isolate
	Thread	1 group	many	many	1 platform thread many Dart/UI threads 1 IO thread pool 1 raster thread
	Window	1	many	many	many
	View	many	many	many	many

Evaluation	Prior art	This is basically what Alibaba did with flutter_boost. All the toggling is done inside Flutter-land as an indexed stack.	This is essentially what ByteDance did to support multiple Flutters (as documented in their blog).	This is the default no-additional-work solution. If we just tell people to start using multiple FlutterEngines, this mostly already works.	A hypothetical combination of various traits to get combine the advantages where possible.
	Pros	• No work needed. We can just point people to flutter_boost. • Memory efficient. Having many n->f->n->f cycles is the same as having many routes inside Flutter. • Easy to share data inside Flutter.	• No more moving the platform view around vs approach 1. • Memory efficient. • Easy to share data. • Can render concurrently.	• Ergonomic API with clear encapsulation. • No work needed. • Per engine/view lifecycles are clear. • Can render concurrently.	• Ergonomic API with encapsulation. • Clear lifecycle. • May be able to avoid cost associated with per thread stack cost and GL contexts cost. • Shared isolate for data sharing. • Can render concurrently.

PUBLICLY SHARED

	Cons	• No encapsulation or isolation. UI can cross-contaminate. • Flutter views can't concurrently render. • Edge cases where multiple Flutter views show such as an iOS edge swipe is done with a screenshot. • Widgets can't subscribe as WidgetsBindingsObserver and conveniently make sense of AppLifecycleState. • Have to reparent and move the platform view around.	• Still minimal encapsulation via Zones. A new engine starting translates to an entrypoint against an already running isolate.	• May be memory intensive. • Hard to share data.	• Have to implement.

Prototypes and measurements

TODO