What desktop multi-window support means for Flutter

Overview

There have been a number of documents circulating about aspects of “multi-window” support for Flutter and it is time to step back a bit and consider what is needed from multiple windows in a desktop environment. This document aims to provide an overview of the requirements, constraints and a reference to the more detailed discussions.

Windows in a desktop environment

There are a lot more windows used by an application than might naively be thought. Top level windows and dialogs are obvious.

Menus, popups and tooltips are less obviously windows but need to be considered as such: for most DEs they are not constrained by the edges of the application window and in some DEs are constrained by elements of the desktop.

There are Flutter platforms that are not DEs for which a single window controlling the full screen is all that is needed. In this environment, painting popups and tooltips directly in the display works at the application level. But this isn’t true when they are represented as separate windows.

On the desktop, for windows to be handled correctly by Flutter the application code needs to inform Flutter how they relate to each other; and, what they represent (menus are placed and drawn differently to dialogs).

Desktop environments

Desktop environments [DEs] share the screen or screens between multiple applications allowing each application to present one or more windows. These environments also display windows that do not belong to any application.

Desktop environments have information about all windows both from applications and their own. But different ones share varying amounts of information about windows (from the same application and others) and screens with applications.

They also offer varying support for applications managing their own windows: some allow the application to place windows anywhere, others only allow requests for where a window is placed with respect to its parent and interpret the request in the context of other elements of the display.

While this discussion is largely about the placement of windows (because it has proved the most contentious) it should be noted that there are other differences between desktop environments: the way that focus is handled for related windows, client-side window decoration vs server-side decorations, etc.

Applications written using Flutter need to work with DEs across this range of behaviours and ideally Flutter should be concealing the differences from the application code.

Windows in an application

Although there is an unbounded range of behaviours that applications might want from their windows there is a much smaller range of behaviours that are familiar to both application developers and users. We have offered one taxonomy of these common window behaviours in Window management language.

Flutter needs to guarantee support for a core set of familiar window behaviours across the full range of DEs. Providing a core set of window behaviours across multiple target systems makes life easier for developers. Initially it is probably best to begin with the minimal useful set of behaviours (menu, popup, tip) and extend this once the necessary changes have been made to the Flutter architecture.

Outside this core set of window behaviours there may be behaviours desired by the application that are unsupportable on the platform. This possibility implies that there needs to be some negotiation between the application, Flutter and the embedder's knowledge of the host environment to determine what is possible.

Flutter architecture for multi-window support

The two layers in the Flutter architecture that most need to be discussed here are the high level Framework layer that the application interacts with and the low level Embedder layer that interacts with the underlying OS including its DE. These are the elements that constrain the implementation of multi-wIndow.

The embedder

The embedder provides the capabilities needed by Flutter that need to be implemented using the capabilities of the underlying platform.

As discussed above the windowing capabilities of the underlying DE are not uniform: it may be possible to position windows absolutely on the display; or, it may only be possible to request the size and relative placement with respect to a parent window. In order to support the latter the parent and placement with respect to the parent needs to be passed to the embedder.

For the core set of window behaviours mentioned above an API that provides support for "parent and placement with respect to the parent" is a necessary API on some platforms (primarily Wayland on Linux).

It is possible to derive an absolute position from such an API and, in theory, the API could be required everywhere. However, the embedder is intended to be the minimal platform specific code required. Making the embedder responsible for interpreting a placement request would be an unnecessary burden on every embedder implementation for a platform that only supports absolute placement (and that is the majority at present).

An additional consideration is that window behaviours outside the core set will require support from APIs beyond those linked above. Which of these can be implemented will depend upon the underlying platform. Thus in any case, the embedder will need to provide information on which APIs are available.

The engine

The changes needed to the engine are twofold: support for rendering into multiple Views which is an active discussion elsewhere (not discussed herein); and, mediating between the embedder and framework regarding window behaviour.

In order to support the core set of window behaviours on an API along the lines discussed in More Desktop Multi-Window support for Flutter provides support for “parent and placement with respect to the parent” and is a necessary API on some platforms.

Similarly to the above discussion in “the embedder”, the engine could provide a core set of window behaviours based on “parent and placement with respect to the parent” and map these according to the capabilities of the embedded and current platform.

The framework

The framework exposes the capabilities of Flutter to the application code. This is the place where information about the behaviour expected by the application needs to be collected by Flutter. It seems logical to expose elements such as “menu” at this level that correspond to the core set of window behaviours. It ought to be possible to have these degrade to Widgets within the parent window on platforms that do not support multi window desktops.

The architectural decision needed

Somewhere in Flutter we need to translate what the application wants into the host platform APIs or protocols to fulfil that requirement. This translation could occur in the embedder, the engine or the framework. Wherever it happens the information needed for the translation must be passed through any intermediate from the application.

A single embedder API complicates the embedder almost everywhere. The alternative is to allow the embedder to offer alternative APIs depending upon the platform it is running on. (Note, that in the Linux world this is determined at runtime: the DE may be based on either X11 or Wayland and these protocols have very different philosophies about Window management.)

If the embedder supports alternative APIs then the complexity could reside in the engine. That would allow for a single version of the translation code that could be shared across platforms and the widgets and designs in the framework.

Putting the complexity in the framework gives the greatest flexibility there, but would likely lead to duplication of the logic and difficulty testing that logic for framework developers that focus on a particular platform.

While there are clearly trade-offs for and against each approach, allowing embedders to offer alternative APIs according to the capabilities of the platform and translating in the engine is the recommended approach. This does come with the challenge of designing alternative embedder APIs and an engine API that can work across these embedder APIs without losing the expressiveness needed by the framework.

Implementation strategy

There has already been some work on Multiple rendering surfaces on macOS and the ability of the engine and embedder to render multiple views is a prerequisite for supporting multiple windows on the desktop.

Following this, there needs to be a choice of an initial set of “core window behaviours” to be supported through the architecture. That will allow most of the architectural decisions to be proven without running into the many corner cases that will come with supporting everything imaginable everywhere.

As noted above there is an architectural decision to be made about where the complexity of supporting different platform capabilities belongs. Consensus on that decision will facilitate design and review of the APIs that are needed to implement the management window behaviours.

References

- Desktop Multi-Window Support (PUBLICLY SHARED)
- Window management language
- More Desktop Multi-Window support for Flutter
- Multiple Views (PUBLICLY SHARED)
- Multiple rendering surfaces on macOS (PUBLICLY SHARED)