Mozilla Source Tree Docs

Release 50.0a1

August 02, 2016

Contents

CHAPTER 1

SSL Error Reporting

With the introduction of HPKP, it becomes useful to be able to capture data on pin violations. SSL Error Reporting is an opt-in mechanism to allow users to send data on such violations to mozilla.

1.1 Payload Format

An example report:

```
{
  "hostname":"example.com",
  "port":443,
  "timestamp":1413490449,
  "errorCode":-16384,
  "failedCertChain":[
    ],
  "userAgent":"Mozilla/5.0 (X11; Linux x86_64; rv:36.0) Gecko/20100101 Firefox/36.0",
  "version":1,
  "build":"20141022164419",
  "product":"Firefox",
  "channel":"default"
}
```

Where the data represents the following:

"hostname" The name of the host the connection was being made to.

"port" The TCP port the connection was being made to.

"timestamp" The (local) time at which the report was generated. Seconds since 1 Jan 1970, UTC.

"errorCode" The error code. This is the error code from certificate verification. Here's a small list of the most commonly-encountered errors: https://wiki.mozilla.org/SecurityEngineering/x509Certs#Error_Codes_in_Firefox In theory many of the errors from sslerr.h, secerr.h, and pkixnss.h could be encountered. We're starting with just MOZILLA_PKIX_ERROR_KEY_PINNING_FAILURE, which means that key pinning failed (i.e. there wasn't an intersection between the keys in any computed trusted certificate chain and the expected list of keys for the domain the user is attempting to connect to).

"failedCertChain" The certificate chain which caused the pinning violation (array of base64 encoded PEM)

"user agent" The user agent string of the browser sending the report

"build" The build ID

"product" The product name

"channel" The user's release channel

1.2 Preferences

The following preferences are used by SSL Error reporting:

"security.ssl.errorReporting.enabled" Should the SSL Error Reporting UI be shown on pin violations? Default value: `true`

"security.ssl.errorReporting.url" Where should SSL error reports be sent? Default value: `https://incoming.telemetry.mozilla.org/submit/sslreports/`

"security.ssl.errorReporting.automatic" Should error reports be sent without user interaction. Default value: `false`. Note: this pref is overridden by the value of `security.ssl.errorReporting.enabled` This is only set when specifically requested by the user. The user can set this value (or unset it) by checking the "Automatically report errors in the future" checkbox when about:neterror is displayed for SSL Errors.

CHAPTER 2

Firefox

This is the nascent documentation of the Firefox front-end code.

2.1 Directory Links Architecture and Data Formats

Directory links are enhancements to the new tab experience that combine content Firefox already knows about from user browsing with external content. There are 3 kinds of links:

- directory links fill in additional tiles on the new tab page if there would have been empty tiles because the user has a clean profile or cleared history
- suggested links are shown if certain triggering criteria matches the user's browsing behavior, i.e., if the user has a top site that matches one of several possible sites. E.g., only show a sports suggestion if the user has a sport site as a top site
- enhanced links replace a matching user's visible history tile from the same site but only the visual aspects: title, image, and rollover image

To power the above features, DirectoryLinksProvider module downloads, at most once per 24 hours, the directory source links as JSON with enough data for Firefox to determine what should be shown or not. This module also handles reporting back data about the tiles via asynchronous pings that don't return data from the server.

For the directory source and ping endpoints, the default preference values point to Mozilla key-pinned servers with encryption. No cookies are set by the servers and Firefox enforces this by making anonymous requests.

- default directory source endpoint: https://tiles.services.mozilla.com/v3/links/fetch/%LOCALE%/%CHANNEL%
- default directory ping endpoint: https://tiles.services.mozilla.com/v3/links/

2.1.1 Preferences

There are two main preferences that control downloading links and reporting metrics.

`browser.newtabpage.directory.source`

This endpoint tells Firefox where to download directory source file as a GET request. It should return JSON of the appropriate format containing the relevant links data. The value can be a data URI, e.g., an empty JSON object effectively turns off remote downloading: `data:text/plain,{}`

The preference value will have %LOCALE% and %CHANNEL% replaced by the appropriate values for the build of Firefox, e.g.,

- directory source endpoint: https://tiles.services.mozilla.com/v3/links/fetch/en-US/release

browser.newtabpage.directory.ping

This endpoint tells Firefox where to report Tiles metrics as a POST request. The data is sent as a JSON blob. Setting it to empty effectively turns off reporting of Tiles data.

A path segment will be appended to the endpoint of "view" or "click" depending on the type of ping, e.g.,

- `view` ping endpoint: https://tiles.services.mozilla.com/v3/links/view
- `click` ping endpoint: https://tiles.services.mozilla.com/v3/links/click

2.1.2 Data Flow

When Firefox starts, it checks for a cached directory source file. If one exists, it checks for its timestamp to determine if a new file should be downloaded.

If a directory source file needs to be downloaded, a GET request is made then cacheed and unpacked the JSON into the different types of links. Various checks filter out invalid links, e.g., those with http-hosted images or those that don't fit the allowed suggestions.

When a new tab page is built, DirectoryLinksProvider module provides additional link data that is combined with history link data to determine which links can be displayed or not.

When a new tab page is shown, a `view` ping is sent with relevant tiles data. Similarly, when the user clicks on various parts of tiles (to load the page, pin, block, etc.), a `click` ping is sent with similar data. Both of these can trigger downloading of fresh directory source links if 24 hours have elapsed since last download.

Users can turn off the ping with in-new-tab-page controls.

As the new tab page is rendered, any images for tiles are downloaded if not already cached. The default servers hosting the images are Mozilla CDN that don't use cookies: https://tiles.cdn.mozilla.net/ and Firefox enforces that the images come from mozilla.net or data URIs when using the default directory source.

2.1.3 Source JSON Format

Firefox expects links data in a JSON object with top level keys each providing an array of tile objects. The keys correspond to the different types of links: `directory`, `suggested`, and `enhanced`.

Example

Below is an example directory source file:

```
{
    "directory": [
        {
            "bgColor": "",
            "directoryId": 498,
            "enhancedImageURI": "https://tiles.cdn.mozilla.net/images/d11ba0b3095bb19d8092cd29be9cbb
            "imageURI": "https://tiles.cdn.mozilla.net/images/1332a68badf11e3f7f69bf7364e79c0a7e2753
            "title": "Mozilla Community",
            "type": "affiliate",
            "url": "http://contribute.mozilla.org/"
        }
    ],
```

```
    "enhanced": [
        {
            "bgColor": "",
            "directoryId": 776,
            "enhancedImageURI": "https://tiles.cdn.mozilla.net/images/44a14fc405cebc299ead86514dff0e3
            "imageURI": "https://tiles.cdn.mozilla.net/images/20e24aa2219ec7542cc8cf0fd79f0c81e16ebea
            "title": "TurboTax",
            "type": "sponsored",
            "url": "https://turbotax.intuit.com/"
        }
    ],
    "suggested": [
        {
            "adgroup_name": "open-source browser",
            "bgColor": "#cae1f4",
            "check_inadjacency": true,
            "directoryId": 702,
            "explanation": "Suggested for %1$S enthusiasts who visit sites like %2$S",
            "frecent_sites": [
                "addons.mozilla.org",
                "air.mozilla.org",
                "blog.mozilla.org",
                "bugzilla.mozilla.org",
                "developer.mozilla.org",
                "etherpad.mozilla.org",
                "hacks.mozilla.org",
                "hg.mozilla.org",
                "mozilla.org",
                "planet.mozilla.org",
                "quality.mozilla.org",
                "support.mozilla.org",
                "treeherder.mozilla.org",
                "wiki.mozilla.org"
            ],
            "frequency_caps": {"daily": 3, "total": 10},
            "imageURI": "https://tiles.cdn.mozilla.net/images/9ee2b265678f2775de2e4bf680df600b502e603
            "time_limits": {"start": "2014-01-01T00:00:00.000Z", "end": "2014-02-01T00:00:00.000Z"},
            "title": "Thanks for testing!",
            "type": "affiliate",
            "url": "https://www.mozilla.com/firefox/tiles"
        }
    ]
}
```


Link Object

Each link object has various values that Firefox uses to display a tile:

- `url` - string url for the page to be loaded when the tile is clicked. Only https and http URLs are allowed.
- `title` - string that appears below the tile.
- `type` - string relationship of the link to Mozilla. Expected values: affiliate, organic, sponsored.
- `imageURI` - string url for the tile image to show. Only https and data URIs are allowed.
- `enhancedImageURI` - string url for the image to be shown before the user hovers. Only https and data URIs are allowed.

- `bgColor` - string css color for additional fill background color.
- `directoryId` - id of the tile to be used during ping reporting

Suggested Link Object Extras

A suggested link has additional values:

- `adgroup_name` - string to override the hardcoded display name of the triggering set of sites in Firefox.
- `check_inadjacency` - boolean if true prevents the suggested link from being shown if the new tab page is showing a site from an inadjacency list.
- `explanation` - string to override the default explanation that appears below a Suggested Tile. %1$S is replaced by the triggering adgroup name and %2$S is replaced by the triggering site.
- `frecent_sites` - array of strings of the sites that can trigger showing a Suggested Tile if the user has the site in one of the top 100 most-frecent pages.
- `frequency_caps` - an object consisting of daily and total frequency caps that limit the number of times a Suggested Tile can be shown in the new tab per day and overall.
- `time_limits` - an object consisting of start and end timestamps specifying when a Suggested Tile may start and has to stop showing in the newtab. The timestamp is expected in ISO_8601 format: '2014-01-10T20:00:00.000Z'

The inadjacency list is packaged with Firefox as base64-encoded 1-way-hashed sites that tend to have adult, gambling, alcohol, drug, and similar content. Its location: chrome://browser/content/newtab/newTab.inadjacent.json

The preapproved arrays follow a policy for determining what topic grouping is allowed as well as the composition of a grouping. The topics are broad uncontroversial categories, e.g., Mobile Phone, News, Technology, Video Game, Web Development. There are at least 5 sites within a grouping, and as many popular sites relevant to the topic are included to avoid having one site be clearly dominant. These requirements provide some deniability of which site actually triggered a suggestion during ping reporting, so it's more difficult to determine if a user has gone to a specific site.

2.1.4 Ping JSON Format

Firefox reports back an action and the state of tiles on the new tab page based on the user opening a new tab or clicking a tile. The top level keys of the ping:

- `locale` - string locale of the Firefox build
- `tiles` - array of tiles ping objects

An additional key at the top level indicates which action triggered the ping. The value associated to the action key is the 0-based index into the tiles array of which tile triggered the action. Valid actions: block, click, pin, sponsored, sponsored_link, unpin, view. E.g., if the second tile is being clicked, the ping will have `"click": 1`

Example

Below is an example `click` ping with 3 tiles: a pinned suggested tile followed by a history tile and a directory tile. The first tile is being blocked:

```
{
    "locale": "en-US",
    "tiles": [
        {
```

```
            "id": 702,
            "pin": 1,
            "past_impressions": {"total": 5, "daily": 1},
        },
        {},
        {
            "id": 498,
        }
    ],
    "block": 0
}
```

Tiles Ping Object

Each tile of the new tab page is reported back as part of the ping with some or none of the following optional values:

- `id` - id that was provided as part of the downloaded link object (for all types of links: directory, suggested, enhanced); not present if the tile was created from user behavior, e.g., visiting pages
- `past_impressions` - number of impressions (new tab "views") a suggested tile was shown before it was clicked, pinned or blocked. Where the "total" counter is the overall number of impressions accumulated prior to a click action, and "daily" counter is the number impressions occurred on same calendar day of a click. This infomration is submitted once per a suggested tile upon click, pin or block
- `pinned` - 1 if the tile is pinned; not present otherwise
- `pos` - integer position if the tile is not in the natural order, e.g., a pinned tile after an empty slot; not present otherwise
- `score` - integer truncated score based on the tile's frecency; not present if 0
- `url` - empty string if it's an enhanced tile; not present otherwise

2.2 UITelemetry data format

UI Telemetry sends its data as a JSON blob. This document describes the different parts of the JSON blob.

2.2.1 `toolbars`

This tracks the state of the user's UI customizations. It has the following properties:

- `sizemode` - string indicating whether the window is in maximized, normal (restored) or fullscreen mode;
- `bookmarksBarEnabled` - boolean indicating whether the bookmarks bar is visible;
- `menuBarEnabled` - boolean indicating whether the menu bar is visible (always false on OS X);
- `titleBarEnabled` - boolean indicating whether the (real) titlebar is visible (rather than having tabs in the titlebar);
- `defaultKept` - list of strings identifying toolbar buttons and items that are still in their default position. Only the IDs of builtin widgets are sent (ie not add-on widgets);
- `defaultMoved` - list of strings identifying toolbar buttons and items that are no longer in their default position, but have not been removed to the palette. Only the IDs of builtin widgets are sent (ie not add-on widgets);

- `nondefaultAdded` - list of strings identifying toolbar buttons and items that have been added from the palette. Only the IDs of builtin widgets are sent (ie not add-on widgets);
- `defaultRemoved` - list of strings identifying toolbar buttons and items that are in the palette that are elsewhere by default. Only the IDs of builtin widgets are sent (ie not add-on widgets);
- `addonToolbars` - the number of non-default toolbars that are customizable. 1 by default because it counts the add-on bar shim;
- `visibleTabs` - array of the number of visible tabs per window;
- `hiddenTabs` - array of the number of hidden tabs per window (ie tabs in panorama groups which are not the current group);
- `countableEvents` - please refer to the next section.
- `durations` - an object mapping descriptions to duration records, which records the amount of time a user spent doing something. Currently only has one property:
  - `customization` - how long a user spent customizing the browser. This is an array of objects, where each object has a `duration` property indicating the time in milliseconds, and a `bucket` property indicating a bucket in which the duration info falls.

2.2.2 countableEvents

Countable events are stored under the `toolbars` section. They count the number of times certain events happen. No timing or other correlating information is stored - purely the number of times things happen.

`countableEvents` contains a list of buckets as its properties. A bucket represents the state the browser was in when these events occurred, such as currently running an interactive tour. There are 3 types of buckets:

- `__DEFAULT__` - No bucket, for times when the browser is not in any special state.
- `bucket_<NAME>` - Normal buckets, for when the browser is in a special state. The `<NAME>` in the bucket ID is the name associated with the bucket and may be further broken down into parts by the `|` character.
- `bucket_<NAME>|<INTERVAL>` - Expiring buckets, which are similar to a countdown timer. The `<INTERVAL>` in the bucket ID describes the time interval the recorded event happened in. The intervals are `1m` (one minute), `3m` (three minutes), `10m` (ten minutes), and `1h` (one hour). After one hour, the `__DEFAULT__` bucket is automatically used again.

Each bucket is an object with the following properties:

- `click-builtin-item` is an object tracking clicks on builtin customizable toolbar items, keyed off the item IDs, with an object for each item with keys `left`, `middle` and `right` each storing a number indicating how often the respective type of click has happened.
- `click-menu-button` is the same, except the item ID is always 'button'.
- `click-bookmarks-bar` is the same, with the item IDs being replaced by either `container` for clicks on bookmark or livemark folders, and `item` for individual bookmarks.
- `click-menubar` is similar, with the item IDs being replaced by one of `menu`, `menuitem` or `other`, depending on the kind of item clicked. Note that this is not tracked on OS X, where we can't listen for these events because of the global menubar.
- `click-bookmarks-menu-button` is also similar, with the item IDs being replaced by:
  - `menu` for clicks on the 'menu' part of the item;
  - `add` for clicks that add a bookmark;
  - `edit` for clicks that open the panel to edit an existing bookmark;

  - **`in-panel` for clicks when the button is in the menu panel, and clicking it does none of the** above;

- `customize` tracks different types of customization events without the `left`, `middle` and `right` distinctions. The different events are the following, with each storing a count of the number of times they occurred:
  - `start` counts the number of times the user starts customizing;
  - `add` counts the number of times an item is added somewhere from the palette;
  - `move` counts the number of times an item is moved somewhere else (but not to the palette);
  - `remove` counts the number of times an item is removed to the palette;
  - `reset` counts the number of times the 'restore defaults' button is used;
- **`search` is an object tracking searches of various types, keyed off the search** location, storing a number indicating how often the respective type of search has happened.
  - There are also two special keys that mean slightly different things.
    * `urlbar-keyword` records searches that would have been an invalid-protocol error, but are now keyword searches. They are also counted in the `urlbar` keyword (along with all the other urlbar searches).
    * `selection` searches records selections of search suggestions. They include the source, the index of the selection, and the kind of selection (mouse or enter key). Selection searches are also counted in their sources.

2.2.3 `UITour`

The UITour API provides ways for pages on trusted domains to safely interact with the browser UI and request it to perform actions such as opening menus and showing highlights over the browser chrome - for the purposes of interactive tours. We track some usage of this API via the `UITour` object in the UI Telemetry output.

Each page is able to register itself with an identifier, a `Page ID`. A list of Page IDs that have been seen over the last 8 weeks is available via `seenPageIDs`.

Page IDs are also used to identify buckets for *countableEvents*, in the following circumstances:

- The current tab is a tour page. This will be a normal bucket with the name `UITour|<PAGEID>`, where `<PAGEID>` is the page's registered ID. This will result in bucket IDs such as `bucket_UITour|australis-tour`.
- A tour tab is open but another tab is active. This will be an expiring bucket with the name `UITour|<PAGEID>|inactive`. This will result in bucket IDs such as `bucket_UITour|australis-tour|inactive|1m`.
- A tour tab has recently been open but has been closed. This will be an expiring bucket with the name `UITour|<PAGEID>|closed`. This will result in bucket IDs such as `bucket_UITour|australis-tour|closed|10m`.

2.2.4 `contextmenu`

We track context menu interactions to figure out which ones are most often used and/or how effective they are. In the `contextmenu` object, we first store things per-bucket. Next, we divide the following different context menu situations:

- `selection` if there is content on the page that's selected on which the user clicks;
- `link` if the user opened the context menu for a link

- `image-link` if the user opened the context menu on an image or canvas that's a link;
- `image` if the user opened the context menu on an image (that isn't a link);
- `canvas` if the user opened the context menu on a canvas (that isn't a link);
- `media` if the user opened the context menu on an HTML video or audio element;
- `input` if the user opened the context menu on a text input element;
- `social` if the user opened the context menu inside a social frame;
- `other` for all other openings of the content menu;

Each of these objects (if they exist) then gets a "withcustom" and/or a "withoutcustom" property for context menus opened with custom page-created items and without them, and each of those properties holds an object with IDs corresponding to a count of how often an item with that ID was activated in the context menu. Only builtin context menu items are tracked, and besides those items there are four special items which get counts:

- `close-without-interaction` is incremented when the user closes the context menu without interacting with it;
- `custom-page-item` is incremented when the user clicks an item that was created by the page;
- `unknown` is incremented when an item without an ID was clicked;
- `other-item` is incremented when an add-on-provided menuitem is clicked.

CHAPTER 3

Telemetry Experiments

Telemetry Experiments is a feature of Firefox that allows the installation of add-ons called experiments to a subset of the Firefox population for the purposes of experimenting with changes and collecting data on specific aspects of application usage.

3.1 Experiments Manifests

Experiments Manifests are documents that describe the set of active experiments a client may run.

Experiments Manifests are fetched periodically by clients. When fetched, clients look at the experiments within the manifest and determine which experiments are applicable. If an experiment is applicable, the client may download and start the experiment.

3.1.1 Manifest Format

Manifests are JSON documents where the main element is an object.

The *schema* of the object is versioned and defined by the presence of a top-level `version` property, whose integer value is the schema version used by that manifest. Each version is documented in the sections below.

Version 1

Version 1 is the original manifest format.

The following properties may exist in the root object:

experiments An array of objects describing candidate experiments. The format of these objects is documented below.

An array is used to create an explicit priority of experiments. Experiments listed at the beginning of the array take priority over experiments that follow.

Experiments Objects

Each object in the `experiments` array may contain the following properties:

id (required) String identifier of this experiment. The identifier should be treated as opaque by clients. It is used to uniquely identify an experiment for all of time.

xpiURL (required) String URL of the XPI that implements this experiment.

If the experiment is activated, the client will download and install this XPI.

xpiHash (required) String hash of the XPI that implements this experiment.

The value is composed of a hash identifier followed by a colon followed by the hash value. e.g. *sha1:f677428b9172e22e9911039aef03f3736e7f78a7*. *sha1* and *sha256* are the two supported hashing mechanisms. The hash value is the hex encoding of the binary hash.

When the client downloads the XPI for the experiment, it should compare the hash of that XPI against this value. If the hashes don't match, the client should not install the XPI.

Clients may also use this hash as a means of determining when an experiment's XPI has changed and should be refreshed.

startTime Integer seconds since UNIX epoch that this experiment should start. Clients should not start an experiment if *now()* is less than this value.

maxStartTime (optional) Integer seconds since UNIX epoch after which this experiment should no longer start.

Some experiments may wish to impose hard deadlines after which no new clients should activate the experiment. This property may be used to facilitate that.

endTime Integer seconds since UNIX epoch after which this experiment should no longer run. Clients should cease an experiment when the current time is beyond this value.

maxActiveSeconds Integer seconds defining the max wall time this experiment should be active for.

The client should deactivate the experiment this many seconds after initial activation.

This value only involves wall time, not browser activity or session time.

appName Array of application names this experiment should run on.

An application name comes from `nsIXULAppInfo.name`. It is a value like `Firefox`, `Fennec`, or *B2G*.

The client should compare its application name against the members of this array. If a match is found, the experiment is applicable.

minVersion (optional) String version number of the minimum application version this experiment should run on.

A version number is something like `27.0.0` or `28`.

The client should compare its version number to this value. If the client's version is greater or equal to this version (using a version-aware comparison function), the experiment is applicable.

If this is not specified, there is no lower bound to versions this experiment should run on.

maxVersion (optional) String version number of the maximum application version this experiment should run on.

This is similar to `minVersion` except it sets the upper bound for application versions.

If the client's version is less than or equal to this version, the experiment is applicable.

If this is not specified, there is no upper bound to versions this experiment should run on.

version (optional) Array of application versions this experiment should run on.

This is similar to `minVersion` and `maxVersion` except only a whitelisted set of specific versions are allowed.

The client should compare its version to members of this array. If a match is found, the experiment is applicable.

minBuildID (optional) String minimum Build ID this experiment should run on.

Build IDs are values like `201402261424`.

The client should perform a string comparison of its Build ID against this value. If its value is greater than or equal to this value, the experiment is applicable.

maxBuildID (optional) String maximum Build ID this experiment should run on.

This is similar to `minBuildID` except it sets the upper bound for Build IDs.

The client should perform a string comparison of its Build ID against this value. If its value is less than or equal to this value, the experiment is applicable.

buildIDs (optional) Array of Build IDs this experiment should run on.

This is similar to `minBuildID` and `maxBuildID` except only a whitelisted set of Build IDs are considered.

The client should compare its Build ID to members of this array. If a match is found, the experiment is applicable.

os (optional) Array of operating system identifiers this experiment should run on.

Values for this array come from `nsIXULRuntime.OS`.

The client will compare its operating system identifier to members of this array. If a match is found, the experiment is applicable to the client.

channel (optional) Array of release channel identifiers this experiment should run on.

The client will compare its channel to members of this array. If a match is found, the experiment is applicable.

If this property is not defined, the client should assume the experiment is to run on all channels.

locale (optional) Array of locale identifiers this experiment should run on.

A locale identifier is a string like `en-US` or `zh-CN` and is obtained by looking at `nsIXULChromeRegistry.getSelectedLocale("global")`.

The client should compare its locale identifier to members of this array. If a match is found, the experiment is applicable.

If this property is not defined, the client should assume the experiment is to run on all locales.

sample (optional) Decimal number indicating the sampling rate for this experiment.

This will contain a value between `0.0` and `1.0`. The client should generate a random decimal between `0.0` and `1.0`. If the randomly generated number is less than or equal to the value of this field, the experiment is applicable.

disabled (optional) Boolean value indicating whether an experiment is disabled.

Normally, experiments are deactivated after a certain time has passed or after the experiment itself determines it no longer needs to run (perhaps it collected sufficient data already).

This property serves as a backup mechanism to remotely disable an experiment before it was scheduled to be disabled. It can be used to kill experiments that are found to be doing wrong or bad things or that aren't useful.

If this property is not defined or is false, the client should assume the experiment is active and a candidate for activation.

frozen (optional) Boolean value indicating this experiment is frozen and no longer accepting new enrollments.

If a client sees a true value in this field, it should not attempt to activate an experiment.

jsfilter (optional) JavaScript code that will be evaluated to determine experiment applicability.

This property contains the string representation of JavaScript code that will be evaluated in a sandboxed environment using JavaScript's `eval()`.

The string is expected to contain the definition of a JavaScript function `filter(context)`. This function receives as its argument an object holding application state. See the section below for the definition of this object.

The purpose of this property is to allow experiments to define complex rules and logic for evaluating experiment applicability in a manner that is privacy conscious and doesn't require the transmission of excessive data.

The return value of this filter indicates whether the experiment is applicable. Functions should return true if the experiment is applicable.

If an experiment is not applicable, they should throw an Error whose message contains the reason the experiment is not applicable. This message may be logged and sent to remote servers, so it should not contain private or otherwise sensitive data that wouldn't normally be submitted.

If a falsey (or undefined) value is returned, the client should assume the experiment is not applicable.

If this property is not defined, the client does not consider a custom JavaScript filter function when determining whether an experiment is applicable.

JavaScript Filter Context Objects

The object passed to a `jsfilter` `filter()` function contains the following properties:

healthReportSubmissionEnabled This property contains a boolean indicating whether Firefox Health Report has its data submission flag enabled (whether Firefox Health Report is sending data to remote servers).

healthReportPayload This property contains the current Firefox Health Report payload.

The payload format is documented at *Payload Format*.

telemetryPayload This property contains the current Telemetry payload.

The evaluation sandbox for the JavaScript filters may be destroyed immediately after `filter()` returns. This function should not assume async code will finish.

3.1.2 Experiment Applicability and Client Behavior

The point of an experiment manifest is to define which experiments are available and where and how to run them. This section explains those rules in more detail.

Many of the properties in *Experiment Objects* are related to determining whether an experiment should run on a given client. This evaluation is performed client side.

1. Multiple conditions in an experiment

If multiple conditions are defined for an experiment, the client should combine each condition with a logical *AND*: all conditions must be satisfied for an experiment to run. If one condition fails, the experiment is not applicable.

2. Active experiment disappears from manifest

If a specific experiment disappears from the manifest, the client should continue conducting an already-active experiment. Furthermore, the client should remember what the expiration events were for an experiment and honor them.

The rationale here is that we want to prevent an accidental deletion or temporary failure on the server to inadvertantly deactivate supposed-to-be-active experiments. We also don't want premature deletion of an experiment from the manifest to result in indefinite activation periods.

3. Inactive experiment disappears from manifest

If an inactive but scheduled-to-be-active experiment disappears from the manifest, the client should not activate the experiment.

If that experiment reappears in the manifest, the client should not treat that experiment any differently than any other new experiment. Put another way, the fact an inactive experiment disappears and then reappears should not be significant.

The rationale here is that server operators should have complete control of an inactive experiment up to it's go-live date.

4. Re-evaluating applicability on manifest refresh

When an experiment manifest is refreshed or updated, the client should re-evaluate the applicability of each experiment therein.

The rationale here is that the server may change the parameters of an experiment and want clients to pick those up.

5. Activating a previously non-applicable experiment

If the conditions of an experiment change or the state of the client changes to allow an experiment to transition from previously non-applicable to applicable, the experiment should be activated.

For example, if a client is running version 28 and the experiment initially requires version 29 or above, the client will not mark the experiment as applicable. But if the client upgrades to version 29 or if the manifest is updated to require 28 or above, the experiment will become applicable.

6. Deactivating a previously active experiment

If the conditions of an experiment change or the state of the client changes and an active experiment is no longer applicable, that experiment should be deactivated.

7. Calculation of sampling-based applicability

For calculating sampling-based applicability, the client will associate a random value between `0.0` and `1.0` for each observed experiment ID. This random value will be generated the first time sampling applicability is evaluated. This random value will be persisted and used in future applicability evaluations for this experiment.

By saving and re-using the value, the client is able to reliably and consistently evaluate applicability, even if the sampling threshold in the manifest changes.

Clients should retain the randomly-generated sampling value for experiments that no longer appear in a manifest for a period of at least 30 days. The rationale is that if an experiment disappears and reappears from a manifest, the client will not have multiple opportunities to generate a random value that satisfies the sampling criteria.

8. Incompatible version numbers

If a client receives a manifest with a version number that it doesn't recognize, it should ignore the manifest.

9. Usage of old manifests

If a client experiences an error fetching a manifest (server not available) or if the manifest is corrupt, not readable, or compatible, the client may use a previously-fetched (cached) manifest.

10. Updating XPIs

If the URL or hash of an active experiment's XPI changes, the client should fetch the new XPI, uninstall the old XPI, and install the new XPI.

3.1.3 Examples

Here is an example manifest:

```
{
  "version": 1,
  "experiments": [
    {
      "id": "da9d7f4f-f3f9-4f81-bacd-6f0626ffa360",
      "xpiURL": "https://experiments.mozilla.org/foo.xpi",
      "xpiHash": "sha1:cb1eb32b89d86d78b7326f416cf404548c5e0099",
      "startTime": 1393000000,
      "endTime": 1394000000,
      "appName": ["Firefox", "Fennec"],
      "minVersion": "28",
      "maxVersion": "30",
      "os": ["windows", "linux", "osx"],
      "jsfilter": "function filter(context) { return context.healthReportEnabled; }"
    }
  ]
}
```

CHAPTER 4

Build System

4.1 Important Concepts

4.1.1 Glossary

clobber build A build performed with an initially empty object directory. All build actions must be performed.

config.status An executable file produced by **configure** that takes the generated build config and writes out files used to build the tree. Traditionally, config.status writes out a bunch of Makefiles.

configure A generated shell script which detects the current system environment, applies a requested set of build configuration options, and writes out metadata to be consumed by the build system.

incremental build A build performed with the result of a previous build in an object directory. The build should not have to work as hard because it will be able to reuse the work from previous builds.

install manifest A file containing metadata describing file installation rules. A large part of the build system consists of copying files around to appropriate places. We write out special files describing the set of required operations so we can process the actions effeciently. These files are install manifests.

mozconfig A shell script used to configure the build system.

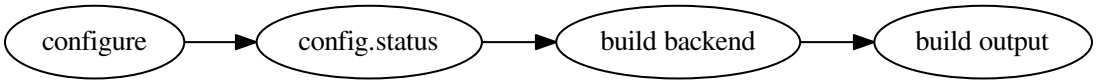
mozinfo An API for accessing a common and limited subset of the build and run-time configuration. See *mozinfo*.

object directory A directory holding the output of the build system. The build system attempts to isolate all file modifications to this directory. By convention, object directories are commonly directories under the source directory prefixed with **obj-**. e.g. **obj-firefox**.

4.1.2 Build System Overview

This document provides an overview on how the build system works. It is targeted at people wanting to learn about internals of the build system. It is not meant for persons who casually interact with the build system. That being said, knowledge empowers, so consider reading on.

The build system is composed of many different components working in harmony to build the source tree. We begin with a graphic overview.



Phase 1: Configuration

Phase 1 centers around the `configure` script, which is a bash shell script. The file is generated from a file called `configure.in` which is written in M4 and processed using Autoconf 2.13 to create the final configure script. You don't have to worry about how you obtain a `configure` file: the build system does this for you.

The primary job of `configure` is to determine characteristics of the system and compiler, apply options passed into it, and validate everything looks OK to build. The primary output of the `configure` script is an executable file in the object directory called `config.status`. `configure` also produces some additional files (like `autoconf.mk`). However, the most important file in terms of architecture is `config.status`.

The existence of a `config.status` file may be familiar to those who have worked with Autoconf before. However, Mozilla's `config.status` is different from almost any other `config.status` you've ever seen: it's written in Python! Instead of having our `configure` script produce a shell script, we have it generating Python.

Now is as good a time as any to mention that Python is prevalent in our build system. If we need to write code for the build system, we do it in Python. That's just how we roll. For more, see *Python and the Build System*.

`config.status` contains 2 parts: data structures representing the output of `configure` and a command-line interface for preparing/configuring/generating an appropriate build backend. (A build backend is merely a tool used to build the tree - like GNU Make or Tup). These data structures essentially describe the current state of the system and what the existing build configuration looks like. For example, it defines which compiler to use, how to invoke it, which application features are enabled, etc. You are encouraged to open up `config.status` to have a look for yourself!

Once we have emitted a `config.status` file, we pass into the realm of phase 2.

Phase 2: Build Backend Preparation and the Build Definition

Once `configure` has determined what the current build configuration is, we need to apply this to the source tree so we can actually build.

What essentially happens is the automatically-produced `config.status` Python script is executed as soon as `configure` has generated it. `config.status` is charged with the task of tell a tool how to build the tree. To do this, `config.status` must first scan the build system definition.

The build system definition consists of various `moz.build` files in the tree. There is roughly one `moz.build` file per directory or per set of related directories. Each `moz.build` files defines how its part of the build config works. For example it says *I want these C++ files compiled* or *look for additional information in these directories.* config.status starts with the `moz.build` file from the root directory and then descends into referenced `moz.build` files by following `DIRS` variables or similar.

As the `moz.build` files are read, data structures describing the overall build system definition are emitted. These data structures are then fed into a build backend, which then performs actions, such as writing out files to be read by a build tool. e.g. a `make` backend will write a `Makefile`.

When `config.status` runs, you'll see the following output:

```
Reticulating splines...
Finished reading 1096 moz.build files into 1276 descriptors in 2.40s
Backend executed in 2.39s
2188 total backend files. 0 created; 1 updated; 2187 unchanged
Total wall time: 5.03s; CPU time: 3.79s; Efficiency: 75%
```

What this is saying is that a total of *1096* `moz.build` files were read. Altogether, *1276* data structures describing the build configuration were derived from them. It took *2.40s* wall time to just read these files and produce the data structures. The *1276* data structures were fed into the build backend which then determined it had to manage *2188* files derived from those data structures. Most of them already existed and didn't need changed. However, *1* was updated as a result of the new configuration. The whole process took *5.03s*. Although, only *3.79s* was in CPU time. That likely means we spent roughly *25%* of the time waiting on I/O.

For more on how `moz.build` files work, see *moz.build Files*.

Phase 3: Invokation of the Build Backend

When most people think of the build system, they think of phase 3. This is where we take all the code in the tree and produce Firefox or whatever application you are creating. Phase 3 effectively takes whatever was generated by phase 2 and runs it. Since the dawn of Mozilla, this has been make consuming Makefiles. However, with the transition to moz.build files, you may soon see non-Make build backends, such as Tup or Visual Studio.

When building the tree, most of the time is spent in phase 3. This is when header files are installed, C++ files are compiled, files are preprocessed, etc.

4.1.3 Supported Configurations

This page attempts to document supported build configurations.

Windows

We support building on Windows XP and newer operating systems using Visual Studio 2010 and newer.

The following are not fully supported by Mozilla (but may work):

- Building without the latest *MozillaBuild* Windows development environment
- Building with Mingw or any other non-Visual Studio toolchain.

OS X

We support building on OS X 10.6 and newer with the OS X 10.6 SDK.

The tree should build with the following OS X releases and SDK versions:

- 10.6 Snow Leopard
- 10.7 Lion
- 10.8 Mountain Lion
- 10.9 Mavericks

The tree requires building with Clang 3.3 and newer. This corresponds to version of 4.2 of Apple's Clang that ships with Xcode. This corresponds to Xcode 4.6 and newer. Xcode 4.6 only runs on OS X 10.7.4 and newer. So, OS X 10.6 users will need to install a non-Apple toolchain. Running `mach bootstrap` should install an appropriate toolchain from Homebrew or MacPorts automatically.

The tree should build with GCC 4.4 and newer on OS X. However, this build configuration isn't as widely used (and differs from what Mozilla uses to produce OS X builds), so it's recommended to stick with Clang.

Linux

Linux 2.6 and later kernels are supported.

Most distributions are supported as long as the proper package dependencies are in place. Running `mach bootstrap` should install packages for popular Linux distributions. `configure` will typically detect missing dependencies and inform you how to disable features to work around unsatisfied dependencies.

Clang 3.3 or GCC 4.4 is required to build the tree.

4.1.4 mozconfig Files

mozconfig files are used to configure how a build works.

mozconfig files are actually shell scripts. They are executed in a special context with specific variables and functions exposed to them.

API

Functions

The following special functions are available to a mozconfig script.

ac_add_options This function is used to declare extra options/arguments to pass into configure.

e.g.:

```
ac_add_options --disable-tests
ac_add_options --enable-optimize
```

mk_add_options This function is used to inject statements into client.mk for execution. It is typically used to define variables, notably the object directory.

e.g.:

```
mk_add_options AUTOCLOBBER=1
```

ac_add_options This is a variant of ac_add_options() which only adds configure options for a specified application. This is only used when building multiple applications through client.mk. This function is typically not needed.

Special mk_add_options Variables

For historical reasons, the method for communicating certain well-defined variables is via mk_add_options(). In this section, we document what those special variables are.

MOZ_OBJDIR This variable is used to define the *object directory* for the current build.

Finding the active mozconfig

Multiple mozconfig files can exist to provide different configuration options for different tasks. The rules for finding the active mozconfig are defined in the *mozbuild.mozconfig.MozconfigLoader.find_mozconfig()* method:

class mozbuild.mozconfig.**MozconfigLoader**(*topsrcdir*)

Handles loading and parsing of mozconfig files.

find_mozconfig(*env={'LANG': 'C.UTF-8', 'READTHEDOCS_PROJECT': 'gfritzsche-demo', 'READTHEDOCS': 'True', 'APPDIR': '/app', 'DEBIAN_FRONTEND': 'noninteractive', 'OLDPWD': '/', 'HOSTNAME': 'build-4258433-project-55928-gfritzsche-demo', u'SHELL': u'/bin/bash', 'PWD': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs', 'BIN_PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/envs/latest/bin', 'READTHEDOCS_VERSION': 'latest', 'PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs/_build/latex/_venv/bin:/home/docs/checkouts/readthedocs.org/user_build*
demo/envs/latest/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin', 'HOME': '/home/docs'})

Find the active mozconfig file for the current environment.

This emulates the logic in mozconfig-find.

1. If ENV[MOZCONFIG] is set, use that
2. If $TOPSRCDIR/mozconfig or $TOPSRCDIR/.mozconfig exists, use it.
3. If both exist or if there are legacy locations detected, error out.

The absolute path to the found mozconfig will be returned on success. None will be returned if no mozconfig could be found. A MozconfigFindException will be raised if there is a bad state, including conditions from #3 above.

4.1.5 moz.build Files

`moz.build` files are the mechanism by which tree metadata (notably the build configuration) is defined.

Directories in the tree contain `moz.build` files which declare functionality for their respective part of the tree. This includes things such as the list of C++ files to compile, where to find tests, etc.

`moz.build` files are actually Python scripts. However, their execution is governed by special rules. This is explained below.

moz.build Python Sandbox

As mentioned above, `moz.build` files are Python scripts. However, they are executed in a special Python *sandbox* that significantly changes and limits the execution environment. The environment is so different, it's doubtful most `moz.build` files would execute without error if executed by a vanilla Python interpreter (e.g. `python moz.build`.

The following properties make execution of `moz.build` files special:

1. The execution environment exposes a limited subset of Python.
2. There is a special set of global symbols and an enforced naming convention of symbols.
3. Some symbols are inherited from previously-executed `moz.build` files.

The limited subset of Python is actually an extremely limited subset. Only a few symbols from __builtins__ are exposed. These include `True`, `False`, and `None`. Global functions like `import`, `print`, and `open` aren't available. Without these, `moz.build` files can do very little. *This is by design*.

The execution sandbox treats all `UPPERCASE` variables specially. Any `UPPERCASE` variable must be known to the sandbox before the script executes. Any attempt to read or write to an unknown `UPPERCASE` variable will result in an exception being raised. Furthermore, the types of all `UPPERCASE` variables is strictly enforced. Attempts to assign an incompatible type to an `UPPERCASE` variable will result in an exception being raised.

The strictness of behavior with `UPPERCASE` variables is a very intentional design decision. By ensuring strict behavior, any operation involving an `UPPERCASE` variable is guaranteed to have well-defined side-effects. Previously, when the build configuration was defined in `Makefiles`, assignments to variables that did nothing would go unnoticed. `moz.build` files fix this problem by eliminating the potential for false promises.

After a `moz.build` file has completed execution, only the `UPPERCASE` variables are used to retrieve state.

The set of variables and functions available to the Python sandbox is defined by the *mozbuild.frontend.context* module. The data structures in this module are consumed by the *mozbuild.frontend.reader.MozbuildSandbox* class to construct the sandbox. There are tests to ensure that the set of symbols exposed to an empty sandbox are all defined in the `context` module. This module also contains documentation for each symbol, so nothing can sneak into the sandbox without being explicitly defined and documented.

Reading and Traversing moz.build Files

The process for reading `moz.build` files roughly consists of:

1. Start at the root `moz.build` (`<topsrcdir>/moz.build`).
2. Evaluate the `moz.build` file in a new sandbox.
3. Emit the main *context* and any *sub-contexts* from the executed sandbox.
4. Extract a set of `moz.build` files to execute next.
5. For each additional `moz.build` file, goto #2 and repeat until all referenced files have executed.

From the perspective of the consumer, the output of reading is a stream of `mozbuild.frontend.reader.context.Context` instances. Each `Context` defines a particular aspect of data. Consumers iterate over these objects and do something with the data inside. Each object is essentially a dictionary of all the `UPPERCASE` variables populated during its execution.

Note: Historically, there was only one `context` per `moz.build` file. As the number of things tracked by `moz.build` files grew and more and more complex processing was desired, it was necessary to split these contexts into multiple logical parts. It is now common to emit multiple contexts per `moz.build` file.

Build System Reading Mode

The traditional mode of evaluation of `moz.build` files is what's called *build system traversal mode.* In this mode, the `CONFIG` variable in each `moz.build` sandbox is populated from data coming from `config.status`, which is produced by `configure`.

During evaluation, `moz.build` files often make decisions conditional on the state of the build configuration. e.g. *only compile foo.cpp if feature X is enabled*.

In this mode, traversal of `moz.build` files is governed by variables like `DIRS` and `TEST_DIRS`. For example, to execute a child directory, `foo`, you would add `DIRS += ['foo']` to a `moz.build` file and `foo/moz.build` would be evaluated.

Filesystem Reading Mode

There is an alternative reading mode that doesn't involve the build system and doesn't use `DIRS` variables to control traversal into child directories. This mode is called *filesystem reading mode*.

In this reading mode, the `CONFIG` variable is a dummy, mostly empty object. Accessing all but a few special variables will return an empty value. This means that nearly all `if CONFIG['FOO']:` branches will not be taken.

Instead of using content from within the evaluated `moz.build` file to drive traversal into subsequent `moz.build` files, the set of files to evaluate is controlled by the thing doing the reading.

A single `moz.build` file is not guaranteed to be executable in isolation. Instead, we must evaluate all *parent* `moz.build` files first. For example, in order to evaluate `/foo/moz.build`, one must execute `/moz.build` and have its state influence the execution of `/foo/moz.build`.

Filesystem reading mode is utilized to power the *Files Metadata* feature.

Technical Details

The code for reading `moz.build` files lives in *`mozbuild.frontend.reader`*. The Python sandboxes evaluation results (*`mozbuild.frontend.context.Context`*) are passed into *`mozbuild.frontend.emitter`*, which converts them to classes defined in *`mozbuild.frontend.data`*. Each class in this module defines a domain-specific component of tree metdata. e.g. there will be separate classes that represent a JavaScript file vs a compiled C++ file or test manifests. This means downstream consumers of this data can filter on class types to only consume what they are interested in.

There is no well-defined mapping between `moz.build` file instances and the number of *`mozbuild.frontend.data`* classes derived from each. Depending on the content of the `moz.build` file, there may be 1 object derived or 100.

The purpose of the `emitter` layer between low-level sandbox execution and metadata representation is to facilitate a unified normalization and verification step. There are multiple downstream consumers of the `moz.build`-derived data and many will perform the same actions. This logic can be complicated, so we have a component dedicated to it.

`mozbuild.frontend.reader.BuildReader` and `mozbuild.frontend.reader.TreeMetadataEmitter` have a stream-based API courtesy of generators. When you hook them up properly, the *`mozbuild.frontend.data`* classes are emitted before all `moz.build` files have been read. This means that downstream errors are raised soon after sandbox execution.

Lots of the code for evaluating Python sandboxes is applicable to non-Mozilla systems. In theory, it could be extracted into a standalone and generic package. However, until there is a need, there will likely be some tightly coupled bits.

4.1.6 mozbuild Sandbox Symbols

Sub-Context: Files

Metadata attached to files.

It is common to want to annotate files with metadata, such as which Bugzilla component tracks issues with certain files. This sub-context is where we stick that metadata.

The argument to this sub-context is a file matching pattern that is applied against the host file's directory. If the pattern matches a file whose info is currently being sought, the metadata attached to this instance will be applied to that file.

Patterns are collections of filename characters with `/` used as the directory separate (UNIX-style paths) and `*` and `**` used to denote wildcard matching.

Patterns without the `*` character are literal matches and will match at most one entity.

Patterns with `*` or `**` are wildcard matches. `*` matches files at least within a single directory. `**` matches files across several directories.

`foo.html` Will match only the `foo.html` file in the current directory.

`*.jsm` Will match all `.jsm` files in the current directory.

`**/*.cpp` Will match all `.cpp` files in this and all child directories.

`foo/*.css` Will match all `.css` files in the `foo/` directory.

`bar/*` Will match all files in the `bar/` directory and all of its children directories.

`bar/**` This is equivalent to `bar/*` above.

`bar/**/foo` Will match all `foo` files in the `bar/` directory and all of its children directories.

The difference in behavior between `*` and `**` is only evident if a pattern follows the `*` or `**`. A pattern ending with `*` is greedy. `**` is needed when you need an additional pattern after the wildcard. e.g. `**/foo`.

BUG_COMPONENT

The bug component that tracks changes to these files.

Storage Type `TypedTuple`

Input Type `tuple`

Values are a 2-tuple of unicode describing the Bugzilla product and component. e.g. `('Core', 'Build Config')`.

FINAL

Mark variable assignments as finalized.

Storage Type `bool`

Input Type `bool`

During normal processing, values from newer Files contexts overwrite previously set values. Last write wins. This behavior is not always desired. `FINAL` provides a mechanism to prevent further updates to a variable.

When `FINAL` is set, the value of all variables defined in this context are marked as frozen and all subsequent writes to them are ignored during metadata reading.

See *Finalizing Values* for more info.

IMPACTED_TESTS

File patterns, tags, and flavors for tests relevant to these files.

Storage Type `_TypedRecord`

Input Type `list`

Maps source files to the tests potentially impacted by those files. Tests can be specified by file pattern, tag, or flavor.

For example:

with Files('runtests.py'):

    IMPACTED_TESTS.files += ['**',

    ]

in testing/mochitest/moz.build will suggest that any of the tests under testing/mochitest may be impacted by a change to runtests.py.

File patterns may be made relative to the topsrcdir with a leading '/', so

with Files('httpd.js'):

    IMPACTED_TESTS.files += ['/testing/mochitest/tests/Harness_sanity/**',

    ]

in netwerk/test/httpserver/moz.build will suggest that any change to httpd.js will be relevant to the mochitest sanity tests.

Tags and flavors are sorted string lists (flavors are limited to valid values).

For example:

with Files('toolkit/devtools/*'):

    IMPACTED_TESTS.tags += ['devtools',

    ]

in the root moz.build would suggest that any test tagged 'devtools' would potentially be impacted by a change to a file under toolkit/devtools, and

with Files('dom/base/nsGlobalWindow.cpp'):

    IMPACTED_TESTS.flavors += ['mochitest',

    ]

Would suggest that nsGlobalWindow.cpp is potentially relevant to any plain mochitest.

Variables

A11Y_MANIFESTS

List of manifest files defining a11y tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

ALLOW_COMPILER_WARNINGS

Whether to allow compiler warnings (i.e. *not* treat them as

Storage Type `bool`

Input Type `bool`

errors).

This is commonplace (almost mandatory, in fact) in directories containing third-party code that we regularly update from upstream and thus do not control, but is otherwise discouraged.

ANDROID_APK_NAME

The name of an Android APK file to generate.

Storage Type `unicode`

Input Type `unicode`

ANDROID_APK_PACKAGE

The name of the Android package to generate R.java for, like org.mozilla.gecko.

Storage Type `unicode`

Input Type `unicode`

ANDROID_ASSETS_DIRS

Android assets directories.

Storage Type `_TypedListWithItems`

Input Type `list`

This variable contains a list of directories containing static files to package into an 'assets' directory and merge into an APK file.

ANDROID_ECLIPSE_PROJECT_TARGETS

Defines Android Eclipse project targets.

Storage Type `dict`

Input Type `dict`

This variable should not be populated directly. Instead, it should populated by calling add_android_eclipse{_library}_project().

ANDROID_EXTRA_PACKAGES

The name of extra Android packages to generate R.java for, like ['org.mozilla.other'].

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

ANDROID_EXTRA_RES_DIRS

Android extra package resource directories.

Storage Type `_TypedListWithItems`

Input Type `list`

This variable contains a list of directories containing static files to package into a 'res' directory and merge into an APK file. These directories are packaged into the APK but are assumed to be static unchecked dependencies that should not be otherwise re-distributed.

ANDROID_GENERATED_RESFILES

Android resource files generated as part of the build.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

This variable contains a list of files that are expected to be generated (often by preprocessing) into a 'res' directory as part of the build process, and subsequently merged into an APK file.

ANDROID_INSTRUMENTATION_MANIFESTS

List of manifest files defining Android instrumentation tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

ANDROID_RES_DIRS

Android resource directories.

Storage Type `_TypedListWithItems`

Input Type `list`

This variable contains a list of directories containing static files to package into a 'res' directory and merge into an APK file.

ASFLAGS

Flags passed to the assembler for all of the assembly source files

Storage Type `List`

Input Type `list`

declared in this directory.

Note that the ordering of flags matters here; these flags will be added to the assembler's command line in the same order as they appear in the moz.build file.

BRANDING_FILES

List of files to be installed into the branding directory.

Storage Type `_TypedListWithItems`

Input Type `list`

`BRANDING_FILES` will copy (or symlink, if the platform supports it) the contents of its files to the `dist/branding` directory. Files that are destined for a subdirectory can be specified by accessing a field. For example, to export `foo.png` to the top-level directory and `bar.png` to the directory `images/subdir`, append to `BRANDING_FILES` like so:

```
BRANDING_FILES += ['foo.png']
BRANDING_FILES.images.subdir += ['bar.png']
```

BROWSER_CHROME_MANIFESTS

List of manifest files defining browser chrome tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

CFLAGS

Flags passed to the C compiler for all of the C source files

Storage Type `List`

Input Type `list`

declared in this directory.

Note that the ordering of flags matters here, these flags will be added to the compiler's command line in the same order as they appear in the moz.build file.

CMFLAGS

Flags passed to the Objective-C compiler for all of the Objective-C

Storage Type `List`

Input Type `list`

source files declared in this directory.

Note that the ordering of flags matters here; these flags will be added to the compiler's command line in the same order as they appear in the moz.build file.

CMMFLAGS

Flags passed to the Objective-C++ compiler for all of the

Storage Type `List`

Input Type `list`

Objective-C++ source files declared in this directory.

Note that the ordering of flags matters here; these flags will be added to the compiler's command line in the same order as they appear in the moz.build file.

CONFIGURE_DEFINE_FILES

Output files generated from configure/config.status.

Storage Type `_TypedList`

Input Type `list`

This is a substitute for `AC_CONFIG_HEADER` in autoconf. This is very similar to `CONFIGURE_SUBST_FILES` except the generation logic takes into account the values of `AC_DEFINE` instead of `AC_SUBST`.

CONFIGURE_SUBST_FILES

Output files that will be generated using configure-like substitution.

Storage Type `_TypedList`

Input Type `list`

This is a substitute for `AC_OUTPUT` in autoconf. For each path in this list, we will search for a file in the srcdir having the name `{path}.in`. The contents of this file will be read and variable patterns like `@foo@` will be substituted with the values of the `AC_SUBST` variables declared during configure.

CPP_UNIT_TESTS

Compile a list of C++ unit test names.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

Each name in this variable corresponds to an executable built from the corresponding source file with the same base name.

If the configuration token `BIN_SUFFIX` is set, its value will be automatically appended to each name. If a name already ends with `BIN_SUFFIX`, the name will remain unchanged. This variable is only available in templates.

CRASHTEST_MANIFESTS

List of manifest files defining crashtests.

Storage Type `_OrderedListWithAction`

Input Type `list`

These are commonly named crashtests.list.

CXXFLAGS

Flags passed to the C++ compiler for all of the C++ source files

Storage Type `List`

Input Type `list`

declared in this directory.

Note that the ordering of flags matters here; these flags will be added to the compiler's command line in the same order as they appear in the moz.build file.

DEFFILE

The program .def (module definition) file.

Storage Type `unicode`

Input Type `unicode`

This variable can only be used on Windows.

DEFINES

Dictionary of compiler defines to declare.

Storage Type `InitializedDefines`

Input Type `dict`

These are passed in to the compiler as `-Dkey='value'` for string values, `-Dkey=value` for numeric values, or `-Dkey` if the value is True. Note that for string values, the outer-level of single-quotes will be consumed by the shell. If you want to have a string-literal in the program, the value needs to have double-quotes.

Example:

```
DEFINES['NS_NO_XPCOM'] = True
DEFINES['MOZ_EXTENSIONS_DB_SCHEMA'] = 15
DEFINES['DLL_SUFFIX'] = '".so"'
```

This will result in the compiler flags `-DNS_NO_XPCOM`, `-DMOZ_EXTENSIONS_DB_SCHEMA=15`, and `-DDLL_SUFFIX='".so"'`, respectively. These could also be combined into a single update:

```
DEFINES.update({
    'NS_NO_XPCOM': True,
    'MOZ_EXTENSIONS_DB_SCHEMA': 15,
    'DLL_SUFFIX': '".so"',
})
```

DELAYLOAD_DLLS

Delay-loaded DLLs.

Storage Type `List`

Input Type `list`

This variable contains a list of DLL files which the module being linked should load lazily. This only has an effect when building with MSVC.

DIRS

Child directories to descend into looking for build frontend files.

Storage Type `_TypedList`

Input Type `list`

This works similarly to the `DIRS` variable in make files. Each str value in the list is the name of a child directory. When this file is done parsing, the build reader will descend into each listed directory and read the frontend file there. If there is no frontend file, an error is raised.

Values are relative paths. They can be multiple directory levels above or below. Use `..` for parent directories and `/` for path delimiters.

DISABLE_STL_WRAPPING

Disable the wrappers for STL which allow it to work with C++ exceptions

Storage Type `bool`

Input Type `bool`

disabled.

DIST_INSTALL

Whether to install certain files into the dist directory.

Storage Type `EnumClass`

Input Type `bool`

By default, some files types are installed in the dist directory, and some aren't. Set this variable to True to force the installation of some files that wouldn't be installed by default. Set this variable to False to force to not install some files that would be installed by default.

This is confusing for historical reasons, but eventually, the behavior will be made explicit.

DIST_SUBDIR

The name of an alternate directory to install files to.

Storage Type `unicode`

Input Type `unicode`

When this variable is present, the results of this directory will end up being placed in the $(DIST_SUBDIR) subdirectory of where it would otherwise be placed.

EXPORTS

List of files to be exported, and in which subdirectories.

Storage Type `_TypedListWithItems`

Input Type `list`

`EXPORTS` is generally used to list the include files to be exported to `dist/include`, but it can be used for other files as well. This variable behaves as a list when appending filenames for export in the top-level directory. Files can also be appended to a field to indicate which subdirectory they should be exported to. For example, to export `foo.h` to the top-level directory, and `bar.h` to `mozilla/dom/`, append to `EXPORTS` like so:

```
EXPORTS += ['foo.h']
EXPORTS.mozilla.dom += ['bar.h']
```

Entries in `EXPORTS` are paths, so objdir paths may be used, but any files listed from the objdir must also be listed in `GENERATED_FILES`.

EXTRA_DSO_LDOPTS

Flags passed to the linker when linking a shared library.

Storage Type `List`

Input Type `list`

Note that the ordering of flags matter here, these flags will be added to the linker's command line in the same order as they appear in the moz.build file.

FILES_PER_UNIFIED_FILE

The number of source files to compile into each unified source file.

Storage Type `int`

Input Type `int`

FINAL_LIBRARY

Library in which the objects of the current directory will be linked.

Storage Type `unicode`

Input Type `unicode`

This variable contains the name of a library, defined elsewhere with `LIBRARY_NAME`, in which the objects of the current directory will be linked.

FINAL_TARGET

The name of the directory to install targets to.

Storage Type `FinalTargetValue`

Input Type `unicode`

The directory is relative to the top of the object directory. The default value is dependent on the values of XPI_NAME and DIST_SUBDIR. If neither are present, the result is dist/bin. If XPI_NAME is present, the result is dist/xpi-stage/$(XPI_NAME). If DIST_SUBDIR is present, then the $(DIST_SUBDIR) directory of the otherwise default value is used.

FINAL_TARGET_FILES

List of files to be installed into the application directory.

Storage Type `_TypedListWithItems`

Input Type `list`

`FINAL_TARGET_FILES` will copy (or symlink, if the platform supports it) the contents of its files to the directory specified by `FINAL_TARGET` (typically `dist/bin`). Files that are destined for a subdirectory can be specified by accessing a field, or as a dict access. For example, to export `foo.png` to the top-level directory and `bar.svg` to the directory `images/do-not-use`, append to `FINAL_TARGET_FILES` like so:

```
FINAL_TARGET_FILES += ['foo.png']
FINAL_TARGET_FILES.images['do-not-use'] += ['bar.svg']
```

FINAL_TARGET_PP_FILES

Like `FINAL_TARGET_FILES`, with preprocessing.

Storage Type `_TypedListWithItems`

Input Type `list`

FORCE_SHARED_LIB

Whether the library in this directory is a shared library.

Storage Type `bool`

Input Type `bool`

This variable is only available in templates.

FORCE_STATIC_LIB

Whether the library in this directory is a static library.

Storage Type `bool`

Input Type `bool`

GENERATED_EVENTS_WEBIDL_FILES

WebIDL source files for generated events.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

These will be parsed and converted to `.cpp` and `.h` files.

GENERATED_FILES

Generic generated files.

Storage Type `StrictOrderingOnAppendListWithFlagsSpecialization`

Input Type `list`

This variable contains a list of files for the build system to generate at export time. The generation method may be declared with optional `script` and `inputs` flags on individual entries. If the optional `script` flag is not present on an entry, it is assumed that rules for generating the file are present in the associated Makefile.in.

Example:

```
GENERATED_FILES += ['bar.c', 'baz.c', 'foo.c']
bar = GENERATED_FILES['bar.c']
bar.script = 'generate.py'
bar.inputs = ['datafile-for-bar']
foo = GENERATED_FILES['foo.c']
foo.script = 'generate.py'
foo.inputs = ['datafile-for-foo']
```

This definition will generate bar.c by calling the main method of generate.py with a open (for writing) file object for bar.c, and the string `datafile-for-bar`. In a similar fashion, the main method of generate.py will also be called with an open (for writing) file object for foo.c and the string `datafile-for-foo`. Please note that only string arguments are supported for passing to scripts, and that all arguments provided to the script should be filenames relative to the directory in which the moz.build file is located.

To enable using the same script for generating multiple files with slightly different non-filename parameters, alternative entry points into `script` can be specified:

```
GENERATED_FILES += ['bar.c']
bar = GENERATED_FILES['bar.c']
bar.script = 'generate.py:make_bar'
```

The chosen script entry point may optionally return a set of strings, indicating extra files the output depends on.

GENERATED_WEBIDL_FILES

Generated WebIDL source files.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

These will be generated from some other files.

GYP_DIRS

Defines a list of object directories handled by gyp configurations.

Storage Type `StrictOrderingOnAppendListWithFlagsSpecialization`

Input Type `list`

Elements of this list give the relative object directory. For each element of the list, GYP_DIRS may be accessed as a dictionary (GYP_DIRS[foo]). The object this returns has attributes that need to be set to further specify gyp processing:

- input, gives the path to the root gyp configuration file for that object directory.

- variables, a dictionary containing variables and values to pass to the gyp processor.
- sandbox_vars, a dictionary containing variables and values to pass to the mozbuild processor on top of those derived from gyp configuration.
- non_unified_sources, a list containing sources files, relative to the current moz.build, that should be excluded from source file unification.

Typical use looks like: GYP_DIRS += ['foo', 'bar'] GYP_DIRS['foo'].input = 'foo/foo.gyp' GYP_DIRS['foo'].variables = {

'foo': 'bar', (...)

} (...)

HAS_MISC_RULE

Whether this directory should be traversed in the `misc` tier.

Storage Type `bool`

Input Type `bool`

Many `libs` rules still exist in Makefile.in files. We highly prefer that these rules exist in the `misc` tier/target so that they can be executed concurrently during tier traversal (the `misc` tier is fully concurrent).

Presence of this variable indicates that this directory should be traversed by the `misc` tier.

Please note that converting `libs` rules to the `misc` tier must be done with care, as there are many implicit dependencies that can break the build in subtle ways.

HOST_CFLAGS

Flags passed to the host C compiler for all of the C source files

Storage Type `List`

Input Type `list`

declared in this directory.

Note that the ordering of flags matters here, these flags will be added to the compiler's command line in the same order as they appear in the moz.build file.

HOST_CXXFLAGS

Flags passed to the host C++ compiler for all of the C++ source files

Storage Type `List`

Input Type `list`

declared in this directory.

Note that the ordering of flags matters here; these flags will be added to the compiler's command line in the same order as they appear in the moz.build file.

HOST_DEFINES

Dictionary of compiler defines to declare for host compilation.

Storage Type `InitializedDefines`

Input Type `dict`

See `DEFINES` for specifics.

HOST_LIBRARY_NAME

Name of target library generated when cross compiling.

Storage Type `unicode`

Input Type `unicode`

This variable is only available in templates.

HOST_OS_LIBS

List of system libraries for host programs and libraries.

Storage Type `List`

Input Type `list`

HOST_PROGRAM

Compiled host executable name.

Storage Type `unicode`

Input Type `unicode`

If the configuration token `HOST_BIN_SUFFIX` is set, its value will be automatically appended to `HOST_PROGRAM`. If `HOST_PROGRAM` already ends with `HOST_BIN_SUFFIX`, `HOST_PROGRAM` will remain unchanged. This variable is only available in templates.

HOST_SIMPLE_PROGRAMS

Compile a list of host executable names.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

Each name in this variable corresponds to a hosst executable built from the corresponding source file with the same base name.

If the configuration token `HOST_BIN_SUFFIX` is set, its value will be automatically appended to each name. If a name already ends with `HOST_BIN_SUFFIX`, the name will remain unchanged. This variable is only available in templates.

HOST_SOURCES

Source code files to compile with the host compiler.

Storage Type `_TypedList`

Input Type `list`

This variable contains a list of source code files to compile. with the host compiler.

HOST_USE_LIBS

List of libraries to link to host programs and libraries.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

IPDL_SOURCES

IPDL source files.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

These are `.ipdl` files that will be parsed and converted to `.cpp` files.

IS_COMPONENT

Whether the library contains a binary XPCOM component manifest.

Storage Type `bool`

Input Type `bool`

Implies FORCE_SHARED_LIB. This variable is only available in templates.

IS_FRAMEWORK

Whether the library to build should be built as a framework on OSX.

Storage Type `bool`

Input Type `bool`

This implies the name of the library won't be prefixed nor suffixed. Implies FORCE_SHARED_LIB. This variable is only available in templates.

JAR_MANIFESTS

JAR manifest files that should be processed as part of the build.

Storage Type `_TypedList`

Input Type `list`

JAR manifests are files in the tree that define how to package files into JARs and how chrome registration is performed. For more info, see *JAR Manifests*.

JAVA_JAR_TARGETS

Defines Java JAR targets to be built.

Storage Type `dict`

Input Type `dict`

This variable should not be populated directly. Instead, it should populated by calling add_java_jar().

JETPACK_ADDON_MANIFESTS

List of manifest files defining jetpack addon tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

JETPACK_PACKAGE_MANIFESTS

List of manifest files defining jetpack package tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

LDFLAGS

Flags passed to the linker when linking all of the libraries and

Storage Type `List`

Input Type `list`

executables declared in this directory.

Note that the ordering of flags matters here; these flags will be added to the linker's command line in the same order as they appear in the moz.build file.

LD_VERSION_SCRIPT

The linker version script for shared libraries.

Storage Type `unicode`

Input Type `unicode`

This variable can only be used on Linux.

LIBRARY_DEFINES

Dictionary of compiler defines to declare for the entire library.

Storage Type `OrderedDict`

Input Type `dict`

This variable works like DEFINES, except that declarations apply to all libraries that link into this library via FINAL_LIBRARY.

LIBRARY_NAME

The code name of the library generated for a directory.

Storage Type `unicode`

Input Type `unicode`

By default STATIC_LIBRARY_NAME and SHARED_LIBRARY_NAME take this name. In `example/components/moz.build`,:

```
LIBRARY_NAME = 'xpcomsample'
```

would generate `example/components/libxpcomsample.so` on Linux, or `example/components/xpcomsample.lib` on Windows. This variable is only available in templates.

LOCAL_INCLUDES

Additional directories to be searched for include files by the compiler.

Storage Type `_TypedList`

Input Type `list`

MARIONETTE_LAYOUT_MANIFESTS

List of manifest files defining marionette-layout tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

MARIONETTE_LOOP_MANIFESTS

List of manifest files defining marionette-loop tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

MARIONETTE_UNIT_MANIFESTS

List of manifest files defining marionette-unit tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

MARIONETTE_UPDATE_MANIFESTS

List of manifest files defining marionette-update tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

MARIONETTE_WEBAPI_MANIFESTS

List of manifest files defining marionette-webapi tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

METRO_CHROME_MANIFESTS

List of manifest files defining metro browser chrome tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

MOCHITEST_CHROME_MANIFESTS

List of manifest files defining mochitest chrome tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

MOCHITEST_MANIFESTS

List of manifest files defining mochitest tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

NO_COMPONENTS_MANIFEST

Do not create a binary-component manifest entry for the

Storage Type `bool`

Input Type `bool`

corresponding XPCOMBinaryComponent.

NO_EXPAND_LIBS

Forces to build a real static library, and no corresponding fake

Storage Type `bool`

Input Type `bool`

library.

NO_JS_MANIFEST

Explicitly disclaims responsibility for manifest listing in EXTRA_COMPONENTS.

Storage Type `bool`

Input Type `bool`

Normally, if you have .js files listed in `EXTRA_COMPONENTS` or `EXTRA_PP_COMPONENTS`, you are expected to have a corresponding .manifest file to go with those .js files. Setting `NO_JS_MANIFEST` indicates that the relevant .manifest file and entries for those .js files are elsehwere (jar.mn, for instance) and this state of affairs is OK.

NO_PGO

Whether profile-guided optimization is disable in this directory.

Storage Type `bool`

Input Type `bool`

NO_VISIBILITY_FLAGS

Build sources listed in this file without VISIBILITY_FLAGS.

Storage Type `bool`

Input Type `bool`

OBJDIR_FILES

List of files to be installed anywhere in the objdir. Use sparingly.

Storage Type `_TypedListWithItems`

Input Type `list`

`OBJDIR_FILES` is similar to FINAL_TARGET_FILES, but it allows copying anywhere in the object directory. This is intended for various one-off cases, not for general use. If you wish to add entries to OBJDIR_FILES, please consult a build peer.

OBJDIR_PP_FILES

Like `OBJDIR_FILES`, with preprocessing. Use sparingly.

Storage Type `_TypedListWithItems`

Input Type `list`

OS_LIBS

System link libraries.

Storage Type `List`

Input Type `list`

This variable contains a list of system libaries to link against.

PREPROCESSED_TEST_WEBIDL_FILES

Preprocessed test WebIDL source files.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

These will be preprocessed, then parsed and converted to .cpp and `.h` files if tests are enabled.

PREPROCESSED_WEBIDL_FILES

Preprocessed WebIDL source files.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

These will be preprocessed before being parsed and converted.

PROGRAM

Compiled executable name.

Storage Type `unicode`

Input Type `unicode`

If the configuration token `BIN_SUFFIX` is set, its value will be automatically appended to `PROGRAM`. If `PROGRAM` already ends with `BIN_SUFFIX`, `PROGRAM` will remain unchanged. This variable is only available in templates.

PYTHON_UNIT_TESTS

A list of python unit tests.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

RCFILE

The program .rc file.

Storage Type `unicode`

Input Type `unicode`

This variable can only be used on Windows.

RCINCLUDE

The resource script file to be included in the default .res file.

Storage Type `unicode`

Input Type `unicode`

This variable can only be used on Windows.

REFTEST_MANIFESTS

List of manifest files defining reftests.

Storage Type `_OrderedListWithAction`

Input Type `list`

These are commonly named reftest.list.

RESFILE

The program .res file.

Storage Type `unicode`

Input Type `unicode`

This variable can only be used on Windows.

SDK_FILES

List of files to be installed into the sdk directory.

Storage Type `_TypedListWithItems`

Input Type `list`

`SDK_FILES` will copy (or symlink, if the platform supports it) the contents of its files to the `dist/sdk` directory. Files that are destined for a subdirectory can be specified by accessing a field. For example, to export `foo.py` to the top-level directory and `bar.py` to the directory `subdir`, append to `SDK_FILES` like so:

```
SDK_FILES += ['foo.py']
SDK_FILES.subdir += ['bar.py']
```

SDK_LIBRARY

Whether the library built in the directory is part of the SDK.

Storage Type `bool`

Input Type `bool`

The library will be copied into `SDK_LIB_DIR` (`$DIST/sdk/lib`).

SHARED_LIBRARY_NAME

The name of the static library generated for a directory, if it needs to

Storage Type `unicode`

Input Type `unicode`

differ from the library code name.

Implies FORCE_SHARED_LIB.

SIMPLE_PROGRAMS

Compile a list of executable names.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

Each name in this variable corresponds to an executable built from the corresponding source file with the same base name.

If the configuration token `BIN_SUFFIX` is set, its value will be automatically appended to each name. If a name already ends with `BIN_SUFFIX`, the name will remain unchanged. This variable is only available in templates.

SONAME

The soname of the shared object currently being linked

Storage Type `unicode`

Input Type `unicode`

soname is the "logical name" of a shared object, often used to provide version backwards compatibility. This variable makes sense only for shared objects, and is supported only on some unix platforms.

SOURCES

Source code files.

Storage Type `_TypedListWithItems`

Input Type `list`

This variable contains a list of source code files to compile. Accepts assembler, C, C++, Objective C/C++.

SPHINX_PYTHON_PACKAGE_DIRS

Directories containing Python packages that Sphinx documents.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

SPHINX_TREES

Describes what the Sphinx documentation tree will look like.

Storage Type `dict`

Input Type `dict`

Keys are relative directories inside the final Sphinx documentation tree to install files into. Values are directories (relative to this file) whose content to copy into the Sphinx documentation tree.

STATIC_LIBRARY_NAME

The name of the static library generated for a directory, if it needs to

Storage Type `unicode`

Input Type `unicode`

differ from the library code name.

Implies FORCE_STATIC_LIB.

SYMBOLS_FILE

A file containing a list of symbols to export from a shared library.

Storage Type `SourcePath`

Input Type `unicode`

The given file contains a list of symbols to be exported, and is preprocessed. A special marker "@DATA@" must be added after a symbol name if it points to data instead of code, so that the Windows linker can treat them correctly.

TEST_HARNESS_FILES

List of files to be installed for test harnesses.

Storage Type `_TypedListWithItems`

Input Type `list`

`TEST_HARNESS_FILES` can be used to install files to any directory under $objdir/_tests. Files can be appended to a field to indicate which subdirectory they should be exported to. For example, to export `foo.py` to `_tests/foo`, append to `TEST_HARNESS_FILES` like so:

```
TEST_HARNESS_FILES.foo += ['foo.py']
```

Files from topsrcdir and the objdir can also be installed by prefixing the path(s) with a '/' character and a '!' character, respectively:

```
TEST_HARNESS_FILES.path += ['/build/bar.py', '!quux.py']
```

TEST_WEBIDL_FILES

Test WebIDL source files.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

These will be parsed and converted to `.cpp` and `.h` files if tests are enabled.

UNIFIED_SOURCES

Source code files that can be compiled together.

Storage Type `_TypedList`

Input Type `list`

This variable contains a list of source code files to compile, that can be concatenated all together and built as a single source file. This can help make the build faster and reduce the debug info size.

USE_EXTENSION_MANIFEST

Controls the name of the manifest for JAR files.

Storage Type `bool`

Input Type `bool`

By default, the name of the manifest is ${JAR_MANIFEST}.manifest. Setting this variable to `True` changes the name of the manifest to chrome.manifest.

USE_LIBS

List of libraries to link to programs and libraries.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

USE_STATIC_LIBS

Whether the code in this directory is a built against the static

Storage Type `bool`

Input Type `bool`

runtime library.

This variable only has an effect when building with MSVC.

USE_YASM

Use the yasm assembler to assemble assembly files from SOURCES.

Storage Type `bool`

Input Type `bool`

By default, the build will use the toolchain assembler, $(AS), to assemble source files in assembly language (.s or .asm files). Setting this value to `True` will cause it to use yasm instead.

If yasm is not available on this system, or does not support the current target architecture, an error will be raised.

WEBIDL_EXAMPLE_INTERFACES

Names of example WebIDL interfaces to build as part of the build.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

Names in this list correspond to WebIDL interface names defined in WebIDL files included in the build from one of the *WEBIDL_FILES variables.

WEBIDL_FILES

WebIDL source files.

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

These will be parsed and converted to `.cpp` and `.h` files.

WEBRTC_SIGNALLING_TEST_MANIFESTS

List of manifest files defining WebRTC signalling tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

WEB_PLATFORM_TESTS_MANIFESTS

List of (manifest_path, test_path) defining web-platform-tests.

Storage Type `_TypedListWithAction`

Input Type `list`

WIN32_EXE_LDFLAGS

Flags passed to the linker when linking a Windows .exe executable

Storage Type `List`

Input Type `list`

declared in this directory.

Note that the ordering of flags matter here, these flags will be added to the linker's command line in the same order as they appear in the moz.build file.

This variable only has an effect on Windows.

XPCSHELL_TESTS_MANIFESTS

List of manifest files defining xpcshell tests.

Storage Type `_OrderedListWithAction`

Input Type `list`

XPIDL_MODULE

XPCOM Interface Definition Module Name.

Storage Type `unicode`

Input Type `unicode`

This is the name of the `.xpt` file that is created by linking `XPIDL_SOURCES` together. If unspecified, it defaults to be the same as `MODULE`.

XPIDL_NO_MANIFEST

Indicate that the XPIDL module should not be added to a manifest.

Storage Type `bool`

Input Type `bool`

This flag exists primarily to prevent test-only XPIDL modules from being added to the application's chrome manifest. Most XPIDL modules should not use this flag.

XPIDL_SOURCES

XPCOM Interface Definition Files (xpidl).

Storage Type `StrictOrderingOnAppendList`

Input Type `list`

This is a list of files that define XPCOM interface definitions. Entries must be files that exist. Entries are almost certainly `.idl` files.

XPI_NAME

The name of an extension XPI to generate.

Storage Type `unicode`

Input Type `unicode`

When this variable is present, the results of this directory will end up being packaged into an extension instead of the main dist/bin results.

Functions

add_android_eclipse_library_project

Declare an Android Eclipse library project.

Arguments (str)

This is one of the supported ways to populate the ANDROID_ECLIPSE_PROJECT_TARGETS variable.

The parameters are: * name - project name.

This returns a rich Android Eclipse project type, described at *mozbuild.frontend.data.AndroidEclipseProjectData*.

add_android_eclipse_project

Declare an Android Eclipse project.

Arguments (str, str)

This is one of the supported ways to populate the ANDROID_ECLIPSE_PROJECT_TARGETS variable.

The parameters are: * name - project name. * manifest - path to AndroidManifest.xml.

This returns a rich Android Eclipse project type, described at *mozbuild.frontend.data.AndroidEclipseProjectData*.

add_java_jar

Declare a Java JAR target to be built.

Arguments (str)

This is the supported way to populate the JAVA_JAR_TARGETS variable.

The parameters are: * dest - target name, without the trailing .jar. (required)

This returns a rich Java JAR type, described at *mozbuild.frontend.data.JavaJarData*.

error

Issue a fatal error.

Arguments (str)

If this function is called, processing is aborted immediately.

export

Make the specified variable available to all child directories.

Arguments (str)

The variable specified by the argument string is added to the environment of all directories specified in the DIRS and TEST_DIRS variables. If those directories themselves have child directories, the variable will be exported to all of them.

The value used for the variable is the final value at the end of the moz.build file, so it is possible (but not recommended style) to place the export before the definition of the variable.

This function is limited to the upper-case variables that have special meaning in moz.build files.

NOTE: Please consult with a build peer before adding a new use of this function.

Example usage To make all children directories install as the given extension:

```
XPI_NAME = 'cool-extension'
export('XPI_NAME')
```

include

Include another mozbuild file in the context of this one.

Arguments (SourcePath)

This is similar to a `#include` in C languages. The filename passed to the function will be read and its contents will be evaluated within the context of the calling file.

If a relative path is given, it is evaluated as relative to the file currently being processed. If there is a chain of multiple include(), the relative path computation is from the most recent/active file.

If an absolute path is given, it is evaluated from `TOPSRCDIR`. In other words, `include('/foo')` references the path `TOPSRCDIR + '/foo'`.

Example usage Include `sibling.build` from the current directory.:

```
include('sibling.build')
```

Include `foo.build` from a path within the top source directory:

```
include('/elsewhere/foo.build')
```

template

Decorator for template declarations.

Arguments (function)

Templates are a special kind of functions that can be declared in mozbuild files. Uppercase variables assigned in the function scope are considered to be the result of the template.

Contrary to traditional python functions:

- return values from template functions are ignored,
- template functions don't have access to the global scope.

Example template The following `Program` template sets two variables `PROGRAM` and `USE_LIBS`. `PROGRAM` is set to the argument given on the template invocation, and `USE_LIBS` to contain "mozglue":

```
@template
def Program(name):
    PROGRAM = name
    USE_LIBS += ['mozglue']
```

Template invocation A template is invoked in the form of a function call:

```
Program('myprog')
```

The result of the template, being all the uppercase variable it sets is mixed to the existing set of variables defined in the mozbuild file invoking the template:

```
FINAL_TARGET = 'dist/other'
USE_LIBS += ['mylib']
Program('myprog')
USE_LIBS += ['otherlib']
```

The above mozbuild results in the following variables set:

- `FINAL_TARGET` is 'dist/other'
- `USE_LIBS` is ['mylib', 'mozglue', 'otherlib']
- `PROGRAM` is 'myprog'

warning

Issue a warning.

Arguments (str)

Warnings are string messages that are printed during execution.

Warnings are ignored during execution.

Special Variables

CONFIG

Dictionary containing the current configuration variables.

Type `dict`

All the variables defined by the configuration system are available through this object. e.g. `ENABLE_TESTS`, `CFLAGS`, etc.

Values in this container are read-only. Attempts at changing values will result in a run-time error.

Access to an unknown variable will return None.

EXTRA_COMPONENTS

Additional component files to distribute.

Type `list`

This variable contains a list of files to copy into `$(FINAL_TARGET)/components/`.

EXTRA_JS_MODULES

Additional JavaScript files to distribute.

Type `list`

This variable contains a list of files to copy into ‘‘$(FINAL_TARGET)/modules.

EXTRA_PP_COMPONENTS

Javascript XPCOM files.

Type `list`

This variable contains a list of files to preprocess. Generated files will be installed in the `/components` directory of the distribution.

EXTRA_PP_JS_MODULES

Additional JavaScript files to distribute.

Type `list`

This variable contains a list of files to copy into `$(FINAL_TARGET)/modules`, after preprocessing.

JS_PREFERENCE_FILES

Exported javascript files.

Type `list`

A list of files copied into the dist directory for packaging and installation. Path will be defined for gre or application prefs dir based on what is building.

JS_PREFERENCE_PP_FILES

Like JS_PREFERENCE_FILES, preprocessed..

Type `list`

OBJDIR

The path to the object directory for this file.

Type `str`

Is is the same as `TOPOBJDIR + RELATIVEDIR`.

RELATIVEDIR

Constant defining the relative path of this file.

Type `str`

The relative path is from `TOPSRCDIR`. This is defined as relative to the main file being executed, regardless of whether additional files have been included using `include()`.

RESOURCE_FILES

List of resources to be exported, and in which subdirectories.

Type `list`

`RESOURCE_FILES` is used to list the resource files to be exported to `dist/bin/res`, but it can be used for other files as well. This variable behaves as a list when appending filenames for resources in the top-level directory. Files can also be appended to a field to indicate which subdirectory they should be exported to. For example, to export `foo.res` to the top-level directory, and `bar.res` to `fonts/`, append to `RESOURCE_FILES` like so:

```
RESOURCE_FILES += ['foo.res']
RESOURCE_FILES.fonts += ['bar.res']
```

SRCDIR

Constant defining the source directory of this file.

Type `str`

This is the path inside `TOPSRCDIR` where this file is located. It is the same as `TOPSRCDIR + RELATIVEDIR`.

TESTING_JS_MODULES

JavaScript modules to install in the test-only destination.

Type `list`

Some JavaScript modules (JSMs) are test-only and not distributed with Firefox. This variable defines them.

To install modules in a subdirectory, use properties of this variable to control the final destination. e.g.

`TESTING_JS_MODULES.foo += ['module.jsm'].`

TEST_DIRS

Like DIRS but only for directories that contain test-only code.

Type `list`

If tests are not enabled, this variable will be ignored.

This variable may go away once the transition away from Makefiles is complete.

TOPOBJDIR

Constant defining the top object directory.

Type `str`

The top object directory is the parent directory which will contain the output of the build. This is commonly referred to as "the object directory."

TOPSRCDIR

Constant defining the top source directory.

Type `str`

The top source directory is the parent directory containing the source code and all build files. It is typically the root directory of a cloned repository.

4.1.7 Files Metadata

moz.build Files provide a mechanism for attaching metadata to files. Essentially, you define some flags to set on a file or file pattern. Later, some tool or process queries for metadata attached to a file of interest and it does something intelligent with that data.

Defining Metadata

Files metadata is defined by using the *Files Sub-Context* in `moz.build` files. e.g.:

```
with Files('**/Makefile.in'):
    BUG_COMPONENT = ('Core', 'Build Config')
```

This working example says, *for all Makefile.in files in every directory underneath this one - including this directory - set the Bugzilla component to Core :: Build Config*.

For more info, read the *docs on Files*.

How Metadata is Read

`Files` metadata is extracted in *Filesystem Reading Mode*.

Reading starts by specifying a set of files whose metadata you are interested in. For each file, the filesystem is walked to the root of the source directory. Any `moz.build` encountered during this walking are marked as relevant to the file.

Let's say you have the following filesystem content:

```
/moz.build
/root_file
/dir1/moz.build
/dir1/foo
/dir1/subdir1/foo
/dir2/foo
```

For `/root_file`, the relevant `moz.build` files are just `/moz.build`.

For `/dir1/foo` and `/dir1/subdir1/foo`, the relevant files are `/moz.build` and `/dir1/moz.build`.

For `/dir2`, the relevant file is just `/moz.build`.

Once the list of relevant `moz.build` files is obtained, each `moz.build` file is evaluated. Root `moz.build` file first, leaf-most files last. This follows the rules of *Filesystem Reading Mode*, with the set of evaluated `moz.build` files being controlled by filesystem content, not `DIRS` variables.

The file whose metadata is being resolved maps to a set of `moz.build` files which in turn evaluates to a list of contexts. For file metadata, we only care about one of these contexts: *Files*.

We start with an empty `Files` instance to represent the file. As we encounter a *files sub-context*, we see if it is appropriate to this file. If it is, we apply its values. This process is repeated until all *files sub-contexts* have been applied or skipped. The final state of the `Files` instance is used to represent the metadata for this particular file.

It may help to visualize this. Say we have 2 `moz.build` files:

```
# /moz.build
with Files('*.cpp'):
    BUG_COMPONENT = ('Core', 'XPCOM')

with Files('**/*.js'):
    BUG_COMPONENT = ('Firefox', 'General')

# /foo/moz.build
with Files('*.js'):
    BUG_COMPONENT = ('Another', 'Component')
```

Querying for metadata for the file `/foo/test.js` will reveal 3 relevant `Files` sub-contexts. They are evaluated as follows:

1. `/moz.build - Files('*.cpp')`. Does `/*.cpp` match `/foo/test.js`? **No**. Ignore this context.
2. `/moz.build - Files('**/*.js')`. Does `/**/*.js` match `/foo/test.js`? **Yes**. Apply `BUG_COMPONENT = ('Firefox', 'General')` to us.
3. `/foo/moz.build - Files('*.js')`. Does `/foo/*.js` match `/foo/test.js`? **Yes**. Apply `BUG_COMPONENT = ('Another', 'Component')`.

At the end of execution, we have `BUG_COMPONENT = ('Another', 'Component')` as the metadata for `/foo/test.js`.

One way to look at file metadata is as a stack of data structures. Each `Files` sub-context relevant to a given file is applied on top of the previous state, starting from an empty state. The final state wins.

Finalizing Values

The default behavior of `Files` sub-context evaluation is to apply new values on top of old. In most circumstances, this results in desired behavior. However, there are circumstances where this may not be desired. There is thus a mechanism to *finalize* or *freeze* values.

Finalizing values is useful for scenarios where you want to prevent wildcard matches from overwriting previously-set values. This is useful for one-off files.

Let's take `Makefile.in` files as an example. The build system module policy dictates that `Makefile.in` files are part of the `Build Config` module and should be reviewed by peers of that module. However, there exist `Makefile.in` files in many directories in the source tree. Without finalization, a `*` or `**` wildcard matching rule would match `Makefile.in` files and overwrite their metadata.

Finalizing of values is performed by setting the `FINAL` variable on `Files` sub-contexts. See the *Files documentation* for more.

Here is an example with `Makefile.in` files, showing how it is possible to finalize the `BUG_COMPONENT` value.:

```
# /moz.build
with Files('**/Makefile.in'):
    BUG_COMPONENT = ('Core', 'Build Config')
    FINAL = True

# /foo/moz.build
```

```
with Files('**'):
    BUG_COMPONENT = ('Another', 'Component')
```

If we query for metadata of `/foo/Makefile.in`, both `Files` sub-contexts match the file pattern. However, since `BUG_COMPONENT` is marked as finalized by `/moz.build`, the assignment from `/foo/moz.build` is ignored. The final value for `BUG_COMPONENT` is `('Core', 'Build Config')`.

Here is another example:

```
with Files('*.cpp'):
    BUG_COMPONENT = ('One-Off', 'For C++')
    FINAL = True

with Files('**'):
    BUG_COMPONENT = ('Regular', 'Component')
```

For every files except `foo.cpp`, the bug component will be resolved as `Regular :: Component`. However, `foo.cpp` has its value of `One-Off :: For C++` preserved because it is finalized.

Important: `FINAL` only applied to variables defined in a context.

If you want to mark one variable as finalized but want to leave another mutable, you'll need to use 2 `Files` contexts.

Guidelines for Defining Metadata

In general, values defined towards the root of the source tree are generic and become more specific towards the leaves. For example, the `BUG_COMPONENT` for `/browser` might be `Firefox :: General` whereas `/browser/components/preferences` would list `Firefox :: Preferences`.

4.1.8 Profile Guided Optimization

PGO (Profile Guided Optimization) is the process of adding probes to a compiled binary, running said binary, then using the run-time information to *recompile* the binary to (hopefully) make it faster.

How PGO Builds Work

The supported interface for invoking a PGO build is to evaluate the *build* target of client.mk with *MOZ_PGO* defined. e.g.:

```
$ make -f client.mk MOZ_PGO=1
```

This is equivalent to:

```
$ make -f client.mk profiledbuild
```

Which is roughly equivalent to:

1. Perform a build with *MOZ_PROFILE_GENERATE=1* and *MOZ_PGO_INSTRUMENTED=1*
2. Package with *MOZ_PGO_INSTRUMENTED=1*
3. Performing a run of the instrumented binaries
4. $ make maybe_clobber_profiledbuild
5. Perform a build with *MOZ_PROFILE_USE=1*

Differences between toolchains

There are some implementation differences depending on the compiler toolchain being used.

The *maybe_clobber_profiledbuild* step gets its name because of a difference. On Windows, this step merely moves some *.pgc* files around. Using GCC or Clang, it is equivalent to a *make clean*.

4.1.9 Why the Build System is Slow

A common complaint about the build system is that it's slow. There are many reasons contributing to its slowness. We will attempt to document them here.

First, it is important to distinguish between a *clobber build* and an *incremental build*. The reasons for why each are slow can be different.

The build does a lot of work

It may not be obvious, but the main reason the build system is slow is because it does a lot of work! The source tree consists of a few thousand C++ files. On a modern machine, we spend over 120 minutes of CPU core time compiling files! So, if you are looking for the root cause of slow clobber builds, look at the sheer volume of C++ files in the tree.

You don't have enough CPU cores and MHz

The build should be CPU bound. If the build system maintainers are optimizing the build system perfectly, every CPU core in your machine should be 100% saturated during a build. While this isn't currently the case (keep reading below), generally speaking, the more CPU cores you have in your machine and the more total MHz in your machine, the better.

We highly recommend building with no fewer than 4 physical CPU cores. Please note the *physical* in this sentence. Hyperthreaded cores (an Intel Core i7 will report 8 CPU cores but only 4 are physical for example) only yield at most a 1.25x speedup per core.

We also recommend using the most modern CPU model possible. Haswell chips deliver much more performance per CPU cycle than say Sandy Bridge CPUs.

This cause impacts both clobber and incremental builds.

You are building with a slow I/O layer

The build system can be I/O bound if your I/O layer is slow. Linking libxul on some platforms and build architectures can perform gigabytes of I/O.

To minimize the impact of slow I/O on build performance, **we highly recommend building with an SSD.** Power users with enough memory may opt to build from a RAM disk. Mechanical disks should be avoided if at all possible.

Some may dispute the importance of an SSD on build times. It is true that the beneficial impact of an SSD can be mitigated if your system has lots of memory and the build files stay in the page cache. However, operating system memory management is complicated. You don't really have control over what or when something is evicted from the page cache. Therefore, unless your machine is a dedicated build machine or you have more memory than is needed by everything running on your machine, chances are you'll run into page cache eviction and you I/O layer will impact build performance. That being said, an SSD certainly doesn't hurt build times. And, anyone who has used a machine with an SSD will tell you how great of an investment it is for performance all around the operating system. On top of that, some automated tests are I/O bound (like those touching SQLite databases), so an SSD will make tests faster.

This cause impacts both clobber and incremental builds.

You don't have enough memory

The build system allocates a lot of memory, especially when building many things in parallel. If you don't have enough free system memory, the build will cause swap activity, slowing down your system and the build. Even if you never get to the point of swapping, the build system performs a lot of I/O and having all accessed files in memory and the page cache can significantly reduce the influence of the I/O layer on the build system.

We recommend building with no less than 8 GB of system memory. As always, the more memory you have, the better. For a bare bones machine doing nothing more than building the source tree, anything more than 16 GB is likely entering the point of diminishing returns.

This cause impacts both clobber and incremental builds.

You are building on Windows

New processes on Windows are about a magnitude slower to spawn than on UNIX-y systems such as Linux. This is because Windows has optimized new threads while the *NIX platforms typically optimize new processes. Anyway, the build system spawns thousands of new processes during a build. Parts of the build that rely on rapid spawning of new processes are slow on Windows as a result. This is most pronounced when running *configure*. The configure file is a giant shell script and shell scripts rely heavily on new processes. This is why configure on Windows can run over a minute slower on Windows.

Another reason Windows builds are slower is because Windows lacks proper symlink support. On systems that support symlinks, we can generate a file into a staging area then symlink it into the final directory very quickly. On Windows, we have to perform a full file copy. This incurs much more I/O. And if done poorly, can muck with file modification times, messing up build dependencies. As of the summer of 2013, the impact of symlinks is being mitigated through the use of an *install manifest*.

These issues impact both clobber and incremental builds.

Recursive make traversal is slow

The build system has traditionally been built by employing recursive make. Recursive make involves make iterating through directories / make files sequentially and executing each in turn. This is inefficient for directories containing few targets/tasks because make could be *starved* for work when processing these directories. Any time make is starved, the build isn't using all available CPU cycles and the build is slower as a result.

Work has started in bug 907365 to fix this issue by changing the way make traverses all the make files.

The impact of slow recursive make traversal is mostly felt on incremental builds. Traditionally, most of the wall time during a no-op build is spent in make traversal.

make is inefficient

Compared to modern build backends like Tup or Ninja, make is slow and inefficient. We can only make make so fast. At some point, we'll hit a performance plateau and will need to use a different tool to make builds faster.

Please note that clobber and incremental builds are different. A clobber build with make will likely be as fast as a clobber build with e.g. Tup. However, Tup should vastly outperform make when it comes to incremental builds. Therefore, this issue is mostly seen when performing incremental builds.

C++ header dependency hell

Modifying a *.h* file can have significant impact on the build system. If you modify a *.h* that is used by 1000 C++ files, all of those 1000 C++ files will be recompiled.

Our code base has traditionally been sloppy managing the impact of changed headers on build performance. Bug 785103 tracks improving the situation.

This issue mostly impacts the times of an *incremental build*.

A search/indexing service on your machine is running

Many operating systems have a background service that automatically indexes filesystem content to make searching faster. On Windows, you have the Windows Search Service. On OS X, you have Finder.

These background services sometimes take a keen interest in the files being produced as part of the build. Since the build system produces hundreds of megabytes or even a few gigabytes of file data, you can imagine how much work this is to index! If this work is being performed while the build is running, your build will be slower.

OS X's Finder is notorious for indexing when the build is running. And, it has a tendency to suck up a whole CPU core. This can make builds several minutes slower. If you build with `mach` and have the optional `psutil` package built (it requires Python development headers - see *Python and the Build System* for more) and Finder is running during a build, mach will print a warning at the end of the build, complete with instructions on how to fix it.

4.1.10 Environment Variables Impacting the Build System

Various environment variables have an impact on the behavior of the build system. This document attempts to document them.

AUTOCLOBBER If defines, the build system will automatically clobber as needed. The default behavior is to print a message and error out when a clobber is needed.

This variable is typically defined in a *mozconfig* file via `mk_add_options`.

REBUILD_CHECK If defined, the build system will print information about why certain files were rebuilt.

This feature is disabled by default because it makes the build slower.

MACH_NO_TERMINAL_FOOTER If defined, the terminal footer displayed when building with mach in a TTY is disabled.

MACH_NO_WRITE_TIMES If defined, mach commands will not prefix output lines with the elapsed time since program start. This option is equivalent to passing `--log-no-times` to mach.

4.1.11 Build Targets

When you build with `mach build`, there are some special targets that can be built. This page attempts to document them.

Partial Tree Targets

The targets in this section only build part of the tree. Please note that partial tree builds can be unreliable. Use at your own risk.

export Build the *export* tier. The *export* tier builds everything that is required for C/C++ compilation. It stages all header files, processes IDLs, etc.

compile Build the *compile* tier. The *compile* tier compiles all C/C++ files.

libs Build the *libs* tier. The *libs* tier performs linking and performs most build steps which aren't related to compilation.

tools Build the *tools* tier. The *tools* tier mostly deals with supplementary tools and compiled tests. It will link tools against libXUL, including compiled test binaries.

binaries: Recompiles and relinks C/C++ files. Only works after a complete normal build, but allows for much faster rebuilds of C/C++ code. For performance reasons, however, it skips nss, nspr, icu and ffi. This is targeted to improve local developer workflow when touching C/C++ code.

install-manifests Process install manifests. Install manifests handle the installation of files into the object directory.

Unless `NO_REMOVE=1` is defined in the environment, files not accounted in the install manifests will be deleted from the object directory.

install-tests Processes the tests install manifest.

Common Actions

The targets in this section correspond to common build-related actions. Many of the actions in this section are effectively frontends to shell scripts. These actions will likely all be replaced by mach commands someday.

buildsymbols Create a symbols archive for the current build.

This must be performed after a successful build.

check Run build system tests.

4.1.12 Python and the Build System

The Python programming language is used significantly in the build system. If we need to write code for the build system or for a tool related to the build system, Python is typically the first choice.

Python Requirements

The tree requires Python 2.7.3 or greater but not Python 3 to build. All Python packages not in the Python distribution are included in the source tree. So all you should need is a vanilla Python install and you should be good to go.

Only CPython (the Python distribution available from www.python.org) is supported.

We require Python 2.7.3 (and not say 2.7.2) to build because Python 2.7.3 contains numerous bug fixes, especially around the area of Unicode handling. These bug fixes are extremely annoying and have to be worked around. The build maintainers were tired of doing this, so the minimum version requirement was upped (bug 870420).

We intend to eventually support Python 3. This will come by way of dual 2.7/3.x compatibility because a single flag day conversion to 3.x will be too cumbersome given the amount of Python that would need converted. We will not know which 3.x minor release we are targeting until this effort is underway. This is tracked in bug 636155.

Compiled Python Packages

There are some features of the build that rely on compiled Python packages (packages containing C source). These features are currently all optional because not every system contains the Python development headers required to build these extensions.

We recommend you have the Python development headers installed (`mach bootstrap` should do this for you) so you can take advantage of these features.

Issues with OS X System Python

The Python that ships with OS X has historically been littered with subtle bugs and suboptimalities. Furthermore, OS X up through 10.8 don't ship with Python 2.7.3 (10.8 ships with 2.7.2).

OS X 10.8 and below users will be required to install a new Python distribution. This may not be necessary for OS X 10.9+. However, we still recommend installing a separate Python because of the history with OS X's system Python issues.

We recommend installing Python through Homebrew or MacPorts. If you run `mach bootstrap`, this should be done for you.

Virtualenvs

The build system relies heavily on virtualenvs. Virtualenvs are standalone and isolated Python environments. The problem a virtualenv solves is that of dependencies across multiple Python components. If two components on a system relied on different versions of a package, there could be a conflict. Instead of managing multiple versions of a package simultaneously, Python and virtualenvs take the route that it is easier to just keep them separate so there is no potential for conflicts.

Very early in the build process, a virtualenv is created inside the *object directory*. The virtualenv is configured such that it can find all the Python packages in the source tree. The code for this lives in *`mozbuild.virtualenv`*.

Deficiencies

There are numerous deficiencies with the way virtualenvs are handled in the build system.

- mach reinvents the virtualenv.

  There is code in `build/mach_bootstrap.py` that configures `sys.path` much the same way the virtualenv does. There are various bugs tracking this. However, no clear solution has yet been devised. It's not a huge problem and thus not a huge priority.

- They aren't preserved across copies and packaging.

  If you attempt to copy an entire tree from one machine to another or from one directory to another, chances are the virtualenv will fall apart. It would be nice if we could preserve it somehow. Instead of actually solving portable virtualenvs, all we really need to solve is encapsulating the logic for populating the virtualenv along with all dependent files in the appropriate place.

- .pyc files written to source directory.

  We rely heavily on `.pth` files in our virtualenv. A `.pth` file is a special file that contains a list of paths. Python will take the set of listed paths encountered in `.pth` files and add them to `sys.path`.

  When Python compiles a `.py` file to bytecode, it writes out a `.pyc` file so it doesn't have to perform this compilation again. It puts these `.pyc` files alongside the `.pyc` file. Python provides very little control for determing where these `.pyc` files go, even in Python 3 (which offers customer importers).

  With `.pth` files pointing back to directories in the source tree and not the object directory, `.pyc` files are created in the source tree. This is bad because when Python imports a module, it first looks for a `.pyc` file before the `.py` file. If there is a `.pyc` file but no `.py` file, it will happily import the module. This wreaks havoc during file moves, refactoring, etc.

  There are various proposals for fixing this. See bug 795995.

Installing Python Manually

We highly recommend you use your system's package manager or a well-supported 3rd party package manager to install Python for you. If these are not available to you, we recommend the following tools for installing Python:

- buildout.python
- pyenv
- An official installer from http://www.python.org.

If all else fails, consider compiling Python from source manually. But this should be viewed as the least desirable option.

Common Issues with Python

Upgrading your Python distribution breaks the virtualenv

If you upgrade the Python distribution (e.g. install Python 2.7.5 from 2.7.3, chances are parts of the virtualenv will break. This commonly manifests as a cryptic `Cannot import XXX` exception. More often than not, the module being imported contains binary/compiled components.

If you upgrade or reinstall your Python distribution, we recommend clobbering your build.

Packages installed at the system level conflict with build system's

It is common for people to install Python packages using `sudo` (e.g. `sudo pip install psutil`) or with the system's package manager (e.g. `apt-get install python-mysql`.

A problem with this is that packages installed at the system level may conflict with the package provided by the source tree. As of bug 907902 and changeset f18eae7c3b27 (September 16, 2013), this should no longer be an issue since the virtualenv created as part of the build doesn't add the system's `site-packages` directory to `sys.path`. However, poorly installed packages may still find a way to creep into the mix and interfere with our virtualenv.

As a general principle, we recommend against using your system's package manager or using `sudo` to install Python packages. Instead, create virtualenvs and isolated Python environments for all of your Python projects.

Python on $PATH is not appropriate

Tools like `mach` will look for Python by performing `/usr/bin/env python` or equivalent. Please be sure the appropriate Python 2.7.3+ path is on $PATH. On OS X, this likely means you'll need to modify your shell's init script to put something ahead of `/usr/bin`.

4.1.13 Test Manifests

Many test suites have their test metadata defined in files called **test manifests**.

Test manifests are divided into two flavors: *ManifestParser Manifests* and *Reftest Manifests*.

Naming Convention

The build system does not enforce file naming for test manifest files. However, the following convention is used.

mochitest.ini For the *plain* flavor of mochitests.

chrome.ini For the *chrome* flavor of mochitests.

browser.ini For the *browser chrome* flavor of mochitests.

a11y.ini For the *a11y* flavor of mochitests.

xpcshell.ini For *xpcshell* tests.

ManifestParser Manifests

ManifestParser manifests are essentially ini files that conform to a basic set of assumptions.

The reference documentation for manifestparser manifests describes the basic format of test manifests.

In summary, manifests are ini files with section names describing test files:

```
[test_foo.js]
[test_bar.js]
```

Keys under sections can hold metadata about each test:

```
[test_foo.js]
skip-if = os == "win"
[test_foo.js]
skip-if = os == "linux" && debug
[test_baz.js]
fail-if = os == "mac" || os == "android"
```

There is a special **DEFAULT** section whose keys/metadata apply to all sections/tests:

```
[DEFAULT]
property = value

[test_foo.js]
```

In the above example, **test_foo.js** inherits the metadata **property = value** from the **DEFAULT** section.

Recognized Metadata

Test manifests can define some common keys/metadata to influence behavior. Those keys are as follows:

head List of files that will be executed before the test file. (Used in xpcshell tests.)

tail List of files that will be executed after the test file. (Used in xpcshell tests.)

support-files List of additional files required to run tests. This is typically defined in the **DEFAULT** section.

> Unlike other file lists, *support-files* supports a globbing mechanism to facilitate pulling in many files with minimal typing. This globbing mechanism is activated if an entry in this value contains a `*` character. A single `*` will wildcard match all files in a directory. A double `**` will descend into child directories. For example, `data/*` will match `data/foo` but not `data/subdir/bar` where `data/**` will match `data/foo` and `data/subdir/bar`.
>
> Support files starting with `/` are placed in a root directory, rather than a location determined by the manifest location. For mochitests, this allows for the placement of files at the server root. The source file is selected from the base name (e.g., `foo` for `/path/foo`). Files starting with `/` cannot be selected using globbing.
>
> Some support files are used by tests across multiple directories. In this case, a test depending on a support file from another directory must note that dependency with the path to the required support file in

its own **support-files** entry. These use a syntax where paths starting with `!/` will indicate the beginning of the path to a shared support file starting from the root of the srcdir. For example, if a manifest at `dom/base/test/mochitest.ini` has a support file, `dom/base/test/server-script.sjs`, and a mochitest in `dom/workers/test` depends on that support file, the test manifest at `dom/workers/test/mochitest.ini` must include `!/dom/base/test/server-script.sjs` in its **support-files** entry.

generated-files List of files that are generated as part of the build and don't exist in the source tree.

The build system assumes that each manifest file, test file, and file listed in **head**, **tail**, and **support-files** is static and provided by the source tree (and not automatically generated as part of the build). This variable tells the build system not to make this assumption.

This variable will likely go away sometime once all generated files are accounted for in the build config.

If a generated file is not listed in this key, a clobber build will likely fail.

dupe-manifest Record that this manifest duplicates another manifest.

The common scenario is two manifest files will include a shared manifest file via the `[include:file]` special section. The build system enforces that each test file is only provided by a single manifest. Having this key present bypasses that check.

The value of this key is ignored.

skip-if Skip this test if the specified condition is true. See *Manifest Filter Language*.

fail-if Expect test failure if the specified condition is true. See *Manifest Filter Language*.

run-sequentially If present, the test should not be run in parallel with other tests.

Some test harnesses support parallel test execution on separate processes and/or threads (behavior varies by test harness). If this key is present, the test harness should not attempt to run this test in parallel with any other test.

By convention, the value of this key is a string describing why the test can't be run in parallel.

Manifest Filter Language

Some manifest keys accept a special filter syntax as their values. These values are essentially boolean expressions that are evaluated at test execution time.

The expressions can reference a well-defined set of variables, such as `os` and `debug`. These variables are populated from the `mozinfo.json` file. For the full list of available variables, see the *mozinfo documentation*.

See the source for the full documentation of the expression syntax until it is documented here.

File Installation

Files referenced by manifests are automatically installed into the object directory into paths defined in `mozbuild.frontend.emitter.TreeMetadataEmitter._process_test_manifest()`.

Relative paths resolving to parent directory (e.g. `support-files = ../foo.txt` have special behavior.

For `support-files`, the file will be installed to the default destination for that manifest. Only the file's base name is used to construct the final path: directories are irrelevant. Files starting with `/` are an exception, these are installed relative to the root of the destination; the base name is instead used to select the file..

For all other entry types, the file installation is skipped.

Reftest Manifests

See MDN.

4.1.14 mozinfo

`mozinfo` is a solution for representing a subset of build configuration and run-time data.

`mozinfo` data is typically accessed through a `mozinfo.json` file which is written to the *object directory* during build configuration. The code for writing this file lives in *`mozbuild.mozinfo`*.

`mozinfo.json` is an object/dictionary of simple string values.

The attributes in `mozinfo.json` are used for many purposes. One use is to filter tests for applicability to the current build. For more on this, see *Test Manifests*.

mozinfo.json Attributes

`mozinfo` currently records the following attributes.

appname The application being built.

Value comes from `MOZ_APP_NAME` from `config.status`.

Optional.

asan Whether address sanitization is enabled.

Values are `true` and `false`.

Always defined.

bin_suffix The file suffix for binaries produced with this build.

Values may be an empty string, as not all platforms have a binary suffix.

Always defined.

bits The number of bits in the CPU this build targets.

Values are typically `32` or `64`.

Universal Mac builds do not have this key defined.

Unkown processor architectures (see `processor` below) may not have this key defined.

Optional.

buildapp The path to the XUL application being built.

For desktop Firefox, this is `browser`. For Fennec, it's `mobile/android`. For B2G, it's `b2g`.

crashreporter Whether the crash reporter is enabled for this build.

Values are `true` and `false`.

Always defined.

datareporting Whether data reporting (MOZ_DATA_REPORTING) is enabled for this build.

Values are `true` and `false`.

Always defined.

debug Whether this is a debug build.

Values are `true` and `false`.

Always defined.

healthreport Whether the Health Report feature is enabled.

Values are `true` and `false`.

Always defined.

mozconfig The path of the *mozconfig file* used to produce this build.

Optional.

nightly_build Whether this is a nightly build.

Values are `true` and `false`.

Always defined.

os The operating system the build is produced for. Values for tier-1 supported platforms are `linux`, `win`, `mac`, `b2g`, and `android`. For other platforms, the value is the lowercase version of the `OS_TARGET` variable from `config.status`.

Always defined.

processor Information about the processor architecture this build targets.

Values come from `TARGET_CPU`, however some massaging may be performed.

If the build is a universal build on Mac (it targets both 32-bit and 64-bit), the value is `universal-x86-x86_64`.

If the value starts with `arm`, the value is `arm`.

If the value starts with a string of the form `i[3-9]86]`, the value is `x86`.

Always defined.

release_build Whether this is a release build.

Values are `true` and `false`.

Always defined.

sm_promise Whether spidermonkey promises have been enabled or not. This is set by adding –enable-sm-promise to the mozconfig file.

Values are `true` and `false`.

Always defined.

tests_enabled Whether tests are enabled for this build.

Values are `true` and `false`.

Always defined.

toolkit The widget toolkit in case. The value comes from the `MOZ_WIDGET_TOOLKIT` `config.status` variable.

Always defined.

topsrcdir The path to the source directory the build came from.

Always defined.

wave Whether Wave audio support is enabled.

Values are `true` and `false`.

Always defined.

webm Whether WebM support is enabled.

Values are `true` and `false`.

Always defined.

4.1.15 Text Preprocessor

The build system contains a text preprocessor similar to the C preprocessor, meant for processing files which have no built-in preprocessor such as XUL and JavaScript documents. It is implemented at `python/mozbuild/mozbuild/preprocessor.py` and is typically invoked via *JAR Manifests*.

While used to preprocess CSS files, the directives are changed to begin with `%` instead of `#` to avoid conflict of the id selectors.

Directives

Variable Definition

define

```
#define variable
#define variable value
```

Defines a preprocessor variable.

Note that, unlike the C preprocessor, instances of this variable later in the source are not automatically replaced (see #filter). If value is not supplied, it defaults to `1`.

Note that whitespace is significant, so `"#define foo one"` and `"#define foo one "` is different (in the second case, `foo` is defined to be a four-character string).

undef

```
#undef variable
```

Undefines a preprocessor variable.

Conditionals

if

```
#if variable
#if !variable
#if variable==string
#if variable!=string
```

Disables output if the conditional is false. This can be nested to arbitrary depths. Note that in the equality checks, the variable must come first, and the comparison operator must not be surrounded by any whitespace.

else

```
#else
```

Reverses the state of the previous conditional block; for example, if the last `#if` was true (output was enabled), an `#else` makes it off (output gets disabled).

> **Warning:** An `#else` is relative to the last conditional block only, unlike the C preprocessor.
> It does not matter whether any blocks before it were true. This behavior changed on trunk (Gecko 1.9) on 2006-12-07; see Bug 277122 for details.

```
#if 1
  always included
#elif 1
  never included
#else
  always included
#endif
```

endif

```
#endif
```

Ends the conditional block.

ifdef / ifndef

```
#ifdef variable
#ifndef variable
```

An `#if` conditional that is true only if the preprocessor variable variable is defined (in the case of `ifdef`) or not defined (`ifndef`).

elif / elifdef / elifndef

```
#elif variable
#elif !variable
#elif variable == string
#elif variable != string
#elifdef variable
#elifndef variable
```

A shorthand to mean an `#else` combined with the relevant conditional. The following two blocks are equivalent:

```
#ifdef foo
  block 1
#elifdef bar
  block 2
#endif
```

```
#ifdef foo
  block 1
#else
#ifdef bar
  block 2
#endif
#endif
```

Warning: An `#elif`, `#elifdef`, or `#elifndef` is relative to the last conditional block only (as well as the condition it implies), unlike the C preprocessor. It does not matter whether any blocks before it were true. This behavior changed on trunk (Gecko 1.9) on 2006-12-07. See Bug 277122 for details.

File Inclusion

include

```
#include filename
```

The file specified by filename is processed as if the contents was placed at this position. This also means that preprocessor conditionals can even be started in one file and ended in another (but is highly discouraged). There is no limit on depth of inclusion, or repeated inclusion of the same file, or self inclusion; thus, care should be taken to avoid infinite loops.

includesubst

```
#includesubst @variable@filename
```

Same as a `#include` except that all instances of variable in the included file is also expanded as in `#filter` substitution

expand

```
#expand string
```

All variables wrapped in __ are replaced with their value, for this line only. If the variable is not defined, it expands to an empty string. For example, if `foo` has the value `bar`, and `baz` is not defined, then:

```
#expand This <__foo__> <__baz__> gets expanded
```

Is expanded to:

```
This <bar> <> gets expanded
```

filter / unfilter

```
#filter filter1 filter2 ... filterN
#unfilter filter1 filter2 ... filterN
```

`#filter` turns on the given filter.

Filters are run in alphabetical order on a per-line basis.

`#unfilter` turns off the given filter. Available filters are:

emptyLines strips blank lines from the output

slashslash strips everything from the first two consecutive slash (`/`) characters until the end of the line

spaces collapses consecutive sequences of spaces into a single space, and strips leading and trailing spaces

substitution all variables wrapped in @ are replaced with their value. If the variable is not defined, it is a fatal error. Similar to `#expand` and `#filter`

attemptSubstitution all variables wrapped in `@` are replaced with their value, or an empty string if the variable is not defined. Similar to `#expand`.

literal

```
#literal string
```

Output the string (i.e. the rest of the line) literally, with no other fixups. This is useful to output lines starting with #, or to temporarily disable filters.

Other

#error

```
#error string
```

Cause a fatal error at this point, with the error message being the given string.

4.1.16 JAR Manifests

JAR Manifests are plaintext files in the tree that are used to package chrome files into the correct JARs, and create Chrome Registration manifests. JAR Manifests are commonly named `jar.mn`. They are declared in `moz.build` files using the `JAR_MANIFESTS` variable.

`jar.mn` files are automatically processed by the build system when building a source directory that contains one. The `jar`.mn is run through the *Text Preprocessor* before being passed to the manifest processor. In order to have `@variables@` expanded (such as `@AB_CD@`) throughout the file, add the line `#filter substitution` at the top of your `jar.mn` file.

The format of a jar.mn is fairly simple; it consists of a heading specifying which JAR file is being packaged, followed by indented lines listing files and chrome registration instructions.

To see a simple `jar.mn` file at work, see `toolkit/profile/jar.mn`. A much more complex `jar.mn` is at `toolkit/locales/jar.mn`.

Shipping Chrome Files

To ship chrome files in a JAR, an indented line indicates a file to be packaged:

```
<jarfile>.jar:
  path/in/jar/file_name.xul     (source/tree/location/file_name.xul)
```

The JAR location may be preceded with a base path between square brackets::

> **[base/path] <jarfile>.jar:** path/in/jar/file_name.xul (source/tree/location/file_name.xul)

In this case, the jar will be directly located under the given `base/bath`, while without a base path, it will be under a `chrome` directory.

If the JAR manifest and packaged file live in the same directory, the path and parenthesis can be omitted. In other words, the following two lines are equivalent:

```
path/in/jar/same_place.xhtml     (same_place.xhtml)
path/in/jar/same_place.xhtml
```

The source tree location may also be an *absolute* path (taken from the top of the source tree:

```
path/in/jar/file_name.xul     (/path/in/sourcetree/file_name.xul)
```

An asterisk marker (*) at the beginning of the line indicates that the file should be processed by the *Text Preprocessor* before being packaged:

```
* path/in/jar/preprocessed.xul  (source/tree/location/file_name.xul)
```

Preprocessed files always replace existing files, to ensure that changes in `#expand` or `#include` directives are picked up.

There is a special source-directory format for localized files (note the percent sign in the source file location): this format reads `localized.dtd` from the `en-US` directory if building an English version, and reads the file from the alternate localization source tree `/l10n/<locale>/path/localized.dtd` if building a localized version:

```
locale/path/localized.dtd     (%localized/path/localized.dtd)
```

The source tree location can also use wildcards, in which case the path in jar is expected to be a base directory. Paths before the wildcard are not made part of the destination path:

```
path/in/jar/                  (source/tree/location/*.xul)
```

The above will install all xul files under `source/tree/location` as `path/in/jar/*.xul`.

Register Chrome

Chrome Registration instructions are marked with a percent sign (`%`) at the beginning of the line, and must be part of the definition of a JAR file. Any additional percents signs are replaced with an appropriate relative URL of the JAR file being packaged:

```
% content global %path/in/jar/
% overlay chrome://blah/content/blah.xul chrome://foo/content/overlay.xul
```

There are two possible locations for a manifest file. If the chrome is being built into a standalone application, the `jar.mn` processor creates a `<jarfilename>.manifest` next to the JAR file itself. This is the default behavior.

If the build specifies `USE_EXTENSION_MANIFEST = 1`, the `jar.mn` processor creates a single `chrome.manifest` file suitable for registering chrome as an extension.

4.1.17 Defining Binaries for the Build System

One part of what the build system does is compile C/C++ and link the resulting objects to produce executables and/or libraries. This document describes the basics of defining what is going to be built and how. All the following describes constructs to use in moz.build files.

Source files

Source files to be used in a given directory are registered in the `SOURCES` and `UNIFIED_SOURCES` variables. `UNIFIED_SOURCES` have a special behavior in that they are aggregated by batches of 16, requiring, for example, that there are no conflicting variables in those source files.

`SOURCES` and `UNIFIED_SOURCES` are lists which must be appended to, and each append requires the given list to be alphanumerically ordered.

```
UNIFIED_SOURCES += [
    'FirstSource.cpp',
    'SecondSource.cpp',
    'ThirdSource.cpp',
]

SOURCES += [
```

```
    'OtherSource.cpp',
]
```

SOURCES and UNIFIED_SOURCES can contain a mix of different file types, for C, C++, and Objective C.

Static Libraries

To build a static library, other than defining the source files (see above), one just needs to define a library name with the Library template.

```
Library('foo')
```

The library file name will be libfoo.a on UNIX systems and foo.lib on Windows.

If the static library needs to aggregate other static libraries, a list of Library names can be added to the USE_LIBS variable. Like SOURCES, it requires the appended list to be alphanumerically ordered.

```
USE_LIBS += ['bar', 'baz']
```

If there are multiple directories containing the same Library name, it is possible to disambiguate by prefixing with the path to the wanted one (relative or absolute):

```
USE_LIBS += [
    '/path/from/topsrcdir/to/bar',
    '../relative/baz',
]
```

Note that the leaf name in those paths is the Library name, not an actual file name.

Note that currently, the build system may not create an actual library for static libraries. It is an implementation detail that shouldn't need to be worried about.

As a special rule, USE_LIBS is allowed to contain references to shared libraries. In such cases, programs and shared libraries linking this static library will inherit those shared library dependencies.

Intermediate (Static) Libraries

In many cases in the tree, static libraries are built with the only purpose of being linked into another, bigger one (like libxul). Instead of adding all required libraries to USE_LIBS for the bigger one, it is possible to tell the build system that the library built in the current directory is meant to be linked to that bigger library, with the FINAL_LIBRARY variable.

```
FINAL_LIBRARY = 'xul'
```

The FINAL_LIBRARY value must match a unique Library name somewhere in the tree.

As a special rule, those intermediate libraries don't need a Library name for themselves.

Shared Libraries

Sometimes, we want shared libraries, a.k.a. dynamic libraries. Such libraries are defined similarly to static libraries, using the SharedLibrary template instead of Library.

```
SharedLibrary('foo')
```

When this template is used, no static library is built. See further below to build both types of libraries.

With a `SharedLibrary` name of `foo`, the library file name will be `libfoo.dylib` on OSX, `libfoo.so` on ELF systems (Linux, etc.), and `foo.dll` on Windows. On Windows, there is also an import library named `foo.lib`, used on the linker command line. `libfoo.dylib` and `libfoo.so` are considered the import library name for, resp. OSX and ELF systems.

On OSX, one may want to create a special kind of dynamic library: frameworks. This is done with the `Framework` template.

```
Framework('foo')
```

With a `Framework` name of `foo`, the framework file name will be `foo`. This template however affects the behavior on all platforms, so it needs to be set only on OSX.

Executables

Executables, a.k.a. programs, are, in the simplest form, defined with the `Program` template.

```
Program('foobar')
```

On UNIX systems, the executable file name will be `foobar`, while on Windows, it will be `foobar.exe`.

Like static and shared libraries, the build system can be instructed to link libraries to the executable with `USE_LIBS`, listing various `Library` names.

In some cases, we want to create an executable per source file in the current directory, in which case we can use the `SimplePrograms` template

```
SimplePrograms([
    'FirstProgram',
    'SecondProgram',
])
```

Contrary to `Program`, which requires corresponding `SOURCES`, when using `SimplePrograms`, the corresponding `SOURCES` are implied. If the corresponding `sources` have an extension different from `.cpp`, it is possible to specify the proper extension:

```
SimplePrograms([
    'ThirdProgram',
    'FourthProgram',
], ext='.c')
```

Please note this construct was added for compatibility with what already lives in the mozilla tree ; it is recommended not to add new simple programs with sources with a different extension than `.cpp`.

Similar to `SimplePrograms`, is the `CppUnitTests` template, which defines, with the same rules, C++ unit tests programs. Like `SimplePrograms`, it takes an `ext` argument to specify the extension for the corresponding `SOURCES`, if it's different from `.cpp`.

Linking with system libraries

Programs and libraries usually need to link with system libraries, such as a widget toolkit, etc. Those required dependencies can be given with the `OS_LIBS` variable.

```
OS_LIBS += [
    'foo',
    'bar',
]
```

This expands to `foo.lib bar.lib` when building with MSVC, and `-lfoo -lbar` otherwise.

For convenience with `pkg-config`, `OS_LIBS` can also take linker flags such as `-L/some/path` and `-llib`, such that it is possible to directly assign `LIBS` variables from `CONFIG`, such as:

```
OS_LIBS += CONFIG['MOZ_PANGO_LIBS']
```

(assuming `CONFIG['MOZ_PANGO_LIBS']` is a list, not a string)

Like `USE_LIBS`, this variable applies to static and shared libraries, as well as programs.

Libraries from third party build system

Some libraries in the tree are not built by the moz.build-governed build system, and there is no `Library` corresponding to them.

However, `USE_LIBS` allows to reference such libraries by giving a full path (like when disambiguating identical `Library` names). The same naming rules apply as other uses of `USE_LIBS`, so only the library name without prefix and suffix shall be given.

```
USE_LIBS += [
    '/path/from/topsrcdir/to/third-party/bar',
    '../relative/third-party/baz',
]
```

Note that `/path/from/topsrcdir/to/third-party` and `../relative/third-party/baz` must lead under a subconfigured directory (a directory with an AC_OUTPUT_SUBDIRS in configure.in), or `security/nss`.

Building both static and shared libraries

When both types of libraries are required, one needs to set both `FORCE_SHARED_LIB` and `FORCE_STATIC_LIB` boolean variables.

```
FORCE_SHARED_LIB = True
FORCE_STATIC_LIB = True
```

But because static libraries and Windows import libraries have the same file names, either the static or the shared library name needs to be different than the name given to the `Library` template.

The `STATIC_LIBRARY_NAME` and `SHARED_LIBRARY_NAME` variables can be used to change either the static or the shared library name.

```
Library('foo')
STATIC_LIBRARY_NAME = 'foo_s'
```

With the above, on Windows, `foo_s.lib` will be the static library, `foo.dll` the shared library, and `foo.lib` the import library.

In some cases, for convenience, it is possible to set both `STATIC_LIBRARY_NAME` and `SHARED_LIBRARY_NAME`. For example:

```
Library('mylib')
STATIC_LIBRARY_NAME = 'mylib_s'
SHARED_LIBRARY_NAME = CONFIG['SHARED_NAME']
```

This allows to use `mylib` in the `USE_LIBS` of another library or executable.

When refering to a `Library` name building both types of libraries in `USE_LIBS`, the shared library is chosen to be linked. But sometimes, it is wanted to link the static version, in which case the `Library` name needs to be prefixed with `static:` in `USE_LIBS`

```
a/moz.build:
   Library('mylib')
   FORCE_SHARED_LIB = True
   FORCE_STATIC_LIB = True
   STATIC_LIBRARY_NAME = 'mylib_s'
b/moz.build:
   Program('myprog')
   USE_LIBS += [
       'static:mylib',
   ]
```

Miscellaneous

The `SDK_LIBRARY` boolean variable defines whether the library in the current directory is going to be installed in the SDK.

The `SONAME` variable declares a "shared object name" for the library. It defaults to the `Library` name or the `SHARED_LIBRARY_NAME` if set. When linking to a library with a `SONAME`, the resulting library or program will have a dependency on the library with the name corresponding to the `SONAME` instead of the `Library` name. This only impacts ELF systems.

```
a/moz.build:
   Library('mylib')
b/moz.build:
   Library('otherlib')
   SONAME = 'foo'
c/moz.build:
   Program('myprog')
   USE_LIBS += [
       'mylib',
       'otherlib',
   ]
```

On e.g. Linux, the above `myprog` will have DT_NEEDED markers for `libmylib.so` and `libfoo.so` instead of `libmylib.so` and `libotherlib.so` if there weren't a `SONAME`. This means the runtime requirement for `myprog` is `libfoo.so` instead of `libotherlib.so`.

Gecko-related binaries

Some programs or libraries are totally independent of Gecko, and can use the above mentioned templates. Others are Gecko-related in some way, and may need XPCOM linkage, mozglue. These things are tedious. A set of additional templates exists to ease defining such programs and libraries. They are essentially the same as the above mentioned templates, prefixed with "Gecko":

- `GeckoProgram`
- `GeckoSimplePrograms`
- `GeckoCppUnitTests`
- `GeckoSharedLibrary`
- `GeckoFramework`

There is also `XPCOMBinaryComponent` for XPCOM components, which is a special kind of library.

All the Gecko-prefixed templates take the same arguments as their non-Gecko-prefixed counterparts, and can take a few more arguments for non-standard cases. See the definition of `GeckoBinary` in build/gecko_templates.mozbuild for more details, but most usecases should not require these additional arguments.

4.1.18 Creating Toolchain Archives

There are various scripts in the repository for producing archives of the build tools (e.g. compilers and linkers) required to build.

Clang

See the `build/build-clang` directory. Read `build/build-clang/README` for more.

Windows

The `build/windows_toolchain.py` script is used to build and manage Windows toolchain archives containing Visual Studio executables, SDKs, etc.

The way Firefox build automation works is an archive containing the toolchain is produced and uploaded to an internal Mozilla server. The build automation will download, verify, and extract this archive before building. The archive is self-contained so machines don't need to install Visual Studio, SDKs, or various other dependencies. Unfortunately, Microsoft's terms don't allow Mozilla to distribute this archive publicly. However, the same tool can be used to create your own copy.

Configuring Your System

It is **highly** recommended to perform this process on a fresh installation of Windows 7 or 10 (such as in a VM). Installing all updates through Windows Update is not only acceptable - it is encouraged. Although it shouldn't matter.

Next, install Visual Studio 2015 Community. The download link can be found at https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx. Be sure to follow these install instructions:

1. Choose a `Custom` installation and click `Next`
2. Select `Programming Languages` -> `Visual C++` (make sure all sub items are selected)
3. Under `Windows and Web Development` uncheck everything except `Universal Windows App Development Tools` and the items under it (should be `Tools (1.3.1)...` and the `Windows 10 SDK`).

Once Visual Studio 2015 Community has been installed, from a checkout of mozilla-central, run something like the following to produce a ZIP archive:

```
$ ./mach python build/windows_toolchain.py create-zip vs2015u2
```

The produced archive will be the argument to `create-zip` + `.zip`.

Firefox for Android with Gradle

To build Firefox for Android with Gradle in automation, archives containing both the Gradle executable and a Maven repository comprising the exact build dependencies are produced and uploaded to an internal Mozilla server. The build automation will download, verify, and extract these archive before building. These archives provide a self-contained Gradle and Maven repository so that machines don't need to fetch additional Maven dependencies at build time. (Gradle and the downloaded Maven dependencies can be both redistributed publicly.)

Archiving the Gradle executable is straight-forward, but archiving a local Maven repository is not. Therefore a special Task Cluster Docker image and job exist for producing the required archives. The Docker image definition is rooted in `taskcluster/docker/android-gradle-build`. The Task Cluster job definition is in `testing/taskcluster/tasks/builds/android_api_15_gradle_dependencies.yml`. The job runs in a container based on the custom Docker image and spawns a Sonatype Nexus proxying Maven repository process in the background. The job builds Firefox for Android using Gradle and the in-tree Gradle configuration rooted at `build.gradle`. The spawned proxying Maven repository downloads external dependencies and collects them. After the Gradle build completes, the job archives the Gradle version used to build, and the downloaded Maven repository, and exposes them as Task Cluster artifacts.

Here is an example try job fetching these dependencies. The resulting task produced a Gradle archive and a Maven repository archive. These archives were then uploaded (manually) to Mozilla automation using tooltool for consumption in Gradle builds.

To update the version of Gradle in the archive produced, update `gradle/wrapper/gradle-wrapper.properties`. Be sure to also update the SHA256 checksum to prevent poisoning the build machines!

To update the versions of Gradle dependencies used, update `dependencies` sections in the in-tree Gradle configuration rooted at `build.gradle`. Once you are confident your changes build locally, push a fresh try build with an invocation like:

```
$ hg push-to-try -m "try: -b o -p android-api-15-gradle-dependencies"
```

Then upload your archives to tooltool, update the in-tree manifests in `mobile/android/config/tooltool-manifests`, and push a fresh try build.

4.1.19 Localization (l10n)

Single-locale language repacks

To save on build time, the build system and automation collaborate to allow downloading a packaged en-US Firefox, performing some locale-specific post-processing, and re-packaging a locale-specific Firefox. Such artifacts are termed "single-locale language repacks". There is another concept of a "multi-locale language build", which is more like a regular build and less like a re-packaging post-processing step.

There are scripts in-tree in mozharness to orchestrate these re-packaging steps for Desktop and Android but they rely heavily on buildbot information so they are almost impossible to run locally.

The following instructions are extracted from the Android script with hg hash 494289c7, and may need to be updated and slightly modified for Desktop.

Step by step instructions for Android

This assumes that `$AB_CD` is the locale you want to repack with; I tested with "ar" and "en-GB".

> **Warning:** l10n repacks do not work with artifact builds. Repackaging compiles no code so supporting `--disable-compile-environment` would not save much, if any, time.

1. You must have a built and packaged object directory, or a pre-built `en-US` package.

```
./mach build
./mach package
```

2. Clone `l10n-central/$AB_CD` so that it is a sibling to your `mozilla-central` directory.

```
$ ls -al
mozilla-central
...
$ mkdir -p l10n-central
$ hg clone https://hg.mozilla.org/l10n-central/$AB_CD l10n-central/$AB_CD
$ ls -al
mozilla-central
l10n-central/$AB_CD
...
```

3. Copy your `mozconfig` to `mozconfig.l10n` and add the following.

```
ac_add_options --with-l10n-base=../../l10n-central
ac_add_options --disable-tests
mk_add_options MOZ_OBJDIR=./objdir-l10n
```

4. Configure and prepare the l10n object directory.

```
MOZCONFIG=mozconfig.l10n ./mach configure
MOZCONFIG=mozconfig.l10n ./mach build -C config export
MOZCONFIG=mozconfig.l10n ./mach build buildid.h
```

5. Copy your built package and unpack it into the l10n object directory.

```
cp $OBJDIR/dist/fennec-*en-US*.apk ./objdir-l10n/dist
MOZCONFIG=mozconfig.l10n ./mach build -C mobile/android/locales unpack
```

6. Run the `compare-locales` script to write locale-specific changes into `objdir-l10n/merged`.

```
MOZCONFIG=mozconfig.l10n ./mach compare-locales --merge-dir objdir-l10n/merged $AB_CD
```

7. Finally, repackage using the locale-specific changes.

```
MOZCONFIG=mozconfig.l10n LOCALE_MERGEDIR=`realpath objdir-l10n/merged` ./mach build -C mobile/an
```

   (Note the absolute path for `LOCALE_MERGEDIR`.) You should find a re-packaged build at `objdir-l10n/dist/fennec-*$AB_CD*.apk`.

4.2 integrated development environment (IDE)

4.2.1 Android Eclipse Projects

The build system contains alpha support for generating Android Eclipse project files to aid with development.

To generate Android Eclipse project files, you'll need to have a fully built and packaged tree:

```
mach build && mach package
```

(This is because Eclipse itself packages an APK containing `omni.ja`, and `omni.ja` is only assembled during packaging.)

Then, simply generate the Android Eclipse build backend:

```
mach build-backend -b AndroidEclipse
```

If all goes well, the path to the generated projects should be printed (currently, `$OBJDIR/android_eclipse`).

To use the generated Android Eclipse project files, you'll need to have a recent version of Eclipse (see *Tested Versions*) with the Eclipse ADT plugin installed. You can then import all the projects into Eclipse using *File* > *Import ...* > *General* > *Existing Projects into Workspace*.

Updating Project Files

As you pull and update the source tree, your Android Eclipse files may fall out of sync with the build configuration. The tree should still build fine from within Eclipse, but source files may be missing and in rare circumstances Eclipse's index may not have the proper build configuration.

To account for this, you'll want to periodically regenerate the Android Eclipse project files. You can do this by running `mach build && mach package && mach build-backend -b AndroidEclipse` from the command line. It's a good idea to refresh and clean build all projects in Eclipse after doing this.

In future, we'd like to include an Android Eclipse run configuration or build target that integrates updating the project files.

Currently, regeneration rewrites the original project files. **If you've made any customizations to the projects, they will likely get overwritten.** We would like to improve this user experience in the future.

Troubleshooting

If Eclipse's builder gets confused, you should always refresh and clean build all projects. If Eclipse's builder is continually confused, you can see a log of what is happening at `$OBJDIR/android_eclipse/build.log`.

If you run into memory problems executing `dex`, you should Increase Eclipse's memory limits.

The produced Android Eclipse project files are unfortunately not portable. Please don't move them around.

Structure of Android Eclipse projects

The Android Eclipse backend generates several projects spanning Fennec itself and its tests. You'll mostly interact with the *Fennec* project itself.

In future, we'd like to expand this documentation to include some of the technical details of how the Eclipse integration works, and how to add additional Android Eclipse projects using the `moz.build` system.

Tested Versions

OS	Version	Working as of
Mac OS X	Luna (Build id: 20130919-0819)	February 2014
Mac OS X	Kepler (Build id: 20131219-0014)	February 2014
Mac OS X 10.8.5	Kepler (Build id: 20130919-0819)	February 2014

4.2.2 Cpp Eclipse Projects

For additional information on using Eclipse CDT see the MDN page.

The build system contains alpha support for generating C++ Eclipse project files to aid with development.

Please report bugs to bugzilla and make them depend on bug 973770.

To generate a C++ Eclipse project files, you'll need to have a fully built tree:

```
mach build
```

Then, simply generate the Android Eclipse build backend:

```
mach build-backend -b CppEclipse
```

If all goes well, the path to the generated workspace should be printed (currently, `$OBJDIR/android_eclipse`).

To use the generated Android Eclipse project files, you'll need to have a Eclipse CDT 8.3 (We plan to follow the latest Eclipse release) Eclipse CDT plugin installed. You can then import all the projects into Eclipse using *File* > *Import ...* > *General* > *Existing Projects into Workspace* -only- if you have not ran the background indexer.

Updating Project Files

As you pull and update the source tree, your C++ Eclipse files may fall out of sync with the build configuration. The tree should still build fine from within Eclipse, but source files may be missing and in rare circumstances Eclipse's index may not have the proper build configuration.

To account for this, you'll want to periodically regenerate the Android Eclipse project files. You can do this by running `mach build && mach build-backend -b CppEclipse` from the command line.

Currently, regeneration rewrites the original project files. **If you've made any customizations to the projects, they will likely get overwritten.** We would like to improve this user experience in the future.

4.2.3 Visual Studio Projects

The build system contains alpha support for generating Visual Studio project files to aid with development.

To generate Visual Studio project files, you'll need to have a configured tree:

```
mach configure
```

(If you have built recently, your tree is already configured.)

Then, simply generate the Visual Studio build backend:

```
mach build-backend -b VisualStudio
```

If all goes well, the path to the generated Solution (`.sln`) file should be printed. You should be able to open that solution with Visual Studio 2010 or newer.

Currently, output is hard-coded to the Visual Studio 2010 format. If you open the solution in a newer Visual Studio release, you will be prompted to upgrade projects. Simply click through the wizard to do that.

Structure of Solution

The Visual Studio solution consists of hundreds of projects spanning thousands of files. To help with organization, the solution is divided into the following trees/folders:

Build Targets This folder contains common build targets. The *full* project is used to perform a full build. The *binaries* project is used to build just binaries. The *visual-studio* project can be built to regenerate the Visual Studio project files.

Performing the *clean* action on any of these targets will clean the *entire* build output.

Binaries This folder contains common binaries that can be executed from within Visual Studio. If you are building the Firefox desktop application, the *firefox* project will launch firefox.exe. You probably want one of these set to your startup project.

Libraries This folder contains entries for each static library that is produced as part of the build. These roughly correspond to each directory in the tree containing C/C++. e.g. code from `dom/base` will be contained in the `dom_base` project.

These projects don't do anything when built. If you build a project here, the *binaries* build target project is built.

Updating Project Files

As you pull and update the source tree, your Visual Studio files may fall out of sync with the build configuration. The tree should still build fine from within Visual Studio. But source files may be missing and IntelliSense may not have the proper build configuration.

To account for this, you'll want to periodically regenerate the Visual Studio project files. You can do this within Visual Studio by building the `Build Targets :: visual-studio` project or by running `mach build-backend -b VisualStudio` from the command line.

Currently, regeneration rewrites the original project files. **If you've made any customizations to the solution or projects, they will likely get overwritten.** We would like to improve this user experience in the future.

Moving Project Files Around

The produced Visual Studio solution and project files should be portable. If you want to move them to a non-default directory, they should continue to work from wherever they are. If they don't, please file a bug.

Invoking mach through Visual Studio

It's possible to build the tree via Visual Studio. There is some light magic involved here.

Alongside the Visual Studio project files is a batch script named `mach.bat`. This batch script sets the environment variables present in your *MozillaBuild* development environment at the time of Visual Studio project generation and invokes *mach* inside an msys shell with the arguments specified to the batch script. This script essentially allows you to invoke mach commands inside the MozillaBuild environment without having to load MozillaBuild.

While projects currently only utilize the `mach build` command, the batch script does not limit it's use: any mach command can be invoked. Developers may abuse this fact to add custom projects and commands that invoke other mach commands.

4.3 mozbuild

mozbuild is a Python package containing a lot of the code for the Mozilla build system.

4.3.1 mozbuild

mozbuild is a Python package providing functionality used by Mozilla's build system.

Modules Overview

- mozbuild.backend – Functionality for producing and interacting with build backends. A build backend is an entity that consumes build system metadata (from mozbuild.frontend) and does something useful with it (typically writing out files that can be used by a build tool to build the tree).
- mozbuild.compilation – Functionality related to compiling. This includes managing compiler warnings.

- mozbuild.frontend – Functionality for reading build frontend files (what defines the build system) and converting them to data structures which are fed into build backends to produce backend configurations.
- mozpack – Functionality related to packaging builds.

Overview

The build system consists of frontend files that define what to do. They say things like "compile X" "copy Y."

The mozbuild.frontend package contains code for reading these frontend files and converting them to static data structures. The set of produced static data structures for the tree constitute the current build configuration.

There exist entities called build backends. From a high level, build backends consume the build configuration and do something with it. They typically produce tool-specific files such as make files which can be used to build the tree.

Piecing it all together, we have frontend files that are parsed into data structures. These data structures are fed into a build backend. The output from build backends is used by builders to build the tree.

4.3.2 dumbmake

dumbmake is a simple dependency tracker for make.

It turns lists of make targets into longer lists of make targets that include dependencies. For example:

> netwerk, package

might be turned into

> netwerk, netwerk/build, toolkit/library, package

The dependency list is read from the plain text file *topsrcdir/build/dumbmake-dependencies*. The format best described by example:

> **build_this** when_this_changes

Interpret this to mean that *build_this* is a dependency of *when_this_changes*. More formally, a line (CHILD) indented more than the preceding line (PARENT) means that CHILD should trigger building PARENT. That is, building CHILD will trigger building first CHILD and then PARENT.

This structure is recursive:

> **build_this_when_either_change**
>
> > **build_this_only_when** this_changes

This means that *build_this_when_either_change* is a dependency of *build_this_only_when* and *this_changes*, and *build_this_only_when* is a dependency of *this_changes*. Building *this_changes* will build first *this_changes*, then *build_this_only_when*, and finally *build_this_when_either_change*.

CHAPTER 5

WebIDL

WebIDL describes interfaces web browsers are supposed to implement.

The interaction between WebIDL and the build system is somewhat complex. This document will attempt to explain how it all works.

5.1 Overview

`.webidl` files throughout the tree define interfaces the browser implements. Since Gecko/Firefox is implemented in C++, there is a mechanism to convert these interfaces and associated metadata to C++ code. That's where the build system comes into play.

All the code for interacting with `.webidl` files lives under `dom/bindings`. There is code in the build system to deal with WebIDLs explicitly.

5.2 WebIDL source file flavors

Not all `.webidl` files are created equal! There are several flavors, each represented by a separate symbol from *mozbuild Sandbox Symbols*.

WEBIDL_FILES Refers to regular/static `.webidl` files. Most WebIDL interfaces are defined this way.

GENERATED_EVENTS_WEBIDL_FILES In addition to generating a binding, these `.webidl` files also generate a source file implementing the event object in C++

PREPROCESSED_WEBIDL_FILES The `.webidl` files are generated by preprocessing an input file. They otherwise behave like *WEBIDL_FILES*.

TEST_WEBIDL_FILES Like *WEBIDL_FILES* but the interfaces are for testing only and aren't shipped with the browser.

PREPROCESSED_TEST_WEBIDL_FILES Like *TEST_WEBIDL_FILES* except the `.webidl` is obtained via preprocessing, much like *PREPROCESSED_WEBIDL_FILES*.

GENERATED_WEBIDL_FILES The `.webidl` for these is obtained through an *external* mechanism. Typically there are custom build rules for producing these files.

5.3 Producing C++ code

The most complicated part about WebIDLs is the process by which `.webidl` files are converted into C++.

This process is handled by code in the *mozwebidlcodegen* package. *mozwebidlcodegen.WebIDLCodegenManager* is specifically where you want to look for how code generation is performed. This includes complex dependency management.

5.4 Requirements

This section aims to document the build and developer workflow requirements for WebIDL.

Parser unit tests There are parser tests provided by `dom/bindings/parser/runtests.py` that should run as part of `make check`. There must be a mechanism to run the tests in *human* mode so they output friendly error messages.

The current mechanism for this is `mach webidl-parser-test`.

Mochitests There are various mochitests under `dom/bindings/test`. They should be runnable through the standard mechanisms.

Working with test interfaces `TestExampleGenBinding.cpp` calls into methods from the `TestExampleInterface` and `TestExampleProxyInterface` interfaces. These interfaces need to be generated as part of the build. These interfaces should not be exported or packaged.

There is a `compiletests` make target in `dom/bindings` that isn't part of the build that facilitates turnkey code generation and test file compilation.

Minimal rebuilds Reprocessing every output for every change is expensive. So we don't inconvenience people changing `.webidl` files, the build system should only perform a minimal rebuild when sources change.

This logic is mostly all handled in *mozwebidlcodegen.WebIDLCodegenManager*. The unit tests for that Python code should adequately test typical rebuild scenarios.

Bug 940469 tracks making the existing implementation better.

Explicit method for performing codegen There needs to be an explicit method for invoking code generation. It needs to cover regular and test files.

This is implemented via `make export` in `dom/bindings`.

No-op binding generation should be fast So developers touching `.webidl` files are not inconvenienced, no-op binding generation should be fast. Watch out for the build system processing large dependency files it doesn't need in order to perform code generation.

Ability to generate example files *Any* interface can have example `.h`/`.cpp` files generated. There must be a mechanism to facilitate this.

This is currently facilitated through `mach webidl-example`. e.g. `mach webidl-example HTMLStyleElement`.

CHAPTER 6

Graphics

The graphics team's documentation is currently using doxygen. We're tracking the work to integrate it better at https://bugzilla.mozilla.org/show_bug.cgi?id=1150232.

For now you can read the graphics source code documentation here:

http://people.mozilla.org/~bgirard/doxygen/gfx/

CHAPTER 7

Firefox for Android

Contents:

7.1 Runtime locale switching in Fennec

Bug 917480 built on Bug 936756 to allow users to switch between supported locales at runtime, within Fennec, without altering the system locale.

This document aims to describe the overall architecture of the solution, along with guidelines for Fennec developers.

7.1.1 Overview

There are two places that locales are relevant to an Android application: the Java `Locale` object and the Android configuration itself.

Locale switching involves manipulating these values (to affect future UI), persisting them for future activities, and selectively redisplaying existing UI elements to give the appearance of responsive switching.

The user's choice of locale is stored in a per-app pref, `"locale"`. If missing, the system default locale is used. If set, it should be a locale code like `"es"` or `"en-US"`.

`BrowserLocaleManager` takes care of updating the active locale when asked to do so. It also manages persistence and retrieval of the locale preference.

The question, then, is when to do so.

7.1.2 Locale events

One might imagine that we need only set the locale when our Application is instantiated, and when a new locale is set. Alas, that's not the case: whenever there's a configuration change (*e.g.*, screen rotation), when a new activity is started, and at other apparently random times, Android will supply our activities with a configuration that's been reset to the system locale.

For this reason, each starting activity must ask `BrowserLocaleManager` to fix its locale.

Ideally, we also need to perform some amount of work when our configuration changes, when our activity is resumed, and perhaps when a result is returned from another activity, if that activity can change the app locale (as is the case for any activity that calls out to `GeckoPreferences` – see `BrowserApp#onActivityResult`).

`GeckoApp` itself does some additional work, because it has particular performance constraints, and also is the typical root of the preferences activity.

Here's an example of the work that a typical activity should do:

```
// This is cribbed from o.m.g.sync.setup.activities.LocaleAware.
public static void initializeLocale(Context context) {
  final LocaleManager localeManager = BrowserLocaleManager.getInstance();
  if (Build.VERSION.SDK_INT < Build.VERSION_CODES.GINGERBREAD) {
    localeManager.getAndApplyPersistedLocale(context);
  } else {
    final StrictMode.ThreadPolicy savedPolicy = StrictMode.allowThreadDiskReads();
    StrictMode.allowThreadDiskWrites();
    try {
      localeManager.getAndApplyPersistedLocale(context);
    } finally {
      StrictMode.setThreadPolicy(savedPolicy);
    }
  }
}

@Override
public void onConfigurationChanged(Configuration newConfig) {
  final LocaleManager localeManager = BrowserLocaleManager.getInstance();
  final Locale changed = localeManager.onSystemConfigurationChanged(this, getResources(), newConfig,
  if (changed != null) {
    // Redisplay to match the locale.
    onLocaleChanged(BrowserLocaleManager.getLanguageTag(changed));
  }
}

@Override
public void onCreate(Bundle icicle) {
  // Note that we don't do this in onResume. We should,
  // but it's an edge case that we feel free to ignore.
  // We also don't have a hook in this example for when
  // the user picks a new locale.
  initializeLocale(this);

  super.onCreate(icicle);
}
```

`GeckoApplication` itself handles correcting locales when the configuration changes; your activity shouldn't need to do this itself. See `GeckoApplication`'s and `GeckoApp`'s `onConfigurationChanged` methods.

7.1.3 System locale changes

Fennec can be in one of two states.

If the user has not explicitly chosen a Fennec-specific locale, we say we are "mirroring" the system locale.

When we are not mirroring, system locale changes do not impact Fennec and are essentially ignored; the user's locale selection is the only thing we care about, and we actively correct incoming configuration changes to reflect the user's chosen locale.

By contrast, when we are mirroring, system locale changes cause Fennec to reflect the new system locale, as if the user picked the new locale.

When the system locale changes when we're mirroring, your activity will receive an `onConfigurationChanged` call. Simply pass this on to `BrowserLocaleManager`, and then handle the response appropriately.

7.1.4 Further reference

`GeckoPreferences`, `GeckoApp`, and `BrowserApp` are excellent resources for figuring out what you should do.

7.2 UI Telemetry

Fennec records UI events using a telemetry framework called UITelemetry.

Some links:

- Project page
- Wiki page
- User research notes

7.2.1 Sessions

Sessions are essentially scopes. They are meant to provide context to events; this allows events to be simpler and more reusable. Sessions are usually bound to some component of the UI, some user action with a duration, or some transient state.

For example, a session might be begun when a user begins interacting with a menu, and stopped when the interaction ends. Or a session might encapsulate period of no network connectivity, the first five seconds after the browser launched, the time spent with an active download, or a guest mode session.

Sessions implicitly record the duration of the interaction.

A simple use-case for sessions is the bookmarks panel in about:home. We start a session when the user swipes into the panel, and stop it when they swipe away. This bookmarks session does two things: firstly, it gives scope to any generic event that may occur within the panel (*e.g.*, loading a URL). Secondly, it allows us to figure out how much time users are spending in the bookmarks panel.

To start a session, call `Telemetry.startUISession(String sessionName)`.

`sessionName` The name of the session. Session names should be brief, lowercase, and should describe which UI component the user is interacting with. In certain cases where the UI component is dynamic, they could include an ID, essential to identifying that component. An example of this is dynamic home panels: we use session names of the format `homepanel:<panel_id>` to identify home panel sessions.

To stop a session, call `Telemetry.stopUISession(String sessionName, String reason)`.

`sessionName` The name of the open session

`reason` (Optional) A descriptive cause for ending the session. It should be brief, lowercase, and generic so it can be reused in different places. Examples reasons are:

`switched` The user transitioned to a UI element of equal level.

`exit` The user left for an entirely different element.

7.2.2 Events

Events capture key occurrences. They should be brief and simple, and should not contain sensitive or excess information. Context for events should come from the session (scope). An event can be created with four fields (via `Telemetry.sendUIEvent`): `action`, `method`, `extras`, and `timestamp`.

action The name of the event. Should be brief and lowercase. If needed, you can make use of namespacing with a '.' separator. Example event names: `panel.switch`, `panel.enable`, `panel.disable`, `panel.install`.

method (Optional) Used for user actions that can be performed in many ways. This field specifies the method by which the action was performed. For example, users can add an item to their reading list either by long-tapping the reader icon in the address bar, or from within reader mode. We would use the same event name for both user actions but specify two methods: `addressbar` and `readermode`.

extras (Optional) For extra information that may be useful in understanding the event. Make an effort to keep this brief.

timestamp (Optional) The time at which the event occurred. If not specified, this field defaults to the current value of the realtime clock.

7.2.3 Versioning

As a we improve on our Telemetry methods, it is foreseeable that our probes will change over time. Different versions of a probe could carry different data or have different interpretations on the server-side. To make it easier for the server to handle these changes, you should add version numbers to your event and session names. An example of a versioned session is `homepanel.1`; this is version 1 of the `homepanel` session. This approach should also be applied to event names, an example being: `panel.enable.1` and `panel.enable.2`.

7.2.4 Clock

Times are relative to either elapsed realtime (an arbitrary monotonically increasing clock that continues to tick when the device is asleep), or elapsed uptime (which doesn't tick when the device is in deep sleep). We default to elapsed realtime.

See the documentation in the source for more details.

7.2.5 Dictionary

Events

action.1 Generic action, usually for tracking menu and toolbar actions.

cancel.1 Cancel a state, action, etc.

cast.1 Start casting a video.

edit.1 Sent when the user edits a top site.

launch.1 Launching (opening) an external application. Note: Only used in JavaScript for now.

loadurl.1 Loading a URL.

locale.browser.reset.1 When the user chooses "System default" in the browser locale picker.

locale.browser.selected.1 When the user chooses a locale in the browser locale picker. The selected locale is provided as the extra.

locale.browser.unselected.1 When the user chose a different locale in the browser locale picker, this event is fired with the previous locale as the extra. If the previous locale could not be determined, "unknown" is provided.

neterror.1 When the user performs actions on the in-content network error page. This should probably be a `Session`, but it's difficult to start and stop the session reliably.

panel.hide.1 Hide a built-in home panel.

panel.move.1 Move a home panel up or down.

panel.remove.1 Remove a custom home panel.

panel.setdefault.1 Set default home panel.

panel.show.1 Show a hidden built-in home panel.

pin.1, unpin.1 Sent when the user pinned or unpinned a top site.

policynotification.success.1:true Sent when a user has accepted the data notification policy. Can be `false` instead of `true` if an error occurs.

sanitize.1 Sent when the user chooses to clear private data.

save.1, unsave.1 Saving or unsaving a resource (reader, bookmark, etc.) for viewing later.

search.1 Sent when the user performs a search. Currently used in the search activity.

search.remove.1 Sent when the user removes a search engine.

search.restore.1 Sent when the user restores the search engine configuration back to the built-in configuration.

search.setdefault.1 Sent when the user sets a search engine to be the default.

share.1 Sharing content.

show.1 Sent when a contextual UI element is shown to the user.

undo.1 Sent when performing an undo-style action, like undoing a closed tab.

Methods

actionbar Action triggered from an ActionBar UI.

back Action triggered from the back button.

banner Action triggered from a banner (such as HomeBanner).

button Action triggered from a button. Note: Only used in JavaScript for now.

content Action triggered from a content page.

contextmenu Action triggered from a contextmenu. Could be from chrome or content.

dialog Action triggered from a dialog.

doorhanger Action triggered from a doorhanger popup prompt.

griditem Action triggered from a griditem, such as those used in Top Sites panel.

homescreen Action triggered from a homescreen shortcut icon.

intent Action triggered from a system Intent, usually sent from the OS.

list Action triggered from an unmanaged list of items, usually provided by the OS.

listitem Action triggered from a listitem.

menu Action triggered from the main menu.

notification Action triggered from a system notification.

pageaction Action triggered from a pageaction, displayed in the URL bar.

service Action triggered from an automatic system making a decision.

settings Action triggered from a content page.

shareoverlay Action triggered from a content page.

suggestion Action triggered from a suggested result, like those from search engines or default tiles.

system Action triggered from an OS level action, like application foreground / background.

toast Action triggered from an unobtrusive, temporary notification.

widget Action triggered from a widget placed on the homescreen.

Sessions

awesomescreen.1 Awesomescreen (including frecency search) is active.

firstrun.1 Started the very first time we believe the application has been launched.

frecency.1 Awesomescreen frecency search is active.

homepanel.1 Started when a user enters a given home panel. Session name is dynamic, encoded as "home-panel.1:<panel_id>" Built-in home panels have fixed IDs

reader.1 Reader viewer becomes active in the foreground.

searchactivity.1 Started when the user launches the search activity (onStart) and stopped when they leave the search activity.

settings.1 Settings activity is active.

7.3 Install tracking with the Adjust SDK

Fennec (Firefox for Android) tracks certain types of installs using a third party install tracking framework called Adjust. The intention is to determine the origin of Fennec installs by answering the question, "Did this user on this device install Fennec in response to a specific advertising campaign performed by Mozilla?"

Mozilla is using a third party framework in order to answer this question for the Firefox for Android 38.0.5 release. We hope to remove the framework from Fennec in the future.

The framework consists of a software development kit (SDK) built into Fennec and a data-collecting Internet service backend run by the German company adjust GmbH. The Adjust SDK is open source and MIT licensed: see the github repository. Fennec ships a copy of the SDK (currently not modified from upstream) in `mobile/android/thirdparty/com/adjust/sdk`. The SDK is documented at https://docs.adjust.com.

7.3.1 Data collection

When is data collected and sent to the Adjust backend?

Data is never collected (or sent to the Adjust backend) unless

- the Fennec binary is an official Mozilla binary [1]; and
- the release channel is Release or Beta [2].

If both of the above conditions are true, then data is collected and sent to the Adjust backend in the following two circumstances: first, when

[1] Data is not sent for builds not produced by Mozilla: this would include redistributors such as the Palemoon project.
[2] Data is not sent for Aurora, Nightly, or custom builds.

- Fennec is started on the device [3].

Second, when

- the Fennec binary was installed from the Google Play Store; and
- the Google Play Store sends the installed Fennec binary an INSTALL_REFERRER Intent, and the received Intent includes Google Play Store campaign tracking information. This happens when thea Google Play Store install is in response to a campaign-specific Google Play Store link. For details, see the developer documentation at https://developers.google.com/analytics/devguides/collection/android/v4/campaigns.

In these two limited circumstances, data is collected and sent to the Adjust backend.

Where does data sent to the Adjust backend go?

The Adjust SDK is hard-coded to send data to the endpoint https://app.adjust.com. The endpoint is defined by `com.adjust.sdk.Constants.BASE_URL` at https://hg.mozilla.org/mozilla-central/file/f76f02793f7a/mobile/android/thirdparty/com/adjust/sdk/Constants.java#l27.

The Adjust backend then sends a limited subset of the collected data – limited but sufficient to uniquely identify the submitting device – to a set of advertising network providers that Mozilla elects to share the collected data with. Those advertising networks then confirm or deny that the identifying information corresponds to a specific advertising campaign performed by Mozilla.

What data is collected and sent to the Adjust backend?

The Adjust SDK collects and sends two messages to the Adjust backend. The messages have the following parameters:

```
V/Adjust  ( 6508): Parameters:
V/Adjust  ( 6508):    screen_format    normal
V/Adjust  ( 6508):    device_manufacturer samsung
V/Adjust  ( 6508):    session_count    1
V/Adjust  ( 6508):    device_type      phone
V/Adjust  ( 6508):    screen_size      normal
V/Adjust  ( 6508):    package_name     org.mozilla.firefox
V/Adjust  ( 6508):    app_version      39.0a1
V/Adjust  ( 6508):    android_uuid     <guid>
V/Adjust  ( 6508):    display_width    720
V/Adjust  ( 6508):    country          GB
V/Adjust  ( 6508):    os_version       18
V/Adjust  ( 6508):    needs_attribution_data 0
V/Adjust  ( 6508):    environment      sandbox
V/Adjust  ( 6508):    device_name      Galaxy Nexus
V/Adjust  ( 6508):    os_name          android
V/Adjust  ( 6508):    tracking_enabled 1
V/Adjust  ( 6508):    created_at       2015-03-24T17:53:38.452Z-0400
V/Adjust  ( 6508):    app_token        <private>
V/Adjust  ( 6508):    screen_density   high
V/Adjust  ( 6508):    language         en
V/Adjust  ( 6508):    display_height   1184
V/Adjust  ( 6508):    gps_adid         <guid>

V/Adjust  ( 6508): Parameters:
V/Adjust  ( 6508):    needs_attribution_data 0
```

[3] *Started* means more than just when the user taps the Fennec icon or otherwise causes the Fennec user interface to appear directly. It includes, for example, when a Fennec service (like the Update Service, or Background Sync), starts and Fennec was not previously running on the device. See http://developer.android.com/reference/android/app/Application.html#onCreate%28%29 for details.

```
V/Adjust  ( 6508):    app_token        <private>
V/Adjust  ( 6508):    environment      production
V/Adjust  ( 6508):    android_uuid     <guid>
V/Adjust  ( 6508):    tracking_enabled 1
V/Adjust  ( 6508):    gps_adid         <guid>
```

The available parameters (including ones not exposed to Mozilla) are documented at https://partners.adjust.com/placeholders/.

Notes on what data is collected

The *android_uuid* uniquely identifies the device.

The *gps_adid* is a Google Advertising ID. It is capable of uniquely identifying a device to any advertiser, across all applications. If a Google Advertising ID is not available, Adjust may fall back to an Android ID, or, as a last resort, the device's WiFi MAC address.

The *tracking_enabled* flag is only used to allow or disallow contextual advertising to be sent to a user. It can be, and is, ignored for general install tracking of the type Mozilla is using the Adjust SDK for. (This flag might be used by consumers using the Adjust SDK to provide in-App advertising.)

It is not clear how much entropy their is in the set of per-device parameters that do not *explicitly* uniquely identify the device. That is, it is not known if the device parameters are likely to uniquely fingerprint the device, in the way that user agent capabilities are likely to uniquely fingerprint the user.

7.3.2 Technical notes

Build flags controlling the Adjust SDK integration

Add the following to your mozconfig to compile with the Adjust SDK:

```
export MOZ_INSTALL_TRACKING=1
export MOZ_NATIVE_DEVICES=1
export RELEASE_BUILD=1
ac_add_options --with-adjust-sdk-keyfile="$topsrcdir/mobile/android/base/adjust-sdk-sandbox.token"
```

`MOZ_NATIVE_DEVICES` && `RELEASE_BUILD` are required for an unknown reason. If you build without them, the `StubAdjustHelper` will be returned.

No trace of the Adjust SDK should be present in Fennec if `MOZ_INSTALL_TRACKING` is not defined.

Access to the Adjust backend is controlled by a private App-specific token. Fennec's token is managed by Release Engineering and should not be exposed if at all possible; for example, it should *not* leak to build logs. The value of the token is read from the file specified using the `configure` flag `--with-adjust-sdk-keyfile=KEYFILE` and stored in the build variable `MOZ_INSTALL_TRACKING_ADJUST_SDK_APP_TOKEN`. The mozconfig specified above defaults to submitting data to a special Adjust sandbox allowing a developer to test Adjust without submitting false data to our backend.

We throw an assertion if `MOZ_INSTALL_TRACKING` is specified but `--with-adjust-sdk-keyfile` is not to ensure our builders have a proper adjust token for release and beta builds. It's great to catch some errors at compile-time rather than in release. That being said, ideally we'd specify a default `--with-adjust-sdk-keyfile` for developer builds but I don't know how to do that.

Technical notes on the Adjust SDK integration

The *Adjust install tracking SDK* is a pure-Java library that is conditionally compiled into Fennec. It's not trivial to integrate such conditional feature libraries into Fennec without pre-processing. To minimize such pre-processing, we define a trivial `AdjustHelperInterface` and define two implementations: the real `AdjustHelper`, which requires the Adjust SDK, and a no-op `StubAdjustHelper`, which has no additional requirements. We use the existing pre-processed `AppConstants.java.in` to switch, at build-time, between the two implementations.

Notes and links

7.4 Shipping Default Domains

Firefox for Mobile (Android and iOS) ships sets of default content in order to improve the first-run experience. There are two primary places where default sets of domains are used: URLBar domain auto-completion, and Top Sites suggested thumbnails.

The source of these domains is typically the Alexa top sites lists, global and by-country. Before shipping the sets of domains, the lists are sanitized.

7.4.1 Domain Auto-completion

As you type in the URLBar, Firefox will scan your history and auto-complete previously visited domains that match what you have entered. This can make navigating to web sites faster because it can avoid significant amounts of typing. During your first few uses, Firefox does not have any history and you are forced to type full URLs. Shipping a set of top domains provides a fallback.

The top domains list can be localized, but Firefox will fallback to using en-US as the default for all locales that do not provide a specific set. The list can have several hundred domains, but due to size concerns, is usually capped to five hundred or less.

Sanitizing Methods

After getting a source list, e.g. Alexa top global sites, we apply some simple guidelines to the list of domains:

- Remove any sites in the Alexa adult site list.
- Remove any locale-specific domain duplicates. We assume primary URLs (.com) will redirect to the correct locale (.co.jp) at run-time.
- Remove any explicit adult content* domains.
- Remove any sites that use explicit or adult advertising*.
- Remove any URL shorteners and redirecters.
- Remove any content/CDN domains. Some sites use separate domains to store images and other static content.
- Remove any sites primarily used for advertising or management of advertising.
- Remove any sites that fail to load in mobile browsers.
- Remove any time/date specific sites that may have appeared on the list due to seasonal spikes.

7.4.2 Suggested Sites

Suggested sites are default thumbnails, displayed on the Top Sites home panel. A suggested site consists of a title, thumbnail image, background color and URL. Multiple images are usually required to handle the variety of device DPIs.

Suggested sites can be localized, but Firefox will fallback to using en-US as the default for all locales that do not provide a specific set. The list is usually small, with perhaps fewer than ten sites.

Sanitizing Methods

After getting a source list, e.g. Alexa top global sites, we apply some simple guidelines to the list of domains:

- Remove pure search engines. We handle search engines differently and don't consider them to be suggested sites.
- Remove any locale-specific domain duplicates. We assume primary URLs (.com) will redirect to the correct locale (.co.jp) at run-time.
- Remove any explicit adult content domains.
- Remove any sites that use explicit or adult advertising.
- Remove any URL shorteners and redirecters.
- Remove any content/CDN domains. Some sites use separate domains to store images and other static content.

7.4.3 Guidelines for Adult Content

Generally the Adult category includes sites whose dominant theme is either:

- To appeal to the prurient interest in sex without any serious literary, artistic, political, or scientific value
- The depiction or description of nudity, including sexual or excretory activities or organs in a lascivious way
- The depiction or description of sexually explicit conduct in a lascivious way (e.g. for entertainment purposes)

For a more complete definition and guidelines of adult content, use the full DMOZ guidelines at http://www.dmoz.org/docs/en/guidelines/adult/general.html.

7.4.4 Updating Lists

After approximately every two releases, Product (with Legal) will review current lists and sanitizing methods, and update the lists accordingly.

7.5 The Firefox for Android install bouncer

Bug 1234629 and Bug 1163082 combine to allow building a very small Fennec-like "bouncer" APK that redirects (bounces) a potential Fennec user to the marketplace of their choice – usually the Google Play Store – to install the real Firefox for Android application APK.

The real APK should install seamlessly over top of the bouncer APK. Care is taken to keep the bouncer and application APK <permission> manifest definitions identical, and to have the bouncer APK <activity> manifest definitions look similar to the application APK <activity> manifest definitions.

In addition, the bouncer APK can carry a Fennec distribution, which it copies onto the device before redirecting to the marketplace. The application APK recognizes the installed distribution and customizes itself accordingly on first run.

The motivation is to allow partners to pre-install the very small bouncer APK on shipping devices and to have a smooth path to upgrade to the full application APK, with a partner-specific distribution in place.

7.5.1 Technical details

To build the bouncer APK, define `MOZ_ANDROID_PACKAGE_INSTALL_BOUNCER`. To pack a distribution into the bouncer APK (and *not* into the application APK), add a line like:

```
ac_add_options --with-android-distribution-directory=/path/to/fennec-distribution-sample
```

to your `mozconfig` file. See the general distribution documentation on the wiki for more information.

The `distribution` directory should end up in the `assets/distribution` directory of the bouncer APK. It will be copied into `/data/data/$ANDROID_PACKAGE_NAME/distribution` when the bouncer executes.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

CHAPTER 9

Localization

9.1 Glossary

L10n *Numeronym* for Localization, *L*, 10 chars, *n*

l10n-merge nick-name for the process of merging `en-US` and a particular localization into one joint artifact without any missing strings, and without technical errors, as far as possible.

L12y Numeronym for Localizability

Localizability Enabling a piece of software to be localized. This is mostly externalizing English strings, and writing build support to pick up localized search engines etc.

Localization The process of creating content in a native language, including translation, but also customizations like Search.

The documentation here is targeted at developers, writing localizable code for Firefox and Firefox for Android, as well as Thunderbird and SeaMonkey.

If you haven't dealt with localization in gecko code before, it's a good idea to check the Glossary for what localization is, and which terms we use for what.

9.2 Exposing strings

Localizers only handle a few file formats in well-known locations in the source tree.

The locations are in directories like

```
browser/locales/en-US/subdir/file.ext
```

The first thing to note is that only files beneath `locales/en-US` are exposed to localizers. The second thing to note is that only a few directories are exposed. Which directories are exposed is defined in files called `l10n.ini`, which are at a few places in the source code.

An example looks like this

```
[general]
depth = ../..

[compare]
dirs = browser
    browser/branding/official
```

```
[includes]
toolkit = toolkit/locales/l10n.ini
```

This tells the l10n infrastructure three things: Resolve the paths against the directory two levels up, include files in `browser/locales/en-US` and `browser/branding/official/locales/en-US`, and load more data from `toolkit/locales/l10n.ini`.

For projects like Thunderbird and SeaMonkey in `comm-central`, additional data needs to be provided when including an `l10n.ini` from a different repository:

```
[include_toolkit]
type = hg
mozilla = mozilla-central
repo = http://hg.mozilla.org/
l10n.ini = toolkit/locales/l10n.ini
```

This tells the l10n pieces where to find the repository, and where inside that repository the `l10n.ini` file is. This is needed because for local builds, `mail/locales/l10n.ini` references `mozilla/toolkit/locales/l10n.ini`, which is where the comm-central build setup expects toolkit to be.

Now that the directories exposed to l10n are known, we can talk about the supported file formats.

9.3 File formats

This is just a quick overview, please check the XUL Tutorial for an in-depth tour.

The following file formats are known to the l10n tool chains:

DTD Used in XUL and XHTML. Also for Android native strings.

Properties Used from JavaScript and C++. When used from js, also comes with plural support.

ini Used by the crashreporter and updater, avoid if possible.

foo.defines Used during builds, for example to create file:*install.rdf* for language packs.

Adding new formats involves changing various different tools, and is strongly discouraged.

9.4 Exceptions

Generally, anything that exists in `en-US` needs a one-to-one mapping in all localizations. There are a few cases where that's not wanted, notably around search settings and spell-checking dictionaries.

To enable tools to adjust to those exceptions, there's a python-coded `filter.py`, implementing `test()`, with the following signature

```
def test(mod, path, entity = None):
    if does_not_matter:
        return "ignore"
    if show_but_do_not_merge:
        return "report"
    # default behavior, localizer or build need to do something
    return "error"
```

For any missing file, this function is called with `mod` being the *module*, and `path` being the relative path inside `locales/en-US`. The module is the top-level dir as referenced in `l10n.ini`.

For missing strings, the `entity` parameter is the key of the string in the en-US file.

9.5 l10n-merge

Gecko doesn't support fallback from a localization to `en-US` at runtime. Thus, the build needs to ensure that the localization as it's built into the package has all required strings, and that the strings don't contain errors. To ensure that, we're *merging* the localization and `en-US` at build time, nick-named *l10n-merge*.

The process is usually triggered via

```
$obj-dir/browser/locales> make merge-de LOCALE_MERGEDIR=$PWD/merge-de
```

It creates another directory in the object dir, `merge-ab-CD`, in which the modified files are stored. The actual repackaging process looks for the localized files in the merge dir first, then the localized file, and then in `en-US`. Thus, for the `de` localization of `browser/locales/en-US/chrome/browser/browser.dtd`, it checks

1. `$objdir/browser/locales/merge-de/browser/chrome/browser/browser.dtd`
2. `$(LOCALE_BASEDIR)/de/browser/chrome/browser/browser.dtd`
3. `browser/locales/en-US/chrome/browser/browser.dtd`

and will include the first of those files it finds.

l10n-merge modifies a file if it supports the particular file type, and there are missing strings which are not filtered out, or if an existing string shows an error. See the Checks section below for details.

9.6 Checks

As part of the build and other localization tool chains, we run a variety of source-based checks. Think of them as linters.

The suite of checks is usually determined by file type, i.e., there's a suite of checks for DTD files and one for properties files, etc. An exception are Android-specific checks.

9.6.1 Android

For Android, we need to localize `strings.xml`. We're doing so via DTD files, which is mostly OK. But the strings inside the XML file have to satisfy additional constraints about quotes etc, that are not part of XML. There's probably some historic background on why things are the way they are.

The Android-specific checks are enabled for DTD files that are in `mobile/android/base/locales/en-US/`.

9.7 Localizations

Now that we talked in-depth about how to expose content to localizers, where are the localizations?

We host a mercurial repository per locale and per branch. Most of our localizations only work starting with aurora, so the bulk of the localizations is found on https://hg.mozilla.org/releases/l10n/mozilla-aurora/. We have several localizations continuously working with mozilla-central, those repositories are on https://hg.mozilla.org/l10n-central/.

You can search inside our localized files on Transvision and http://mxr.mozilla.org/l10n-mozilla-aurora/.

CHAPTER 10

mach

Mach (German for *do*) is a generic command dispatcher for the command line.

To use mach, you install the mach core (a Python package), create an executable *driver* script (named whatever you want), and write mach commands. When the *driver* is executed, mach dispatches to the requested command handler automatically.

10.1 Features

On a high level, mach is similar to using argparse with subparsers (for command handling). When you dig deeper, mach offers a number of additional features:

Distributed command definitions With optparse/argparse, you have to define your commands on a central parser instance. With mach, you annotate your command methods with decorators and mach finds and dispatches to them automatically.

Command categories Mach commands can be grouped into categories when displayed in help. This is currently not possible with argparse.

Logging management Mach provides a facility for logging (both classical text and structured) that is available to any command handler.

Settings files Mach provides a facility for reading settings from an ini-like file format.

10.2 Components

Mach is conceptually composed of the following components:

core The mach core is the core code powering mach. This is a Python package that contains all the business logic that makes mach work. The mach core is common to all mach deployments.

commands These are what mach dispatches to. Commands are simply Python methods registered as command names. The set of commands is unique to the environment mach is deployed in.

driver The *driver* is the entry-point to mach. It is simply an executable script that loads the mach core, tells it where commands can be found, then asks the mach core to handle the current request. The driver is unique to the deployed environment. But, it's usually based on an example from this source tree.

10.3 Project State

mach was originally written as a command dispatching framework to aid Firefox development. While the code is mostly generic, there are still some pieces that closely tie it to Mozilla/Firefox. The goal is for these to eventually be removed and replaced with generic features so mach is suitable for anybody to use. Until then, mach may not be the best fit for you.

10.3.1 Implementing Commands

Mach commands are defined via Python decorators.

All the relevant decorators are defined in the *mach.decorators* module. The important decorators are as follows:

CommandProvider A class decorator that denotes that a class contains mach commands. The decorator takes no arguments.

Command A method decorator that denotes that the method should be called when the specified command is requested. The decorator takes a command name as its first argument and a number of additional arguments to configure the behavior of the command.

CommandArgument A method decorator that defines an argument to the command. Its arguments are essentially proxied to ArgumentParser.add_argument()

SubCommand A method decorator that denotes that the method should be a sub-command to an existing `@Command`. The decorator takes the parent command name as its first argument and the sub-command name as its second argument.

`@CommandArgument` can be used on `@SubCommand` instances just like they can on `@Command` instances.

Classes with the `@CommandProvider` decorator **must** have an `__init__` method that accepts 1 or 2 arguments. If it accepts 2 arguments, the 2nd argument will be a *mach.base.CommandContext* instance.

Here is a complete example:

```
from mach.decorators import (
    CommandArgument,
    CommandProvider,
    Command,
)

@CommandProvider
class MyClass(object):
    @Command('doit', help='Do ALL OF THE THINGS.')
    @CommandArgument('--force', '-f', action='store_true',
        help='Force doing it.')
    def doit(self, force=False):
        # Do stuff here.
```

When the module is loaded, the decorators tell mach about all handlers. When mach runs, it takes the assembled metadata from these handlers and hooks it up to the command line driver. Under the hood, arguments passed to the decorators are being used to help mach parse command arguments, formulate arguments to the methods, etc. See the documentation in the *mach.base* module for more.

The Python modules defining mach commands do not need to live inside the main mach source tree.

Conditionally Filtering Commands

Sometimes it might only make sense to run a command given a certain context. For example, running tests only makes sense if the product they are testing has been built, and said build is available. To make sure a command is only runnable from within a correct context, you can define a series of conditions on the *Command* decorator.

A condition is simply a function that takes an instance of the *mach.decorators.CommandProvider()* class as an argument, and returns `True` or `False`. If any of the conditions defined on a command return `False`, the command will not be runnable. The docstring of a condition function is used in error messages, to explain why the command cannot currently be run.

Here is an example:

```
from mach.decorators import (
    CommandProvider,
    Command,
)

def build_available(cls):
    """The build needs to be available."""
    return cls.build_path is not None

 @CommandProvider
class MyClass(MachCommandBase):
    def __init__(self, build_path=None):
        self.build_path = build_path

    @Command('run_tests', conditions=[build_available])
    def run_tests(self):
        # Do stuff here.
```

It is important to make sure that any state needed by the condition is available to instances of the command provider.

By default all commands without any conditions applied will be runnable, but it is possible to change this behaviour by setting `require_conditions` to `True`:

```
m = mach.main.Mach()
m.require_conditions = True
```

Minimizing Code in Commands

Mach command modules, classes, and methods work best when they are minimal dispatchers. The reason is import bloat. Currently, the mach core needs to import every Python file potentially containing mach commands for every command invocation. If you have dozens of commands or commands in modules that import a lot of Python code, these imports could slow mach down and waste memory.

It is thus recommended that mach modules, classes, and methods do as little work as possible. Ideally the module should only import from the *mach* package. If you need external modules, you should import them from within the command method.

To keep code size small, the body of a command method should be limited to:

1. Obtaining user input (parsing arguments, prompting, etc)
2. Calling into some other Python package
3. Formatting output

Of course, these recommendations can be ignored if you want to risk slower performance.

In the future, the mach driver may cache the dispatching information or have it intelligently loaded to facilitate lazy loading.

10.3.2 Drivers

Entry Points

It is possible to use setuptools' entry points to load commands directly from python packages. A mach entry point is a function which returns a list of files or directories containing mach command providers. e.g.:

```
def list_providers():
    providers = []
    here = os.path.abspath(os.path.dirname(__file__))
    for p in os.listdir(here):
        if p.endswith('.py'):
            providers.append(os.path.join(here, p))
    return providers
```

See http://pythonhosted.org/setuptools/setuptools.html#dynamic-discovery-of-services-and-plugins for more information on creating an entry point. To search for entry point plugins, you can call *mach.main.Mach.load_commands_from_entry_point()*. e.g.:

```
mach.load_commands_from_entry_point("mach.external.providers")
```

Adding Global Arguments

Arguments to mach commands are usually command-specific. However, mach ships with a handful of global arguments that apply to all commands.

It is possible to extend the list of global arguments. In your *mach driver*, simply call *mach.main.Mach.add_global_argument()*. e.g.:

```
mach = mach.main.Mach(os.getcwd())

# Will allow --example to be specified on every mach command.
mach.add_global_argument('--example', action='store_true',
    help='Demonstrate an example global argument.')
```

10.3.3 Logging

Mach configures a built-in logging facility so commands can easily log data.

What sets the logging facility apart from most loggers you've seen is that it encourages structured logging. Instead of conventional logging where simple strings are logged, the internal logging mechanism logs all events with the following pieces of information:

- A string *action*
- A dict of log message fields
- A formatting string

Essentially, instead of assembling a human-readable string at logging-time, you create an object holding all the pieces of data that will constitute your logged event. For each unique type of logged event, you assign an *action* name.

Depending on how logging is configured, your logged event could get written a couple of different ways.

JSON Logging

Where machines are the intended target of the logging data, a JSON logger is configured. The JSON logger assembles an array consisting of the following elements:

- Decimal wall clock time in seconds since UNIX epoch
- String *action* of message
- Object with structured message data

The JSON-serialized array is written to a configured file handle. Consumers of this logging stream can just perform a readline() then feed that into a JSON deserializer to reconstruct the original logged message. They can key off the *action* element to determine how to process individual events. There is no need to invent a parser. Convenient, isn't it?

Logging for Humans

Where humans are the intended consumer of a log message, the structured log message are converted to more human-friendly form. This is done by utilizing the *formatting* string provided at log time. The logger simply calls the *format* method of the formatting string, passing the dict containing the message's fields.

When *mach* is used in a terminal that supports it, the logging facility also supports terminal features such as colorization. This is done automatically in the logging layer - there is no need to control this at logging time.

In addition, messages intended for humans typically prepends every line with the time passed since the application started.

Logging HOWTO

Structured logging piggybacks on top of Python's built-in logging infrastructure provided by the *logging* package. We accomplish this by taking advantage of *logging.Logger.log()*'s *extra* argument. To this argument, we pass a dict with the fields *action* and *params*. These are the string *action* and dict of message fields, respectively. The formatting string is passed as the *msg* argument, like normal.

If you were logging to a logger directly, you would do something like:

```
logger.log(logging.INFO, 'My name is {name}',
    extra={'action': 'my_name', 'params': {'name': 'Gregory'}})
```

The JSON logging would produce something like:

```
[1339985554.306338, "my_name", {"name": "Gregory"}]
```

Human logging would produce something like:

```
0.52 My name is Gregory
```

Since there is a lot of complexity using logger.log directly, it is recommended to go through a wrapping layer that hides part of the complexity for you. The easiest way to do this is by utilizing the LoggingMixin:

```
import logging
from mach.mixin.logging import LoggingMixin

class MyClass(LoggingMixin):
    def foo(self):
        self.log(logging.INFO, 'foo_start', {'bar': True},
            'Foo performed. Bar: {bar}')
```

10.3.4 Settings

Mach can read settings in from a set of configuration files. These configuration files are either named `machrc` or `.machrc` and are specified by the bootstrap script. In mozilla-central, these files can live in `~/.mozbuild` and/or `topsrcdir`.

Settings can be specified anywhere, and used both by mach core or individual commands.

Core Settings

These settings are implemented by mach core.

- alias - Create a command alias. This is useful if you want to alias a command to something else, optionally including some defaults. It can either be used to create an entire new command, or provide defaults for an existing one. For example:

```
[alias]
mochitest = mochitest -f browser
browser-test = mochitest -f browser
```

Defining Settings

Settings need to be explicitly defined, along with their type, otherwise mach will throw when trying to access them.

To define settings, use the `SettingsProvider()` decorator in an existing mach command module. E.g:

```
from mach.decorators import SettingsProvider

@SettingsProvider
class ArbitraryClassName(object):
    config_settings = [
        ('foo.bar', 'string'),
        ('foo.baz', 'int', 0, set([0,1,2])),
    ]
```

`@SettingsProvider`'s must specify a variable called `config_settings` that returns a list of tuples. Alternatively, it can specify a function called `config_settings` that returns a list of tuples.

Each tuple is of the form:

```
('<section>.<option>', '<type>', default, extra)
```

`type` is a string and can be one of: string, boolean, int, pos_int, path

`default` is optional, and provides a default value in case none was specified by any of the configuration files.

`extra` is also optional and is a dict containing additional key/value pairs to add to the setting's metadata. The following keys may be specified in the `extra` dict:

- `choices` - A set of allowed values for the setting.

Wildcards

Sometimes a section should allow arbitrarily defined options from the user, such as the `alias` section mentioned above. To define a section like this, use `*` as the option name. For example:

```
('foo.*', 'string')
```

This allows configuration files like this:

```
[foo]
arbitrary1 = some string
arbitrary2 = some other string
```

Documenting Settings

All settings must at least be documented in the en_US locale. Otherwise, running `mach settings` will raise. Mach uses gettext to perform localization.

A handy command exists to generate the localization files:

```
mach settings locale-gen <section>
```

You'll be prompted to add documentation for all options in section with the en_US locale. To add documentation in another locale, pass in `--locale`.

Accessing Settings

Now that the settings are defined and documented, they're accessible from individual mach commands if the command receives a context in its constructor. For example:

```
from mach.decorators import (
    Command,
    CommandProvider,
    SettingsProvider,
)

@SettingsProvider
class ExampleSettings(object):
    config_settings = [
        ('a.b', 'string', 'default'),
        ('foo.bar', 'string'),
        ('foo.baz', 'int', 0, {'choices': set([0,1,2])}),
    ]

@CommandProvider
class Commands(object):
    def __init__(self, context):
        self.settings = context.settings

    @Command('command', category='misc',
             description='Prints a setting')
    def command(self):
        print(self.settings.a.b)
        for option in self.settings.foo:
            print(self.settings.foo[option])
```

CHAPTER 11

CloudSync

CloudSync is a service that provides access to tabs and bookmarks data for third-party sync addons. Addons can read local bookmarks and tabs. Bookmarks and tab data can be merged from remote devices.

Addons are responsible for maintaining an upstream representation, as well as sending and receiving data over the network.

11.1 Architecture

CloudSync offers functionality similar to Firefox Sync for data sources. Third-party addons (sync adapters) consume local data, send and receive updates from the cloud, and merge remote data.

11.1.1 Files

CloudSync.jsm Main module; Includes other modules and exposes them.

CloudSyncAdapters.jsm Provides an API for addons to register themselves. Will be used to list available adapters and to notify adapters when sync operations are requested manually by the user.

CloudSyncBookmarks.jsm Provides operations for interacting with bookmarks.

CloudSyncBookmarksFolderCache.jsm Implements a cache used to store folder hierarchy for filtering bookmark events.

CloudSyncEventSource.jsm Implements an event emitter. Used to provide addEventListener and removeEventListener for tabs and bookmarks.

CloudSyncLocal.jsm Provides information about the local device, such as name and a unique id.

CloudSyncPlacesWrapper.jsm Wraps parts of the Places API in promises. Some methods are implemented to be asynchronous where they are not in the places API.

CloudSyncTabs.jsm Provides operations for fetching local tabs and for populating the about:sync-tabs page.

11.1.2 Data Sources

CloudSync provides data for tabs and bookmarks. For tabs, local open pages can be enumerated and remote tabs can be merged for displaying in about:sync-tabs. For bookmarks, updates are tracked for a named folder (given by each adapter) and handled by callbacks registered using addEventListener, and remote changes can be merged into the local database.

11.1.3 Versioning

The API carries an integer version number (clouySync.version). Data records are versioned separately and individually.

11.2 Data Format

All fields are required unless noted otherwise.

11.2.1 Bookmarks

Record

type: record type; one of CloudSync.bookmarks.{BOOKMARK, FOLDER, SEPARATOR, QUERY, LIVEMARK}

id: GUID for this bookmark item

parent: id of parent folder

index: item index in parent folder; should be unique and contiguous, or they will be adjusted internally

title: bookmark or folder title; not meaningful for separators

dateAdded: timestamp (in milliseconds) for item added

lastModified: timestamp (in milliseconds) for last modification

uri: bookmark URI; not meaningful for folders or separators

version: data layout version

11.2.2 Tabs

ClientRecord

id: GUID for this client

name: name for this client; not guaranteed to be unique

tabs: list of tabs open on this client; see TabRecord

version: data layout version

TabRecord

title: name for this tab

url: URL for this tab; only one tab for each URL is stored

icon: favicon URL for this tab; optional

lastUsed: timetamp (in milliseconds) for last use

version: data layout version

11.3 Example

```
Cu.import("resource://gre/modules/CloudSync.jsm");

let HelloWorld = {
  onLoad: function() {
    let cloudSync = CloudSync();
    console.log("CLOUDSYNC -- hello world", cloudSync.local.id, cloudSync.local.name, cloudSync.adapt
    cloudSync.adapters.register('helloworld', {});
    console.log("CLOUDSYNC -- " + JSON.stringify(cloudSync.adapters.getAdapterNames()));


    cloudSync.tabs.addEventListener("change", function() {
      console.log("tab change");
      cloudSync.tabs.getLocalTabs().then(
        function(records) {
          console.log(JSON.stringify(records));
        }
      );
    });

    cloudSync.tabs.getLocalTabs().then(
      function(records) {
        console.log(JSON.stringify(records));
      }
    );

    let remoteClient = {
      id: "001",
      name: "FakeClient",
    };
    let remoteTabs1 = [
      {url:"https://www.google.ca",title:"Google",icon:"https://www.google.ca/favicon.ico",lastUsed:
    ];
    let remoteTabs2 = [
      {url:"https://www.google.ca",title:"Google Canada",icon:"https://www.google.ca/favicon.ico",las
      {url:"http://www.reddit.com",title:"Reddit",icon:"http://www.reddit.com/favicon.ico",lastUsed:
    ];
    cloudSync.tabs.mergeRemoteTabs(remoteClient, remoteTabs1).then(
      function() {
        return cloudSync.tabs.mergeRemoteTabs(remoteClient, remoteTabs2);
      }
    ).then(
      function() {
        return cloudSync.tabs.getRemoteTabs();
      }
    ).then(
      function(tabs) {
        console.log("remote tabs:", tabs);
      }
    );

    cloudSync.bookmarks.getRootFolder("Hello World").then(
      function(rootFolder) {
        console.log(rootFolder.name, rootFolder.id);
        rootFolder.addEventListener("add", function(guid) {
          console.log("CLOUDSYNC -- bookmark item added: " + guid);
          rootFolder.getLocalItemsById([guid]).then(
```

```
      function(items) {
        console.log("CLOUDSYNC -- items: " + JSON.stringify(items));
      }
    );
  });
  rootFolder.addEventListener("remove", function(guid) {
    console.log("CLOUDSYNC -- bookmark item removed: " + guid);
    rootFolder.getLocalItemsById([guid]).then(
      function(items) {
        console.log("CLOUDSYNC -- items: " + JSON.stringify(items));
      }
    );
  });
  rootFolder.addEventListener("change", function(guid) {
    console.log("CLOUDSYNC -- bookmark item changed: " + guid);
    rootFolder.getLocalItemsById([guid]).then(
      function(items) {
        console.log("CLOUDSYNC -- items: " + JSON.stringify(items));
      }
    );
  });
  rootFolder.addEventListener("move", function(guid) {
    console.log("CLOUDSYNC -- bookmark item moved: " + guid);
    rootFolder.getLocalItemsById([guid]).then(
      function(items) {
        console.log("CLOUDSYNC -- items: " + JSON.stringify(items));
      }
    );
  });

  function logLocalItems() {
    return rootFolder.getLocalItems().then(
      function(items) {
        console.log("CLOUDSYNC -- local items: " + JSON.stringify(items));
      }
    );
  }

  let items = [
      {"id":"9fdoci2KOME6","type":rootFolder.FOLDER,"parent":rootFolder.id,"title":"My Bookmar
      {"id":"1fdoci2KOME5","type":rootFolder.FOLDER,"parent":rootFolder.id,"title":"My Bookmar
      {"id":"G_UL4ZhOyX8m","type":rootFolder.BOOKMARK,"parent":"1fdoci2KOME5","title":"reddit:
    ];
  function mergeSomeItems() {
    return rootFolder.mergeRemoteItems(items);
  }

  logLocalItems().then(
    mergeSomeItems
  ).then(
    function(processedItems) {
      console.log("!!!", processedItems);
      console.log("merge complete");
    },
    function(error) {
      console.log("merge failed:", error);
    }
  ).then(
```

```
          logLocalItems
        );
      }
    );


  },
};

window.addEventListener("load", function(e) { HelloWorld.onLoad(e); }, false);
```

CHAPTER 12

TaskCluster Task-Graph Generation

The `taskcluster` directory contains support for defining the graph of tasks that must be executed to build and test the Gecko tree. This is more complex than you might suppose! This implementation supports:

- A huge array of tasks
- Different behavior for different repositories
- "Try" pushes, with special means to select a subset of the graph for execution
- Optimization – skipping tasks that have already been performed
- Extremely flexible generation of a variety of tasks using an approach of incrementally transforming job descriptions into task definitions.

This section of the documentation describes the process in some detail, referring to the source where necessary. If you are reading this with a particular goal in mind and would rather avoid becoming a task-graph expert, check out the how-to section.

12.1 TaskGraph Mach Command

The task graph is built by linking different kinds of tasks together, pruning out tasks that are not required, then optimizing by replacing subgraphs with links to already-completed tasks.

12.1.1 Concepts

- *Task Kind* - Tasks are grouped by kind, where tasks of the same kind do not have interdependencies but have substantial similarities, and may depend on tasks of other kinds. Kinds are the primary means of supporting diversity, in that a developer can add a new kind to do just about anything without impacting other kinds.
- *Task Attributes* - Tasks have string attributes by which can be used for filtering. Attributes are documented in Task Attributes.
- *Task Labels* - Each task has a unique identifier within the graph that is stable across runs of the graph generation algorithm. Labels are replaced with TaskCluster TaskIds at the latest time possible, facilitating analysis of graphs without distracting noise from randomly-generated taskIds.
- *Optimization* - replacement of a task in a graph with an equivalent, already-completed task, or a null task, avoiding repetition of work.

12.1.2 Kinds

Kinds are the focal point of this system. They provide an interface between the large-scale graph-generation process and the small-scale task-definition needs of different kinds of tasks. Each kind may implement task generation differently. Some kinds may generate task definitions entirely internally (for example, symbol-upload tasks are all alike, and very simple), while other kinds may do little more than parse a directory of YAML files.

A `kind.yml` file contains data about the kind, as well as referring to a Python class implementing the kind in its `implementation` key. That implementation may rely on lots of code shared with other kinds, or contain a completely unique implementation of some functionality.

The full list of pre-defined keys in this file is:

`implementation` Class implementing this kind, in the form `<module-path>:<object-path>`. This class should be a subclass of `taskgraph.kind.base:Kind`.

`kind-dependencies` Kinds which should be loaded before this one. This is useful when the kind will use the list of already-created tasks to determine which tasks to create, for example adding an upload-symbols task after every build task.

Any other keys are subject to interpretation by the kind implementation.

The result is a nice segmentation of implementation so that the more esoteric in-tree projects can do their crazy stuff in an isolated kind without making the bread-and-butter build and test configuration more complicated.

12.1.3 Dependencies

Dependencies between tasks are represented as labeled edges in the task graph. For example, a test task must depend on the build task creating the artifact it tests, and this dependency edge is named 'build'. The task graph generation process later resolves these dependencies to specific taskIds.

12.1.4 Decision Task

The decision task is the first task created when a new graph begins. It is responsible for creating the rest of the task graph.

The decision task for pushes is defined in-tree, in `.taskcluster.yml`. That task description invokes `mach taskcluster decision` with some metadata about the push. That mach command determines the optimized task graph, then calls the TaskCluster API to create the tasks.

Note that this mach command is *not* designed to be invoked directly by humans. Instead, use the mach commands described below, supplying `parameters.yml` from a recent decision task. These commands allow testing everything the decision task does except the command-line processing and the `queue.createTask` calls.

12.1.5 Graph Generation

Graph generation, as run via `mach taskgraph decision`, proceeds as follows:

1. For all kinds, generate all tasks. The result is the "full task set"
2. Create links between tasks using kind-specific mechanisms. The result is the "full task graph".
3. Select the target tasks (based on try syntax or a tree-specific specification). The result is the "target task set".
4. Based on the full task graph, calculate the transitive closure of the target task set. That is, the target tasks and all requirements of those tasks. The result is the "target task graph".

5. Optimize the target task graph based on kind-specific optimization methods. The result is the "optimized task graph" with fewer nodes than the target task graph.
6. Create tasks for all tasks in the optimized task graph.

12.1.6 Optimization

The objective of optimization to remove as many tasks from the graph as possible, as efficiently as possible, thereby delivering useful results as quickly as possible. For example, ideally if only a test script is modified in a push, then the resulting graph contains only the corresponding test suite task.

A task is said to be "optimized" when it is either replaced with an equivalent, already-existing task, or dropped from the graph entirely.

A task can be optimized if all of its dependencies can be optimized and none of its inputs have changed. For a task on which no other tasks depend (a "leaf task"), the optimizer can determine what has changed by looking at the version-control history of the push: if the relevant files are not modified in the push, then it considers the inputs unchanged. For tasks on which other tasks depend ("non-leaf tasks"), the optimizer must replace the task with another, equivalent task, so it generates a hash of all of the inputs and uses that to search for a matching, existing task.

In some cases, such as try pushes, tasks in the target task set have been explicitly requested and are thus excluded from optimization. In other cases, the target task set is almost the entire task graph, so targetted tasks are considered for optimization. This behavior is controlled with the `optimize_target_tasks` parameter.

12.1.7 Action Tasks

Action Tasks are tasks which help you to schedule new jobs via Treeherder's "Add New Jobs" feature. The Decision Task creates a YAML file named `action.yml` which can be used to schedule Action Tasks after suitably replacing `{{decision_task_id}}` and `{{task_labels}}`, which correspond to the decision task ID of the push and a comma separated list of task labels which need to be scheduled.

This task invokes `mach taskgraph action-task` which builds up a task graph of the requested tasks. This graph is optimized using the tasks running initially in the same push, due to the decision task.

So for instance, if you had already requested a build task in the `try` command, and you wish to add a test which depends on this build, the original build task is re-used.

This feature is only present on `try` pushes for now.

12.1.8 Mach commands

A number of mach subcommands are available aside from `mach taskgraph decision` to make this complex system more accesssible to those trying to understand or modify it. They allow you to run portions of the graph-generation process and output the results.

`mach taskgraph tasks` Get the full task set

`mach taskgraph full` Get the full task graph

`mach taskgraph target` Get the target task set

`mach taskgraph target-graph` Get the target task graph

`mach taskgraph optimized` Get the optimized task graph

Each of these commands taskes a `--parameters` option giving a file with parameters to guide the graph generation. The decision task helpfully produces such a file on every run, and that is generally the easiest way to get a parameter

file. The parameter keys and values are described in Parameters; using that information, you may modify an existing `parameters.yml` or create your own.

12.1.9 Task Parameterization

A few components of tasks are only known at the very end of the decision task – just before the `queue.createTask` call is made. These are specified using simple parameterized values, as follows:

`{"relative-datestamp": "certain number of seconds/hours/days/years"}` Objects of this form will be replaced with an offset from the current time just before the `queue.createTask` call is made. For example, an artifact expiration might be specified as `{"relative-timestamp": "1 year"}`.

`{"task-reference": "string containing <dep-name>"}` The task definition may contain "task references" of this form. These will be replaced during the optimization step, with the appropriate taskId for the named dependency substituted for `<dep-name>` in the string. Multiple labels may be substituted in a single string, and `<>` can be used to escape a literal `<`.

The `mach taskgraph action-task` subcommand is used by Action Tasks to create a task graph of the requested jobs and its non-optimized dependencies. Action Tasks are currently scheduled by [pulse_actions](https://github.com/mozilla/pulse_actions)

12.1.10 Taskgraph JSON Format

Task graphs – both the graph artifacts produced by the decision task and those output by the `--json` option to the `mach taskgraph` commands – are JSON objects, keyed by label, or for optimized task graphs, by taskId. For convenience, the decision task also writes out `label-to-taskid.json` containing a mapping from label to taskId. Each task in the graph is represented as a JSON object.

Each task has the following properties:

`task_id` The task's taskId (only for optimized task graphs)

`label` The task's label

`attributes` The task's attributes

`dependencies` The task's in-graph dependencies, represented as an object mapping dependency name to label (or to taskId for optimized task graphs)

`task` The task's TaskCluster task definition.

`kind_implementation` The module and the class name which was used to implement this particular task. It is always of the form `<module-path>:<object-path>`

The results from each command are in the same format, but with some differences in the content:

- The `tasks` and `target` subcommands both return graphs with no edges. That is, just collections of tasks without any dependencies indicated.
- The `optimized` subcommand returns tasks that have been assigned taskIds. The dependencies array, too, contains taskIds instead of labels, with dependencies on optimized tasks omitted. However, the `task.dependencies` array is populated with the full list of dependency taskIds. All task references are resolved in the optimized graph.

The output of the `mach taskgraph` commands are suitable for processing with the jq utility. For example, to extract all tasks' labels and their dependencies:

```
jq 'to_entries | map({label: .value.label, dependencies: .value.dependencies})'
```

12.2 Parameters

Task-graph generation takes a collection of parameters as input, in the form of a JSON or YAML file.

During decision-task processing, some of these parameters are supplied on the command line or by environment variables. The decision task helpfully produces a full parameters file as one of its output artifacts. The other `mach taskgraph` commands can take this file as input. This can be very helpful when working on a change to the task graph.

The properties of the parameters object are described here, divided rougly by topic.

12.2.1 Push Information

`base_repository` The repository from which to do an initial clone, utilizing any available caching.

`head_repository` The repository containing the changeset to be built. This may differ from `base_repository` in cases where `base_repository` is likely to be cached and only a few additional commits are needed from `head_repository`.

`head_rev` The revision to check out; this can be a short revision string

`head_ref` For Mercurial repositories, this is the same as `head_rev`. For git repositories, which do not allow pulling explicit revisions, this gives the symbolic ref containing `head_rev` that should be pulled from `head_repository`.

`owner` Email address indicating the person who made the push. Note that this value may be forged and *must not* be relied on for authentication.

`message` The commit message

`pushlog_id` The ID from the `hg.mozilla.org` pushlog

12.2.2 Tree Information

`project` Another name for what may otherwise be called tree or branch or repository. This is the unqualified name, such as `mozilla-central` or `cedar`.

`level` The SCM level associated with this tree. This dictates the names of resources used in the generated tasks, and those tasks will fail if it is incorrect.

12.2.3 Target Set

The "target set" is the set of task labels which must be included in a task graph. The task graph generation process will include any tasks required by those in the target set, recursively. In a decision task, this set can be specified programmatically using one of a variety of methods (e.g., parsing try syntax or reading a project-specific configuration file).

The decision task writes its task set to the `target_tasks.json` artifact, and this can be copied into `parameters.target_tasks` and `parameters.target_tasks_method` set to `"from_parameters"` for debugging with other `mach taskgraph` commands.

target_tasks_method (optional) The method to use to determine the target task set. This is the suffix of one of the functions in `tascluster/taskgraph/target_tasks.py`. If omitted, all tasks are targeted.

target_tasks (optional) The target set method `from_parameters` reads the target set, as a list of task labels, from this parameter.

optimize_target_tasks (optional; default True) If true, then target tasks are eligible for optimization.

12.3 Task Attributes

Tasks can be filtered, for example to support "try" pushes which only perform a subset of the task graph or to link dependent tasks. This filtering is the difference between a full task graph and a target task graph.

Filtering takes place on the basis of attributes. Each task has a dictionary of attributes and filters over those attributes can be expressed in Python. A task may not have a value for every attribute.

The attributes, and acceptable values, are defined here. In general, attribute names and values are the short, lower-case form, with underscores.

12.3.1 kind

A task's `kind` attribute gives the name of the kind that generated it, e.g., `build` or `legacy`.

12.3.2 build_platform

The build platform defines the platform for which the binary was built. It is set for both build and test jobs, although test jobs may have a different `test_platform`.

12.3.3 build_type

The type of build being performed. This is a subdivision of `build_platform`, used for different kinds of builds that target the same platform. Values are

- `debug`
- `opt`

12.3.4 test_platform

The test platform defines the platform on which tests are run. It is only defined for test jobs and may differ from `build_platform` when the same binary is tested on several platforms (for example, on several versions of Windows). This applies for both talos and unit tests.

Unlike build_platform, the test platform is represented in a slash-separated format, e.g., `linux64/opt`.

12.3.5 unittest_suite

This is the unit test suite being run in a unit test task. For example, `mochitest` or `cppunittest`.

12.3.6 unittest_flavor

If a unittest suite has subdivisions, those are represented as flavors. Not all suites have flavors, in which case this attribute should be set to match the suite. Examples: `mochitest-devtools-chrome-chunked` or `a11y`.

12.3.7 unittest_try_name

(deprecated) This is the name used to refer to a unit test via try syntax. It may not match either of `unittest_suite` or `unittest_flavor`.

12.3.8 talos_try_name

(deprecated) This is the name used to refer to a talos job via try syntax.

12.3.9 test_chunk

This is the chunk number of a chunked test suite (talos or unittest). Note that this is a string!

12.3.10 e10s

For test suites which distinguish whether they run with or without e10s, this boolean value identifies this particular run.

12.3.11 legacy_kind

(deprecated) The kind of task as created by the legacy kind. This is valid only for the `legacy` kind. One of `build`, `unittest`,, `talos`, `post_build`, or `job`.

12.3.12 job

(deprecated) The name of the job (corresponding to a `-j` option or the name of a post-build job). This is valid only for the `legacy` kind.

12.3.13 post_build

(deprecated) The name of the post-build activity. This is valid only for the `legacy` kind.

12.3.14 image_name

For the `docker_image` kind, this attribute contains the docker image name.

12.4 Task Kinds

This section lists and documents the available task kinds.

12.4.1 Builds

Builds are currently implemented by the `legacy` kind.

12.4.2 Tests

Test tasks for Gecko products are divided into several kinds, but share a common implementation. The process goes like this, based on a set of YAML files named in `kind.yml`:

- For each build task, determine the related test platforms based on the build platform. For example, a Windows 2010 build might be tested on Windows 7 and Windows 10. Each test platform specifies a "test set" indicating which tests to run. This is configured in the file named `test-platforms.yml`.
- Each test set is expanded to a list of tests to run. This is configured in the file named by `test-sets.yml`.
- Each named test is looked up in the file named by `tests.yml` to find a test description. This test description indicates what the test does, how it is reported to treeherder, and how to perform the test, all in a platform-independent fashion.
- Each test description is converted into one or more tasks. This is performed by a sequence of transforms defined in the `transforms` key in `kind.yml`. See Transforms: for more information on these transforms.
- The resulting tasks become a part of the task graph.

Important: This process generates *all* test jobs, regardless of tree or try syntax. It is up to a later stage of the task-graph generation (the target set) to select the tests that will actually be performed.

desktop-test

The `desktop-test` kind defines tests for Desktop builds. Its `tests.yml` defines the full suite of desktop tests and their particulars, leaving it to the transforms to determine how those particulars apply to Linux, OS X, and Windows.

android-test

The `android-test` kind defines tests for Android builds.

It is very similar to `desktop-test`, but the details of running the tests differ substantially, so they are defined separately.

12.4.3 legacy

The legacy kind is the old, templated-yaml-based task definition mechanism. It is still used for builds and generic tasks, but not for long!

12.4.4 docker-image

Tasks of the `docker-image` kind build the Docker images in which other Docker tasks run.

The tasks to generate each docker image have predictable labels: `build-docker-image-<name>`.

Docker images are built from subdirectories of `testing/docker`, using `docker build`. There is currently no capability for one Docker image to depend on another in-tree docker image, without uploading the latter to a Docker repository

The task definition used to create the image-building tasks is given in `image.yml` in the kind directory, and is interpreted as a YAML Template.

12.5 Transforms

Many task kinds generate tasks by a process of transforming job descriptions into task definitions. The basic operation is simple, although the sequence of transforms applied for a particular kind may not be!

12.5.1 Overview

To begin, a kind implementation generates a collection of items. For example, the test kind implementation generates a list of tests to run for each matching build, representing each as a test description. The items are simply Python dictionaries.

The kind also defines a sequence of transformations. These are applied, in order, to each item. Early transforms might apply default values or break items up into smaller items (for example, chunking a test suite). Later transforms rewrite the items entirely, with the final result being a task definition.

Each transformation looks like this:

The `config` argument is a Python object containing useful configuration for the kind, and is a subclass of `taskgraph.transforms.base.TransformConfig`, which specifies a few of its attributes. Kinds may subclass and add additional attributes if necessary.

While most transforms yield one item for each item consumed, this is not always true: items that are not yielded are effectively filtered out. Yielding multiple items for each consumed item implements item duplication; this is how test chunking is accomplished, for example.

The `transforms` object is an instance of `taskgraph.transforms.base.TransformSequence`, which serves as a simple mechanism to combine a sequence of transforms into one.

12.5.2 Schemas

The items used in transforms are validated against some simple schemas at various points in the transformation process. These schemas accomplish two things: they provide a place to add comments about the meaning of each field, and they enforce that the fields are actually used in the documented fashion.

12.5.3 Keyed By

Several fields in the input items can be "keyed by" another value in the item. For example, a test description's chunks may be keyed by `test-platform`. In the item, this looks like:

```
chunks:
    by-test-platform:
        linux64/debug: 12
        linux64/opt: 8
        default: 10
```

This is a simple but powerful way to encode business rules in the items provided as input to the transforms, rather than expressing those rules in the transforms themselves. If you are implementing a new business rule, prefer this mode where possible. The structure is easily resolved to a single value using `taskgraph.transform.base.get_keyed_by()`.

12.5.4 Task-Generation Transforms

Every kind needs to create tasks, and all of those tasks have some things in common. They all run on one of a small set of worker implementations, each with their own idiosyncracies. And they all report to TreeHerder in a similar way.

The transforms in `taskcluster/taskgraph/transforms/make_task.py` implement this common functionality. They expect a "task description", and produce a task definition. The schema for a task description is defined at the top of `make_task.py`, with copious comments. The result is a dictionary with keys `label`, `attributes`, `task`, and `dependencies`, with the latter having the same format as the input dependencies.

These transforms assign names to treeherder groups using an internal list of group names. Feel free to add additional groups to this list as necessary.

12.5.5 Test Transforms

The transforms configured for test kinds proceed as follows, based on configuration in `kind.yml`:

- The test description is validated to conform to the schema in `taskcluster/taskgraph/transforms/tests/test_des` This schema is extensively documented and is a the primary reference for anyone modifying tests.
- Kind-specific transformations are applied. These may apply default settings, split tests (e.g., one to run with feature X enabled, one with it disabled), or apply across-the-board business rules such as "all desktop debug test platforms should have a max-run-time of 5400s".
- Transformations generic to all tests are applied. These apply policies which apply to multiple kinds, e.g., for treeherder tiers. This is also the place where most values which differ based on platform are resolved, and where chunked tests are split out into a test per chunk.
- The test is again validated against the same schema. At this point it is still a test description, just with defaults and policies applied, and per-platform options resolved. So transforms up to this point do not modify the "shape" of the test description, and are still governed by the schema in `test_description.py`.
- The `taskgraph.transforms.tests.make_task_description:transforms` then take the test description and create a *task* description. This transform embodies the specifics of how test runs work: invoking mozharness, various worker options, and so on.
- Finally, the `taskgraph.transforms.make_task:transforms`, described above under "Task-Generation Transforms", are applied.

Test dependencies are produced in the form of a dictionary mapping dependency name to task label.

12.6 Task Definition YAML Templates

Many kinds of tasks are described using templated YAML files. These files allow some limited forms of inheritance and template substitution as well as the usual YAML features, as described below.

Please use these features sparingly. In many cases, it is better to add a feature to the implementation of a task kind rather than add complexity to the YAML files.

12.6.1 Inheritance

One YAML file can "inherit" from another by including a top-level `$inherits` key. That key specifies the parent file in `from`, and optionally a collection of variables in `variables`. For example:

```
$inherits:
  from: 'tasks/builds/base_linux32.yml'
  variables:
    build_name: 'linux32'
    build_type: 'dbg'
```

Inheritance proceeds as follows: First, the child document has its template substitutions performed and is parsed as YAML. Then, the parent document is parsed, with substitutions specified by `variables` added to the template substitutions. Finally, the child document is merged with the parent.

To merge two JSON objects (dictionaries), each value is merged individually. Lists are merged by concatenating the lists from the parent and child documents. Atomic values (strings, numbers, etc.) are merged by preferring the child document's value.

12.6.2 Substitution

Each document is expanded using the PyStache template engine before it is parsed as YAML. The parameters for this expansion are specific to the task kind.

Simple value substitution looks like `{{variable}}`. Function calls look like `{{#function}}argument{{/function}}`.

12.7 How Tos

All of this equipment is here to help you get your work done more efficiently. However, learning how task-graphs are generated is probably not the work you are interested in doing. This section should help you accomplish some of the more common changes to the task graph with minimal fuss.

Important: If you cannot accomplish what you need with the information provided here, please consider whether you can achieve your goal in a different way. Perhaps something simpler would cost a bit more in compute time, but save the much more expensive resource of developers' mental bandwidth. Task-graph generation is already complex enough!

If you want to proceed, you may need to delve into the implementation of task-graph generation. The documentation and code are designed to help, as are the authors - `hg blame` may help track down helpful people.

As you write your new transform or add a new kind, please consider the next developer. Where possible, make your change data-driven and general, so that others can make a much smaller change. Document the semantics of what you are changing clearly, especially if it involves modifying a transform schema. And if you are adding complexity temporarily while making a gradual transition, please open a new bug to remind yourself to remove the complexity when the transition is complete.

12.7.1 Hacking Task Graphs

The recommended process for changing task graphs is this:

1. Find a recent decision task on the project or branch you are working on, and download its `parameters.yml` from the Task Inspector. This file contains all of the inputs to the task-graph generation process. Its contents are simple enough if you would like to modify it, and it is documented in Parameters.
2. Run one of the `mach taskgraph` subcommands (see TaskGraph Mach Command) to generate a baseline against which to measure your changes. For example:

```
./mach taskgraph --json -p parameters.yml tasks > old-tasks.json
```

3. Make your modifications under `tsakcluster/`.
4. Run the same `mach taskgraph` command, sending the output to a new file, and use `diff` to compare the old and new files. Make sure your changes have the desired effect and no undesirable side-effects.
5. When you are satisfied with the changes, push them to try to ensure that the modified tasks work as expected.

12.7.2 Common Changes

Changing Test Characteristics

First, find the test description. This will be in `taskcluster/ci/*/tests.yml`, for the appropriate kind (consult Task Kinds). You will find a YAML stanza for each test suite, and each stanza defines the test's characteristics. For example, the `chunks` property gives the number of chunks to run. This can be specified as a simple integer if all platforms have the same chunk count, or it can be keyed by test platform. For example:

```
chunks:
    by-test-platform:
        linux64/debug: 10
        default: 8
```

The full set of available properties is in `taskcluster/taskgraph/transform/tests/test_description.py`. Some other commonly-modified properties are `max-run-time` (useful if tests are being killed for exceeding maxRunTime) and `treeherder-symbol`.

Note: Android tests are also chunked at the mozharness level, so you will need to modify the relevant mozharness config, as well.

Adding a Test Suite

To add a new test suite, you will need to know the proper mozharness invocation for that suite, and which kind it fits into (consult Task Kinds).

Add a new stanza to `taskcluster/ci/<kind>/tests.yml`, copying from the other stanzas in that file. The meanings should be clear, but authoritative documentation is in `taskcluster/taskgraph/transform/tests/test_description.py` should you need it. The stanza name is the name by which the test will be referenced in try syntax.

Add your new test to a test set in `test-sets.yml` in the same directory. If the test should only run on a limited set of platforms, you may need to define a new test set and reference that from the appropriate platforms in `test-platforms.yml`. If you do so, include some helpful comments in `test-sets.yml` for the next person.

Greening Up a New Test

When a test is not yet reliably green, configuration for that test should not be landed on integration branches. Of course, you can control where the configuration is landed! For many cases, it is easiest to green up a test in try: push the configuration to run the test to try along with your work to fix the remaining test failures.

When working with a group, check out a "twig" repository to share among your group, and land the test configuration in that repository. Once the test is green, merge to an integration branch and the test will begin running there as well.

Something Else?

If you make another change not described here that turns out to be simple or common, please include an update to this file in your patch.

12.8 Docker Images

TaskCluster Docker images are defined in the source directory under `testing/docker`. Each directory therein contains the name of an image used as part of the task graph.

12.8.1 Adding Extra Files to Images

Dockerfile syntax has been extended to allow *any* file from the source checkout to be added to the image build *context*. (Traditionally you can only `ADD` files from the same directory as the Dockerfile.)

Simply add the following syntax as a comment in a Dockerfile:

```
# %include <path>
```

e.g.

    # %include mach # %include testing/mozharness

The argument to `# %include` is a relative path from the root level of the source directory. It can be a file or a directory. If a file, only that file will be added. If a directory, every file under that directory will be added (even files that are untracked or ignored by version control).

Files added using `# %include` syntax are available inside the build context under the `topsrcdir/` path.

Files are added as they exist on disk. e.g. executable flags should be preserved. However, the file owner/group is changed to `root` and the `mtime` of the file is normalized.

Here is an example Dockerfile snippet:

```
# %include mach
ADD topsrcdir/mach /home/worker/mach
```

CHAPTER 13

Crash Manager

The **Crash Manager** is a service and interface for managing crash data within the Gecko application.

From JavaScript, the service can be accessed via:

```
Cu.import("resource://gre/modules/Services.jsm");
let crashManager = Services.crashmanager;
```

That will give you an instance of `CrashManager` from `CrashManager.jsm`. From there, you can access and manipulate crash data.

13.1 Other Documents

13.1.1 Crash Events

Crash Events refers to a special subsystem of Gecko that aims to capture events of interest related to process crashing and hanging.

When an event worthy of recording occurs, a file containing that event's information is written to a well-defined location on the filesystem. The Gecko process periodically scans for produced files and consolidates information into a more unified and efficient backend store.

Crash Event Files

When a crash-related event occurs, a file describing that event is written to a well-defined directory. That directory is likely in the directory of the currently-active profile. However, if a profile is not yet active in the Gecko process, that directory likely resides in the user's *app data* directory (*UAppData* from the directory service).

The filename of the event file is not relevant. However, producers need to choose a filename intelligently to avoid name collisions and race conditions. Since file locking is potentially dangerous at crash time, the convention of generating a UUID and using it as a filename has been adopted.

File Format

All crash event files share the same high-level file format. The format consists of the following fields delimited by a UNIX newline (*n*) character:

- String event name (valid UTF-8, but likely ASCII)
- String representation of integer seconds since UNIX epoch

- Payload

The payload is event specific and may contain UNIX newline characters. The recommended method for parsing is to split at most 3 times on UNIX newline and then dispatch to an event-specific parsed based on the event name.

If an unknown event type is encountered, the event can safely be ignored until later. This helps ensure that application downgrades (potentially due to elevated crash rate) don't result in data loss.

The format and semantics of each event type are meant to be constant once that event type is committed to the main Firefox repository. If new metadata needs to be captured or the meaning of data captured in an event changes, that change should be expressed through the invention of a new event type. For this reason, event names are highly recommended to contain a version. e.g. instead of a *Gecko process crashed* event, we prefer a *Gecko process crashed v1* event.

Event Types

Each subsection documents the different types of crash events that may be produced. Each section name corresponds to the first line of the crash event file.

Currently only main process crashes produce event files. Because crashes and hangs in child processes can be easily recorded by the main process, we do not foresee the need for writing event files for child processes, design considerations below notwithstanding.

crash.main.2 This event is produced when the main process crashes.

The payload of this event is delimited by UNIX newlines (*n*) and contains the following fields:

- The crash ID string, very likely a UUID
- 0 or more lines of metadata, each containing one key=value pair of text

crash.main.1 This event is produced when the main process crashes.

The payload of this event is the string crash ID, very likely a UUID. There should be `UUID.dmp` and `UUID.extra` files on disk, saved by Breakpad.

crash.submission.1 This event is produced when a crash is submitted.

The payload of this event is delimited by UNIX newlines (*n*) and contains the following fields:

- The crash ID string
- "true" if the submission succeeded or "false" otherwise
- The remote crash ID string if the submission succeeded

Aggregated Event Log

Crash events are aggregated together into a unified event *log*. Currently, this *log* is really a JSON file. However, this is an implementation detail and it could change at any time. The interface to crash data provided by the JavaScript API is the only supported interface.

Design Considerations

There are many considerations influencing the design of this subsystem. We attempt to document them in this section.

Decoupling of Event Files from Final Data Structure

While it is certainly possible for the Gecko process to write directly to the final data structure on disk, there is an intentional decoupling between the production of events and their transition into final storage. Along the same vein, the choice to have events written to multiple files by producers is deliberate.

Some recorded events are written immediately after a process crash. This is a very uncertain time for the host system. There is a high liklihood the system is in an exceptional state, such as memory exhaustion. Therefore, any action taken after crashing needs to be very deliberate about what it does. Excessive memory allocation and certain system calls may cause the system to crash again or the machine's condition to worsen. This means that the act of recording a crash event must be very light weight. Writing a new file from nothing is very light weight. This is one reason we write separate files.

Another reason we write separate files is because if the main Gecko process itself crashes (as opposed to say a plugin process), the crash reporter (not Gecko) is running and the crash reporter needs to handle the writing of the event info. If this writing is involved (say loading, parsing, updating, and reserializing back to disk), this logic would need to be implemented in both Gecko and the crash reporter or would need to be implemented in such a way that both could use. Neither of these is very practical from a software lifecycle management perspective. It's much easier to have separate processes write a simple file and to let a single implementation do all the complex work.

Idempotent Event Processing

Processing of event files has been designed such that the result is idempotent regardless of what order those files are processed in. This is not only a good design decision, but it is arguably necessary. While event files are processed in order by file mtime, filesystem times may not have the resolution required for proper sorting. Therefore, processing order is merely an optimistic assumption.

Aggregated Storage Format

Crash events are aggregated into a unified data structure on disk. That data structure is currently LZ4-compressed JSON and is represented by a single file.

The choice of a single JSON file was initially driven by time and complexity concerns. Before changing the format or adding significant amounts of new data, some considerations must be taken into account.

First, in well-behaving installs, crash data should be minimal. Crashes and hangs will be rare and thus the size of the crash data should remain small over time.

The choice of a single JSON file has larger implications as the amount of crash data grows. As new data is accumulated, we need to read and write an entire file to make small updates. LZ4 compression helps reduce I/O. But, there is a potential for unbounded file growth. We establish a limit for the max age of records. Anything older than that limit is pruned. We also establish a daily limit on the number of crashes we will store. All crashes beyond the first N in a day have no payload and are only recorded by the presence of a count. This count ensures we can distinguish between `N` and `100 * N`, which are very different values!

CHAPTER 14

Telemetry

Telemetry is a feature that allows data collection. This is being used to collect performance metrics and other information about how Firefox performs in the wild.

Client-side, this consists of:

- data collection in Histograms, Scalars and other data structures
- assembling Telemetry pings with the general information and the data payload
- sending them to the server and local ping retention

Note: the data collection policy documents the process and requirements that are applied here.

14.1 Concepts

There are common concepts used throughout Telemetry:

- pings - the packets we use to submit data
- sessions & subsessions - how we slice a users time in the browser
- *measurements* - how we collect data
- *opt-in* & *opt-out* - the different sets of data we collect
- submission - how we send data to the servers
- archiving - retaining ping data locally
- crashes - the different data crashes generate

14.1.1 Telemetry pings

A *Telemetry ping* is the data that we send to Mozillas Telemetry servers.

That data is stored as a JSON object client-side and contains common information to all pings and a payload specific to a certain *ping types*.

The top-level structure is defined by the common ping format format. It contains:

- some basic information shared between different ping types
- the environment data (optional)
- the data specific to the *ping type*, the *payload*.

Ping types

We send Telemetry with different ping types. The main ping is the ping that contains the bulk of the Telemetry measurements for Firefox. For more specific use-cases, we send other ping types.

Pings sent from code that ships with Firefox are listed in the data documentation.

Important examples are:

- main - contains the information collected by Telemetry (Histograms, hang stacks, ...)
- saved-session - has the same format as a main ping, but it contains the *"classic"* Telemetry payload with measurements covering the whole browser session. This is only a separate type to make storage of saved-session easier server-side. This is temporary and will be removed soon.
- crash - a ping that is captured and sent after Firefox crashed.
- `activation` - *planned* - sent right after installation or profile creation
- `upgrade` - *planned* - sent right after an upgrade
- deletion - sent when FHR upload is disabled, requesting deletion of the data associated with this user

14.1.2 Crashes

There are many different kinds of crashes for Firefox, there is not a single system used to record all of them.

Main process crashes

If the Firefox main process dies, that should be recorded as an aborted session. We would submit a main ping with the reason `aborted-session`. If we have a crash dump for that crash, we should also submit a crash ping.

The `aborted-session` information is first written to disk 60 seconds after startup, any earlier crashes will not trigger an `aborted-session` ping. Also, the `aborted-session` is updated at least every 5 minutes, so it may lag behind the last session state.

Crashes during startup should be recorded in the next sessions main ping in the `STARTUP_CRASH_DETECTED` histogram.

Child process crashes

If a Firefox plugin, content or gmplugin process dies unexpectedly, this is recorded in the main pings `SUBPROCESS_ABNORMAL_ABORT` keyed histogram.

If we catch a crash report for this, then additionally the `SUBPROCESS_CRASHES_WITH_DUMP` keyed histogram is incremented.

14.1.3 Archiving

When archiving is enabled through the relative preference, pings submitted to `TelemetryController` are also stored locally in the user profile directory, in `<profile-dir>/datareporting/archived`.

To allow for cheaper lookup of archived pings, storage follows a specific naming scheme for both the directory and the ping file name: *<YYYY-MM>/<timestamp>.<UUID>.<type>.json*.

- `<YYYY-MM>` - The subdirectory name, generated from the ping creation date.
- `<timestamp>` - Timestamp of the ping creation date.

- `<UUID>` - The ping identifier.
- `<type>` - The ping type.

14.1.4 Sessions

A *session* is the time from when Firefox starts until it shut down. A session can be very long-running. E.g. for Mac users that are used to always put their laptops into sleep-mode, Firefox may run for weeks. We slice the sessions into smaller logical units called *subsessions*.

Subsessions

A subsessions data consists of:

- general information: the date the subsession started, how long it lasted, etc.
- specific measurements: histogram & scalar data, etc.

This has some advantages:

- Latency - Sending a ping with all the data of a subsession immediately after it ends means we get the data from installs faster. For `main` pings, we aim to send a ping at least daily by starting a new subsession at local midnight.
- Correlation - By starting new subsessions when fundamental settings change (i.e. changes to the *environment*), we can correlate a subsessions data better to those settings.

Subsession splits

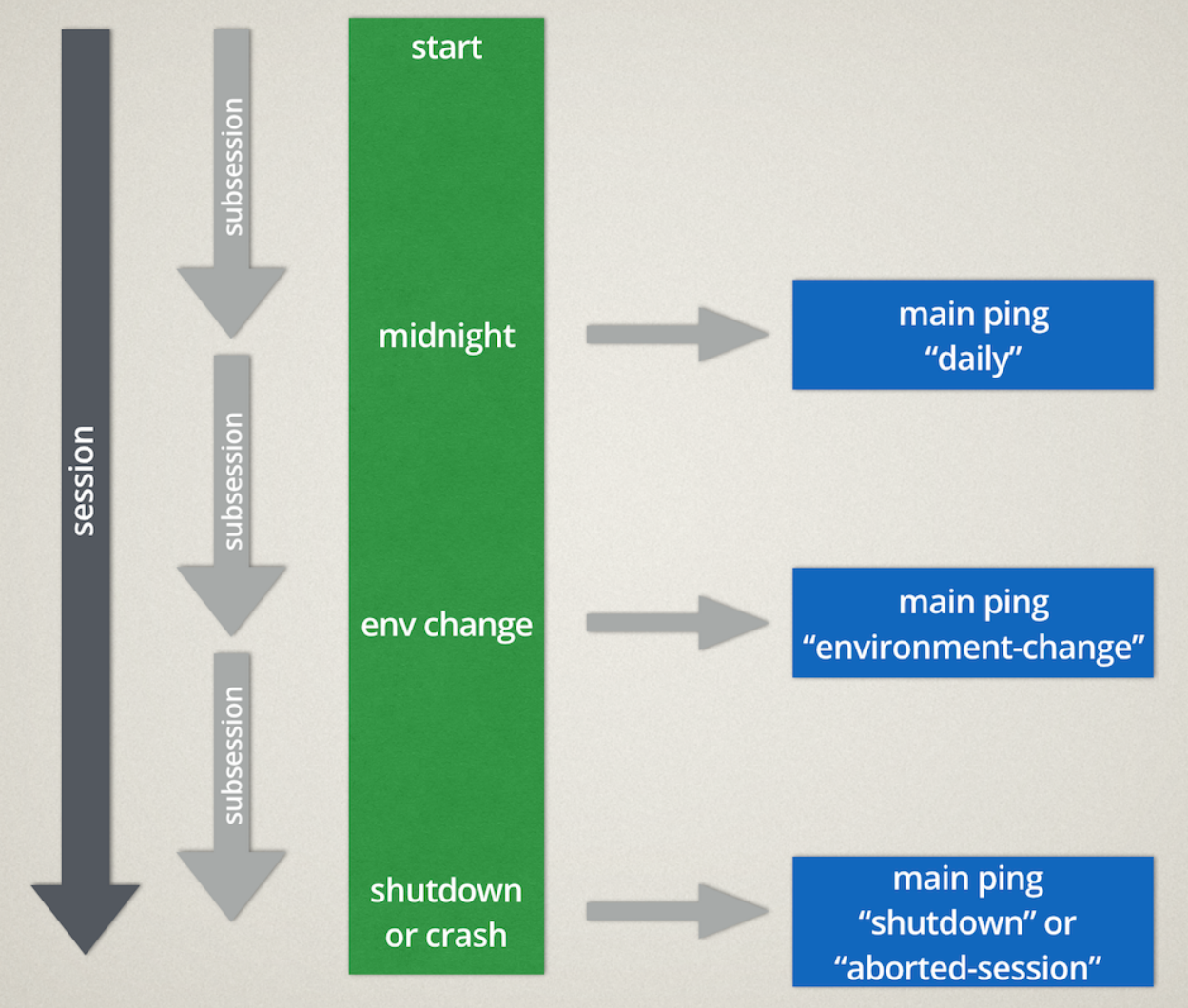
The first subsession starts when the browser starts. After that, we split the subsession for different reasons:

- `daily`, when crossing local midnight. This keeps latency acceptable by triggering a ping at least daily for most active users.
- `environment-change`, when a change to the *environment* happens. This happens for important changes to the Firefox settings and when addons activate or deactivate.

On a subsession split, a main ping with that reason will be submitted. We store the reason in the pings payload, to see what triggered it.

A session always ends with a subsession with one of two reason:

- `shutdown`, when the browser was cleanly shut down. To avoid delaying shutdown, we only save this ping to disk and send it at the next opportunity (typically the next browsing session).
- `aborted-session`, when the browser crashed. While Firefox is active, we write the current `main` ping data to disk every 5 minutes. If the browser crashes, we find this data on disk on the next start and send it with this reason.



14.1.5 Submission

Note: The server-side behaviour is documented in the HTTP Edge Server specification.

Pings are submitted via a common API on `TelemetryController`. If a ping fails to successfully submit to the server immediately (e.g. because of missing internet connection), Telemetry will store it on disk and retry to send it until the maximum ping age is exceeded (14 days).

Note: the main pings are kept locally even after successful submission to enable the HealthReport and SelfSupport features. They will be deleted after their retention period of 180 days.

Submission logic

Sending of pending pings starts as soon as the delayed startup is finished. They are sent in batches, newest-first, with up to 10 persisted pings per batch plus all unpersisted pings. The send logic then waits for each batch to complete.

If it succeeds we trigger the next send of a ping batch. This is delayed as needed to only trigger one batch send per minute.

If ping sending encounters an error that means retrying later, a backoff timeout behavior is triggered, exponentially increasing the timeout for the next try from 1 minute up to a limit of 120 minutes. Any new ping submissions and "idle-daily" events reset this behavior as a safety mechanism and trigger immediate ping sending.

Status codes

The telemetry server team is working towards the common services status codes, but for now the following logic is sufficient for Telemetry:

- *2XX* - success, don't resubmit
- *4XX* - there was some problem with the request - the client should not try to resubmit as it would just receive the same response
- *5XX* - there was a server-side error, the client should try to resubmit later

14.2 Data collection

There are different APIs and formats to collect data in Firefox, all suiting different use cases.

In general, we aim to submit data in a common format where possible. This has several advantages; from common code and tooling to sharing analysis know-how.

In cases where this isn't possible and more flexibility is needed, we can submit custom pings or consider adding different data formats to existing pings.

Note: Every new data collection must go through a data collection review.

The current data collection possibilities include:

- Scalars allow recording of a single value (string, boolean, a number)
- Histograms can efficiently record multiple data points
- `environment` data records information about the system and settings a session occurs in
- `TelemetryLog` allows collecting ordered event entries
- measuring elapsed time
- custom pings

14.2.1 Scalars

Historically we started to overload our histogram mechanism to also collect scalar data, such as flag values, counts, labels and others. The scalar measurement types are the suggested way to collect that kind of scalar data. We currently only support recording of scalars from the parent process. The serialized scalar data is submitted with the main pings.

The API

Scalar probes can be managed either through the nsITelemetry interface or the C++ API.

JS API

Probes in privileged JavaScript code can use the following functions to manipulate scalars:

```
Services.telemetry.scalarAdd(aName, aValue);
Services.telemetry.scalarSet(aName, aValue);
Services.telemetry.scalarSetMaximum(aName, aValue);
```

These functions can throw if, for example, an operation is performed on a scalar type that doesn't support it (e.g. calling scalarSetMaximum on a scalar of the string kind). Please look at the code documentation for additional informations.

C++ API

Probes in native code can use the more convenient helper functions declared in Telemetry.h:

```
void ScalarAdd(mozilla::Telemetry::ScalarID aId, uint32_t aValue);
void ScalarSet(mozilla::Telemetry::ScalarID aId, uint32_t aValue);
void ScalarSet(mozilla::Telemetry::ScalarID aId, const nsAString& aValue);
void ScalarSet(mozilla::Telemetry::ScalarID aId, bool aValue);
void ScalarSetMaximum(mozilla::Telemetry::ScalarID aId, uint32_t aValue);
```

The YAML definition file

Scalar probes are required to be registered, both for validation and transparency reasons, in the Scalars.yaml definition file.

The probes in the definition file are represented in a fixed-depth, two-level structure:

```
# The following is a group.
a.group.hierarchy:
  a_probe_name:
    kind: uint
    ...
  another_probe:
    kind: string
    ...
  ...
group2:
  probe:
    kind: int
    ...
```

Group and probe names need to follow a few rules:

- they cannot exceed 40 characters each;
- group names must be alpha-numeric + `.`, with no leading/trailing digit or `.`;
- probe names must be alpha-numeric + _, with no leading/trailing digit or _.

A probe can be defined as follows:

```
a.group.hierarchy:
  a_scalar:
    bug_numbers:
      - 1276190
    description: A nice one-line description.
    expires: never
```

```
    kind: uint
    notification_emails:
      - telemetry-client-dev@mozilla.com
```

Required Fields

- `bug_numbers`: A list of unsigned integers representing the number of the bugs the probe was introduced in.
- `description`: A single or multi-line string describing what data the probe collects and when it gets collected.
- `expires`: The version number in which the scalar expires, e.g. "30"; a version number of type "N" and "N.0" is automatically converted to "N.0a1" in order to expire the scalar also in the development channels. A telemetry probe acting on an expired scalar will print a warning into the browser console. For scalars that never expire the value `never` can be used.
- `kind`: A string representing the scalar type. Allowed values are `uint`, `string` and `boolean`.
- `notification_emails`: A list of email addresses to notify with alerts of expiring probes. More importantly, these are used by the data steward to verify that the probe is still useful.

Optional Fields

- `cpp_guard`: A string that gets inserted as an `#ifdef` directive around the automatically generated C++ declaration. This is typically used for platform-specific scalars, e.g. `ANDROID`.
- `release_channel_collection`: This can be either `opt-in` (default) or `opt-out`. With the former the scalar is submitted by default on pre-release channels; on the release channel only if the user opted into additional data collection. With the latter the scalar is submitted by default on release and pre-release channels, unless the user opted out.

String type restrictions

To prevent abuses, the content of a string scalar is limited to 50 characters in length. Trying to set a longer string will result in an error and no string being set.

The processor scripts

The scalar definition file is processed and checked for correctness at compile time. If it conforms to the specification, the processor scripts generate two C++ headers files, included by the Telemetry C++ core.

gen-scalar-data.py

This script is called by the build system to generate the `TelemetryScalarData.h` C++ header file out of the scalar definitions. This header file contains an array holding the scalar names and version strings, in addition to an array of `ScalarInfo` structures representing all the scalars.

gen-scalar-enum.py

This script is called by the build system to generate the `TelemetryScalarEnums.h` C++ header file out of the scalar definitions. This header file contains an enum class with all the scalar identifiers used to access them from code through the C++ API.

14.2.2 Histograms

Recording into histograms is currently documented in a MDN article.

14.2.3 Environment

14.2.4 Measuring elapsed time

To make it easier to measure how long operations take, we have helpers for both JavaScript and C++. These helpers record the elapsed time into histograms, so you have to create suitable histograms for them first.

From JavaScript

JavaScript can measure elapsed time using TelemetryStopwatch.jsm.

`TelemetryStopwatch` is a helper that simplifies recording elapsed time (in milliseconds) into histograms (plain or keyed).

API:

```
TelemetryStopwatch = {
  // Start, cancel & finish recording elapsed time into a histogram.
  // |aObject| is optional. If specificied, the timer is associated with this
  // object, so multiple time measurements can be done concurrently.
  start(histogramId, aObject);
  cancel(histogramId, aObject);
  finish(histogramId, aObject);
  // Start, cancel & finished recording elapsed time into a keyed histogram.
  // |key| specificies the key to record into.
  // |aObject| is optional and used as above.
  startKeyed(histogramId, key, aObject);
  cancelKeyed(histogramId, key, aObject);
  finishKeyed(histogramId, key, aObject);
};
```

Example:

```
TelemetryStopwatch.start("SAMPLE_FILE_LOAD_TIME_MS");
// ... start loading file.
if (failedToOpenFile) {
  // Cancel this if the operation failed early etc.
  TelemetryStopwatch.cancel("SAMPLE_FILE_LOAD_TIME_MS");
  return;
}
// ... do more work.
TelemetryStopwatch.finish("SAMPLE_FILE_LOAD_TIME_MS");
```

From C++

API:

```
// This helper class is the preferred way to record elapsed time.
template<ID id, TimerResolution res = MilliSecond>
class AutoTimer {
  // Record into a plain histogram.
```

```
  explicit AutoTimer(TimeStamp aStart = TimeStamp::Now());
  // Record into a keyed histogram, with key |aKey|.
  explicit AutoTimer(const nsCString& aKey,
                     TimeStamp aStart = TimeStamp::Now());
};

void AccumulateTimeDelta(ID id, TimeStamp start, TimeStamp end = TimeStamp::Now());
```

14.2.5 Submitting custom pings

Custom pings can be submitted from JavaScript using:

TelemetryController.submitExternalPing(type, payload, options)

- `type` - a `string` that is the type of the ping, limited to `/^[a-z0-9][a-z0-9-]+[a-z0-9]$/i`.
- `payload` - the actual payload data for the ping, should be a JSON style object.
- **`options` - optional, an object containing additional options:**
  - `addClientId`- whether to add the client id to the ping, defaults to `false`
  - `addEnvironment` - whether to add the environment data to the ping, defaults to `false`
  - `overrideEnvironment` - a JSON style object that overrides the environment data

`TelemetryController` will assemble a ping with the passed payload and the specified options. That ping will be archived locally for use with Shield and inspection in `about:telemetry`. If the preferences allow upload of Telemetry pings, the ping will be uploaded at the next opportunity (this is subject to throttling, retry-on-failure, etc.).

Tools

Helpful tools for designing new pings include:

- gzipServer - a Python script that can run locally and receives and saves Telemetry pings. Making Firefox send to it allows inspecting outgoing pings easily.
- `about:telemetry` - allows inspecting submitted pings from the local archive, including all custom ones.

Designing custom pings

In general, creating a new custom ping means you don't benefit automatically from the existing tooling. Further work is needed to make data show up in re:dash or other analysis tools.

Other questions to guide a new pings design include:

- **Submission interval & triggers:**
  - What events trigger ping submission?
  - What interval is the ping submitted in?
  - Is there a throttling mechanism?
  - What is the desired latency? (submitting "at least daily" still leads to longer latency tails)
- **Size and volume:**
  - What's the size of the submitted payload?
  - What's the full ping size including metadata in the pipeline?

  - What's the target population?
  - What's the overall estimated volume?

- **Dataset:**
  - Is it opt-out?
  - Does it need to be opt-out?
  - Does it need to be in a separate ping? (why can't the data live in probes?)
- **Privacy:**
  - Is there risk to leak PII?
  - How is that risk mitigated?
- **Data contents:**
  - Does the submitted data answer the posed product questions?
  - Does the shape of the data allow to answer the questions efficiently?
  - Is the data limited to whats needed to answer the questions?
  - Does the data use common formats? (i.e. can we re-use tooling or analysis know-how)

14.3 Data documentation

14.3.1 Common ping format

This defines the top-level structure of a Telemetry ping. It contains basic information shared between different ping types, which enables proper storage and processing of the raw pings server-side.

It also contains optional further information:

- the environment data, which contains important info to correlate the measurements against
- the `clientId`, a UUID identifying a profile and allowing user-oriented correlation of data

Note: Both are not submitted with all ping types due to privacy concerns. This and the data it that can be correlated against is inspected under the data collection policy.

Finally, the structure also contains the *payload*, which is the specific data submitted for the respective *ping type*.

Structure:

```
{
  type: <string>, // "main", "activation", "deletion", "saved-session", ...
  id: <UUID>, // a UUID that identifies this ping
  creationDate: <ISO date>, // the date the ping was generated
  version: <number>, // the version of the ping format, currently 4

  application: {
    architecture: <string>, // build architecture, e.g. x86
    buildId: <string>, // "20141126041045"
    name: <string>, // "Firefox"
    version: <string>, // "35.0"
    displayVersion: <string>, // "35.0b3"
    vendor: <string>, // "Mozilla"
    platformVersion: <string>, // "35.0"
    xpcomAbi: <string>, // e.g. "x86-msvc"
```

```
    channel: <string>, // "beta"
  },

  clientId: <UUID>, // optional
  environment: { ... }, // optional, not all pings contain the environment
  payload: { ... }, // the actual payload data for this ping type
}
```

14.3.2 Environment

The environment consists of data that is expected to be characteristic for performance and other behavior and not expected to change too often.

Changes to most of these data points are detected (where possible and sensible) and will lead to a session split in the "main" ping. The environment data may also be submitted by other ping types.

Note: This is not submitted with all ping types due to privacy concerns. This and other data is inspected under the data collection policy.

Some parts of the environment must be fetched asynchronously at startup. We don't want other Telemetry components to block on waiting for the environment, so some items may be missing from it until the async fetching finished. This currently affects the following sections:

- profile
- addons

Structure:

```
{
  build: {
    applicationId: <string>, // nsIXULAppInfo.ID
    applicationName: <string>, // "Firefox"
    architecture: <string>, // e.g. "x86", build architecture for the active build
    architecturesInBinary: <string>, // e.g. "i386-x86_64", from nsIMacUtils.architecturesInBinary, c
    buildId: <string>, // e.g. "20141126041045"
    version: <string>, // e.g. "35.0"
    vendor: <string>, // e.g. "Mozilla"
    platformVersion: <string>, // e.g. "35.0"
    xpcomAbi: <string>, // e.g. "x86-msvc"
    hotfixVersion: <string>, // e.g. "20141211.01"
  },
  settings: {
    addonCompatibilityCheckEnabled: <bool>, // Whether application compatibility is respected for ado
    blocklistEnabled: <bool>, // true on failure
    isDefaultBrowser: <bool>, // null on failure, not available on Android
    defaultSearchEngine: <string>, // e.g. "yahoo"
    defaultSearchEngineData: {, // data about the current default engine
      name: <string>, // engine name, e.g. "Yahoo"; or "NONE" if no default
      loadPath: <string>, // where the engine line is located; missing if no default
      origin: <string>, // 'default', 'verified', 'unverified', or 'invalid'; based on the presence a
      submissionURL: <string> // missing if no default or for user-installed engines
    },
    searchCohort: <string>, // optional, contains an identifier for any active search A/B experiments
    e10sEnabled: <bool>, // whether e10s is on, i.e. browser tabs open by default in a different proc
    e10sCohort: <string>, // which e10s cohort was assigned for this user
    telemetryEnabled: <bool>, // false on failure
    locale: <string>, // e.g. "it", null on failure
    update: {
```

```
    channel: <string>, // e.g. "release", null on failure
    enabled: <bool>, // true on failure
    autoDownload: <bool>, // true on failure
  },
  userPrefs: {
    // Only prefs which are changed from the default value are listed
    // in this block
    "pref.name.value": value // some prefs send the value
    "pref.name.url": "<user-set>" // For some privacy-sensitive prefs
      // only the fact that the value has been changed is recorded
  },
},
profile: {
  creationDate: <integer>, // integer days since UNIX epoch, e.g. 16446
  resetDate: <integer>, // integer days since UNIX epoch, e.g. 16446 - optional
},
partner: { // This section may not be immediately available on startup
  distributionId: <string>, // pref "distribution.id", null on failure
  distributionVersion: <string>, // pref "distribution.version", null on failure
  partnerId: <string>, // pref mozilla.partner.id, null on failure
  distributor: <string>, // pref app.distributor, null on failure
  distributorChannel: <string>, // pref app.distributor.channel, null on failure
  partnerNames: [
    // list from prefs app.partner.<name>=<name>
  ],
},
system: {
  memoryMB: <number>,
  virtualMaxMB: <number>, // windows-only
  isWow64: <bool>, // windows-only
  cpu: {
      count: <number>,  // desktop only, e.g. 8, or null on failure - logical cpus
      cores: <number>, // desktop only, e.g., 4, or null on failure - physical cores
      vendor: <string>, // desktop only, e.g. "GenuineIntel", or null on failure
      family: <number>, // desktop only, null on failure
      model: <number, // desktop only, null on failure
      stepping: <number>, // desktop only, null on failure
      l2cacheKB: <number>, // L2 cache size in KB, only on windows & mac
      l3cacheKB: <number>, // desktop only, L3 cache size in KB
      speedMHz: <number>, // desktop only, cpu clock speed in MHz
      extensions: [
        <string>,
        ...
        // as applicable:
        // "MMX", "SSE", "SSE2", "SSE3", "SSSE3", "SSE4A", "SSE4_1",
        // "SSE4_2", "AVX", "AVX2", "EDSP", "ARMv6", "ARMv7", "NEON"
      ],
  },
  device: { // This section is only available on mobile devices.
    model: <string>, // the "device" from FHR, null on failure
    manufacturer: <string>, // null on failure
    hardware: <string>, // null on failure
    isTablet: <bool>, // null on failure
  },
  os: {
      name: <string>, // "Windows_NT" or null on failure
      version: <string>, // e.g. "6.1", null on failure
      kernelVersion: <string>, // android/b2g only or null on failure
```

```
      servicePackMajor: <number>, // windows only or null on failure
      servicePackMinor: <number>, // windows only or null on failure
      windowsBuildNumber: <number>, // windows 10 only or null on failure
      windowsUBR: <number>, // windows 10 only or null on failure
      installYear: <number>, // windows only or null on failure
      locale: <string>, // "en" or null on failure
  },
  hdd: {
    profile: { // hdd where the profile folder is located
        model: <string>, // windows only or null on failure
        revision: <string>, // windows only or null on failure
    },
    binary:  { // hdd where the application binary is located
        model: <string>, // windows only or null on failure
        revision: <string>, // windows only or null on failure
    },
    system:  { // hdd where the system files are located
        model: <string>, // windows only or null on failure
        revision: <string>, // windows only or null on failure
    },
  },
  gfx: {
      D2DEnabled: <bool>, // null on failure
      DWriteEnabled: <bool>, // null on failure
      //DWriteVersion: <string>, // temporarily removed, pending bug 1154500
      adapters: [
        {
          description: <string>, // e.g. "Intel(R) HD Graphics 4600", null on failure
          vendorID: <string>, // null on failure
          deviceID: <string>, // null on failure
          subsysID: <string>, // null on failure
          RAM: <number>, // in MB, null on failure
          driver: <string>, // null on failure
          driverVersion: <string>, // null on failure
          driverDate: <string>, // null on failure
          GPUActive: <bool>, // currently always true for the first adapter
        },
        ...
      ],
      // Note: currently only added on Desktop. On Linux, only a single
      // monitor is returned representing the entire virtual screen.
      monitors: [
        {
          screenWidth: <number>,  // screen width in pixels
          screenHeight: <number>, // screen height in pixels
          refreshRate: <number>,  // refresh rate in hertz (present on Windows only).
                                  //  (values <= 1 indicate an unknown value)
          pseudoDisplay: <bool>,  // networked screen (present on Windows only)
          scale: <number>,        // backing scale factor (present on Mac only)
        },
        ...
      ],
      features: {
        compositor: <string>,     // Layers backend for compositing (eg "d3d11", "none", "opengl")

        // Each the following features can have one of the following statuses:
        //   "unused"      - This feature has not been requested.
        //   "unavailable" - Safe Mode or OS restriction prevents use.
```

```
          //   "blocked"     - Blocked due to an internal condition such as safe mode.
          //   "blacklisted" - Blocked due to a blacklist restriction.
          //   "disabled"    - User explicitly disabled this default feature.
          //   "failed"      - This feature was attempted but failed to initialize.
          //   "available"   - User has this feature available.
          "d3d11" { // This feature is Windows-only.
            status: <string>,
            warp: <bool>,           // Software rendering (WARP) mode was chosen.
            textureSharing: <bool>  // Whether or not texture sharing works.
            version: <number>,      // The D3D11 device feature level.
            blacklisted: <bool>,    // Whether D3D11 is blacklisted; use to see whether WARP
                                    // was blacklist induced or driver-failure induced.
          },
          "d2d" { // This feature is Windows-only.
            status: <string>,
            version: <string>,      // Either "1.0" or "1.1".
          },
        },
      },
  },
  addons: {
    activeAddons: { // the currently enabled addons
      <addon id>: {
        blocklisted: <bool>,
        description: <string>, // null if not available
        name: <string>,
        userDisabled: <bool>,
        appDisabled: <bool>,
        version: <string>,
        scope: <integer>,
        type: <string>, // "extension", "service", ...
        foreignInstall: <bool>,
        hasBinaryComponents: <bool>
        installDay: <number>, // days since UNIX epoch, 0 on failure
        updateDay: <number>, // days since UNIX epoch, 0 on failure
        signedState: <integer>, // whether the add-on is signed by AMO, only present for extensions
        isSystem: <bool>, // true if this is a System Add-on
      },
      ...
    },
    theme: { // the active theme
      id: <string>,
      blocklisted: <bool>,
      description: <string>,
      name: <string>,
      userDisabled: <bool>,
      appDisabled: <bool>,
      version: <string>,
      scope: <integer>,
      foreignInstall: <bool>,
      hasBinaryComponents: <bool>
      installDay: <number>, // days since UNIX epoch, 0 on failure
      updateDay: <number>, // days since UNIX epoch, 0 on failure
    },
    activePlugins: [
      {
        name: <string>,
        version: <string>,
```

```
        description: <string>,
        blocklisted: <bool>,
        disabled: <bool>,
        clicktoplay: <bool>,
        mimeTypes: [<string>, ...],
        updateDay: <number>, // days since UNIX epoch, 0 on failure
      },
      ...
    ],
    activeGMPlugins: {
        <gmp id>: {
            version: <string>,
            userDisabled: <bool>,
            applyBackgroundUpdates: <integer>,
        },
        ...
    },
    activeExperiment: { // section is empty if there's no active experiment
        id: <string>, // id
        branch: <string>, // branch name
    },
    persona: <string>, // id of the current persona, null on GONK
  },
}
```

build

buildId

Firefox builds downloaded from mozilla.org use a 14-digit buildId. Builds included in other distributions may have a different format (e.g. only 10 digits).

Settings

defaultSearchEngine

Note: Deprecated, use defaultSearchEngineData instead.

Contains the string identifier or name of the default search engine provider. This will not be present in environment data collected before the Search Service initialization.

The special value `NONE` could occur if there is no default search engine.

The special value `UNDEFINED` could occur if a default search engine exists but its identifier could not be determined.

This field's contents are `Services.search.defaultEngine.identifier` (if defined) or `"other-" + Services.search.defaultEngine.name` if not. In other words, search engines without an `.identifier` are prefixed with `other-`.

defaultSearchEngineData

Contains data identifying the engine currently set as the default.

The object contains:

- a `name` property with the name of the engine, or `NONE` if no engine is currently set as the default.
- a `loadPath` property: an anonymized path of the engine xml file, e.g. jar:[app]/omni.ja!browser/engine.xml (where 'browser' is the name of the chrome package, not a folder) [profile]/searchplugins/engine.xml [distribution]/searchplugins/common/engine.xml [other]/engine.xml
- an `origin` property: the value will be `default` for engines that are built-in or from distribution partners, `verified` for user-installed engines with valid verification hashes, `unverified` for non-default engines without verification hash, and `invalid` for engines with broken verification hashes.
- a `submissionURL` property with the HTTP url we would use to search. For privacy, we don't record this for user-installed engines.

`loadPath` and `submissionURL` are not present if `name` is `NONE`.

searchCohort

If the user has been enrolled into a search default change experiment, this contains the string identifying the experiment the user is taking part in. Most user profiles will never be part of any search default change experiment, and will not send this value.

userPrefs

This object contains user preferences.

Each key in the object is the name of a preference. A key's value depends on the policy with which the preference was collected. There are two such policies, "value" and "state". For preferences collected under the "value" policy, the value will be the preference's value. For preferences collected under the "state" policy, the value will be an opaque marker signifying only that the preference has a user value. The "state" policy is therefore used when user privacy is a concern.

The following is a partial list of collected preferences.

- `browser.search.suggest.enabled`: The "master switch" for search suggestions everywhere in Firefox (search bar, urlbar, etc.). Defaults to true.
- `browser.urlbar.suggest.searches`: True if search suggestions are enabled in the urlbar. Defaults to false.
- `browser.urlbar.userMadeSearchSuggestionsChoice`: True if the user has clicked Yes or No in the urlbar's opt-in notification. Defaults to false.
- `browser.zoom.full`: True if zoom is enabled for both text and images, that is if "Zoom Text Only" is not enabled. Defaults to true. Collection of this preference has been enabled in Firefox 50 and will be disabled again in Firefox 53 (Bug 979323).

partner

If the user is using a partner repack, this contains information identifying the repack being used, otherwise "partnerNames" will be an empty array and other entries will be null. The information may be missing when the profile just becomes available. In Firefox for desktop, the information along with other customizations defined in distribution.ini are processed later in the startup phase, and will be fully applied when "distribution-customization-complete" notification is sent.

Distributions are most reliably identified by the `distributionId` field. Partner information can be found in the partner repacks (the old one is deprecated): it contains one private repository per partner. Important values for `distributionId` include:

- "MozillaOnline" for the Mozilla China repack.
- "canonical", for the Ubuntu Firefox repack.
- "yandex", for the Firefox Build by Yandex.

system

os

This object contains operating system information.

- `name`: the name of the OS.
- `version`: a string representing the OS version.
- `kernelVersion`: an Android/B2G only string representing the kernel version.
- `servicePackMajor`: the Windows only major version number for the installed service pack.
- `servicePackMinor`: the Windows only minor version number for the installed service pack.
- `windowsBuildNumber`: the Windows build number, only available for Windows >= 10.
- `windowsUBR`: the Windows UBR number, only available for Windows >= 10. This value is incremented by Windows cumulative updates patches.
- `installYear`: the Windows only integer representing the year the OS was installed.
- `locale`: the string representing the OS locale.

addons

activeAddons

Starting from Firefox 44, the length of the following string fields: `name`, `description` and `version` is limited to 100 characters. The same limitation applies to the same fields in `theme` and `activePlugins`.

14.3.3 "main" ping

This is the "main" Telemetry ping type, whose payload contains most of the measurements that are used to track the performance and health of Firefox in the wild. It includes the histograms and other performance and diagnostic data.

This ping is triggered by different scenarios, which is documented by the `reason` field:

- `aborted-session` - this ping is regularly saved to disk (every 5 minutes), overwriting itself, and deleted at shutdown. If a previous aborted session ping is found at startup, it gets sent to the server. The first aborted-session ping is generated as soon as Telemetry starts
- `environment-change` - the Environment changed, so the session measurements got reset and a new sub-session starts
- `shutdown` - triggered when the browser session ends
- `daily` - a session split triggered in 24h hour intervals at local midnight. If an `environment-change` ping is generated by the time it should be sent, the daily ping is rescheduled for the next midnight
- `saved-session` - the *"classic"* Telemetry payload with measurements covering the whole browser session (only submitted for a transition period)

Most reasons lead to a session split, initiating a new *subsession*. We reset important measurements for those subsessions.

After a new subsession split, the `internal-telemetry-after-subsession-split` topic is notified to all the observers. *This is an internal topic and is only meant for internal Telemetry usage.*

Note: `saved-session` is sent with a different ping type (`saved-session`, not `main`), but otherwise has the same format as discussed here.

Structure:

```
{
  version: 4,

  info: {
    reason: <string>, // what triggered this ping: "saved-session", "environment-change", "shutdown",
    revision: <string>, // the Histograms.json revision
    timezoneOffset: <integer>, // time-zone offset from UTC, in minutes, for the current locale
    previousBuildId: <string>, // null if this is the first run, or the previous build ID is unknown

    sessionId: <uuid>,  // random session id, shared by subsessions
    subsessionId: <uuid>,  // random subsession id
    previousSessionId: <uuid>, // session id of the previous session, null on first run.
    previousSubsessionId: <uuid>, // subsession id of the previous subsession (even if it was in a di
                                  // null on first run.

    subsessionCounter: <unsigned integer>, // the running no. of this subsession since the start of t
    profileSubsessionCounter: <unsigned integer>, // the running no. of all subsessions for the whole

    sessionStartDate: <ISO date>, // daily precision
    subsessionStartDate: <ISO date>, // daily precision, ISO date in local time
    sessionLength: <integer>, // the session length until now in seconds, monotonic
    subsessionLength: <integer>, // the subsession length in seconds, monotonic

    flashVersion: <string>, // obsolete, use ``environment.addons.activePlugins``
    addons: <string>, // obsolete, use ``environment.addons``
  },

  processes: {...},
  childPayloads: [...], // only present with e10s; reduced payloads from content processes, null on f
  simpleMeasurements: {...},

  // The following properties may all be null if we fail to collect them.
  histograms: {...},
  keyedHistograms: {...},
  chromeHangs: {...},
  threadHangStats: [...],
  log: [...],
  webrtc: {...},
  fileIOReports: {...},
  lateWrites: {...},
  addonDetails: {...},
  addonHistograms: {...},
  UIMeasurements: [...],
  slowSQL: {...},
  slowSQLstartup: {...},
}
```

info

sessionLength

The length of the current session so far in seconds. This uses a monotonic clock, so this may mismatch with other measurements that are not monotonic like calculations based on `Date.now()`.

If the monotonic clock failed, this will be `-1`.

subsessionLength

The length of this subsession in seconds. This uses a monotonic clock, so this may mismatch with other measurements that are not monotonic (e.g. based on Date.now()).

If `sessionLength` is `-1`, the monotonic clock is not working.

processes

This section contains per-process data.

Structure:

```
"processes" : {
  ... other processes ...
  "parent": {
    scalars: {...},
  },
}
```

scalars

This section contains the Scalars that are valid for the current platform. Scalars are not created nor submitted if no data was added to them, and are only reported with subsession pings. Scalar data is only currently reported for the main process. Their type and format is described by the `Scalars.yaml` file. Its most recent version is available here. The `info.revision` field indicates the revision of the file that describes the reported scalars.

childPayloads

The Telemetry payloads sent by child processes, recorded on child process shutdown (event `content-child-shutdown` observed) and whenever `TelemetrySession.requestChildPayloads()` is called (currently only used in tests). They are reduced session payloads, only available with e10s. Among some other things, they don't report addon details, addon histograms or UI Telemetry.

Any histogram whose Accumulate call happens on a child process will be accumulated into a childPayload's histogram, not the parent's. As such, some histograms in childPayloads will contain different data (e.g. `GC_MS` will be much different in childPayloads, for instance, because the child GC needs to content with content scripts and parent doesn't) and some histograms will be absent (`EVENTLOOP_UI_ACTIVITY` is parent-process-only because it measures inter-event timings where the OS delivers the events in the parent).

Note: Child payloads are not collected and cleared with subsession splits, they are currently only meaningful when analysed from `saved-session` or `main` pings with `reason` set to `shutdown`.

simpleMeasurements

This section contains a list of simple measurements, or counters. In addition to the ones highlighted below, Telemetry timestamps (see here and here) can be reported.

totalTime

A non-monotonic integer representing the number of seconds the session has been alive.

uptime

A non-monotonic integer representing the number of minutes the session has been alive.

addonManager

Only available in the extended set of measures, it contains a set of counters related to Addons. See here for a list of recorded measures.

UITelemetry

Only available in the extended set of measures. For more see *UITelemetry data format*.

startupInterrupted

A boolean set to true if startup was interrupted by an interactive prompt.

js

This section contains a series of counters from the JavaScript engine.

Structure:

```
"js" : {
  "setProto": <unsigned integer>, // Number of times __proto__ is set
  "customIter": <unsigned integer> // Number of times __iterator__ is used (i.e., is found for a for-
}
```

maximalNumberOfConcurrentThreads

An integer representing the highest number of threads encountered so far during the session.

startupSessionRestoreReadBytes

Windows-only integer representing the number of bytes read by the main process up until the session store has finished restoring the windows.

startupSessionRestoreWriteBytes

Windows-only integer representing the number of bytes written by the main process up until the session store has finished restoring the windows.

startupWindowVisibleReadBytes

Windows-only integer representing the number of bytes read by the main process up until after a XUL window is made visible.

startupWindowVisibleWriteBytes

Windows-only integer representing the number of bytes written by the main process up until after a XUL window is made visible.

debuggerAttached

A boolean set to true if a debugger is attached to the main process.

shutdownDuration

The time, in milliseconds, it took to complete the last shutdown.

failedProfileLockCount

The number of times the system failed to lock the user profile.

savedPings

Integer count of the number of pings that need to be sent.

activeTicks

Integer count of the number of five-second intervals ('ticks') the user was considered 'active' (sending UI events to the window). An extra event is fired immediately when the user becomes active after being inactive. This is for some mouse and gamepad events, and all touch, keyboard, wheel, and pointer events (see EventStateManager.cpp). This measure might be useful to give a trend of how much a user actually interacts with the browser when compared to overall session duration. It does not take into account whether or not the window has focus or is in the foreground. Just if it is receiving these interaction events. Note that in `main` pings, this measure is reset on subsession splits, while in `saved-session` pings it covers the whole browser session.

pingsOverdue

Integer count of pending pings that are overdue.

histograms

This section contains the histograms that are valid for the current platform. `Flag` and `count` histograms are always created and submitted, with their default value being respectively `false` and `0`. Other histogram types (see here) are not created nor submitted if no data was added to them. The type and format of the reported histograms is described by the `Histograms.json` file. Its most recent version is available here. The `info.revision` field indicates the revision of the file that describes the reported histograms.

keyedHistograms

This section contains the keyed histograms available for the current platform.

As of Firefox 48, this section does not contain empty keyed histograms anymore.

threadHangStats

Contains the statistics about the hangs in main and background threads. Note that hangs in this section capture the [C++ pseudostack](https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Profiling_with_the_Built-in_Profiler#Native_stack_vs._Pseudo_stack) and an incomplete JS stack, which is not 100% precise.

To avoid submitting overly large payloads, some limits are applied:

- Identical, adjacent "(chrome script)" or "(content script)" stack entries are collapsed together. If a stack is reduced, the "(reduced stack)" frame marker is added as the oldest frame.
- The depth of the reported stacks is limited to 11 entries. This value represents the 99.9th percentile of the thread hangs stack depths reported by Telemetry.

Structure:

```
"threadHangStats" : [
  {
    "name" : "Gecko",
    "activity" : {...}, // a time histogram of all task run times
    "hangs" : [
      {
        "stack" : [
          "Startup::XRE_Main",
          "Timer::Fire",
          "(content script)",
          "IPDL::PPluginScriptableObject::SendGetChildProperty",
          ... up to 11 frames ...
        ],
        "nativeStack": [...], // optionally available
        "histogram" : {...}, // the time histogram of the hang times
        "annotations" : [
          {
            "pluginName" : "Shockwave Flash",
            "pluginVersion" : "18.0.0.209"
          },
          ... other annotations ...
        ]
      },
    ],
  },
  ... other threads ...
 ]
```

chromeHangs

Contains the statistics about the hangs happening exclusively on the main thread of the parent process. Precise C++ stacks are reported. This is only available on Nightly Release on Windows, when building using "–enable-profiling" switch.

Some limits are applied:

- Reported chrome hang stacks are limited in depth to 50 entries.
- The maximum number of reported stacks is 50.

Structure:

```
"chromeHangs" : {
  "memoryMap" : [
    ["wgdi32.pdb", "08A541B5942242BDB4AEABD8C87E4CFF2"],
    ["igd10iumd32.pdb", "D36DEBF2E78149B5BE1856B772F1C3991"],
    ... other entries in the format ["module name", "breakpad identifier"] ...
   ],
  "stacks" : [
    [
      [
        0, // the module index or -1 for invalid module indices
        190649 // the offset of this program counter in its module or an absolute pc
      ],
      [1, 2540075],
      ... other frames, up to 50 ...
     ],
     ... other stacks, up to 50 ...
  ],
  "durations" : [8, ...], // the hang durations (in seconds)
  "systemUptime" : [692, ...], // the system uptime (in minutes) at the time of the hang
  "firefoxUptime" : [672, ...], // the Firefox uptime (in minutes) at the time of the hang
  "annotations" : [
    [
      [0, ...], // the indices of the related hangs
      {
        "pluginName" : "Shockwave Flash",
        "pluginVersion" : "18.0.0.209",
        ... other annotations as key:value pairs ...
      }
    ],
    ...
  ]
},
```

log

This section contains a log of important or unusual events reported through Telemetry.

Structure:

```
"log": [
  [
    "Event_ID",
    3785, // the timestamp (in milliseconds) for the log entry
    ... other data ...
  ],
```

```
  ...
]
```

webrtc

Contains special statistics gathered by WebRTC related components.

So far only a bitmask for the ICE candidate type present in a successful or failed WebRTC connection is getting reported through C++ code as IceCandidatesStats, because the required bitmask is too big to be represented in a regular enum histogram. Further this data differentiates between Loop (aka Firefox Hello) connections and everything else, which is categorized as WebRTC.

Note: in most cases the webrtc and loop dictionaries inside of IceCandidatesStats will simply be empty as the user has not used any WebRTC PeerConnection at all during the ping report time.

Structure:

```
"webrtc": {
  "IceCandidatesStats": {
    "webrtc": {
      "34526345": {
        "successCount": 5
      },
      "2354353": {
        "failureCount": 1
      }
    },
    "loop": {
      "2349346359": {
        "successCount": 3
      },
      "73424": {
        "successCount": 1,
        "failureCount": 5
      }
    }
  }
},
```

fileIOReports

Contains the statistics of main-thread I/O recorded during the execution. Only the I/O stats for the XRE and the profile directories are currently reported, neither of them disclosing the full local path.

Structure:

```
"fileIOReports": {
  "{xre}": [
    totalTime, // Accumulated duration of all operations
    creates, // Number of create/open operations
    reads, // Number of read operations
    writes, // Number of write operations
    fsyncs, // Number of fsync operations
    stats, // Number of stat operations
  ],
  "{profile}": [ ... ],
```

```
  ...
}
```

lateWrites

This sections reports writes to the file system that happen during shutdown. The reported data contains the stack and the loaded libraries at the time the writes happened.

Structure:

```
"lateWrites" : {
  "memoryMap" : [
    ["wgdi32.pdb", "08A541B5942242BDB4AEABD8C87E4CFF2"],
    ... other entries in the format ["module name", "breakpad identifier"] ...
   ],
  "stacks" : [
    [
      [
        0, // the module index or -1 for invalid module indices
        190649 // the offset of this program counter in its module or an absolute pc
      ],
      [1, 2540075],
      ... other frames ...
     ],
     ... other stacks ...
  ],
},
```

addonDetails

This section contains per-addon telemetry details, as reported by each addon provider. The XPI provider is the only one reporting at the time of writing (see DXR). Telemetry does not manipulate or enforce a specific format for the supplied provider's data.

Structure:

```
"addonDetails": {
  "XPI": {
    "adbhelper@mozilla.org": {
      "scan_items": 24,
      "scan_MS": 3,
      "location": "app-profile",
      "name": "ADB Helper",
      "creator": "Mozilla & Android Open Source Project",
      "startup_MS": 30
    },
    ...
  },
  ...
}
```

addonHistograms

This section contains the histogram registered by the addons (see here). This section is not present if no addon histogram is available.

UITelemetry

See the `UITelemetry data format` documentation.

slowSQL

This section contains the informations about the slow SQL queries for both the main and other threads. The execution of an SQL statement is considered slow if it takes 50ms or more on the main thread or 100ms or more on other threads. Slow SQL statements will be automatically trimmed to 1000 characters. This limit doesn't include the ellipsis and database name, that are appended at the end of the stored statement.

Structure:

```
"slowSQL": {
  "mainThread": {
    "Sanitized SQL Statement": [
      1, // the number of times this statement was hit
      200  // the total time (in milliseconds) that was spent on this statement
    ],
    ...
  },
  "otherThreads": {
    "VACUUM /* places.sqlite */": [
      1,
      330
    ],
    ...
  }
},
```

slowSQLStartup

This section contains the slow SQL statements gathered at startup (until the "sessionstore-windows-restored" event is fired). The structure of this section resembles the one for *slowSQL*.

UIMeasurements

This section contains UI specific telemetry measurements and events. This section is mainly populated with Android-specific data and events (see here).

Structure:

```
"UIMeasurements": [
  {
    "type": "event", // either "session" or "event"
    "action": "action.1",
    "method": "menu",
    "sessions": [],
    "timestamp": 12345,
    "extras": "settings"
  },
  {
    "type": "session",
    "name": "awesomescreen.1",
    "reason": "commit",
```

```
    "start": 123,
    "end": 456
  }
  ...
],
```

14.3.4 "deletion" ping

This ping is generated when a user turns off FHR upload from the Preferences panel, changing the related `datareporting.healthreport.uploadEnabled` preference. This requests that all associated data from that user be deleted.

This ping contains the client id and no environment data.

Structure:

```
{
  version: 4,
  type: "deletion",
  ... common ping data
  clientId: <UUID>,
  payload: { }
}
```

14.3.5 "crash" ping

This ping is captured after the main Firefox process crashes, whether or not the crash report is submitted to crash-stats.mozilla.org. It includes non-identifying metadata about the crash.

The environment block that is sent with this ping varies: if Firefox was running long enough to record the environment block before the crash, then the environment at the time of the crash will be recorded and `hasCrashEnvironment` will be true. If Firefox crashed before the environment was recorded, `hasCrashEnvironment` will be false and the recorded environment will be the environment at time of submission.

The client ID is submitted with this ping.

Structure:

```
{
  version: 1,
  type: "crash",
  ... common ping data
  clientId: <UUID>,
  environment: { ... },
  payload: {
    crashDate: "YYYY-MM-DD",
    sessionId: <UUID>, // may be missing for crashes that happen early
                       // in startup. Added in Firefox 48 with the
                       // intention of uplifting to Firefox 46
    metadata: {...}, // Annotations saved while Firefox was running. See nsExceptionHandler.cpp for r
    hasCrashEnvironment: bool
  }
}
```

14.3.6 "core" ping

This mobile-specific ping is intended to provide the most critical data in a concise format, allowing for frequent uploads.

Since this ping is used to measure retention, it should be sent each time the browser is opened.

Submission will be per the Edge server specification:

```
/submit/telemetry/docId/docType/appName/appVersion/appUpdateChannel/appBuildID
```

- `docId` is a UUID for deduping
- `docType` is "core"
- `appName` is "Fennec"
- `appVersion` is the version of the application (e.g. "46.0a1")
- `appUpdateChannel` is "release", "beta", etc.
- `appBuildID` is the build number

Note: Counts below (e.g. search & usage times) are "since the last ping", not total for the whole application lifetime.

Structure:

```
{
  "v": 7, // ping format version
  "clientId": <string>, // client id, e.g.
                        // "c641eacf-c30c-4171-b403-f077724e848a"
  "seq": <positive integer>, // running ping counter, e.g. 3
  "locale": <string>, // application locale, e.g. "en-US"
  "os": <string>, // OS name.
  "osversion": <string>, // OS version.
  "device": <string>, // Build.MANUFACTURER + " - " + Build.MODEL
                      // where manufacturer is truncated to 12 characters
                      // & model is truncated to 19 characters
  "arch": <string>, // e.g. "arm", "x86"
  "profileDate": <pos integer>, // Profile creation date in days since
                                // UNIX epoch.
  "defaultSearch": <string>, // Identifier of the default search engine,
                             // e.g. "yahoo".
  "distributionId": <string>, // Distribution identifier (optional)
  "created": <string>, // date the ping was created
                       // in local time, "yyyy-mm-dd"
  "tz": <integer>, // timezone offset (in minutes) of the
                   // device when the ping was created
  "sessions": <integer>, // number of sessions since last upload
  "durations": <integer>, // combined duration, in seconds, of all
                                // sessions since last upload
  "searches": <object>, // Optional, object of search use counts in the
                        // format: { "engine.source": <pos integer> }
                        // e.g.: { "yahoo.suggestion": 3, "other.listitem": 1 }
  "experiments": [<string>, ...], // Optional, array of identifiers
                                  // for the active experiments
}
```

Field details

device

The `device` field is filled in with information specified by the hardware manufacturer. As such, it could be excessively long and use excessive amounts of limited user data. To avoid this, we limit the length of the field. We're more likely have collisions for models within a manufacturer (e.g. "Galaxy S5" vs. "Galaxy Note") than we are for shortened manufacturer names so we provide more characters for the model than the manufacturer.

distributionId

The `distributionId` contains the distribution ID as specified by preferences.json for a given distribution. More information on distributions can be found here.

It is optional.

defaultSearch

On Android, this field may be `null`. To get the engine, we rely on `SearchEngineManager#getDefaultEngine`, which searches in several places in order to find the search engine identifier:

- Shared Preferences
- The distribution (if it exists)
- The localized default engine

If the identifier could not be retrieved, this field is `null`. If the identifier is retrieved, we attempt to create an instance of the search engine from the search plugins (in order):

- In the distribution
- From the localized plugins shipped with the browser
- The third-party plugins that are installed in the profile directory

If the plugins fail to create a search engine instance, this field is also `null`.

This field can also be `null` when a custom search engine is set as the default.

sessions & durations

On Android, a session is the time when Firefox is focused in the foreground. *sessions* tracks the number of sessions since the last upload and *durations* is the accumulated duration in seconds of all of these sessions. Note that showing a dialog (including a Firefox dialog) will take Firefox out of focus & end the current session.

An implementation that records a session when Firefox is completely hidden is preferrable (e.g. to avoid the dialog issue above), however, it's more complex to implement and so we chose not to, at least for the initial implementation.

profileDate

On Android, this value is created at profile creation time and retrieved or, for legacy profiles, taken from the package install time (note: this is not the same exact metric as profile creation time but we compromised in favor of ease of implementation).

Additionally on Android, this field may be `null` in the unlikely event that all of the following events occur:

1. The times.json file does not exist
2. The package install date could not be persisted to disk

The reason we don't just return the package install time even if the date could not be persisted to disk is to ensure the value doesn't change once we start sending it: we only want to send consistent values.

searches

In the case a search engine is added by a user, the engine identifier "other" is used, e.g. "other.<source>".

Sources in Android are based on the existing UI telemetry values and are as follows:

- actionbar: the user types in the url bar and hits enter to use the default search engine
- listitem: the user selects a search engine from the list of secondary search engines at the bottom of the screen
- suggestion: the user clicks on a search suggestion or, in the case that suggestions are disabled, the row corresponding with the main engine

Other parameters

HTTP "Date" header

This header is used to track the submission date of the core ping in the format specified by rfc 2616 sec 14.18, et al (e.g. "Tue, 01 Feb 2011 14:00:00 GMT").

Version history

- v7: added `sessionCount` & `sessionDuration`
- v6: added `searches`
- v5: added `created` & `tz`
- v4: `profileDate` will return package install time when times.json is not available
- v3: added `defaultSearch`
- v2: added `distributionId`
- v1: initial version

Notes

- `distributionId` (v2) actually landed after `profileDate` (v4) but was uplifted to 46, whereas `profileDate` landed on 47. The version numbers in code were updated to be increasing (bug 1264492) and the version history docs rearranged accordingly.

Android implementation notes

On Android, the uploader has a high probability of delivering the complete data for a given client but not a 100% probability. This was a conscious decision to keep the code simple. The cases where we can lose data:

- Resetting the field measurements (including incrementing the sequence number) and storing a ping for upload are not atomic. Android can kill our process for memory pressure in between these distinct operations so we can just lose a ping's worth of data. That sequence number will be missing on the server.
- If we exceed some number of pings on disk that have not yet been uploaded, we remove old pings to save storage space. For those pings, we will lose their data and their sequence numbers will be missing on the server.

Note: we never expect to drop data without also dropping a sequence number so we are able to determine when data loss occurs.

14.3.7 "heartbeat" ping

This ping is submitted after a Firefox Heartbeat survey. Even if the user exits the browser, closes the survey window, or ignores the survey, Heartbeat will provide a ping to Telemetry for sending during the same session.

The payload contains the user's survey response (if any) as well as timestamps of various Heartbeat events (survey shown, survey closed, link clicked, etc).

The ping will also report the "surveyId", "surveyVersion" and "testing" Heartbeat survey parameters (if they are present in the survey config). These "meta fields" will be repeated verbatim in the payload section.

The environment block and client ID are submitted with this ping.

Structure:

```
{
  type: "heartbeat",
  version: 4,
  clientId: <UUID>,
  environment: { ... }
  ... common ping data ...
  payload: {
    version: 1,
    flowId: <string>,
    ... timestamps below ...
    offeredTS: <integer epoch timestamp>,
    learnMoreTS: <integer epoch timestamp>,
    votedTS: <integer epoch timestamp>,
    engagedTS: <integer epoch timestamp>,
    closedTS: <integer epoch timestamp>,
    expiredTS: <integer epoch timestamp>,
    windowClosedTS: <integer epoch timestamp>,
    ... user's rating below ...
    score: <integer>,
    ... survey meta fields below ...
    surveyId: <string>,
    surveyVersion: <integer>,
    testing: <boolean>
  }
}
```

Notes:

- Pings will **NOT** have all possible timestamps, timestamps are only reported for events that actually occurred.

- **Timestamp meanings:**
  - offeredTS: when the survey was shown to the user
  - learnMoreTS: when the user clicked on the "Learn More" link
  - votedTS: when the user voted
  - engagedTS: when the user clicked on the survey-provided button (alternative to voting feature)
  - closedTS: when the Heartbeat notification bar was closed
  - expiredTS: indicates that the survey expired after 2 hours of no interaction (threshold regulated by "browser.uitour.surveyDuration" pref)
  - windowClosedTS: the user closed the entire Firefox window containing the survey, thus ending the survey. This timestamp will also be reported when the survey is ended by the browser being shut down.
- The surveyId/surveyVersion fields identify a specific survey (like a "1040EZ" tax paper form). The flowID is a UUID that uniquely identifies a single user's interaction with the survey. Think of it as a session token.
- The self-support page cannot include additional data in this payload. Only the the 4 flowId/surveyId/surveyVersion/testing fields are under the self-support page's control.

See also: common ping fields

14.3.8 "sync" ping

This ping is generated after a sync is completed, for both successful and failed syncs. It's payload contains measurements pertaining to sync performance and error information. It does not contain the enviroment block, nor the clientId.

A JSON-schema document describing the exact format of the ping's payload property can be found at services/sync/tests/unit/sync_ping_schema.json.

Structure:

```
{
  version: 4,
  type: "sync",
  ... common ping data
  payload: {
    version: 1,
    when: <integer milliseconds since epoch>,
    took: <integer duration in milliseconds>,
    uid: <string>, // FxA unique ID, or empty string.
    didLogin: <bool>, // Optional, is this the first sync after login? Excluded if we don't know.
    why: <string>, // Optional, why the sync occured, excluded if we don't know.

    // Optional, excluded if there was no error.
    failureReason: {
      name: <string>, // "httperror", "networkerror", "shutdownerror", etc.
      code: <integer>, // Only present for "httperror" and "networkerror".
      error: <string>, // Only present for "otherերror" and "unexpectederror".
      from: <string>, // Optional, and only present for "autherror".
    },
    // Internal sync status information. Omitted if it would be empty.
    status: {
      sync: <string>, // The value of the Status.sync property, unless it indicates success.
      service: <string>, // The value of the Status.service property, unless it indicates success.
```

```
    },
    // Information about each engine's sync.
    engines: [
      {
        name: <string>, // "bookmarks", "tabs", etc.
        took: <integer duration in milliseconds>, // Optional, values of 0 are omitted.

        status: <string>, // The value of Status.engines, if it holds a non-success value.

        // Optional, excluded if all items would be 0. A missing item indicates a value of 0.
        incoming: {
          applied: <integer>, // Number of records applied
          succeeded: <integer>, // Number of records that applied without error
          failed: <integer>, // Number of records that failed to apply
          newFailed: <integer>, // Number of records that failed for the first time this sync
          reconciled: <integer>, // Number of records that were reconciled
        },

        // Optional, excluded if it would be empty. Records that would be
        // empty (e.g. 0 sent and 0 failed) are omitted.
        outgoing: [
          {
            sent: <integer>, // Number of outgoing records sent. Zero values are omitted.
            failed: <integer>, // Number that failed to send. Zero values are omitted.
          }
        ],
        // Optional, excluded if there were no errors
        failureReason: { ... }, // Same as above.

        // Optional, excluded if it would be empty or if the engine cannot
        // or did not run validation on itself. Entries with a count of 0
        // are excluded.
        validation: [
          {
            name: <string>, // The problem identified.
            count: <integer>, // Number of times it occurred.
          }
        ]
      }
    ]
  }
}
```

info

took

These values should be monotonic. If we can't get a monotonic timestamp, -1 will be reported on the payload, and the values will be omitted from the engines. Additionally, the value will be omitted from an engine if it would be 0 (either due to timer inaccuracy or finishing instantaneously).

uid

This property containing the FxA account identifier, which is provided by the FxA auth server APIs: https://github.com/mozilla/fxa-auth-server/blob/master/docs/api.md. It may be an empty string in the case that we

are unable to authenticate with FxA, and have never authenticated in the past. If present, it should be a 32 character hexidecimal string.

why

One of the following values:

- `startup`: This is the first sync triggered after browser startup.
- `schedule`: This is a sync triggered because it has been too long since the last sync.
- `score`: This sync is triggered by a high score value one of sync's trackers, indicating that many changes have occurred since the last sync.
- `user`: The user manually triggered the sync.
- `tabs`: The user opened the synced tabs sidebar, which triggers a sync.

status

The `engine.status`, `payload.status.sync`, and `payload.status.service` properties are sync error codes, which are listed in services/sync/modules/constants.js, and success values are not reported.

failureReason

Stores error information, if any is present. Always contains the "name" property, which identifies the type of error it is. The types can be.

- `httperror`: Indicates that we recieved an HTTP error response code, but are unable to be more specific about the error. Contains the following properties:
  - `code`: Integer HTTP status code.
- `nserror`: Indicates that an exception with the provided error code caused sync to fail.
  - `code`: The nsresult error code (integer).
- `shutdownerror`: Indicates that the sync failed because we shut down before completion.
- `autherror`: Indicates an unrecoverable authentication error.
  - `from`: Where the authentication error occurred, one of the following values: `tokenserver`, `fxaccounts`, or `hawkclient`.
- `othererror`: Indicates that it is a sync error code that we are unable to give more specific information on. As with the `syncStatus` property, it is a sync error code, which are listed in services/sync/modules/constants.js.
  - `error`: String identifying which error was present.
- `unexpectederror`: Indicates that some other error caused sync to fail, typically an uncaught exception.
  - `error`: The message provided by the error.

engine.name

Third-party engines are not reported, so only the following values are allowed: `addons`, `bookmarks`, `clients`, `forms`, `history`, `passwords`, `prefs`, and `tabs`.

engine.validation

For engines that can run validation on themselves, an array of objects describing validation errors that have occurred. Items that would have a count of 0 are excluded. Each engine will have its own set of items that it might put in the `name` field, but there are a finite number. See `BookmarkProblemData.getSummary` in services/sync/modules/bookmark_validator.js for an example.

14.3.9 "uitour-tag" ping

This ping is submitted via the UITour setTreatmentTag API. It may be used by the tour to record what settings were made by a user or to track the result of A/B experiments.

The client ID is submitted with this ping.

Structure:

```
{
  version: 1,
  type: "uitour-tag",
  clientId: <string>,
  payload: {
    tagName: <string>,
    tagValue: <string>
  }
}
```

See also: common ping fields

14.4 Internals

14.4.1 Preferences

Telemetry behaviour is controlled through the preferences listed here.

Default behaviors

Sending only happens on official builds (i.e. with `MOZILLA_OFFICIAL` set) with `MOZ_TELEMETRY_REPORTING` defined. All other builds drop all outgoing pings, so they will also not retry sending them later.

Preferences

`toolkit.telemetry.unified`

This controls whether unified behavior is enabled. If true:

- Telemetry is always enabled and recording *base* data.
- Telemetry will send additional `main` pings.

`toolkit.telemetry.enabled`

If `unified` is off, this controls whether the Telemetry module is enabled. If `unified` is on, this controls whether to record *extended* data. This preference is controlled through the *Preferences* dialog.

Note that the default value here of this pref depends on the define `RELEASE_BUILD` and the channel. If `RELEASE_BUILD` is set, `MOZ_TELEMETRY_ON_BY_DEFAULT` gets set, which means this pref will default to `true`. This is overridden by the preferences code on the "beta" channel, the pref also defaults to `true` there.

`datareporting.healthreport.uploadEnabled`

Send the data we record if user has consented to FHR. This preference is controlled through the *Preferences* dialog.

`toolkit.telemetry.archive.enabled`

Allow pings to be archived locally. This can only be enabled if `unified` is on.

`toolkit.telemetry.server`

The server Telemetry pings are sent to.

`toolkit.telemetry.log.level`

This sets the Telemetry logging verbosity per `Log.jsm`, with `Trace` or `0` being the most verbose and the default being `Warn`. By default logging goes only the console service.

`toolkit.telemetry.log.dump`

Sets whether to dump Telemetry log messages to `stdout` too.

Data-choices notification

`toolkit.telemetry.reportingpolicy.firstRun`

This preference is not present until the first run. After, its value is set to false. This is used to show the infobar with a more aggressive timeout if it wasn't shown yet.

`datareporting.policy.dataSubmissionEnabled`

This is the data submission master kill switch. If disabled, no policy is shown or upload takes place, ever.

`datareporting.policy.dataSubmissionPolicyNotifiedTime`

Records the date user was shown the policy. This preference is also used on Android.

`datareporting.policy.dataSubmissionPolicyAcceptedVersion`

Records the version of the policy notified to the user. This preference is also used on Android.

`datareporting.policy.dataSubmissionPolicyBypassNotification`

Used in tests, it allows to skip the notification check.

`datareporting.policy.currentPolicyVersion`

Stores the current policy version, overrides the default value defined in TelemetryReportingPolicy.jsm.

`datareporting.policy.minimumPolicyVersion`

The minimum policy version that is accepted for the current policy. This can be set per channel.

`datareporting.policy.minimumPolicyVersion.channel-NAME`

This is the only channel-specific version that we currently use for the minimum policy version.

Testing

The following prefs are for testing purpose only.

`toolkit.telemetry.initDelay`

Delay before initializing telemetry (seconds).

`toolkit.telemetry.minSubsessionLength`

Minimum length of a telemetry subsession (seconds).

`toolkit.telemetry.collectInterval`

Minimum interval between data collection (seconds).

`toolkit.telemetry.scheduler.tickInterval`

Interval between scheduler ticks (seconds).

`toolkit.telemetry.scheduler.idleTickInterval`

Interval between scheduler ticks when the user is idle (seconds).

`toolkit.telemetry.idleTimeout`

Timeout until we decide whether a user is idle or not (seconds).

14.5 Firefox Health Report (Obsolete)

Firefox Health Report (FHR) is obsolete and no longer ships with Firefox. This documentation will live here for a few more cycles.

Firefox Health Report is a background service that collects application metrics and periodically submits them to a central server. The core parts of the service are implemented in this directory. However, the actual XPCOM service is implemented in the ‘‘data_reporting_service‘.

The core types can actually be instantiated multiple times and used to power multiple data submission services within a single Gecko application. In other words, everything in this directory is effectively a reusable library. However, the terminology and some of the features are very specific to what the Firefox Health Report feature requires.

14.5.1 Architecture

`healthreporter.jsm` contains the main interface for FHR, the `HealthReporter` type. An instance of this is created by the ‘‘data_reporting_service‘.

`providers.jsm` contains numerous `Metrics.Provider` and `Metrics.Measurement` used for collecting application metrics. If you are looking for the FHR probes, this is where they are.

Storage

Firefox Health Report stores data in 3 locations:

- Metrics measurements and provider state is stored in a SQLite database (via `Metrics.Storage`).
- Service state (such as the IDs of documents uploaded) is stored in a JSON file on disk (via OS.File).
- Lesser state and run-time options are stored in preferences.

Preferences

Preferences controlling behavior of Firefox Health Report live in the `datareporting.healthreport.*` branch.

Service and Data Control

The follow preferences control behavior of the service and data upload.

service.enabled Controls whether the entire health report service runs. The overall service performs data collection, storing, and submission.

This is the primary kill switch for Firefox Health Report outside of the build system variable. i.e. if you are using an official Firefox build and wish to disable FHR, this is what you should set to false to prevent FHR from not only submitting but also collecting data.

uploadEnabled Whether uploading of data is enabled. This is the preference the checkbox in the preferences UI reflects. If this is disabled, FHR still collects data - it just doesn't upload it.

service.loadDelayMsec How long (in milliseconds) after initial application start should FHR wait before initializing.

FHR may initialize sooner than this if the FHR service is requested. This will happen if e.g. the user goes to `about:healthreport`.

service.loadDelayFirstRunMsec How long (in milliseconds) FHR should wait to initialize on first application run.

FHR waits longer than normal to initialize on first application run because first-time initialization can use a lot of I/O to initialize the SQLite database and this I/O should not interfere with the first-run user experience.

documentServerURI The URI of a Bagheera server that FHR should interface with for submitting documents.

You typically do not need to change this.

documentServerNamespace The namespace on the document server FHR should upload documents to.

You typically do not need to change this.

infoURL The URL of a page containing more info about FHR, it's privacy policy, etc.

about.reportUrl The URL to load in `about:healthreport`.

about.reportUrlUnified The URL to load in `about:healthreport`. This is used instead of `reportUrl` for UnifiedTelemetry when it is not opt-in.

service.providerCategories A comma-delimited list of category manager categories that contain registered `Metrics.Provider` records. Read below for how provider registration works.

If the entire service is disabled, you lose data collection. This means that **local** data analysis won't be available because there is no data to analyze! Keep in mind that Firefox Health Report can be useful even if it's not submitting data to remote servers!

Logging

The following preferences allow you to control the logging behavior of Firefox Health Report.

logging.consoleEnabled Whether to write log messages to the web console. This is true by default.

logging.consoleLevel The minimum log level FHR messages must have to be written to the web console. By default, only FHR warnings or errors will be written to the web console. During normal/expected operation, no messages of this type should be produced.

logging.dumpEnabled Whether to write log messages via `dump()`. If true, FHR will write messages to stdout/stderr.

This is typically only enabled when developing FHR.

logging.dumpLevel The minimum log level messages must have to be written via `dump()`.

State

currentDaySubmissionFailureCount How many submission failures the client has encountered while attempting to upload the most recent document.

lastDataSubmissionFailureTime The time of the last failed document upload.

lastDataSubmissionRequestedTime The time of the last document upload attempt.

lastDataSubmissionSuccessfulTime The time of the last successful document upload.

nextDataSubmissionTime The time the next data submission is scheduled for. FHR will not attempt to upload a new document before this time.

pendingDeleteRemoteData Whether the client currently has a pending request to delete remote data. If true, the client will attempt to delete all remote data before an upload is performed.

FHR stores various state in preferences.

Registering Providers

Firefox Health Report providers are registered via the category manager. See `HealthReportComponents.manifest` for providers defined in this directory.

Essentially, the category manager receives the name of a JS type and the URI of a JSM to import that exports this symbol. At run-time, the providers registered in the category manager are instantiated.

Providers are registered via the category manager to make registration simple and less prone to errors. Any XPCOM component can create a category manager entry. Therefore, new data providers can be added without having to touch core Firefox Health Report code. Additionally, category manager registration means providers are more likely to be registered on FHR's terms, when it wants. If providers were registered in code at application run-time, there would be the risk of other components prematurely instantiating FHR (causing a performance hit if performed at an inopportune time) or semi-complicated code around observers or listeners. Category manager entries are only 1 line per provider and leave FHR in control: they are simple and safe.

Document Generation and Lifecycle

FHR will attempt to submit a JSON document containing data every 24 wall clock hours.

At upload time, FHR will query the database for **all** information from the last 180 days and assemble this data into a JSON document. We attempt to upload this JSON document with a client-generated UUID to the configured server.

Before we attempt upload, the generated UUID is stored in the JSON state file on local disk. At this point, the client assumes the document with that UUID has been successfully stored on the server.

If the client is aware of other document UUIDs that presumably exist on the server, those UUIDs are sent with the upload request so the client can request those UUIDs be deleted. This helps ensure that each client only has 1 document/UUID on the server at any one time.

Importance of Persisting UUIDs

The choices of how, where, and when document UUIDs are stored and updated are very important. One should not attempt to change things unless she has a very detailed understanding of why things are the way they are.

The client is purposefully very conservative about forgetting about generated UUIDs. In other words, once a UUID is generated, the client deliberately holds on to that UUID until it's very confident that UUID is no longer stored on the server. The reason we do this is because *orphaned* documents/UUIDs on the server can lead to faulty analysis, such as over-reporting the number of Firefox installs that stop being used.

When uploading a new UUID, we update the state and save the state file to disk *before* an upload attempt because if the upload succeeds but the response never makes it back to the client, we want the client to know about the uploaded UUID so it can delete it later to prevent an orphan.

We maintain a list of UUIDs locally (not simply the last UUID) because multiple upload attempts could fail the same way as the previous paragraph describes and we have no way of knowing which (if any) actually succeeded. The safest approach is to assume every document produced managed to get uploaded some how.

We store the UUIDs on a file on disk and not anywhere else because we want storage to be robust. We originally stored UUIDs in preferences, which only flush to disk periodically. Writes to preferences were apparently getting lost. We switched to writing directly to files to eliminate this window.

14.5.2 Payload Format

Currently, the Firefox Health Report is submitted as a compressed JSON document. The root JSON element is an object. A *version* field defines the version of the payload which in turn defines the expected contents the object.

As of 2013-07-03, desktop submits Version 2, and Firefox for Android submits Version 3 payloads.

Version 3

Version 3 is a complete rebuild of the document format. Events are tracked in an "environment". Environments are computed from a large swath of local data (e.g., add-ons, CPU count, versions), and a new environment comes into being when one of its attributes changes.

Client documents, then, will include descriptions of many environments, and measurements will be attributed to one particular environment.

A map of environments is present at the top level of the document, with the current named "current" in the map. Each environment has a hash identifier and a set of attributes. The current environment is completely described, and has its hash present in a "hash" attribute. All other environments are represented as a tree diff from the current environment, with their hash as the key in the "environments" object.

A removed add-on has the value 'null'.

There is no "last" data at present.

Daily data is hierarchical: by day, then by environment, and then by measurement, and is present in "data", just as in v2.

Leading by example:

```
{
  "lastPingDate": "2013-06-29",
  "thisPingDate": "2013-07-03",
  "version": 3,
  "environments": {
    "current": {
```

```
    "org.mozilla.sysinfo.sysinfo": {
      "memoryMB": 1567,
      "cpuCount": 4,
      "architecture": "armeabi-v7a",
      "_v": 1,
      "version": "4.1.2",
      "name": "Android"
    },
    "org.mozilla.profile.age": {
      "_v": 1,
      "profileCreation": 15827
    },
    "org.mozilla.addons.active": {
      "QuitNow@TWiGSoftware.com": {
        "appDisabled": false,
        "userDisabled": false,
        "scope": 1,
        "updateDay": 15885,
        "foreignInstall": false,
        "hasBinaryComponents": false,
        "blocklistState": 0,
        "type": "extension",
        "installDay": 15885,
        "version": "1.18.02"
      },
      "{dbbf9331-b713-6eda-1006-205efead09dc}": {
        "appDisabled": false,
        "userDisabled": "askToActivate",
        "scope": 8,
        "updateDay": 15779,
        "foreignInstall": true,
        "blocklistState": 0,
        "type": "plugin",
        "installDay": 15779,
        "version": "11.1 r115"
      },
      "desktopbydefault@bnicholson.mozilla.org": {
        "appDisabled": false,
        "userDisabled": true,
        "scope": 1,
        "updateDay": 15870,
        "foreignInstall": false,
        "hasBinaryComponents": false,
        "blocklistState": 0,
        "type": "extension",
        "installDay": 15870,
        "version": "1.1"
      },
      "{6e092a7f-ba58-4abb-88c1-1a4e50b217e4}": {
        "appDisabled": false,
        "userDisabled": false,
        "scope": 1,
        "updateDay": 15828,
        "foreignInstall": false,
        "hasBinaryComponents": false,
        "blocklistState": 0,
        "type": "extension",
        "installDay": 15828,
```

```
        "version": "1.1.0"
      },
      "{46551EC9-40F0-4e47-8E18-8E5CF550CFB8}": {
        "appDisabled": false,
        "userDisabled": true,
        "scope": 1,
        "updateDay": 15879,
        "foreignInstall": false,
        "hasBinaryComponents": false,
        "blocklistState": 0,
        "type": "extension",
        "installDay": 15879,
        "version": "1.3.2"
      },
      "_v": 1
    },
    "org.mozilla.appInfo.appinfo": {
      "_v": 3,
      "appLocale": "en_us",
      "osLocale": "en_us",
      "distribution": "",
      "acceptLangIsUserSet": 0,
      "isTelemetryEnabled": 1,
      "isBlocklistEnabled": 1
    },
    "geckoAppInfo": {
      "updateChannel": "nightly",
      "id": "{aa3c5121-dab2-40e2-81ca-7ea25febc110}",
      "os": "Android",
      "platformBuildID": "20130703031323",
      "platformVersion": "25.0a1",
      "vendor": "Mozilla",
      "name": "fennec",
      "xpcomabi": "arm-eabi-gcc3",
      "appBuildID": "20130703031323",
      "_v": 1,
      "version": "25.0a1"
    },
    "hash": "tB4Pnnep9yTxnMDymc3dAB2RRB0=",
    "org.mozilla.addons.counts": {
      "extension": 4,
      "plugin": 1,
      "_v": 1,
      "theme": 0
    }
  },
  "k2O3hlreMeS7L1qtxeMsYWxgWWQ=": {
    "geckoAppInfo": {
      "platformBuildID": "20130630031138",
      "appBuildID": "20130630031138",
      "_v": 1
    },
    "org.mozilla.appInfo.appinfo": {
      "_v": 2,
    }
  },
  "1+KN9TutMpzdl4TJEl+aCxK+xcw=": {
    "geckoAppInfo": {
```

```
        "platformBuildID": "20130626031100",
        "appBuildID": "20130626031100",
        "_v": 1
      },
      "org.mozilla.addons.active": {
        "QuitNow@TWiGSoftware.com": null,
        "{dbbf9331-b713-6eda-1006-205efead09dc}": null,
        "desktopbydefault@bnicholson.mozilla.org": null,
        "{6e092a7f-ba58-4abb-88c1-1a4e50b217e4}": null,
        "{46551EC9-40F0-4e47-8E18-8E5CF550CFB8}": null,
        "_v": 1
      },
      "org.mozilla.addons.counts": {
        "extension": 0,
        "plugin": 0,
        "_v": 1
      }
    }
  },
  "data": {
    "last": {},
    "days": {
      "2013-07-03": {
        "tB4Pnnep9yTxnMDymc3dAB2RRB0=": {
          "org.mozilla.appSessions": {
            "normal": [
              {
                "r": "P",
                "d": 2,
                "sj": 653
              },
              {
                "r": "P",
                "d": 22
              },
              {
                "r": "P",
                "d": 5
              },
              {
                "r": "P",
                "d": 0
              },
              {
                "r": "P",
                "sg": 3560,
                "d": 171,
                "sj": 518
              },
              {
                "r": "P",
                "d": 16
              },
              {
                "r": "P",
                "d": 1079
              }
            ],
```

```
            "_v": "4"
          }
        },
        "k2O3hlreMeS7L1qtxeMsYWxgWWQ=": {
          "org.mozilla.appSessions": {
            "normal": [
              {
                "r": "P",
                "d": 27
              },
              {
                "r": "P",
                "d": 19
              },
              {
                "r": "P",
                "d": 55
              }
            ],
            "_v": "4"
          },
          "org.mozilla.searches.counts": {
            "bartext": {
              "google": 1
            },
            "_v": "4"
          },
          "org.mozilla.experiment": {
            "lastActive": "some.experiment.id"
            "_v": "1"
          }
        }
      }
    }
  }
}
```

App sessions in Version 3

Sessions are divided into "normal" and "abnormal". Session objects are stored as discrete JSON:

```
"org.mozilla.appSessions": {
  _v: 4,
  "normal": [
    {"r":"P", "d": 123},
  ],
  "abnormal": [
    {"r":"A", "oom": true, "stopped": false}
  ]
}
```

Keys are:

"r" reason. Values are "P" (activity paused), "A" (abnormal termination).

"d" duration. Value in seconds.

"sg" Gecko startup time (msec). Present if this is a clean launch. This corresponds to the telemetry timer *FENNEC_STARTUP_TIME_GECKOREADY*.

"sj" Java activity init time (msec). Present if this is a clean launch. This corresponds to the telemetry timer *FENNEC_STARTUP_TIME_JAVAUI*, and includes initialization tasks beyond initial *onWindowFocusChanged*.

Abnormal terminations will be missing a duration and will feature these keys:

"oom" was the session killed by an OOM exception?

"stopped" was the session stopped gently?

Version 3.2

As of Firefox 35, the search counts measurement is now bumped to v6, including the *activity* location for the search activity.

Version 3.1

As of Firefox 27, *appinfo* is now bumped to v3, including *osLocale*, *appLocale* (currently always the same as *osLocale*), *distribution* (a string containing the distribution ID and version, separated by a colon), and *acceptLangIsUserSet*, an integer-boolean that describes whether the user set an *intl.accept_languages* preference.

The search counts measurement is now at version 5, which indicates that non-partner searches are recorded. You'll see identifiers like "other-Foo Bar" rather than "other".

Version 3.2

In Firefox 32, Firefox for Android includes a device configuration section in the environment description:

```
"org.mozilla.device.config": {
  "hasHardwareKeyboard": false,
  "screenXInMM": 58,
  "screenLayout": 2,
  "uiType": "default",
  "screenYInMM": 103,
  "_v": 1,
  "uiMode": 1
}
```

Of these, the only keys that need explanation are:

uiType One of "default", "smalltablet", "largetablet".

uiMode A mask of the Android *Configuration.uiMode* value, e.g., *UI_MODE_TYPE_CAR*.

screenLayout A mask of the Android *Configuration.screenLayout* value. One of the *SCREENLAYOUT_SIZE_* constants.

Note that screen dimensions can be incorrect due to device inaccuracies and platform limitations.

Other notable differences from Version 2

- There is no default browser indicator on Android.
- Add-ons include a *blocklistState* attribute, as returned by AddonManager.

- Searches are now version 4, and are hierarchical: how the search was started (bartext, barkeyword, barsuggest), and then counts per provider.

Version 2

Version 2 is the same as version 1 with the exception that it has an additional top-level field, *geckoAppInfo*, which contains basic application info.

geckoAppInfo

This field is an object that is a simple map of string keys and values describing basic application metadata. It is very similar to the *appinfo* measurement in the *last* section. The difference is this field is almost certainly guaranteed to exist whereas the one in the data part of the payload may be omitted in certain scenarios (such as catastrophic client error).

Its keys are as follows:

appBuildID The build ID/date of the application. e.g. "20130314113542".

version The value of nsXREAppData.version. This is the application's version. e.g. "21.0.0".

vendor The value of nsXREAppData.vendor. Can be empty an empty string. For official Mozilla builds, this will be "Mozilla".

name The value of nsXREAppData.name. For official Firefox builds, this will be "Firefox".

id The value of nsXREAppData.ID.

platformVersion The version of the Gecko platform (as opposed to the app version). For Firefox, this is almost certainly equivalent to the *version* field.

platformBuildID The build ID/date of the Gecko platfor (as opposed to the app version). This is commonly equivalent to *appBuildID*.

os The name of the operating system the application is running on.

xpcomabi The binary architecture of the build.

updateChannel The name of the channel used for application updates. Official Mozilla builds have one of the values {release, beta, aurora, nightly}. Local and test builds have *default* as the channel.

Version 1

Top-level Properties

The main JSON object contains the following properties:

lastPingDate UTC date of the last upload. If this is the first upload from this client, this will not be present.

thisPingDate UTC date when this payload was constructed.

version Integer version of this payload format. Currently only 1 is defined.

clientID An identifier that identifies the client that is submitting data.

This property may not be present in older clients.

See *Identifiers* for more info on identifiers.

clientIDVersion Integer version associated with the generation semantics for the `clientID`.

If the value is `1`, `clientID` is a randomly-generated UUID.

This property may not be present in older clients.

data Object holding data constituting health report.

Data Properties

The bulk of the health report is contained within the *data* object. This object has the following keys:

days Object mapping UTC days to measurements from that day. Keys are in the *YYYY-MM-DD* format. e.g. "2013-03-14"

last Object mapping measurement names to their values.

The value of *days* and *last* are objects mapping measurement names to that measurement's values. The values are always objects. Each object contains a *_v* property. This property defines the version of this measurement. Additional non-underscore-prefixed properties are defined by the measurement itself (see sections below).

Example

Here is an example JSON document for version 1:

```
{
  "version": 1,
  "thisPingDate": "2013-03-11",
  "lastPingDate": "2013-03-10",
  "data": {
    "last": {
      "org.mozilla.addons.active": {
        "masspasswordreset@johnathan.nightingale": {
          "userDisabled": false,
          "appDisabled": false,
          "version": "1.05",
          "type": "extension",
          "scope": 1,
          "foreignInstall": false,
          "hasBinaryComponents": false,
          "installDay": 14973,
          "updateDay": 15317
        },
        "places-maintenance@bonardo.net": {
          "userDisabled": false,
          "appDisabled": false,
          "version": "1.3",
          "type": "extension",
          "scope": 1,
          "foreignInstall": false,
          "hasBinaryComponents": false,
          "installDay": 15268,
          "updateDay": 15379
        },
        "_v": 1
      },
      "org.mozilla.appInfo.appinfo": {
        "_v": 1,
```

```
        "appBuildID": "20130309030841",
        "distributionID": "",
        "distributionVersion": "",
        "hotfixVersion": "",
        "id": "{ec8030f7-c20a-464f-9b0e-13a3a9e97384}",
        "locale": "en-US",
        "name": "Firefox",
        "os": "Darwin",
        "platformBuildID": "20130309030841",
        "platformVersion": "22.0a1",
        "updateChannel": "nightly",
        "vendor": "Mozilla",
        "version": "22.0a1",
        "xpcomabi": "x86_64-gcc3"
      },
      "org.mozilla.profile.age": {
        "_v": 1,
        "profileCreation": 12444
      },
      "org.mozilla.appSessions.current": {
        "_v": 3,
        "startDay": 15773,
        "activeTicks": 522,
        "totalTime": 70858,
        "main": 1245,
        "firstPaint": 2695,
        "sessionRestored": 3436
      },
      "org.mozilla.sysinfo.sysinfo": {
        "_v": 1,
        "cpuCount": 8,
        "memoryMB": 16384,
        "architecture": "x86-64",
        "name": "Darwin",
        "version": "12.2.1"
      }
    },
    "days": {
      "2013-03-11": {
        "org.mozilla.addons.counts": {
          "_v": 1,
          "extension": 15,
          "plugin": 12,
          "theme": 1
        },
        "org.mozilla.places.places": {
          "_v": 1,
          "bookmarks": 757,
          "pages": 104858
        },
        "org.mozilla.appInfo.appinfo": {
          "_v": 1,
          "isDefaultBrowser": 1
        }
      },
      "2013-03-10": {
        "org.mozilla.addons.counts": {
          "_v": 1,
```

```
          "extension": 15,
          "plugin": 12,
          "theme": 1
        },
        "org.mozilla.places.places": {
          "_v": 1,
          "bookmarks": 757,
          "pages": 104857
        },
        "org.mozilla.searches.counts": {
          "_v": 1,
          "google.urlbar": 4
        },
        "org.mozilla.appInfo.appinfo": {
          "_v": 1,
          "isDefaultBrowser": 1
        }
      }
    }
  }
}
```

Measurements

The bulk of payloads consists of measurement data. An individual measurement is merely a collection of related values e.g. *statistics about the Places database* or *system information*.

Each measurement has an integer version number attached. When the fields in a measurement or the semantics of data within that measurement change, the version number is incremented.

All measurements are defined alphabetically in the sections below.

org.mozilla.addons.addons

This measurement contains information about the currently-installed add-ons.

Version 2 This version adds the human-readable fields *name* and *description*, both coming directly from the Addon instance as most properties in version 1. Also, all plugin details are now in org.mozilla.addons.plugins.

Version 1 The measurement object is a mapping of add-on IDs to objects containing add-on metadata.

Each add-on contains the following properties:

- userDisabled
- appDisabled
- version
- type
- scope
- foreignInstall
- hasBinaryComponents
- installDay

- updateDay

With the exception of *installDay* and *updateDay*, all these properties come direct from the Addon instance. See https://developer.mozilla.org/en-US/docs/Addons/Add-on_Manager/Addon. *installDay* and *updateDay* are the number of days since UNIX epoch of the add-ons *installDate* and *updateDate* properties, respectively.

Notes Add-ons that have opted out of AMO updates via the *extensions._id_.getAddons.cache.enabled* preference are, since Bug 868306 (Firefox 24), included in the list of submitted add-ons.

Example

```
"org.mozilla.addons.addons": {
  "_v": 2,
  "{d10d0bf8-f5b5-c8b4-a8b2-2b9879e08c5d}": {
    "userDisabled": false,
    "appDisabled": false,
    "name": "Adblock Plus",
    "version": "2.4.1",
    "type": "extension",
    "scope": 1,
    "description": "Ads were yesterday!",
    "foreignInstall": false,
    "hasBinaryComponents": false,
    "installDay": 16093,
    "updateDay": 16093
  },
  "{e4a8a97b-f2ed-450b-b12d-ee082ba24781}": {
    "userDisabled": true,
    "appDisabled": false,
    "name": "Greasemonkey",
    "version": "1.14",
    "type": "extension",
    "scope": 1,
    "description": "A User Script Manager for Firefox",
    "foreignInstall": false,
    "hasBinaryComponents": false,
    "installDay": 16093,
    "updateDay": 16093
  }
}
```

org.mozilla.addons.plugins

This measurement contains information about the currently-installed plugins.

Version 1 The measurement object is a mapping of plugin IDs to objects containing plugin metadata.

The plugin ID is constructed of the plugins filename, name, version and description. Every plugin has at least a filename and a name.

Each plugin contains the following properties:

- name
- version
- description

- blocklisted
- disabled
- clicktoplay
- mimeTypes
- updateDay

With the exception of *updateDay* and *mimeTypes*, all these properties come directly from `nsIPluginTag` via `nsIPluginHost`. *updateDay* is the number of days since UNIX epoch of the plugins last modified time. *mimeTypes* is the list of mimetypes the plugin supports, see `nsIPluginTag.getMimeTypes()`.

Example

```
"org.mozilla.addons.plugins": {
  "_v": 1,
  "Flash Player.plugin:Shockwave Flash:12.0.0.38:Shockwave Flash 12.0 r0": {
    "mimeTypes": [
      "application/x-shockwave-flash",
      "application/futuresplash"
    ],
    "name": "Shockwave Flash",
    "version": "12.0.0.38",
    "description": "Shockwave Flash 12.0 r0",
    "blocklisted": false,
    "disabled": false,
    "clicktoplay": false
  },
  "Default Browser.plugin:Default Browser Helper:537:Provides information about the default web brows
    "mimeTypes": [
      "application/apple-default-browser"
    ],
    "name": "Default Browser Helper",
    "version": "537",
    "description": "Provides information about the default web browser",
    "blocklisted": false,
    "disabled": true,
    "clicktoplay": false
  }
}
```

org.mozilla.addons.counts

This measurement contains information about historical add-on counts.

Version 1 The measurement object consists of counts of different add-on types. The properties are:

extension Integer count of installed extensions.

plugin Integer count of installed plugins.

theme Integer count of installed themes.

lwtheme Integer count of installed lightweigh themes.

Notes Add-ons opted out of AMO updates are included in the counts. This differs from the behavior of the active add-ons measurement.

If no add-ons of a particular type are installed, the property for that type will not be present (as opposed to an explicit property with value of 0).

Example

```
"2013-03-14": {
  "org.mozilla.addons.counts": {
    "_v": 1,
    "extension": 21,
    "plugin": 4,
    "theme": 1
  }
}
```

org.mozilla.appInfo.appinfo

This measurement contains basic XUL application and Gecko platform information. It is reported in the *last* section.

Version 2 In addition to fields present in version 1, this version has the following fields appearing in the *days* section:

isBlocklistEnabled Whether the blocklist ping is enabled. This is an integer, 0 or 1. This does not indicate whether the blocklist ping was sent but merely whether the application will try to send the blocklist ping.

isTelemetryEnabled Whether Telemetry is enabled. This is an integer, 0 or 1.

Version 1 The measurement object contains mostly string values describing the current application and build. The properties are:

- vendor
- name
- id
- version
- appBuildID
- platformVersion
- platformBuildID
- os
- xpcomabi
- updateChannel
- distributionID
- distributionVersion
- hotfixVersion
- locale
- isDefaultBrowser

Notes All of the properties appear in the *last* section except for *isDefaultBrowser*, which appears under *days*.

Example This example comes from an official OS X Nightly build:

```
"org.mozilla.appInfo.appinfo": {
  "_v": 1,
  "appBuildID": "20130311030946",
  "distributionID": "",
  "distributionVersion": "",
  "hotfixVersion": "",
  "id": "{ec8030f7-c20a-464f-9b0e-13a3a9e97384}",
  "locale": "en-US",
  "name": "Firefox",
  "os": "Darwin",
  "platformBuildID": "20130311030946",
  "platformVersion": "22.0a1",
  "updateChannel": "nightly",
  "vendor": "Mozilla",
  "version": "22.0a1",
  "xpcomabi": "x86_64-gcc3"
},
```

org.mozilla.appInfo.update

This measurement contains information about the application update mechanism in the application.

Version 1 The following daily values are reported:

enabled Whether automatic application update checking is enabled. 1 for yes, 0 for no.

autoDownload Whether automatic download of available updates is enabled.

Notes This measurement was merged to mozilla-central for JS FHR on 2013-07-15.

Example

```
"2013-07-15": {
  "org.mozilla.appInfo.update": {
    "_v": 1,
    "enabled": 1,
    "autoDownload": 1,
  }
}
```

org.mozilla.appInfo.versions

This measurement contains a history of application version numbers.

Version 2 Version 2 reports more fields than version 1 and is not backwards compatible. The following fields are present in version 2:

appVersion An array of application version strings.

appBuildID An array of application build ID strings.

platformVersion An array of platform version strings.

platformBuildID An array of platform build ID strings.

When the application is upgraded, the new version and/or build IDs are appended to their appropriate fields.

Version 1 When the application version (*version* from *org.mozilla.appinfo.appinfo*) changes, we record the new version on the day the change was seen. The new versions for a day are recorded in an array under the *version* property.

Notes If the application isn't upgraded, this measurement will not be present. This means this measurement will not be present for most days if a user is on the release channel (since updates are typically released every 6 weeks). However, users on the Nightly and Aurora channels will likely have a lot of these entries since those builds are updated every day.

Values for this measurement are collected when performing the daily collection (typically occurs at upload time). As a result, it's possible the actual upgrade day may not be attributed to the proper day - the reported day may lag behind.

The app and platform versions and build IDs should be identical for most clients. If they are different, we are possibly looking at a *Frankenfox*.

Example

```
"2013-03-27": {
  "org.mozilla.appInfo.versions": {
    "_v": 2,
    "appVersion": [
       "22.0.0"
    ],
    "appBuildID": [
      "20130325031100"
    ],
    "platformVersion": [
      "22.0.0"
    ],
    "platformBuildID": [
      "20130325031100"
    ]
  }
}
```

org.mozilla.appSessions.current

This measurement contains information about the currently running XUL application's session.

Version 3 This measurement has the following properties:

startDay Integer days since UNIX epoch when this session began.

activeTicks Integer count of *ticks* the session was active for. Gecko periodically sends out a signal when the session is active. Session activity involves keyboard or mouse interaction with the application. Each tick represents a window of 5 seconds where there was interaction.

totalTime Integer seconds the session has been alive.

main Integer milliseconds it took for the Gecko process to start up.

firstPaint Integer milliseconds from process start to first paint.

sessionRestored Integer milliseconds from process start to session restore.

Example

```
"org.mozilla.appSessions.current": {
  "_v": 3,
  "startDay": 15775,
  "activeTicks": 4282,
  "totalTime": 249422,
  "main": 851,
  "firstPaint": 3271,
  "sessionRestored": 5998
}
```

org.mozilla.appSessions.previous

This measurement contains information about previous XUL application sessions.

Version 3 This measurement contains per-day lists of all the sessions started on that day. The following properties may be present:

cleanActiveTicks Active ticks of sessions that were properly shut down.

cleanTotalTime Total number of seconds for sessions that were properly shut down.

abortedActiveTicks Active ticks of sessions that were not properly shut down.

abortedTotalTime Total number of seconds for sessions that were not properly shut down.

main Time in milliseconds from process start to main process initialization.

firstPaint Time in milliseconds from process start to first paint.

sessionRestored Time in milliseconds from process start to session restore.

Notes Sessions are recorded on the date on which they began.

If a session was aborted/crashed, the total time may be less than the actual total time. This is because we don't always update total time during periods of inactivity and the abort/crash could occur after a long period of idle, before we've updated the total time.

The lengths of the arrays for {cleanActiveTicks, cleanTotalTime}, {abortedActiveTicks, abortedTotalTime}, and {main, firstPaint, sessionRestored} should all be identical.

The length of the clean sessions plus the length of the aborted sessions should be equal to the length of the {main, firstPaint, sessionRestored} properties.

It is not possible to distinguish the main, firstPaint, and sessionRestored values from a clean vs aborted session: they are all lumped together.

For sessions spanning multiple UTC days, it's not possible to know which days the session was active for. It's possible a week long session only had activity for 2 days and there's no way for us to tell which days.

Example

```
"org.mozilla.appSessions.previous": {
  "_v": 3,
  "cleanActiveTicks": [
    78,
    1785
  ],
  "cleanTotalTime": [
    4472,
    88908
  ],
  "main": [
    32,
    952
  ],
  "firstPaint": [
    2755,
    3497
  ],
  "sessionRestored": [
    5149,
    5520
  ]
}
```

org.mozilla.crashes.crashes

This measurement contains a historical record of application crashes.

Version 6 This version adds tracking for out-of-memory (OOM) crashes in the main process. An OOM crash will be counted as both main-crash and main-crash-oom.

This measurement will be reported on each day there was a crash or crash submission. Records may contain the following fields, whose values indicate the number of crashes, hangs, or submissions that occurred on the given day:

- content-crash
- content-crash-submission-succeeded
- content-crash-submission-failed
- content-hang
- content-hang-submission-succeeded
- content-hang-submission-failed
- gmplugin-crash
- gmplugin-crash-submission-succeeded
- gmplugin-crash-submission-failed
- main-crash
- main-crash-oom
- main-crash-submission-succeeded
- main-crash-submission-failed

- main-hang
- main-hang-submission-succeeded
- main-hang-submission-failed
- plugin-crash
- plugin-crash-submission-succeeded
- plugin-crash-submission-failed
- plugin-hang
- plugin-hang-submission-succeeded
- plugin-hang-submission-failed

Version 5 This version adds support for Gecko media plugin (GMP) crashes.

This measurement will be reported on each day there was a crash or crash submission. Records may contain the following fields, whose values indicate the number of crashes, hangs, or submissions that occurred on the given day:

- content-crash
- content-crash-submission-succeeded
- content-crash-submission-failed
- content-hang
- content-hang-submission-succeeded
- content-hang-submission-failed
- gmplugin-crash
- gmplugin-crash-submission-succeeded
- gmplugin-crash-submission-failed
- main-crash
- main-crash-submission-succeeded
- main-crash-submission-failed
- main-hang
- main-hang-submission-succeeded
- main-hang-submission-failed
- plugin-crash
- plugin-crash-submission-succeeded
- plugin-crash-submission-failed
- plugin-hang
- plugin-hang-submission-succeeded
- plugin-hang-submission-failed

Version 4 This version follows up from version 3, adding submissions which are now tracked by the *Crash Manager*.

This measurement will be reported on each day there was a crash or crash submission. Records may contain the following fields, whose values indicate the number of crashes, hangs, or submissions that occurred on the given day:

- main-crash
- main-crash-submission-succeeded
- main-crash-submission-failed
- main-hang
- main-hang-submission-succeeded
- main-hang-submission-failed
- content-crash
- content-crash-submission-succeeded
- content-crash-submission-failed
- content-hang
- content-hang-submission-succeeded
- content-hang-submission-failed
- plugin-crash
- plugin-crash-submission-succeeded
- plugin-crash-submission-failed
- plugin-hang
- plugin-hang-submission-succeeded
- plugin-hang-submission-failed

Version 3 This version follows up from version 2, building on improvements to the *Crash Manager*.

This measurement will be reported on each day there was a crash. Records may contain the following fields, whose values indicate the number of crashes or hangs that occurred on the given day:

- main-crash
- main-hang
- content-crash
- content-hang
- plugin-crash
- plugin-hang

Version 2 The switch to version 2 coincides with the introduction of the *Crash Manager*, which provides a more robust source of crash data.

This measurement will be reported on each day there was a crash. The following fields may be present in each record:

mainCrash The number of main process crashes that occurred on the given day.

Yes, version 2 does not track submissions like version 1. It is very likely submissions will be re-added later.

Also absent from version 2 are plugin crashes and hangs. These will be re-added, likely in version 3.

Version 1 This measurement will be reported on each day there was a crash. The following properties are reported:

pending The number of crash reports that haven't been submitted.

submitted The number of crash reports that were submitted.

Notes Main process crashes are typically submitted immediately after they occur (by checking a box in the crash reporter, which should appear automatically after a crash). If the crash reporter submits the crash successfully, we get a submitted crash. Else, we leave it as pending.

A pending crash does not mean it will eventually be submitted.

Pending crash reports can be submitted post-crash by going to about:crashes.

If a pending crash is submitted via about:crashes, the submitted count increments but the pending count does not decrement. This is because FHR does not know which pending crash was just submitted and therefore it does not know which day's pending crash to decrement.

Example

```
"org.mozilla.crashes.crashes": {
  "_v": 1,
  "pending": 1,
  "submitted": 2
},
"org.mozilla.crashes.crashes": {
  "_v": 2,
  "mainCrash": 2
}
"org.mozilla.crashes.crashes": {
  "_v": 4,
  "main-crash": 2,
  "main-crash-submission-succeeded": 1,
  "main-crash-submission-failed": 1,
  "main-hang": 1,
  "plugin-crash": 2
}
```

org.mozilla.healthreport.submissions

This measurement contains a history of FHR's own data submission activity. It was added in Firefox 23 in early May 2013.

Version 2 This is the same as version 1 except an additional field has been added.

uploadAlreadyInProgress A request for upload was initiated while another upload was in progress. This should not occur in well-behaving clients. It (along with a lock preventing simultaneous upload) was added to ensure this never occurs.

Version 1 Daily counts of upload events are recorded.

firstDocumentUploadAttempt An attempt was made to upload the client's first document to the server. These are uploads where the client is not aware of a previous document ID on the server. Unless the client had disabled upload, there should be at most one of these in the history of the client.

continuationUploadAttempt An attempt was made to upload a document that replaces an existing document on the server. Most upload attempts should be attributed to this as opposed to *firstDocumentUploadAttempt*.

uploadSuccess The upload attempt recorded by *firstDocumentUploadAttempt* or *continuationUploadAttempt* was successful.

uploadTransportFailure An upload attempt failed due to transport failure (network unavailable, etc).

uploadServerFailure An upload attempt failed due to a server-reported failure. Ideally these are failures reported by the FHR server itself. However, intermediate proxies, firewalls, etc may trigger this depending on how things are configured.

uploadClientFailure An upload attempt failued due to an error/exception in the client. This almost certainly points to a bug in the client.

The result for an upload attempt is always attributed to the same day as the attempt, even if the result occurred on a different day from the attempt. Therefore, the sum of the result counts should equal the result of the attempt counts.

org.mozilla.hotfix.update

This measurement contains results from the Firefox update hotfix.

The Firefox update hotfix bypasses the built-in application update mechanism and installs a modern Firefox.

Version 1 The fields in this measurement are dynamically created based on which versions of the update hotfix state file are found on disk.

The general format of the fields is `<version>.<thing>` where `version` is a hotfix version like `v20140527` and `thing` is a key from the hotfix state file, e.g. `upgradedFrom`. Here are some of the `things` that can be defined.

upgradedFrom String identifying the Firefox version that the hotfix upgraded from. e.g. `16.0` or `17.0.1`.

uninstallReason String with enumerated values identifying why the hotfix was uninstalled. Value will be `STILL_INSTALLED` if the hotfix is still installed.

downloadAttempts Integer number of times the hotfix started downloading an installer. Download resumes are part of this count.

downloadFailures Integer count of times a download supposedly completed but couldn't be validated. This likely represents something wrong with the network connection. The ratio of this to `downloadAttempts` should be low.

installAttempts Integer count of times the hotfix attempted to run the installer. This should ideally be 1. It should only be greater than 1 if UAC elevation was cancelled or not allowed.

installFailures Integer count of total installation failures this client experienced. Can be 0. `installAttempts - installFailures` implies install successes.

notificationsShown Integer count of times a notification was displayed to the user that they are running an older Firefox.

org.mozilla.places.places

This measurement contains information about the Places database (where Firefox stores its history and bookmarks).

Version 1 Daily counts of items in the database are reported in the following properties:

bookmarks Integer count of bookmarks present.

pages Integer count of pages in the history database.

Example

```
"org.mozilla.places.places": {
  "_v": 1,
  "bookmarks": 388,
  "pages": 94870
}
```

org.mozilla.profile.age

This measurement contains information about the current profile's age (and in version 2, the profile's most recent reset date)

Version 2 *profileCreation* and *profileReset* properties are present. Both define the integer days since UNIX epoch that the current profile was created or reset accordingly.

Version 1 A single *profileCreation* property is present. It defines the integer days since UNIX epoch that the current profile was created.

Notes It is somewhat difficult to obtain a reliable *profile born date* due to a number of factors, but since Version 2, improvements have been made - on a "profile reset" we copy the profileCreation date from the old profile and record the time of the reset in profileReset.

Example

```
"org.mozilla.profile.age": {
  "_v": 2,
  "profileCreation": 15176
  "profileReset": 15576
}
```

org.mozilla.searches.counts

This measurement contains information about searches performed in the application.

Version 6 (mobile) This adds two new search locations: *widget* and *activity*, corresponding to the search widget and search activity respectively.

Version 2 This behaves like version 1 except we added all search engines that Mozilla has a partner agreement with. Like version 1, we concatenate a search engine ID with a search origin.

Another difference with version 2 is we should no longer misattribute a search to the *other* bucket if the search engine name is localized.

The set of search engine providers is:

- amazon-co-uk
- amazon-de
- amazon-en-GB
- amazon-france

- amazon-it
- amazon-jp
- amazondotcn
- amazondotcom
- amazondotcom-de
- aol-en-GB
- aol-web-search
- bing
- eBay
- eBay-de
- eBay-en-GB
- eBay-es
- eBay-fi
- eBay-france
- eBay-hu
- eBay-in
- eBay-it
- google
- google-jp
- google-ku
- google-maps-zh-TW
- mailru
- mercadolibre-ar
- mercadolibre-cl
- mercadolibre-mx
- seznam-cz
- twitter
- twitter-de
- twitter-ja
- yahoo
- yahoo-NO
- yahoo-answer-zh-TW
- yahoo-ar
- yahoo-bid-zh-TW
- yahoo-br
- yahoo-ch

- yahoo-cl
- yahoo-de
- yahoo-en-GB
- yahoo-es
- yahoo-fi
- yahoo-france
- yahoo-fy-NL
- yahoo-id
- yahoo-in
- yahoo-it
- yahoo-jp
- yahoo-jp-auctions
- yahoo-mx
- yahoo-sv-SE
- yahoo-zh-TW
- yandex
- yandex-ru
- yandex-slovari
- yandex-tr
- yandex.by
- yandex.ru-be

And of course, *other*.

The sources for searches remain:

- abouthome
- contextmenu
- searchbar
- urlbar

The measurement will only be populated with providers and sources that occurred that day.

If a user switches locales, searches from default providers on the older locale will still be supported. However, if that same search engine is added by the user to the new build and is *not* a default search engine provider, its searches will be attributed to the *other* bucket.

Version 1 We record counts of performed searches grouped by search engine and search origin. Only search engines with which Mozilla has a business relationship are explicitly counted. All other search engines are grouped into an *other* bucket.

The following search engines are explicitly counted:

- Amazon.com
- Bing

- Google
- Yahoo
- Other

The following search origins are distinguished:

about:home Searches initiated from the search text box on about:home.

context menu Searches initiated from the context menu (highlight text, right click, and select "search for...")

search bar Searches initiated from the search bar (the text field next to the Awesomebar)

url bar Searches initiated from the awesomebar/url bar.

Due to the localization of search engine names, non en-US locales may wrongly attribute searches to the *other* bucket. This is fixed in version 2.

Example

```
"org.mozilla.searches.counts": {
  "_v": 1,
  "google.searchbar": 3,
  "google.urlbar": 7
},
```

org.mozilla.searches.engines

This measurement contains information about search engines.

Version 1 This version debuted with Firefox 31 on desktop. It contains the following properties:

default Daily string identifier or name of the default search engine provider.

This field will only be collected if Telemetry is enabled. If Telemetry is enabled and then later disabled, this field may disappear from future days in the payload.

The special value `NONE` could occur if there is no default search engine.

The special value `UNDEFINED` could occur if a default search engine exists but its identifier could not be determined.

This field's contents are `Services.search.defaultEngine.identifier` (if defined) or `"other-" + Services.search.defaultEngine.name` if not. In other words, search engines without an `.identifier` are prefixed with `other-`.

Version 2 Starting with Firefox 40, there is an additional optional value:

cohort Daily cohort string identifier, recorded if the user is part of search defaults A/B testing.

org.mozilla.sync.sync

This daily measurement contains information about the Sync service.

Values should be recorded for every day FHR measurements occurred.

Version 1 This version debuted with Firefox 30 on desktop. It contains the following properties:

enabled Daily numeric indicating whether Sync is configured and enabled. 1 if so, 0 otherwise.

preferredProtocol String version of the maximum Sync protocol version the client supports. This will be `1.1` for for legacy Sync and `1.5` for clients that speak the Firefox Accounts protocol.

actualProtocol The actual Sync protocol version the client is configured to use.

This will be `1.1` if the client is configured with the legacy Sync service or if the client only supports `1.1`.

It will be `1.5` if the client supports `1.5` and either a) the client is not configured b) the client is using Firefox Accounts Sync.

syncStart Count of sync operations performed.

syncSuccess Count of sync operations that completed successfully.

syncError Count of sync operations that did not complete successfully.

This is a measure of overall sync success. This does *not* reflect recoverable errors (such as record conflict) that can occur during sync. This is thus a rough proxy of whether the sync service is operating without error.

org.mozilla.sync.devices

This daily measurement contains information about the device type composition for the configured Sync account.

Version 1 Version 1 was introduced with Firefox 30.

Field names are dynamic according to the client-reported device types from Sync records. All fields are daily last seen integer values corresponding to the number of devices of that type.

Common values include:

desktop Corresponds to a Firefox desktop client.

mobile Corresponds to a Fennec client.

org.mozilla.sync.migration

This daily measurement contains information about sync migration (that is, the semi-automated process of migrating a legacy sync account to an FxA account.)

Measurements will start being recorded after a migration is offered by the sync server and stop after migration is complete or the user elects to "unlink" their sync account. In other words, it is expected that users with Sync setup for FxA or with sync unconfigured will not collect data, and that for users where data is collected, the collection will only be for a relatively short period.

Version 1 Version 1 was introduced with Firefox 37 and includes the following properties:

state Corresponds to either a STATE_USER_* string or a STATE_INTERNAL_* string in FxaMigration.jsm. This reflects a state where we are waiting for the user, or waiting for some internal process to complete on the way to completing the migration.

declined Corresponds to the number of times the user closed the migration infobar.

unlinked Set if the user declined to migrate and instead "unlinked" Sync from the browser.

accepted Corresponds to the number of times the user explicitly elected to start or continue the migration - it counts how often the user clicked on any UI created specifically for migration. The "ideal" UX for migration would see this at exactly 1, some known edge-cases (eg, browser restart required to finish) could expect this to be 2, and anything more means we are doing something wrong.

org.mozilla.sysinfo.sysinfo

This measurement contains basic information about the system the application is running on.

Version 2 This version debuted with Firefox 29 on desktop.

A single property was introduced.

isWow64 If present, this property indicates whether the machine supports WoW64. This property can be used to identify whether the host machine is 64-bit.

This property is only present on Windows machines. It is the preferred way to identify 32- vs 64-bit support in that environment.

Version 1 The following properties may be available:

cpuCount Integer number of CPUs/cores in the machine.

memoryMB Integer megabytes of memory in the machine.

manufacturer The manufacturer of the device.

device The name of the device (like model number).

hardware Unknown.

name OS name.

version OS version.

architecture OS architecture that the application is built for. This is not the actual system architecture.

Example

```
"org.mozilla.sysinfo.sysinfo": {
  "_v": 1,
  "cpuCount": 8,
  "memoryMB": 8192,
  "architecture": "x86-64",
  "name": "Darwin",
  "version": "12.2.0"
}
```

org.mozilla.translation.translation

This daily measurement contains information about the usage of the translation feature. It is a special telemetry measurement which will only be recorded in FHR if telemetry is enabled.

Version 1 Daily counts are reported in the following properties:

translationOpportunityCount Integer count of the number of opportunities there were to translate a page.

missedTranslationOpportunityCount Integer count of the number of missed opportunities there were to translate a page. A missed opportunity is when the page language is not supported by the translation provider.

pageTranslatedCount Integer count of the number of pages translated.

charactersTranslatedCount Integer count of the number of characters translated.

detectedLanguageChangedBefore Integer count of the number of times the user manually adjusted the detected language before translating.

detectedLanguageChangedAfter Integer count of the number of times the user manually adjusted the detected language after having first translated the page.

targetLanguageChanged Integer count of the number of times the user manually adjusted the target language.

deniedTranslationOffer Integer count of the number of times the user opted-out offered page translation, either by the Not Now button or by the notification's close button in the "offer" state.

autoRejectedTranlationOffer Integer count of the number of times the user is not offered page translation because they had previously clicked "Never translate this language" or "Never translate this site".

showOriginalContent Integer count of the number of times the user activated the Show Original command.

Additional daily counts broken down by language are reported in the following properties:

translationOpportunityCountsByLanguage A mapping from language to count of opportunities to translate that language.

missedTranslationOpportunityCountsByLanguage A mapping from language to count of missed opportunities to translate that language.

pageTranslatedCountsByLanguage A mapping from language to the counts of pages translated from that language. Each language entry will be an object containing a "total" member along with individual counts for each language translated to.

Other properties:

detectLanguageEnabled Whether automatic language detection is enabled. This is an integer, 0 or 1.

showTranslationUI Whether the translation feature UI will be shown. This is an integer, 0 or 1.

Example

```
"org.mozilla.translation.translation": {
  "_v": 1,
  "detectLanguageEnabled": 1,
  "showTranslationUI": 1,
  "translationOpportunityCount": 134,
  "missedTranslationOpportunityCount": 32,
  "pageTranslatedCount": 6,
  "charactersTranslatedCount": "1126",
  "detectedLanguageChangedBefore": 1,
  "detectedLanguageChangedAfter": 2,
  "targetLanguageChanged": 0,
  "deniedTranslationOffer": 3,
  "autoRejectedTranlationOffer": 1,
  "showOriginalContent": 2,
  "translationOpportunityCountsByLanguage": {
    "fr": 100,
    "es": 34
```

```
  },
  "missedTranslationOpportunityCountsByLanguage": {
    "it": 20,
    "nl": 10,
    "fi": 2
  },
  "pageTranslatedCountsByLanguage": {
    "fr": {
      "total": 6,
      "es": 5,
      "en": 1
    }
  }
}
```

org.mozilla.experiments.info

Daily measurement reporting information about the Telemetry Experiments service.

Version 1 Property:

lastActive ID of the final Telemetry Experiment that is active on a given day, if any.

Version 2 Adds an additional optional property:

lastActiveBranch If the experiment uses branches, the branch identifier string.

Example

```
"org.mozilla.experiments.info": {
  "_v": 2,
  "lastActive": "some.experiment.id",
  "lastActiveBranch": "control"
}
```

org.mozilla.uitour.treatment

Daily measurement reporting information about treatment tagging done by the UITour module.

Version 1 Daily text values in the following properties:

<tag>: Array of discrete strings corresponding to calls for setTreatmentTag(tag, value).

Example

```
"org.mozilla.uitour.treatment": {
  "_v": 1,
  "treatment": [
    "optin",
    "optin-DNT"
  ],
  "another-tag": [
    "foobar-value"
```

```
  ]
}
```

org.mozilla.passwordmgr.passwordmgr

Daily measurement reporting information about the Password Manager

Version 1 Property:

numSavedPasswords number of passwords saved in the Password Manager

enabled Whether or not the user has disabled the Password Manager in prefernces

Example

```
"org.mozilla.passwordmgr.passwordmgr": {
  "_v": 1,
  "numSavedPasswords": 5,
  "enabled": 0,
}
```

Version 2 More detailed measurements of login forms & their behavior

numNewSavedPasswordsInSession Number of passwords saved to the password manager this session.

numSuccessfulFills Number of times the password manager filled in password fields for user this session.

numTotalLoginsEncountered Number of times a login form was encountered by the user in the session.

Example

::

"org.mozilla.passwordmgr.passwordmgr": { "_v": 2, "numSavedPasswords": 32, "enabled": 1, "numNewSavedPasswords": 5, "numSuccessfulFills": 11, "numTotalLoginsEncountered": 23,

}

14.5.3 Identifiers

Firefox Health Report records some identifiers to keep track of clients and uploaded documents.

Identifier Types

Document/Upload IDs

A random UUID called the *Document ID* or *Upload ID* is generated when the FHR client creates or uploads a new document.

When clients generate a new *Document ID*, they persist this ID to disk **before** the upload attempt.

As part of the upload, the client sends all old *Document IDs* to the server and asks the server to delete them. In well-behaving clients, the server has a single record for each client with a randomly-changing *Document ID*.

Client IDs

A *Client ID* is an identifier that **attempts** to uniquely identify an individual FHR client. Please note the emphasis on *attempts* in that last sentence: *Client IDs* do not guarantee uniqueness.

The *Client ID* is generated when the client first runs or as needed.

The *Client ID* is transferred to the server as part of every upload. The server is thus able to affiliate multiple document uploads with a single *Client ID*.

Client ID Versions The semantics for how a *Client ID* is generated are versioned.

Version 1 The *Client ID* is a randomly-generated UUID.

History of Identifiers

In the beginning, there were just *Document IDs*. The thinking was clients would clean up after themselves and leave at most 1 active document on the server.

Unfortunately, this did not work out. Using brute force analysis to deduplicate records on the server, a number of interesting patterns emerged.

Orphaning Clients would upload a new payload while not deleting the old payload.

Divergent records Records would share data up to a certain date and then the data would almost completely diverge. This appears to be indicative of profile copying.

Rollback Records would share data up to a certain date. Each record in this set would contain data for a day or two but no extra data. This could be explained by filesystem rollback on the client.

A significant percentage of the records on the server belonged to misbehaving clients. Identifying these records was extremely resource intensive and error-prone. These records were undermining the ability to use Firefox Health Report data.

Thus, the *Client ID* was born. The intent of the *Client ID* was to uniquely identify clients so the extreme effort required and the questionable reliability of deduplicating server data would become problems of the past.

The *Client ID* was originally a randomly-generated UUID (version 1). This allowed detection of orphaning and rollback. However, these version 1 *Client IDs* were still susceptible to use on multiple profiles and machines if the profile was copied.

14.5.4 Legal and Privacy Concerns

Because Firefox Health Report collects and submits data to remote servers and is an opt-out feature, there are legal and privacy concerns over what data may be collected and submitted. **Additions or changes to submitted data should be signed off by responsible parties.**

CHAPTER 15

Crash Reporter

15.1 Overview

The **crash reporter** is a subsystem to record and manage application crash data.

While the subsystem is known as *crash reporter*, it helps to think of it more as a *process dump manager*. This is because the heart of this subsystem is really managing process dump files and these files are created not only from process crashes but also from hangs and other exceptional events.

The crash reporter subsystem is composed of a number of pieces working together.

Breakpad Breakpad is a library and set of tools to make collecting process information (notably dumps from crashes) easy. Breakpad is a 3rd party project (originaly developed by Google) that is imported into the tree.

Dump files Breakpad produces files called *dump files* that hold process data (stacks, heap data, etc).

Crash Reporter Client The crash reporter client is a standalone executable that is launched to handle dump files. This application optionally submits crashes to Mozilla (or the configured server).

15.2 How Main-Process Crash Handling Works

The crash handler is hooked up very early in the Gecko process lifetime. It all starts in `XREMain::XRE_mainInit()` from `nsAppRunner.cpp`. Assuming crash reporting is enabled, this startup function registers an exception handler for the process and tells the crash reporter subsystem about basic metadata such as the application name and version.

The registration of the crash reporter exception handler doubles as initialization of the crash reporter itself. This happens in `CrashReporter::SetExceptionHandler()` from `nsExceptionHandler.cpp`. The crash reporter figures out what application to use for reporting dumped crashes and where to store these dump files on disk. The Breakpad exception handler (really just a mechanism for dumping process state) is initialized as part of this function. The Breakpad exception handler is a `google_breakpad::ExceptionHandler` instance and it's stored as `gExceptionHandler`.

As the application runs, various other systems may write *annotations* or *notes* to the crash reporter to indicate state of the application, help with possible reasons for a current or future crash, etc. These are performed via `CrashReporter::AnnotateCrashReport()` and `CrashReporter::AppendAppNotesToCrashReport()` from `nsExceptionHandler.h`.

For well running applications, this is all that happens. However, if a crash or similar exceptional event occurs (such as a hang), we need to write a crash report.

When an event worthy of writing a dump occurs, the Breakpad exception handler is invoked and Breakpad does its thing. When Breakpad has finished, it calls back into `CrashReporter::MinidumpCallback()` from `nsExceptionHandler.cpp` to tell the crash reporter about what was written.

`MinidumpCallback()` performs a number of actions once a dump has been written. It writes a file with the time of the crash so other systems can easily determine the time of the last crash. It supplements the dump file with an *extra* file containing Mozilla-specific metadata. This data includes the annotations set via `CrashReporter::AnnotateCrashReport()` as well as time since last crash, whether garbage collection was active at the time of the crash, memory statistics, etc.

If the *crash reporter client* is enabled, `MinidumpCallback()` invokes it. It simply tries to create a new *crash reporter client* process (e.g. *crashreporter.exe*) with the path to the written minidump file as an argument.

The *crash reporter client* performs a number of roles. There's a lot going on, so you may want to look at `main()` in `crashreporter.cpp`. First, it verifies the dump data is sane. If it isn't (e.g. required metadata is missing), the dump data is ignored. If dump data looks sane, the dump data is moved into the *pending* directory for the configured data directory (defined via the `MOZ_CRASHREPORTER_DATA_DIRECTORY` environment variable or from the UI). Once this is done, the main crash reporter UI is displayed via `UIShowCrashUI()`. The crash reporter UI is platform specific: there are separate versions for Windows, OS X, and various *NIX presentation flavors (such as GTK). The basic gist is a dialog is displayed to the user and the user has the opportunity to submit this dump data to a remote server.

If a dump is submitted via the crash reporter, the raw dump files are removed from the *pending* directory and a file containing the crash ID from the remote server for the submitted dump is created in the *submitted* directory.

If the user chooses not to submit a dump in the crash reporter UI, the dump files are deleted.

And that's pretty much what happens when a crash/dump is written!

15.3 Plugin and Child Process Crashes

Crashes in plugin and child processes are also managed by the crash reporting subsystem.

Child process crashes are handled by the `mozilla::dom::CrashReporterParent` class defined in `dom/ipc`. When a child process crashes, the toplevel IPDL actor should check for it by calling TakeMinidump in its `ActorDestroy` Method: see `mozilla::plugins::PluginModuleParent::ActorDestroy` and `mozilla::plugins::PluginModuleParent::ProcessFirstMinidump`. That method is responsible for calling `mozilla::dom::CrashReporterParent::GenerateCrashReportForMinidump` with appropriate crash annotations specific to the crash. All child-process crashes are annotated with a `ProcessType` annotation, such as "content" or "plugin".

Submission of child process crashes is handled by application code. This code prompts the user to submit crashes in context-appropriate UI and then submits the crashes using `CrashSubmit.jsm`.

15.4 Memory Reports

When a process detects that it is running low on memory, a memory report is saved. If the process crashes, the memory report will be included with the crash report. `nsThread::SaveMemoryReportNearOOM()` checks to see if the process is low on memory every 30 seconds at most and saves a report every 3 minutes at most. Since a child process cannot actually save to the hard drive, it instead notifies its parent process, which saves the report for it. If a crash does occur, the memory report is moved to the *pending* directory with the other dump data and an annotation is added to indicate the presence of the report. This happens in `nsExceptionHandler.cpp`, but occurs in different functions depending on what process crashed. When the main process crashes, this happens in `MinidumpCallback()`. When a child process crashes, it happens in `OnChildProcessDumpRequested()`, with the annotation being added in `WriteExtraData()`.

15.5 Flash Process Crashes

On Windows Vista+, the Adobe Flash plugin creates two extra processes in its Firefox plugin to implement OS-level sandboxing. In order to catch crashes in these processes, Firefox injects a crash report handler into the process using the code at `InjectCrashReporter.cpp`. When these crashes occur, the ProcessType=plugin annotation is present, and an additional annotation FlashProcessDump has the value "Sandbox" or "Broker".

15.6 Plugin Hangs

Plugin hangs are handled as crash reports. If a plugin doesn't respond to an IPC message after 60 seconds, the plugin IPC code will take minidumps of all of the processes involved and then kill the plugin.

In this case, there will be only one .ini file with the crash report metadata, but there will be multiple dump files: at least one for the browser process and one for the plugin process, and perhaps also additional dumps for the Flash sandbox and broker processes. All of these files are submitted together as a unit. Before submission, the filenames of the files are linked:

- **uuid.ini** - *annotations, includes an additional_minidumps field*
- **uuid.dmp** - *plugin process dump file*
- **uuid-<other>.dmp** - *other process dump file as listed in additional_minidumps*

15.7 Browser Hangs

There is a feature of Firefox that will crash Firefox if it stops processing messages after a certain period of time. This feature doesn't work well and is disabled by default. See `xpcom/threads/HangMonitor.cpp`. Hang crashes are annotated with `Hang=1`.

15.8 about:crashes

If the crash reporter subsystem is enabled, the *about:crashes* page will be registered with the application. This page provides information about previous and submitted crashes.

It is also possible to submit crashes from *about:crashes*.

CHAPTER 16

Supbrocess Module

The Subprocess module allows a caller to spawn a native host executable, and communicate with it asynchronously over its standard input and output pipes.

Processes are launched asynchronously `Subprocess.call` method, based on the properties of a single options object. The method returns a promise which resolves, once the process has successfully launched, to a `Process` object, which can be used to communicate with and control the process.

A simple Hello World invocation, which writes a message to a process, reads it back, logs it, and waits for the process to exit looks something like:

```
let proc = await Subprocess.call({
  command: "/bin/cat",
});

proc.stdin.write("Hello World!");

let result = await proc.stdout.readString();
console.log(result);

proc.stdin.close();
let {exitCode} = await proc.wait();
```

16.1 Input and Output Redirection

Communication with the child process happens entirely via one-way pipes tied to its standard input, standard output, and standard error file descriptors. While standard input and output are always redirected to pipes, standard error is inherited from the parent process by default. Standard error can, however, optionally be either redirected to its own pipe or merged into the standard output pipe.

The module is designed primarily for use with processes following a strict IO protocol, with predictable message sizes. Its read operations, therefore, either complete after reading the exact amount of data specified, or do not complete at all. For cases where this is not desirable, `read()` and `readString` may be called without any length argument, and will return a chunk of data of an arbitrary size.

16.2 Process and Pipe Lifecycles

Once the process exits, any buffered data from its output pipes may still be read until the pipe is explicitly closed. Unless the pipe is explicitly closed, however, any pending buffered data *must* be read from the pipe, or the resources associated with the pipe will not be freed.

Beyond this, no explicit cleanup is required for either processes or their pipes. So long as the caller ensures that the process exits, and there is no pending input to be read on its `stdout` or `stderr` pipes, all resources will be freed automatically.

The preferred way to ensure that a process exits is to close its input pipe and wait for it to exit gracefully. Processes which haven't exited gracefully by shutdown time, however, must be forcibly terminated:

```
let proc = await Subprocess.call({
  command: "/usr/bin/subprocess.py",
});

// Kill the process if it hasn't gracefully exited by shutdown time.
let blocker = () => proc.kill();

AsyncShutdown.profileBeforeChange.addBlocker(
  "Subprocess: Killing hung process",
  blocker);

proc.wait().then(() => {
  // Remove the shutdown blocker once we've exited.
  AsyncShutdown.profileBeforeChange.removeBlocker(blocker);

  // Close standard output, in case there's any buffered data we haven't read.
  proc.stdout.close();
});

// Send a message to the process, and close stdin, so the process knows to
// exit.
proc.stdin.write(message);
proc.stdin.close();
```

In the simpler case of a short-running process which takes no input, and exits immediately after producing output, it's generally enough to simply read its output stream until EOF:

```
let proc = await Subprocess.call({
  command: await Subprocess.pathSearch("ifconfig"),
});

// Read all of the process output.
let result = "";
let string;
while ((string = await proc.stdout.readString())) {
  result += string;
}
console.log(result);

// The output pipe is closed and no buffered data remains to be read.
// This means the process has exited, and no further cleanup is necessary.
```

16.3 Bidirectional IO

When performing bidirectional IO, special care needs to be taken to avoid deadlocks. While all IO operations in the Subprocess API are asynchronous, careless ordering of operations can still lead to a state where both processes are blocked on a read or write operation at the same time. For example,

```
let proc = await Subprocess.call({
  command: "/bin/cat",
```

```
});

let size = 1024 * 1024;
await proc.stdin.write(new ArrayBuffer(size));

let result = await proc.stdout.read(size);
```

The code attempts to write 1MB of data to an input pipe, and then read it back from the output pipe. Because the data is big enough to fill both the input and output pipe buffers, though, and because the code waits for the write operation to complete before attempting any reads, the `cat` process will block trying to write to its output indefinitely, and never finish reading the data from its standard input.

In order to avoid the deadlock, we need to avoid blocking on the write operation:

```
let size = 1024 * 1024;
proc.stdin.write(new ArrayBuffer(size));

let result = await proc.stdout.read(size);
```

There is no silver bullet to avoiding deadlocks in this type of situation, though. Any input operations that depend on output operations, or vice versa, have the possibility of triggering deadlocks, and need to be thought out carefully.

16.4 Arguments

Arguments may be passed to the process in the form an array of strings. Arguments are never split, or subjected to any sort of shell expansion, so the target process will receive the exact arguments array as passed to `Subprocess.call`. Argument 0 will always be the full path to the executable, as passed via the `command` argument:

```
let proc = await Subprocess.call({
  command: "/bin/sh",
  arguments: ["-c", "echo -n $0"],
});

let output = await proc.stdout.readString();
assert(output === "/bin/sh");
```

16.5 Process Environment

By default, the process is launched with the same environment variables and working directory as the parent process, but either can be changed if necessary. The working directory may be changed simply by passing a `workdir` option:

```
let proc = await Subprocess.call({
  command: "/bin/pwd",
  workdir: "/tmp",
});

let output = await proc.stdout.readString();
assert(output === "/tmp\n");
```

The process's environment variables can be changed using the `environment` and `environmentAppend` options. By default, passing an `environment` object replaces the process's entire environment with the properties in that object:

```
let proc = await Subprocess.call({
  command: "/bin/pwd",
  environment: {FOO: "BAR"},
});

let output = await proc.stdout.readString();
assert(output === "FOO=BAR\n");
```

In order to add variables to, or change variables from, the current set of environment variables, the `environmentAppend` object must be passed in addition:

```
let proc = await Subprocess.call({
  command: "/bin/pwd",
  environment: {FOO: "BAR"},
  environmentAppend: true,
});

let output = "";
while ((string = await proc.stdout.readString())) {
  output += string;
}

assert(output.includes("FOO=BAR\n"));
```

CHAPTER 17

Toolkit modules

The `/toolkit/modules` directory contains a number of self-contained toolkit modules considered small enough that they do not deserve individual directories.

17.1 AsyncShutdown

During shutdown of the process, subsystems are closed one after another. `AsyncShutdown` is a module dedicated to express shutdown-time dependencies between: - services and their clients; - shutdown phases (e.g. profile-before-change) and their clients.

17.1.1 Barriers: Expressing shutdown dependencies towards a service

Consider a service FooService. At some point during the shutdown of the process, this service needs to: - inform its clients that it is about to shut down; - wait until the clients have completed their final operations based on FooService (often asynchronously); - only then shut itself down.

This may be expressed as an instance of `AsyncShutdown.Barrier`. An instance of `AsyncShutdown.Barrier` provides: - a capability `client` that may be published to clients, to let them register or unregister blockers; - methods for the owner of the barrier to let it consult the state of blockers and wait until all client-registered blockers have been resolved.

Shutdown timeouts

By design, an instance of `AsyncShutdown.Barrier` will cause a crash if it takes more than 60 seconds *awake* for its clients to lift or remove their blockers (*awake* meaning that seconds during which the computer is asleep or too busy to do anything are not counted). This mechanism helps ensure that we do not leave the process in a state in which it can neither proceed with shutdown nor be relaunched.

If the CrashReporter is enabled, this crash will report: - the name of the barrier that failed; - for each blocker that has not been released yet:

- the name of the blocker;
- the state of the blocker, if a state function has been provided (see *Example 3: More sophisticated Barrier client*).

Example 1: Simple Barrier client

The following snippet presents an example of a client of FooService that has a shutdown dependency upon FooService. In this case, the client wishes to ensure that FooService is not shutdown before some state has been reached. An example is clients that need write data asynchronously and need to ensure that they have fully written their state to disk before shutdown, even if due to some user manipulation shutdown takes place immediately.

```
// Some client of FooService called FooClient

Components.utils.import("resource://gre/modules/FooService.jsm", this);

// FooService.shutdown is the `client` capability of a `Barrier`.
// See example 2 for the definition of `FooService.shutdown`
FooService.shutdown.addBlocker(
  "FooClient: Need to make sure that we have reached some state",
  () => promiseReachedSomeState
);
// promiseReachedSomeState should be an instance of Promise resolved once
// we have reached the expected state
```

Example 2: Simple Barrier owner

The following snippet presents an example of a service FooService that wishes to ensure that all clients have had a chance to complete any outstanding operations before FooService shuts down.

```
// Module FooService

Components.utils.import("resource://gre/modules/AsyncShutdown.jsm", this);
Components.utils.import("resource://gre/modules/Task.jsm", this);

this.exports = ["FooService"];

let shutdown = new AsyncShutdown.Barrier("FooService: Waiting for clients before shutting down");

// Export the `client` capability, to let clients register shutdown blockers
FooService.shutdown = shutdown.client;

// This Task should be triggered at some point during shutdown, generally
// as a client to another Barrier or Phase. Triggering this Task is not covered
// in this snippet.
let onshutdown = Task.async(function*() {
  // Wait for all registered clients to have lifted the barrier
  yield shutdown.wait();

  // Now deactivate FooService itself.
  // ...
});
```

Frequently, a service that owns a `AsyncShutdown.Barrier` is itself a client of another Barrier.

Example 3: More sophisticated Barrier client

The following snippet presents FooClient2, a more sophisticated client of FooService that needs to perform a number of operations during shutdown but before the shutdown of FooService. Also, given that this client is more sophisticated, we provide a function returning the state of FooClient2 during shutdown. If for some reason FooClient2's blocker is never lifted, this state can be reported as part of a crash report.

```
// Some client of FooService called FooClient2

Components.utils.import("resource://gre/modules/FooService.jsm", this);

FooService.shutdown.addBlocker(
  "FooClient2: Collecting data, writing it to disk and shutting down",
  () => Blocker.wait(),
  () => Blocker.state
);

let Blocker = {
  // This field contains information on the status of the blocker.
  // It can be any JSON serializable object.
  state: "Not started",

  wait: Task.async(function*() {
    // This method is called once FooService starts informing its clients that
    // FooService wishes to shut down.

    // Update the state as we go. If the Barrier is used in conjunction with
    // a Phase, this state will be reported as part of a crash report if FooClient fails
    // to shutdown properly.
    this.state = "Starting";

    let data = yield collectSomeData();
    this.state = "Data collection complete";

    try {
      yield writeSomeDataToDisk(data);
      this.state = "Data successfully written to disk";
    } catch (ex) {
      this.state = "Writing data to disk failed, proceeding with shutdown: " + ex;
    }

    yield FooService.oneLastCall();
    this.state = "Ready";
  }.bind(this)
};
```

Example 4: A service with both internal and external dependencies

```
    // Module FooService2

    Components.utils.import("resource://gre/modules/AsyncShutdown.jsm", this);
    Components.utils.import("resource://gre/modules/Task.jsm", this);
    Components.utils.import("resource://gre/modules/Promise.jsm", this);

    this.exports = ["FooService2"];

    let shutdown = new AsyncShutdown.Barrier("FooService2: Waiting for clients before shutting down"

    // Export the `client` capability, to let clients register shutdown blockers
    FooService2.shutdown = shutdown.client;

    // A second barrier, used to avoid shutting down while any connections are open.
    let connections = new AsyncShutdown.Barrier("FooService2: Waiting for all FooConnections to be c
```

```
    let isClosed = false;

    FooService2.openFooConnection = function(name) {
      if (isClosed) {
        throw new Error("FooService2 is closed");
      }

      let deferred = Promise.defer();
      connections.client.addBlocker("FooService2: Waiting for connection " + name + " to close",  de

      // ...


      return {
        // ...
        // Some FooConnection object. Presumably, it will have additional methods.
        // ...
        close: function() {
          // ...
          // Perform any operation necessary for closing
          // ...

          // Don't hoard blockers.
          connections.client.removeBlocker(deferred.promise);

          // The barrier MUST be lifted, even if removeBlocker has been called.
          deferred.resolve();
        }
      };
    };


    // This Task should be triggered at some point during shutdown, generally
    // as a client to another Barrier. Triggering this Task is not covered
    // in this snippet.
    let onshutdown = Task.async(function*() {
      // Wait for all registered clients to have lifted the barrier.
      // These clients may open instances of FooConnection if they need to.
      yield shutdown.wait();

      // Now stop accepting any other connection request.
      isClosed = true;

      // Wait for all instances of FooConnection to be closed.
      yield connections.wait();

      // Now finish shutting down FooService2
      // ...
    });
```

17.1.2 Phases: Expressing dependencies towards phases of shutdown

The shutdown of a process takes place by phase, such as: - `profileBeforeChange` (once this phase is complete, there is no guarantee that the process has access to a profile directory); - `webWorkersShutdown` (once this phase is complete, JavaScript does not have access to workers anymore); - ...

Much as services, phases have clients. For instance, all users of web workers MUST have finished using their web

workers before the end of phase `webWorkersShutdown`.

Module `AsyncShutdown` provides pre-defined barriers for a set of well-known phases. Each of the barriers provided blocks the corresponding shutdown phase until all clients have lifted their blockers.

List of phases

`AsyncShutdown.profileChangeTeardown`

> The client capability for clients wishing to block asynchronously during observer notification "profile-change-teardown".

`AsyncShutdown.profileBeforeChange`

> The client capability for clients wishing to block asynchronously during observer notification "profile-change-teardown". Once the barrier is resolved, clients other than Telemetry MUST NOT access files in the profile directory and clients MUST NOT use Telemetry anymore.

`AsyncShutdown.sendTelemetry`

> The client capability for clients wishing to block asynchronously during observer notification "profile-before-change-telemetry". Once the barrier is resolved, Telemetry must stop its operations.

`AsyncShutdown.webWorkersShutdown`

> The client capability for clients wishing to block asynchronously during observer notification "web-workers-shutdown". Once the phase is complete, clients MUST NOT use web workers.

CHAPTER 18

Add-on Manager

This is the nascent documentation of the Add-on Manager code.

The public Add-on Manager interfaces are documented on MDN:

https://developer.mozilla.org/en-US/Add-ons/Add-on_Manager

18.1 Firefox System Add-on Update Protocol

This document describes the protocol that Firefox uses when retrieving updates for System Add-ons from the automatic update service (AUS, currently Balrog), and the expected behavior of Firefox based on the updater service's response.

18.1.1 System Add-ons

System add-ons:

- Are add-ons that ship with Firefox and cannot be disabled
- Can be updated by Firefox depending on the AUS response to Firefox's update request
- Are stored in two locations:
  - The **default** set ships with Firefox and is stored in the application directory.
  - The **update** set is stored in the user's profile directory. If an add-on is both in the update and default set, the update version gets precedence.

18.1.2 Update Request

To determine what updates to install, Firefox makes an HTTP **GET** request to AUS once a day via a URL of the form:

```
https://aus5.mozilla.org/update/3/SystemAddons/%VERSION%/%BUILD_ID%/%BUILD_TARGET%/%LOCALE%/%CHANNEL%
```

The path segments surrounded by `%` symbols are variable fields that Firefox fills in with information about itself and the environment it's running in:

`VERSION` Firefox version number

`BUILD_ID` Build ID

`BUILD_TARGET` Build target

`LOCALE` Build locale

CHANNEL Update channel

OS_VERSION OS Version

DISTRIBUTION Firefox Distribution

DISTRIBUTION_VERSION Firefox Distribution version

18.1.3 Update Response

AUS should respond with an XML document that looks something like this:

```
<?xml version="1.0"?>
<updates>
  <addons>
    <addon id="loop@mozilla.org" URL="https://ftp.mozilla.org/pub/system-addons/hello/loop@mozilla.or
    <addon id="pocket@mozilla.org" URL="https://ftp.mozilla.org/pub/system-addons/pocket/pocket@mozil
  </addons>
</updates>
```

- The root element is `<updates>`, used for all updater responses.
- The only child of `<updates>` is `<addons>`, which represents a list of system add-ons to update.
- Within `<addons>` are several `<addon>` tags, each one corresponding to a system add-on to update.

`<addon>` tags **must** have the following attributes:

id The extension ID

URL URL to a signed XPI of the specified add-on version to download

hashFunction Identifier of the hash function used to generate the hashValue attribute.

hashValue Hash of the XPI file linked from the URL attribute, calculated using the function specified in the hashValue attribute.

size Size (in bytes) of the XPI file linked from the URL attribute.

version Version number of the add-on

18.1.4 Update Behavior

After receiving the update response, Firefox modifies the **update** add-ons according to the following algorithm:

1. If the `<addons>` tag is empty (`<addons></addons>`) in the response, **disable all system add-ons**, including both the **update** and **default** sets.
2. If no add-ons were specified in the response (i.e. the `<addons>` tag is not present), do nothing and finish.
3. If the **update** add-on set is equal to the set of add-ons specified in the update response, do nothing and finish.
4. If the set of **default** add-ons is equal to the set of add-ons specified in the update response, remove all the **update** add-ons and finish.
5. Download each add-on specified in the update response and store them in the "downloaded add-on set". A failed download **must** abort the entire system add-on update.
6. Validate the downloaded add-ons. The following **must** be true for all downloaded add-ons, or the update process is aborted:
   (a) The ID and version of the downloaded add-on must match the specified ID or version in the update response.

(b) The hash provided in the update response must match the downloaded add-on file.

(c) The downloaded add-on file size must match the size given in the update response.

(d) The add-on must be compatible with Firefox (i.e. it must not be for a different application, such as Thunderbird).

(e) The add-on must be packed (i.e. be an XPI file).

(f) The add-on must be restartless.

(g) The add-on must be signed by the system add-on root certificate.

6. Once all downloaded add-ons are validated, install them into the profile directory as part of the **update** set.
7. Disable any **default** add-ons that were not present in the update response.

Notes on the update process:

- Add-ons are considered "equal" if they have the same ID and version number.

18.1.5 Examples

The follow section describes common situations that we have or expect to run into and how the protocol described above handles them.

For simplicity, unless otherwise specified, all examples assume that there are two system add-ons in existence: **Loop** and **Pocket**.

Basic

A user has Firefox 45, which shipped with Loop 1.0 and Pocket 1.0. We want to update users to Loop 2.0. AUS sends out the following update response:

```
<updates>
  <addons>
    <addon id="loop@mozilla.org" URL="https://ftp.mozilla.org/pub/system-addons/hello/loop@mozilla.or
    <addon id="pocket@mozilla.org" URL="https://ftp.mozilla.org/pub/system-addons/pocket/pocket@mozil
  </addons>
</updates>
```

Firefox will download Loop 2.0 and Pocket 1.0 and store them in the profile directory.

Missing Add-on

A user has Firefox 45, which shipped with Loop 1.0 and Pocket 1.0. We want to update users to Loop 2.0, but accidentally forget to specify Pocket in the update response. AUS sends out the following:

```
<updates>
  <addons>
    <addon id="loop@mozilla.org" URL="https://ftp.mozilla.org/pub/system-addons/hello/loop@mozilla.or
  </addons>
</updates>
```

Firefox will download Loop 2.0 and store it in the profile directory. It will disable Pocket completely.

Disable all system add-ons

A response from AUS with an empty add-on set will *disable all system add-ons*:

```
<updates>
  <addons></addons>
</updates>
```

Rollout

A user has Firefox 45, which shipped with Loop 1.0 and Pocket 1.0. We want to rollout Loop 2.0 at a 10% sample rate. 10% of the time, AUS sends out:

```
<updates>
  <addons>
    <addon id="loop@mozilla.org" URL="https://ftp.mozilla.org/pub/system-addons/hello/loop@mozilla.or
    <addon id="pocket@mozilla.org" URL="https://ftp.mozilla.org/pub/system-addons/pocket/pocket@mozil
  </addons>
</updates>
```

With this response, Firefox will download Pocket 1.0 and Loop 2.0 and install them into the profile directory.

The other 90% of the time, AUS sends out an empty response:

```
<updates></updates>
```

With the empty response, Firefox will not make any changes. This means users who haven't seen the 10% update response will stay on Loop 1.0, and users who have seen it will stay on Loop 2.0.

Once we're happy with the rollout and want to switch to 100%, AUS will send the 10% update response to 100% of users, upgrading everyone to Loop 2.0.

Rollback

This example continues from the "Rollout" example. If, during the 10% rollout, we find a major issue with Loop 2.0, we want to roll all users back to Loop 1.0. AUS sends out the following:

```
<updates>
  <addons>
    <addon id="loop@mozilla.org" URL="https://ftp.mozilla.org/pub/system-addons/hello/loop@mozilla.or
    <addon id="pocket@mozilla.org" URL="https://ftp.mozilla.org/pub/system-addons/pocket/pocket@mozil
  </addons>
</updates>
```

For users who have updated, Firefox will download Loop 1.0 and Pocket 1.0 and install them into the profile directory. For users that haven't yet updated, Firefox will see that the **default** add-on set matches the set in the update ping and clear the **update** add-on set.

Disable an Add-on

A user has Firefox 45, with Pocket 1.0 and Loop 1.0. Loop 1.0 ends up having a serious bug, and we want to disable the add-on completely while we work on a fix. AUS sends out the following:

```
<updates>
  <addons>
    <addon id="pocket@mozilla.org" URL="https://ftp.mozilla.org/pub/system-addons/pocket/pocket@mozil
```

```
  </addons>
</updates>
```

Firefox will download Pocket 1.0 and install it to the profile directory, and disable Loop.

CHAPTER 19

Linting

Linters are used in mozilla-central to help enforce coding style and avoid bad practices. Due to the wide variety of languages in use and the varying style preferences per team, this is not an easy task. In addition, linters should be runnable from editors, from the command line, from review tools and from continuous integration. It's easy to see how the complexity of running all of these different kinds of linters in all of these different places could quickly balloon out of control.

`Mozlint` is a library that accomplishes two goals:

1. It provides a standard method for adding new linters to the tree, which can be as easy as defining a json object in a `.lint` file. This helps keep lint related code localized, and prevents different teams from coming up with their own unique lint implementations.
2. It provides a streamlined interface for running all linters at once. Instead of running N different lint commands to test your patch, a single `mach lint` command will automatically run all applicable linters. This means there is a single API surface that other tools can use to invoke linters.

`Mozlint` isn't designed to be used directly by end users. Instead, it can be consumed by things like mach, mozreview and taskcluster.

19.1 Running Linters Locally

You can run all the various linters in the tree using the `mach lint` command. Simply pass in the directory or file you wish to lint (defaults to current working directory):

```
./mach lint path/to/files
```

Multiple paths are allowed:

```
./mach lint path/to/foo.js path/to/bar.py path/to/dir
```

`Mozlint` will automatically determine which types of files exist, and which linters need to be run against them. For example, if the directory contains both JavaScript and Python files then mozlint will automatically run both ESLint and Flake8 against those files respectively.

To restrict which linters are invoked manually, pass in `-l/--linter`:

```
./mach lint -l eslint path/to/files
```

Finally, `mozlint` can lint the files touched by a set of revisions or the working directory using the `-r/--rev` and `-w/--workdir` arguments respectively. These work both with mercurial and git. In the case of `--rev` the value is passed directly to the underlying vcs, so normal revision specifiers will work. For example, say we want to lint all files touched by the last three commits. In mercurial, this would be:

```
./mach lint -r ".~2::."
```

In git, this would be:

```
./mach lint -r "HEAD~2 HEAD"
```

19.2 Adding a New Linter to the Tree

A linter is a python file with a `.lint` extension and a global dict called LINTER. Depending on how complex it is, there may or may not be any actual python code alongside the LINTER definition.

Here's a trivial example:

no-eval.lint

```
LINTER = {
    'name': 'EvalLinter',
    'description': "Ensures the string 'eval' doesn't show up."
    'include': "**/*.js",
    'type': 'string',
    'payload': 'eval',
}
```

Now `no-eval.lint` gets passed into `LintRoller.read()`.

19.2.1 Linter Types

There are three types of linters, though more may be added in the future.

1. string - fails if substring is found
2. regex - fails if regex matches
3. external - fails if a python function returns a non-empty result list

As seen from the example above, string and regex linters are very easy to create, but they should be avoided if possible. It is much better to use a context aware linter for the language you are trying to lint. For example, use eslint to lint JavaScript files, use flake8 to lint python files, etc.

Which brings us to the third and most interesting type of linter, external. External linters call an arbitrary python function which is responsible for not only running the linter, but ensuring the results are structured properly. For example, an external type could shell out to a 3rd party linter, collect the output and format it into a list of `ResultContainer` objects.

19.2.2 LINTER Definition

Each `.lint` file must have a variable called LINTER which is a dict containing metadata about the linter. Here are the supported keys:

- name - The name of the linter (required)
- description - A brief description of the linter's purpose (required)
- type - One of 'string', 'regex' or 'external' (required)
- payload - The actual linting logic, depends on the type (required)
- include - A list of glob patterns that must be matched (optional)

- exclude - A list of glob patterns that must not be matched (optional)
- setup - A function that sets up external dependencies (optional)

In addition to the above, some `.lint` files correspond to a single lint rule. For these, the following additional keys may be specified:

- message - A string to print on infraction (optional)
- hint - A string with a clue on how to fix the infraction (optional)
- rule - An id string for the lint rule (optional)
- level - The severity of the infraction, either 'error' or 'warning' (optional)

19.2.3 Example

Here is an example of an external linter that shells out to the python flake8 linter:

```
import json
import os
import subprocess
from collections import defaultdict

from mozlint import result


FLAKE8_NOT_FOUND = """
Could not find flake8! Install flake8 and try again.
""".strip()


def lint(files, **lintargs):
    import which

    binary = os.environ.get('FLAKE8')
    if not binary:
        try:
            binary = which.which('flake8')
        except which.WhichError:
            print(FLAKE8_NOT_FOUND)
            return 1

    # Flake8 allows passing in a custom format string. We use
    # this to help mold the default flake8 format into what
    # mozlint's ResultContainer object expects.
    cmdargs = [
        binary,
        '--format',
        '{"path":"%(path)s","lineno":%(row)s,"column":%(col)s,"rule":"%(code)s","message":"%(text)s"}
    ] + files

    proc = subprocess.Popen(cmdargs, stdout=subprocess.PIPE, env=os.environ)
    output = proc.communicate()[0]

    # all passed
    if not output:
        return []

    results = []
```

```
    for line in output.splitlines():
        # res is a dict of the form specified by --format above
        res = json.loads(line)

        # parse level out of the id string
        if 'code' in res and res['code'].startswith('W'):
            res['level'] = 'warning'

        # result.from_linter is a convenience method that
        # creates a ResultContainer using a LINTER definition
        # to populate some defaults.
        results.append(result.from_linter(LINTER, **res))

    return results


LINTER = {
    'name': "flake8",
    'description': "Python linter",
    'include': ['**/*.py'],
    'type': 'external',
    'payload': lint,
}
```

19.3 Flake8

Flake8 is a popular lint wrapper for python. Under the hood, it runs three other tools and combines their results:

- pep8 for checking style
- pyflakes for checking syntax
- mccabe for checking complexity

19.3.1 Run Locally

The mozlint integration of flake8 can be run using mach:

```
$ mach lint --linter flake8 <file paths>
```

Alternatively, omit the `--linter flake8` and run all configured linters, which will include flake8.

19.3.2 Configuration

Only directories explicitly whitelisted will have flake8 run against them. To enable flake8 linting in a source directory, it must be added to the `include` directive in `'tools/lint/flake8.lint`. If you wish to exclude a subdirectory of an included one, you can add it to the `exclude` directive.

The default configuration file lives in `topsrcdir/.flake8`. The default configuration can be overriden for a given subdirectory by creating a new `.flake8` file in the subdirectory. Be warned that `.flake8` files cannot inherit from one another, so all configuration you wish to keep must be re-defined.

> **Warning:** Only `.flake8` files that live in a directory that is explicitly included in the `include` directive will be considered. See bug 1277851 for more details.

For an overview of the supported configuration, see flake8's documentation.

CHAPTER 20

Indices and tables

- genindex
- modindex
- search

CHAPTER 21

Mozilla ESLint Plugin

`balanced-listeners` checks that every addEventListener has a removeEventListener (and does the same for on/off).

`components-imports` adds the filename of imported files e.g. `Cu.import("some/path/Blah.jsm")` adds Blah to the global scope.

`import-globals-from` When the "import-globals-from <path>" comment is found in a file, then all globals from the file at <path> will be imported in the current scope.

`import-headjs-globals` imports globals from head.js and from any files that should be imported by head.js (as far as we can correctly resolve the path).

`mark-test-function-used` simply marks test (the test method) as used. This avoids ESLint telling us that the function is never called.

`no-aArgs` prevents using the hungarian notation in function arguments.

`no-cpows-in-tests` checks if the file is a browser mochitest and, if so, checks for possible CPOW usage.

`no-single-arg-cu-import` rejects calls to "Cu.import" that do not supply a second argument (meaning they add the exported properties into global scope).

`reject-importGlobalProperties` rejects calls to "Cu.importGlobalProperties". Use of this function is undesirable in some parts of the tree.

`reject-some-requires` rejects some calls to `require`, according to a regexp passed in as an option.

`this-top-level-scope` treats top-level assignments like `this.mumble = value` as declaring a global.

Note: These are string matches so we will miss situations where the parent object is assigned to another variable e.g.:

```
var b = gBrowser;
b.content // Would not be detected as a CPOW.
```

`var-only-at-top-level` marks all var declarations that are not at the top level invalid.

Possible values for all rules	
Value	Meaning
0	Deactivated
1	Warning
2	Error

Example configuration:

```
"rules": {
  "mozilla/balanced-listeners": 2,
  "mozilla/components-imports": 1,
```

```
  "mozilla/import-globals-from": 1,
  "mozilla/import-headjs-globals": 1,
  "mozilla/mark-test-function-used": 1,
  "mozilla/var-only-at-top-level": 1,
  "mozilla/no-cpows-in-tests": 1,
}
```

21.1 balanced-listeners

21.1.1 Rule Details

Checks that for every occurences of 'addEventListener' or 'on' there is an occurence of 'removeEventListener' or 'off' with the same event name.

21.2 import-headjs-globals

21.2.1 Rule Details

Import globals from head.js and from any files that were imported by head.js (as far as we can correctly resolve the path).

The following file import patterns are supported:

- `Services.scriptloader.loadSubScript(path)`
- `loader.loadSubScript(path)`
- `loadSubScript(path)`
- `loadHelperScript(path)`
- `import-globals-from path`

If path does not exist because it is generated e.g. `testdir + "/somefile.js"` we do our best to resolve it.

The following patterns are supported:

- `Cu.import("resource://devtools/client/shared/widgets/ViewHelpers.jsm");`
- `loader.lazyImporter(this, "name1");`
- `loader.lazyRequireGetter(this, "name2"`
- `loader.lazyServiceGetter(this, "name3"`
- `XPCOMUtils.defineLazyModuleGetter(this, "setNamedTimeout", ...)`
- `loader.lazyGetter(this, "toolboxStrings"`
- `XPCOMUtils.defineLazyGetter(this, "clipboardHelper"`

21.3 mark-test-function-used

21.3.1 Rule Details

Simply marks test (the test method) as used. This avoids ESLint telling us that the function is never called.

21.4 no-aArgs

21.4.1 Rule Details

Checks that function argument names don't start with lowercase 'a' followed by a capital letter. This is to prevent the use of Hungarian notation whereby the first letter is a prefix that indicates the type or intended use of a variable.

21.5 no-cpows-in-tests

21.5.1 Rule Details

This rule checks if the file is a browser mochitest and, if so, checks for possible CPOW usage by checking for the following strings:

- "gBrowser.contentWindow"
- "gBrowser.contentDocument"
- "gBrowser.selectedBrowser.contentWindow"
- "browser.contentDocument"
- "window.content"
- "content"
- "content."

Note: These are string matches so we will miss situations where the parent object is assigned to another variable e.g.:

```
var b = gBrowser;
b.content // Would not be detected as a CPOW.
```

21.6 reject-importGlobalProperties

21.6.1 Rule Details

Reject calls to Cu.importGlobalProperties.

21.7 reject-some-requires

21.7.1 Rule Details

This takes an option, a regular expression. Invocations of `require` with a string literal argument are matched against this regexp; and if it matches, the `require` use is flagged.

21.8 var-only-at-top-level

21.8.1 Rule Details

Marks all var declarations that are not at the top level invalid.

CHAPTER 22

Python Packages

22.1 mach package

22.1.1 Subpackages

mach.commands package

Submodules

mach.commands.commandinfo module

mach.commands.settings module

Module contents

mach.mixin package

Submodules

mach.mixin.logging module

class mach.mixin.logging.**LoggingMixin**
Bases: object

Provides functionality to control logging.

log(*level*, *action*, *params*, *format_str*)
Log a structured log event.

A structured log event consists of a logging level, a string action, a dictionary of attributes, and a formatting string.

The logging level is one of the logging.* constants, such as logging.INFO.

The action string is essentially the enumeration of the event. Each different type of logged event should have a different action.

The params dict is the metadata constituting the logged event.

The formatting string is used to convert the structured message back to human-readable format. Conversion back to human-readable form is performed by calling format() on this string, feeding into it the dict of attributes constituting the event.

self.log(logging.DEBUG, 'login', {'username': 'johndoe'}, 'User login: {username}')

populate_logger(*name=None*)
Ensure this class instance has a logger associated with it.

Users of this mixin that call log() will need to ensure self._logger is a logging.Logger instance before they call log(). This function ensures self._logger is defined by populating it if it isn't.

mach.mixin.process module

class mach.mixin.process.**ProcessExecutionMixin**
Bases: *mach.mixin.logging.LoggingMixin*

Mix-in that provides process execution functionality.

run_process(*args=None*, *cwd=None*, *append_env=None*, *explicit_env=None*, *log_name=None*, *log_level=20*, *line_handler=None*, *require_unix_environment=False*, *ensure_exit_code=0*, *ignore_children=False*, *pass_thru=False*)
Runs a single process to completion.

Takes a list of arguments to run where the first item is the executable. Runs the command in the specified directory and with optional environment variables.

append_env – Dict of environment variables to append to the current set of environment variables.

explicit_env – Dict of environment variables to set for the new process. Any existing environment variables will be ignored.

require_unix_environment if True will ensure the command is executed within a UNIX environment. Basically, if we are on Windows, it will execute the command via an appropriate UNIX-like shell.

ignore_children is proxied to mozprocess's ignore_children.

ensure_exit_code is used to ensure the exit code of a process matches what is expected. If it is an integer, we raise an Exception if the exit code does not match this value. If it is True, we ensure the exit code is 0. If it is False, we don't perform any exit code validation.

pass_thru is a special execution mode where the child process inherits this process's standard file handles (stdin, stdout, stderr) as well as additional file descriptors. It should be used for interactive processes where buffering from mozprocess could be an issue. pass_thru does not use mozprocess. Therefore, arguments like log_name, line_handler, and ignore_children have no effect.

Module contents

mach.test package

Subpackages

mach.test.providers package

Submodules

mach.test.providers.basic module

mach.test.providers.conditions module

mach.test.providers.conditions_invalid module

mach.test.providers.throw module

mach.test.providers.throw2 module

mach.test.providers.throw2.**throw_deep**(*message*)
mach.test.providers.throw2.**throw_real**(*message*)

Module contents

Submodules

mach.test.common module

class mach.test.common.**TestBase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

get_mach(*provider_file=None*, *entry_point=None*, *context_handler=None*)

provider_dir = u'/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/python/mach/m

mach.test.test_conditions module

class mach.test.test_conditions.**TestConditions**(*methodName='runTest'*)
Bases: *mach.test.common.TestBase*

Tests for conditionally filtering commands.

test_conditions_pass()
Test that a command which passes its conditions is runnable.

test_help_message()
Test that commands that are not runnable do not show up in help.

test_invalid_context_message()
Test that commands which do not pass all their conditions print the proper failure message.

test_invalid_type()
Test that a condition which is not callable raises an exception.

mach.test.test_config module

class mach.test.test_config.**Provider1**
Bases: object

config_settings = [(u'foo.bar', <class 'mach.config.StringType'>), (u'foo.baz', <class 'mach.config.PathType'>)]

config_settings_locale_directory = u'/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/che

class mach.test.test_config.**Provider2**
Bases: object

config_settings = [(u'a.string', <class 'mach.config.StringType'>), (u'a.boolean', <class 'mach.config.BooleanType'>

config_settings_locale_directory = u'/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/che

class mach.test.test_config.**Provider3**
Bases: object

classmethod config_settings()

config_settings_locale_directory = u'/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/che

class mach.test.test_config.**Provider4**
Bases: object

config_settings = [(u'foo.abc', <class 'mach.config.StringType'>, u'a', {u'choices': set([u'a', u'c', u'b'])}), (u'foo.xyz

config_settings_locale_directory = u'/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/che

class mach.test.test_config.**Provider5**
Bases: object

config_settings = [(u'foo.*', u'string'), (u'foo.bar', u'string')]

config_settings_locale_directory = u'/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/che

class mach.test.test_config.**ProviderDuplicate**
Bases: object

config_settings = [(u'dupesect.foo', <class 'mach.config.StringType'>), (u'dupesect.foo', <class 'mach.config.StringT

config_settings_locale_directory = u'/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/che

class mach.test.test_config.**TestConfigSettings**(*methodName='runTest'*)
Bases: unittest.case.TestCase

retrieval_type_helper(*provider*)

test_assignment_validation()

test_choices_validation()

test_duplicate_option()

test_empty()

test_file_reading_missing()
Missing files should silently be ignored.

test_file_reading_multiple()
Loading multiple files has proper overwrite behavior.

test_file_reading_single()

test_file_writing()

test_retrieval_type()

test_simple()

test_wildcard_options()

mach.test.test_dispatcher module

class mach.test.test_dispatcher.**TestDispatcher**(*methodName='runTest'*)
Bases: *mach.test.common.TestBase*

Tests dispatch related code

get_parser(*config=None*)

test_command_aliases()

mach.test.test_entry_point module

class mach.test.test_entry_point.**Entry**(*providers*)
Stub replacement for pkg_resources.EntryPoint

load()

class mach.test.test_entry_point.**TestEntryPoints**(*methodName='runTest'*)
Bases: *mach.test.common.TestBase*

Test integrating with setuptools entry points

provider_dir = u'/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/python/mach/m

test_load_entry_point_from_directory(**args*, ***keywargs*)

test_load_entry_point_from_file(**args*, ***keywargs*)

mach.test.test_error_output module

class mach.test.test_error_output.**TestErrorOutput**(*methodName='runTest'*)
Bases: *mach.test.common.TestBase*

test_command_error()

test_invoked_error()

mach.test.test_logger module

class mach.test.test_logger.**DummyLogger**(*cb*)
Bases: logging.Logger

handle(*record*)

class mach.test.test_logger.**TestStructuredHumanFormatter**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_non_ascii_logging()

Module contents

22.1.2 Submodules

22.1.3 mach.base module

class mach.base.**CommandContext**(*cwd=None*, *settings=None*, *log_manager=None*, *commands=None*, ***kwargs*)
Bases: object

Holds run-time state so it can easily be passed to command providers.

exception mach.base.**MachError**
Bases: exceptions.Exception

Base class for all errors raised by mach itself.

exception mach.base.**NoCommandError**
Bases: *mach.base.MachError*

No command was passed into mach.

exception mach.base.**UnknownCommandError**(*command*, *verb*, *suggested_commands=None*)
Bases: *mach.base.MachError*

Raised when we attempted to execute an unknown command.

exception mach.base.**UnrecognizedArgumentError**(*command*, *arguments*)
Bases: *mach.base.MachError*

Raised when an unknown argument is passed to mach.

22.1.4 mach.config module

This file defines classes for representing config data/settings.

Config data is modeled as key-value pairs. Keys are grouped together into named sections. Individual config settings (options) have metadata associated with them. This metadata includes type, default value, valid values, etc.

The main interface to config data is the ConfigSettings class. 1 or more ConfigProvider classes are associated with ConfigSettings and define what settings are available.

Descriptions of individual config options can be translated to multiple languages using gettext. Each option has associated with it a domain and locale directory. By default, the domain is the section the option is in and the locale directory is the "locale" directory beneath the directory containing the module that defines it.

People implementing ConfigProvider instances are expected to define a complete gettext .po and .mo file for the en_US locale. The **|mach settings locale-gen|** command can be used to populate these files.

class mach.config.**BooleanType**
Bases: *mach.config.ConfigType*

static from_config(*config*, *section*, *option*)

static to_config(*value*)

static validate(*value*)

exception mach.config.**ConfigException**
Bases: exceptions.Exception

class mach.config.**ConfigSettings**
Bases: _abcoll.Mapping

Interface for configuration settings.

This is the main interface to the configuration.

A configuration is a collection of sections. Each section contains key-value pairs.

When an instance is created, the caller first registers ConfigProvider instances with it. This tells the ConfigSettings what individual settings are available and defines extra metadata associated with those settings. This is used for validation, etc.

Once ConfigProvider instances are registered, a config is populated. It can be loaded from files or populated by hand.

ConfigSettings instances are accessed like dictionaries or by using attributes. e.g. the section "foo" is accessed through either settings.foo or settings['foo'].

Sections are modeled by the ConfigSection class which is defined inside this one. They look just like dicts or classes with attributes. To access the "bar" option in the "foo" section:

value = settings.foo.bar value = settings['foo']['bar'] value = settings.foo['bar']

Assignment is similar:

settings.foo.bar = value settings['foo']['bar'] = value settings['foo'].bar = value

You can even delete user-assigned values:

del settings.foo.bar del settings['foo']['bar']

If there is a default, it will be returned.

When settings are mutated, they are validated against the registered providers. Setting unknown settings or setting values to illegal values will result in exceptions being raised.

class **ConfigSection**(*config*, *name*, *settings*)
Bases: `_abcoll.MutableMapping`, `object`

Represents an individual config section.

get_meta(*option*)

options

`ConfigSettings`.**load_file**(*filename*)

`ConfigSettings`.**load_files**(*filenames*)
Load a config from files specified by their paths.

Files are loaded in the order given. Subsequent files will overwrite values from previous files. If a file does not exist, it will be ignored.

`ConfigSettings`.**load_fps**(*fps*)
Load config data by reading file objects.

`ConfigSettings`.**option_help**(*section*, *option*)
Obtain the translated help messages for an option.

`ConfigSettings`.**register_provider**(*provider*)
Register a SettingsProvider with this settings interface.

`ConfigSettings`.**write**(*fh*)
Write the config to a file object.

class `mach.config`.**ConfigType**
Bases: `object`

Abstract base class for config values.

static **from_config**(*config*, *section*, *option*)
Obtain the value of this type from a RawConfigParser.

Receives a RawConfigParser instance, a str section name, and the str option in that section to retrieve.

The implementation may assume the option exists in the RawConfigParser instance.

Implementations are not expected to validate the value. But, they should return the appropriate Python type.

static **to_config**(*value*)

static validate(*value*)
Validates a Python value conforms to this type.

Raises a TypeError or ValueError if it doesn't conform. Does not do anything if the value is valid.

class mach.config.**DefaultValue**
Bases: object

class mach.config.**IntegerType**
Bases: *mach.config.ConfigType*

static from_config(*config*, *section*, *option*)

static validate(*value*)

class mach.config.**PathType**
Bases: *mach.config.StringType*

static from_config(*config*, *section*, *option*)

static validate(*value*)

class mach.config.**PositiveIntegerType**
Bases: *mach.config.IntegerType*

static validate(*value*)

class mach.config.**StringType**
Bases: *mach.config.ConfigType*

static from_config(*config*, *section*, *option*)

static validate(*value*)

mach.config.**reraise_attribute_error**(*func*)
Used to make sure __getattr__ wrappers around __getitem__ raise AttributeError instead of KeyError.

22.1.5 mach.decorators module

class mach.decorators.**Command**(*name*, ***kwargs*)
Bases: object

Decorator for functions or methods that provide a mach command.

The decorator accepts arguments that define basic attributes of the command. The following arguments are recognized:

category – The string category to which this command belongs. Mach's help will group commands by category.

description – A brief description of what the command does.

parser – an optional argparse.ArgumentParser instance or callable that returns an argparse.ArgumentParser instance to use as the basis for the command arguments.

For example:

@Command('foo', category='misc', description='Run the foo action') def foo(self):

pass

class mach.decorators.**CommandArgument**(**args*, ***kwargs*)
Bases: object

Decorator for additional arguments to mach subcommands.

This decorator should be used to add arguments to mach commands. Arguments to the decorator are proxied to ArgumentParser.add_argument().

For example:

@Command('foo', help='Run the foo action') @CommandArgument('-b', '–bar', action='store_true', default=False,

help='Enable bar mode.')

def foo(self): pass

class mach.decorators.**CommandArgumentGroup**(*group_name*)
Bases: object

Decorator for additional argument groups to mach commands.

This decorator should be used to add arguments groups to mach commands. Arguments to the decorator are proxied to ArgumentParser.add_argument_group().

For example:

@Command('foo', helps='Run the foo action') @CommandArgumentGroup('group1') @CommandArgument('-b', '–bar', group='group1', action='store_true',

default=False, help='Enable bar mode.')

def foo(self): pass

The name should be chosen so that it makes sense as part of the phrase 'Command Arguments for <name>' because that's how it will be shown in the help message.

mach.decorators.**CommandProvider**(*cls*)
Class decorator to denote that it provides subcommands for Mach.

When this decorator is present, mach looks for commands being defined by methods inside the class.

mach.decorators.**SettingsProvider**(*cls*)
Class decorator to denote that this class provides Mach settings.

When this decorator is encountered, the underlying class will automatically be registered with the Mach registrar and will (likely) be hooked up to the mach driver.

class mach.decorators.**SubCommand**(*command*, *subcommand*, *description=None*)
Bases: object

Decorator for functions or methods that provide a sub-command.

Mach commands can have sub-commands. e.g. `mach command foo` or `mach command bar`. Each sub-command has its own parser and is effectively its own mach command.

The decorator accepts arguments that define basic attributes of the sub command:

command – The string of the command this sub command should be attached to.

subcommand – The string name of the sub command to register.

description – A textual description for this sub command.

22.1.6 mach.dispatcher module

class mach.dispatcher.**CommandAction**(*option_strings*, *dest*, *required=True*, *default=None*, *registrar=None*, *context=None*)

Bases: argparse.Action

An argparse action that handles mach commands.

This class is essentially a reimplementation of argparse's sub-parsers feature. We first tried to use sub-parsers. However, they were missing features like grouping of commands (http://bugs.python.org/issue14037).

The way this works involves light magic and a partial understanding of how argparse works.

Arguments registered with an argparse.ArgumentParser have an action associated with them. An action is essentially a class that when called does something with the encountered argument(s). This class is one of those action classes.

An instance of this class is created doing something like:

parser.add_argument('command', action=CommandAction, registrar=r)

Note that a mach.registrar.Registrar instance is passed in. The Registrar holds information on all the mach commands that have been registered.

When this argument is registered with the ArgumentParser, an instance of this class is instantiated. One of the subtle but important things it does is tell the argument parser that it's interested in *all* of the remaining program arguments. So, when the ArgumentParser calls this action, we will receive the command name plus all of its arguments.

For more, read the docs in __call__.

class mach.dispatcher.**CommandFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: argparse.HelpFormatter

Custom formatter to format just a subcommand.

add_usage(**args*)

class mach.dispatcher.**DispatchSettings**

config_settings = [(u'alias.*', u'string')]

config_settings_locale_directory = u'/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/che

class mach.dispatcher.**NoUsageFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: argparse.HelpFormatter

mach.dispatcher.**format_docstring**(*docstring*)

Format a raw docstring into something suitable for presentation.

This function is based on the example function in PEP-0257.

22.1.7 mach.logging module

class mach.logging.**ConvertToStructuredFilter**(*name=''*)

Bases: logging.Filter

Filter that converts unstructured records into structured ones.

filter(*record*)

class mach.logging.**LoggingManager**
Bases: object

Holds and controls global logging state.

An application should instantiate one of these and configure it as needed.

This class provides a mechanism to configure the output of logging data both from mach and from the overall logging system (e.g. from other modules).

add_json_handler(*fh*)
Enable JSON logging on the specified file object.

add_terminal_logging(*fh=<open file '<stdout>', mode 'w'>*, *level=20*, *write_interval=False*, *write_times=True*)
Enable logging to the terminal.

disable_unstructured()
Disable logging of unstructured messages.

enable_unstructured()
Enable logging of unstructured messages.

register_structured_logger(*logger*)
Register a structured logger.

This needs to be called for all structured loggers that don't chain up to the mach logger in order for their output to be captured.

replace_terminal_handler(*handler*)
Replace the installed terminal handler.

Returns the old handler or None if none was configured. If the new handler is None, removes any existing handler and disables logging to the terminal.

terminal

class mach.logging.**StructuredHumanFormatter**(*start_time*, *write_interval=False*, *write_times=True*)
Bases: logging.Formatter

Log formatter that writes structured messages for humans.

It is important that this formatter never be added to a logger that produces unstructured/classic log messages. If it is, the call to format() could fail because the string could contain things (like JSON) that look like formatting character sequences.

Because of this limitation, format() will fail with a KeyError if an unstructured record is passed or if the structured message is malformed.

format(*record*)

class mach.logging.**StructuredJSONFormatter**(*fmt=None*, *datefmt=None*)
Bases: logging.Formatter

Log formatter that writes a structured JSON entry.

format(*record*)

class mach.logging.**StructuredTerminalFormatter**(*start_time*, *write_interval=False*, *write_times=True*)
Bases: *mach.logging.StructuredHumanFormatter*

Log formatter for structured messages writing to a terminal.

format(*record*)

set_terminal(*terminal*)

mach.logging.**format_seconds**(*total*)
Format number of seconds to MM:SS.DD form.

22.1.8 mach.main module

class mach.main.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *version=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*)
Bases: argparse.ArgumentParser

Custom implementation argument parser to make things look pretty.

error(*message*)
Custom error reporter to give more helpful text on bad commands.

format_help()

class mach.main.**ContextWrapper**(*context*, *handler*)
Bases: object

class mach.main.**Mach**(*cwd*)
Bases: object

Main mach driver type.

This type is responsible for holding global mach state and dispatching a command from arguments.

The following attributes may be assigned to the instance to influence behavior:

populate_context_handler – If defined, it must be a callable. The

callable signature is the following: populate_context_handler(context, key=None)

It acts as a fallback getter for the mach.base.CommandContext instance. This allows to augment the context instance with arbitrary data for use in command handlers. For backwards compatibility, it is also called before command dispatch without a key, allowing the context handler to add attributes to the context instance.

require_conditions – If True, commands that do not have any condition functions applied will be skipped. Defaults to False.

settings_paths – A list of files or directories in which to search for settings files to load.

USAGE = u'%(prog)s [global arguments] command [command arguments]\n\nmach (German for "do") is the main interfa

add_global_argument(**args*, ***kwargs*)
Register a global argument with the argument parser.

Arguments are proxied to ArgumentParser.add_argument()

define_category(*name*, *title*, *description*, *priority=50*)
Provide a description for a named command category.

get_argument_parser(*context*)
Returns an argument parser for the command-line interface.

load_commands_from_directory(*path*)
Scan for mach commands from modules in a directory.

This takes a path to a directory, loads the .py files in it, and registers and found mach command providers with this mach instance.

load_commands_from_entry_point(*group=u'mach.providers'*)
Scan installed packages for mach command provider entry points. An entry point is a function that returns a list of paths to files or directories containing command providers.

This takes an optional group argument which specifies the entry point group to use. If not specified, it defaults to 'mach.providers'.

load_commands_from_file(*path*, *module_name=None*)
Scan for mach commands from a file.

This takes a path to a file and loads it as a Python module under the module name specified. If no name is specified, a random one will be chosen.

load_settings(*paths*)
Load the specified settings files.

If a directory is specified, the following basenames will be searched for in this order:

machrc, .machrc

log(*level*, *action*, *params*, *format_str*)
Helper method to record a structured log event.

require_conditions

run(*argv*, *stdin=None*, *stdout=None*, *stderr=None*)
Runs mach with arguments provided from the command line.

Returns the integer exit code that should be used. 0 means success. All other values indicate failure.

22.1.9 mach.registrar module

class mach.registrar.**MachRegistrar**
Bases: object

Container for mach command and config providers.

dispatch(*name*, *context=None*, *argv=None*, *subcommand=None*, ***kwargs*)
Dispatch/run a command.

Commands can use this to call other commands.

register_category(*name*, *title*, *description*, *priority=50*)

register_command_handler(*handler*)

register_settings_provider(*cls*)

22.1.10 mach.terminal module

This file contains code for interacting with terminals.

All the terminal interaction code is consolidated so the complexity can be in one place, away from code that is commonly looked at.

class mach.terminal.**LoggingHandler**
Bases: logging.Handler

Custom logging handler that works with terminal window dressing.

This is alternative terminal logging handler which contains smarts for emitting terminal control characters properly. Currently, it has generic support for "footer" elements at the bottom of the screen. Functionality can be added when needed.

emit(*record*)

flush()

class mach.terminal.**TerminalFooter**(*terminal*)
Bases: object

Represents something drawn on the bottom of a terminal.

clear()

draw()

22.1.11 Module contents

22.2 mozbuild package

22.2.1 Subpackages

mozbuild.action package

Submodules

mozbuild.action.buildlist module

A generic script to add entries to a file if the entry does not already exist.

Usage: buildlist.py <filename> <entry> [<entry> ...]

mozbuild.action.buildlist.**addEntriesToListFile**(*listFile*, *entries*)
Given a file **|listFile|** containing one entry per line, add each entry in **|entries|** to the file, unless it is already present.

mozbuild.action.buildlist.**main**(*args*)

mozbuild.action.cl module

mozbuild.action.explode_aar module

mozbuild.action.explode_aar.**explode**(*aar*, *destdir*)

mozbuild.action.explode_aar.**main**(*argv*)

mozbuild.action.file_generate module

mozbuild.action.generate_browsersearch module

Script to generate the browsersearch.json file for Fennec.

This script follows these steps:

1. Read the region.properties file in all the given source directories (see srcdir option). Merge all properties into a single dict accounting for the priority of source directories.

2. Read the default search plugin from 'browser.search.defaultenginename'.

3. Read the list of search plugins from the 'browser.search.order.INDEX' properties with values identifying particular search plugins by name.

4. Read each region-specific default search plugin from each property named like 'browser.search.defaultenginename.REGION'.

5. Read the list of region-specific search plugins from the 'browser.search.order.REGION.INDEX' properties with values identifying particular search plugins by name. Here, REGION is derived from a REGION for which we have seen a region-specific default plugin.

6. Generate a JSON representation of the above information, and write the result to browsersearch.json in the locale-specific raw resource directory e.g. raw/browsersearch.json, raw-pt-rBR/browsersearch.json.

mozbuild.action.generate_browsersearch.**main**(*args*)

mozbuild.action.generate_browsersearch.**merge_properties**(*filename*, *srcdirs*)
Merges properties from the given file in the given source directories.

mozbuild.action.generate_suggestedsites module

Script to generate the suggestedsites.json file for Fennec.

This script follows these steps:

1. Read the region.properties file in all the given source directories (see srcdir option). Merge all properties into a single dict accounting for the priority of source directories.

2. Read the list of sites from the list 'browser.suggestedsites.list.INDEX' and 'browser.suggestedsites.restricted.list.INDEX' properties with value of these keys being an identifier for each suggested site e.g. browser.suggestedsites.list.0=mozilla, browser.suggestedsites.list.1=fxmarketplace.

3. For each site identifier defined by the list keys, look for matching branches containing the respective properties i.e. url, title, etc. For example, for a 'mozilla' identifier, we'll look for keys like: browser.suggestedsites.mozilla.url, browser.suggestedsites.mozilla.title, etc.

4. Generate a JSON representation of each site, join them in a JSON array, and write the result to suggestedsites.json on the locale-specific raw resource directory e.g. raw/suggestedsites.json, raw-pt-rBR/suggestedsites.json.

mozbuild.action.generate_suggestedsites.**main**(*args*)

mozbuild.action.generate_suggestedsites.**merge_properties**(*filename*, *srcdirs*)
Merges properties from the given file in the given source directories.

mozbuild.action.generate_symbols_file module

mozbuild.action.jar_maker module

mozbuild.action.jar_maker.**main**(*args*)

mozbuild.action.make_dmg module

mozbuild.action.make_dmg.**main**(*args*)

mozbuild.action.make_dmg.**make_dmg**(*source_directory*, *output_dmg*)

mozbuild.action.package_fennec_apk module

mozbuild.action.package_geckolibs_aar module

Script to produce an Android ARchive (.aar) containing the compiled Gecko library binaries. The AAR file is intended for use by local developers using Gradle.

mozbuild.action.package_geckolibs_aar.**main**(*args*)

mozbuild.action.package_geckolibs_aar.**package_geckolibs_aar**(*topsrcdir*, *distdir*, *appname*, *output_file*)

mozbuild.action.package_geckolibs_aar.**package_geckoview_aar**(*topsrcdir*, *distdir*, *appname*, *output_file*)

mozbuild.action.preprocessor module

mozbuild.action.preprocessor.**main**(*args*)

mozbuild.action.process_define_files module

mozbuild.action.process_install_manifest module

mozbuild.action.process_install_manifest.**main**(*argv*)

mozbuild.action.process_install_manifest.**process_manifest**(*destdir*, *paths*, *track=None*, *remove_unaccounted=True*, *remove_all_directory_symlinks=True*, *remove_empty_directories=True*, *defines={}*)

mozbuild.action.test_archive module

mozbuild.action.webidl module

mozbuild.action.webidl.**main**(*argv*)
Perform WebIDL code generation required by the build system.

mozbuild.action.xpccheck module

A generic script to verify all test files are in the corresponding .ini file.

Usage: xpccheck.py <directory> [<directory> ...]

mozbuild.action.xpccheck.**getIniTests**(*testdir*)

mozbuild.action.xpccheck.**main**(*argv*)

mozbuild.action.xpccheck.**verifyDirectory**(*initests*, *directory*)

mozbuild.action.xpccheck.**verifyIniFile**(*initests*, *directory*)

mozbuild.action.xpidl-process module

mozbuild.action.zip module

mozbuild.action.zip.**main**(*args*)

Module contents

mozbuild.backend package

Submodules

mozbuild.backend.android_eclipse module

class mozbuild.backend.android_eclipse.**AndroidEclipseBackend**(*environment*)
Bases: *mozbuild.backend.common.CommonBackend*

Backend that generates Android Eclipse project files.

consume_finished()
The common backend handles WebIDL and test files. We don't handle these, so we don't call our superclass.

consume_object(*obj*)
Write out Android Eclipse project files.

summary()

mozbuild.backend.android_eclipse.**pretty_print**(*element*)
Return a pretty-printed XML string for an Element.

mozbuild.backend.base module

class mozbuild.backend.base.**BuildBackend**(*environment*)
Bases: *mach.mixin.logging.LoggingMixin*

Abstract base class for build backends.

A build backend is merely a consumer of the build configuration (the output of the frontend processing). It does something with said data. What exactly is the discretion of the specific implementation.

consume(*objs*)
Consume a stream of TreeMetadata instances.

This is the main method of the interface. This is what takes the frontend output and does something with it.

Child classes are not expected to implement this method. Instead, the base class consumes objects and calls methods (possibly) implemented by child classes.

consume_finished()
Called when consume() has completed handling all objects.

consume_object(*obj*)
Consumes an individual TreeMetadata instance.

This is the main method used by child classes to react to build metadata.

summary()

mozbuild.backend.base.**HybridBackend**(**backends*)
A HybridBackend is the combination of one or more PartialBackends with a non-partial BuildBackend.

Build configuration objects are passed to each backend, stopping at the first of them that declares having handled them.

class mozbuild.backend.base.**PartialBackend**(*environment*)
Bases: *mozbuild.backend.base.BuildBackend*

A PartialBackend is a BuildBackend declaring that its consume_object method may not handle all build configuration objects it's passed, and that it's fine.

mozbuild.backend.common module

class mozbuild.backend.common.**BinariesCollection**
Bases: object

Tracks state of binaries produced by the build.

class mozbuild.backend.common.**CommonBackend**(*environment*)
Bases: *mozbuild.backend.base.BuildBackend*

Holds logic common to all build backends.

consume_finished()

consume_object(*obj*)

class mozbuild.backend.common.**TestManager**(*config*)
Bases: object

Helps hold state related to tests.

add(*t*, *flavor*, *topsrcdir*, *default_supp_files*)

add_installs(*obj*, *topsrcdir*)

class mozbuild.backend.common.**WebIDLCollection**
Bases: object

Collects WebIDL info referenced during the build.

all_basenames()

all_non_static_basenames()

all_non_static_sources()

all_preprocessed_sources()

all_regular_basenames()

all_regular_bindinggen_stems()

all_regular_cpp_basenames()

all_regular_sources()

all_regular_stems()

all_sources()

all_static_sources()

all_stems()

all_test_basenames()

all_test_cpp_basenames()

all_test_sources()

all_test_stems()

generated_events_basenames()

generated_events_stems()

class mozbuild.backend.common.**XPIDLManager**(*config*)
Bases: object

Helps manage XPCOM IDLs in the context of the build system.

register_idl(*idl*, *allow_existing=False*)
Registers an IDL file with this instance.

The IDL file will be built, installed, etc.

mozbuild.backend.configenvironment module

class mozbuild.backend.configenvironment.**BuildConfig**
Bases: object

Represents the output of configure.

classmethod from_config_status(*path*)
Create an instance from a config.status file.

class mozbuild.backend.configenvironment.**ConfigEnvironment**(*topsrcdir*, *topobjdir*, *defines=None*, *non_global_defines=None*, *substs=None*, *source=None*, *mozconfig=None*)
Bases: object

Perform actions associated with a configured but bare objdir.

The purpose of this class is to preprocess files from the source directory and output results in the object directory.

There are two types of files: config files and config headers, each treated through a different member function.

Creating a ConfigEnvironment requires a few arguments:

- topsrcdir and topobjdir are, respectively, the top source and the top object directory.
- defines is a dict filled from AC_DEFINE and AC_DEFINE_UNQUOTED in autoconf.
- non_global_defines are a list of names appearing in defines above that are not meant to be exported in ACDEFINES (see below)
- substs is a dict filled from AC_SUBST in autoconf.

ConfigEnvironment automatically defines one additional substs variable from all the defines not appearing in non_global_defines:

- ACDEFINES contains the defines in the form -DNAME=VALUE, for use on preprocessor command lines. The order in which defines were given when creating the ConfigEnvironment is preserved.

and two other additional subst variables from all the other substs:

- ALLSUBSTS contains the substs in the form NAME = VALUE, in sorted order, for use in autoconf.mk. It includes ACDEFINES Only substs with a VALUE are included, such that the resulting file doesn't change when new empty substs are added. This results in less invalidation of build dependencies in the case of autoconf.mk..
- ALLEMPTYSUBSTS contains the substs with an empty value, in the form NAME =.

ConfigEnvironment expects a "top_srcdir" subst to be set with the top source directory, in msys format on windows. It is used to derive a "srcdir" subst when treating config files. It can either be an absolute path or a path relative to the topobjdir.

static `from_config_status`(*path*)

`is_artifact_build`

mozbuild.backend.cpp_eclipse module

class mozbuild.backend.cpp_eclipse.**CppEclipseBackend**(*environment*)
Bases: *mozbuild.backend.common.CommonBackend*

Backend that generates Cpp Eclipse project files.

`consume_finished`()

`consume_object`(*obj*)

static `get_workspace_path`(*topsrcdir*, *topobjdir*)

`summary`()

mozbuild.backend.fastermake module

class mozbuild.backend.fastermake.**FasterMakeBackend**(*environment*)
Bases: *mozbuild.backend.common.CommonBackend*, *mozbuild.backend.base.PartialBackend*

`consume_finished`()

`consume_object`(*obj*)

mozbuild.backend.mach_commands module

mozbuild.backend.recursivemake module

class mozbuild.backend.recursivemake.**BackendMakeFile**(*srcdir*, *objdir*, *environment*, *topsrcdir*, *topobjdir*)
Bases: object

Represents a generated backend.mk file.

This is both a wrapper around a file handle as well as a container that holds accumulated state.

It's worth taking a moment to explain the make dependencies. The generated backend.mk as well as the Makefile.in (if it exists) are in the GLOBAL_DEPS list. This means that if one of them changes, all targets in that Makefile are invalidated. backend.mk also depends on all of its input files.

It's worth considering the effect of file mtimes on build behavior.

Since we perform an "all or none" traversal of moz.build files (the whole tree is scanned as opposed to individual files), if we were to blindly write backend.mk files, the net effect of updating a single mozbuild file in the tree is all backend.mk files have new mtimes. This would in turn invalidate all make targets across the whole tree! This would effectively undermine incremental builds as any mozbuild change would cause the entire tree to rebuild!

The solution is to not update the mtimes of backend.mk files unless they actually change. We use FileAvoidWrite to accomplish this.

add_statement(*stmt*)

close()

diff

write(*buf*)

write_once(*buf*)

class mozbuild.backend.recursivemake.**RecursiveMakeBackend**(*environment*)
Bases: *mozbuild.backend.common.CommonBackend*

Backend that integrates with the existing recursive make build system.

This backend facilitates the transition from Makefile.in to moz.build files.

This backend performs Makefile.in -> Makefile conversion. It also writes out .mk files containing content derived from moz.build files. Both are consumed by the recursive make builder.

This backend may eventually evolve to write out non-recursive make files. However, as long as there are Makefile.in files in the tree, we are tied to recursive make and thus will need this backend.

class Substitution
Bases: object

BaseConfigSubstitution-like class for use with _create_makefile.

config

input_path

output_path

topobjdir

topsrcdir

RecursiveMakeBackend.**consume_finished**()

RecursiveMakeBackend.**consume_object**(*obj*)
Write out build files necessary to build with recursive make.

RecursiveMakeBackend.**summary**()

class mozbuild.backend.recursivemake.**RecursiveMakeTraversal**
Bases: object

Helper class to keep track of how the "traditional" recursive make backend recurses subdirectories. This is useful until all adhoc rules are removed from Makefiles.

Each directory may have one or more types of subdirectories:

- (normal) dirs
- tests

class SubDirectories
Bases: mozbuild.backend.recursivemake.SubDirectories

RecursiveMakeTraversal.**SubDirectoriesTuple**
alias of *SubDirectories*

RecursiveMakeTraversal.**SubDirectoryCategories = [u'dirs', u'tests']**

RecursiveMakeTraversal.**add**(*dir*, *dirs=[]*, *tests=[]*)
Adds a directory to traversal, registering its subdirectories, sorted by categories. If the directory was already added to traversal, adds the new subdirectories to the already known lists.

RecursiveMakeTraversal.**call_filter**(*current*, *filter*)
Helper function to call a filter from compute_dependencies and traverse.

RecursiveMakeTraversal.**compute_dependencies**(*filter=None*)
Compute make dependencies corresponding to the registered directory traversal.

filter is a function with the following signature: def filter(current, subdirs)

where current is the directory being traversed, and subdirs the SubDirectories instance corresponding to it. The filter function returns a tuple (filtered_current, filtered_parallel, filtered_dirs) where filtered_current is either current or None if the current directory is to be skipped, and filtered_parallel and filtered_dirs are lists of parallel directories and sequential directories, which can be rearranged from whatever is given in the SubDirectories members.

The default filter corresponds to a default recursive traversal.

static RecursiveMakeTraversal.**default_filter**(*current*, *subdirs*)
Default filter for use with compute_dependencies and traverse.

RecursiveMakeTraversal.**get_subdirs**(*dir*)
Returns all direct subdirectories under the given directory.

RecursiveMakeTraversal.**traverse**(*start*, *filter=None*)
Iterate over the filtered subdirectories, following the traditional make traversal order.

mozbuild.backend.recursivemake.**make_quote**(*s*)

mozbuild.backend.visualstudio module

class mozbuild.backend.visualstudio.**VisualStudioBackend**(*environment*)
Bases: *mozbuild.backend.common.CommonBackend*

Generate Visual Studio project files.

This backend is used to produce Visual Studio projects and a solution to foster developing Firefox with Visual Studio.

This backend is currently considered experimental. There are many things not optimal about how it works.

consume_finished()

consume_object(*obj*)

summary()

static **write_vs_project**(*fh*, *version*, *project_id*, *name*, *includes=[]*, *forced_includes=[]*, *defines=[]*, *build_command=None*, *clean_command=None*, *debugger=None*, *headers=[]*, *sources=[]*)

mozbuild.backend.visualstudio.**get_id**(*name*)

mozbuild.backend.visualstudio.**visual_studio_product_to_platform_toolset_version**(*version*)

mozbuild.backend.visualstudio.**visual_studio_product_to_solution_version**(*version*)

Module contents

mozbuild.backend.**get_backend_class**(*name*)

mozbuild.codecoverage package

Submodules

mozbuild.codecoverage.chrome_map module

class mozbuild.codecoverage.chrome_map.**ChromeManifestHandler**
Bases: object

handle_manifest_entry(*entry*)

class mozbuild.codecoverage.chrome_map.**ChromeMapBackend**(*environment*)
Bases: *mozbuild.backend.common.CommonBackend*

consume_finished()

consume_object(*obj*)

mozbuild.codecoverage.packager module

mozbuild.codecoverage.packager.**cli**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

mozbuild.codecoverage.packager.**package_gcno_tree**(*root*, *output_file*)

Module contents

mozbuild.compilation package

Submodules

mozbuild.compilation.codecomplete module

mozbuild.compilation.database module

class mozbuild.compilation.database.**CompileDBBackend**(*environment*)
Bases: *mozbuild.backend.common.CommonBackend*

CFLAGS = {'.c': 'CFLAGS', '.mm': 'CXXFLAGS', '.cpp': 'CXXFLAGS', '.m': 'CFLAGS'}

COMPILERS = {'.c': 'CC', '.mm': 'CXX', '.cpp': 'CXX', '.m': 'CC'}

consume_finished()

consume_object(*obj*)

mozbuild.compilation.util module

mozbuild.compilation.util.**check_top_objdir**(*topobjdir*)

mozbuild.compilation.util.**get_build_vars**(*directory*, *cmd*)

mozbuild.compilation.util.**sanitize_cflags**(*flags*)

mozbuild.compilation.warnings module

class mozbuild.compilation.warnings.**CompilerWarning**
Bases: dict

Represents an individual compiler warning.

class mozbuild.compilation.warnings.**WarningsCollector**(*database=None*, *objdir=None*, *resolve_files=True*)
Bases: object

Collects warnings from text data.

Instances of this class receive data (usually the output of compiler invocations) and parse it into warnings and add these warnings to a database.

The collector works by incrementally receiving data, usually line-by-line output from the compiler. Therefore, it can maintain state to parse multi-line warning messages.

process_line(*line*)
Take a line of text and process it for a warning.

class mozbuild.compilation.warnings.**WarningsDatabase**
Bases: object

Holds a collection of warnings.

The warnings database is a semi-intelligent container that holds warnings encountered during builds.

The warnings database is backed by a JSON file. But, that is transparent to consumers.

Under most circumstances, the warnings database is insert only. When a warning is encountered, the caller simply blindly inserts it into the database. The database figures out whether it is a dupe, etc.

During the course of development, it is common for warnings to change slightly as source code changes. For example, line numbers will disagree. The WarningsDatabase handles this by storing the hash of a file a warning occurred in. At warning insert time, if the hash of the file does not match what is stored in the database, the existing warnings for that file are purged from the database.

Callers should periodically prune old, invalid warnings from the database by calling prune(). A good time to do this is at the end of a build.

deserialize(*fh*)
Load serialized content from a handle into the current instance.

has_file(*filename*)
Whether we have any warnings for the specified file.

insert(*warning*, *compute_hash=True*)

load_from_file(*filename*)
Load the database from a file.

prune()
Prune the contents of the database.

This removes warnings that are no longer valid. A warning is no longer valid if the file it was in no longer exists or if the content has changed.

The check for changed content catches the case where a file previously contained warnings but no longer does.

save_to_file(*filename*)
Save the database to a file.

serialize(*fh*)
Serialize the database to an open file handle.

type_counts(*dirpath=None*)
Returns a mapping of warning types to their counts.

warnings
All the CompilerWarning instances in this database.

warnings_for_file(*filename*)
Obtain the warnings for the specified file.

Module contents

mozbuild.configure package

Submodules

mozbuild.configure.check_debug_ranges module

mozbuild.configure.check_debug_ranges.**get_range_for**(*compilation_unit*, *debug_info*)
Returns the range offset for a given compilation unit in a given debug_info.

mozbuild.configure.check_debug_ranges.**get_range_length**(*range*, *debug_ranges*)
Returns the number of items in the range starting at the given offset.

mozbuild.configure.check_debug_ranges.**main**(*bin*, *compilation_unit*)

mozbuild.configure.constants module

mozbuild.configure.help module

class mozbuild.configure.help.**HelpFormatter**(*argv0*)
Bases: object

add(*option*)

usage(*out*)

mozbuild.configure.libstdcxx module

mozbuild.configure.libstdcxx.**cmp_ver**(*a*, *b*)
Compare versions in the form 'a.b.c'

mozbuild.configure.libstdcxx.**encode_ver**(*v*)
Encode the version as a single number.

mozbuild.configure.libstdcxx.**find_version**(*e*)
Given the value of environment variable CXX or HOST_CXX, find the version of the libstdc++ it uses.

mozbuild.configure.libstdcxx.**parse_ld_line**(*x*)
Parse a line from the output of ld -t. The output of gold is just the full path, gnu ld prints "-lstdc++ (path)".

mozbuild.configure.libstdcxx.**parse_readelf_line**(*x*)
Return the version from a readelf line that looks like: 0x00ec: Rev: 1 Flags: none Index: 8 Cnt: 2 Name: GLIBCXX_3.4.6

mozbuild.configure.libstdcxx.**split_ver**(*v*)
Covert the string '1.2.3' into the list [1,2,3]

mozbuild.configure.options module

class mozbuild.configure.options.**CommandLineHelper**(*environ={'LANG': 'C.UTF-8', 'READTHEDOCS_PROJECT': 'gfritzsche-demo', 'READTHEDOCS': 'True', 'APPDIR': '/app', 'DEBIAN_FRONTEND': 'noninteractive', 'OLDPWD': '/', 'HOSTNAME': 'build-4258433-project-55928-gfritzsche-demo', u'SHELL': u'/bin/bash', 'PWD': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche*
demo/checkouts/latest/tools/docs', 'BIN_PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche
demo/envs/latest/bin', 'READTHEDOCS_VERSION': 'latest', 'PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche
demo/checkouts/latest/tools/docs/_build/latex/_venv/bin:/home
demo/envs/latest/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/u
'HOME': '/home/docs'}, argv=['/home/docs/checkouts/readthedocs.org/user_builds/gfr
demo/envs/latest/bin/sphinx-build', '-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex'])
Bases: object

Helper class to handle the various ways options can be given either on the command line of through the environment.

For instance, an Option('–foo', env='FOO') can be passed as –foo on the command line, or as FOO=1 in the environment *or* on the command line.

If multiple variants are given, command line is prefered over the environment, and if different values are given on the command line, the last one wins. (This mimicks the behavior of autoconf, avoiding to break existing mozconfigs using valid options in weird ways)

Extra options can be added afterwards through API calls. For those, conflicting values will raise an exception.

add(*arg*, *origin=u'command-line'*, *args=None*)

handle(*option*)
Return the OptionValue corresponding to the given Option instance, depending on the command line, environment, and extra arguments, and the actual option or variable that set it. Only works once for a given Option.

exception mozbuild.configure.options.**ConflictingOptionError**(*message*, ***format_data*)
Bases: *mozbuild.configure.options.InvalidOptionError*

exception mozbuild.configure.options.**InvalidOptionError**
Bases: exceptions.Exception

class mozbuild.configure.options.**NegativeOptionValue**(*origin=u'unknown'*)
Bases: *mozbuild.configure.options.OptionValue*

Represents the value for a negative option (–disable/–without)

This is effectively an empty tuple with a *origin* attribute.

class mozbuild.configure.options.**Option**(*name=None*, *env=None*, *nargs=None*, *default=None*, *possible_origins=None*, *choices=None*, *help=None*)
Bases: object

Represents a configure option

A configure option can be a command line flag or an environment variable or both.

- *name* is the full command line flag (e.g. –enable-foo).
- *env* is the environment variable name (e.g. ENV)
- *nargs* is the number of arguments the option may take. It can be a number or the special values '?' (0 or 1), '*' (0 or more), or '+' (1 or more).
- *default* can be used to give a default value to the option. When the *name* of the option starts with '–enable-' or '–with-', the implied default is an empty PositiveOptionValue. When it starts with '–disable-' or '–without-', the implied default is a NegativeOptionValue.
- *choices* restricts the set of values that can be given to the option.
- *help* is the option description for use in the –help output.
- *possible_origins* is a tuple of strings that are origins accepted for this option. Example origins are 'mozconfig', 'implied', and 'environment'.

choices

default

env

get_value(*option=None*, *origin=u'unknown'*)
Given a full command line option (e.g. –enable-foo=bar) or a variable assignment (FOO=bar), returns the corresponding OptionValue.

Note: variable assignments can come from either the environment or from the command line (e.g. *../configure CFLAGS=-O2*)

help

id

maxargs

minargs

name

nargs

option

possible_origins

prefix

static split_option(*option*)
Split a flag or variable into a prefix, a name and values

Variables come in the form NAME=values (no prefix). Flags come in the form –name=values or –prefix-name=values where prefix is one of 'with', 'without', 'enable' or 'disable'. The '=values' part is optional. Values are separated with commas.

class mozbuild.configure.options.**OptionValue**(*values=()*, *origin=u'unknown'*)
Bases: tuple

Represents the value of a configure option.

This class is not meant to be used directly. Use its subclasses instead.

The *origin* attribute holds where the option comes from (e.g. environment, command line, or default)

format(*option*)

class mozbuild.configure.options.**PositiveOptionValue**(*values=()*, *origin=u'unknown'*)
Bases: *mozbuild.configure.options.OptionValue*

Represents the value for a positive option (–enable/–with/–foo) in the form of a tuple for when values are given to the option (in the form –option=value[,value2...].

mozbuild.configure.options.**istupleofstrings**(*obj*)

mozbuild.configure.util module

class mozbuild.configure.util.**ConfigureOutputHandler**(*stdout=<open file '<stdout>', mode 'w'>*, *stderr=<open file '<stderr>', mode 'w'>*, *maxlen=20*)
Bases: logging.Handler

A logging handler class that sends info messages to stdout and other messages to stderr.

Messages sent to stdout are not formatted with the attached Formatter. Additionally, if they end with '... ', no newline character is printed, making the next message printed follow the '... '.

Only messages above log level INFO (included) are logged.

Messages below that level can be kept until an ERROR message is received, at which point the last *maxlen* accumulated messages below INFO are printed out. This feature is only enabled under the *queue_debug* context manager.

INTERRUPTED = 2

KEEP = 1

PRINT = 2

THROW = 0

WAITING = 1

emit(*record*)

queue_debug(**args*, ***kwds*)

class mozbuild.configure.util.**LineIO**(*callback*)
Bases: object

File-like class that sends each line of the written data to a callback (without carriage returns).

close()

write(*buf*)

class mozbuild.configure.util.**Version**(*version*)
Bases: distutils.version.LooseVersion

A simple subclass of distutils.version.LooseVersion. Adds attributes for *major*, *minor*, *patch* for the first three version components so users can easily pull out major/minor versions, like:

v = Version('1.2b') v.major == 1 v.minor == 2 v.patch == 0

mozbuild.configure.util.**getpreferredencoding**()

Module contents

exception mozbuild.configure.**ConfigureError**
Bases: exceptions.Exception

class mozbuild.configure.**ConfigureSandbox**(*config, environ={'LANG': 'C.UTF-8', 'READTHEDOCS_PROJECT': 'gfritzsche-demo', 'READTHEDOCS': 'True', 'APPDIR': '/app', 'DEBIAN_FRONTEND': 'noninteractive', 'OLDPWD': '/', 'HOSTNAME': 'build-4258433-project-55928-gfritzsche-demo', u'SHELL': u'/bin/bash', 'PWD': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs', 'BIN_PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/envs/latest/bin', 'READTHEDOCS_VERSION': 'latest', 'PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs/_build/latex/_venv/bin:/home/docs/checkou demo/envs/latest/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/ 'HOME': '/home/docs'}, argv=['/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/envs/latest/bin/sphinx-build', '-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex'], stdout=<open file '<stdout>', mode 'w'>, stderr=<open file '<stderr>', mode 'w'>, logger=None*)
Bases: dict

Represents a sandbox for executing Python code for build configuration. This is a different kind of sandboxing than the one used for moz.build processing.

The sandbox has 8 primitives: - option - depends - template - imports - include - set_config - set_define - imply_option

option, *include*, *set_config*, *set_define* and *imply_option* are functions. *depends*, *template*, and *imports* are decorators.

These primitives are declared as name_impl methods to this class and the mapping name -> name_impl is done automatically in __getitem__.

Additional primitives should be frowned upon to keep the sandbox itself as simple as possible. Instead, helpers should be created within the sandbox with the existing primitives.

The sandbox is given, at creation, a dict where the yielded configuration will be stored.

config = { } sandbox = ConfigureSandbox(config) sandbox.run(path) do_stuff(config)

BUILTINS = {u'None': None, u'set': <type 'set'>, u'tuple': <type 'tuple'>, u'int': <type 'int'>, '__import__': <function f

OS = <ReadOnlyNamespace {'path': <ReadOnlyNamespace {'isdir': <function isdir at 0x7f862856d1b8>, 'realpath': <fu

RE_MODULE = <_sre.SRE_Pattern object>

depends_impl(**args*)

Implementation of @depends() This function is a decorator. It returns a function that subsequently takes a function and returns a dummy function. The dummy function identifies the actual function for the sandbox, while preventing further function calls from within the sandbox.

@depends() takes a variable number of option strings or dummy function references. The decorated function is called as soon as the decorator is called, and the arguments it receives are the OptionValue or function results corresponding to each of the arguments to @depends. As an exception, when a HelpFormatter is attached, only functions that have '–help' in their @depends argument list are called.

The decorated function is altered to use a different global namespace for its execution. This different global namespace exposes a limited set of functions from os.path.

imply_option_impl(*option*, *value*, *reason=None*)

Implementation of imply_option(). Injects additional options as if they had been passed on the command line. The *option* argument is a string as in option()'s *name* or *env*. The option must be declared after *imply_option* references it. The *value* argument indicates the value to pass to the option. It can be: - True. In this case *imply_option* injects the positive option

(–enable-foo/–with-foo). imply_option('–enable-foo', True) imply_option('–disable-foo', True)

are both equivalent to *–enable-foo* on the command line.

- False. In this case *imply_option* injects the negative option (–disable-foo/–without-foo).

  imply_option('–enable-foo', False) imply_option('–disable-foo', False)

  are both equivalent to *–disable-foo* on the command line.

- **None. In this case *imply_option* does nothing.** imply_option('–enable-foo', None) imply_option('–disable-foo', None)

  are both equivalent to not passing any flag on the command line.

- a string or a tuple. In this case *imply_option* injects the positive option with the given value(s).

  imply_option('–enable-foo', 'a') imply_option('–disable-foo', 'a')

  are both equivalent to *–enable-foo=a* on the command line. imply_option('–enable-foo', ('a', 'b')) imply_option('–disable-foo', ('a', 'b'))

  are both equivalent to *–enable-foo=a,b* on the command line.

Because imply_option('–disable-foo', ...) can be misleading, it is recommended to use the positive form ('–enable' or '–with') for *option*.

The *value* argument can also be (and usually is) a reference to a @depends function, in which case the result of that function will be used as per the descripted mapping above.

The *reason* argument indicates what caused the option to be implied. It is necessary when it cannot be inferred from the *value*.

imports_impl (*_import*, *_from=None*, *_as=None*)
Implementation of @imports. This decorator imports the given _import from the given _from module optionally under a different _as name. The options correspond to the various forms for the import builtin.

@imports('sys') @imports(_from='mozpack', _import='path', _as='mozpath')

include_file (*path*)
Include one file in the sandbox. Users of this class probably want

Note: this will execute all template invocations, as well as @depends functions that depend on '–help', but nothing else.

include_impl (*what*)
Implementation of include(). Allows to include external files for execution in the sandbox. It is possible to use a @depends function as argument, in which case the result of the function is the file name to include. This latter feature is only really meant for –enable-application/–enable-project.

option_impl (**args*, ***kwargs*)
Implementation of option() This function creates and returns an Option() object, passing it the resolved arguments (uses the result of functions when functions are passed). In most cases, the result of this function is not expected to be used. Command line argument/environment variable parsing for this Option is handled here.

run (*path=None*)
Executes the given file within the sandbox, as well as everything pending from any other included file, and ensure the overall consistency of the executed script(s).

set_config_impl (*name*, *value*)
Implementation of set_config(). Set the configuration items with the given name to the given value. Both *name* and *value* can be references to @depends functions, in which case the result from these functions is used. If the result of either function is None, the configuration item is not set.

set_define_impl (*name*, *value*)
Implementation of set_define(). Set the define with the given name to the given value. Both *name* and *value* can be references to @depends functions, in which case the result from these functions is used. If the result of either function is None, the define is not set. If the result is False, the define is explicitly undefined (-U).

template_impl (*func*)
Implementation of @template. This function is a decorator. Template functions are called immediately. They are altered so that their global namespace exposes a limited set of functions from os.path, as well as *depends* and *option*. Templates allow to simplify repetitive constructs, or to implement helper decorators and somesuch.

class mozbuild.configure.**DependsFunction**
Bases: object

Sandbox-visible representation of @depends functions.

class mozbuild.configure.**SandboxedGlobal**
Bases: dict

Identifiable dict type for use as function global

mozbuild.configure.**forbidden_import**(**args*, ***kwargs*)

mozbuild.controller package

Submodules

mozbuild.controller.building module

class mozbuild.controller.building.**BuildDriver**(*topsrcdir*, *settings*, *log_manager*, *topobjdir=None*, *mozconfig=<object object>*)

Bases: *mozbuild.base.MozbuildObject*

Provides a high-level API for build actions.

install_tests(*test_objs*)
Install test files.

class mozbuild.controller.building.**BuildMonitor**(*topsrcdir*, *settings*, *log_manager*, *topobjdir=None*, *mozconfig=<object object>*)

Bases: *mozbuild.base.MozbuildObject*

Monitors the output of the build.

ccache_stats()

finish(*record_usage=True*)
Record the end of the build.

get_resource_usage()
Produce a data structure containing the low-level resource usage information.

This data structure can e.g. be serialized into JSON and saved for subsequent analysis.

If no resource usage is available, None is returned.

have_excessive_swapping()
Determine whether there was excessive swapping during the build.

Returns a tuple of (excessive, swap_in, swap_out). All values are None if no swap information is available.

have_high_finder_usage()
Determine whether there was high Finder CPU usage during the build.

Returns True if there was high Finder CPU usage, False if there wasn't, or None if there is nothing to report.

have_resource_usage
Whether resource usage is available.

init(*warnings_path*)
Create a new monitor.

warnings_path is a path of a warnings database to use.

log_resource_usage(*usage*)
Summarize the resource usage of this build in a log message.

on_line(*line*)
Consume a line of output from the build system.

This will parse the line for state and determine whether more action is needed.

Returns a BuildOutputResult instance.

In this named tuple, warning will be an object describing a new parsed warning. Otherwise it will be None.

state_changed indicates whether the build system changed state with this line. If the build system changed state, the caller may want to query this instance for the current state in order to update UI, etc.

for_display is a boolean indicating whether the line is relevant to the user. This is typically used to filter whether the line should be presented to the user.

start()
Record the start of the build.

start_resource_recording()

class mozbuild.controller.building.**BuildOutputResult**(*warning*, *state_changed*, *for_display*)
Bases: tuple

for_display
Alias for field number 2

state_changed
Alias for field number 1

warning
Alias for field number 0

class mozbuild.controller.building.**CCacheStats**(*output=None*)
Bases: object

Holds statistics from ccache.

Instances can be subtracted from each other to obtain differences. print() or str() the object to show a `ccache -s` like output of the captured stats.

ABSOLUTE_KEYS = set([u'cache_size', u'cache_files', u'cache_max_size'])

DIRECTORY_DESCRIPTION = u'cache directory'

FORMAT_KEYS = set([u'cache_size', u'cache_max_size'])

GiB = 1073741824

KiB = 1024

MiB = 1048576

PRIMARY_CONFIG_DESCRIPTION = u'primary config'

SECONDARY_CONFIG_DESCRIPTION = u'secondary config (readonly)'

STATS_KEYS = [(u'cache_hit_direct', u'cache hit (direct)'), (u'cache_hit_preprocessed', u'cache hit (preprocessed)'), (u'c

hit_rate_message()

hit_rates()

class mozbuild.controller.building.**TierStatus**(*resources*)
Bases: object

Represents the state and progress of tier traversal.

The build system is organized into linear phases called tiers. Each tier executes in the order it was defined, 1 at a time.

add_resource_fields_to_dict(*d*)

add_resources_to_dict(*entry*, *start=None*, *end=None*, *phase=None*)
Helper function to append resource information to a dict.

begin_tier(*tier*)
Record that execution of a tier has begun.

finish_tier(*tier*)
Record that execution of a tier has finished.

set_tiers(*tiers*)
Record the set of known tiers.

tiered_resource_usage()
Obtains an object containing resource usage for tiers.

The returned object is suitable for serialization.

mozbuild.controller.clobber module

class mozbuild.controller.clobber.**Clobberer**(*topsrcdir*, *topobjdir*)
Bases: object

clobber_cause()
Obtain the cause why a clobber is required.

This reads the cause from the CLOBBER file.

This returns a list of lines describing why the clobber was required. Each line is stripped of leading and trailing whitespace.

clobber_needed()
Returns a bool indicating whether a tree clobber is required.

ensure_objdir_state()
Ensure the CLOBBER file in the objdir exists.

This is called as part of the build to ensure the clobber information is configured properly for the objdir.

maybe_do_clobber(*cwd*, *allow_auto=False*, *fh=<open file '<stderr>', mode 'w'>*)
Perform a clobber if it is required. Maybe.

This is the API the build system invokes to determine if a clobber is needed and to automatically perform that clobber if we can.

This returns a tuple of (bool, bool, str). The elements are:

- Whether a clobber was/is required.
- Whether a clobber was performed.
- The reason why the clobber failed or could not be performed. This will be None if no clobber is required or if we clobbered without error.

mozbuild.controller.clobber.**main**(*args*, *env*, *cwd*, *fh=<open file '<stderr>', mode 'w'>*)

Module contents

mozbuild.frontend package

Submodules

mozbuild.frontend.context module

This module contains the data structure (context) holding the configuration from a moz.build. The data emitted by the frontend derives from those contexts.

It also defines the set of variables and functions available in moz.build. If you are looking for the absolute authority on what moz.build files can contain, you've come to the right place.

class mozbuild.frontend.context.**AbsolutePath**(*context*, *value=None*)
Bases: *mozbuild.frontend.context.Path*

Like Path, but allows arbitrary paths outside the source and object directories.

class mozbuild.frontend.context.**Context**(*allowed_variables={}*, *config=None*, *finder=None*)
Bases: *mozbuild.util.KeyedDefaultDict*

Represents a moz.build configuration context.

Instances of this class are filled by the execution of sandboxes. At the core, a Context is a dict, with a defined set of possible keys we'll call variables. Each variable is associated with a type.

When reading a value for a given key, we first try to read the existing value. If a value is not found and it is defined in the allowed variables set, we return a new instance of the class for that variable. We don't assign default instances until they are accessed because this makes debugging the end-result much simpler. Instead of a data structure with lots of empty/default values, you have a data structure with only the values that were read or touched.

Instances of variables classes are created by invoking `class_name()`, except when class_name derives from `ContextDerivedValue` or `SubContext`, in which case `class_name(instance_of_the_context)` or `class_name(self)` is invoked. A value is added to those calls when instances are created during assignment (setitem).

allowed_variables is a dict of the variables that can be set and read in this context instance. Keys in this dict are the strings representing keys in this context which are valid. Values are tuples of stored type, assigned type, default value, a docstring describing the purpose of the variable, and a tier indicator (see comment above the VARIABLES declaration in this module).

config is the ConfigEnvironment for this context.

add_source(*path*)
Adds the given path as source of the data from this context.

all_paths
Returns all paths ever added to the context.

error_is_fatal
Returns True if the error function should be fatal.

pop_source()
Get back to the previous current path for the context.

push_source(*path*)
Adds the given path as source of the data from this context and make it the current path for the context.

relsrcdir

source_stack
Returns the current stack of pushed sources.

srcdir

update(*iterable={}*, ***kwargs*)
Like dict.update(), but using the context's setitem.

This function is transactional: if setitem fails for one of the values, the context is not updated at all.

mozbuild.frontend.context.**ContextDerivedTypedHierarchicalStringList**
Specialized HierarchicalStringList for use with ContextDerivedValue types.

mozbuild.frontend.context.**ContextDerivedTypedList**
Specialized TypedList for use with ContextDerivedValue types.

mozbuild.frontend.context.**ContextDerivedTypedListWithItems**
Specialized TypedList for use with ContextDerivedValue types.

mozbuild.frontend.context.**ContextDerivedTypedRecord**
Factory for objects with certain properties and dynamic type checks.

This API is extremely similar to the TypedNamedTuple API, except that properties may be mutated. This supports syntax like:

VARIABLE_NAME.property += ['item1', 'item2',

]

class mozbuild.frontend.context.**ContextDerivedValue**
Bases: object

Classes deriving from this one receive a special treatment in a Context. See Context documentation.

mozbuild.frontend.context.**DependentTestsEntry**
alias of _TypedRecord

mozbuild.frontend.context.**Enum**(**values*)

class mozbuild.frontend.context.**Files**(*parent*, *pattern=None*)
Bases: *mozbuild.frontend.context.SubContext*

Metadata attached to files.

It is common to want to annotate files with metadata, such as which Bugzilla component tracks issues with certain files. This sub-context is where we stick that metadata.

The argument to this sub-context is a file matching pattern that is applied against the host file's directory. If the pattern matches a file whose info is currently being sought, the metadata attached to this instance will be applied to that file.

Patterns are collections of filename characters with / used as the directory separate (UNIX-style paths) and * and ** used to denote wildcard matching.

Patterns without the * character are literal matches and will match at most one entity.

Patterns with * or ** are wildcard matches. * matches files at least within a single directory. ** matches files across several directories.

foo.html Will match only the foo.html file in the current directory.

***.jsm** Will match all .jsm files in the current directory.

****/*.cpp** Will match all .cpp files in this and all child directories.

foo/*.css Will match all .css files in the foo/ directory.

bar/* Will match all files in the `bar/` directory and all of its children directories.

bar/** This is equivalent to `bar/*` above.

bar//foo** Will match all `foo` files in the `bar/` directory and all of its children directories.

The difference in behavior between `*` and `**` is only evident if a pattern follows the `*` or `**`. A pattern ending with `*` is greedy. `**` is needed when you need an additional pattern after the wildcard. e.g. `**/foo`.

VARIABLES = {u'BUG_COMPONENT': (<class 'mozbuild.util.TypedTuple'>, <type 'tuple'>, u"The bug component that

static aggregate(*files*)
Given a mapping of path to Files, obtain aggregate results.

Consumers may want to extract useful information from a collection of Files describing paths. e.g. given the files info data for N paths, recommend a single bug component based on the most frequent one. This function provides logic for deriving aggregate knowledge from a collection of path File metadata.

Note: the intent of this function is to operate on the result of *mozbuild.frontend.reader.BuildReader.files_info()*. The *mozbuild.frontend.context.Files()* instances passed in are thus the "collapsed" (__iadd__``ed) results of all ``Files from all moz.build files relevant to a specific path, not individual `Files` instances from a single moz.build file.

asdict()
Return this instance as a dict with built-in data structures.

Call this to obtain an object suitable for serializing.

class mozbuild.frontend.context.**FinalTargetValue**
Bases: *mozbuild.frontend.context.ContextDerivedValue*, unicode

class mozbuild.frontend.context.**InitializedDefines**(*context*, *value=None*)
Bases: *mozbuild.frontend.context.ContextDerivedValue*, collections.OrderedDict

mozbuild.frontend.context.**ManifestparserManifestList**
alias of _OrderedListWithAction

class mozbuild.frontend.context.**ObjDirPath**(*context*, *value=None*)
Bases: *mozbuild.frontend.context.Path*

Like Path, but limited to paths in the object directory.

mozbuild.frontend.context.**OrderedListWithAction**(*action*)
Returns a class which behaves as a StrictOrderingOnAppendList, but invokes the given callable with each input and a context as it is read, storing a tuple including the result and the original item.

This used to extend moz.build reading to make more data available in filesystem-reading mode.

mozbuild.frontend.context.**OrderedSourceList**
alias of _TypedList

class mozbuild.frontend.context.**Path**(*context*, *value=None*)
Bases: *mozbuild.frontend.context.ContextDerivedValue*, unicode

Stores and resolves a source path relative to a given context

This class is used as a backing type for some of the sandbox variables. It expresses paths relative to a context. Supported paths are:

- '/topsrcdir/relative/paths'
- 'srcdir/relative/paths'
- '!/topobjdir/relative/paths'

- '!objdir/relative/paths'
- '%/filesystem/absolute/paths'

join(**p*)
ContextDerived equivalent of mozpath.join(self, *p), returning a new Path instance.

class mozbuild.frontend.context.**PathMeta**
Bases: type

Meta class for the Path family of classes.

It handles calling __new__ and __init__ with the right arguments in cases where a Path is instantiated with another instance of Path instead of having received a context.

It also makes Path(context, value) instantiate one of the subclasses depending on the value, allowing callers to do standard type checking (isinstance(path, ObjDirPath)) instead of checking the value itself (path.startswith('!')).

mozbuild.frontend.context.**ReftestManifestList**
alias of _OrderedListWithAction

class mozbuild.frontend.context.**RenamedSourcePath**(*context*, *value*)
Bases: *mozbuild.frontend.context.SourcePath*

Like SourcePath, but with a different base name when installed.

The constructor takes a tuple of (source, target_basename).

This class is not meant to be exposed to moz.build sandboxes as of now, and is not supported by the RecursiveMake backend.

target_basename

class mozbuild.frontend.context.**SourcePath**(*context*, *value*)
Bases: *mozbuild.frontend.context.Path*

Like Path, but limited to paths in the source directory.

class mozbuild.frontend.context.**SubContext**(*parent*)
Bases: *mozbuild.frontend.context.Context*, *mozbuild.frontend.context.ContextDerivedValue*

A Context derived from another Context.

Sub-contexts are intended to be used as context managers.

Sub-contexts inherit paths and other relevant state from the parent context.

class mozbuild.frontend.context.**TemplateContext**(*template=None*, *allowed_variables={}*, *config=None*)
Bases: *mozbuild.frontend.context.Context*

mozbuild.frontend.context.**TypedListWithAction**(*typ*, *action*)
Returns a class which behaves as a TypedList with the provided type, but invokes the given given callable with each input and a context as it is read, storing a tuple including the result and the original item.

This used to extend moz.build reading to make more data available in filesystem-reading mode.

mozbuild.frontend.context.**WptManifestList**
alias of _TypedListWithAction

mozbuild.frontend.context.**cls**
alias of *Files*

mozbuild.frontend.data module

Data structures representing Mozilla's source tree.

The frontend files are parsed into static data structures. These data structures are defined in this module.

All data structures of interest are children of the TreeMetadata class.

Logic for populating these data structures is not defined in this class. Instead, what we have here are dumb container classes. The emitter module contains the code for converting executed mozbuild files into these data structures.

class mozbuild.frontend.data.**AndroidAssetsDirs**(*context*, *paths*)
Bases: *mozbuild.frontend.data.ContextDerived*

Represents Android assets directories.

paths

class mozbuild.frontend.data.**AndroidEclipseProjectData**(*name*)
Bases: object

Represents an Android Eclipse project.

add_classpathentry(*path*, *srcdir*, *dstdir*, *exclude_patterns=[]*, *ignore_warnings=False*)

assets

extra_jars

filtered_resources

included_projects

is_library

libs

manifest

name

package_name

recursive_make_targets

referenced_projects

res

class mozbuild.frontend.data.**AndroidExtraPackages**(*context*, *packages*)
Bases: *mozbuild.frontend.data.ContextDerived*

Represents Android extra packages.

packages

class mozbuild.frontend.data.**AndroidExtraResDirs**(*context*, *paths*)
Bases: *mozbuild.frontend.data.ContextDerived*

Represents Android extra resource directories.

Extra resources are resources provided by libraries and including in a packaged APK, but not otherwise redistributed. In practice, this means resources included in Fennec but not in GeckoView.

paths

class mozbuild.frontend.data.**AndroidResDirs**(*context*, *paths*)
Bases: *mozbuild.frontend.data.ContextDerived*

Represents Android resource directories.

paths

class mozbuild.frontend.data.**BaseConfigSubstitution**(*context*)
Bases: *mozbuild.frontend.data.ContextDerived*

Base class describing autogenerated files as part of config.status.

input_path

output_path

relpath

class mozbuild.frontend.data.**BaseDefines**(*context*, *defines*)
Bases: *mozbuild.frontend.data.ContextDerived*

Context derived container object for DEFINES/HOST_DEFINES, which are OrderedDicts.

defines

get_defines()

update(*more_defines*)

class mozbuild.frontend.data.**BaseLibrary**(*context*, *basename*)
Bases: *mozbuild.frontend.data.Linkable*

Generic context derived container object for libraries.

basename

import_name

lib_name

refs

class mozbuild.frontend.data.**BaseProgram**(*context*, *program*, *is_unit_test=False*)
Bases: *mozbuild.frontend.data.Linkable*

Context derived container object for programs, which is a unicode string.

This class handles automatically appending a binary suffix to the program name. If the suffix is not defined, the program name is unchanged. Otherwise, if the program name ends with the given suffix, it is unchanged Otherwise, the suffix is appended to the program name.

DICT_ATTRS = set([u'relobjdir', u'install_target', u'KIND', u'program'])

program

class mozbuild.frontend.data.**BaseSources**(*context*, *files*, *canonical_suffix*)
Bases: *mozbuild.frontend.data.ContextDerived*

Base class for files to be compiled during the build.

canonical_suffix

files

class mozbuild.frontend.data.**BrandingFiles**(*sandbox*, *files*)
Bases: *mozbuild.frontend.data.FinalTargetFiles*

Sandbox container object for BRANDING_FILES, which is a HierarchicalStringList.

We need an object derived from ContextDerived for use in the backend, so this object fills that role. It just has a reference to the underlying HierarchicalStringList, which is created when parsing BRANDING_FILES.

install_target

class mozbuild.frontend.data.**ChromeManifestEntry**(*context*, *manifest_path*, *entry*)
Bases: *mozbuild.frontend.data.ContextDerived*

Represents a chrome.manifest entry.

entry

path

class mozbuild.frontend.data.**ClassPathEntry**
Bases: object

Represents a classpathentry in an Android Eclipse project.

dstdir

exclude_patterns

ignore_warnings

path

srcdir

class mozbuild.frontend.data.**ConfigFileSubstitution**(*context*)
Bases: *mozbuild.frontend.data.BaseConfigSubstitution*

Describes a config file that will be generated using substitutions.

class mozbuild.frontend.data.**ContextDerived**(*context*)
Bases: *mozbuild.frontend.data.TreeMetadata*

Build object derived from a single Context instance.

It holds fields common to all context derived classes. This class is likely never instantiated directly but is instead derived from.

config

context_all_paths

context_main_path

defines

install_target

objdir

relativedir

relobjdir

srcdir

topobjdir

topsrcdir

class mozbuild.frontend.data.**ContextWrapped**(*context*, *wrapped*)
Bases: *mozbuild.frontend.data.ContextDerived*

Generic context derived container object for a wrapped rich object.

Use this wrapper class to shuttle a rich build system object completely defined in moz.build files through the tree metadata emitter to the build backend for processing as-is.

wrapped

class mozbuild.frontend.data.**Defines**(*context*, *defines*)
Bases: *mozbuild.frontend.data.BaseDefines*

class mozbuild.frontend.data.**DirectoryTraversal**(*context*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes how directory traversal for building should work.

This build object is likely only of interest to the recursive make backend. Other build backends should (ideally) not attempt to mimic the behavior of the recursive make backend. The only reason this exists is to support the existing recursive make backend while the transition to mozbuild frontend files is complete and we move to a more optimal build backend.

Fields in this class correspond to similarly named variables in the frontend files.

dirs

class mozbuild.frontend.data.**ExampleWebIDLInterface**(*context*, *name*)
Bases: *mozbuild.frontend.data.ContextDerived*

An individual WebIDL interface to generate.

name

class mozbuild.frontend.data.**Exports**(*sandbox*, *files*)
Bases: *mozbuild.frontend.data.FinalTargetFiles*

Context derived container object for EXPORTS, which is a HierarchicalStringList.

We need an object derived from ContextDerived for use in the backend, so this object fills that role. It just has a reference to the underlying HierarchicalStringList, which is created when parsing EXPORTS.

install_target

class mozbuild.frontend.data.**ExternalLibrary**
Bases: object

Empty mixin for libraries built by an external build system.

class mozbuild.frontend.data.**ExternalSharedLibrary**(*context*, *basename*, *real_name=None*, *is_sdk=False*, *soname=None*, *variant=None*, *symbols_file=False*)
Bases: *mozbuild.frontend.data.SharedLibrary*, *mozbuild.frontend.data.ExternalLibrary*

Context derived container for shared libraries built by an external build system.

class mozbuild.frontend.data.**ExternalStaticLibrary**(*context*, *basename*, *real_name=None*, *is_sdk=False*, *link_into=None*, *no_expand_lib=False*)
Bases: *mozbuild.frontend.data.StaticLibrary*, *mozbuild.frontend.data.ExternalLibrary*

Context derived container for static libraries built by an external build system.

class mozbuild.frontend.data.**FinalTargetFiles**(*sandbox*, *files*)
Bases: *mozbuild.frontend.data.ContextDerived*

Sandbox container object for FINAL_TARGET_FILES, which is a HierarchicalStringList.

We need an object derived from ContextDerived for use in the backend, so this object fills that role. It just has a reference to the underlying HierarchicalStringList, which is created when parsing FINAL_TARGET_FILES.

files

class mozbuild.frontend.data.**FinalTargetPreprocessedFiles**(*sandbox*, *files*)
Bases: *mozbuild.frontend.data.ContextDerived*

Sandbox container object for FINAL_TARGET_PP_FILES, which is a HierarchicalStringList.

We need an object derived from ContextDerived for use in the backend, so this object fills that role. It just has a reference to the underlying HierarchicalStringList, which is created when parsing FINAL_TARGET_PP_FILES.

files

class mozbuild.frontend.data.**GeneratedEventWebIDLFile**(*context*, *path*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes an individual .webidl source file.

basename

class mozbuild.frontend.data.**GeneratedFile**(*context*, *script*, *method*, *outputs*, *inputs*, *flags=()*)
Bases: *mozbuild.frontend.data.ContextDerived*

Represents a generated file.

flags

inputs

method

outputs

script

class mozbuild.frontend.data.**GeneratedSources**(*context*, *files*, *canonical_suffix*)
Bases: *mozbuild.frontend.data.BaseSources*

Represents generated files to be compiled during the build.

class mozbuild.frontend.data.**GeneratedWebIDLFile**(*context*, *path*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes an individual .webidl source file that is generated from build rules.

basename

class mozbuild.frontend.data.**HostDefines**(*context*, *defines*)
Bases: *mozbuild.frontend.data.BaseDefines*

class mozbuild.frontend.data.**HostLibrary**(*context*, *basename*)
Bases: *mozbuild.frontend.data.HostMixin*, *mozbuild.frontend.data.BaseLibrary*

Context derived container object for a host library

KIND = u'host'

class mozbuild.frontend.data.**HostMixin**
Bases: object

defines

class mozbuild.frontend.data.**HostProgram**(*context*, *program*, *is_unit_test=False*)
Bases: *mozbuild.frontend.data.HostMixin*, *mozbuild.frontend.data.BaseProgram*

Context derived container object for HOST_PROGRAM

KIND = u'host'

SUFFIX_VAR = u'HOST_BIN_SUFFIX'

class mozbuild.frontend.data.**HostSimpleProgram**(*context*, *program*, *is_unit_test=False*)
Bases: *mozbuild.frontend.data.HostMixin*, *mozbuild.frontend.data.BaseProgram*

Context derived container object for each program in HOST_SIMPLE_PROGRAMS

KIND = u'host'

SUFFIX_VAR = u'HOST_BIN_SUFFIX'

class mozbuild.frontend.data.**HostSources**(*context*, *files*, *canonical_suffix*)
Bases: *mozbuild.frontend.data.HostMixin*, *mozbuild.frontend.data.BaseSources*

Represents files to be compiled for the host during the build.

class mozbuild.frontend.data.**IPDLFile**(*context*, *path*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes an individual .ipdl source file.

basename

class mozbuild.frontend.data.**InstallationTarget**(*context*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes the rules that affect where files get installed to.

enabled

is_custom()
Returns whether or not the target is not derived from the default given xpiname and subdir.

subdir

target

xpiname

class mozbuild.frontend.data.**JARManifest**(*context*, *path*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes an individual JAR manifest file and how to process it.

This class isn't very useful for optimizing backends yet because we don't capture defines. We can't capture defines safely until all of them are defined in moz.build and not Makefile.in files.

path

class mozbuild.frontend.data.**JavaJarData**(*name*, *sources=[]*, *generated_sources=[]*, *extra_jars=[]*, *javac_flags=[]*)
Bases: object

Represents a Java JAR file.

A Java JAR has the following members:

- sources - strictly ordered list of input java sources
- generated_sources - strictly ordered list of generated input java sources
- extra_jars - list of JAR file dependencies to include on the javac compiler classpath
- javac_flags - list containing extra flags passed to the javac compiler

extra_jars

generated_sources

javac_flags

name

sources

class mozbuild.frontend.data.**Library**(*context*, *basename*, *real_name=None*, *is_sdk=False*)
Bases: *mozbuild.frontend.data.BaseLibrary*

Context derived container object for a library

KIND = u'target'

is_sdk

class mozbuild.frontend.data.**Linkable**(*context*)
Bases: *mozbuild.frontend.data.ContextDerived*

Generic context derived container object for programs and libraries

lib_defines

link_library(*obj*)

link_system_library(*lib*)

linked_libraries

linked_system_libs

exception mozbuild.frontend.data.**LinkageWrongKindError**
Bases: exceptions.Exception

Error thrown when trying to link objects of the wrong kind

class mozbuild.frontend.data.**LocalInclude**(*context*, *path*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes an individual local include path.

path

class mozbuild.frontend.data.**ObjdirFiles**(*sandbox*, *files*)
Bases: *mozbuild.frontend.data.ContextDerived*

Sandbox container object for OBJDIR_FILES, which is a HierarchicalStringList.

files

install_target

class mozbuild.frontend.data.**ObjdirPreprocessedFiles**(*sandbox*, *files*)
Bases: *mozbuild.frontend.data.ContextDerived*

Sandbox container object for OBJDIR_PP_FILES, which is a HierarchicalStringList.

files

install_target

class mozbuild.frontend.data.**PerSourceFlag**(*context*, *file_name*, *flags*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes compiler flags specified for individual source files.

file_name

flags

class mozbuild.frontend.data.**PreprocessedTestWebIDLFile**(*context*, *path*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes an individual test-only .webidl source file that requires preprocessing.

basename

class mozbuild.frontend.data.**PreprocessedWebIDLFile**(*context*, *path*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes an individual .webidl source file that requires preprocessing.

basename

class mozbuild.frontend.data.**Program**(*context*, *program*, *is_unit_test=False*)
Bases: *mozbuild.frontend.data.BaseProgram*

Context derived container object for PROGRAM

KIND = u'target'

SUFFIX_VAR = u'BIN_SUFFIX'

class mozbuild.frontend.data.**RustRlibLibrary**(*context*, *basename*, *crate_name*, *rlib_filename*, *link_into*)
Bases: *mozbuild.frontend.data.Library*

Context derived container object for a Rust rlib

class mozbuild.frontend.data.**SdkFiles**(*sandbox*, *files*)
Bases: *mozbuild.frontend.data.FinalTargetFiles*

Sandbox container object for SDK_FILES, which is a HierarchicalStringList.

We need an object derived from ContextDerived for use in the backend, so this object fills that role. It just has a reference to the underlying HierarchicalStringList, which is created when parsing SDK_FILES.

install_target

class mozbuild.frontend.data.**SharedLibrary**(*context*, *basename*, *real_name=None*, *is_sdk=False*, *soname=None*, *variant=None*, *symbols_file=False*)
Bases: *mozbuild.frontend.data.Library*

Context derived container object for a shared library

COMPONENT = 2

DICT_ATTRS = set([u'install_target', u'soname', u'basename', u'relobjdir', u'lib_name', u'import_name'])

FRAMEWORK = 1

MAX_VARIANT = 3

soname

symbols_file

variant

class mozbuild.frontend.data.**SimpleProgram**(*context*, *program*, *is_unit_test=False*)
Bases: *mozbuild.frontend.data.BaseProgram*

Context derived container object for each program in SIMPLE_PROGRAMS

KIND = u'target'

SUFFIX_VAR = u'BIN_SUFFIX'

class mozbuild.frontend.data.**Sources**(*context*, *files*, *canonical_suffix*)
Bases: *mozbuild.frontend.data.BaseSources*

Represents files to be compiled during the build.

class mozbuild.frontend.data.**StaticLibrary**(*context*, *basename*, *real_name=None*, *is_sdk=False*, *link_into=None*, *no_expand_lib=False*)
Bases: *mozbuild.frontend.data.Library*

Context derived container object for a static library

link_into

no_expand_lib

class mozbuild.frontend.data.**TestHarnessFiles**(*sandbox*, *files*)
Bases: *mozbuild.frontend.data.FinalTargetFiles*

Sandbox container object for TEST_HARNESS_FILES, which is a HierarchicalStringList.

install_target

class mozbuild.frontend.data.**TestManifest**(*context*, *path*, *manifest*, *flavor=None*, *install_prefix=None*, *relpath=None*, *dupe_manifest=False*)
Bases: *mozbuild.frontend.data.ContextDerived*

Represents a manifest file containing information about tests.

default_support_files

deferred_installs

directory

dupe_manifest

external_installs

flavor

install_prefix

installs

manifest

manifest_obj_relpath

manifest_relpath

path

pattern_installs

tests

class mozbuild.frontend.data.**TestWebIDLFile**(*context*, *path*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes an individual test-only .webidl source file.

basename

class mozbuild.frontend.data.**TreeMetadata**
Bases: object

Base class for all data being captured.

to_dict()

class mozbuild.frontend.data.**UnifiedSources**(*context*, *files*, *canonical_suffix*, *files_per_unified_file=16*)
Bases: *mozbuild.frontend.data.BaseSources*

Represents files to be compiled in a unified fashion during the build.

have_unified_mapping

unified_source_mapping

class mozbuild.frontend.data.**VariablePassthru**(*context*)
Bases: *mozbuild.frontend.data.ContextDerived*

A dict of variables to pass through to backend.mk unaltered.

The purpose of this object is to facilitate rapid transitioning of variables from Makefile.in to moz.build. In the ideal world, this class does not exist and every variable has a richer class representing it. As long as we rely on this class, we lose the ability to have flexibility in our build backends since we will continue to be tied to our rules.mk.

variables

class mozbuild.frontend.data.**WebIDLFile**(*context*, *path*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes an individual .webidl source file.

basename

class mozbuild.frontend.data.**XPIDLFile**(*context*, *source*, *module*, *add_to_manifest*)
Bases: *mozbuild.frontend.data.ContextDerived*

Describes an XPIDL file to be compiled.

add_to_manifest

basename

module

source_path

mozbuild.frontend.emitter module

class mozbuild.frontend.emitter.**TreeMetadataEmitter**(*config*)
Bases: *mach.mixin.logging.LoggingMixin*

Converts the executed mozbuild files into data structures.

This is a bridge between reader.py and data.py. It takes what was read by reader.BuildReader and converts it into the classes defined in the data module.

LIBRARY_NAME_VAR = {u'host': u'HOST_LIBRARY_NAME', u'target': u'LIBRARY_NAME'}

emit(*output*)
Convert the BuildReader output into data structures.

The return value from BuildReader.read_topsrcdir() (a generator) is typically fed into this function.

emit_from_context(*context*)
Convert a Context to tree metadata objects.

This is a generator of mozbuild.frontend.data.ContextDerived instances.

summary()

mozbuild.frontend.gyp_reader module

class mozbuild.frontend.gyp_reader.**GypContext**(*config*, *relobjdir*)
Bases: *mozbuild.frontend.context.TemplateContext*

Specialized Context for use with data extracted from Gyp.

config is the ConfigEnvironment for this context. relobjdir is the object directory that will be used for this context, relative to the topobjdir defined in the ConfigEnvironment.

mozbuild.frontend.gyp_reader.**encode**(*value*)

mozbuild.frontend.gyp_reader.**read_from_gyp**(*config*, *path*, *output*, *vars*, *non_unified_sources=set([])*)
Read a gyp configuration and emits GypContexts for the backend to process.

config is a ConfigEnvironment, path is the path to a root gyp configuration file, output is the base path under which the objdir for the various gyp dependencies will be, and vars a dict of variables to pass to the gyp processor.

mozbuild.frontend.mach_commands module

mozbuild.frontend.reader module

Read build frontend files into data structures.

In terms of code architecture, the main interface is BuildReader. BuildReader starts with a root mozbuild file. It creates a new execution environment for this file, which is represented by the Sandbox class. The Sandbox class is used to fill a Context, representing the output of an individual mozbuild file. The

The BuildReader contains basic logic for traversing a tree of mozbuild files. It does this by examining specific variables populated during execution.

class mozbuild.frontend.reader.**BuildReader**(*config*, *finder=<mozpack.files.FileFinder object>*)
Bases: object

Read a tree of mozbuild files into data structures.

This is where the build system starts. You give it a tree configuration (the output of configuration) and it executes the moz.build files and collects the data they define.

The reader can optionally call a callable after each sandbox is evaluated but before its evaluated content is processed. This gives callers the opportunity to modify contexts before side-effects occur from their content. This callback receives the Context containing the result of each sandbox evaluation. Its return value is ignored.

all_mozbuild_paths()
Iterator over all available moz.build files.

This method has little to do with the reader. It should arguably belong elsewhere.

files_info(*paths*)
Obtain aggregate data from Files for a set of files.

Given a set of input paths, determine which moz.build files may define metadata for them, evaluate those moz.build files, and apply file metadata rules defined within to determine metadata values for each file requested.

Essentially, for each input path:

1. Determine the set of moz.build files relevant to that file by looking for moz.build files in ancestor directories.
2. Evaluate moz.build files starting with the most distant.
3. Iterate over Files sub-contexts.
4. If the file pattern matches the file we're seeking info on, apply attribute updates.
5. Return the most recent value of attributes.

find_sphinx_variables()
This function finds all assignments of Sphinx documentation variables.

This is a generator of tuples of (moz.build path, var, key, value). For variables that assign to keys in objects, key will be defined.

With a little work, this function could be made more generic. But if we end up writing a lot of ast code, it might be best to import a high-level AST manipulation library into the tree.

read_mozbuild(*path*, *config*, *descend=True*, *metadata={}*)
Read and process a mozbuild file, descending into children.

This starts with a single mozbuild file, executes it, and descends into other referenced files per our traversal logic.

The traversal logic is to iterate over the *DIRS variables, treating each element as a relative directory path. For each encountered directory, we will open the moz.build file located in that directory in a new Sandbox and process it.

If descend is True (the default), we will descend into child directories and files per variable values.

Arbitrary metadata in the form of a dict can be passed into this function. This feature is intended to facilitate the build reader injecting state and annotations into moz.build files that is independent of the sandbox's execution context.

Traversal is performed depth first (for no particular reason).

read_relevant_mozbuilds(*paths*)
Read and process moz.build files relevant for a set of paths.

For an iterable of relative-to-root filesystem paths `paths`, find all moz.build files that may apply to them based on filesystem hierarchy and read those moz.build files.

The return value is a 2-tuple. The first item is a dict mapping each input filesystem path to a list of Context instances that are relevant to that path. The second item is a list of all Context instances. Each Context instance is in both data structures.

read_topsrcdir()
Read the tree of linked moz.build files.

This starts with the tree's top-most moz.build file and descends into all linked moz.build files until all relevant files have been evaluated.

This is a generator of Context instances. As each moz.build file is read, a new Context is created and emitted.

summary()

test_defaults_for_path(*ctxs*)

exception mozbuild.frontend.reader.**BuildReaderError**(*file_stack*, *trace*, *sandbox_exec_error=None*, *sandbox_load_error=None*, *validation_error=None*, *other_error=None*, *sandbox_called_error=None*)

Bases: exceptions.Exception

Represents errors encountered during BuildReader execution.

The main purpose of this class is to facilitate user-actionable error messages. Execution errors should say:

- Why they failed
- Where they failed
- What can be done to prevent the error

A lot of the code in this class should arguably be inside sandbox.py. However, extraction is somewhat difficult given the additions MozbuildSandbox has over Sandbox (e.g. the concept of included files - which affect error messages, of course).

actual_file

main_file

sandbox_error

class mozbuild.frontend.reader.**EmptyConfig**(*topsrcdir*)

Bases: object

A config object that is empty.

This config object is suitable for using with a BuildReader on a vanilla checkout, without any existing configuration. The config is simply bootstrapped from a top source directory path.

class PopulateOnGetDict(*default_factory*, **args*, ***kwargs*)

Bases: *mozbuild.util.ReadOnlyDefaultDict*

A variation on ReadOnlyDefaultDict that populates during .get().

This variation is needed because CONFIG uses .get() to access members. Without it, None (instead of our EmptyValue types) would be returned.

get(*key*, *default=None*)

class mozbuild.frontend.reader.**MozbuildSandbox**(*context*, *metadata={}*, *finder=<mozpack.files.FileFinder object>*)

Bases: *mozbuild.frontend.sandbox.Sandbox*

Implementation of a Sandbox tailored for mozbuild files.

We expose a few useful functions and expose the set of variables defining Mozilla's build system.

context is a Context instance.

metadata is a dict of metadata that can be used during the sandbox evaluation.

add_android_eclipse_project_helper(*name*)

Add an Android Eclipse project target.

exec_file(*path*)

Override exec_file to normalize paths and restrict file loading.

Paths will be rejected if they do not fall under topsrcdir or one of the external roots.

recompute_exports()
Recompute the variables to export to subdirectories with the current values in the subdirectory.

exception mozbuild.frontend.reader.**SandboxCalledError**(*file_stack*, *message*)
Bases: *mozbuild.frontend.sandbox.SandboxError*

Represents an error resulting from calling the error() function.

exception mozbuild.frontend.reader.**SandboxValidationError**(*message*, *context*)
Bases: exceptions.Exception

Represents an error encountered when validating sandbox results.

class mozbuild.frontend.reader.**TemplateFunction**(*func*, *sandbox*)
Bases: object

class RewriteName(*sandbox*, *global_name*)
Bases: ast.NodeTransformer

AST Node Transformer to rewrite variable accesses to go through a dict.

visit_Name(*node*)

visit_Str(*node*)

TemplateFunction.**exec_in_sandbox**(*sandbox*, **args*, ***kwargs*)
Executes the template function in the given sandbox.

mozbuild.frontend.reader.**is_read_allowed**(*path*, *config*)
Whether we are allowed to load a mozbuild file at the specified path.

This is used as cheap security to ensure the build is isolated to known source directories.

We are allowed to read from the main source directory and any defined external source directories. The latter is to allow 3rd party applications to hook into our build system.

mozbuild.frontend.reader.**log**(*logger*, *level*, *action*, *params*, *formatter*)

mozbuild.frontend.sandbox module

Python sandbox implementation for build files.

This module contains classes for Python sandboxes that execute in a highly-controlled environment.

The main class is *Sandbox*. This provides an execution environment for Python code and is used to fill a Context instance for the takeaway information from the execution.

Code in this module takes a different approach to exception handling compared to what you'd see elsewhere in Python. Arguments to built-in exceptions like KeyError are machine parseable. This machine-friendly data is used to present user-friendly error messages in the case of errors.

class mozbuild.frontend.sandbox.**Sandbox**(*context*, *builtins=None*, *finder=<mozpack.files.FileFinder object>*)
Bases: dict

Represents a sandbox for executing Python code.

This class provides a sandbox for execution of a single mozbuild frontend file. The results of that execution is stored in the Context instance given as the context argument.

Sandbox is effectively a glorified wrapper around compile() + exec(). You point it at some Python code and it executes it. The main difference from executing Python code like normal is that the executed code is very limited in what it can do: the sandbox only exposes a very limited set of Python functionality. Only specific

types and functions are available. This prevents executed code from doing things like import modules, open files, etc.

Sandbox instances act as global namespace for the sandboxed execution itself. They shall not be used to access the results of the execution. Those results are available in the given Context instance after execution.

The Sandbox itself is responsible for enforcing rules such as forbidding reassignment of variables.

Implementation note: Sandbox derives from dict because exec() insists that what it is given for namespaces is a dict.

BUILTINS = {u'int': <type 'int'>, u'False': False, u'None': None, u'True': True, u'sorted': <function alphabetical_sorte

exec_file(*path*)
Execute code at a path in the sandbox.

The path must be absolute.

exec_function(*func*, *args=()*, *kwargs={}*, *path=u''*, *becomes_current_path=True*)
Execute function with the given arguments in the sandbox.

exec_source(*source*, *path=u''*)
Execute Python code within a string.

The passed string should contain Python code to be executed. The string will be compiled and executed.

You should almost always go through exec_file() because exec_source() does not perform extra path normalization. This can cause relative paths to behave weirdly.

get(*key*, *default=None*)

pop_subcontext(*context*)
Pop a SubContext off the execution stack.

SubContexts must be pushed and popped in opposite order. This is validated as part of the function call to ensure proper consumer API use.

push_subcontext(*context*)
Push a SubContext onto the execution stack.

When called, the active context will be set to the specified context, meaning all variable accesses will go through it. We also record this SubContext as having been executed as part of this sandbox.

exception mozbuild.frontend.sandbox.**SandboxError**(*file_stack*)
Bases: exceptions.Exception

exception mozbuild.frontend.sandbox.**SandboxExecutionError**(*file_stack*, *exc_type*, *exc_value*, *trace*)
Bases: *mozbuild.frontend.sandbox.SandboxError*

Represents errors encountered during execution of a Sandbox.

This is a simple container exception. It's purpose is to capture state so something else can report on it.

exception mozbuild.frontend.sandbox.**SandboxLoadError**(*file_stack*, *trace*, *illegal_path=None*, *read_error=None*)
Bases: *mozbuild.frontend.sandbox.SandboxError*

Represents errors encountered when loading a file for execution.

This exception represents errors in a Sandbox that occurred as part of loading a file. The error could have occurred in the course of executing a file. If so, the file_stack will be non-empty and the file that caused the load will be on top of the stack.

mozbuild.frontend.sandbox.**alphabetical_sorted**(*iterable*, *cmp=None*, *key=<function <lambda>>*, *reverse=False*)
sorted() replacement for the sandbox, ordering alphabetically by default.

Module contents

mozbuild.test package

Subpackages

mozbuild.test.backend package

Submodules

mozbuild.test.backend.common module

class mozbuild.test.backend.common.**BackendTester**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

tearDown()

mozbuild.test.backend.test_android_eclipse module

class mozbuild.test.backend.test_android_eclipse.**TestAndroidEclipseBackend**(**args*, ***kwargs*)
Bases: *mozbuild.test.backend.common.BackendTester*

assertExists(**args*)

assertInManifest(*project_name*, **args*)

assertNotExists(**args*)

assertNotInManifest(*project_name*, **args*)

test_classpathentries()
Ensure we produce reasonable classpathentries.

test_extra_jars()
Ensure we add class path entries to extra jars iff asked to.

test_included_projects()
Ensure we include another project correctly.

test_library_manifest()
Ensure we generate manifest for library projects.

test_library_project_files()
Ensure we generate reasonable files for library projects.

test_library_project_setting()
Ensure we declare a library project correctly.

test_main_project_files()
Ensure we generate reasonable files for main (non-library) projects.

test_manifest_assets()
Ensure we symlink assets/ iff asked to.

test_manifest_classpathentries()
Ensure we symlink classpathentries correctly.

test_manifest_main_manifest()
Ensure we symlink manifest if asked to for main projects.

test_manifest_res()
Ensure we symlink res/ iff asked to.

test_referenced_projects()
Ensure we reference another project correctly.

mozbuild.test.backend.test_build module

mozbuild.test.backend.test_configenvironment module

class mozbuild.test.backend.test_configenvironment.**ConfigEnvironment**(**args*, ***kwargs*)
Bases: *mozbuild.backend.configenvironment.ConfigEnvironment*

class mozbuild.test.backend.test_configenvironment.**TestEnvironment**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_auto_substs()
Test the automatically set values of ACDEFINES, ALLSUBSTS and ALLEMPTYSUBSTS.

mozbuild.test.backend.test_recursivemake module

class mozbuild.test.backend.test_recursivemake.**TestRecursiveMakeBackend**(*methodName='runTest'*)
Bases: *mozbuild.test.backend.common.BackendTester*

test_android_eclipse()

test_backend_mk()
Ensure backend.mk file is written out properly.

test_basic()
Ensure the RecursiveMakeBackend works without error.

test_binary_components()
Ensure binary components are correctly handled.

test_branding_files()
Ensure BRANDING_FILES is handled properly.

test_config()
Test that CONFIGURE_SUBST_FILES are properly handled.

test_defines()
Test that DEFINES are written to backend.mk correctly.

test_exports()
Ensure EXPORTS is handled properly.

test_exports_generated()
Ensure EXPORTS that are listed in GENERATED_FILES are handled properly.

test_final_target()
Test that FINAL_TARGET is written to backend.mk correctly.

test_final_target_pp_files()
Test that FINAL_TARGET_PP_FILES is written to backend.mk correctly.

test_generated_files()
Ensure GENERATED_FILES is handled properly.

test_generated_includes()
Test that GENERATED_INCLUDES are written to backend.mk correctly.

test_host_defines()
Test that HOST_DEFINES are written to backend.mk correctly.

test_install_manifests_package_tests()
Ensure test suites honor package_tests=False.

test_install_manifests_written()

test_install_substitute_config_files()
Ensure we recurse into the dirs that install substituted config files.

test_ipdl_sources()
Test that IPDL_SOURCES are written to ipdlsrcs.mk correctly.

test_jar_manifests()

test_local_includes()
Test that LOCAL_INCLUDES are written to backend.mk correctly.

test_makefile_conversion()
Ensure Makefile.in is converted properly.

test_missing_makefile_in()
Ensure missing Makefile.in results in Makefile creation.

test_mtime_no_change()
Ensure mtime is not updated if file content does not change.

test_old_install_manifest_deleted()

test_output_files()
Ensure proper files are generated.

test_resources()
Ensure RESOURCE_FILES is handled properly.

test_sdk_files()
Ensure SDK_FILES is handled properly.

test_sources()
Ensure SOURCES and HOST_SOURCES are handled properly.

test_substitute_config_files()
Ensure substituted config files are produced.

test_test_manifest_deffered_installs_written()
Shared support files are written to their own data file by the backend.

test_test_manifest_pattern_matches_recorded()
Pattern matches in test manifests' support-files should be recorded.

test_test_manifests_duplicate_support_files()
Ensure duplicate support-files in test manifests work.

test_test_manifests_files_written()
Ensure test manifests get turned into files.

test_variable_passthru()
Ensure variable passthru is written out correctly.

test_xpidl_generation()
Ensure xpidl files and directories are written out.

class mozbuild.test.backend.test_recursivemake.**TestRecursiveMakeTraversal**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_traversal()

test_traversal_2()

test_traversal_filter()

mozbuild.test.backend.test_visualstudio module

class mozbuild.test.backend.test_visualstudio.**TestVisualStudioBackend**(*methodName='runTest'*)
Bases: *mozbuild.test.backend.common.BackendTester*

test_basic(**args*, ***kwargs*)
Ensure we can consume our stub project.

Module contents

mozbuild.test.compilation package

Submodules

mozbuild.test.compilation.test_warnings module

class mozbuild.test.compilation.test_warnings.**TestCompilerWarning**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_comparison()

test_equivalence()

class mozbuild.test.compilation.test_warnings.**TestWarningsDatabase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_basic()

test_hashing()
Ensure that hashing files on insert works.

test_pruning()
Ensure old warnings are removed from database appropriately.

class mozbuild.test.compilation.test_warnings.**TestWarningsParsing**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_clang_parsing()

test_msvc_parsing()

mozbuild.test.compilation.test_warnings.**get_warning**()

Module contents

mozbuild.test.controller package

Submodules

mozbuild.test.controller.test_ccachestats module

class mozbuild.test.controller.test_ccachestats.**TestCcacheStats**(*methodName='runTest'*)

Bases: unittest.case.TestCase

STAT0 = u'\n cache directory /home/tlin/.ccache\n cache hit (direct) 0\n cache hit (preprocessed) 0\n cache miss 0\n files in

STAT1 = u'\n cache directory /home/tlin/.ccache\n cache hit (direct) 100\n cache hit (preprocessed) 200\n cache miss 2500

STAT2 = u'\n cache directory /home/tlin/.ccache\n cache hit (direct) 1900\n cache hit (preprocessed) 300\n cache miss 260

STAT3 = u'\n cache directory /Users/tlin/.ccache\n primary config /Users/tlin/.ccache/ccache.conf\n secondary config (read

STAT4 = u'\n cache directory /Users/tlin/.ccache\n primary config /Users/tlin/.ccache/ccache.conf\n secondary config (read

STAT5 = u'\n cache directory /Users/tlin/.ccache\n primary config /Users/tlin/.ccache/ccache.conf\n secondary config (read

STAT_GARBAGE = u'A garbage line which should be failed to parse'

test_cache_size_shrinking()

test_hit_rate_of_diff_stats()

test_parse_garbage_stats_message()

test_parse_zero_stats_message()

test_stats_contains_data()

test_stats_version32()

mozbuild.test.controller.test_clobber module

class mozbuild.test.controller.test_clobber.**TestClobberer**(*methodName='runTest'*)

Bases: unittest.case.TestCase

get_tempdir()

get_topsrcdir()

setUp()

tearDown()

test_cwd_is_topobjdir()
If cwd is topobjdir, we can still clobber.

test_cwd_under_topobjdir()
If cwd is under topobjdir, we can't clobber.

test_mozconfig_opt_in()
Auto clobber iff AUTOCLOBBER is in the environment.

test_no_objdir()
If topobjdir does not exist, no clobber is needed.

test_objdir_clobber_newer()
If CLOBBER in topobjdir is newer, do nothing.

test_objdir_clobber_older()
If CLOBBER in topobjdir is older, we clobber.

test_objdir_is_srcdir()
If topobjdir is the topsrcdir, refuse to clobber.

test_objdir_no_clobber_file()
If CLOBBER does not exist in topobjdir, treat as empty.

Module contents

mozbuild.test.frontend package

Submodules

mozbuild.test.frontend.test_context module

class mozbuild.test.frontend.test_context.**TestContext**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_context_dirs()

test_context_paths()

test_defaults()

test_type_check()

test_update()

class mozbuild.test.frontend.test_context.**TestFiles**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_aggregate_empty()

test_multiple_bug_components()

test_no_recommended_bug_component()
If there is no clear count winner, we don't recommend a bug component.

test_single_bug_component()

class mozbuild.test.frontend.test_context.**TestPaths**(*methodName='runTest'*)
Bases: unittest.case.TestCase

classmethod setUpClass()

test_absolute_path()

test_objdir_path()

test_path()

test_path_typed_hierarchy_list()

test_path_typed_list()

test_path_with_mixed_contexts()

test_source_path()

class mozbuild.test.frontend.test_context.**TestSymbols**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_documentation_formatting()

class mozbuild.test.frontend.test_context.**TestTypedRecord**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_coercion()

test_fields()

mozbuild.test.frontend.test_emitter module

class mozbuild.test.frontend.test_emitter.**TestEmitterBasic**(*methodName='runTest'*)

Bases: unittest.case.TestCase

read_topsrcdir(*reader*, *filter_common=True*)

reader(*name*, *enable_tests=False*, *extra_substs=None*)

setUp()

tearDown()

test_android_res_dirs()
Test that ANDROID_RES_DIRS works properly.

test_binary_components()
Test that IS_COMPONENT/NO_COMPONENTS_MANIFEST work properly.

test_branding_files()

test_config_file_substitution()

test_defines()

test_dirs_traversal_simple()

test_empty_test_manifest_rejected()
A test manifest without any entries is rejected.

test_exports()

test_exports_generated()

test_exports_missing()
Missing files in EXPORTS is an error.

test_exports_missing_generated()
An objdir file in EXPORTS that is not in GENERATED_FILES is an error.

test_final_target_pp_files()
Test that FINAL_TARGET_PP_FILES works properly.

test_final_target_pp_files_non_srcdir()
Test that non-srcdir paths in FINAL_TARGET_PP_FILES throws errors.

test_generated_files()

test_generated_files_absolute_script()

test_generated_files_method_names()

test_generated_files_no_inputs()

test_generated_files_no_python_script()

test_generated_files_no_script()

test_generated_includes()
Test that GENERATED_INCLUDES is emitted correctly.

test_generated_sources()
Test that GENERATED_SOURCES works properly.

test_host_defines()

test_host_sources()
Test that HOST_SOURCES works properly.

test_install_shared_lib()
Test that we can install a shared library with TEST_HARNESS_FILES

test_ipdl_sources()

test_jar_manifests()

test_jar_manifests_multiple_files()

test_library_defines()
Test that LIBRARY_DEFINES is propagated properly.

test_local_includes()
Test that LOCAL_INCLUDES is emitted correctly.

test_missing_final_target_pp_files()
Test that FINAL_TARGET_PP_FILES with missing files throws errors.

test_missing_local_includes()
LOCAL_INCLUDES containing non-existent directories should be rejected.

test_program()

test_python_unit_test_missing()
Missing files in PYTHON_UNIT_TESTS should raise.

test_sdk_files()

test_sources()
Test that SOURCES works properly.

test_test_harness_files()

test_test_harness_files_root()

test_test_manifest_absolute_support_files()
Support files starting with '/' are placed relative to the install root

test_test_manifest_deffered_install_missing()
A non-existent shared support file reference produces an error.

test_test_manifest_dupe_support_files()
A test manifest with dupe support-files in a single test is not supported.

test_test_manifest_includes()
Ensure that manifest objects from the emitter list a correct manifest.

test_test_manifest_install_includes()
Ensure that any [include:foo.ini] are copied to the objdir.

test_test_manifest_install_to_subdir()

test_test_manifest_just_support_files()
A test manifest with no tests but support-files is not supported.

test_test_manifest_keys_extracted()
Ensure all metadata from test manifests is extracted.

test_test_manifest_missing_manifest()
A missing manifest file should result in an error.

test_test_manifest_missing_test_error()
Missing test files should result in error.

test_test_manifest_missing_test_error_unfiltered()
Missing test files should result in error, even when the test list is not filtered.

test_test_manifest_parent_support_files_dir()
support-files referencing a file in a parent directory works.

test_test_manifest_shared_support_files()
Support files starting with '!' are given separate treatment, so their installation can be resolved when running tests.

test_test_manifest_unmatched_generated()

test_traversal_all_vars()

test_traversal_all_vars_enable_tests()

test_unified_sources()
Test that UNIFIED_SOURCES works properly.

test_unified_sources_non_unified()
Test that UNIFIED_SOURCES with FILES_PER_UNIFIED_FILE=1 works properly.

test_use_yasm()

test_variable_passthru()

test_xpidl_module_no_sources()
XPIDL_MODULE without XPIDL_SOURCES should be rejected.

mozbuild.test.frontend.test_namespaces module

class mozbuild.test.frontend.test_namespaces.**Fuga**(*value*)
Bases: object

class mozbuild.test.frontend.test_namespaces.**Piyo**(*context*, *value*)
Bases: *mozbuild.frontend.context.ContextDerivedValue*

lower()

class mozbuild.test.frontend.test_namespaces.**TestContext**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_allowed_set()

test_coercion()

test_context_derived_coercion()

test_context_derived_typed_list()

test_context_derived_typed_list_with_items()

test_key_checking()

test_key_rejection()

test_value_checking()

mozbuild.test.frontend.test_reader module

class mozbuild.test.frontend.test_reader.**TestBuildReader**(*methodName='runTest'*)
Bases: unittest.case.TestCase

config(*name*, ***kwargs*)

file_path(*name*, **args*)

reader(*name*, *enable_tests=False*, *error_is_fatal=True*, ***kwargs*)

setUp()

tearDown()

test_dirs_traversal_all_variables()

test_dirs_traversal_no_descend()

test_dirs_traversal_simple()

test_error_bad_dir()

test_error_basic()

test_error_empty_list()

test_error_error_func()

test_error_error_func_ok()

test_error_illegal_path()

test_error_included_from()

test_error_missing_include_path()

test_error_read_unknown_global()

test_error_repeated_dir()

test_error_script_error()

test_error_syntax_error()

test_error_write_bad_value()

test_error_write_unknown_global()

test_file_test_deps()

test_file_test_deps_default()

test_file_test_deps_tags()

test_files_bad_bug_component()

test_files_bug_component_different_matchers()

test_files_bug_component_final()

test_files_bug_component_simple()

test_files_bug_component_static()

test_find_relevant_mozbuilds()

test_inheriting_variables()

test_invalid_flavor()

test_outside_topsrcdir()

test_read_relevant_mozbuilds()

test_relative_dirs()

test_repeated_dirs_ignored()

mozbuild.test.frontend.test_sandbox module

class mozbuild.test.frontend.test_sandbox.**TestMozbuildSandbox**(*methodName='runTest'*)

Bases: unittest.case.TestCase

sandbox(*data_path=None*, *metadata={}*)

test_config_access()

test_default_state()

test_error()

test_exec_source_reassign_exported()

test_function_args()

test_include_basic()

test_include_error_stack()

test_include_missing()

test_include_outside_topsrcdir()

test_include_relative_from_child_dir()

test_include_topsrcdir_relative()

test_invalid_exports_set_base()

test_invalid_utf8_substs()

Ensure invalid UTF-8 in substs is converted with an error.

test_path_calculation()

test_special_variables()

test_substitute_config_files()

test_symbol_presence()

test_templates()

class mozbuild.test.frontend.test_sandbox.**TestSandbox**(*methodName='runTest'*)

Bases: unittest.case.TestCase

sandbox()

test_exec_compile_error()

test_exec_import_denied()

test_exec_source_illegal_key_set()

test_exec_source_multiple()

test_exec_source_reassign()

test_exec_source_reassign_builtin()

test_exec_source_success()

class mozbuild.test.frontend.test_sandbox.**TestedSandbox**(*context*, *metadata={}*, *finder=<mozpack.files.FileFinder object>*)

Bases: *mozbuild.frontend.reader.MozbuildSandbox*

Version of MozbuildSandbox with a little more convenience for testing.

It automatically normalizes paths given to exec_file and exec_source. This helps simplify the test code.

exec_file(*path*)

exec_source(*source*, *path=u''*)

normalize_path(*path*)

source_path(*path*)

Module contents

Submodules

mozbuild.test.common module

class mozbuild.test.common.**MockConfig**(*topsrcdir=u'/path/to/topsrcdir'*, *extra_substs={}*, *error_is_fatal=True*)

Bases: object

mozbuild.test.test_android_version_code module

class mozbuild.test.test_android_version_code.**TestAndroidVersionCode**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_android_version_code_v0()

test_android_version_code_v0_relative_v1()

Verify that the first v1 code is greater than the equivalent v0 code.

test_android_version_code_v1()

test_android_version_code_v1_overflow()

Verify that it is an error to ask for v1 codes that actually does overflow.

test_android_version_code_v1_running_low()

Verify there is an informative message if one asks for v1 codes that are close to overflow.

test_android_version_code_v1_underflow()

Verify that it is an error to ask for v1 codes predating the cutoff.

mozbuild.test.test_base module

mozbuild.test.test_containers module

class mozbuild.test.test_containers.**TestKeyedDefaultDict**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_defaults()

test_simple()

class mozbuild.test.test_containers.**TestList**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test_add_list()

test_add_string()

test_none()
As a special exception, we allow None to be treated as an empty list.

class mozbuild.test.test_containers.**TestOrderedDefaultDict**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_defaults()

test_simple()

class mozbuild.test.test_containers.**TestReadOnlyDefaultDict**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_assignment()

test_defaults()

test_simple()

class mozbuild.test.test_containers.**TestReadOnlyDict**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_basic()

test_del()

test_update()

class mozbuild.test.test_containers.**TestReadOnlyKeyedDefaultDict**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_defaults()

class mozbuild.test.test_containers.**TestReadOnlyNamespace**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_basic()

mozbuild.test.test_dotproperties module

class mozbuild.test.test_dotproperties.**TestDotProperties**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_bad_unicode_from_file()

test_get()

test_get_dict()

test_get_dict_with_shared_prefix()

test_get_dict_with_value_prefix()

test_get_list()

test_get_list_with_shared_prefix()

test_unicode()

test_update()

test_valid_unicode_from_file()

mozbuild.test.test_expression module

class mozbuild.test.test_expression.**TestContext**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for the Context class

setUp()

test_in()
test 'var in context' to not fall for fallback

test_string_literal()
test string literal, fall-through for undefined var in a Context

test_variable()
test value for defined var in the Context class

class mozbuild.test.test_expression.**TestExpression**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for the Expression class evaluate() is called with a context {FAIL: 'PASS'}

setUp()

test_defined()
Test for the defined() value

test_equals()
Test for the == operator

test_logical_and()
Test for the && operator

test_logical_ops()
Test for the && and || operators precedence

test_logical_or()
Test for the || operator

test_not()
Test for the ! operator

test_notequals()
Test for the != operator

test_string_literal()
Test for a string literal in an Expression

test_variable()
Test for variable value in an Expression

mozbuild.test.test_jarmaker module

class mozbuild.test.test_jarmaker.**TestJarMaker**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for JarMaker.py

debug = False

setUp()

tearDown()

test_a_simple_jar()
Test a simple jar.mn

test_a_simple_symlink()
Test a simple jar.mn with a symlink

test_a_wildcard_jar()
Test a wildcard in jar.mn

test_a_wildcard_symlink()
Test a wildcard in jar.mn with symlinks

class mozbuild.test.test_jarmaker.**Test_relativesrcdir**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

tearDown()

test_en_US()

test_l10n_merge()

test_l10n_no_merge()

test_override()

test_override_l10n()

mozbuild.test.test_jarmaker.**is_symlink_to**(*dest*, *src*)

mozbuild.test.test_jarmaker.**symlinks_supported**(*path*)

mozbuild.test.test_line_endings module

class mozbuild.test.test_line_endings.**TestLineEndings**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for the Context class

createFile(*lineendings*)

setUp()

tearDown()

testMac()

testUnix()

testWindows()

mozbuild.test.test_makeutil module

class mozbuild.test.test_makeutil.**TestMakefile**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_makefile()

test_path_normalization(**args*, ***kwargs*)

test_read_dep_makefile()

test_rule()

test_statement()

test_write_dep_makefile()

mozbuild.test.test_mozconfig module

class mozbuild.test.test_mozconfig.**TestMozconfigLoader**(*methodName='runTest'*)
Bases: unittest.case.TestCase

get_loader()

get_temp_dir()

setUp()

tearDown()

test_find_abs_path_not_exist()
Ensure a missing absolute path is detected.

test_find_default_files()
Ensure default paths are used when present.

test_find_deprecated_home_paths()
Ensure we error when deprecated home directory paths are present.

test_find_deprecated_path_srcdir()
Ensure we error when deprecated path locations are present.

test_find_legacy_env()
Ensure legacy mozconfig path definitions result in error.

test_find_multiple_but_identical_configs()
Ensure multiple relative-path MOZCONFIGs pointing at the same file are OK.

test_find_multiple_configs()
Ensure multiple relative-path MOZCONFIGs result in error.

test_find_multiple_defaults()
Ensure we error when multiple default files are present.

test_find_no_relative_configs()
Ensure a missing relative-path MOZCONFIG is detected.

test_find_path_not_file()
Ensure non-file paths are detected.

test_find_relative_mozconfig()
Ensure a relative MOZCONFIG can be found in the srcdir.

test_read_ac_app_options()

test_read_ac_options_substitution()
Ensure ac_add_options values are substituted.

test_read_capture_ac_options()
Ensures ac_add_options calls are captured.

test_read_capture_mk_options()
Ensures mk_add_options calls are captured.

test_read_capture_mk_options_objdir_environ()
Ensures mk_add_options calls are captured and override the environ.

test_read_empty_mozconfig()

test_read_empty_mozconfig_objdir_environ()

test_read_empty_variable_value()
Ensure empty variable values are parsed properly.

test_read_exported_variables()
Exported variables are caught as new variables.

test_read_load_exception()
Ensure non-0 exit codes in mozconfigs are handled properly.

test_read_modify_variables()
Variables modified by mozconfig are detected.

test_read_moz_objdir_substitution()
Ensure @TOPSRCDIR@ substitution is recognized in MOZ_OBJDIR.

test_read_multiline_variables()
Ensure multi-line variables are captured properly.

test_read_new_variables()
New variables declared in mozconfig file are detected.

test_read_no_mozconfig()

test_read_removed_variables()
Variables unset by the mozconfig are detected.

test_read_topsrcdir_defined()
Ensure $topsrcdir references work as expected.

test_read_unmodified_variables()
Variables modified by mozconfig are detected.

mozbuild.test.test_mozinfo module

class mozbuild.test.test_mozinfo.**Base**
Bases: object

class mozbuild.test.test_mozinfo.**TestBuildDict**(*methodName='runTest'*)
Bases: unittest.case.TestCase, *mozbuild.test.test_mozinfo.Base*

test_android()

test_arm()
Test that all arm CPU architectures => arm.

test_crashreporter()
Test that crashreporter values are properly detected.

test_debug()
Test that debug values are properly detected.

test_linux()

test_mac()

test_mac_universal()

test_missing()
Test that missing required values raises.

test_unknown()
Test that unknown values pass through okay.

test_win()

test_x86()
Test that various i?86 values => x86.

class mozbuild.test.test_mozinfo.**TestWriteMozinfo**(*methodName='runTest'*)
Bases: unittest.case.TestCase, *mozbuild.test.test_mozinfo.Base*

Test the write_mozinfo function.

setUp()

tearDown()

test_basic()
Test that writing to a file produces correct output.

test_fileobj()
Test that writing to a file-like object produces correct output.

mozbuild.test.test_preprocessor module

class mozbuild.test.test_preprocessor.**TestPreprocessor**(*methodName='runTest'*)
Bases: unittest.case.TestCase

Unit tests for the Context class

do_include_compare(*content_lines*, *expected_lines*)

do_include_pass(*content_lines*)

setUp()

test_command_line_literal_at()

test_conditional_if_0()

test_conditional_if_0_elif_1()

test_conditional_if_0_or_1()

test_conditional_if_1()

test_conditional_if_1_elif_1_else()

test_conditional_if_1_if_1()

test_conditional_not_0()

test_conditional_not_0_and_1()

test_conditional_not_1()

test_conditional_not_emptyval()

test_conditional_not_nullval()

test_default_defines()

test_error()

test_expand()

test_filterDefine()

test_filter_attemptSubstitution()

test_filter_emptyLines()

test_filter_slashslash()

test_filter_spaces()

test_filter_substitution()

test_include()

test_include_line()

test_include_literal_at()

test_include_missing_file()

test_include_undefined_variable()

test_javascript_line()

test_literal()

test_no_marker()

test_number_value()

test_number_value_equals()

test_number_value_equals_defines()

test_number_value_not_equals_quoted_defines()

test_octal_value_equals()

test_octal_value_equals_defines()

test_octal_value_not_equals_quoted_defines()

test_octal_value_quoted_expansion()

test_string_value()

test_undef_defined()

test_undef_undefined()

test_undefined_variable()

test_value_quoted_expansion()
Quoted values on the commandline don't currently have quotes stripped. Pike says this is for compat reasons.

test_var_directory()

test_var_file()

test_var_if_0()

test_var_if_0_elifdef()

test_var_if_0_elifndef()

test_var_ifdef_0()

test_var_ifdef_1_or_undef()

test_var_ifdef_undef()

test_var_ifndef_0()

test_var_ifndef_0_and_undef()

test_var_ifndef_undef()

test_var_line()

mozbuild.test.test_pythonutil module

class mozbuild.test.test_pythonutil.**TestIterModules**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_iter_modules_in_path()

mozbuild.test.test_testing module

class mozbuild.test.test_testing.**Base**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

tearDown()

class mozbuild.test.test_testing.**TestTestMetadata**(*methodName='runTest'*)
Bases: *mozbuild.test.test_testing.Base*

test_load()

test_resolve_all()

test_resolve_by_dir()

test_resolve_filter_flavor()

test_resolve_multiple_paths()

test_resolve_path_prefix()

test_resolve_support_files()

test_resolve_under_path()

class mozbuild.test.test_testing.**TestTestResolver**(*methodName='runTest'*)
Bases: *mozbuild.test.test_testing.Base*

FAKE_TOPSRCDIR = u'/Users/gps/src/firefox'

setUp()

tearDown()

test_cwd_children_only()
If cwd is defined, only resolve tests under the specified cwd.

test_subsuites()
Test filtering by subsuite.

test_various_cwd()
Test various cwd conditions are all equal.

test_wildcard_patterns()
Test matching paths by wildcard.

mozbuild.test.test_util module

class mozbuild.test.test_util.**TestEnumString**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_string()

class mozbuild.test.test_util.**TestFileAvoidWrite**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_diff_create()
Diffs are produced when files are created.

test_diff_not_default()
Diffs are not produced by default.

test_diff_update()
Diffs are produced on file update.

test_file_avoid_write()

class mozbuild.test.test_util.**TestGroupUnifiedFiles**(*methodName='runTest'*)
Bases: unittest.case.TestCase

FILES = [u'a.cpp', u'b.cpp', u'c.cpp', u'd.cpp', u'e.cpp', u'f.cpp', u'g.cpp', u'h.cpp', u'i.cpp', u'j.cpp', u'k.cpp', u'l.cpp',

letter = 'z'

test_multiple_files()

test_unsorted_files()

class mozbuild.test.test_util.**TestHashing**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_hash_file_known_hash()
Ensure a known hash value is recreated.

test_hash_file_large()
Ensure that hash_file seems to work with a large file.

class mozbuild.test.test_util.**TestHierarchicalStringList**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

test_del_exports()

test_exports_append()

test_exports_multiple_subdir()

test_exports_subdir()

test_invalid_exports_append()

test_invalid_exports_append_base()

test_invalid_exports_bool()

test_invalid_exports_set()

test_merge()

test_reassign()

test_unsorted()

test_walk()

class mozbuild.test.test_util.**TestListWithAction**(*methodName='runTest'*)
Bases: unittest.case.TestCase

assertSameList(*expected*, *actual*)

setUp()

test_add()

test_extend()

test_iadd()

test_init()

test_slicing()

class mozbuild.test.test_util.**TestMemoize**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_memoize()

test_memoize_method()

test_memoized_property()

class mozbuild.test.test_util.**TestMisc**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_expand_variables()

test_pair()

class mozbuild.test.test_util.**TestResolveTargetToMake**(*methodName='runTest'*)
Bases: unittest.case.TestCase

assertResolve(*path*, *expected*)

setUp()

test_Makefile()

test_dir()

test_regular_file()

test_root_path()

test_top_level()

class mozbuild.test.test_util.**TestStrictOrderingOnAppendList**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_add()

test_add_StrictOrderingOnAppendList()

test_add_after_iadd()

test_extend()

test_iadd()

test_init()

test_slicing()

class mozbuild.test.test_util.**TestStrictOrderingOnAppendListWithFlagsFactory**(*methodName='runTest*
Bases: unittest.case.TestCase

test_strict_ordering_on_append_list_with_flags_factory()

test_strict_ordering_on_append_list_with_flags_factory_extend()

class mozbuild.test.test_util.**TestTypedList**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_add()

test_add_coercion()

test_extend()

test_iadd()

test_init()

test_memoized()

test_slicing()

class mozbuild.test.test_util.**TestTypedNamedTuple**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_simple()

class mozbuild.test.test_util.**TypedTestStrictOrderingOnAppendList**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_init()

Module contents

22.2.2 Submodules

22.2.3 mozbuild.android_version_code module

mozbuild.android_version_code.**android_version_code**(*buildid*, **args*, ***kwargs*)

mozbuild.android_version_code.**android_version_code_v0**(*buildid*, *cpu_arch=None*, *min_sdk=0*, *max_sdk=0*)

mozbuild.android_version_code.**android_version_code_v1**(*buildid*, *cpu_arch=None*, *min_sdk=0*, *max_sdk=0*)

Generate a v1 android:versionCode.

The important consideration is that version codes be monotonically increasing (per Android package name) for all published builds. The input build IDs are based on timestamps and hence are always monotonically increasing.

The generated v1 version codes look like (in binary):

0111 1000 0010 tttt tttt tttt tttt txpg

The 17 bits labelled 't' represent the number of hours since midnight on September 1, 2015. (2015090100 in YYYYMMMDDHH format.) This yields a little under 15 years worth of hourly build identifiers, since 2**17 / (366 * 24) =~ 14.92.

The bits labelled 'x', 'p', and 'g' are feature flags.

The bit labelled 'x' is 1 if the build is for an x86 architecture and 0 otherwise, which means the build is for an ARM architecture. (Fennec no longer supports ARMv6, so ARM is equivalent to ARMv7 and above.)

The bit labelled 'p' is a placeholder that is always 0 (for now).

Firefox no longer supports API 14 or earlier.

This version code computation allows for a split on API levels that allowed us to ship builds specifically for Gingerbread (API 9-10); we preserve that functionality for sanity's sake, and to allow us to reintroduce a split in the future.

At present, the bit labelled 'g' is 1 if the build is an ARM build targeting API 15+, which will always be the case.

We throw an explanatory exception when we are within one calendar year of running out of build events. This gives lots of time to update the version scheme. The responsible individual should then bump the range (to allow builds to continue) and use the time remaining to update the version scheme via the reserved high order bits.

N.B.: the reserved 0 bit to the left of the highest order 't' bit can, sometimes, be used to bump the version scheme. In addition, by reducing the granularity of the build identifiers (for example, moving to identifying builds every 2 or 4 hours), the version scheme may be adjusted further still without losing a (valuable) high order bit.

mozbuild.android_version_code.**main**(*argv*)

22.2.4 mozbuild.artifacts module

22.2.5 mozbuild.base module

exception mozbuild.base.**BadEnvironmentException**
Bases: exceptions.Exception

Base class for errors raised when the build environment is not sane.

exception mozbuild.base.**BuildEnvironmentNotFoundException**
Bases: *mozbuild.base.BadEnvironmentException*

Raised when we could not find a build environment.

class mozbuild.base.**ExecutionSummary**(*summary_format*, ***data*)
Bases: dict

Helper for execution summaries.

extend(*summary_format*, ***data*)

class mozbuild.base.**MachCommandBase**(*context*)
Bases: *mozbuild.base.MozbuildObject*

Base class for mach command providers that wish to be MozbuildObjects.

This provides a level of indirection so MozbuildObject can be refactored without having to change everything that inherits from it.

class mozbuild.base.**MachCommandConditions**
Bases: object

A series of commonly used condition functions which can be applied to mach commands with providers deriving from MachCommandBase.

static is_android()
Must have an Android build.

static is_b2g()
Must have a B2G build.

static is_b2g_desktop()
Must have a B2G desktop build.

static is_emulator()
Must have a B2G build with an emulator configured.

static is_firefox()
Must have a Firefox build.

static is_git()
Must have a git source checkout.

static is_hg()
Must have a mercurial source checkout.

static is_mulet()
Must have a Mulet build.

class mozbuild.base.**MozbuildObject**(*topsrcdir*, *settings*, *log_manager*, *topobjdir=None*, *mozconfig=<object object>*)
Bases: *mach.mixin.process.ProcessExecutionMixin*

Base class providing basic functionality useful to many modules.

Modules in this package typically require common functionality such as accessing the current config, getting the location of the source directory, running processes, etc. This classes provides that functionality. Other modules can inherit from this class to obtain this functionality easily.

bindir

config_environment
Returns the ConfigEnvironment for the current build configuration.

This property is only available once configure has executed.

If configure's output is not available, this will raise.

defines

distdir

classmethod from_environment(*cwd=None*, *detect_virtualenv_mozinfo=True*)
Create a MozbuildObject by detecting the proper one from the env.

This examines environment state like the current working directory and creates a MozbuildObject from the found source directory, mozconfig, etc.

The role of this function is to identify a topsrcdir, topobjdir, and mozconfig file.

If the current working directory is inside a known objdir, we always use the topsrcdir and mozconfig associated with that objdir.

If the current working directory is inside a known srcdir, we use that topsrcdir and look for mozconfigs using the default mechanism, which looks inside environment variables.

If the current Python interpreter is running from a virtualenv inside an objdir, we use that as our objdir.

If we're not inside a srcdir or objdir, an exception is raised.

detect_virtualenv_mozinfo determines whether we should look for a mozinfo.json file relative to the virtualenv directory. This was added to facilitate testing. Callers likely shouldn't change the default.

get_binary_path(*what=u'app'*, *validate_exists=True*, *where=u'default'*)
Obtain the path to a compiled binary for this build configuration.

The what argument is the program or tool being sought after. See the code implementation for supported values.

If validate_exists is True (the default), we will ensure the found path exists before returning, raising an exception if it doesn't.

If where is 'staged-package', we will return the path to the binary in the package staging directory.

If no arguments are specified, we will return the main binary for the configured XUL application.

have_winrm()

includedir

is_clobber_needed()

mozconfig
Returns information about the current mozconfig file.

This a dict as returned by MozconfigLoader.read_mozconfig()

non_global_defines

notify(*msg*)
Show a desktop notification with the supplied message

On Linux and Mac, this will show a desktop notification with the message, but on Windows we can only flash the screen.

remove_objdir()
Remove the entire object directory.

static resolve_config_guess(*mozconfig*, *topsrcdir*)

static resolve_mozconfig_topobjdir(*topsrcdir*, *mozconfig*, *default=None*)

statedir

substs

topobjdir

virtualenv_manager

exception mozbuild.base.**ObjdirMismatchException**(*objdir1*, *objdir2*)
Bases: *mozbuild.base.BadEnvironmentException*

Raised when the current dir is an objdir and doesn't match the mozconfig.

class mozbuild.base.**PathArgument**(*arg*, *topsrcdir*, *topobjdir*, *cwd=None*)
Bases: object

Parse a filesystem path argument and transform it in various ways.

objdir_path()

relpath()
Return a path relative to the topsrcdir or topobjdir.

If the argument is a path to a location in one of the base directories (topsrcdir or topobjdir), then strip off the base directory part and just return the path within the base directory.

srcdir_path()

mozbuild.base.**ancestors**(*path*)
Emit the parent directories of a path.

mozbuild.base.**samepath**(*path1*, *path2*)

22.2.6 mozbuild.config_status module

mozbuild.config_status.**config_status**(*topobjdir='.'*, *topsrcdir='.'*, *defines=None*, *non_global_defines=None*, *substs=None*, *source=None*, *mozconfig=None*)
Main function, providing config.status functionality.

Contrary to config.status, it doesn't use CONFIG_FILES or CONFIG_HEADERS variables.

Without the -n option, this program acts as config.status and considers the current directory as the top object directory, even when config.status is in a different directory. It will, however, treat the directory containing config.status as the top object directory with the -n option.

The options to this function are passed when creating the ConfigEnvironment. These lists, as well as the actual wrapper script around this function, are meant to be generated by configure. See build/autoconf/config.status.m4.

22.2.7 mozbuild.doctor module

class mozbuild.doctor.**Doctor**(*srcdir*, *objdir*, *fix*)
Bases: object

check_all()

check_disk_8dot3(*path*, *disk*)

check_mount_lastaccess(*mount*)

cpu

fs_8dot3

fs_lastaccess

getmount(*path*)

memory

mozillabuild

platform

prompt_bool(*prompt*, *limit=5*)
Prompts the user with prompt and requires a boolean value.

report(*results*)

storage_freespace

22.2.8 mozbuild.dotproperties module

class mozbuild.dotproperties.**DotProperties**(*file=None*)
A thin representation of a key=value .properties file.

get(*key*, *default=None*)

get_dict(*prefix*, *required_keys=[]*)
Turns {'foo.title':'title', ...} into {'title':'title', ...}.

If |required_keys| is present, it must be an iterable of required key names. If a required key is not present, ValueError is thrown.

Returns {} to indicate an empty or missing dict.

get_list(*prefix*)
Turns {'list.0':'foo', 'list.1':'bar'} into ['foo', 'bar'].

Returns [] to indicate an empty or missing list.

update(*file*)
Updates properties from a file name or file-like object.

Ignores empty lines and comment lines.

22.2.9 mozbuild.html_build_viewer module

class mozbuild.html_build_viewer.**BuildViewerServer**(*address=u'localhost'*, *port=0*)
Bases: object

add_resource_json_file(*key*, *path*)
Register a resource JSON file with the server.

The file will be made available under the name/key specified.

add_resource_json_url(*key*, *url*)
Register a resource JSON file at a URL.

run()

url

class mozbuild.html_build_viewer.**HTTPHandler**(*request*, *client_address*, *server*)
Bases: BaseHTTPServer.BaseHTTPRequestHandler

do_GET()

do_POST()

serve_docroot(*root*, *path*)

22.2.10 mozbuild.jar module

jarmaker.py provides a python class to package up chrome content by processing jar.mn files.

See the documentation for jar.mn on MDC for further details on the format.

class mozbuild.jar.**JarMaker**(*outputFormat='flat'*, *useJarfileManifest=True*, *useChromeManifest=False*)
Bases: object

JarMaker reads jar.mn files and process those into jar files or flat directories, along with chrome.manifest files.

class OutputHelper_flat(*basepath*)
Bases: object

Provide getDestModTime and getOutput for a given flat output directory. The helper method ensureDirFor is used by the symlink subclass.

ensureDirFor(*name*)

getDestModTime(*aPath*)

getOutput(*name*)

class JarMaker.**OutputHelper_jar**(*jarfile*)
Bases: object

Provide getDestModTime and getOutput for a given jarfile.

getDestModTime(*aPath*)

getOutput(*name*)

class JarMaker.**OutputHelper_symlink**(*basepath*)
Bases: mozbuild.jar.OutputHelper_flat

Subclass of OutputHelper_flat that provides a helper for creating a symlink including creating the parent directories.

symlink(*src*, *dest*)

JarMaker.**finalizeJar**(*jardir*, *jarbase*, *jarname*, *chromebasepath*, *register*, *doZip=True*)

Helper method to write out the chrome registration entries to jarfile.manifest or chrome.manifest, or both.

The actual file processing is done in updateManifest.

JarMaker.**generateLocaleDirs**(*relativesrcdir*)

JarMaker.**getCommandLineParser**()
Get a optparse.OptionParser for jarmaker.

This OptionParser has the options for jarmaker as well as the options for the inner PreProcessor.

JarMaker.**makeJar**(*infile*, *jardir*)
makeJar is the main entry point to JarMaker.

It takes the input file, the output directory, the source dirs and the top source dir as argument, and optionally the l10n dirs.

JarMaker.**processJarSection**(*jarinfo*, *jardir*)
Internal method called by makeJar to actually process a section of a jar.mn file.

JarMaker.**updateManifest**(*manifestPath*, *chromebasepath*, *register*)
updateManifest replaces the % in the chrome registration entries with the given chrome base path, and updates the given manifest file.

22.2.11 mozbuild.mach_commands module

22.2.12 mozbuild.makeutil module

class mozbuild.makeutil.**Makefile**
Bases: object

Provides an interface for writing simple makefiles

Instances of this class are created, populated with rules, then written.

add_statement(*statement*)
Add a raw statement in the makefile. Meant to be used for simple variable assignments.

create_rule(*targets=[]*)
Create a new rule in the makefile for the given targets. Returns the corresponding Rule instance.

dump(*fh*, *removal_guard=True*)
Dump all the rules to the given file handle. Optionally (and by default), add guard rules for file removals (empty rules for other rules' dependencies)

class mozbuild.makeutil.**Rule**(*targets=[]*)
Bases: object

Class handling simple rules in the form: target1 target2 ... : dep1 dep2 ...

command1 command2 ...

add_commands(*commands*)
Add commands to the rule.

add_dependencies(*deps*)
Add dependencies to the rule.

add_targets(*targets*)
Add additional targets to the rule.

commands()
Return an iterator on the rule commands.

dependencies()
Return an iterator on the rule dependencies.

dump(*fh*)
Dump the rule to the given file handle.

targets()
Return an iterator on the rule targets.

mozbuild.makeutil.**read_dep_makefile**(*fh*)
Read the file handler containing a dep makefile (simple makefile only containing dependencies) and returns an iterator of the corresponding Rules it contains. Ignores removal guard rules.

mozbuild.makeutil.**write_dep_makefile**(*fh*, *target*, *deps*)
Write a Makefile containing only target's dependencies to the file handle specified.

22.2.13 mozbuild.milestone module

mozbuild.milestone.**get_milestone_ab_with_num**(*milestone*)
Returns the alpha and beta tag with its number (a1, a2, b3, ...).

mozbuild.milestone.**get_milestone_major**(*milestone*)
Returns the major (first) part of the milestone.

mozbuild.milestone.**get_official_milestone**(*path*)
Returns the contents of the first line in *path* that starts with a digit.

mozbuild.milestone.**main**(*args*)

22.2.14 mozbuild.mozconfig module

exception mozbuild.mozconfig.**MozconfigFindException**
Bases: exceptions.Exception

Raised when a mozconfig location is not defined properly.

exception mozbuild.mozconfig.**MozconfigLoadException**(*path*, *message*, *output=None*)
Bases: exceptions.Exception

Raised when a mozconfig could not be loaded properly.

This typically indicates a malformed or misbehaving mozconfig file.

class mozbuild.mozconfig.**MozconfigLoader**(*topsrcdir*)
Bases: object

Handles loading and parsing of mozconfig files.

AUTODETECT = <object object>

DEFAULT_TOPSRCDIR_PATHS = (u'.mozconfig', u'mozconfig')

DEPRECATED_HOME_PATHS = (u'.mozconfig', u'.mozconfig.sh', u'.mozmyconfig.sh')

DEPRECATED_TOPSRCDIR_PATHS = (u'mozconfig.sh', u'myconfig.sh')

ENVIRONMENT_VARIABLES = set([u'LDFLAGS', u'CXX', u'CXXFLAGS', u'CC', u'CFLAGS', u'MOZ_OBJDIR'])

IGNORE_SHELL_VARIABLES = set([u'_'])

RE_MAKE_VARIABLE = <_sre.SRE_Pattern object>

find_mozconfig(*env={'LANG': 'C.UTF-8', 'READTHEDOCS_PROJECT': 'gfritzsche-demo', 'READTHEDOCS': 'True', 'APPDIR': '/app', 'DEBIAN_FRONTEND': 'noninteractive', 'OLDPWD': '/', 'HOSTNAME': 'build-4258433-project-55928-gfritzsche-demo', u'SHELL': u'/bin/bash', 'PWD': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs', 'BIN_PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/envs/latest/bin', 'READTHEDOCS_VERSION': 'latest', 'PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs/_build/latex/_venv/bin:/home/docs/checkouts/readthedocs.org/user_build*
demo/envs/latest/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin', 'HOME': '/home/docs'})
Find the active mozconfig file for the current environment.

This emulates the logic in mozconfig-find.

1. If ENV[MOZCONFIG] is set, use that
2. If $TOPSRCDIR/mozconfig or $TOPSRCDIR/.mozconfig exists, use it.
3. If both exist or if there are legacy locations detected, error out.

The absolute path to the found mozconfig will be returned on success. None will be returned if no mozconfig could be found. A MozconfigFindException will be raised if there is a bad state, including conditions from #3 above.

read_mozconfig(*path=None*, *moz_build_app=None*)
Read the contents of a mozconfig into a data structure.

This takes the path to a mozconfig to load. If the given path is AUTODETECT, will try to find a mozconfig from the environment using find_mozconfig().

mozconfig files are shell scripts. So, we can't just parse them. Instead, we run the shell script in a wrapper which allows us to record state from execution. Thus, the output from a mozconfig is a friendly static data structure.

22.2.15 mozbuild.mozinfo module

mozbuild.mozinfo.**build_dict**(*config*, *env={'LANG': 'C.UTF-8', 'READTHEDOCS_PROJECT': 'gfritzsche-demo', 'READTHEDOCS': 'True', 'APPDIR': '/app', 'DEBIAN_FRONTEND': 'noninteractive', 'OLDPWD': '/', 'HOSTNAME': 'build-4258433-project-55928-gfritzsche-demo', u'SHELL': u'/bin/bash', 'PWD': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs', 'BIN_PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/envs/latest/bin', 'READTHEDOCS_VERSION': 'latest', 'PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs/_build/latex/_venv/bin:/home/docs/checkouts/readthedocs.or*
demo/envs/latest/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin', 'HOME': '/home/docs'})

Build a dict containing data about the build configuration from the environment.

mozbuild.mozinfo.**write_mozinfo**(*file*, *config*, *env={'LANG': 'C.UTF-8', 'READTHEDOCS_PROJECT': 'gfritzsche-demo', 'READTHEDOCS': 'True', 'APPDIR': '/app', 'DEBIAN_FRONTEND': 'noninteractive', 'OLDPWD': '/', 'HOSTNAME': 'build-4258433-project-55928-gfritzsche-demo', u'SHELL': u'/bin/bash', 'PWD': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs', 'BIN_PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/envs/latest/bin', 'READTHEDOCS_VERSION': 'latest', 'PATH': '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs/_build/latex/_venv/bin:/home/docs/checkouts/readthedo*
demo/envs/latest/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin', 'HOME': '/home/docs'})

Write JSON data about the configuration specified in config and an environment variable dict to **|file|**, which may be a filename or file-like object. See build_dict for information about what environment variables are used, and what keys are produced.

22.2.16 mozbuild.preprocessor module

This is a very primitive line based preprocessor, for times when using a C preprocessor isn't an option.

It currently supports the following grammar for expressions, whitespace is ignored:

expression : and_cond ('||' expression) ? ;

and_cond: test ('&&' and_cond) ? ;

test: unary (('==' | '!=') unary) ? ;

unary : '!'? value ;

value : [0-9]+ # integer | 'defined(' w+ ')' | w+ # string identifier or value;

class mozbuild.preprocessor.**Context**

Bases: dict

This class holds variable values by subclassing dict, and while it truthfully reports True and False on

name in context

it returns the variable name itself on

context["name"]

to reflect the ambiguity between string literals and preprocessor variables.

class mozbuild.preprocessor.**Expression**(*expression_string*)

exception ParseError(*expression*)
Bases: exceptions.StandardError

Error raised when parsing fails. It has two members, offset and content, which give the offset of the error and the offending content.

Expression.**evaluate**(*context*)
Evaluate the expression with the given context

class mozbuild.preprocessor.**Preprocessor**(*defines=None*, *marker='#'*)
Class for preprocessing text files.

exception Error(*cpp*, *MSG*, *context*)
Bases: exceptions.RuntimeError

Preprocessor.**addDefines**(*defines*)
Adds the specified defines to the preprocessor. defines may be a dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two)

Preprocessor.**applyFilters**(*aLine*)

Preprocessor.**clone**()
Create a clone of the current processor, including line ending settings, marker, variable definitions, output stream.

Preprocessor.**computeDependencies**(*input*)
Reads the input stream, and computes the dependencies for that input.

Preprocessor.**do_define**(*args*)

Preprocessor.**do_elif**(*args*)

Preprocessor.**do_elifdef**(*args*)

Preprocessor.**do_elifndef**(*args*)

Preprocessor.**do_else**(*args*, *ifState=2*)

Preprocessor.**do_endif**(*args*)

Preprocessor.**do_error**(*args*)

Preprocessor.**do_expand**(*args*)

Preprocessor.**do_filter**(*args*)

Preprocessor.**do_if**(*args*, *replace=False*)

Preprocessor.**do_ifdef**(*args*, *replace=False*)

Preprocessor.**do_ifndef**(*args*, *replace=False*)

Preprocessor.**do_include**(*args*, *filters=True*)
Preprocess a given file. args can either be a file name, or a file-like object. Files should be opened, and will be closed after processing.

Preprocessor.**do_includesubst**(*args*)

Preprocessor.**do_literal**(*args*)

Preprocessor.**do_undef**(*args*)

Preprocessor.**do_unfilter**(*args*)

Preprocessor.**ensure_not_else**()

Preprocessor.**failUnused**(*file*)

Preprocessor.**filter_attemptSubstitution**(*aLine*)

Preprocessor.**filter_emptyLines**(*aLine*)

Preprocessor.**filter_slashslash**(*aLine*)

Preprocessor.**filter_spaces**(*aLine*)

Preprocessor.**filter_substitution**(*aLine*, *fatal=True*)

Preprocessor.**getCommandLineParser**(*unescapeDefines=False*)

Preprocessor.**handleCommandLine**(*args*, *defaultToStdin=False*)
Parse a commandline into this parser. Uses OptionParser internally, no args mean sys.argv[1:].

Preprocessor.**handleLine**(*aLine*)
Handle a single line of input (internal).

Preprocessor.**noteLineInfo**()

Preprocessor.**processFile**(*input*, *output*, *depfile=None*)
Preprocesses the contents of the `input` stream and writes the result to the `output` stream. If `depfile` is set, the dependencies of `output` file are written to `depfile` in Makefile format.

Preprocessor.**setMarker**(*aMarker*)
Set the marker to be used for processing directives. Used for handling CSS files, with pp.setMarker('%'), for example. The given marker may be None, in which case no markers are processed.

Preprocessor.**setSilenceDirectiveWarnings**(*value*)
Sets whether missing directive warnings are silenced, according to `value`. The default behavior of the preprocessor is to emit such warnings.

Preprocessor.**write**(*aLine*)
Internal method for handling output.

mozbuild.preprocessor.**preprocess**(*includes=[<open file '<stdin>', mode 'r' at 0x7f86286050c0>], defines={}, output=<open file '<stdout>', mode 'w'>, marker='#'*)

22.2.17 mozbuild.pythonutil module

mozbuild.pythonutil.**iter_modules_in_path**(**paths*)

22.2.18 mozbuild.shellutil module

exception mozbuild.shellutil.**MetaCharacterException**(*char*)
Bases: exceptions.Exception

mozbuild.shellutil.**split**(*cline*)
Split the given command line string.

mozbuild.shellutil.**quote**(**strings*)
Given one or more strings, returns a quoted string that can be used literally on a shell command line.

```
>>> quote('a', 'b')
"a b"
>>> quote('a b', 'c')
"'a b' c"
```

22.2.19 mozbuild.sphinx module

class mozbuild.sphinx.**MozbuildSymbols**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)
Bases: docutils.parsers.rst.Directive

Directive to insert mozbuild sandbox symbol information.

required_arguments = 1

run()

mozbuild.sphinx.**format_module**(*m*)

mozbuild.sphinx.**function_reference**(*f*, *attr*, *args*, *doc*)

mozbuild.sphinx.**setup**(*app*)

mozbuild.sphinx.**special_reference**(*v*, *func*, *typ*, *doc*)

mozbuild.sphinx.**variable_reference**(*v*, *st_type*, *in_type*, *doc*)

22.2.20 mozbuild.testing module

class mozbuild.testing.**SupportFilesConverter**
Bases: object

Processes a "support-files" entry from a test object, either from a parsed object from a test manifests or its representation in moz.build and returns the installs to perform for this test object.

Processing the same support files multiple times will not have any further effect, and the structure of the parsed objects from manifests will have a lot of repeated entries, so this class takes care of memoizing.

convert_support_files(*test*, *install_root*, *manifest_dir*, *out_dir*)

class mozbuild.testing.**TestInstallInfo**
Bases: object

class mozbuild.testing.**TestMetadata**(*filename=None*)
Bases: object

Holds information about tests.

This class provides an API to query tests active in the build configuration.

resolve_tests(*paths=None*, *flavor=None*, *subsuite=None*, *under_path=None*, *tags=None*)
Resolve tests from an identifier.

This is a generator of dicts describing each test.

paths can be an iterable of values to use to identify tests to run. If an entry is a known test file, tests associated with that file are returned (there may be multiple configurations for a single file). If an entry is a directory, or a prefix of a directory containing tests, all tests in that directory are returned. If the

string appears in a known test file, that test file is considered. If the path contains a wildcard pattern, tests matching that pattern are returned.

If `under_path` is a string, it will be used to filter out tests that aren't in the specified path prefix relative to topsrcdir or the test's installed dir.

If `flavor` is a string, it will be used to filter returned tests to only be the flavor specified. A flavor is something like `xpcshell`.

If `subsuite` is a string, it will be used to filter returned tests to only be in the subsuite specified.

If `tags` are specified, they will be used to filter returned tests to only those with a matching tag.

tests_with_flavor(*flavor*)
Obtain all tests having the specified flavor.

This is a generator of dicts describing each test.

class mozbuild.testing.**TestResolver**(**args*, ***kwargs*)
Bases: *mozbuild.base.MozbuildObject*

Helper to resolve tests from the current environment to test files.

resolve_tests(*cwd=None*, ***kwargs*)
Resolve tests in the context of the current environment.

This is a more intelligent version of TestMetadata.resolve_tests().

This function provides additional massaging and filtering of low-level results.

Paths in returned tests are automatically translated to the paths in the _tests directory under the object directory.

If cwd is defined, we will limit our results to tests under the directory specified. The directory should be defined as an absolute path under topsrcdir or topobjdir for it to work properly.

mozbuild.testing.**all_test_flavors**()

mozbuild.testing.**install_test_files**(*topsrcdir*, *topobjdir*, *tests_root*, *test_objs*)
Installs the requested test files to the objdir. This is invoked by test runners to avoid installing tens of thousands of test files when only a few tests need to be run.

mozbuild.testing.**read_manifestparser_manifest**(*context*, *manifest_path*)

mozbuild.testing.**read_reftest_manifest**(*context*, *manifest_path*)

mozbuild.testing.**read_wpt_manifest**(*context*, *paths*)

mozbuild.testing.**rewrite_test_base**(*test*, *new_base*, *honor_install_to_subdir=False*)
Rewrite paths in a test to be under a new base path.

This is useful for running tests from a separate location from where they were defined.

honor_install_to_subdir and the underlying install-to-subdir field are a giant hack intended to work around the restriction where the mochitest runner can't handle single test files with multiple configurations. This argument should be removed once the mochitest runner talks manifests (bug 984670).

22.2.21 mozbuild.util module

class mozbuild.util.**DefinesAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)
Bases: `argparse.Action`

An ArgumentParser action to handle -Dvar[=value] type of arguments.

class mozbuild.util.**EmptyValue**
Bases: unicode

A dummy type that behaves like an empty string and sequence.

This type exists in order to support *mozbuild.frontend.reader.EmptyConfig*. It should likely not be used elsewhere.

class mozbuild.util.**EnumString**(*value*)
Bases: unicode

A string type that only can have a limited set of values, similarly to an Enum, and can only be compared against that set of values.

The class is meant to be subclassed, where the subclass defines POSSIBLE_VALUES. The *subclass* method is a helper to create such subclasses.

POSSIBLE_VALUES = ()

static subclass(**possible_values*)

exception mozbuild.util.**EnumStringComparisonError**
Bases: exceptions.Exception

class mozbuild.util.**FileAvoidWrite**(*filename*, *capture_diff=False*, *dry_run=False*, *mode=u'rU'*)
Bases: _io.BytesIO

File-like object that buffers output and only writes if content changed.

We create an instance from an existing filename. New content is written to it. When we close the file object, if the content in the in-memory buffer differs from what is on disk, then we write out the new content. Otherwise, the original file is untouched.

Instances can optionally capture diffs of file changes. This feature is not enabled by default because it a) doesn't make sense for binary files b) could add unwanted overhead to calls.

Additionally, there is dry run mode where the file is not actually written out, but reports whether the file was existing and would have been updated still occur, as well as diff capture if requested.

close()
Stop accepting writes, compare file contents, and rewrite if needed.

Returns a tuple of bools indicating what action was performed:

(file existed, file updated)

If `capture_diff` was specified at construction time and the underlying file was changed, `.diff` will be populated with the diff of the result.

write(*buf*)

mozbuild.util.**FlagsFactory**(*flags*)
Returns a class which holds optional flags for an item in a list.

The flags are defined in the dict given as argument, where keys are the flag names, and values the type used for the value of that flag.

The resulting class is used by the various <TypeName>WithFlagsFactory functions below.

class mozbuild.util.**HierarchicalStringList**
Bases: object

A hierarchy of lists of strings.

Each instance of this object contains a list of strings, which can be set or appended to. A sub-level of the hierarchy is also an instance of this class, can be added by appending to an attribute instead.

For example, the moz.build variable EXPORTS is an instance of this class. We can do:

EXPORTS += ['foo.h'] EXPORTS.mozilla.dom += ['bar.h']

In this case, we have 3 instances (EXPORTS, EXPORTS.mozilla, and EXPORTS.mozilla.dom), and the first and last each have one element in their list.

class `StringListAdaptor`(*hsl*)
Bases: `_abcoll.Sequence`

`HierarchicalStringList.`**`walk`**`()`
Walk over all HierarchicalStringLists in the hierarchy.

This is a generator of (path, sequence).

The path is '' for the root level and '/'-delimited strings for any descendants. The sequence is a read-only sequence of the strings contained at that level.

class `mozbuild.util.`**`KeyedDefaultDict`**(*default_factory*, **args*, ***kwargs*)
Bases: `dict`

Like a defaultdict, but the default_factory function takes the key as argument

class `mozbuild.util.`**`List`**(*iterable=None*, ***kwargs*)
Bases: *mozbuild.util.ListMixin*, `list`

A list specialized for moz.build environments.

We overload the assignment and append operations to require that the appended thing is a list. This avoids bad surprises coming from appending a string to a list, which would just add each letter of the string.

class `mozbuild.util.`**`ListMixin`**(*iterable=None*, ***kwargs*)
Bases: `object`

`extend`(*l*)

class `mozbuild.util.`**`ListWithAction`**(*iterable=None*, ***kwargs*)
Bases: *mozbuild.util.ListMixin*, *mozbuild.util.ListWithActionMixin*, `list`

A list that accepts a callable to be applied to each item.

A callable (action) may optionally be passed to the constructor to run on each item of input. The result of calling the callable on each item will be stored in place of the original input.

class `mozbuild.util.`**`ListWithActionMixin`**(*iterable=None*, *action=None*)
Bases: `object`

Mixin to create lists with pre-processing. See ListWithAction.

`extend`(*l*)

class `mozbuild.util.`**`LockFile`**(*lockfile*)
Bases: `object`

LockFile is used by the lock_file method to hold the lock.

This object should not be used directly, but only through the lock_file method below.

exception `mozbuild.util.`**`MozbuildDeletionError`**
Bases: `exceptions.Exception`

class `mozbuild.util.`**`OrderedDefaultDict`**(*default_factory*, **args*, ***kwargs*)
Bases: `collections.OrderedDict`

A combination of OrderedDict and defaultdict.

class mozbuild.util.**ReadOnlyDefaultDict**(*default_factory*, **args*, ***kwargs*)
Bases: *mozbuild.util.ReadOnlyDict*

A read-only dictionary that supports default values on retrieval.

class mozbuild.util.**ReadOnlyDict**(**args*, ***kwargs*)
Bases: dict

A read-only dictionary.

update(**args*, ***kwargs*)

class mozbuild.util.**ReadOnlyKeyedDefaultDict**(*default_factory*, **args*, ***kwargs*)
Bases: *mozbuild.util.KeyedDefaultDict*, *mozbuild.util.ReadOnlyDict*

Like KeyedDefaultDict, but read-only.

class mozbuild.util.**ReadOnlyNamespace**(***kwargs*)
Bases: object

A class for objects with immutable attributes set at initialization.

class mozbuild.util.**StrictOrderingOnAppendList**(*iterable=None*, ***kwargs*)
Bases: *mozbuild.util.ListMixin*, *mozbuild.util.StrictOrderingOnAppendListMixin*, list

A list specialized for moz.build environments.

We overload the assignment and append operations to require that incoming elements be ordered. This enforces cleaner style in moz.build files.

class mozbuild.util.**StrictOrderingOnAppendListMixin**(*iterable=None*, ***kwargs*)
Bases: object

static ensure_sorted(*l*)

extend(*l*)

class mozbuild.util.**StrictOrderingOnAppendListWithAction**(*iterable=None*, ***kwargs*)
Bases: *mozbuild.util.StrictOrderingOnAppendListMixin*, *mozbuild.util.ListMixin*, *mozbuild.util.ListWithActionMixin*, list

An ordered list that accepts a callable to be applied to each item.

A callable (action) passed to the constructor is run on each item of input. The result of running the callable on each item will be stored in place of the original input, but the original item must be used to enforce sortedness. Note that the order of superclasses is therefore significant.

class mozbuild.util.**StrictOrderingOnAppendListWithFlags**(*iterable=None*, ***kwargs*)
Bases: *mozbuild.util.StrictOrderingOnAppendList*

A list with flags specialized for moz.build environments.

Each subclass has a set of typed flags; this class lets us use *isinstance* for natural testing.

mozbuild.util.**StrictOrderingOnAppendListWithFlagsFactory**(*flags*)
Returns a StrictOrderingOnAppendList-like object, with optional flags on each item.

The flags are defined in the dict given as argument, where keys are the flag names, and values the type used for the value of that flag.

Example:

FooList = StrictOrderingOnAppendListWithFlagsFactory({ 'foo': bool, 'bar': unicode

}) foo = FooList(['a', 'b', 'c']) foo['a'].foo = True foo['b'].bar = 'bar'

mozbuild.util.**TypedList**
A list with type coercion.

The given `type` is what list elements are being coerced to. It may do strict validation, throwing ValueError exceptions.

A `base_class` type can be given for more specific uses than a List. For example, a Typed StrictOrderingOnAppendList can be created with:

TypedList(unicode, StrictOrderingOnAppendList)

class mozbuild.util.**TypedListMixin**(*iterable=None*, ***kwargs*)
Bases: `object`

Mixin for a list with type coercion. See TypedList.

append(*other*)

extend(*l*)

mozbuild.util.**TypedNamedTuple**(*name*, *fields*)
Factory for named tuple types with strong typing.

Arguments are an iterable of 2-tuples. The first member is the the field name. The second member is a type the field will be validated to be.

Construction of instances varies from `collections.namedtuple`.

First, if a single tuple argument is given to the constructor, this is treated as the equivalent of passing each tuple value as a separate argument into __init__. e.g.:

```
t = (1, 2)
TypedTuple(t) == TypedTuple(1, 2)
```

This behavior is meant for moz.build files, so vanilla tuples are automatically cast to typed tuple instances.

Second, fields in the tuple are validated to be instances of the specified type. This is done via an `isinstance()` check. To allow multiple types, pass a tuple as the allowed types field.

exception mozbuild.util.**UnsortedError**(*srtd*, *original*)
Bases: `exceptions.Exception`

mozbuild.util.**ensureParentDir**(*path*)
Ensures the directory parent to the given file exists.

mozbuild.util.**exec_**(*object*, *globals=None*, *locals=None*)
Wrapper around the exec statement to avoid bogus errors like:

SyntaxError: unqualified exec is not allowed in function ... it is a nested function.

or

SyntaxError: unqualified exec is not allowed in function ... it contains a nested function with free variable

which happen with older versions of python 2.7.

mozbuild.util.**expand_variables**(*s*, *variables*)
Given a string with $(var) variable references, replace those references with the corresponding entries from the given *variables* dict.

If a variable value is not a string, it is iterated and its items are joined with a whitespace.

mozbuild.util.**group_unified_files**(*files*, *unified_prefix*, *unified_suffix*, *files_per_unified_file*)
Return an iterator of (unified_filename, source_filenames) tuples.

We compile most C and C++ files in "unified mode"; instead of compiling `a.cpp`, `b.cpp`, and `c.cpp` separately, we compile a single file that looks approximately like:

```
#include "a.cpp"
#include "b.cpp"
#include "c.cpp"
```

This function handles the details of generating names for the unified files, and determining which original source files go in which unified file.

mozbuild.util.**hash_file**(*path*, *hasher=None*)
Hashes a file specified by the path given and returns the hex digest.

mozbuild.util.**lock_file**(*lockfile*, *max_wait=600*)
Create and hold a lockfile of the given name, with the given timeout.

To release the lock, delete the returned object.

class mozbuild.util.**memoize**(*func*)
Bases: `dict`

A decorator to memoize the results of function calls depending on its arguments. Both functions and instance methods are handled, although in the instance method case, the results are cache in the instance itself.

method_call(*instance*, **args*)

class mozbuild.util.**memoized_property**(*func*)
Bases: `object`

A specialized version of the memoize decorator that works for class instance properties.

mozbuild.util.**mkdir**(*path*, *not_indexed=False*)
Ensure a directory exists.

If `not_indexed` is True, an attribute is set that disables content indexing on the directory.

mozbuild.util.**pair**(*iterable*)
Given an iterable, returns an iterable pairing its items.

For example, list(pair([1,2,3,4,5,6]))

returns [(1,2), (3,4), (5,6)]

mozbuild.util.**resolve_target_to_make**(*topobjdir*, *target*)
Resolve *target* (a target, directory, or file) to a make target.

topobjdir is the object directory; all make targets will be rooted at or below the top-level Makefile in this directory.

Returns a pair *(reldir, target)* where *reldir* is a directory relative to *topobjdir* containing a Makefile and *target* is a make target (possibly *None*).

A directory resolves to the nearest directory at or above containing a Makefile, and target *None*.

A regular (non-Makefile) file resolves to the nearest directory at or above the file containing a Makefile, and an appropriate target.

A Makefile resolves to the nearest parent strictly above the Makefile containing a different Makefile, and an appropriate target.

mozbuild.util.**simple_diff**(*filename*, *old_lines*, *new_lines*)
Returns the diff between old_lines and new_lines, in unified diff form, as a list of lines.

old_lines and new_lines are lists of non-newline terminated lines to compare. old_lines can be None, indicating a file creation. new_lines can be None, indicating a file deletion.

class mozbuild.util.**undefined_default**
Bases: object

Represents an undefined argument value that isn't None.

22.2.22 mozbuild.virtualenv module

class mozbuild.virtualenv.**VirtualenvManager**(*topsrcdir*, *topobjdir*, *virtualenv_path*, *log_handle*, *manifest_path*)
Bases: object

Contains logic for managing virtualenvs for building the tree.

activate()
Activate the virtualenv in this Python context.

If you run a random Python script and wish to "activate" the virtualenv, you can simply instantiate an instance of this class and call .ensure() and .activate() to make the virtualenv active.

activate_path

bin_path

build(*python='/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/envs/latest/bin/python'*)
Build a virtualenv per tree conventions.

This returns the path of the created virtualenv.

call_setup(*directory*, *arguments*)
Calls setup.py in a directory.

create(*python='/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/envs/latest/bin/python'*)
Create a new, empty virtualenv.

Receives the path to virtualenv's virtualenv.py script (which will be called out to), the path to create the virtualenv in, and a handle to write output to.

ensure(*python='/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/envs/latest/bin/python'*)
Ensure the virtualenv is present and up to date.

If the virtualenv is up to date, this does nothing. Otherwise, it creates and populates the virtualenv as necessary.

This should be the main API used from this class as it is the highest-level.

get_exe_info()
Returns the version and file size of the python executable that was in use when this virutalenv was created.

install_pip_package(*package*)
Install a package via pip.

The supplied package is specified using a pip requirement specifier. e.g. 'foo' or 'foo==1.0'.

If the package is already installed, this is a no-op.

packages()

populate()
Populate the virtualenv.

The manifest file consists of colon-delimited fields. The first field specifies the action. The remaining fields are arguments to that action. The following actions are supported:

setup.py – Invoke setup.py for a package. Expects the arguments:

1. relative path directory containing setup.py.
2. argument(s) to setup.py. e.g. "develop". Each program argument is delimited by a colon. Arguments with colons are not yet supported.

filename.pth – Adds the path given as argument to filename.pth under the virtualenv site packages directory.

optional – This denotes the action as optional. The requested action is attempted. If it fails, we issue a warning and go on. The initial "optional" field is stripped then the remaining line is processed like normal. e.g. "optional:setup.py:python/foo:built_ext:-i"

copy – Copies the given file in the virtualenv site packages directory.

packages.txt – Denotes that the specified path is a child manifest. It will be read and processed as if its contents were concatenated into the manifest being read.

objdir – Denotes a relative path in the object directory to add to the search path. e.g. "objdir:build" will add $topobjdir/build to the search path.

Note that the Python interpreter running this function should be the one from the virtualenv. If it is the system Python or if the environment is not configured properly, packages could be installed into the wrong place. This is how virtualenv's work.

python_path

up_to_date(*python='/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/envs/latest/bin/python'*)
Returns whether the virtualenv is present and up to date.

virtualenv_script_path
Path to virtualenv's own populator script.

write_exe_info(*python*)
Records the the version of the python executable that was in use when this virutalenv was created. We record this explicitly because on OS X our python path may end up being a different or modified executable.

mozbuild.virtualenv.**verify_python_version**(*log_handle*)
Ensure the current version of Python is sufficient.

22.2.23 Module contents

22.3 mozlint package

22.3.1 Subpackages

mozlint.formatters package

Submodules

mozlint.formatters.stylish module

class mozlint.formatters.stylish.**NullTerminal**
Bases: object

Replacement for *blessings.Terminal()* that does no formatting.

class **NullCallableString**
Bases: unicode

A dummy callable Unicode stolen from blessings

class mozlint.formatters.stylish.**StylishFormatter**(*disable_colors=None*)
Bases: object

Formatter based on the eslint default.

fmt = u' {c1}{lineno}{column} {c2}{level}{normal} {message} {c1}{rule}({linter}){normal}'

fmt_summary = u'{t.bold}{c}\u2716 {problem} ({error}, {warning}){t.normal}'

mozlint.formatters.treeherder module

class mozlint.formatters.treeherder.**TreeherderFormatter**
Bases: object

Formatter for treeherder friendly output.

This formatter looks ugly, but prints output such that treeherder is able to highlight the errors and warnings. This is a stop-gap until bug 1276486 is fixed.

fmt = u'TEST-UNEXPECTED-{level} | {path}:{lineno}{column} | {message} ({rule})'

Module contents

class mozlint.formatters.**JSONFormatter**
Bases: object

mozlint.formatters.**get**(*name*, ***fmtargs*)

22.3.2 Submodules

22.3.3 mozlint.cli module

class mozlint.cli.**MozlintParser**(***kwargs*)
Bases: argparse.ArgumentParser

arguments = [[[u'paths'], {u'default': None, u'nargs': u'*', u'help': u"Paths to file or directories to lint, like 'browser/c

class mozlint.cli.**VCFiles**
Bases: object

by_rev(*rev*)

by_workdir()

is_git

is_hg

vcs

mozlint.cli.**find_linters**(*linters=None*)

mozlint.cli.**run**(*paths*, *linters*, *fmt*, *rev*, *workdir*, ***lintargs*)

22.3.4 mozlint.errors module

exception mozlint.errors.**LintException**
Bases: exceptions.Exception

exception mozlint.errors.**LinterNotFound**(*path*)
Bases: *mozlint.errors.LintException*

exception mozlint.errors.**LinterParseError**(*path*, *message*)
Bases: *mozlint.errors.LintException*

exception mozlint.errors.**LintersNotConfigured**
Bases: *mozlint.errors.LintException*

22.3.5 mozlint.parser module

class mozlint.parser.**Parser**
Bases: object

Reads and validates *.lint* files.

parse(*path*)
Read a linter and return its LINTER definition.

Parameters **path** – Path to the linter.

Returns Linter definition (dict)

Raises LinterNotFound, LinterParseError

required_attributes = ('name', 'description', 'type', 'payload')

22.3.6 mozlint.pathutils module

class mozlint.pathutils.**FilterPath**(*path*, *exclude=None*)
Bases: object

Helper class to make comparing and matching file paths easier.

contains(*other*)
Return True if other is a subdirectory of self or equals self.

exists

finder

isdir

isfile

join(**args*)

match(*patterns*)

mozlint.pathutils.**filterpaths**(*paths*, *include=None*, *exclude=None*)
Filters a list of paths.

Given a list of paths, and a list of include and exclude directives, return the set of paths that should be linted.

Parameters

- **paths** – A starting list of paths to possibly lint.
- **include** – A list of include directives. May contain glob patterns.
- **exclude** – A list of exclude directives. May contain glob patterns.

Returns A tuple containing a list of file paths to lint, and a list of file paths that should be excluded (but that the algorithm was unable to apply).

22.3.7 mozlint.result module

class mozlint.result.**ResultContainer**(*linter*, *path*, *message*, *lineno*, *column=None*, *hint=None*, *source=None*, *level=None*, *rule=None*, *lineoffset=None*)
Bases: object

Represents a single lint error and its related metadata.

Parameters

- **linter** – name of the linter that flagged this error
- **path** – path to the file containing the error
- **message** – text describing the error
- **lineno** – line number that contains the error
- **column** – column containing the error
- **level** – severity of the error, either 'warning' or 'error' (default 'error')
- **hint** – suggestion for fixing the error (optional)
- **source** – source code context of the error (optional)
- **rule** – name of the rule that was violated (optional)

- **lineoffset** – denotes an error spans multiple lines, of the form (<lineno offset>, <num lines>) (optional)

column

hint

level

lineno

lineoffset

linter

message

path

rule

source

class mozlint.result.**ResultEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Bases: json.encoder.JSONEncoder

Class for encoding :class:'~result.ResultContainer's to json.

Usage:

json.dumps(results, cls=ResultEncoder)

default(*o*)

mozlint.result.**from_linter**(*lintobj*, ***kwargs*)

Create a ResultContainer from a LINTER definition.

Convenience method that pulls defaults from a LINTER definition and forwards them.

Parameters

- **lintobj** – LINTER obj as defined in a .lint file
- **kwargs** – same as ResultContainer

Returns ResultContainer object

22.3.8 mozlint.roller module

class mozlint.roller.**LintRoller**(***lintargs*)

Bases: object

Registers and runs linters.

Parameters **lintargs** – Arguments to pass to the underlying linter(s).

read(*paths*)

Parse one or more linters and add them to the registry.

Parameters **paths** – A path or iterable of paths to linter definitions.

roll(*paths*, *num_procs=None*)

Run all of the registered linters against the specified file paths.

Parameters

- **paths** – An iterable of files and/or directories to lint.
- **num_procs** – The number of processes to use. Default: cpu count

Returns A dictionary with file names as the key, and a list of :class:‘~result.ResultContainer‘s as the value.

22.3.9 mozlint.types module

class mozlint.types.**BaseType**
Bases: object

Abstract base class for all types of linters.

batch = False

class mozlint.types.**ExternalType**
Bases: *mozlint.types.BaseType*

Linter type that runs an external function.

The function is responsible for properly formatting the results into a list of ResultContainer objects.

batch = True

class mozlint.types.**LineType**
Bases: *mozlint.types.BaseType*

Abstract base class for linter types that check each line individually.

Subclasses of this linter type will read each file and check the provided payload against each line one by one.

condition (*payload*, *line*)

class mozlint.types.**RegexType**
Bases: *mozlint.types.LineType*

Linter type that checks whether a regex match is found.

condition (*payload*, *line*)

class mozlint.types.**StringType**
Bases: *mozlint.types.LineType*

Linter type that checks whether a substring is found.

condition (*payload*, *line*)

mozlint.types.**supported_types = {u’regex’: <mozlint.types.RegexType object at 0x7f861e34c6d0>, u’string’: <mozlint**
Mapping of type string to an associated instance.

22.3.10 Module contents

22.4 mozpack package

22.4.1 Subpackages

mozpack.chrome package

Submodules

mozpack.chrome.flags module

class mozpack.chrome.flags.**Flag**(*name*)
Bases: object

Class for flags in manifest entries in the form: "flag" (same as "flag=true") "flag=yes|true|1" "flag=no|false|0"

add_definition(*definition*)
Add a flag value definition. Replaces any previously set value.

matches(*value*)
Return whether the flag value matches the given value. The values are canonicalized for comparison.

class mozpack.chrome.flags.**Flags**(**flags*)
Bases: collections.OrderedDict

Class to handle a set of flags definitions given on a single manifest entry.

FLAGS = {'process': <class 'mozpack.chrome.flags.StringFlag'>, 'appversion': <class 'mozpack.chrome.flags.VersionFlag

RE = <_sre.SRE_Pattern object>

match(***filter*)

Return whether the set of flags match the set of given filters.

flags = Flags('contentaccessible=yes', 'appversion>=3.5', 'application=foo')

flags.match(application='foo') returns True flags.match(application='foo', appversion='3.5') returns True flags.match(application='foo', appversion='3.0') returns False

class mozpack.chrome.flags.**StringFlag**(*name*)
Bases: object

Class for string flags in manifest entries in the form: "flag=string" "flag!=string"

add_definition(*definition*)
Add a string flag definition.

matches(*value*)
Return whether one of the string flag definitions matches the given value. For example,

flag = StringFlag('foo') flag.add_definition('foo!=bar') flag.matches('bar') returns False flag.matches('qux') returns True flag = StringFlag('foo') flag.add_definition('foo=bar') flag.add_definition('foo=baz') flag.matches('bar') returns True flag.matches('baz') returns True flag.matches('qux') returns False

class mozpack.chrome.flags.**VersionFlag**(*name*)
Bases: object

Class for version flags in manifest entries in the form: “flag=version” “flag<=version” “flag<version” “flag>=version” “flag>version”

add_definition(*definition*)
Add a version flag definition.

matches(*value*)
Return whether one of the version flag definitions matches the given value. For example,

flag = VersionFlag(‘foo’) flag.add_definition(‘foo>=1.0’) flag.matches(‘1.0’) returns True flag.matches(‘1.1’) returns True flag.matches(‘0.9’) returns False flag = VersionFlag(‘foo’) flag.add_definition(‘foo>=1.0’) flag.add_definition(‘foo<0.5’) flag.matches(‘0.4’) returns True flag.matches(‘1.0’) returns True flag.matches(‘0.6’) returns False

mozpack.chrome.manifest module

class mozpack.chrome.manifest.**Manifest**(*base*, *relpath*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestEntryWithRelPath*

Class for ‘manifest’ entries. manifest some/path/to/another.manifest

type = ‘manifest’

class mozpack.chrome.manifest.**ManifestBinaryComponent**(*base*, *relpath*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestEntryWithRelPath*

Class for ‘binary-component’ entries. binary-component some/path/to/a/component.dll

type = ‘binary-component’

class mozpack.chrome.manifest.**ManifestCategory**(*base*, *category*, *name*, *value*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestEntry*

Class for ‘category’ entries. category command-line-handler m-browser @mozilla.org/browser/clh;

type = ‘category’

class mozpack.chrome.manifest.**ManifestChrome**(*base*, *name*, *relpath*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestEntryWithRelPath*

Abstract class for chrome entries.

location

class mozpack.chrome.manifest.**ManifestComponent**(*base*, *cid*, *file*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestEntryWithRelPath*

Class for ‘component’ entries. component {b2bba4df-057d-41ea-b6b1-94a10a8ede68} foo.js

type = ‘component’

class mozpack.chrome.manifest.**ManifestContent**(*base*, *name*, *relpath*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestChrome*

Class for ‘content’ entries. content global content/global/

allowed_flags = [’application’, ‘platformversion’, ‘os’, ‘osversion’, ‘abi’, ‘xpcnativewrappers’, ‘tablet’, ‘process’, ‘cor

type = ‘content’

class mozpack.chrome.manifest.**ManifestContract**(*base*, *contractID*, *cid*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestEntry*

Class for ‘contract’ entries. contract @mozilla.org/foo;1 {b2bba4df-057d-41ea-b6b1-94a10a8ede68}

type = 'contract'

class mozpack.chrome.manifest.**ManifestEntry**(*base*, **flags*)
Bases: object

Base class for all manifest entry types. Subclasses may define the following class or member variables:

•localized: indicates whether the manifest entry is used for localized data.

•type: the manifest entry type (e.g. 'content' in 'content global content/global/')

•allowed_flags: a set of flags allowed to be defined for the given manifest entry type.

A manifest entry is attached to a base path, defining where the manifest entry is bound to, and that is used to find relative paths defined in entries.

allowed_flags = ['application', 'platformversion', 'os', 'osversion', 'abi', 'xpcnativewrappers', 'tablet', 'process']

localized = False

move(*base*)
Return a new manifest entry with a different base path.

rebase(*base*)
Return a new manifest entry with all relative paths defined in the entry relative to a new base directory. The base class doesn't define relative paths, so it is equivalent to move().

serialize(**args*)
Serialize the manifest entry.

type = None

class mozpack.chrome.manifest.**ManifestEntryWithRelPath**(*base*, *relpath*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestEntry*

Abstract manifest entry type with a relative path definition.

path

rebase(*base*)
Return a new manifest entry with all relative paths defined in the entry relative to a new base directory.

class mozpack.chrome.manifest.**ManifestInterfaces**(*base*, *relpath*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestEntryWithRelPath*

Class for 'interfaces' entries. interfaces foo.xpt

type = 'interfaces'

class mozpack.chrome.manifest.**ManifestLocale**(*base*, *name*, *id*, *relpath*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestMultiContent*

Class for 'locale' entries. locale global en-US content/en-US/ locale global fr content/fr/

localized = True

type = 'locale'

class mozpack.chrome.manifest.**ManifestMultiContent**(*base*, *name*, *id*, *relpath*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestChrome*

Abstract class for chrome entries with multiple definitions. Used for locale and skin entries.

type = None

class mozpack.chrome.manifest.**ManifestOverlay**(*base*, *overloaded*, *overload*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestOverload*

Class for 'overlay' entries. overlay chrome://global/content/viewSource.xul chrome://browser/content/viewSourceOverlay.xul

type = 'overlay'

class mozpack.chrome.manifest.**ManifestOverload**(*base*, *overloaded*, *overload*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestEntry*

Abstract class for chrome entries defining some kind of overloading. Used for overlay, override or style entries.

localized

type = None

class mozpack.chrome.manifest.**ManifestOverride**(*base*, *overloaded*, *overload*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestOverload*

Class for 'override' entries. override chrome://global/locale/netError.dtd chrome://browser/locale/netError.dtd

type = 'override'

class mozpack.chrome.manifest.**ManifestResource**(*base*, *name*, *target*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestEntry*

Class for 'resource' entries. resource gre-resources toolkit/res/ resource services-sync resource://gre/modules/services-sync/

The target may be a relative path or a resource or chrome url.

rebase(*base*)

type = 'resource'

class mozpack.chrome.manifest.**ManifestSkin**(*base*, *name*, *id*, *relpath*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestMultiContent*

Class for 'skin' entries. skin global classic/1.0 content/skin/classic/

type = 'skin'

class mozpack.chrome.manifest.**ManifestStyle**(*base*, *overloaded*, *overload*, **flags*)
Bases: *mozpack.chrome.manifest.ManifestOverload*

Class for 'style' entries. style chrome://global/content/customizeToolbar.xul chrome://browser/skin/

type = 'style'

mozpack.chrome.manifest.**is_manifest**(*path*)
Return whether the given path is that of a manifest file.

mozpack.chrome.manifest.**parse_manifest**(*root*, *path*, *fileobj=None*)
Parse a manifest file.

mozpack.chrome.manifest.**parse_manifest_line**(*base*, *line*)
Parse a line from a manifest file with the given base directory and return the corresponding ManifestEntry instance.

Module contents

mozpack.packager package

Submodules

mozpack.packager.formats module

class mozpack.packager.formats.**FlatFormatter**(*copier*)
Bases: *mozpack.packager.formats.PiecemealFormatter*

Formatter for the flat package format.

class mozpack.packager.formats.**FlatSubFormatter**(*copier*)
Bases: object

Sub-formatter for the flat package format.

add(*path*, *content*)

add_interfaces(*path*, *content*)

add_manifest(*entry*)

contains(*path*)

class mozpack.packager.formats.**JarFormatter**(*copier*, *compress=True*, *optimize=True*)
Bases: *mozpack.packager.formats.PiecemealFormatter*

Formatter for the jar package format. Assumes manifest entries related to chrome are registered before the chrome data files are added. Also assumes manifest entries for resources are registered after chrome manifest entries.

class mozpack.packager.formats.**JarSubFormatter**(*copier*, *compress=True*, *optimize=True*)
Bases: *mozpack.packager.formats.PiecemealFormatter*

Sub-formatter for the jar package format. It is a PiecemealFormatter that dispatches between further sub-formatter for each of the jar files it dispatches the chrome data to, and a FlatSubFormatter for the non-chrome files.

add_manifest(*entry*)

class mozpack.packager.formats.**OmniJarFormatter**(*copier*, *omnijar_name*, *compress=True*, *optimize=True*, *non_resources=()*)
Bases: *mozpack.packager.formats.JarFormatter*

Formatter for the omnijar package format.

class mozpack.packager.formats.**OmniJarSubFormatter**(*copier*, *omnijar_name*, *compress=True*, *optimize=True*, *non_resources=()*)
Bases: *mozpack.packager.formats.PiecemealFormatter*

Sub-formatter for the omnijar package format. It is a PiecemealFormatter that dispatches between a FlatSub-Formatter for the resources data and another FlatSubFormatter for the other files.

add_manifest(*entry*)

is_resource(*path*)
Return whether the given path corresponds to a resource to be put in an omnijar archive.

class mozpack.packager.formats.**PiecemealFormatter**(*copier*)
Bases: object

Generic formatter that dispatches across different sub-formatters according to paths.

add(*path*, *content*)

add_base(*base*, *addon=False*)

add_interfaces(*path*, *content*)

add_manifest(*entry*)

contains(*path*)

mozpack.packager.formats.**STARTUP_CACHE_PATHS = ['jsloader', 'jssubloader']**
Formatters are classes receiving packaging instructions and creating the appropriate package layout.

There are three distinct formatters, each handling one of the different chrome formats:

- flat: essentially, copies files from the source with the same file system layout. Manifests entries are grouped in a single manifest per directory, as well as XPT interfaces.
- jar: chrome content is packaged in jar files.
- omni: chrome content, modules, non-binary components, and many other elements are packaged in an omnijar file for each base directory.

The base interface provides the following methods:

- **add_base(path [, addon])** Register a base directory for an application or GRE, or an addon. Base directories usually contain a root manifest (manifests not included in any other manifest) named chrome.manifest. The optional addon argument tells whether the base directory is that of a packed addon (True), unpacked addon ('unpacked') or otherwise (False).
- **add(path, content)** Add the given content (BaseFile instance) at the given virtual path
- **add_interfaces(path, content)** Add the given content (BaseFile instance) and link it to other interfaces in the parent directory of the given virtual path.
- **add_manifest(entry)** Add a ManifestEntry.
- **contains(path)** Returns whether the given virtual path is known of the formatter.

The virtual paths mentioned above are paths as they would be with a flat chrome.

Formatters all take a FileCopier instance they will fill with the packaged data.

mozpack.packager.l10n module

class mozpack.packager.l10n.**LocaleManifestFinder**(*finder*)
Bases: object

mozpack.packager.l10n.**repack**(*source*, *l10n*, *extra_l10n={}*, *non_resources=[]*, *non_chrome=set([])*)
Replace localized data from the *source* directory with localized data from *l10n* and *extra_l10n*.

The *source* argument points to a directory containing a packaged application (in omnijar, jar or flat form). The *l10n* argument points to a directory containing the main localized data (usually in the form of a language pack addon) to use to replace in the packaged application. The *extra_l10n* argument contains a dict associating relative paths in the source to separate directories containing localized data for them. This can be used to point at different language pack addons for different parts of the package application. The *non_resources* argument gives a list of relative paths in the source that should not be added in an omnijar in case the packaged application

is in that format. The *non_chrome* argument gives a list of file/directory patterns for localized files that are not listed in a chrome.manifest.

mozpack.packager.unpack module

class mozpack.packager.unpack.**UnpackFinder**(**args*, ***kargs*)
Bases: *mozpack.files.FileFinder*

Special FileFinder that treats the source package directory as if it were in the flat chrome format, whatever chrome format it actually is in.

This means that for example, paths like chrome/browser/content/... match files under jar:chrome/browser.jar!/content/... in case of jar chrome format.

find(*path*)

mozpack.packager.unpack.**unpack**(*source*)
Transform a jar chrome or omnijar packaged directory into a flat package.

mozpack.packager.unpack.**unpack_to_registry**(*source*, *registry*)
Transform a jar chrome or omnijar packaged directory into a flat package.

The given registry is filled with the flat package.

Module contents

class mozpack.packager.**CallDeque**
Bases: collections.deque

Queue of function calls to make.

append(*function*, **args*)

execute()

class mozpack.packager.**Component**(*name*, *destdir=''*)
Bases: object

Class that represents a component in a package manifest.

KEY_VALUE_RE = <_sre.SRE_Pattern object>

destdir

static from_string(*string*)
Create a component from a string.

name

class mozpack.packager.**PackageManifestParser**(*sink*)
Bases: object

Class for parsing of a package manifest, after preprocessing.

A package manifest is a list of file paths, with some syntaxic sugar: [] designates a toplevel component. Example: [xpcom] - in front of a file specifies it to be removed * wildcard support ** expands to all files and zero or more directories ; file comment

The parser takes input from the preprocessor line by line, and pushes parsed information to a sink object.

The add and remove methods of the sink object are called with the current Component instance and a path.

handle_line(*str*)
Handle a line of input and push the parsed information to the sink object.

class mozpack.packager.**PreprocessorOutputWrapper**(*preprocessor*, *parser*)
Bases: object

File-like helper to handle the preprocessor output and send it to a parser. The parser's handle_line method is called in the relevant errors.context.

write(*str*)

class mozpack.packager.**SimpleManifestSink**(*finder*, *formatter*)
Bases: object

Parser sink for "simple" package manifests. Simple package manifests use the format described in the PackageManifestParser documentation, but don't support file removals, and require manifests, interfaces and chrome data to be explicitely listed. Entries starting with bin/ are searched under bin/ in the FileFinder, but are packaged without the bin/ prefix.

add(*component*, *pattern*)
Add files with the given pattern in the given component.

close(*auto_root_manifest=True*)
Add possibly missing bits and push all instructions to the formatter.

static normalize_path(*path*)
Remove any bin/ prefix.

remove(*component*, *pattern*)
Remove files with the given pattern in the given component.

class mozpack.packager.**SimplePackager**(*formatter*)
Bases: object

Helper used to translate and buffer instructions from the SimpleManifestSink to a formatter. Formatters expect some information to be given first that the simple manifest contents can't guarantee before the end of the input.

UNPACK_ADDON_RE = <_sre.SRE_Pattern object at 0x1b6fb00>

add(*path*, *file*)
Add the given BaseFile instance with the given path.

close()
Push all instructions to the formatter.

get_bases(*addons=True*)
Return all paths under which root manifests have been found. Root manifests are manifests that are included in no other manifest. *addons* indicates whether to include addon bases as well.

mozpack.packager.**preprocess**(*input*, *parser*, *defines={}*)
Preprocess the file-like input with the given defines, and send the preprocessed output line by line to the given parser.

mozpack.packager.**preprocess_manifest**(*sink*, *manifest*, *defines={}*)
Preprocess the given file-like manifest with the given defines, and push the parsed information to a sink. See PackageManifestParser documentation for more details on the sink.

mozpack.test package

Submodules

mozpack.test.test_archive module

class mozpack.test.test_archive.**TestArchive**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_create_tar_basic()

test_create_tar_bz2_basic()

test_create_tar_gz_basic()

test_dirs_refused()

test_executable_preserved()

test_setuid_setgid_refused()

test_tar_gz_name()

mozpack.test.test_archive.**file_hash**(*path*)

mozpack.test.test_chrome_flags module

class mozpack.test.test_chrome_flags.**TestFlag**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_flag()

test_string_flag()

test_version_flag()

class mozpack.test.test_chrome_flags.**TestFlags**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

test_flags_match()

test_flags_match_different()

test_flags_match_unset()

test_flags_match_version()

test_flags_str()

mozpack.test.test_chrome_manifest module

class mozpack.test.test_chrome_manifest.**TestManifest**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_manifest_rebase()

test_parse_manifest()

class mozpack.test.test_chrome_manifest.**TestManifestErrors**(*methodName='runTest'*)
Bases: *mozpack.test.test_errors.TestErrors*, unittest.case.TestCase

test_parse_manifest_errors()

mozpack.test.test_copier module

class mozpack.test.test_copier.**BaseTestFileRegistry**
Bases: *mozpack.test.test_files.MatchTestTemplate*

add(*path*)

do_check(*pattern*, *result*)

do_test_file_registry(*registry*)

do_test_registry_paths(*registry*)

class mozpack.test.test_copier.**TestFileCopier**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

all_dirs(*base*)

all_files(*base*)

test_file_copier()

test_no_remove()

test_no_remove_empty_directories()

test_optional_exists_creates_unneeded_directory()
Demonstrate that a directory not strictly required, but specified as the path to an optional file, will be unnecessarily created.

This behaviour is wrong; fixing it is tracked by Bug 972432; and this test exists to guard against unexpected changes in behaviour.

test_permissions()
Ensure files without write permission can be deleted.

test_remove_unaccounted_directory_symlinks()
Directory symlinks in destination that are not in the way are deleted according to remove_unaccounted and remove_all_directory_symlinks.

test_remove_unaccounted_file_registry()
Test FileCopier.copy(remove_unaccounted=FileRegistry())

test_symlink_directory_replaced()
Directory symlinks in destination are replaced if they need to be real directories.

class mozpack.test.test_copier.**TestFileRegistry**(*methodName='runTest'*)
Bases: *mozpack.test.test_copier.BaseTestFileRegistry*, unittest.case.TestCase

test_file_registry()

test_partial_paths()

test_registry_paths()

test_required_directories()

class mozpack.test.test_copier.**TestFileRegistrySubtree**(*methodName='runTest'*)
Bases: *mozpack.test.test_copier.BaseTestFileRegistry*, unittest.case.TestCase

create_registry()

test_file_registry_subtree()

test_file_registry_subtree_base()

test_registry_paths_subtree()

class mozpack.test.test_copier.**TestJarrer**(*methodName='runTest'*)
Bases: unittest.case.TestCase

check_jar(*dest*, *copier*)

test_jarrer()

test_jarrer_compress()

mozpack.test.test_errors module

class mozpack.test.test_errors.**TestErrors**
Bases: object

get_output()

setUp()

tearDown()

class mozpack.test.test_errors.**TestErrorsImpl**(*methodName='runTest'*)
Bases: *mozpack.test.test_errors.TestErrors*, unittest.case.TestCase

test_error_loop()

test_errors_context()

test_ignore_errors()

test_multiple_errors()

test_no_error()

test_plain_error()

test_simple_error()

mozpack.test.test_files module

class mozpack.test.test_files.**DestNoWrite**(*path*)
Bases: *mozpack.files.Dest*

write(*data*)

class mozpack.test.test_files.**MatchTestTemplate**
Bases: object

do_finder_test(*finder*)

do_match_test()

prepare_match_test(*with_dotfiles=False*)

class mozpack.test.test_files.**MockDest**
Bases: _io.BytesIO, *mozpack.files.Dest*

close()

exists()

read(*length=-1*)

write(*data*)

class mozpack.test.test_files.**TestAbsoluteSymlinkFile**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_absolute_relative()

test_noop()

test_replace_file_with_symlink()

test_replace_symlink()

test_symlink_file()

class mozpack.test.test_files.**TestComposedFinder**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.MatchTestTemplate*, *mozpack.test.test_files.TestWithTmpDir*

add(*path*, *content=None*)

do_check(*pattern*, *result*)

test_composed_finder()

class mozpack.test.test_files.**TestDeflatedFile**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_deflated_file()
Check that DeflatedFile.copy yields the proper content in the destination file in all situations that trigger different code paths (see TestFile.test_file)

test_deflated_file_no_write()
Test various conditions where DeflatedFile.copy is expected not to write in the destination file.

test_deflated_file_open()
Test whether DeflatedFile.open returns an appropriately reset file object.

class mozpack.test.test_files.**TestDest**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_dest()

class mozpack.test.test_files.**TestExistingFile**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_optional_existing_dest()

test_optional_missing_dest()

test_required_existing_dest()

test_required_missing_dest()

class mozpack.test.test_files.**TestFile**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_file()
Check that File.copy yields the proper content in the destination file in all situations that trigger different code paths: - different content - different content of the same size - same content - long content

test_file_dest()
Similar to test_file, but for a destination object instead of a destination file. This ensures the destination object is being used properly by File.copy, ensuring that other subclasses of Dest will work.

test_file_no_write()
Test various conditions where File.copy is expected not to write in the destination file.

test_file_open()
Test whether File.open returns an appropriately reset file object.

class mozpack.test.test_files.**TestFileFinder**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.MatchTestTemplate*, *mozpack.test.test_files.TestWithTmpDir*

add(*path*)

do_check(*pattern*, *result*)

test_dotfiles()
Finder can find files beginning with . is configured.

test_dotfiles_plus_ignore()

test_file_finder()

test_get()

test_ignored_dirs()
Ignored directories should not have results returned.

test_ignored_files()
Ignored files should not have results returned.

test_ignored_patterns()
Ignore entries with patterns should be honored.

class mozpack.test.test_files.**TestGeneratedFile**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_generated_file()
Check that GeneratedFile.copy yields the proper content in the destination file in all situations that trigger different code paths (see TestFile.test_file)

test_generated_file_no_write()
Test various conditions where GeneratedFile.copy is expected not to write in the destination file.

test_generated_file_open()
Test whether GeneratedFile.open returns an appropriately reset file object.

class mozpack.test.test_files.**TestJarFinder**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.MatchTestTemplate*, *mozpack.test.test_files.TestWithTmpDir*

add(*path*)

do_check(*pattern*, *result*)

test_jar_finder()

class mozpack.test.test_files.**TestManifestFile**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_manifest_file()

class mozpack.test.test_files.**TestMercurialNativeRevisionFinder**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestMercurialRevisionFinder*

class mozpack.test.test_files.**TestMercurialRevisionFinder**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.MatchTestTemplate*, *mozpack.test.test_files.TestWithTmpDir*

add(*path*)

do_check(*pattern*, *result*)

setUp()

test_default_revision()

test_old_revision()

test_recognize_repo_paths()

class mozpack.test.test_files.**TestMinifiedJavaScript**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

orig_lines = ['// Comment line', 'let foo = "bar";', 'var bar = true;', '', '// Another comment']

test_minified_javascript()

test_minified_verify_failure()

test_minified_verify_success()

class mozpack.test.test_files.**TestMinifiedProperties**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_minified_properties()

class mozpack.test.test_files.**TestPreprocessedFile**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_preprocess()
Test that copying the file invokes the preprocessor

test_preprocess_file_dependencies()
Test that the preprocess runs if the dependencies of the source change

test_preprocess_file_no_write()
Test various conditions where PreprocessedFile.copy is expected not to write in the destination file.

test_replace_symlink()
Test that if the destination exists, and is a symlink, the target of the symlink is not overwritten by the preprocessor output.

class mozpack.test.test_files.**TestWithTmpDir**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()

tearDown()

tmppath(*relpath*)

class mozpack.test.test_files.**TestXPTFile**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_xpt_file()

mozpack.test.test_files.**do_check**(*test*, *finder*, *pattern*, *result*)

mozpack.test.test_files.**read_interfaces**(*file*)

mozpack.test.test_manifests module

class mozpack.test.test_manifests.**TestInstallManifest**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

test_adds()

test_construct()

test_copier_application()

test_malformed()

test_or()

test_pattern_expansion()

test_populate_registry()

test_preprocessor()

test_preprocessor_dependencies()

test_serialization()

mozpack.test.test_mozjar module

class mozpack.test.test_mozjar.**TestDeflater**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_deflater_compress()

test_deflater_compress_no_gain()

test_deflater_no_compress()

wrap(*data*)

class mozpack.test.test_mozjar.**TestDeflaterMemoryView**(*methodName='runTest'*)
Bases: *mozpack.test.test_mozjar.TestDeflater*

wrap(*data*)

class mozpack.test.test_mozjar.**TestJar**(*methodName='runTest'*)
Bases: unittest.case.TestCase

optimize = False

test_add_from_finder()

test_jar()

test_rejar()

class mozpack.test.test_mozjar.**TestJarLog**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_jarlog()

class mozpack.test.test_mozjar.**TestJarStruct**(*methodName='runTest'*)
Bases: unittest.case.TestCase

class Foo(*data=None*)
Bases: *mozpack.mozjar.JarStruct*

MAGIC = 16909060

STRUCT = OrderedDict([('foo', 'uint32'), ('bar', 'uint16'), ('qux', 'uint16'), ('length', 'uint16'), ('length2', 'uint16'),

TestJarStruct.**do_test_read_jar_struct**(*data*)

TestJarStruct.**test_jar_struct**()

TestJarStruct.**test_read_jar_struct**()

TestJarStruct.**test_read_jar_struct_memoryview**()

class mozpack.test.test_mozjar.**TestOptimizeJar**(*methodName='runTest'*)
Bases: *mozpack.test.test_mozjar.TestJar*

optimize = True

class mozpack.test.test_mozjar.**TestPreload**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_preload()

mozpack.test.test_packager module

class mozpack.test.test_packager.**MockFinder**(*files*)
Bases: object

find(*path*)

class mozpack.test.test_packager.**MockFormatter**
Bases: object

add(**args*)

add_base(**args*)

add_interfaces(**args*)

add_manifest(**args*)

class mozpack.test.test_packager.**TestCallDeque**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_call_deque()

class mozpack.test.test_packager.**TestComponent**(*methodName='runTest'*)
Bases: unittest.case.TestCase

do_from_string(*string*, *name*, *destdir=''*)

do_split(*string*, *name*, *options*)

do_split_error(*string*)

test_component_from_string()

test_component_split_component_and_options()

test_component_split_component_and_options_errors()

class mozpack.test.test_packager.**TestPreprocessManifest**(*methodName='runTest'*)
Bases: unittest.case.TestCase

EXPECTED_LOG = [(('/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs/mar

MANIFEST_PATH = '/home/docs/checkouts/readthedocs.org/user_builds/gfritzsche-demo/checkouts/latest/tools/docs/mani

setUp()

test_preprocess_manifest()

test_preprocess_manifest_defines()

test_preprocess_manifest_missing_define()

class mozpack.test.test_packager.**TestSimpleManifestSink**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_simple_manifest_parser()

class mozpack.test.test_packager.**TestSimplePackager**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_simple_packager()

test_simple_packager_manifest_consistency()

mozpack.test.test_packager_formats module

class mozpack.test.test_packager_formats.**MockDest**
Bases: *mozpack.test.test_files.MockDest*

exists()

class mozpack.test.test_packager_formats.**TestFormatters**(*methodName='runTest'*)
Bases: unittest.case.TestCase

do_test_contents(*formatter*, *contents*)

maxDiff = None

test_bases()

test_flat_formatter()

test_flat_formatter_with_base()

test_jar_formatter()

test_jar_formatter_with_base()

test_omnijar_formatter()

test_omnijar_formatter_with_base()

test_omnijar_is_resource()

mozpack.test.test_packager_formats.**fill_formatter**(*formatter*, *contents*)

mozpack.test.test_packager_formats.**get_contents**(*registry*, *read_all=False*)

mozpack.test.test_packager_formats.**result_with_base**(*results*)

mozpack.test.test_packager_l10n module

class mozpack.test.test_packager_l10n.**TestL10NRepack**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_l10n_repack()

mozpack.test.test_packager_unpack module

class mozpack.test.test_packager_unpack.**TestUnpack**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

maxDiff = None

classmethod setUpClass()

test_flat_unpack()

test_jar_unpack()

test_omnijar_unpack()

mozpack.test.test_path module

class mozpack.test.test_path.**TestPath**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_basedir()

test_basename()

test_commonprefix()

test_dirname()

test_join()

test_match()

test_normpath()

test_rebase()

test_relpath()

test_split()

test_splitext()

mozpack.test.test_unify module

class mozpack.test.test_unify.**TestUnified**(*methodName='runTest'*)
Bases: *mozpack.test.test_files.TestWithTmpDir*

create_both(*path*, *content*)

create_one(*which*, *path*, *content*)

class mozpack.test.test_unify.**TestUnifiedBuildFinder**(*methodName='runTest'*)
Bases: *mozpack.test.test_unify.TestUnified*

test_unified_build_finder()

class mozpack.test.test_unify.**TestUnifiedFinder**(*methodName='runTest'*)
Bases: *mozpack.test.test_unify.TestUnified*

test_unified_finder()

Module contents

22.4.2 Submodules

22.4.3 mozpack.archive module

mozpack.archive.**create_tar_bz2_from_files**(*fp*, *files*, *compresslevel=9*)
Create a tar.bz2 file deterministically from files.

This is a glorified wrapper around `create_tar_from_files` that adds bzip2 compression.

This function is similar to `create_tar_gzip_from_files()`.

mozpack.archive.**create_tar_from_files**(*fp*, *files*)
Create a tar file deterministically.

Receives a dict mapping names of files in the archive to local filesystem paths.

The files will be archived and written to the passed file handle opened for writing.

Only regular files can be written.

FUTURE accept mozpack.files classes for writing FUTURE accept a filename argument (or create APIs to write files)

mozpack.archive.**create_tar_gz_from_files**(*fp*, *files*, *filename=None*, *compresslevel=9*)
Create a tar.gz file deterministically from files.

This is a glorified wrapper around `create_tar_from_files` that adds gzip compression.

The passed file handle should be opened for writing in binary mode. When the function returns, all data has been written to the handle.

22.4.4 mozpack.copier module

class mozpack.copier.**FileCopier**
Bases: *mozpack.copier.FileRegistry*

FileRegistry with the ability to copy the registered files to a separate directory.

copy(*destination*, *skip_if_older=True*, *remove_unaccounted=True*, *remove_all_directory_symlinks=True*, *remove_empty_directories=True*)
Copy all registered files to the given destination path. The given destination can be an existing directory, or not exist at all. It can't be e.g. a file. The copy process acts a bit like rsync: files are not copied when they don't need to (see mozpack.files for details on file.copy).

By default, files in the destination directory that aren't registered are removed and empty directories are deleted. In addition, all directory symlinks in the destination directory are deleted: this is a conservative approach to ensure that we never accidently write files into a directory that is not the destination directory. In the worst case, we might have a directory symlink in the object directory to the source directory.

To disable removing of unregistered files, pass remove_unaccounted=False. To disable removing empty directories, pass remove_empty_directories=False. In rare cases, you might want to maintain directory symlinks in the destination directory (at least those that are not required to be regular directories): pass remove_all_directory_symlinks=False. Exercise caution with this flag: you almost certainly do not want to preserve directory symlinks.

Returns a FileCopyResult that details what changed.

class mozpack.copier.**FileCopyResult**
Bases: object

Represents results of a FileCopier.copy operation.

existing_files_count

removed_directories_count

removed_files_count

updated_files_count

class mozpack.copier.**FileRegistry**
Bases: object

Generic container to keep track of a set of BaseFile instances. It preserves the order under which the files are added, but doesn't keep track of empty directories (directories are not stored at all). The paths associated with the BaseFile instances are relative to an unspecified (virtual) root directory.

registry = FileRegistry() registry.add('foo/bar', file_instance)

add(*path*, *content*)
Add a BaseFile instance to the container, under the given path.

contains(*pattern*)
Return whether the container contains paths matching the given pattern. See the mozpack.path.match documentation for a description of the handled patterns.

match(*pattern*)
Return the list of paths, stored in the container, matching the given pattern. See the mozpack.path.match documentation for a description of the handled patterns.

paths()
Return all paths stored in the container, in the order they were added.

remove(*pattern*)
Remove paths matching the given pattern from the container. See the mozpack.path.match documentation for a description of the handled patterns.

required_directories()
Return the set of directories required by the paths in the container, in no particular order. The returned directories are relative to an unspecified (virtual) root directory (and do not include said root directory).

class mozpack.copier.**FileRegistrySubtree**(*base*, *registry*)
Bases: object

A proxy class to give access to a subtree of an existing FileRegistry.

Note this doesn't implement the whole FileRegistry interface.

add(*path*, *content*)

contains(*pattern*)

match(*pattern*)

paths()

remove(*pattern*)

class mozpack.copier.**Jarrer**(*compress=True*, *optimize=True*)
Bases: *mozpack.copier.FileRegistry*, *mozpack.files.BaseFile*

FileRegistry with the ability to copy and pack the registered files as a jar file. Also acts as a BaseFile instance, to be copied with a FileCopier.

add(*path*, *content*, *compress=None*)

copy(*dest*, *skip_if_older=True*)
Pack all registered files in the given destination jar. The given destination jar may be a path to jar file, or a Dest instance for a jar file. If the destination jar file exists, its (compressed) contents are used instead of the registered BaseFile instances when appropriate.

open()

preload(*paths*)
Add the given set of paths to the list of preloaded files. See mozpack.mozjar.JarWriter documentation for details on jar preloading.

22.4.5 mozpack.dmg module

mozpack.dmg.**check_tools**(**tools*)
Check that each tool named in tools exists in SUBSTS and is executable.

mozpack.dmg.**chmod**(*dir*)
Set permissions of DMG contents correctly

mozpack.dmg.**create_dmg**(*source_directory*, *output_dmg*, *volume_name*, *extra_files*)
Create a DMG disk image at the path output_dmg from source_directory.

Use volume_name as the disk image volume name, and use extra_files as a list of tuples of (filename, relative path) to copy into the disk image.

mozpack.dmg.**create_dmg_from_staged**(*stagedir*, *output_dmg*, *tmpdir*, *volume_name*)
Given a prepared directory stagedir, produce a DMG at output_dmg.

mozpack.dmg.**mkdir**(*dir*)

mozpack.dmg.**rsync**(*source*, *dest*)
rsync the contents of directory source into directory dest

mozpack.dmg.**set_folder_icon**(*dir*)
Set HFS attributes of dir to use a custom icon

22.4.6 mozpack.errors module

exception mozpack.errors.**AccumulatedErrors**
Bases: exceptions.Exception

Exception type raised from errors.accumulate()

class mozpack.errors.**ErrorCollector**
Bases: object

Error handling/logging class. A global instance, errors, is provided for convenience.

Warnings, errors and fatal errors may be logged by calls to the following functions:

errors.warn(message) errors.error(message) errors.fatal(message)

Warnings only send the message on the logging output, while errors and fatal errors send the message and throw an ErrorMessage exception. The exception, however, may be deferred. See further below.

Errors may be ignored by calling: errors.ignore_errors()

After calling that function, only fatal errors throw an exception.

The warnings, errors or fatal errors messages may be augmented with context information when a context is provided. Context is defined by a pair (filename, linenumber), and may be set with errors.context() used as a context manager:

with errors.context(filename, linenumber): errors.warn(message)

Arbitrary nesting is supported, both for errors.context calls:

with errors.context(filename1, linenumber1): errors.warn(message) with errors.context(filename2, linenumber2):

errors.warn(message)

as well as for function calls:

def func(): errors.warn(message)

with errors.context(filename, linenumber): func()

Errors and fatal errors can have their exception thrown at a later time, allowing for several different errors to be reported at once before throwing. This is achieved with errors.accumulate() as a context manager:

with errors.accumulate():

if test1: errors.error(message1)

if test2: errors.error(message2)

In such cases, a single AccumulatedErrors exception is thrown, but doesn't contain information about the exceptions. The logged messages do.

ERROR = 2

FATAL = 3

WARN = 1

accumulate(**args*, ***kwds*)

context(**args*, ***kwds*)

count

error(*msg*)

fatal(*msg*)

get_context()

ignore_errors(*ignore=True*)

out = <open file '<stderr>', mode 'w'>

warn(*msg*)

exception mozpack.errors.**ErrorMessage**

Bases: exceptions.Exception

Exception type raised from errors.error() and errors.fatal()

22.4.7 mozpack.executables module

mozpack.executables.**elfhack**(*path*)
Execute the elfhack command on the given path.

mozpack.executables.**get_type**(*path*)
Check the signature of the give file and returns what kind of executable matches.

mozpack.executables.**is_executable**(*path*)
Return whether a given file path points to an executable or a library, where an executable or library is identified by:

- the file extension on OS/2 and WINNT
- the file signature on OS/X and ELF systems (GNU/Linux, Android, BSD, Solaris)

As this function is intended for use to choose between the ExecutableFile and File classes in FileFinder, and choosing ExecutableFile only matters on OS/2, OS/X, ELF and WINNT (in GCC build) systems, we don't bother detecting other kind of executables.

mozpack.executables.**may_elfhack**(*path*)
Return whether elfhack() should be called

mozpack.executables.**may_strip**(*path*)
Return whether strip() should be called

mozpack.executables.**strip**(*path*)
Execute the STRIP command with STRIP_FLAGS on the given path.

22.4.8 mozpack.files module

class mozpack.files.**AbsoluteSymlinkFile**(*path*)
Bases: *mozpack.files.File*

File class that is copied by symlinking (if available).

This class only works if the target path is absolute.

copy(*dest*, *skip_if_older=True*)

class mozpack.files.**BaseFile**
Bases: object

Base interface and helper for file copying. Derived class may implement their own copy function, or rely on BaseFile.copy using the open() member function and/or the path property.

static any_newer(*dest*, *inputs*)
Compares the modification time of dest to multiple input files, and returns whether any of the inputs is newer (has a later mtime) than dest.

copy(*dest*, *skip_if_older=True*)
Copy the BaseFile content to the destination given as a string or a Dest instance. Avoids replacing existing files if the BaseFile content matches that of the destination, or in case of plain files, if the destination is newer than the original file. This latter behaviour is disabled when skip_if_older is False. Returns whether a copy was actually performed (True) or not (False).

static is_older(*first*, *second*)
Compares the modification time of two files, and returns whether the first file is older than the second file.

mode
Return the file's unix mode, or None if it has no meaning.

open()
Return a file-like object allowing to read() the content of the associated file. This is meant to be overloaded in subclasses to return a custom file-like object.

read()

class mozpack.files.**BaseFinder**(*base*, *minify=False*, *minify_js=False*, *minify_js_verify_command=None*)
Bases: object

contains(*pattern*)
Return whether some files under the base directory match the given pattern. See the mozpack.path.match documentation for a description of the handled patterns.

find(*pattern*)
Yield path, BaseFile_instance pairs for all files under the base directory and its subdirectories that match the given pattern. See the mozpack.path.match documentation for a description of the handled patterns.

get(*path*)
Obtain a single file.

Where find is tailored towards matching multiple files, this method is used for retrieving a single file. Use this method when performance is critical.

Returns a BaseFile if at most one file exists or None otherwise.

class mozpack.files.**ComposedFinder**(*finders*)
Bases: *mozpack.files.BaseFinder*

Composes multiple File Finders in some sort of virtual file system.

A ComposedFinder is initialized from a dictionary associating paths to *Finder instances.

Note this could be optimized to be smarter than getting all the files in advance.

find(*pattern*)

class mozpack.files.**DeflatedFile**(*file*)
Bases: *mozpack.files.BaseFile*

File class for members of a jar archive. DeflatedFile.copy() effectively extracts the file from the jar archive.

open()

class mozpack.files.**Dest**(*path*)
Bases: object

Helper interface for BaseFile.copy. The interface works as follows: - read() and write() can be used to sequentially read/write from the

underlying file.

- a call to read() after a write() will re-open the underlying file and read from it.
- a call to write() after a read() will re-open the underlying file, emptying it, and write to it.

close()

exists()

name

read(*length=-1*)

write(*data*)

class mozpack.files.**ExecutableFile**(*path*)
Bases: *mozpack.files.File*

File class for executable and library files on OS/2, OS/X and ELF systems. (see mozpack.executables.is_executable documentation).

copy(*dest*, *skip_if_older=True*)

class mozpack.files.**ExistingFile**(*required*)
Bases: *mozpack.files.BaseFile*

File class that represents a file that may exist but whose content comes from elsewhere.

This purpose of this class is to account for files that are installed via external means. It is typically only used in manifests or in registries to account for files.

When asked to copy, this class does nothing because nothing is known about the source file/data.

Instances of this class come in two flavors: required and optional. If an existing file is required, it must exist during copy() or an error is raised.

copy(*dest*, *skip_if_older=True*)

class mozpack.files.**File**(*path*)
Bases: *mozpack.files.BaseFile*

File class for plain files.

mode
Return the file's unix mode, as returned by os.stat().st_mode.

read()
Return the contents of the file.

class mozpack.files.**FileFinder**(*base*, *find_executables=True*, *ignore=()*, *find_dotfiles=False*, ***kargs*)
Bases: *mozpack.files.BaseFinder*

Helper to get appropriate BaseFile instances from the file system.

get(*path*)

class mozpack.files.**GeneratedFile**(*content*)
Bases: *mozpack.files.BaseFile*

File class for content with no previous existence on the filesystem.

open()

class mozpack.files.**JarFinder**(*base*, *reader*, ***kargs*)
Bases: *mozpack.files.BaseFinder*

Helper to get appropriate DeflatedFile instances from a JarReader.

class mozpack.files.**ManifestFile**(*base*, *entries=None*)
Bases: *mozpack.files.BaseFile*

File class for a manifest file. It takes individual manifest entries (using the add() and remove() member functions), and adjusts them to be relative to the base path for the manifest, given at creation. Example:

> There is a manifest entry "content foobar foobar/content/" relative to "foobar/chrome". When packaging, the entry will be stored in jar:foobar/omni.ja!/chrome/chrome.manifest, which means the entry will have to be relative to "chrome" instead of "foobar/chrome". This doesn't really matter when serializing the entry, since this base path is not written out, but it matters when moving the entry at the same time, e.g. to jar:foobar/omni.ja!/chrome.manifest, which we don't do currently but could in the future.

add(*entry*)
Add the given entry to the manifest. Entries are rebased at open() time instead of add() time so that they can be more easily remove()d.

isempty()
Return whether there are manifest entries to write

open()
Return a file-like object allowing to read() the serialized content of the manifest.

remove(*entry*)
Remove the given entry from the manifest.

class mozpack.files.**MercurialFile**(*client*, *rev*, *path*)
Bases: *mozpack.files.BaseFile*

File class for holding data from Mercurial.

read()

class mozpack.files.**MercurialRevisionFinder**(*repo*, *rev='.'*, *recognize_repo_paths=False*, ***kwargs*)
Bases: *mozpack.files.BaseFinder*

A finder that operates on a specific Mercurial revision.

get(*path*)

class mozpack.files.**MinifiedJavaScript**(*file*, *verify_command=None*)
Bases: *mozpack.files.BaseFile*

File class for minifying JavaScript files.

open()

class mozpack.files.**MinifiedProperties**(*file*)
Bases: *mozpack.files.BaseFile*

File class for minified properties. This wraps around a BaseFile instance, and removes lines starting with a # from its content.

open()
Return a file-like object allowing to read() the minified content of the properties file.

class mozpack.files.**PreprocessedFile**(*path*, *depfile_path*, *marker*, *defines*, *extra_depends=None*, *silence_missing_directive_warnings=False*)
Bases: *mozpack.files.BaseFile*

File class for a file that is preprocessed. PreprocessedFile.copy() runs the preprocessor on the file to create the output.

copy(*dest*, *skip_if_older=True*)
Invokes the preprocessor to create the destination file.

class mozpack.files.**XPTFile**
Bases: *mozpack.files.GeneratedFile*

File class for a linked XPT file. It takes several XPT files as input (using the add() and remove() member functions), and links them at copy() time.

add(*xpt*)
Add the given XPT file (as a BaseFile instance) to the list of XPTs to link.

copy(*dest*, *skip_if_older=True*)
Link the registered XPTs and place the resulting linked XPT at the destination given as a string or a Dest

instance. Avoids an expensive XPT linking if the interfaces in an existing destination match those of the individual XPTs to link. skip_if_older is ignored.

isempty()
Return whether there are XPT files to link.

open()

remove(*xpt*)
Remove the given XPT file (as a BaseFile instance) from the list of XPTs to link.

22.4.9 mozpack.hg module

22.4.10 mozpack.manifests module

class mozpack.manifests.**InstallManifest**(*path=None*, *fileobj=None*)
Bases: object

Describes actions to be used with a copier.FileCopier instance.

This class facilitates serialization and deserialization of data used to construct a copier.FileCopier and to perform copy operations.

The manifest defines source paths, destination paths, and a mechanism by which the destination file should come into existence.

Entries in the manifest correspond to the following types:

copy – The file specified as the source path will be copied to the destination path.

symlink – The destination path will be a symlink to the source path. If symlinks are not supported, a copy will be performed.

exists – The destination path is accounted for and won't be deleted by the FileCopier. If the destination path doesn't exist, an error is raised.

optional – The destination path is accounted for and won't be deleted by the FileCopier. No error is raised if the destination path does not exist.

patternsymlink – Paths matched by the expression in the source path will be symlinked to the destination directory.

patterncopy – Similar to patternsymlink except files are copied, not symlinked.

preprocess – The file specified at the source path will be run through the preprocessor, and the output will be written to the destination path.

content – The destination file will be created with the given content.

Version 1 of the manifest was the initial version. Version 2 added optional path support Version 3 added support for pattern entries. Version 4 added preprocessed file support. Version 5 added content support.

CONTENT = 8

COPY = 2

CURRENT_VERSION = 5

FIELD_SEPARATOR = u'\x1f'

OPTIONAL_EXISTS = 4

PATTERN_COPY = 6

PATTERN_SYMLINK = 5

PREPROCESS = 7

REQUIRED_EXISTS = 3

SYMLINK = 1

add_content(*content*, *dest*)
Add a file with the given content.

add_copy(*source*, *dest*)
Add a copy to this manifest.

source will be copied to dest.

add_optional_exists(*dest*)
Record that a destination file may exist.

This effectively prevents the listed file from being deleted. Unlike a "required exists" file, files of this type do not raise errors if the destination file does not exist.

add_pattern_copy(*base*, *pattern*, *dest*)
Add a pattern match that results in copies.

See `add_pattern_symlink()` for usage.

add_pattern_symlink(*base*, *pattern*, *dest*)
Add a pattern match that results in symlinks being created.

A `FileFinder` will be created with its base set to `base` and `FileFinder.find()` will be called with `pattern` to discover source files. Each source file will be symlinked under `dest`.

Filenames under `dest` are constructed by taking the path fragment after `base` and concatenating it with `dest`. e.g.

<base>/foo/bar.h -> <dest>/foo/bar.h

add_preprocess(*source*, *dest*, *deps*, *marker=u'#'*, *defines={}*, *silence_missing_directive_warnings=False*)
Add a preprocessed file to this manifest.

`source` will be passed through preprocessor.py, and the output will be written to `dest`.

add_required_exists(*dest*)
Record that a destination file must exist.

This effectively prevents the listed file from being deleted.

add_symlink(*source*, *dest*)
Add a symlink to this manifest.

dest will be a symlink to source.

populate_registry(*registry*, *defines_override={}*)
Populate a mozpack.copier.FileRegistry instance with data from us.

The caller supplied a FileRegistry instance (or at least something that conforms to its interface) and that instance is populated with data from this manifest.

Defines can be given to override the ones in the manifest for preprocessing.

write(*path=None*, *fileobj=None*)
Serialize this manifest to a file or file object.

If path is specified, that file will be written to. If fileobj is specified, the serialized content will be written to that file object.

It is an error if both are specified.

exception mozpack.manifests.**UnreadableInstallManifest**
Bases: exceptions.Exception

Raised when an invalid install manifest is parsed.

22.4.11 mozpack.mozjar module

class mozpack.mozjar.**Deflater**(*compress=True*, *compress_level=9*)
Bases: object

File-like interface to zlib compression. The data is actually not compressed unless the compressed form is smaller than the uncompressed data.

close()
Close the Deflater.

compressed
Return whether the data should be compressed.

compressed_data
Return the compressed data, if the data should be compressed (real compressed size smaller than the uncompressed size), or the uncompressed data otherwise.

compressed_size
Return the compressed size of the data written to the Deflater. If the Deflater is set not to compress, the uncompressed size is returned. Otherwise, if the data should not be compressed (the real compressed size is bigger than the uncompressed size), return the uncompressed size.

crc32
Return the crc32 of the data written to the Deflater.

uncompressed_size
Return the size of the data written to the Deflater.

write(*data*)
Append a buffer to the Deflater.

class mozpack.mozjar.**JarCdirEnd**(*data=None*)
Bases: *mozpack.mozjar.JarStruct*

End of central directory record.

MAGIC = 101010256

STRUCT = OrderedDict([('disk_num', 'uint16'), ('cdir_disk', 'uint16'), ('disk_entries', 'uint16'), ('cdir_entries', 'uint16')

class mozpack.mozjar.**JarCdirEntry**(*data=None*)
Bases: *mozpack.mozjar.JarStruct*

Central directory file header

MAGIC = 33639248

STRUCT = OrderedDict([('creator_version', 'uint16'), ('min_version', 'uint16'), ('general_flag', 'uint16'), ('compression',

class mozpack.mozjar.**JarFileReader**(*header*, *data*)
Bases: object

File-like class for use by JarReader to give access to individual files within a Jar archive.

close()
Free the uncompressed data buffer.

compressed_data
Return the raw compressed data.

read(*length=-1*)
Read some amount of uncompressed data.

readlines()
Return a list containing all the lines of data in the uncompressed data.

seek(*pos*, *whence=0*)
Change the current position in the uncompressed data. Subsequent reads will start from there.

uncompressed_data
Return the uncompressed data.

class mozpack.mozjar.**JarLocalFileHeader**(*data=None*)
Bases: *mozpack.mozjar.JarStruct*

Local file header

MAGIC = 67324752

STRUCT = OrderedDict([('min_version', 'uint16'), ('general_flag', 'uint16'), ('compression', 'uint16'), ('lastmod_time', 'u

class mozpack.mozjar.**JarLog**(*file=None*, *fileobj=None*)
Bases: dict

Helper to read the file Gecko generates when setting MOZ_JAR_LOG_FILE. The jar log is then available as a dict with the jar path as key (see canonicalize for more details on the key value), and the corresponding access log as a list value. Only the first access to a given member of a jar is stored.

static canonicalize(*url*)
The jar path is stored in a MOZ_JAR_LOG_FILE log as a url. This method returns a unique value corresponding to such urls. - file:///{path} becomes {path} - jar:file:///{path}!/{subpath} becomes ({path}, {subpath}) - jar:jar:file:///{path}!/{subpath}!/{subpath2} becomes

({path}, {subpath}, {subpath2})

class mozpack.mozjar.**JarReader**(*file=None*, *fileobj=None*, *data=None*)
Bases: object

Class with methods to read Jar files. Can open standard jar files as well as Mozilla jar files (see further details in the JarWriter documentation).

close()
Free some resources associated with the Jar.

entries
Return an ordered dict of central directory entries, indexed by filename, in the order they appear in the Jar archive central directory. Directory entries are skipped.

is_optimized
Return whether the jar archive is optimized.

last_preloaded
Return the name of the last file that is set to be preloaded. See JarWriter documentation for more details on preloading.

exception mozpack.mozjar.**JarReaderError**
Bases: exceptions.Exception

Error type for Jar reader errors.

class mozpack.mozjar.**JarStruct**(*data=None*)

Bases: object

Helper used to define ZIP archive raw data structures. Data structures handled by this helper all start with a magic number, defined in subclasses MAGIC field as a 32-bits unsigned integer, followed by data structured as described in subclasses STRUCT field.

The STRUCT field contains a list of (name, type) pairs where name is a field name, and the type can be one of 'uint32', 'uint16' or one of the field names. In the latter case, the field is considered to be a string buffer with a length given in that field. For example,

STRUCT = [('version', 'uint32'), ('filename_size', 'uint16'), ('filename', 'filename_size')

]

describes a structure with a 'version' 32-bits unsigned integer field, followed by a 'filename_size' 16-bits unsigned integer field, followed by a filename_size-long string buffer 'filename'.

Fields that are used as other fields size are not stored in objects. In the above example, an instance of such subclass would only have two attributes:

obj['version'] obj['filename']

filename_size would be obtained with len(obj['filename']).

JarStruct subclasses instances can be either initialized from existing data (deserialized), or with empty fields.

TYPE_MAPPING = {'uint16': ('H', 2), 'uint32': ('I', 4)}

static get_data(*type*, *data*)

Deserialize a single field of given type (must be one of JarStruct.TYPE_MAPPING) at the given offset in the given data.

serialize()

Serialize the data structure according to the data structure definition from self.STRUCT.

size

Return the size of the data structure, given the current values of all variable length fields.

class mozpack.mozjar.**JarWriter**(*file=None*, *fileobj=None*, *compress=True*, *optimize=True*, *compress_level=9*)

Bases: object

Class with methods to write Jar files. Can write more-or-less standard jar archives as well as jar archives optimized for Gecko. See the documentation for the close() member function for a description of both layouts.

add(*name*, *data*, *compress=None*, *mode=None*)

Add a new member to the jar archive, with the given name and the given data. The compress option indicates if the given data should be compressed (True), not compressed (False), or compressed according to the default defined when creating the JarWriter (None). When the data should be compressed (True or None with self.compress == True), it is only really compressed if the compressed size is smaller than the uncompressed size. The mode option gives the unix permissions that should be stored for the jar entry. The given data may be a buffer, a file-like instance, a Deflater or a JarFileReader instance. The latter two allow to avoid uncompressing data to recompress it.

finish()

Flush and close the Jar archive.

Standard jar archives are laid out like the following:

- Local file header 1
- File data 1
- Local file header 2

- File data 2
- (...)
- Central directory entry pointing at Local file header 1
- Central directory entry pointing at Local file header 2
- (...)
- End of central directory, pointing at first central directory entry.

Jar archives optimized for Gecko are laid out like the following:

- 32-bits unsigned integer giving the amount of data to preload.
- Central directory entry pointing at Local file header 1
- Central directory entry pointing at Local file header 2
- (...)
- End of central directory, pointing at first central directory entry.
- Local file header 1
- File data 1
- Local file header 2
- File data 2
- (...)
- End of central directory, pointing at first central directory entry.

The duplication of the End of central directory is to accomodate some Zip reading tools that want an end of central directory structure to follow the central directory entries.

preload(*files*)
Set which members of the jar archive should be preloaded when opening the archive in Gecko. This reorders the members according to the order of given list.

exception mozpack.mozjar.**JarWriterError**
Bases: exceptions.Exception

Error type for Jar writer errors.

22.4.12 mozpack.path module

mozpack.path.**abspath**(*path*)

mozpack.path.**basedir**(*path*, *bases*)
Given a list of directories (bases), return which one contains the given path. If several matches are found, the deepest base directory is returned.

basedir('foo/bar/baz', ['foo', 'baz', 'foo/bar']) returns 'foo/bar' ('foo' and 'foo/bar' both match, but 'foo/bar' is the deepest match)

mozpack.path.**basename**(*path*)

mozpack.path.**commonprefix**(*paths*)

mozpack.path.**dirname**(*path*)

mozpack.path.**join**(**paths*)

mozpack.path.**match**(*path*, *pattern*)
Return whether the given path matches the given pattern. An asterisk can be used to match any string, including the null string, in one part of the path:

'foo' matches '*', 'f*' or 'fo*o'

However, an asterisk matching a subdirectory may not match the null string: 'foo/bar' does *not* match 'foo/*/bar'

If the pattern matches one of the ancestor directories of the path, the patch is considered matching:

'foo/bar' matches 'foo'

Two adjacent asterisks can be used to match files and zero or more directories and subdirectories.

'foo/bar' matches 'foo**/bar', or '**/bar'

mozpack.path.**normpath**(*path*)

mozpack.path.**normsep**(*path*)
Normalize path separators, by using forward slashes instead of whatever os.sep is.

mozpack.path.**realpath**(*path*)

mozpack.path.**rebase**(*oldbase*, *base*, *relativepath*)
Return relativepath relative to base instead of oldbase.

mozpack.path.**relpath**(*path*, *start*)

mozpack.path.**split**(*path*)

Return the normalized path as a list of its components. split('foo/bar/baz') returns ['foo', 'bar', 'baz']

mozpack.path.**splitext**(*path*)

22.4.13 mozpack.unify module

class mozpack.unify.**UnifiedBuildFinder**(*finder1*, *finder2*, ***kargs*)
Bases: *mozpack.unify.UnifiedFinder*

Specialized UnifiedFinder for Mozilla applications packaging. It allows "*.manifest" files to differ in their order, and unifies "buildconfig.html" files by merging their content.

unify_file(*path*, *file1*, *file2*)
Unify files taking Mozilla application special cases into account. Otherwise defer to UnifiedFinder.unify_file.

class mozpack.unify.**UnifiedExecutableFile**(*executable1*, *executable2*)
Bases: *mozpack.files.BaseFile*

File class for executable and library files that to be unified with 'lipo'.

copy(*dest*, *skip_if_older=True*)
Create a fat executable from the two Mach-O executable given when creating the instance. skip_if_older is ignored.

class mozpack.unify.**UnifiedFinder**(*finder1*, *finder2*, *sorted=[]*, ***kargs*)
Bases: *mozpack.files.BaseFinder*

Helper to get unified BaseFile instances from two distinct trees on the file system.

unify_file(*path*, *file1*, *file2*)
Given two BaseFiles and the path they were found at, check whether their content match and return the first BaseFile if they do.

mozpack.unify.**may_unify_binary**(*file*)
Return whether the given BaseFile instance is an ExecutableFile that may be unified. Only non-fat Mach-O binaries are to be unified.

22.4.14 Module contents

22.5 mozversioncontrol package

22.5.1 Submodules

22.5.2 mozversioncontrol.repoupdate module

22.5.3 Module contents

22.6 mozwebidlcodegen package

22.6.1 Module contents

class mozwebidlcodegen.**BuildResult**
Bases: object

Represents the result of processing WebIDL files.

This holds a summary of output file generation during code generation.

class mozwebidlcodegen.**BuildSystemWebIDL**(*topsrcdir*, *settings*, *log_manager*, *topobjdir=None*, *mozconfig=<object object>*)
Bases: *mozbuild.base.MozbuildObject*

manager

class mozwebidlcodegen.**WebIDLCodegenManager**(*config_path*, *inputs*, *exported_header_dir*, *codegen_dir*, *state_path*, *cache_dir=None*, *make_deps_path=None*, *make_deps_target=None*)
Bases: *mach.mixin.logging.LoggingMixin*

Manages all code generation around WebIDL.

To facilitate testing, this object is meant to be generic and reusable. Paths, etc should be parameters and not hardcoded.

GLOBAL_DECLARE_FILES = set([u'GeneratedAtomList.h', u'RegisterWorkerBindings.h', u'ResolveSystemBinding.h', u

GLOBAL_DEFINE_FILES = set([u'UnionTypes.cpp', u'RegisterWorkerDebuggerBindings.cpp', u'RegisterWorkerBindin

config

expected_build_output_files()
Obtain the set of files generate_build_files() should write.

generate_build_files()
Generate files required for the build.

This function is in charge of generating all the .h/.cpp files derived from input .webidl files. Please note that there are build actions required to produce .webidl files and these build actions are explicitly not captured here: this function assumes all .webidl files are present and up to date.

This routine is called as part of the build to ensure files that need to exist are present and up to date. This routine may not be called if the build dependencies (generated as a result of calling this the first time) say everything is up to date.

Because reprocessing outputs for every .webidl on every invocation is expensive, we only regenerate the minimal set of files on every invocation. The rules for deciding what needs done are roughly as follows:

1. If any .webidl changes, reparse all .webidl files and regenerate the global derived files. Only regenerate output files (.h/.cpp) impacted by the modified .webidl files.
2. If an non-.webidl dependency (Python files, config file) changes, assume everything is out of date and regenerate the world. This is because changes in those could globally impact every output file.
3. If an output file is missing, ensure it is present by performing necessary regeneration.

generate_example_files(*interface*)
Generates example files for a given interface.

class mozwebidlcodegen.**WebIDLCodegenManagerState**(*fh=None*)
Bases: dict

Holds state for the WebIDL code generation manager.

State is currently just an extended dict. The internal implementation of state should be considered a black box to everyone except WebIDLCodegenManager. But we'll still document it.

Fields:

version The integer version of the format. This is to detect incompatible changes between state. It should be bumped whenever the format changes or semantics change.

webidls A dictionary holding information about every known WebIDL input. Keys are the basenames of input WebIDL files. Values are dicts of metadata. Keys in those dicts are:

- filename - The full path to the input filename.
- inputs - A set of full paths to other webidl files this webidl depends on.
- outputs - Set of full output paths that are created/derived from this file.
- sha1 - The hexidecimal SHA-1 of the input filename from the last processing time.

global_inputs A dictionary defining files that influence all processing. Keys are full filenames. Values are hexidecimal SHA-1 from the last processing time.

VERSION = 1

dump(*fh*)
Dump serialized state to a file handle.

mozwebidlcodegen.**create_build_system_manager**(*topsrcdir*, *topobjdir*, *dist_dir*)
Create a WebIDLCodegenManager for use by the build system.

CHAPTER 23

Managing Documentation

This documentation is generated via the Sphinx tool from sources in the tree.

To build the documentation, run `mach doc`. Run `mach help doc` to see configurable options.

23.1 Adding Documentation

To add new documentation, define the `SPHINX_TREES` and `SPHINX_PYTHON_PACKAGE_DIRS` variables in `moz.build` files in the tree and documentation will automatically get picked up.

Say you have a directory `featureX` you would like to write some documentation for. Here are the steps to create Sphinx documentation for it:

1. Create a directory for the docs. This is typically `docs`. e.g. `featureX/docs`.
2. Create an `index.rst` file in this directory. The `index.rst` file is the root documentation for that section. See `build/docs/index.rst` for an example file.
3. In a `moz.build` file (typically the one in the parent directory of the `docs` directory), define `SPHINX_TREES` to hook up the plumbing. e.g. `SPHINX_TREES['featureX'] = 'docs'`. This says *the ''docs'' directory under the current directory should be installed into the Sphinx documentation tree under ''/featureX''.*
4. If you have Python packages you would like to generate Python API documentation for, you can use `SPHINX_PYTHON_PACKAGE_DIRS` to declare directories containing Python packages. e.g. `SPHINX_PYTHON_PACKAGE_DIRS += ['mozpackage']`.

CHAPTER 24

Indices and tables

- genindex
- modindex
- search

m

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X