Flutter DeviceLab

DeviceLab is a physical lab that tests Flutter on real devices.

This package contains the code for the test framework and tests. More generally the tests are referred to as “tasks” in the API, but since we primarily use it for testing, this document refers to them as “tests”.

Current statuses for the devicelab are available at https://flutter-dashboard.appspot.com/#/build. See dashboard user guide for information on using the dashboards.

Table of Contents

How the DeviceLab runs tests

DeviceLab tests are run against physical devices in Flutter’s lab (the “DeviceLab”).

Tasks specify the type of device they are to run on (`linux_android`, `mac_ios`, `mac_android`, `windows_android`, etc). When a device in the lab is free, it will pickup tasks that need to be completed.

1. If the task succeeds, the test runner reports the success and uploads its performance metrics to Flutter’s infrastructure. Not all tasks record performance metrics.
2. If task fails, an auto rerun happens. Whenever the last run succeeds, the task will be reported as a success. For this case, a flake will be flagged and populated to the test result.
3. If the task fails in all reruns, the test runner reports the failure to Flutter’s infrastructure and no performance metrics are collected

Running tests locally

Do make sure your tests pass locally before deploying to the CI environment. Below is a handful of commands that run tests in a similar way to how the CI environment runs them. These commands are also useful when you need to reproduce a CI test failure locally.

Prerequisites

You must set the `ANDROID_SDK_ROOT` environment variable to run tests on Android. If you have a local build of the Flutter engine, then you have a copy of the Android SDK at `.../engine/src/third_party/android_tools/sdk`.

You can find where your Android SDK is using `flutter doctor`.

Warnings

Running the devicelab will do things to your environment.

Notably, it will start and stop Gradle, for instance.

Running specific tests

To run a test, use option `-t` (`--task`):

```
# from the .../flutter/dev/devicelab directory
../../bin/cache/dart-sdk/bin/dart bin/test_runner.dart test -t {NAME_OR_PATH_OF_TEST}
```

Where `NAME_OR_PATH_OF_TEST` can be either of:

- the *name* of a task, which is a file's basename in `bin/tasks`. Example: `complex_layout__start_up`.
- the path to a Dart *file* corresponding to a task, which resides in `bin/tasks`. Tip: most shells support path auto-completion using the Tab key. Example: `bin/tasks/complex_layout__start_up.dart`.

To run multiple tests, repeat option `-t` (`--task`) multiple times:

```
../../bin/cache/dart-sdk/bin/dart bin/run.dart -t test1 -t test2 -t test3
```

Running tests against a local engine build

To run device lab tests against a local engine build, pass the appropriate flags to `bin/run.dart`:

```
../../bin/cache/dart-sdk/bin/dart bin/run.dart --task=[some_task] \
  --local-engine-src-path=[path_to_local]/engine/src \
  --local-engine=[local_engine_architecture]
```

An example of a local engine architecture is `android_debug_unopt_x86`.

Running an A/B test for engine changes

You can run an A/B test that compares the performance of the default engine against a local engine build. The test runs the same benchmark a specified number of times against both engines, then outputs a tab-separated spreadsheet with the results and stores them in a JSON file for future reference. The results can be copied to a Google Spreadsheet for further inspection and the JSON file can be reprocessed with the `summarize.dart` command for more detailed output.

Example:

```
../../bin/cache/dart-sdk/bin/dart bin/run.dart --ab=10 \
  --local-engine=host_debug_unopt \
  -t bin/tasks/web_benchmarks_canvaskit.dart
```

The `--ab=10` tells the runner to run an A/B test 10 times.

`--local-engine=host_debug_unopt` tells the A/B test to use the `host_debug_unopt` engine build. `--local-engine` is required for A/B test.

`--ab-result-file=filename` can be used to provide an alternate location to output the JSON results file (defaults to `ABresults#.json`). A single `#` character can be used to indicate where to insert a serial number if a file with that name already exists, otherwise, the file will be overwritten.

A/B can run exactly one task. Multiple tasks are not supported.

Example output:

```
Score   Average A (noise)   Average B (noise)   Speed-up
bench_card_infinite_scroll.canvaskit.drawFrameDuration.average  2900.20 (8.44%) 2426.70 (8.9
bench_card_infinite_scroll.canvaskit.totalUiFrame.average   4964.00 (6.29%) 4098.00 (8.03%)
draw_rect.canvaskit.windowRenderDuration.average    1959.45 (16.56%)    2286.65 (0.61%) 0.86
draw_rect.canvaskit.sceneBuildDuration.average  1969.45 (16.37%)    2294.90 (0.58%) 0.86x
draw_rect.canvaskit.drawFrameDuration.average   5335.20 (17.59%)    6437.60 (0.59%) 0.83x
draw_rect.canvaskit.totalUiFrame.average    6832.00 (13.16%)    7932.00 (0.34%) 0.86x
```

The output contains averages and noises for each score. More importantly, it contains the speed-up value, i.e. how much *faster* is the local engine than the default engine. Values less than 1.0 indicate a slow-down. For example, 0.5x means the local engine is twice as slow as the default engine, and 2.0x means it's twice as fast. Higher is better.

Summarize tool example:

```
../../bin/cache/dart-sdk/bin/dart bin/summarize.dart  --[no-]tsv-table --[no-]raw-summary \
    ABresults.json ABresults1.json ABresults2.json ...
```

`--[no-]tsv-table` tells the tool to print the summary in a table with tabs for easy spreadsheet entry. (defaults to on)

`--[no-]raw-summary` tells the tool to print all per-run data collected by the A/B test formatted with tabs for easy spreadsheet entry. (defaults to on)

Multiple trailing filenames can be specified and each such results file will be processed in turn.

Reproducing broken builds locally

To reproduce the breakage locally `git checkout` the corresponding Flutter revision. Note the name of the test that failed. In the example above the failing test is `flutter_gallery__transition_perf`. This name can be passed to the `run.dart` command. For example:

```
../../bin/cache/dart-sdk/bin/dart bin/run.dart -t flutter_gallery__transition_perf
```

Writing tests

A test is a simple Dart program that lives under `bin/tasks` and uses `package:flutter_devicelab/framework/framework.dart` to define and run a *task*.

Example:

```
import 'dart:async';

import 'package:flutter_devicelab/framework/framework.dart';

Future<void> main() async {
  await task(() async {
    ... do something interesting ...

    // Aggregate results into a JSONable Map structure.
    Map<String, dynamic> testResults = ...;

    // Report success.
    return new TaskResult.success(testResults);

    // Or you can also report a failure.
    return new TaskResult.failure('Something went wrong!');
  });
}
```

Only one `task` is permitted per program. However, that task can run any number of tests internally. A task has a name. It succeeds and fails independently of other tasks, and is reported to the dashboard independently of other tasks.

A task runs in its own standalone Dart VM and reports results via Dart VM service protocol. This ensures that tasks do not interfere with each other and lets the CI system time out and clean up tasks that get stuck.

Adding tests to continuous integration

Host only tests should be added to `flutter_tools`.

There are several PRs needed to add a DeviceLab task to CI.

TASK- the name of your test that also matches the name of the file in `bin/tasks` without the `.dart` extension.

1. Add target to .ci.yaml
    - Mirror an existing one that has the recipe `devicelab_drone`

If your test needs to run on multiple operating systems, create a separate target for each operating system.

Adding tests to presubmit

Flutter's DeviceLab has a limited capacity in presubmit. File an infra ticket to investigate feasibility of adding a test to presubmit.