What’s Up With Tests

With Special Guest Stephen

Intro

Today we’re sitting down with Stephen, who previously led Chrome’s involvement in web platform tests, to talk about testing, a huge topic and also an important part of development.
Slack channels for helpful resource: #halp, #wpt, #debugging

Why test?

Firstly, correctness. We always want to ensure our code is working correctly.
Second, testing can serve as documentation. It shows how the code should work. A great way to understand how a piece of code works is to add tests! Other reasons include security, performance, etc.

Chrome’s test types

We have many different types of tests - unit tests, browser tests, web tests, performance tests, tests on Android, manual tests, etc!

Unit tests

These test only one 'unit' of code, e.g. a class, a method, one small isolated code snippet that doesn’t have to talk to the other parts of the code base to work. In Chromium we use a unit test framework called GoogleTest, which is very commonly used in C++ testing.

A typical test suit may look like this:

TEST_P(PendingBeaconHostTest, SendOneOfBeacons)

The first part, TEST_P is one of the three macros that indicate what type of unit test it is.
The three macros are:

- **TEST:** indicates the following content is a testing function.
- **TEST_F**: a testing fixture (wrapper) class that helps set up testing specifications. Fixture classes are derived from a base testing class, testing::Test
- **TEST_P**: a parameterized test. It takes multiple parameters but by setting different values for these parameters avoids using multiple testing methods with these values. Useful if we want to test a new flag to see what would happen with this flag on and off. All parameterized tests have fixture classes.

The second part, **PendingBeaconHostTest,** is the name of the test suite, and in the case of TEST_F is also the name of the container class (fixture class). The third part, **SendOneOfBeacons,** is the test name.

Browser tests

Integration testing. Multiple components are combined and tested at once. It more or less launches the whole chrome (or content_shell, or…) application. A typical browser test looks like this:

IN_PROC_BROWSER_TEST_P(RenderFrameHostManagerTest, ProcessReuseVsBrowsingInstance)

IN_PROC means to run this test in the current process.

base::RunLoop is designed for testing asynchronous code. For instance if a testing function has a callback, it's possible the testing function ends before the callback could run. We use run-loop to "cache" this callback and run-loop keeps running what it has until it's idle. Thus we ensure the asynchronous code is tested properly.

Web tests

End-to-end tests. Web tests bring up the whole browser and then test HTML, Javascript and CSS. They are largely split into two types - javascript tests and rendering tests (aka ref tests). Web tests used to be called "layout tests", and also have web platform tests as a subset. The blink_test target should be built to run these tests.

> Fun fact: inside chromium we don't run the web platform tests using the chrome browser. Instead we use content shell. This is partially historical, and partially due to feasibility. Content shell is much simpler than the Chrome browser. If we are running tests it's faster, more stable and reliable.

Performance tests

We measure performance as a success metric of chrome. Performance tests bring up a chrome browser, launch some page actions against the browser and log the related performance metrics.

Android tests

Android is a major part of our code base and chrome is a cross-platform product. We already have a huge amount of testing framework built around android, allowing for unit tests, browser tests, etc to be written in Java and run on Android devices (or simulators).

Manual testing

Some testing engineers test your code/application in a real production environment. An example is to test accessibility.

Content_shell vs. Headless Mode.
Content_shell does have a small browser UI popping up when you test chrome components, while headless mode has no UI at all; it brings all modern web platform features provided by Chromium and the Blink rendering engine to the command line.

How do we run tests

Different binaries files (e.g. component_unittests, content_unittests, unit_tests) are built for different tests using command

autoninja -C out/default content_unittests

Then you simply run them, using **./out/default/content_unittests**.

You can pass in --**gtest_filter = *foo*** flag to only run the tests of interest to you and thus avoid running all the tests in that target. A flag like --**gtest_repeat = 100** lets you run a test multiple times, which is useful when dealing with flaky tests.

Some pre-compile conditions are used to prevent flaky tests. Flaky tests are those that don’t give consistent results on different platforms. MAYBE_ or DISABLED_ are put in front of tests to let GoogleTest know how to deal with this test.

Make your code more testable

Flakiness. There are many factors that cause flakiness, the most common being timing. We have many integration tests and timing has a direct impact on how the integration tests behave. E.g. You run some tests on your developer machine and they all pass but the committed code may get tested on a machine that has different configurations which cause two of the functions in the code run in different order and thus yield different results. One way to avoid this is to write more unit tests rather than browser tests.

A tip to make your code more testable is be careful of the thing you’re using. E.g. When you rely on a webContents to test, remember to create a mock one and use it as input instead of calling GetCurrentWebContents() and trying to get a real webcontents.