Testing the engine

Pull requests submitted to the engine repository should be tested to prevent functional regressions.

This guide describes how to write and run various types of tests in the engine.

C++ - core engine

If you edit `.cc` files in https://github.com/flutter/engine/tree/master, you're working on the core, portable Flutter engine.

Unit tests

C++ unit tests are co-located with their header and source files. For instance, `fml/file.h` and `fml/file.cc` have a `fml/file_unittest.cc` in the same directory. When editing C++ files, look for its `_unittest.cc` sibling or create one if there isn't one present.

The engine repo has a unified build system to build C, C++, Objective-C, Objective-C++, and Java files using GN and Ninja. Individual `_unittest.cc` files are referenced by the `BUILD.gn` build rule located in the folder or in an ancestor folder.

You can run the C++ unit tests with:

```
testing/run_tests.py --type=engine
```

from the `flutter` directory, after building the engine variant to test (by default `host_debug_unopt`).

Behind the scenes, those tests in the same directory are built together as a testonly executable when you build the engine variant. The `run_tests.py` script executes them one by one.

C++ unit tests are executed during pre-submit on our CI system when submitting PRs to the `flutter/engine` repository.

Google Tests

C++ unit tests in the core engine uses the Google Test C++ testing framework to facilitate C++ test discovery, assertions, etc.

Since the engine is portable, these unit tests are compiled and run directly on and for your host machine architecture.

It's best practice to test only one real production class per test and create mocks for all other dependencies.

Java - Android embedding

If you edit `.java` files in the https://github.com/flutter/engine/tree/master/shell/platform/android directory, you're working on the Android embedding which connects the core C++ engine to the Android SDK APIs and runtime.

Robolectric JUnit tests

For testing logic within a class at a unit level, create or add to a JUnit test.

Existing Java unit tests are located at https://github.com/flutter/engine/tree/master/shell/platform/android/test and follow the Java package directory structure. Files in the `shell/platform/android/io/flutter/` package tree can have a parallel file in the `shell/platform/android/test/io/flutter/` package tree. Files in matching directories are considered package visible as is the case in standard Java.

When editing production files in `shell/platform/android/io/flutter/`, the easiest step to add tests is to look for a matching `...Test.java` file in `shell/platform/android/test/io/flutter/`.

See the Java unit test README for details.

The engine repo has a unified build system to build C, C++, Objective-C, Objective-C++, and Java files using GN and Ninja. Because it doesn't use the more common Gradle build system (which can't build C++ for instance), the tests and its dependencies can't be directly built and run inside Android Studio like a standard Android project.

Instead, the engine provides the script:

```
testing/run_tests.py --type=java
```

to easily build and run the JUnit tests.

This script only has a limited amount of smartness. If you've never built the engine before, it'll build the test and classes under test with a reasonable default configuration. If you've built the engine before, it'll re-build the engine with the same GN flags. You may want to double check your GN flags (https://github.com/flutter/flutter/wiki/Compiling-the-engine#compiling-for-android-from-macos-or-linux) if you haven't built the engine for a while.

Behind the scenes, it invokes GN and Ninja to build a single .jar file containing the test runner and dependencies. Then it uses the system `java` runtime to execute the .jar. JDK v8 must be set as your `$JAVA_HOME` to run the Robolectric tests.

See [[Setting-up-the-Engine-development-environment#using-vscode-as-an-ide-for-the-android-embedding-java]] for tips on setting up Java code completion and syntax highlighting in Visual Studio when working on the engine and tests.

JUnit tests are executed during pre-submit on our CI system when submitting PRs to the `flutter/engine` repository.

Robolectric

Robolectric is a standard Android testing library to mock the Android runtime. It allows tests to be executed on a lightweight Java JVM without booting a heavy Android runtime in an emulator. This allows for rapid test iterations and allows our tests to run better on CI systems.

All engine JUnit tests are Robolectric tests. This means all `android.*` imports are mocked by Robolectric. If you need to modify how Android components (such as an android.view.View or an android.app.Activity) behave in the test, see other tests for examples or see docs at http://robolectric.org/ on how to interact with shadows.

Mockito

Mockito is also a standard Android testing library used to mock non-Android dependencies needed to construct and test interactions with your your under-test production class.

It's best practice to test only one real production class per test and mock all other dependencies with mockito.

The Mockito library is an available test dependency when writing Robolectric tests.

Component integration tests

Component tests test the interaction of multiple embedding Java classes together but they don't test all production classes end-to-end. In the Android embedding case, we test groups of Java classes by their function in `...ComponentTest.java` files that are also in the `shell/platform/android/test/io/flutter/` directory. C++ engine parts via JNI are not tested here.

Component tests are also Robolectric JUnit tests and are invoked together with unit tests when running:

```
testing/run_tests.py --type=java
```

JUnit component tests are executed during pre-submit on our CI system when submitting PRs to the `flutter/engine` repository.

End-to-end tests

End-to-end tests exercise the entire Android embedding with the C++ engine on a real Android runtime in an emulator. It's an integration test ensuring that the engine as a whole on Android is functioning correctly.

The project containing the Android end-to-end engine test is at https://github.com/flutter/engine/tree/master/testing/scenario_app/android.

This test project is build similarly to a normal Flutter app. The Dart code is compiled into AOT and the Android part is compiled via Gradle with a dependency on the prebuilt local engine. The built app then installed and executed on an emulator.

Unlike a normal Flutter app, the Flutter framework on the Dart side is a lightweight fake at https://github.com/flutter/engine/tree/master/testing/scenario_app/lib that implements some of the basic functionalities of `dart:ui` Window rather than using the real Flutter framework at `flutter/flutter`.

The end-to-end test can be executed by running:

```
testing/scenario_app/build_and_run_android_tests.sh
```

Additional end-to-end instrumented tests can be added to https://github.com/flutter/engine/tree/master/testing/scenario_app/android/app/src/androidTest/java/dev/flutter/scenarios.

If supporting logic is needed for the test case, it can be added to the Android app under-test in https://github.com/flutter/engine/tree/master/testing/scenario_app/android/app/src/main/java/dev/flutter/scenarios or to the fake Flutter framework under-test in https://github.com/flutter/engine/tree/master/testing/scenario_app/lib.

As best practice, favor adding unit tests if possible since instrumented tests are, by nature, non-hermetic, slow and flaky.

End-to-end tests on Android are run on presubmit for flutter/engine PRs.

Objective-C - iOS embedding

If you edit `.h` or `.mm` files in the https://github.com/flutter/engine/tree/master/shell/platform/darwin/ios directory, you're working on the iOS embedding which connects the core C++ engine to the iOS SDK APIs and runtime.

XCTest unit tests

For testing logic within a class in isolation, create or add to a XCTestCase.

The iOS unit testing infrastructure is split in 2 different locations. The `...Test.mm` files in https://github.com/flutter/engine/tree/master/shell/platform/darwin/ios contain the unit tests themselves. The https://github.com/flutter/engine/tree/master/testing/ios/IosUnitTests directory contains an Xcode container project to execute the test.

See the iOS unit test README for details on adding new test files.

The engine repo has a unified build system to build C, C++, Objective-C, Objective-C++, and Java files using GN and Ninja. Since GN and Ninja has to build the C++ dependencies that the Objective-C classes reference, the tests aren't built by the Xcode project in https://github.com/flutter/engine/tree/master/testing/ios/IosUnitTests.

Instead, the engine provides the script:

```
testing/run_tests.py --type=objc
```

to easily build and run the XCTests.

This script only has a limited amount of smartness. If you've never built the engine before, it'll build the test and classes under test with a reasonable default configuration. If you've built the engine before, it'll re-build the engine with the same GN flags. You may want to double check your GN flags (https://github.com/flutter/flutter/wiki/Compiling-the-engine#compiling-for-ios-from-macos) if you haven't built the engine for a while.

Behind the scenes, it invokes GN and Ninja to build the tests and dependencies into a single `.dylib`. Then it uses Xcode and the Xcode project at `testing/ios/IosUnitTests` to import and execute the XCTests in the `.dylib`.

See [[Setting-up-the-Engine-development-environment#editor-autocomplete-support]] for tips on setting up C/C++/Objective-C code completion and syntax highlighting when working on the engine and tests.

To debug the XCTests, you can open the Xcode project at `testing/ios/IosUnitTests/IosUnitTests.xcworkspace` and run the tests (such as via ⌘U). Note you cannot modify the test source and build the tests in Xcode for reasons mentioned above. If you modify the test, you need to run `testing/run_tests.py` again.

XCTests are executed during pre-submit on our CI system when submitting PRs to the `flutter/engine` repository.

XCTest

XCTest is the standard way of creating unit tests in Xcode projects. Since iOS has x86 simulators and since we can build x86 engines, we can execute the XCTests directly on macOS in a headless simulator using the real iOS SDK.

OCMock

OCMock is a standard iOS testing library used to mock dependencies needed to construct and test interactions with your under-test production class.

It's best practice to test only one real production class per test and mock all other dependencies with OCMock.

The OCMock library is available as a test dependency when writing XCTests for the engine.

End-to-end tests

End-to-end tests exercise the entire iOS embedding with the C++ engine on a headless iOS simulator. It's an integration test ensuring that the engine as a whole on iOS is functioning correctly.

The project containing the iOS end-to-end engine test is at https://github.com/flutter/engine/tree/master/testing/scenario_app/ios.

This test project is build similarly to a normal debug Flutter app. The Dart code is bundled in JIT mode and is brought into Xcode with a `.framework` dependency on the prebuilt local engine. It's then installed and executed on a simulator via Xcode.

Unlike a normal Flutter app, the Flutter framework on the Dart side is a lightweight fake at https://github.com/flutter/engine/tree/master/testing/scenario_app/lib that implements some of the basic functionalities of `dart:ui` Window rather than using the real Flutter framework at `flutter/flutter`.

The end-to-end test can be executed by running:

```
testing/scenario_app/build_and_run_ios_tests.sh
```

Additional end-to-end instrumented tests can be added to https://github.com/flutter/engine/tree/master/testing/scenario_app/ios/Scenarios/ScenariosTests.

If supporting logic is needed for the test case, it can be added to the Android app under-test in https://github.com/flutter/engine/tree/master/testing/scenario_app/ios/Scenarios/Scenarios or to the fake Flutter framework under-test in https://github.com/flutter/engine/tree/master/testing/scenario_app/lib.

As best practice, favor adding unit tests if possible since end-to-end tests are, by nature, non-hermetic, slow and flaky.

End-to-end tests on iOS are executed during pre-submit on our CI system when submitting PRs to the `flutter/engine` repository.

Dart - dart:ui

If you edit `.dart` files in https://github.com/flutter/engine/tree/master/lib/ui, you're working on the 'dart:ui' package which is the interface between the C++ engine and the Dart Flutter framework.

Unit tests

Dart classes in https://github.com/flutter/engine/tree/master/lib/ui have matching unit tests at https://github.com/flutter/engine/tree/master/testing/dart.

When editing production files in the 'dart:ui' package, add to or create a test file in `testing/dart`.

To run the Dart unit tests, use the script:

```
testing/run_tests.py --type=dart
```

Behind the scenes, it invokes the engine repo's unified GN and Ninja build systems to use a version of the Dart SDK specified in the `DEPS` file to create a `sky_engine` Dart package. Then it compiles and runs each `_test.dart` file under `testing/dart`.

Dart unit tests are executed during pre-submit on our CI system when submitting PRs to the `flutter/engine` repository.

Framework tests

Dart tests in the `flutter/flutter` framework repo are also executed on top of the `dart:ui` package and underlying engine.

These tests are executed during pre-submit on our CI system when submitting PRs to the `flutter/engine` repository.

Assuming your `flutter` and `engine` working directories are siblings, you can run the framework tests locally using the following command from the root of your `flutter` repository:

```
(cd packages/flutter; ../../bin/flutter test --local-engine=host_debug_unopt)
```

Web engine

Web tests are run via the `felt` command. More details can be found in lib/web_ui/README.md.