Get started
Samples & tutorials
Development
- User interface
- Data & backend
- Accessibility & internationalization
- Platform integration
- Packages & plugins
- Add Flutter to existing app
- Tools & techniques
  - Android Studio & IntelliJ
  - Visual Studio Code
  - DevTools
  - Flutter SDK
  - Hot reload
  - Code formatting
- Migration notes

Testing & debugging
Performance & optimization
Deployment
Resources
Reference
- Widget index
- API reference
- Package site

Visual Studio Code

Contents

Android Studio and IntelliJ   Visual Studio Code

Installation and setup

Follow the Set up an editor instructions to install the Dart and Flutter extensions (also called plugins).

Updating the extension

Updates to the extensions are shipped on a regular basis. By default, VS Code automatically updates extensions when updates a
available.

To install updates manually:

1. Click the Extensions button in the Side Bar.
2. If the Flutter extension is shown with an available update, click the update button and then the reload button.
3. Restart VS Code.

Creating projects

There are a couple ways to create a new project.

Creating a new project

To create a new Flutter project from the Flutter starter app template:

1. Open the Command Palette (`Ctrl`+`Shift`+`P` (`Cmd`+`Shift`+`P` on macOS)).
2. Select the **Flutter: New Project** command and press `Enter`.
3. Enter your desired **Project name**.
4. Select a **Project location**.

Opening a project from existing source code

To open an existing Flutter project:

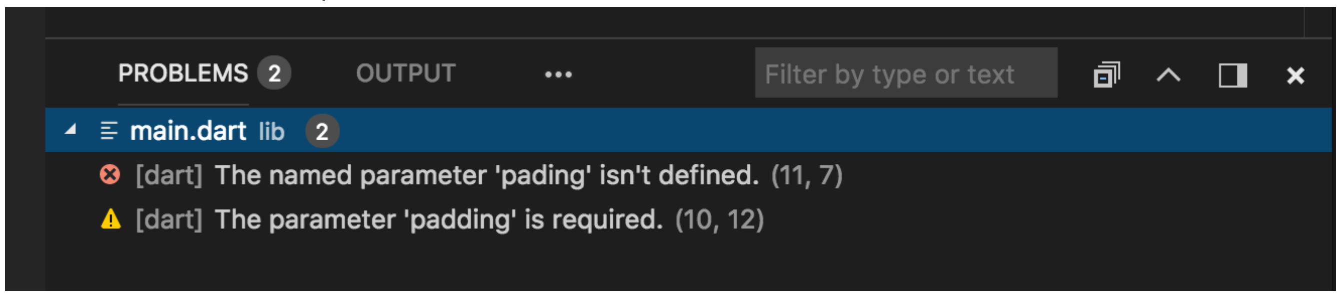
1. Click **File > Open** from the main IDE window.
2. Browse to the directory holding your existing Flutter source code files.

3. Click **Open**.

Editing code and viewing issues

The Flutter extension performs code analysis that enables the following:

- Syntax highlighting
- Code completions based on rich type analysis
- Navigating to type declarations (**Go to Definition** or `F12`), and finding type usages (**Find All References** or `Shift`+`F12`)
- Viewing all current source code problems (**View > Problems** or `Ctrl`+`Shift`+`M` (`Cmd`+`Shift`+`M` on macOS)) Any analysis issue are shown in the Problems pane:



```
[dart] The named parameter 'pading' isn't defined. (11, 7)
[dart] The parameter 'padding' is required. (10, 12)
```

Running and debugging

Note: You can debug your app in a couple of ways.

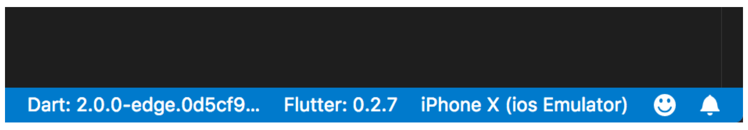
- Using DevTools, a suite of debugging and profiling tools that run in a browser. DevTools replaces the previous browser-based profiling tool, Observatory, and includes functionality previously only available to Android Studio and IntelliJ, such as the Flutter inspector.
- Using VS Code's built-in debugging features, such as setting breakpoints.

The instructions below describe features available in VS Code. For information on using launching DevTools, see Running DevTools from VS Code in the DevTools docs.

Start debugging by clicking **Debug > Start Debugging** from the main IDE window, or press `F5`.

Selecting a target device

When a Flutter project is open in VS Code, you should see a set of Flutter specific entries in the status bar, including a Flutter SDK version and a device name (or the message **No Devices**).



Note:

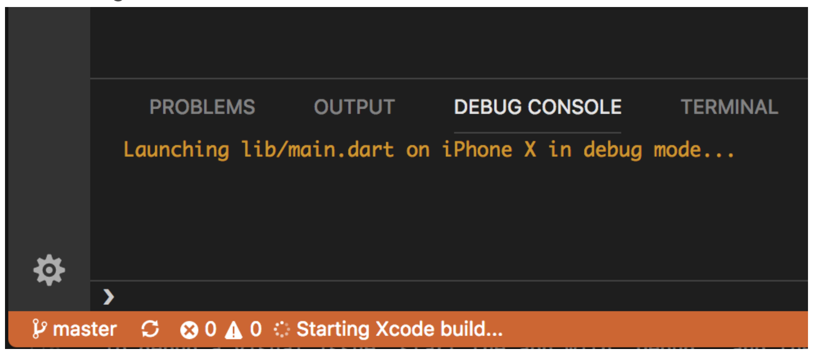
- If you do not see a Flutter version number or device info, your project might not have been detected as a Flutter project. Ensure that the folder that contains your `pubspec.yaml` is inside a VS Code **Workspace Folder**.
- If the status bar reads **No Devices**, Flutter has not been able to discover any connected iOS or Android devices or simulators. You need to connect a device, or start a simulator or emulator, to proceed.

The Flutter extension automatically selects the last device connected. However, if you have multiple devices/simulators connecte click **device** in the status bar to see a pick-list at the top of the screen. Select the device you want to use for running or debugging

Note: If you want to try running your app on the web, but the **Chrome (web)** target doesn't appear in the list of targets, make sure you've enabled web, as described in Building a web application.

Run app without breakpoints

1. Click **Debug > Start Without Debugging** in the main IDE window, or press `Ctrl`+`F5`. The status bar turns orange to show you in a debug session.



```
Launching lib/main.dart on iPhone X in debug mode...
```

Run app with breakpoints

1. If desired, set breakpoints in your source code.
2. Click **Debug > Start Debugging** in the main IDE window, or press `F5`.
   - The left **Debug Sidebar** shows stack frames and variables.
   - The bottom **Debug Console** pane shows detailed logging output.
   - Debugging is based on a default launch configuration. To customize, click the cog at the top of the **Debug Sidebar** to create a `launch.json` file. You can then modify the values.

Fast edit and refresh development cycle

Flutter offers a best-in-class developer cycle enabling you to see the effect of your changes almost instantly with the *Stateful Hot Reload* feature. See Using hot reload for details.

Advanced debugging

Debugging visual layout issues

During a debug session, several additional debugging commands are added to the Command Palette and to the Flutter inspector When space is limited, the icon is used as the visual version of the label.

Toggle Baseline Painting T
Causes each RenderBox to paint a line at each of its baselines.

Toggle Repaint Rainbow
Shows rotating colors on layers when repainting.

Toggle Slow Animations
Slows down animations to enable visual inspection.

Toggle Debug Mode Banner
Hides the debug mode banner even when running a debug build.

Debugging external libraries

By default, debugging an external library is disabled in the Flutter extension. To enable:

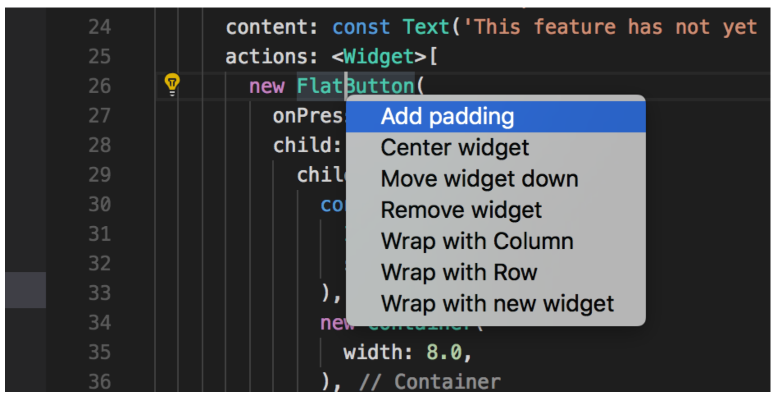
1. Select **Settings > Extensions > Dart Configuration**.
2. Check the `Debug External Libraries` option.

Editing tips for Flutter code

If you have additional tips we should share, let us know!

Assists & quick fixes

Assists are code changes related to a certain code identifier. A number of these are available when the cursor is placed on a Flutt widget identifier, as indicated by the yellow lightbulb icon. The assist can be invoked by clicking the lightbulb, or by using the keyboard shortcut `Ctrl+.` (`Cmd+.` on Mac), as illustrated here:


```
      content: const Text('This feature has not yet
      actions: <Widget>[
        new FlatButton(
          onPres
          child:
            chil
              co


            ),
          ne
            width: 8.0,
          ), // Container
```

Quick fixes are similar, only they are shown with a piece of code has an error and they can assist in correcting it.

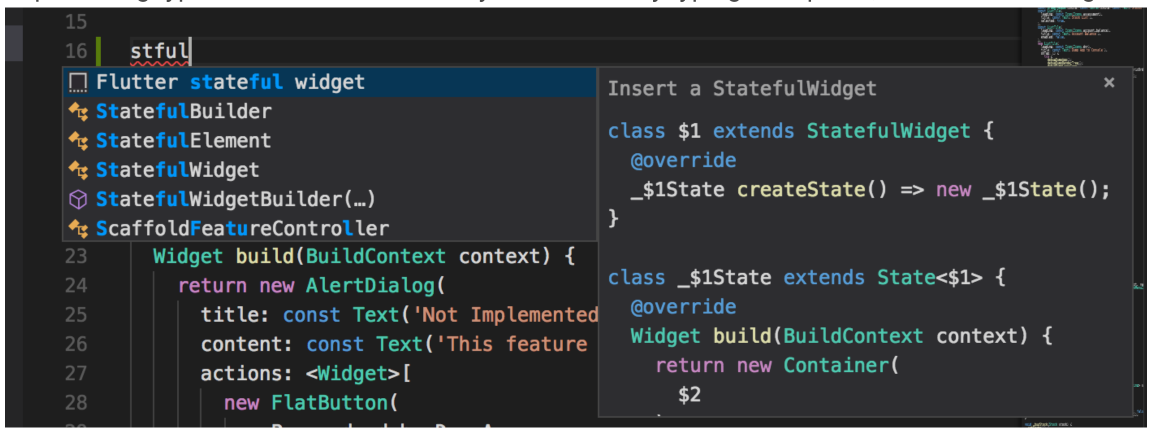
Wrap with new widget assist
This can be used when you have a widget that you want to wrap in a surrounding widget, for example if you want to wrap a widge a `Row` or `Column`.

Wrap widget list with new widget assist
Similar to the assist above, but for wrapping an existing list of widgets rather than an individual widget.

Convert child to children assist
Changes a child argument to a children argument, and wraps the argument value in a list.

Snippets

Snippets can be used to speed up entering typical code structures. They are invoked by typing their prefix, and then selecting fron the code completion window:


```
stful
```

```
class $1 extends StatefulWidget {
  @override
  _$1State createState() => new _$1State();
}

class _$1State extends State<$1> {
  @override
  Widget build(BuildContext context) {
    return new Container(
      $2
```

```
  Widget build(BuildContext context) {
    return new AlertDialog(
      title: const Text('Not Implemented
      content: const Text('This feature
      actions: <Widget>[
        new FlatButton(
```

The Flutter extension includes the following snippets:

- Prefix `stless`: Create a new subclass of `StatelessWidget`.
- Prefix `stful`: Create a new subclass of `StatefulWidget` and its associated State subclass.
- Prefix `stanim`: Create a new subclass of `StatefulWidget`, and its associated State subclass including a field initialized with `AnimationController`.

You can also define custom snippets by executing **Configure User Snippets** from the Command Palette.

Keyboard shortcuts

Hot reload
During a debug session, clicking the **Restart** button on the **Debug Toolbar**, or pressing `Ctrl`+`Shift`+`F5` (`Cmd`+`Shift`+`F5` on macOS) performs a hot reload.
Keyboard mappings can be changed by executing the **Open Keyboard Shortcuts** command from the Command Palette.

Hot reload vs. hot restart

Hot reload works by injecting updated source code files into the running Dart VM (Virtual Machine). This includes not only adding new classes, but also adding methods and fields to existing classes, and changing existing functions. A few types of code chang cannot be hot reloaded though:

- Global variable initializers
- Static field initializers
- The `main()` method of the app

For these changes, fully restart your application without having to end your debugging session. To perform a hot restart, run the **Flutter: Hot Restart** command from the Command Palette, or press `Ctrl`+`F5`.

Troubleshooting

Known issues and feedback

All known bugs are tracked in the issue tracker: Dart and Flutter extensions GitHub issue tracker.

We welcome feedback, both on bugs/issues and feature requests. Prior to filing new issues:

- Do a quick search in the issue trackers to see if the issue is already tracked.
- Make sure you are up to date with the most recent version of the plugin.

When filing new issues, include flutter doctor output.

Except as otherwise noted, this work is licensed under a Creative Commons Attribution 4.0 International License, and code samples are licensed under the BSD License.