Mozilla CI Tools Documentation

Release 0.1.1

Armen Zambrano G.

August 17, 2015

Contents

Mozilla CI Tools (mozci) is designed to allow interacting with the various components which compose Mozilla's Continuous Integration. Specifically, we're talking about the builds and test jobs produced in treeherder.

mozci tools has several modules and command line scripts to help you use them.

CHAPTER 1

Table of contents

1.1 Project definition

1.1.1 Roadmap

Table of Contents

Create prototype to trigger N jobs for a range of revisions

This allows triggering multiple jobs across a range of revisions.

Here's an example:

```
python scripts/trigger.py \
    --buildername "Ubuntu VM 12.04 fx-team opt test jittest-1" \
    --rev e16054134e12 --from-rev fb64168bf663 --times 2
```

This has been accomplished on the 0.2.1 release (13/02/2015).

Add pushlog support

This helps interacting with ranges of revisions.

This has been accomplished on the 0.2.1 release (13/02/2015).

Determine accurately the current state of jobs

We can determine any information on jobs run on the Buildbot CI by retrieving scheduling information through Self-Serve's BuildAPI. The information retrieved can then be matched to the buildjson status dumps that are produced every minute (for the last 4 hours) and every 15 minutes (for the day's worth of data).

This feature is completed. (25/02/2015).

Create prototype to find last good job and backfill up to it

Given a bad job, we can simply scan the previous revisions for the last known good job for it. Known that, we can trigger all jobs required to trigger the missing jobs.

The script will find the last good job or trigger up to a maximum of revisions. We can also indicate that we want multiple jobs instead of just one.

Here's an example:

```
python scripts/trigger.py \
    --buildername "Ubuntu VM 12.04 fx-team opt test jittest-1" \
    --rev e16054134e12 --backfill --max-revisions 30 --times 2
```

This feature was completed on release 0.3.0.

Ability to navigate through merges into other repositories

We need the ability to navigate through pushes and be able to follow through from one repository to another when merge pushes are found.

This is important to find an intermittent failing job.

We have filed issue 127 to track this.

Determine if a test failed in a job

We currently can only tell if a job has failed, however, we don't know which test(s) have failed.

If we want to find intermittent oranges we would need to determine if a job has failed because the test we care about has actually failed.

A possible way to determine this would be to grab the structured log uploaded through blobber.

NOTE: Reftests currently don't generate structured logs. In such case we would have to fallback to parsing logs. We would have to request this to be implemented.

NOTE 2: In the future, we might be able to query this information through the big data project.

We have filed issue 128 to track this.

Determine frequency of test failure

We need to find a way to analyze test failure frequency. That way we can determine the right number of jobs of to retrigger a job in order to find an intermittent orange.

I'm not sure if this will really be needed. We will see.

Make handling Buildbot job information sustainable

We currently have information about Buildbot jobs by grabbing scheduling info and status info. This is something that users should never know about. We should only expose the bits of information which are relevant to the user and allow them to access the raw data only if wished for.

Adding a base class to abstract jobs and have one implementation for Buildbot will make the project ready for the representation of jobs in TaskCluster.

We have filed issue 21 to track this.

Parallelize the analysis of each revision

We currently iterate and analyze one revision at a time. In many places we can parallelize this process since they're isolated processes.

This will speed up the execution time.

NOTES:

- We should only print to the log once a revision has been completely processed.
- We should not print the summary about a revision until all most recent revisions are processed first.
  - i.e. log them in descending order based on their push id.
  - This will ensure that inspecting the log will visually make sense

We have filed issue 129 to track this.

Test framework to test CI data sources

We need to have a way to prevent regressions on Mozilla CI tools. Adding coverage reports would help us fix this issue.

We also need a way to test the data sources structures for changes that could regress us (e.g. new builder naming for Buildbot). We might be able to simply mock it but we might need to set up the various data sources.

This is to be tackled in Q2/Q3 2015.

We have filed issue 130 to track this.

Provide data structure to generate up-to-date trychooser

Currently trychooser's UI is always out-of-date and nothing intelligent can be done with it. With mozci we currently can generate most of the data necessary to create a dynamic UI.

To generate the current data structure you can run this::

```
python scripts/misc/write_tests_per_platform_graph.py
```

graphs.json will be generated. We have filed issue 69 to track this.

NOTE: This will be needed once someone picks up bug 983802.

Integrate backfilling feature into treeherder

This will be similar to the re-trigger button that is part of the treeherder UI. We select a job that is failing and request that we backfill. mozci will determine when was the last time there was a successful job and trigger all missing jobs up to the last known good job.

TreeHerder currently uses client-side triggering for the re-trigger button and it intends to move it to the server side (bug 1077053).

We have filed issue 109 to track this.

Pulse support

Pulse allows you to listen and consume about jobs changing status. This is very important for monitoring jobs going through various states.

We have filed issue 126 to track this.

Add ability to monitor jobs

We currently run a script and let it schedule everything that is needed. However, we assume an ideal case scenario: **everything that we schedule gets run**.

This is a very optimistic approach. We should allow the user to use a mode in which the script watches and notifies the user when our expectations are not met.

For instance:

- A build finishes, however, the tests that is expected to run gets coalesced
  - In this case we would be expecting a completed build job + a completed test job
  - We would need to schedule the test job and watch it
- A job fails, however, we assume it would succeed
  - We need to re-trigger it and watch it

We have filed issue 131 to track this.

Support TaskCluster

As we're transitioning to TaskCluster we should add the support for it.

We are tracking this with the TaskCluster Support milestone.

1.1.2 Use cases

Table of Contents

Case scenario 1: Bisecting permanent issue

- We have a job failing
- There are jobs that were coalesced between the last good and the first bad job
- We need to backfill between good revision and the bad revision

This has been completed by the trigger.py with –backfill.

Case scenario 2: Bisecting intermittent issue

- We have an intermittent job
- We want to determine when it started happening
- It is not only a matter of coalescing but also a matter of frequency
- We want to give a range of changesets and bisect until spotting culprit

NOTE: We trigger more than one job compared to case scenario 1

The script trigger.py helps with triggering multiple times the same jobs. The script generate_cli.py helps with tracking filed intermittent oranges in bugzilla.

This use case requires implementing the following milestones:

- Add the ability to find intermittent oranges
- Allow monitoring jobs

Case scenario 3: Retrigger an intermittent job on a changeset until hit

https://bugzilla.mozilla.org/show_bug.cgi?id=844746

- This is more of an optimization.
- The intent is to hit the orange with extra debugging information.
- We're not bisecting in here.
- We can trigger batches (e.g. 5 at a time)

Not in scope at the moment.

This could be done with a modification of trigger.py where we monitor the jobs running until one of them fails.

This use case requires implementing the following milestones:

- Allow monitoring jobs

and this issue:

- Add the ability to inspect the reason of a job failure

Case scenario 4: Bisecting Talos

- We have a performance regression
- We want to determine when it started showing up
- Given a revision that _failed_
- Re-trigger that revision N times and all revisions prior to it until the last data point + 1 more

We can do this manually by using –back-revisions::

```
python scripts/trigger.py \
    --buildername "Rev5 MacOSX Yosemite 10.10 fx-team talos dromaeojs" \
    --rev e16054134e12 --back-revisions 10
```

We currently don't have the ability to determine the previous baseline and trigger everything up to the last known data point close to the baseline.

Case scenario 5: After uplift we need a new baseline for release branches

- We need several data points to establish a baseline
- After an uplift we need to generate a new baseline
- Once there is a baseline we can determine regression

This can already be accomplished like this::

```
python alltalos.py --repo-name try --times 2 --rev 921277f744d9
```

Case scenario 6: New test validation

- New test validation
- Re-triggering to determine indeterminacy
- Single revision
- All platforms running test

NOTE: Review this thread on how to determine on which job and which platforms do we run a specific test.

Not in scope at the moment.

Case scenario 7: Fill in a changeset

- We know that a changeset is missing jobs
- We want to add all missing jobs

This would need grabbing the list of builders for such repo, determine if there has been a successful job for each builder and trigger it if not. We might need to filter out some periodic builders (i.e. a repo can have pgo periodic jobs).

If we want to watch that revision until it is completely filled we will need to accomplish the following milestones:

- Allow monitoring jobs

Case scenario 8: Developer needs to add missing platforms/jobs for a Try push

- The developer pushes to try specifying only a subset of all jobs
- The developer realizes that it needs more jobs to run on that push
- The developer uses mozci to not have to push again to try with the right syntax

This has been filed as issue 109

Case scenario 9: We generate data to build a dynamic TryChooser UI

- TryChooser UI is always out of date
- mozci can generate the data we need to create an up-to-date TryChooser UI

See write_tests_per_platform_graph.py for an example on how to generate the data needed.

1.1.3 F.A.Q.

Table of Contents

I asked mozci to trigger a test job for me, however, a build got triggered

Test jobs require a build to be tested, hence, needing to trigger a build for it.

In some cases you might see that a build exists for that push and still trigger a build. This is because we have checked for its uploaded files and have been expired. In this case we need to trigger the build job again.

How does mozci deal with running/pending jobs?

mozci expects that running/pending _build_ jobs will trigger the tests we want.

If we want more than one job and we observe one running/pending job, we will trigger as many jobs as needed to meet the request.

How does mozci deal with failed jobs?

mozci is not currently designed to trigger jobs until they succeed. The use must say what it wants to trigger and we will only do a one pass to trigger everything requested. This is purposeful for simplicity and to prevent triggering jobs endlessly.

In the near future, we will add a monitoring module which could be used to keep track of triggered jobs and determine actions upon completion.

Can we schedule a PGO job on any tree?

Yes and no. We can't trigger anything that the Release Engineering's Buildbot CI can trigger. If a tree does not have a builder that generates a PGO build then we can't.

Notice that some trees can trigger both non-PGO jobs and PGO jobs, hence, that tree has two different buildernames (one including "pgo" in its name). Other trees can only trigger PGO jobs and might not include "pgo" in its name (think of mozilla-aurora).

What products do you support?

All Firefox desktop and Firefox for Android. For Firefox OS we have partial support since some of the jobs run on the TaskCluster CI (which is not yet supported).

Can I trigger a nightly build?

Absolutely! You can trigger any nightly build for any repository. Simply find the name of the job that represents it and trigger it.

NOTE: Make sure you have consent from sheriffs to do this and a good reason. Nightly builds are not to be triggered lightly.

If I ask for different test jobs on the same changesets will I get as many builds jobs?

There is a possible 30 seconds delay between making a request to buildapi and it appearing as "pending/running". If you hit this issue please let us know and we can discuss it on how to better address it. There are various options we can consider.

Does this work with TaskCluster?

Not yet.

Can anybody use mozci?

As long as you have LDAP credentials you should be able to use it.

What systems does mozci rely on?

If you look at mozci.sources you will see all CI components we depend on. If the structure of any of these changes, we might need to adjust mozci for it.

What happens if a new platform or suites are added to the CI?

Nothing to worry about! mozci determines platforms dynamically rather than statically.

What use cases are you hoping to address?

Please refer to Use cases.

I see that you store my credentials in plain text on my machine

If you want a different approach please let us know.

Can I run mozci in my web service?

Yes! However, we will need to figure out how to provide credentials. More to come.

Is mozci limited by the try chooser syntax?

No. We hit an API that is not affected by the try parser. We can trigger anything that can be triggered without any limitations. You can add more jobs on a try push than indicated in the try syntax of that push.

Can you trigger jobs on pushes with DONTBUILD?

No, we can not. This is a bug in buildapi. The pushes doesn't even exist for buildapi. You can notice this if you load self-serve/buildapi for a DONTBUILD push.

How do you deal with coalesced and not scheduled jobs?

Coalesced jobs are requests that have been fulfilled by more recent pushes. We coalesce jobs to be able to catch up under high load.

We sometimes not schedule jobs for various reasons including:

- The user has marked the job not to be built with DONTBUILD in the commit message
- The files changed on that push do not affect certain products/platforms

- We have determined that we don't need to trigger that job on every push

self-serve/buildapi does not keep track of jobs that have been coalesced or not scheduled.

mozci determines how many jobs to trigger a job depending on how many successful, running jobs and potential jobs trigger by a build. Coalesced and not scheduled jobs are not considered.

What are the concerns of trigger a large number of jobs in a short period of time?

Self-serve/buildapi is known to be unresponsive if too much is demanded of it. The operations of treeherder will continue as usual since the Buildbot master query the buildbot databases directly rather than through self-serve/buildapi. Re-triggering of jobs would be temporarily unavailable until self-serve auto-recovers. At worse, nagios checks will be triggered and buildduty will have to investigate.

Treeherder could also be affected if buildapi/self-serve did not go down and actually managed to trigger a lot of jobs. It is known that treeherder gets into trouble if several thousands of jobs get triggered in a short period of time.

Proper usage of mozci should not cause any issues, however, **intentional** misuse could cause the issues mentioned above.

What performance constraints does mozci have?

We are currently mainly restrained by two factors: sequential approach to triggering and responsiveness of the data sources.

We currently go through each push in a sequential order. In order to speed this up we could parallelize the work done on each push.

The data sources we use can be slow at times depending on the load on them. If this becomes troublesome we should investigate how to optimize them.

How do you release software?

We use zest.releaser. You simply install it:

```
pip install zest.releaser
```

`fullrelease` is used to bump the versions, tag and upload the new package to pypi.

The releases are documented in here.

How do I generate the docs?

To generate the docs, follow these steps:

- Move inside docs/ directory
- Run:

```
pip install -r requirements.txt
make html
```

- To view the docs on a webserver http://127.0.0.1:8000 and auto-rebuild the documentation when any files are changed:

```
make livehtml
```

How can I contribute?

If you would like to contribute to this project, feel free to pick up one of the issues or tasks in the Trello board (Tasks) or the issues page (Issues).

In order to contribute the code:

- Fork the project
- Create a new branch
- Fix the issue - add the feature
- Run tox successfully
- Commit your code
- Request a pull request

1.1.4 Vision

At Mozilla we run thousands of jobs to build and test Firefox every day. We also try to do smarter scheduling to save resources under high load which results in the need to do further investigation when an issue arises. Even when we have the data, sometimes we have intermittent data or performance data which needs additional runs to detect a pattern. The Mozilla CI Tools project is designed to arbitrarily schedule jobs on given revisions and job types based on different scenarios. This is a difficult problem to solve as we communicate with many systems to get accurate information which is needed for us to ensure we are sending the right parameters to trigger a specific job. In addition we have a set of specific higher level scenarios to solve for when we get a failure or intermittent failure. As the tool chain matures these scenarios will be integrated into existing tools and dashboards.

Some of what this project can potentially accomplish is:

- Trigger any jobs (builds, tests, nightly, L10n et al)
- Query any information related to our VCS systems
- Determine completeness of jobs run on a revision
- Find hidden jobs that are permanently wasting resources
- Help us bisect intermittent oranges
- Help us backfill any missing jobs
- Help us find any files/artifacts generated by any job in our CI

This year's goal is to answer some of these needs based on Release Engineering's current Buildbot CI. In the near future, we should also be able to do the same for the TaskCluster CI.

In order to accomplish this we need to add the following basic features:

- Determine accurately the current state of jobs
- Determine the full set of jobs that can be run for a given revision
- Log jobs triggered in a consumable manner
- Allow a user monitor jobs triggered
- Create test framework to test the various CI data sources or mock them

The remainder of this document will describe our roadmap and potential use cases.

1.2 Using mozci

You can use mozci by using the scripts in the code or include it as part of your project.

To learn how to use the scripts visit the Scripts page.

To create tools or scripts using mozci, you can interact directly with the various **data sources** or use the main two modules:

1.2.1 mozci

This module is generally your starting point.

Instead of accessing directly a module that represents a data source (e.g. buildapi.py), we highly encourage you to use mozci.py instead which interfaces with them through. As the continuous integration changes, you will be better off letting mozci.py determine which source to reach to take the actions you need.

In here, you will also find high level functions that will do various low level interactions with distinct modules to meet your needs.

mozci.mozci.**find_backfill_revlist**(*repo_url*, *revision*, *max_revisions*, *buildername*)
Determine which revisions we need to trigger in order to backfill.

mozci.mozci.**manual_backfill**(*revision*, *buildername*, *max_revisions*, *dry_run=False*)
This function is used to trigger jobs for a range of revisions when a user clicks the backfill icon for a job on Treeherder.

It backfills to the last known job on Treeherder.

mozci.mozci.**query_builders**()
Return list of all builders.

mozci.mozci.**query_repo_name_from_buildername**(*buildername*, *clobber=False*)
Return the repository name from a given buildername.

mozci.mozci.**query_repo_url_from_buildername**(*buildername*)
Return the full repository URL for a given known buildername.

mozci.mozci.**query_revisions_range**(*repo_name*, *from_revision*, *to_revision*)
Return a list of revisions for that range.

mozci.mozci.**set_query_source**(*query_source='buildapi'*)
Function to set the global QUERY_SOURCE

mozci.mozci.**trigger**(*builder*, *revision*, *files=[]*, *dry_run=False*, *extra_properties=None*)
Helper to trigger a job.

Returns a request.

mozci.mozci.**trigger_all_talos_jobs**(*repo_name*, *revision*, *times*, *dry_run=False*)
Trigger talos jobs (excluding 'pgo') for a given revision.

mozci.mozci.**trigger_job**(*revision*, *buildername*, *times=1*, *files=None*, *dry_run=False*, *extra_properties=None*, *trigger_build_if_missing=True*)
Trigger a job through self-serve.

We return a list of all requests made.

mozci.mozci.**trigger_missing_jobs_for_revision**(*repo_name*, *revision*, *dry_run=False*)
Trigger missing jobs for a given revision. Jobs have any of ('hg bundle', 'b2g', 'pgo') in their buildername will not be triggered.

mozci.mozci.**trigger_range**(*buildername*, *revisions*, *times=1*, *dry_run=False*, *files=None*, *extra_properties=None*, *trigger_build_if_missing=True*)
Schedule the job named "buildername" ("times" times) in every revision on 'revisions'.

mozci.mozci.**valid_builder**(*buildername*)
Determine if the builder you're trying to trigger is valid.

1.2.2 platforms

This module helps us connect builds to tests.

mozci.platforms.**build_talos_buildernames_for_repo**(*repo_name*, *pgo_only=False*)
This function aims to generate all possible talos jobs for a given branch.

Here we take the list of talos buildernames for a given branch. When we want pgo, we build a list of pgo buildernames, then find the non-pgo builders which do not have a pgo equivalent. To do this, we hack the buildernames in a temporary set by removing ' pgo' from the name, then finding the unique jobs in the talos_re jobs. Now we can take the pgo jobs and jobs with no pgo equivalent and have a full set of pgo jobs.

mozci.platforms.**build_tests_per_platform_graph**(*builders*)
Return a graph mapping platforms to tests that run in it.

mozci.platforms.**determine_upstream_builder**(*buildername*)
Given a builder name, find the build job that triggered it.

When buildername corresponds to a test job it determines the triggering build job through allthethings.json. When a buildername corresponds to a build job, it returns it unchanged.

mozci.platforms.**filter_buildernames**(*include*, *exclude*, *buildernames*)
Return every buildername that contains the words in include and not the words in exclude.

mozci.platforms.**find_buildernames**(*repo*, *test=None*, *platform=None*, *job_type='opt'*)
Return a list of buildernames matching the criteria.

1) if the developer provides test, repo and platform and job_type return only the specific buildername 2) if the developer provides test and platform only, then return the test on all platforms 3) if the developer provides platform and repo, then return all the tests on that platform

mozci.platforms.**get_associated_platform_name**(*buildername*)
Given a buildername, find the platform in which it is ran.

mozci.platforms.**get_downstream_jobs**(*upstream_job*)
Return all test jobs that are downstream from a build job.

mozci.platforms.**is_downstream**(*buildername*)
Determine if a job requires files to be triggered.

mozci.platforms.**load_relations**()
Loads the upstream to downstream.

Data sources:

1.2.3 allthethings

This module is to extract information from allthethings.json. More info on how this data source is generated can be found in this wiki page:

This module helps you extract data from allthethings.json The data in that file is a dump of buildbot data structures. It contains a dictionary with 4 keys:

- **builders**:
  - a dictionary in which keys are buildernames and values are the associated properties, for example:

```
"Android 2.3 Armv6 Emulator mozilla-esr31 opt test crashtest-1": {
  "properties": {
    "branch": "mozilla-esr31",
    "platform": "android-armv6",
    "product": "mobile",
    "repo_path": "releases/mozilla-esr31",
    "script_repo_revision": "production",
    "slavebuilddir": "test",
    "stage_platform": "android-armv6"
  },
  "shortname": "mozilla-esr31_ubuntu64_vm_armv6_large_test-crashtest-1",
  "slavebuilddir": "test",
  "slavepool": "37085cdc35d8351f600c8c1cbd165c311880decb"
 },
```

- **schedulers**:
  - a dictionary mapping scheduler names to their downstream builders, for example:

```
"Firefox mozilla-aurora linux l10n nightly": {
  "downstream": [
    "Firefox mozilla-aurora linux l10n nightly"
  ]
 },
```

- **master_builders**
- **slavepools**

mozci.sources.allthethings.**fetch_allthethings_data**(*no_caching=False*, *verify=True*)
It fetches the allthethings.json file.

If no_caching is True, we fetch it every time without creating a file. If verify is False, we load from disk without checking. This should only be used if allthethings.json exists and it's trusted.

mozci.sources.allthethings.**list_builders**()
Return a list of all builders running in the buildbot CI.

1.2.4 buildapi

This script is designed to trigger jobs through Release Engineering's buildapi self-serve service.

The API documentation is in here (behind LDAP): https://secure.pub.build.mozilla.org/buildapi/self-serve

The docs can be found in here: http://moz-releng-buildapi.readthedocs.org

mozci.sources.buildapi.**make_cancel_request**(*repo_name*, *request_id*, *dry_run=True*)
Cancel a request using buildapi self-serve. Returns a request.

Buildapi documentation: DELETE /self-serve/{branch}/request/{request_id} Cancel the given request

mozci.sources.buildapi.**make_retrigger_request**(*repo_name*, *request_id*, *count=1*, *priority=0*, *dry_run=True*)
Retrigger a request using buildapi self-serve. Returns a request.

Buildapi documentation: POST /self-serve/{branch}/request Rebuild *request_id*, which must be passed in as a POST parameter. *priority* and *count* are also accepted as optional parameters. *count* defaults to 1, and represents the number of times this build will be rebuilt.

mozci.sources.buildapi.**query_jobs_schedule**(*repo_name*, *revision*)
Query Buildapi for jobs

mozci.sources.buildapi.**query_jobs_url**(*repo_name*, *revision*)
Return URL of where a developer can login to see the scheduled jobs for a revision.

mozci.sources.buildapi.**query_repositories**(*clobber=False*)
Return dictionary with information about the various repositories.

The data about a repository looks like this:

```
"ash": {
    "repo": "https://hg.mozilla.org/projects/ash",
    "graph_branches": ["Ash"],
    "repo_type": "hg"
}
```

mozci.sources.buildapi.**query_repository**(*repo_name*)
Return dictionary with information about a specific repository.

mozci.sources.buildapi.**trigger_arbitrary_job**(*repo_name*, *builder*, *revision*, *files=[]*, *dry_run=False*, *extra_properties=None*)
Request buildapi to trigger a job for us.

We return the request or None if dry_run is True.

mozci.sources.buildapi.**valid_credentials**()
Verify that the user's credentials are valid.

1.2.5 buildjson

This module helps with the buildjson data generated by the Release Engineering systems: http://builddata.pub.build.mozilla.org/builddata/buildjson

mozci.sources.buildjson.**query_job_data**(*complete_at*, *request_id*)
Look for a job identified by *request_id* inside of a buildjson file under the "builds" entry.

Through *complete_at*, we can determine on which day we can find the metadata about this job.

WARNING: "request_ids" and the ones from "properties" can differ. Issue filed.

If found, the returning entry will look like this (only important values are referenced):

```
{
    "builder_id": int, # It is a unique identifier of a builder
    "starttime": int,
    "endtime": int,
    "properties": {
        "blobber_files": json, # Mainly applicable to test jobs
        "buildername": string,
        "buildid": string,
        "log_url", string,
        "packageUrl": string, # It only applies for build jobs
        "revision": string,
        "repo_path": string, # e.g. projects/cedar
        "request_ids": list of ints, # Scheduling ID
        "slavename": string, # e.g. t-w864-ix-120
        "symbolsUrl": string, # It only applies for build jobs
        "testsUrl": string,   # It only applies for build jobs
    },
    "request_ids": list of ints, # Scheduling ID
```

```
        "requesttime": int,
        "result": int, # Job's exit code
        "slave_id": int, # Unique identifier for the machine that run it
    }
```

NOTE: Remove this block once https://bugzilla.mozilla.org/show_bug.cgi?id=1135991 is fixed.

There is so funkiness in here. A buildjson file for a day is produced every 15 minutes all the way until midnight pacific time. After that, a new _UTC_ day commences. However, we will only contain all jobs ending within the UTC day and not the PT day. If you run any of this code in the last 4 hours of the pacific day, you will have a gap of 4 hours for which you won't have buildjson data (between 4-8pm PT). The gap starts appearing after 8pm PT when builds-4hr cannot cover it.

If we look all endtime values on a day and we print the minimum and maximum values, this is what we get:

```
1424649600 Mon, 23 Feb 2015 00:00:00  () Sun, 22 Feb 2015 16:00:00 -0800 (PST)
1424736000 Tue, 24 Feb 2015 00:00:00  () Mon, 23 Feb 2015 16:00:00 -0800 (PST)
```

This means that since 4pm to midnight we generate the same file again and again without adding any new data.

1.2.6 pushlog

This helps us query information about Mozilla's Mercurial repositories.

Documentation found in here: http://mozilla-version-control-tools.readthedocs.org/en/latest/hgmo/pushlog.html

Important notes from the pushlog documentation:

```
When implementing agents that consume pushlog data, please keep in mind
the following best practices:

* Query by push ID, not by changeset or date.
* Always specify a startID and endID.
* Try to avoid full if possible.
* Always use the latest format version.
* Don't be afraid to ask for a new pushlog feature to make your life easier.
```

mozci.sources.pushlog.**query_pushid_range**(*repo_url*, *start_id*, *end_id*, *version=2*)
Return an ordered list of revisions (newest push id first).

repo - represents the URL to clone a repo start_id - from which pushid to start with (oldest) end_id - from which pushid to end with (most recent) version - version of json-pushes to use (see docs)

mozci.sources.pushlog.**query_repo_tip**(*repo_url*)
Return the tip of a branch.

mozci.sources.pushlog.**query_revision_info**(*repo_url*, *revision*, *full=False*)
Return a dictionary with meta-data about a push including:

- changesets
- date
- user

mozci.sources.pushlog.**query_revisions_range**(*repo_url*, *from_revision*, *to_revision*, *version=2*, *tipsonly=1*)
Return an ordered list of revisions (by date - oldest (starting) first).

repo - represents the URL to clone a repo from_revision - from which revision to start with (oldest) to_revision - from which revision to end with (newest) version - version of json-pushes to use (see docs)

mozci.sources.pushlog.**query_revisions_range_from_revision_before_and_after**(*repo_url*, *revision*, *before*, *after*)

Get the start and end revisions based on the number of revisions before and after.

mozci.sources.pushlog.**valid_revision**(*repo_url*, *revision*)

Verify that a revision exists in a given branch.

1.3 Scripts

The scripts directory contains various scripts that have various specific uses and help drive the development of Mozilla CI tools. All the scripts are located in mozci/scripts directory.

For each script, separate instructions are given if you have installed mozci via:

1. "pip install" or
2. Cloned the repository from source for development purposes

1.3.1 trigger.py

This script allows you to trigger a list of buildernames many times across a range of pushes. You can either:

1. give a start and end revision
2. go back N revisions from a given revision
3. use a range based on a delta from a given revision
4. find the last good known job and trigger everything missing up to it

If you have done "pip install", run via commandline:

```
$ mozci-trigger
```

In cloned repository for development:

```
$ python trigger.py
```

Usage:

```
usage: trigger.py [-h] -b BUILDERNAME -r REV [--times TIMES] [--skips SKIPS]
                  [--from-rev FROM_REV] [--max-revisions MAX_REVISIONS]
                  [--dry-run] [--debug] [--delta DELTA]
                  [--back-revisions BACK_REVISIONS] [--backfill]

optional arguments:
  -h, --help            show this help message and exit
  -b BUILDERNAME, --buildername BUILDERNAME
                        Comma-separated list of buildernames used in Treeherder.
  -r REV, --revision REV
                        The 12 characters representing a revision (most
                        recent).
```

```
  --times TIMES         Total number of jobs to have on a push. Eg: If there
                        is 1 job and you want to trigger 1 more time, do
                        --times=2.
  --skips SKIPS         Specify the step size to skip after every retrigger.
  --from-rev FROM_REV   The 12 character representing the oldest push to start
                        from.
  --max-revisions MAX_REVISIONS
                        This flag is used with --backfill. This flag limits how
                        many revisions we will look back until we find the last
                        revision where there was a good job.
  --dry-run             flag to test without actual push.
  --debug               set debug for logging.
  --delta DELTA         Number of jobs to add/subtract from push revision.
  --back-revisions BACK_REVISIONS
                        Number of revisions to go back from current revision
                        (--rev).
  --backfill            We will trigger jobs starting from --rev in reverse
                        chronological order until we find the last revision
                        where there was a good job.
```

1.3.2 generate_triggercli.py

This script allows you to generate a bunch of command line commands that would allow you to investigate the revision to blame for an intermittent orange. You have to specify the bug number for the intermittent orange you're investigating and this script will you give you the scripts you need to run to backfill the jobs you need.

1.3.3 misc/write_tests_per_platform_graph.py

This script generates a graph of every platform and test in try.

The graph contains two main keys: 'opt' and 'debug'. Inside each there is a key for each platform.

For every platform there is a key for every upstream builder, containing a list of its downstream builders and a key 'tests' that contains a list of every test that is run in that platform.

For example, the key 'android-x86' in 'opt' is:

```
"android-x86": {
          "Android 4.2 x86 try build": [
              "Android 4.2 x86 Emulator try opt test androidx86-set-4"
          ],
          "tests": ["androidx86-set-4"]
      },
```

This script is run nightly and its output can be found at http://people.mozilla.org/~armenzg/permanent/graph.json

If you could use a graph like this but the current format is not ideal, please file an issue.

1.3.4 triggerbyfilters.py

This script retriggers N times every job that matches –includes and doesn't match –exclude.

If you have done "pip install", run via commandline:

```
$ mozci-triggerbyfilters
```

In cloned repository for development:

```
$ python triggerbyfilters.py
```

Usage:

```
usage: th_filters.py [-h] REPO REVISION -i INCLUDES [-e EXCLUDE]
                     [--times TIMES] [--limit LIM] [--dry-run] [--debug]

positional arguments:
  repo                  Branch name
  rev                   The 12 character representing a revision (most
                        recent).

optional arguments:
  -h, --help            show this help message and exit
  -i INCLUDES, --includes INCLUDES
                        comma-separated treeherder filters to include.
  -e EXCLUDE, --exclude EXCLUDE
                        comma-separated treeherder filters to exclude.
  --times TIMES         Total number of jobs to have on a push. Eg: If there
                        is 1 job and you want to trigger 1 more time, do
                        --times=2.
  --limit LIM           Maximum number of buildernames to trigger.
  --dry-run             flag to test without actual push.
  --debug               set debug for logging.
```

For example, if you want to retrigger all web-platform-tests on cedar in a debug platform 5 times:

```
python triggerbyfilters.py cedar REVISION --includes "web-platform-tests,debug" --times 5
```

If you want the same thing but without web-platform-tests-2:

```
python triggerbyfilters.py cedar REVISION --includes "web-platform-tests,debug" --exclude "web-platfo
```

Note: this script currently only does string matching on buildernames, so some queries may not be supported. If you encounter any problem, please file an issue.

CHAPTER 2

Resources

- Source
- Docs
- Issues
- Milestones
- Pypi

CHAPTER 3

Indices and tables

- genindex
- modindex

m

B

D

F

G

I

L

M

Q

S

T

V