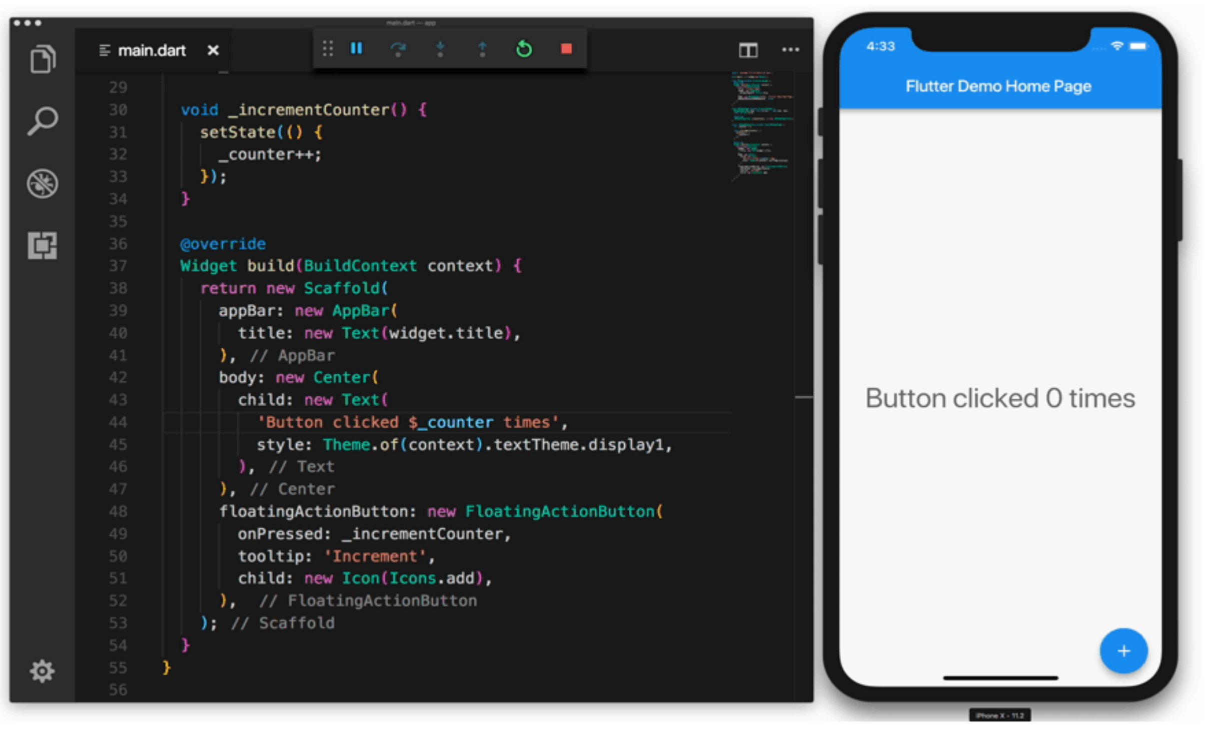
Hot reload

Contents

Flutter's hot reload feature helps you quickly and easily experiment, build UIs, add features, and fix bugs. Hot reload works by injecting updated source code files into the running Dart Virtual Machine (VM). After the VM updates classes with the new versio of fields and functions, the Flutter framework automatically rebuilds the widget tree, allowing you to quickly view the effects of yc changes.

To hot reload a Flutter app:

1. Run the app from a supported Flutter editor or a terminal window. Either a physical or virtual device can be the target. **Only Flutter apps in debug mode can be hot reloaded.**
2. Modify one of the Dart files in your project. Most types of code changes can be hot reloaded; for a list of changes that requi hot restart, see Limitations.
3. If you're working in an IDE/editor that supports Flutter's IDE tools, select **Save All** (`cmd-s`/`ctrl-s`), or click the Hot Reload bu on the toolbar:



```
  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return new Scaffold(
      appBar: new AppBar(
        title: new Text(widget.title),
      ), // AppBar
      body: new Center(
        child: new Text(
          'Button clicked $_counter times',
          style: Theme.of(context).textTheme.display1,
        ), // Text
      ), // Center
      floatingActionButton: new FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: new Icon(Icons.add),
      ),  // FloatingActionButton
    ); // Scaffold
  }
}
```

If you're running the app at the command line using `flutter run`, enter `r` in the terminal window.

After a successful hot reload operation, you'll see a message in the console similar to:

```
Performing hot reload...
Reloaded 1 of 448 libraries in 978ms.
```

The app updates to reflect your change, and the current state of the app—the value of the counter variable in the above example– preserved. Your app continues to execute from where it was prior to running the hot reload command. The code updates and execution continues.

A code change has a visible effect only if the modified Dart code is run again after the change. Specifically, a hot reload causes a the existing widgets to rebuild. Only code involved in the rebuilding of the widgets are automatically re-executed.

The next sections describe common situations where the modified code *won't* run again after hot reload. In some cases, small changes to the Dart code enable you to continue using hot reload for your app.

Compilation errors

When a code change introduces a compilation error, hot reload always generates an error message similar to:

```
Hot reload was rejected:
'/Users/obiwan/Library/Developer/CoreSimulator/Devices/AC94F0FF-16F7-46C8-B4BF-
218B73C547AC/data/Containers/Data/Application/4F72B076-42AD-44A4-A7CF-
57D9F93E895E/tmp/ios_testWIDYdS/ios_test/lib/main.dart': warning: line 16 pos 38: unbalanced '{' opens her
  Widget build(BuildContext context) {
                                     ^
'/Users/obiwan/Library/Developer/CoreSimulator/Devices/AC94F0FF-16F7-46C8-B4BF-
218B73C547AC/data/Containers/Data/Application/4F72B076-42AD-44A4-A7CF-
57D9F93E895E/tmp/ios_testWIDYdS/ios_test/lib/main.dart': error: line 33 pos 5: unbalanced ')'
    );
    ^
```

In this situation, simply correct the errors on the specified lines of Dart code to keep using hot reload.

Previous state is combined with new code

Flutter's Stateful Hot Reload preserves the state of your app. This design enables you to view the effect of the most recent chang
only, without throwing away the current state. For example, if your app requires a user to log in, you can modify and hot reload a p
several levels down in the navigation hierarchy, without re-entering your login credentials. State is kept, which is usually the desire
behavior.

If code changes affect the state of your app (or its dependencies), the data your app has to work with might not be fully consister
with the data it would have if it executed from scratch. The result might be different behavior after hot reload versus a hot restart

For example, if you modify a class definition from extending StatelessWidget to StatefulWidget (or the reverse), after hot reload tl
previous state of your app is preserved. However, the state might not be compatible with the new changes.

Consider the following code:

```
class MyWidget extends StatelessWidget {
  Widget build(BuildContext context) {
    return GestureDetector(onTap: () => print('T'));
  }
}
```

After running the app, if you make the following change:

```
class MyWidget extends StatefulWidget {
  @override
  State<MyWidget> createState() => MyWidgetState();
}

class MyWidgetState extends State<MyWidget> { /*...*/ }
```

Then hot reload; the console displays an assertion failure similar to:

```
MyWidget is not a subtype of StatelessWidget
```

In these situations, a hot restart is required to see the updated app.

Recent code change is included but app state is excluded

In Dart, static fields are lazily initialized. This means that the first time you run a Flutter app and a static field is read, it is set to
whatever value its initializer was evaluated to. Global variables and static fields are treated as state, and are therefore not reinitial
during hot reload.

If you change initializers of global variables and static fields, a full restart is necessary to see the changes. For example, consider
following code:

```
final sampleTable = [
  Table("T1"),
  Table("T2"),
  Table("T3"),
  Table("T4"),
];
```

After running the app, if you make the following change:

```
final sampleTable = [
  Table("T1"),
  Table("T2"),
  Table("T3"),
  Table("T10"),    // modified
];
```

and then hot reload, the change is not reflected.

Conversely, in the following example:

```
const foo = 1;
final bar = foo;
void onClick() {
  print(foo);
  print(bar);
}
```

running the app for the first time prints `1` and `1`. Then, if you make the following change:

```
const foo = 2;    // modified
final bar = foo;
void onClick() {
  print(foo);
  print(bar);
}
```

While changes to `const` field values are always hot reloaded, the static field initializer is not rerun. Conceptually, `const` fields are treated like aliases instead of state.

The Dart VM detects initializer changes and flags when a set of changes needs a hot restart to take effect. The flagging mechani is triggered for most of the initialization work in the above example, but not for cases like:

```
final bar = foo;
```

To be able to update `foo` and view the change after hot reload, consider redefining the field as `const` or using a getter to return th value, rather than using `final`. For example:

```
const bar = foo;
```

or:

```
get bar => foo;
```

Read more about the differences between the `const` and `final` keywords in Dart.

Recent UI change is excluded

Even when a hot reload operation appears successful and generates no exceptions, some code changes might not be visible in th refreshed UI. This behavior is common after changes to the app's `main()` method.

As a general rule, if the modified code is downstream of the root widget's build method, then hot reload behaves as expected. However, if the modified code won't be re-executed as a result of rebuilding the widget tree, then you won't see its effects after hc reload.

For example, consider the following code:

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

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  Widget build(BuildContext context) {
    return GestureDetector(onTap: () => print('tapped'));
  }
}
```

After running this app, you might change the code as follows:

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

void main() {
  runApp(const Center(
      child: const Text('Hello', textDirection: TextDirection.ltr)));
}
```

With a hot restart, the program starts from the beginning, executes the new version of `main()`, and builds a widget tree that displa the text `Hello`.

However, if you hot reload the app after this change, `main()` is not re-executed, and the widget tree is rebuilt with the unchanged instance of `MyApp` as the root widget. The result is no visible change after hot reload.

Limitations

You might also encounter the rare cases where hot reload is not supported at all. These include:

- Changes on `initState()` are not reflected by hot reload. A hot restart is required.
- Enumerated types are changed to regular classes or regular classes are changed to enumerated types. For example, if you change:

  ```
  enum Color {
    red,
    green,
    blue
  }
  ```

  to:

  ```
  class Color {
    Color(this.i, this.j);
    final int i;
    final int j;
  }
  ```

- Generic type declarations are modified. For example, if you change:

  ```
  class A<T> {
    T i;
  }
  ```

  to:

  ```
  class A<T, V> {
    T i;
    V v;
  }
  ```

In these situations, hot reload generates a diagnostic message and fails without committing any changes.

How it works

When hot reload is invoked, the host machine looks at the edited code since the last compilation. The following libraries are recompiled:

- Any libraries with changed code
- The application's main library
- The libraries from the main library leading to affected libraries

In Dart 2, those libraries' Dart source code are turned into kernel files and sent to the mobile device's Dart VM.

The Dart VM re-loads all libraries from the new kernel file. So far no code is re-executed.

The hot reload mechanism then causes the Flutter framework to trigger a rebuild/re-layout/repaint of all existing widgets and ren objects.