Firefox Test Engineering Documentation

Release 0.1alpha

Firefox Test Engineering

Mar 22, 2018

Contents

Hello! Welcome to Firefox Test Engineering, an informal introduction to how to work with our group.

If you're a new contributor or new employee, you can start out by reading the *New Contributor Guide*. If you're already a contributor, you can find generally useful info, such as style guides, in the *Reference*.

The original template used for this guide is based off of the excellent Mozilla WebDev bootcamp. It is maintained in a GitHub repository. Feel free to send pull requests if you see something that could be improved.

CHAPTER 1

New Contributor Guide

First things first: Welcome to Mozilla! We're glad to have you here, whether you're considering volunteering as a contributor or are employed by Mozilla.

Our goal here is to get you up and running to contribute as a test engineer. This guide attempts to be generic enough to be useful to most Firefox Test Engineering projects, but some details (especially around software you need installed) might vary among projects.

Baseline requirements include:

- some coding experience, Python and Selenium WebDriver are used most often
- a basic understanding of GitHub
- a willingness to learn!

Let's get started!

1.1 About the Firefox Test Engineering team

The Firefox Test Engineering team oversees the test automation, infrastructure, and manual testing for all of Firefox's services, sites, and apps that are external to the core browsers (and in some cases within the browsers as well). Our mission is to provide testing and tools to positively impact the quality of Mozilla websites and services.

1.1.1 How to Talk to Us

There are a few channels of communication for our group:

- The #fx-test channel on irc.mozilla.org.
- The firefox-test-engineering@mozilla.com Google Group/mailing list.
- More details about our team can be found on the Firefox Test Engineering page on QMO.

If you ever have a question, regardless of how difficult it is, you can share it on those channels and someone should help you out. *Don't be afraid to ask questions!* You're trying to help us, so it's only fair that we try to help you in return.

1.1.2 Find a Mentor!

It's also a good idea to find someone to mentor you as a new contributor. Having someone you can personally ask for help from is incredibly helpful in finding your way around.

If you're a new employee, your mentor may be your manager, or it may be a coworker. If you're a volunteer, ask someone who works on the project you want to contribute to; if they cannot mentor you themselves, they should be able to direct you to someone who can.

1.1.3 Turn on the Firehose

We are one team, part of a much bigger Mozilla project. Here are some links:

- Newsgroup related to tools at Mozilla: mozilla.tools
- Information on testing: mozilla.dev.quality
- Project wide notices and meeting announcements: mozilla.dev.planning
- Technical discussions regarding the Gecko Platform: mozilla.dev.platform
- Keep abreast of the latest status on the entire project: Planet Mozilla

1.2 Finding a Project

Before you can contribute to Mozilla, you need to find a project that you're interested in contributing to. We currently maintain a list of projects on Service Book.

You may also browse all of the open GitHub Issues for our projects. The Firefox Test Engineering Dashboard lists open Issues, just make sure to find one that no one else is working on.

1.2.1 Getting set up

Once you've identified the project you want to work on, you should get the test automation running locally. All projects have (or should have) a README file in their source tree that either describes this process or links to documentation that does. If you haven't already, you will want to review the New Contributor Guide.

1.2.2 Find a mentor

You may also find it useful to find someone who is working on or responsible for the project you want to contribute to and asking if they can help you find a task to work on and answer any other questions you have. Generally speaking, the project page should have this information. If it doesn't, try looking through the commit log of the source code to see who has been writing code for the project recently.

1.2.3 How to contribute

Once you're set up to work on a project, you'll have to find a task to work on and get coding! Each project should have some information on where their tasks are tracked, generally on Bugzilla or GitHub Issues which are listed on the Firefox Test Engineering Dashboard. The first step is to find a bug or an Issue that is open for contribution, and then adding a comment that you plan to work on it.

1.3 Test Automation Process

While the details vary, there is a general framework for working on Firefox Test Engineering test automation projects. This document attempts to describe that process.

1.3.1 Finding a Bug or GitHub Issue

The first step is finding a good bug or GitHub Issue to work on.

- GitHub Issues are written when new tests need to be written, or changes need to be made to fix existing flaky or failing tests. They include a description of the steps and expected results. You can find a comprehensive list of open GitHub Issues for our team on the Firefox Test Engineering Dashboard.
- Unclaimed Issues are open and available for anyone to work on. You will know if it is unclaimed by reading the comments and checking if it is already assigned to someone. Make sure to comment on the Issue when you find one that you plan to complete.
- Bugs are written when specific features, tests or fixes are to be implemented. Bugs that are marked as 'mentored' are a good place to start as you will have support while learning the project.
- Finding a mentored bug that fits your interests and skill set can be challenging. We recommend going to Bugs Ahoy to find mentored bugs, filtered by skills or areas of interest. Know that listed bugs will include all Mozilla projects, not just ones within the Firefox Test Engineering team.
- Commenting on a bug is generally a good way to indicate you will work on it. However, some teams prefer that you submit a pull request rather than a comment. Contact the bug author if you have questions.

1.3.2 Working on the Bug/Issue

After claiming a Bug/Issue, you will submit your pull request with your work in a GitHub feature branch.

Any mentors or project owners assigned to it will review your work and give constructive feedback and instructions for any changes that need to be made. After the pull request is completed to satisfaction it will be merged into the project code.

The Bug/Issue will be resolved and closed after a successful code merge.

Git and GitHub

For projects using Git and GitHub, the process can be explained in more detail:

- On GitHub, ensure you have forked the repository for your project to your own account and have added it as a remote to your repository.
- Identify the main development branch for your project. This is usually the `master` branch.
- Make sure the current branch is the development branch, and create a new branch off of it for your feature.

- Start running test automation to get familiar with the project.
- Once your work is committed and ready for review, push the branch to your fork on GitHub and submit a pull request.
- If the project uses Bugzilla for issue tracking, create an attachment to your issue's bug pointing at the pull request. Otherwise, if you know who should review your change, add a comment to your pull request with their `@Username` in it and ask for a review.

See also:

Python Style Guide A useful styleguide for Python best practices.

Git and GitHub A short list of tips and tricks for using GitHub.

GitHub Flow A process for branching, reviewing, and merging code that is very similar to the process above.

1.3.3 Mobile

Mobile platforms such as iOS and Android are also an important part of our testing process. Testing is done on mobile platforms, and there are also mobile-specific tests.

iOS

Currently we use Apple's XCUITest framework in Swift. The test can be written and executed on Apple's XCode App, and you can see some of our examples in the Firefox for iOS repository. Some of the older Tests are in KIFTest framework, but since they use undocumented Apple APIs, and XCUITest framework has been maturing, we are trying to create new tests in XCUITest framework.

You can learn more about basics of Swift and XCUITest from below websites:

- Apple Developer Site.
- Swift Language Reference.
- UI Testing Cheat Sheet and Examples.

Android

A few of our Android mobile test-automation projects (particularly those written with Selenium WebDriver) use Appium. For Firefox for Android testing automation that does not involve the testing of GeckoView, there is a proof-of-concept test environment using Appium.

1.3.4 Next steps

At this point you should have all the information and tools you need to make your first contribution to Mozilla! Once you've submitted your work and gotten it merged, it's time to celebrate: you've earned it!

As you continue to contribute, you may want to check out the *Reference* to find generally useful information for contributors of all levels.

Good luck!

1.4 Accounts

We use a few websites/services/applications to manage our test-automation and infrastructure code, bugs, test cases, etc., and while it is possible to get by without signing up for these sites, it's *strongly* recommended that you create accounts on these websites.

1.4.1 GitHub

Many Firefox Test Engineering projects are hosted on GitHub. GitHub provides hosting for our source code, as well as tools we use for collaboration and code review. GitHub is based on Git, a distributed version control system that lets us track the changes we make to our code.

Once you've created a GitHub account, you can check out the GitHub help site for guides on the basics of using Git and GitHub.

See also:

Mozilla Services on GitHub Mozilla Services' organization account on GitHub.

Mozilla on GitHub Mozilla's organization account on GitHub.

1.4.2 Bugzilla

Bugzilla is the issue-tracking system that the entire Mozilla Project uses. Many of our projects use Bugzilla to keep track of any planned changes to or bugs in our projects.

As a new contributor, Bugzilla is a useful tool for finding known issues that you can help fix or finding planned work you want to take on. In order to assign a bug to yourself or to post a comment on a bug, you'll need to create a Bugzilla account. An account also allows you to "CC" yourself on bugs that you are interested in, so that you receive emails when those bugs are changed.

After you've been using Bugzilla for a while as a community member, it's worthwhile applying for expanded permissions. The editbugs permission allows you to assign bugs to yourself and resolve them, for example. See the Bugzilla Permissions Page for details. Note that new employees get this permission automatically; there's no need to ask for it.

Note: It's highly recommended to add your IRC nickname to your real name within Bugzilla to make it easy for others to auto-complete your name.

The standard format is to follow your real name with your IRC name, preceeded by a colon, surrounded by square brackets. For example: `Cave Johnson [:withthelemons]`.

1.5 Limited Access Accounts

Our team also uses a number of services which are accessible only with special accounts and/or permissions. If you're under an NDA you may ask your mentor or manager for access.

1.5.1 fx-test-pubkeys

Used to keep our public keys, which we use for access control/authentication in our AWS' EC2 environments/instances. Please follow the instructions at https://github.com/mozilla-services/fx-test-pubkeys to generate and upload yours.

1.5.2 Amazon Web Services (AWS)

AWS is currently used for much of our server and infrastructure testing. In the future it will likely hold a lot more of our test automation.

You will very likely, at some point (sooner, rather than later) need a Mozilla-privileged AWS account. Please follow these instructions on Mana to get one.

1.5.3 LastPass

We use LastPass to securely share miscellaneous credentials (usernames/passwords/API keys). Please follow the instructions https://mana.mozilla.org/wiki/display/TestEngineering/LastPass to request and use an account with your Mozilla email address.

1.5.4 Jenkins

Jenkins is used for running our test automation. It is in the process of being moved into AWS.

1.5.5 TestRail

TestRail is being set up for use as a test case manager. In the future it will be more accessible and visible to community members.

1.6 Software and Tools

The software you'll need to download and install on your computer in order to contribute varies between projects; please refer to the documentation for the project you want to contribute to for details.

The following information is a generic description of software or tools that you'll most likely need regardless of the project you work on.

1.6.1 Operating Systems: Windows, Linux, or macOS/OS X?

Generally speaking, automation and test tools need to run on the platforms that are being tested. Linux and Mac are often used as development environments because the tooling is much more comprehensive. If you are a Windows user, you may want to use a program like VirtualBox to create a virtual machine running a Linux-based operating system. The rest of this guide assumes you are using macOS/OS X or a Linux-based operating system.

If you are running macOS/OS X, most of the software mentioned here can be installed using the Homebrew package manager.

1.6.2 Git

Git is a distributed version control system. It tracks the history of changes we make to our code, which allows us to see how the code has changed over time. Git also makes it very easy for multiple people to work on the same code at the same time and merge their changes together at the end.

If you are a contributor who is completely new to distributed version control systems, you might enjoy stepping through some or all of this fun and easy tutorial.

See also:

help.github.com A great guide to getting started with Git and GitHub, which hosts most of our Git repositories.

GitHub for Windows A Windows program for interacting with GitHub as an alternative to using Git in a terminal. Useful if you are not used to using a terminal yet.

GitHub for Mac A Mac OS X program for interacting with GitHub as an alternative to using Git in a terminal. Useful if you are not used to using a terminal yet.

1.6.3 Load-Testing Tools

Molotov is used for writing and running load tests.

Ardere is another tool (which replaces loads-broker) to run load tests at scale/distributed.

CHAPTER 2

New Employee Guide

Welcome to the New Employee Guide - hopefully you've already gone through, or are familiar with tools, accounts, processes, etc. in our New Contributor Guide. An additional resource to familiarize yourself with is our Firefox Test Engineering space on Mana.

2.1 Continuous Integration

2.1.1 Production

Our **production** Jenkins instance is available at https://qa-master.fxtest.jenkins.stage.mozaws.net/ and access is restricted according to this documentation.

2.1.2 Sandbox, aka "Dev Jenkins"

Our **sandbox** Jenkins instance is available at https://qa-preprod-master.fxtest.jenkins.stage.mozaws.net/ and requires a connection to the Mozilla VPN. See the documentation for further information regarding this instance.

Plugin Updates

1. Whomever is able to respond and take action first, files a bug in Cloud Services | FXTest-Infra, cc:ing the rest of the core Jenkins/infra team, assigning the bug to themselves, and checking the "Security" checkbox at the bottom of the bug. Include the Jenkins advisory text, with a link (like https://jenkins.io/security/advisory/2017-04-26/), the name of and link to the affected plugin(s), as well as the version to which you've upgraded Jenkins dev. Please use this Bugzilla template, to file.
2. After filing, it's time to upgrade the plugin(s):
3. Update Jenkins dev:

- Log in to the Jenkins dev instance
- Click on "Manage Jenkins" on the left

- Click on “Prepare for Shutdown”
- Click on “Manage Plugins”
- Click the “Check Now” button
- Click the checkbox(es) next to the affected plugin(s), and click the “Download now and install after restart” button
- Also select the checkbox to “Restart Jenkins when installation is complete and no jobs are running”
- Under “Build Queue”, click the “cancel” link to allow Jenkins to safely restart
- Run the sanity.pipeline job, vet the results, looking for new, related failures
- Once the upgrades have completed on dev, resolve the Bugzilla bug as fixed

4. Kick off the “run all builds” test job
5. Carefully vet the results
6. If all goes well, follow the instructions for updating plugins on production Jenkins

Plugin Addition

1. Coordinate with and give peers a heads-up that you’re installing a new plugin on dev (and why)
2. Install the plugin
3. Restart Jenkins
4. Run the `sanity.pipeline` job, and try to ensure there are no new, related failures
5. Once you’re comfortable with the results, do the following:
6. File a bug using this Bugzilla template, requesting the plugin(s) installation. Include the following info:

- the plugin name(s), version(s), link(s) on https://plugins.jenkins.io/
- mention that it’s been successfully tested on the dev instance.

7. Once Ops installs the plugin on Prod, make sure to:

- test affected job(s), and
- ping back in the Production-update bug with the appropriate resolution/verification data

2.1.3 Ops-QA Pipeline

The current flow for a project integrated into the Cloud Ops deploy pipeline is as follows:

1. A tagged or pushed build from dev deploys to staging
2. Cloud Ops’ deploy-pipeline script calls `qaTest("kinto", "stage")`, which remotely runs the project’s corresponding staging (“**stage**”) test job, e.g. `kinto.stage`, in our Jenkins instance
3. If our tests pass (returning exit code/return status of “0”), and after manual confirmation from Ops, the build gets promoted and pushed to production

Getting a project’s tests into the deploy pipeline:

1. A suggestion is to have your project build and run tests in Jenkins, from a Docker image
2. Create a Jenkins job with the following syntax: **project.test_env** (e.g. **kinto.stage**), using the `Pipeline from SCM` option, and pointing to the Jenkinsfile

3. Once your project runs and passes in Jenkins:
4. File a bug (example: bug 1384404), in the most-appropriate component for your project, under the Cloud Services product, requesting Ops enable your jobs in their pipeline
5. Next, from Ops' side, there is a qaTest.groovy file which calls run_jenkins_job, which, in turn, authenticates with QA (prod) Jenkins, and will run /job/${project}.${envName}

2.1.4 Build notifications

Notifications can differ between projects, however typically whenever a build fails a notification is sent to the fte-ci group. When the result of a build changes, a notification is sent to the #fx-test-alerts IRC channel on irc.mozilla.org.

CHAPTER 3

Reference

This is where all the useful information goes that doesn't fit into the *New Contributor Guide*.

3.1 Git and GitHub

The sections below describe some Firefox Test Engineering best practices for using Git and GitHub. For more general information on Git here is a link to Good Resources for Learning Git and GitHub.

3.1.1 Issues

You can find an issue to work on by going to the Firefox Test Engineering Dashboard. Any unclaimed Issue is available to you to work on. You can find out more about our process in the *New Contributor Guide*.

3.1.2 Commit Messages

See Tim Pope's blog post on Git commit messages.

3.1.3 Rebasing Commits

While projects vary in their opinions on whether merge commits should be avoided or not, it is generally a good idea to rebase a feature branch before submitting a pull request.

Rebasing allows you to alter a series of commits, changing the history of your repository. Typically you rebase a branch to:

- Combine smaller commits made during development into larger, logical commits that are easier to understand and review, or split up larger commits into smaller commits for the same purpose.
- Alter commit messages of previous commits.

- Move a branch to be based on the latest commit of the branch you want to merge into and resolve any conflicts that occur.

If you're not familiar with rebasing, you can start with this short guide on how to use the `git rebase` command.

These changes all make the code review process as well as the merging process easier, and are recommended for all pull requests.

> **Warning:** Rebasing code that has already been pushed to a public or shared repository makes it very difficult for others to update their local repositories. Only rebase branches that you are absolutely sure no one else is using, such as feature branches on your personal fork.

3.1.4 Owners and the Mozilla GitHub Organization

See the GitHub page on wiki.mozilla.org for information on the Mozilla organization on GitHub or anything that requires owner access for the organization.

3.2 Python Style Guide

This document is a brief set of guidelines for writing Python code at Mozilla. Individual projects may override these rules; make sure you know the standards for your project!

3.2.1 General Guidelines

- Follow PEP 8.
- Check your code against a linting tool. flake8 is highly recommended for this.

3.2.2 Import Statements

We expand on PEP 8's suggestions for import statements. These greatly improve one's ability to ascertain what is and isn't available in a given file.

Import one module per import statement:

```
import os
import sys
```

not:

```
import os, sys
```

Separate imports into groups with a line of whitespace: standard library; (if a web app) Django or other framework; third-party; and local imports:

```
import os
import sys

from django.conf import settings

import pyquery
```

```
from myapp import models, views
```

Alphabetize your imports; it will make your code easier to scan. See how terrible this is:

```
import cows
import kittens
import bears
```

A simple sort:

```
import bears
import cows
import kittens
```

Imports on top, `from` imports below:

```
import x
import y
import z
from bears import pandas
from xylophone import bar
from zoos import lions
```

That's loads easier to read than:

```
from bears import pandas
import x
from xylophone import bar
import y
import z
from zoos import lions
```

Lastly, when importing things into your namespace from a package use an alphabetized `CONSTANT`, `Class`, `var` order:

```
from models import DATE, TIME, Dog, Kitteh, upload_pets
```

If possible though, it may be easier to import the entire package, especially for methods as it help answers the question, "where did `you` come from?"

Bad:

```
from foo import you


def my_code():
    you()  # wait, is this defined in this file?
```

Good:

```
import foo


def my_code():
    foo.you()  # oh you...
```

3.2.3 Whitespace matters

- Use 4 spaces, not 2—it increases legibility considerably.
- Never use tabs—history has shown that we cannot handle them.

Use single quotes unless double (or triple) quotes would be an improvement:

```
'this is good'

'this\'s bad'

"this's good"

"this is inconsistent, but ok"

"""this's sometimes "necessary"."""

'''nobody really does this'''
```

To continue a new line use a `()` not `\`.

Indenting code should be done in one of two ways: a hanging indent, or 4-space indent on the next line.

Good, using hanging indent. Note that the next line is lined up with the previous line delimiter:

```
log.msg('Something long log message and some vars: {0}, {1}'
        .format(variable_a, variable_b))
```

Good using 4 spaces:

```
accounts = PaymentAccounts.objects.filter(
    accounts__provider__type=2,
    something_else=True
)

# A more compact alternative.
accounts = PaymentAccounts.objects.filter(
    accounts__provider__type=2, something_else=True)

accounts = (PaymentAccounts.objects
    .filter(accounts__provider__type=2)
    .exclude(something_else=False)
)
```

Remember that comprehensibility is the goal here. If following one of the rules above would result in less readable code, don't follow it!

3.3 Common Conventions

The following are some hopefully-helpful pointers to common conventions we're using across *most* automation projects. Their adoption aims to make automation easily repeatable, runnable, more reliable, and with clearer output. This list is neither exhaustive nor authoritative, but we strive to keep it as up-to-date and relevant as possible.

3.3.1 Code Coverage

Purpose: Blurb about code coverage (what/why/how?) – particularly our current defacto, Coveralls – goes here.

3.3.2 Dependencies + Virtual Environments

pyup

Purpose: pyup.io helps us manage updating our dependencies and requirements.

Examples:

1. Example one
2. Example two

pipenv

Purpose: We use pipenv

Typically, we:

1. Pin pipenv
2. Install directly from the Pipfile
3. Ignore generating and using the Pipfile.lock

tox

Purpose: We use tox to run tests using multiple Python versions.

3.3.3 Docker

Purpose: We use Docker to help ensure consistent, portable builds, with dependency/environment control.

In most cases, we build our Docker image directly from its **master** branch in GitHub, in any of the following: Travis CI, Circle CI, and Jenkins.

Typically, we use two (2) main files, for Docker:

1. Dockerfile
2. .dockerignore

Examples:

1. Example one
2. Example two

For both of these files, see the official docs at https://docs.docker.com/engine/reference/builder

3.3.4 Jenkins

3.3.5 Linting

Purpose:

PEP8 and flake8. Other considerations for specific linting are:

1. flake8-isort
2. flake8-docstrings
3. pylint

3.3.6 Travis CI

Purpose:

We use Travis CI for linting pull requests/commits.

3.4 Testing

Testing is a very important part of the development process. It allows us to verify the functionality of our projects as well as judge the quality of our work.

At Mozilla, we have multiple ways of testing our code, including:

- Unit tests and integration tests, which are automated tests that verify that pieces of code work as expected.
- End-to-end tests, automated tests which check the functionality of a project as a whole. For example, simulating clicks in a web browser to test how a site functions.
- Manual testing, which is performed by a human and involves verifying features work as expected and exploratory tests.

3.4.1 Assessing and managing risk

The end goal of testing is to manage the risk of something going wrong with your project. To this end, one of the first steps you should take is to assess the risk of each area of your project.

More concretely, some parts of your project are going to be more likely to fail than others. Also, some parts of your project are more important than others, and it may be more harmful for them to fail than less important parts.

A risk assessment lists out the different parts of your project (such as certain webpages or parts of an API) and ranks them based on their importance. For example, a news site rank being able to read existing articles as more important than being able to submit new articles. Ranking these parts allows you to make decisions about which to test more and what kind of tests to run.

Some projects rely on Travis CI for executing their tests.

For Django projects, these tests live within the `tests` module of each included Django application. For Node-based projects, they normally live in a directory named `test` or `tests` at the root of the repository. Refer to your project's documentation for more details.

3.4.2 End-to-end tests

End-to-end tests simulates how your project will be used by users and verifies that it behaves as expected. This is most commonly applied to websites, where we use tools like Selenium to simulate users interacting with the website.

For many sites, these tests are written by WebQA contributors and run against the various server environments.

3.4.3 Manual testing

Manual testing is good old-fashioned human-powered testing, where a living, breathing human uses your project and checks for any errors. Typically this is either for verifying that a new feature works as expected, or for free-form exploratory testing.

In addition to writing automated tests, you almost certainly should be manually testing any changes you make to a project.

3.4.4 Testing tools

The following is a non-exhaustive, possibly-out-of-date list of tools and libraries that may aid you in testing your projects.

General

- Jenkins is a continuous integration server that builds and/or tests software projects continuously.
- Travis CI is a hosted continuous integration service that integrates with Github.
- Selenium is a tool for automating browsers, often for testing purposes.

Python

- pytest is a highly recommended testing library for Python, with a great plugin ecosystem. Some common plugins we're using include
  - pytest-testrail for sending test run details to our *Testrail* server
  - pytest-bugzilla-notifier for sending test summaries to Bugzilla tickets
- factory-boy replaces test fixtures with factories that generate test objects easily. It integrates with the Django ORM to generate model instances with a very conveninent syntax.
- Mock is one of the most popular libraries for replacing parts of the system you're testing with mock objects and asserting things about their behavior.

Node / JavaScript

- Mocha is a framework for running tests on node.js and in the browser.
- Chai is an assertion library with many interfaces to accomodate different testing styles.
- Karma allows you to execute JavaScript code in multiple real browsers.

3.5 Test Results in ActiveData

Our automated test results are publicly accessible via ActiveData, which allows us to determine areas the need attention. For example, we might want to identify tests that take the longest to run, or tests that fail most often. We can also use ActiveData to see if changing the version of Python or a package has an effect on the duration or outcome of the tests.

Most of our test automation is based on pytest, and in order to these results into ActiveData we need to generate structured logs and upload them to an Amazon S3 bucket. ActiveData scans this bucket, ingests the logs, and the results are then available for querying.

3.5.1 Structured logs

Many test suites at Mozilla use mozlog to generate structured logs. As ActiveData is already familiar with this format, it makes sense to reuse it for our test results. To achieve this, mozlog includes a simple pytest plugin named pytest-mozlog. When mozlog is installed, additional command line options are added to pytest for generating logs in the various available formats. For example, to generate a structured log:

```
pytest --log-raw=raw.log
```

Other formats are available, however the raw format is the only one that ActiveData will be able to process. Here's an example of the output:

```
{"pid": 92739, "run_info": {"Python": "2.7.10", "Plugins": {"mozlog": "3.4", "xdist":
↪"1.15.0", "base-url": "1.3.0", "metadata": "1.3.0", "html": "1.14.2"}, "Packages": {
↪"pytest": "3.0.6", "pluggy": "0.4.0", "py": "1.4.32"}, "Platform": "Darwin-16.4.0-
↪x86_64-i386-64bit"}, "action": "suite_start", "tests": ["test_foo.py::test_foo",
↪"test_bar.py::test_bar"], "component": "pytest", "source": "pytest", "time":␣
↪1489585066381, "thread": "MainThread"}
{"pid": 92739, "test": "test_foo.py::test_foo", "action": "test_start", "component":
↪"pytest", "source": "pytest", "time": 1489585071631, "thread": "MainThread"}
{"pid": 92739, "test": "test_bar.py::test_bar", "action": "test_start", "component":
↪"pytest", "source": "pytest", "time": 1489585071631, "thread": "MainThread"}
{"status": "PASS", "pid": 92739, "test": "test_foo.py::test_foo", "action": "test_end
↪", "component": "pytest", "source": "pytest", "time": 1489585072217, "thread":
↪"MainThread"}
{"status": "PASS", "pid": 92739, "test": "test_foo.py::test_bar", "action": "test_end
↪", "component": "pytest", "source": "pytest", "time": 1489585072219, "thread":
↪"MainThread"}
{"pid": 92739, "action": "suite_end", "component": "pytest", "source": "pytest", "time
↪": 1489585072594, "thread": "MainThread"}
```

3.5.2 Metadata

In order to add context to the results we use the pytest-metadata plugin. This adds details on the platform, Python binary, pytest packages, and pytest plugins used in the test session. It also adds environment variables from several continuous integrations servers, and we use this to associate results with a specific application under test. All of this data is added to the `run_info` in the `suite_start` message within the structured log.

3.5.3 Querying ActiveData

You can use the ActiveData Query Tool to run queries and see the responses from ActiveData. The getting started guide is a good place to start, however let's explore a couple of examples.

Test durations

The following query will return the 90th percentile for test duration, grouped by test name and job:

```
{
  "from":"fx-test",
  "limit":1000,
  "groupby":["test.full_name"],
  "select":[{
    "aggregate":"percentile",
    "percentile":0.9,
    "value":"result.duration"
  }]
}
```

You'll need to sort the results in the client to determine the longest running tests. Then you may want to do further queries to learn if these tests are longer against different environments, or across the board. This might highlight tests that are doing too much, or at least slowing down the feedback loop.

Failing tests

The following query will return the total number of times each test has failed.

```
{
  "from":"fx-test",
  "limit":1000,
  "groupby":["test.full_name"],
  "where":{"eq":{"result.ok":false}},
  "select":[{"aggregate":"count"}]
}
```

Note that this doesn't distinguish between the various outcomes that evaluate as a failure, so this is just wherever the outcome does not match the expectation.

3.5.4 Plotting results

A useful way to visualize the results from ActiveData is to plot them on a chart. This can be achieved using a Jupyter Notebook, with pandas, NumPy, and matplotlib. If you have Docker installed then a really quick way to get started is to use the jupyter/datascience-notebook image:

```
$ docker run -it --rm -p 8888:8888 jupyter/datascience-notebook
[I 17:58:11.744 NotebookApp] Writing notebook server cookie secret to /home/jovyan/.
↪local/share/jupyter/runtime/notebook_cookie_secret
[W 17:58:12.747 NotebookApp] WARNING: The notebook server is listening on all IP␣
↪addresses and not using encryption. This is not recommended.
[I 17:58:12.814 NotebookApp] Serving notebooks from local directory: /home/jovyan/work
[I 17:58:12.814 NotebookApp] 0 active kernels
[I 17:58:12.814 NotebookApp] The Jupyter Notebook is running at: http://[all ip␣
↪addresses on your system]:8888/?token=[TOKEN]
[I 17:58:12.814 NotebookApp] Use Control-C to stop this server and shut down all␣
↪kernels (twice to skip confirmation).
[C 17:58:12.824 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:8888/?token=[TOKEN]
```

Open Jupyter at the URL provided and create a new Notebook with your preferred Python version.

In the first cell, build your query. The following will return all failing tests run in the last two weeks by day and outcome, which will allow us to plot the outcomes on a chart:

```
query = """{
"from":"fx-test",
"edges":[
  {"value":"result.result","allowNulls":false},
  {
    "value":"result.end_time",
    "allowNulls":false,
    "domain":{
      "type":"time",
      "min":"today-2week",
      "max":"tomorrow",
      "interval":"day"
    }
  }
],
"where":{"and":[
  {"gte":{"result.end_time":{"date":"today-2week"}}},
  {"eq":{"result.ok":false}}
]},
"format":"cube",
"limit":1000
}"""
```

In the next cell, post the query and retrieve the JSON results:

```
import requests
data = requests.post('http://activedata.allizom.org/query', data=query).json()
```

Now we import NumPy and Pandas, and build a DataFrame with a series for each outcome:

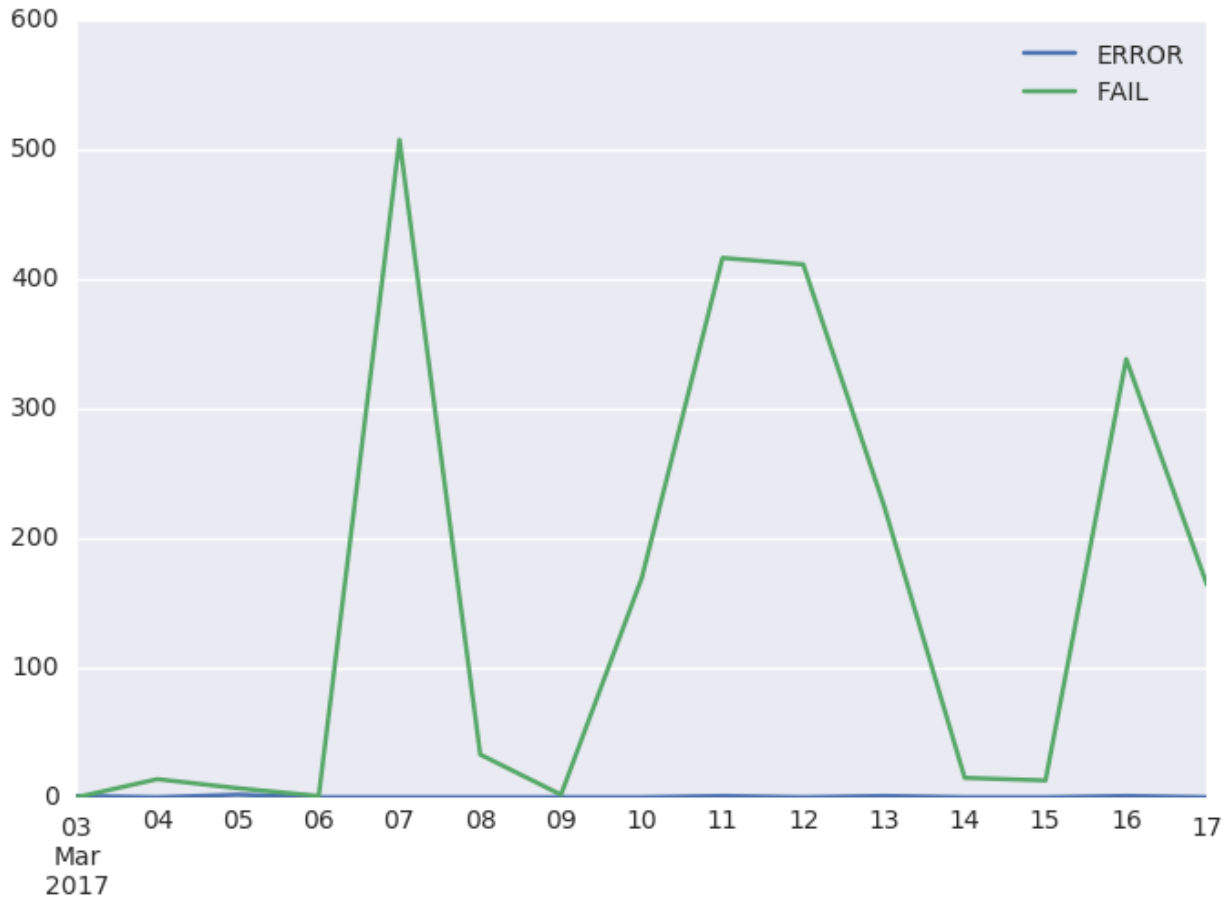
```
import numpy as np
import pandas as pd

d = {}
i = np.array([d['min'] for d in data['edges'][1]['domain']['partitions']]).astype(
↪'datetime64[s]')
for idx, val in enumerate(data['data']['count']):
    result = data['edges'][0]['domain']['partitions'][idx]['value']
    d[result] = pd.Series(val, index=i)
df = pd.DataFrame(d)
```

Finally, we import Seaborn (for more attractive charts) and plot our line chart:

```
import seaborn as sns
sns.set_style('darkgrid')
df.plot.line()
```

The chart will be displayed in the Jupyter Notebook. It's now pretty easy to tweak the query and DataFrame, or try different types of charts.



3.5.5 Known limitations

Unfortunately, mozlog does not currently support Python 3. This means that any suite that produces structured logs for consumption by ActiveData is required to run on legacy Python.

3.6 Test Results in Treeherder

Treeherder is Mozilla's reporting dashboard for checkins, builds, and test results. Our automated test results are published to Treeherder via Pulse.

3.6.1 Submitting from Jenkins

To submit results from Jenkins you will need to write your job as a declarative pipeline using our fxtest shared library. See the documentation for more information.

3.6.2 Viewing results in Treeherder

To view the results for a particular project, open Treeherder and select the appropriate repository from the menu. Most of our projects will be found within the 'qa automation tests' group. You will then see a resultset for each commit

to that repository, and any associated test jobs will be displayed. Click a job to find the test logs and report. The Treeherder User Guide may also be useful.

3.7 PyPOM

PyPOM, or Python Page Object Model, is a Python library that provides a base page object model for use with Selenium functional tests.

PyPOM is a fundamental part of our test automation. Learn more about PyPOM on their readthedocs page.

3.8 IRC Bot

We have a chat bot in our IRC channel providing useful features such as notifications or new issues, pull requests and pushes to our GitHub repositories.

The source code is available on GitHub, and notifications can be configured for any repository by following the steps in the documentation.

3.9 Glossary

At Mozilla, we use a lot of special terms that mean specific, non-obvious things to those who aren't familiar with them. This wiki document attempts to define those terms.

CHAPTER 4

Resources

Useful resources for further reading and learning.

4.1 Videos

- Mozilla Onboarding - things that you'll be using at Mozilla.
- Mozilla QA on YouTube
- Mozilla on YouTube