The `flutter` command-line tool is how developers (or IDEs on behalf of developers) interact with Flutter.

`flutter --help` lists the developer-facing commands that `flutter` supports.

`flutter --help --verbose` lists *all* the commands that `flutter` supports, in particular, it also lists the features that are of use to Flutter contributors.

These include:

- `flutter update-packages`, which downloads all the Dart dependencies for all Dart packages in the Flutter repository.
- `flutter analyze --flutter-repo`, as described on [[Using the Dart analyzer]].

When contributing to Flutter, use `git pull --rebase` or `git rebase upstream/master` rather than `flutter upgrade`.

The `flutter` tool itself is built when you run `flutter` for the first time and each time you run `git pull --rebase` (or `flutter upgrade`, or anything that changes the current commit).

Documentation

Markdown documentation can be found for some commands in flutter/packages/flutter_tools/doc/.

Making changes to the `flutter` tool

If you want to alter and re-test the tool's behavior itself, locally commit your tool change in git and the tool will be rebuilt from Dart sources in `packages/flutter_tools` the next time you run `flutter`. Alternatively, delete the `bin/cache/flutter_tools.snapshot` file. Doing so will force a rebuild of the tool from your local sources the next time you run `flutter`.

The `flutter_tools` tests run inside the Dart command line VM rather than in the flutter shell. To run the tests, navigate to `packages/flutter_tools` and run:

```
../../bin/cache/dart-sdk/bin/pub run test
```

The pre-built flutter tool runs in release mode with the observatory off by default. To enable debugging mode and the observatory on the `flutter` tool, uncomment the `FLUTTER_TOOL_ARGS` line in the `bin/flutter` shell script.

Debugging the `flutter` command-line tool

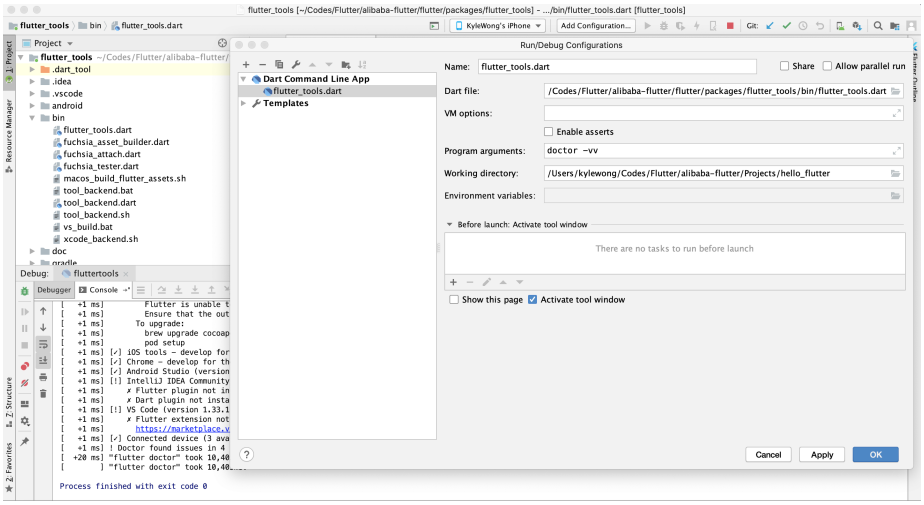
Developers are expected to run flutter command without knowing too much details about the `flutter` command. However, there are some cases in which you may find it useful to debug `flutter` command especially when it's difficult to reproduce your issue.

`flutter` command is just a wrapper and it will finally run `$FLUTTER_ROOT/bin/cache/flutter_tools.snapsho` generated by flutter_tools package.

That's to say, you can debug `flutter` command as a Dart Command Line App. Let's take `flutter doctor -vv` as an example. You can debug it following steps below:

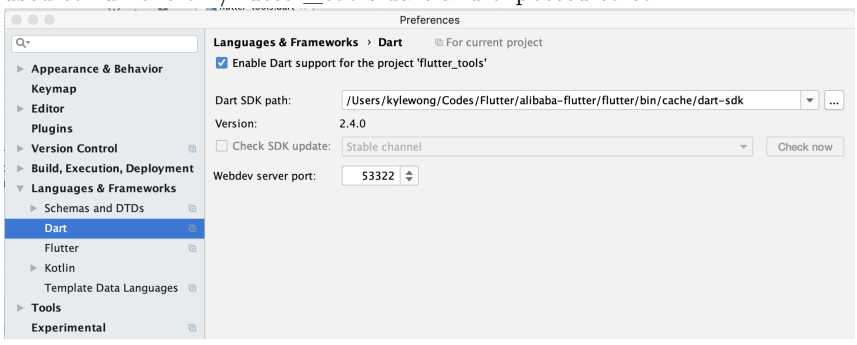
a. Open the flutter_tools package in Android Studio

b. Create a new Dart Command Line App by `Add Configurations` and configure it as below:



The Dart file refers to bin/flutter_tools.dart where the main function is located. Program arguments refers to the arguments for flutter command, it's passed to main method directly. Working directory is which flutter project you want to run the flutter command, and is not always necessary.

c. The dart sdk is used to run the bin/flutter_tools.dart and expected to configure as below:



d. If you make some changes to the flutter_tools package, you may need to do as 'Making changes to the `flutter` tool' says above because flutter command might be triggered implicitly by gradle, etc.

Though those steps given above are under Android Studio, the logic also works for other IDEs.

Adding, removing, or making changes to Dart dependencies

Once you've edited a `pubspec.yaml` file in the Flutter repository to change a package's dependencies, run `flutter update-packages --force-upgrade` to resynchronize all the `pubspec.yaml` files. This does a full cross-package version solve for the entire repository.

If you need to pin a particular version, edit the table at the top of the `update_packages.dart` file.

Using a locally-built engine with the `flutter` tool

To allow the tool to be used with a locally-built engine, the `flutter` tool accepts two global parameters: `local-engine-src-path`, which specifies the path to your engine repository, and `local-engine`, which specifies which build of the engine to use.

A typical invocation would be: `--local-engine-src-path /path/to/engine/src --local-engine=android_debug_unopt`.

If your engine is in a directory called `engine` that is a peer to the framework repository's `flutter` directory, then you can omit `--local-engine-src-path` and only specify `--local-engine`.

You can also set the environment variable `$FLUTTER_ENGINE` instead of specifying `--local-engine-src-path`.

The `--local-engine` should specify the build of the engine to use, e.g. a profile build for Android, a debug build for Android, or whatever. It must match the other arguments provided to the tool, e.g. don't use the `android_debug_unopt` build when you specify `--release`, since the Debug build expects to compile and run Dart code in a JIT environment, while `--release` implies a Release build which uses AOT compilation.

Additionally if you've modified the Dart sources in your engine, you will need to add a `dependency_overrides` section pointing to your modified `package:sky_engine` to the `pubspec.yaml` for the flutter app you're using the custom engine with. A typical example would be:

```
dependency_overrides:
  sky_engine:
    path: /path/to/flutter/engine/out/host_debug/gen/dart-pkg/sky_engine
```

Replace `host_debug` with the actual build that you want to use (similar to `--local-engine`, but typically a host build rather than a device build).

If you do this, you can omit `--local-engine-src-path` and not bother to set `$FLUTTER_ENGINE`, as the `flutter` tool will use these paths to determine the engine also! The tool tries really hard to figure out where your local build of the engine is if you specify `--local-engine`.

Adding dependencies to the Flutter Tool

Each dependency we add to Flutter and the Flutter Tool makes the repo more difficult to update and requires additional work from our clients to update.

Only packages which are developed by the Dart and/or Flutter teams should be permitted into the Flutter Tool. Any third party packages that are currently in use are exempt for historical reasons, but their versions must be pinned in update_packages.dart . These packages should only be updated after a human review of the new version. If a Dart and/or Flutter team package depends transitively on an un-maintained or unknown package, we should work with the owners to remove or replace that transitive dependency.

Instead of adding a new package, ask yourself the following questions:

- Does the functionality already exist in the SDK or an already depended on package?
- Could I develop the same functionality myself in a few hours of work?
- Is the package actively developed and maintained by a trusted party?