Phát triển ứng dụng di động với Flutter

<https://flutter.dev/docs/resources/technical-overview>

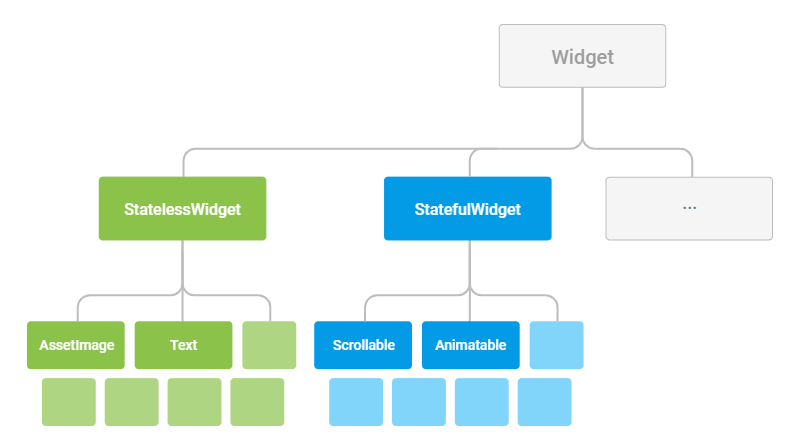
What is Flutter?

Flutter is an app SDK for building high-performance, high-fidelity apps for iOS, Android, and web (tech preview) from a single codebase.

Why use Flutter?

What are some advantages of Flutter? It helps you:

* Be highly productive
  + Develop for iOS and Android from a single codebase
  + Do more with less code, even on a single OS, with a modern, expressive language and a declarative approach
  + Prototype and iterate easily
    - Experiment by changing code and reloading as your app runs (with hot reload)
    - Fix crashes and continue debugging from where the app left off
* Create beautiful, highly-customized user experiences
  + Benefit from a rich set of Material Design and Cupertino (iOS-flavor) widgets built using Flutter’s own framework
  + Realize custom, beautiful, brand-driven designs, without the limitations of OEM widget sets



Everything’s a widget

Widgets are the basic building blocks of a Flutter app’s user interface. Each widget is an immutable declaration of part of the user interface. Unlike other frameworks that separate views, view controllers, layouts, and other properties, Flutter has a consistent, unified object model: the widget.

A widget can define:

* a structural element (like a button or menu)
* a stylistic element (like a font or color scheme)
* an aspect of layout (like padding)

and so on…

Widgets form a hierarchy based on composition. Each widget nests inside, and inherits properties from, its parent. There is no separate “application” object. Instead, the root widget serves this role.

You can respond to events, like user interaction, by telling the framework to replace a widget in the hierarchy with another widget. The framework then compares the new and old widgets and efficiently updates the user interface.

Building widgets

You define the unique characteristics of a widget by implementing a build() function that returns a tree (or hierarchy) of widgets. This tree represents the widget’s part of the user interface in more concrete terms. For example, a toolbar widget might have a build function that returns a horizontal layout of some text and various buttons. The framework then recursively asks each of these widgets to build until the process bottoms out in fully concrete widgets, which the framework then stitches together into a tree.

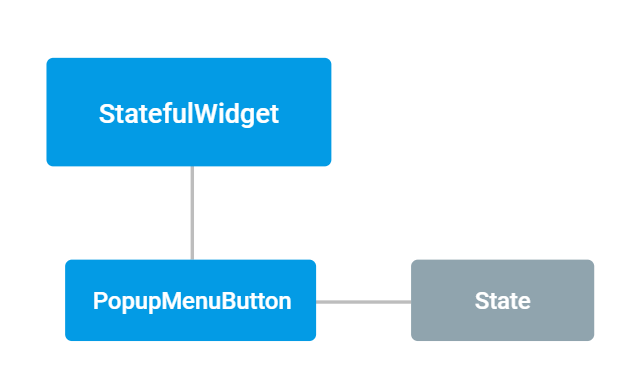
A widget’s build function should be free of side effects. Whenever it is asked to build, the widget should return a new tree of widgets regardless of what the widget previously returned. The framework does the heavy lifting of comparing the previous build with the current build and determining what modifications need to be made to the user interface.

This automated comparison is quite effective, enabling high-performance, interactive apps. And the design of the build function simplifies your code by focusing on declaring what a widget is made of, rather than the complexities of updating the user interface from one state to another.

Handling user interaction

If the unique characteristics of a widget need to change based on user interaction or other factors, that widget is stateful. For example, if a widget has a counter that increments whenever the user taps a button, the value of the counter is the state for that widget. When that value changes, the widget needs to be rebuilt to update the UI.

These widgets subclass StatefulWidget (rather than StatelessWidget) and store their mutable state in a subclass of State.



Whenever you mutate a State object (for example, by incrementing the counter), you must call setState() to signal the framework to update the user interface by calling the State’s build method again. For an example of managing state, see the MyApp template that’s created with each new Flutter project.

Having separate state and widget objects lets other widgets treat stateless and stateful widgets in the same way, without being concerned about losing state. Rather than needing to hold on to a child to preserve its state, the parent is free to create a new instance of the child without losing the child’s persistent state. The framework does all the work of finding and reusing existing state objects when appropriate.