Gerrit Hooks Documentation

Release 1.0

David Caro

May 17, 2018

Contents

CHAPTER 1

Getting started

1.1 Installing the hooks

The hooks infrastructure is separatede in two parts, the hook dispatcher, and the actual hooks.

The dispatcher is in charge of deciding which hooks to run for each event, and gives the final review on the change. The hooks themselves are the ones that actually do checks (or any other action needed) and where the actual login you would want to implement resides.

1.1.1 Install the dispatcher

To install the hook dispatcher just add it to the gerrit configuration as hook for any events you want to create hooks for, for example, soft linking it in $review_site/hooks/ like this:

```
[root@gerrit ~]# ll /home/gerrit2/review_site/hooks/
change-abandoned -> /home/gerrit2/review_site/hooks/hook-dispatcher
change-merged -> /home/gerrit2/review_site/hooks/hook-dispatcher
comment-added -> /home/gerrit2/review_site/hooks/hook-dispatcher
patchset-created -> /home/gerrit2/review_site/hooks/hook-dispatcher
```

That will allow you to manage the events 'change-abandoned', 'change-merged', 'comment-added' and 'patchset-created'.

Alternatively you can configure gerrit to use the hook-dispatcher from gerrit.conf (see the gerrit config help)

Note: Make sure it's executable

1.1.2 Install the hooks

Once the dispatcher is in place, you can add per-project hooks, those are just executable scripts, you can use anything you like, though I suggest taking a look at the libs here, that already handle most of the hustle for you.

So to install a hook, just put it under the $review_site/git/$project.git/hooks/ directory with the name:

```
$event.[$chain.]$name
```

For now, we can ignore the *$chain*, it's explained later in the *Execution flow: chains* section. The *$name* is any string you want to identify the hook you just installed.

I recommend keeping all the hooks you install in the same directory, for example, under *$review_site/hooks/custom_hooks* and just create soft-links to them on the *$review_site/git/$project/hooks* directory for ease of management and maintenance.

For example, the current hooks for the ovirt-engine oVirt project:

```
change-abandoned.update_tracker -> update_tracker
change-merged.set_MODIFIED -> /home/gerrit2/review_site/hooks/custom_hooks/change-
↪merged.set_MODIFIED
change-merged.update_tracker -> update_tracker
comment-added.propagate_review_values -> /home/gerrit2/review_site/hooks/custom_hooks/
↪comment-added.propagate_review_values
patchset-created.bz.0.has_bug_url -> /home/gerrit2/review_site/hooks/custom_hooks/
↪patchset-created.bz.0.has_bug_url
patchset-created.bz.1.is_public -> /home/gerrit2/review_site/hooks/custom_hooks/
↪patchset-created.bz.1.is_public
patchset-created.bz.2.correct_product -> /home/gerrit2/review_site/hooks/custom_hooks/
↪patchset-created.bz.2.correct_product
patchset-created.bz.3.correct_target_milestone -> /home/gerrit2/review_site/hooks/
↪custom_hooks/patchset-created.bz.3.correct_target_milestone
patchset-created.bz.98.set_POST -> /home/gerrit2/review_site/hooks/custom_hooks/
↪patchset-created.bz.98.set_POST
patchset-created.bz.99.review_ok -> /home/gerrit2/review_site/hooks/custom_hooks/
↪patchset-created.bz.99.review_ok
patchset-created.update_tracker -> update_tracker
patchset-created.warn_if_not_merged_to_previous_branch -> /home/gerrit2/review_site/
↪hooks/custom_hooks/patchset-created.warn_if_not_merged_to_previous_branch
update_tracker -> /home/gerrit2/review_site/hooks/custom_hooks/update_tracker
```

1.1.3 Execution flow: chains

So as was said before, when you install a hook you can optionally specify a *$chain* in it's name. That is to allow a better control of the execution flow. You can find a detailed description at the JUC 2014 presentation.

The key idea is that using a chain of hooks, you can control with the return code if the chain is broken and skip the execution of the rest of the chain, jumping directly to the next.

Check the hook_dispatcher.run_hooks docs for more details on the return codes.

1.2 Some tips on configuring the hooks

NOTE: make sure to check the latest docs for each programming language under the bash libs or python libs pages, in the conf.sh/config libs sections

As specified in the bash libs config.sh section, the hooks can get their config from multiple places, the most common pattern is to have a generic configuration file and a per-project one (both are supported also by the python hooks, any other is bash-specific, more on than on *Bash hooks*).

So usually you'd have a *config* file under */home/gerrit2/review_site/hooks* with the general options, like admin email and such. For a list of the needed options check on the hooks section for whichever hooks you have configured.

That file should be a valid shell script, as it will be sourced on runtime and right now there's a limitation that all the values must be on the same line as the variable name (for python hooks to pick them up).

For example:

```
#!/bin/bash
## Credentials to use when connecting to bugzilla
BZ_USER='something@somewh.ere'
BZ_PASS='supersecterpass'

## Gerrit credentials/url used to review the patches (through ssh cli)
GERRIT_SRV="user@gerrit.server"

## Tracker id on bugzilla for the autotracker hook
## 81 -> oVirt gerrit
TRACKER_ID='81'
TRACKER_NAME="oVirt gerrit"

PRODUCT='oVirt'
PRODUCTS=('oVirt' 'Red Hat Enterprise Virtualization Manager')
CLASSIFICATION='oVirt'
```

If there's anything that's specific for a project (like branches and tags) that will go under the project's git repo, under *hooks/cofing*, for example, if the project was named ovirt-engine.git, the config file at */home/gerrit2/review_site/git/ovirt-engine.git/hooks/config* might be:

```
#!/bin/bash
## Branches to take into account
BRANCHES=('ovirt-engine-3.6' 'ovirt-engine-3.6.0' 'ovirt-engine-3.6.1' 'ovirt-engine-
↪3.6.2')
STABLE_BRANCHES="ovirt-engine-3.6 ovirt-engine-3.6.5 ovirt-engine-3.6.6"
CHECK_TARGET_RELEASE=("ovirt-engine-3.6|^3\.[6543210].*")
CHECK_TARGET_MILESTONE=('ovirt-engine-3.6|^.*3\.6.*')
PRODUCT="oVirt"
```

Those values will be available only to hooks for that project, and will override any parameter in the more generic config (like *PRODUCT* here).

1.2.1 Bash hooks

If you are using bash hooks, there are a few more levels of config supported, those are:

- **Per project + chain, for example::** /home/gerrit2/git/ovirt-engine.git/hooks/bz.config
- **Per project + chain + event, like::** /home/gerrit2/git/ovirt-engine.git/hooks/bz.change-merged.config
- **Per project + chain + event + hook::** /home/gerrit2/git/ovirt-engine.git/hooks/bz.change-merged.99.review_ok.config

Those are not used usually and are meant only for very specific cases.

1.3 Small tutorial on adding a new hook

1.3.1 Hello world

To create a dummy hello world, you can just create a binary (C, C++, Python, Ruby, Java, Go. . .) and put it under your project's git repo under the hooks directory with the name:

```
$gerrit_event.hello_world
```

For example, if we used a bash script, just:

```
$ echo -e '#!/bin/bash\necho hello world' \
    > /home/gerrit2/review_site/git/lago.git/hooks/comment-added.hello_world
$ chmod +x /home/gerrit2/review_site/git/lago.git/hooks/comment-added.hello_world
```

And our hook would be run automatically on each new comment, and you will start seeing some feedback on the comments, something like:

```
* hello_word: hello world
```

Note that the stdout and stderr are shown only in the logs, we'll see later how to do reviews.

1.3.2 Using some config values (bash only)

So, imagine that we wanted to get some configuration values extracted, to do so, you have to source the conf.sh library (conveniently added to the path by the hook dispacher), so an example could be:

```
source conf.sh
conf.load

echo "Hello world, the config value of SOMETHING is $SOMETHING"
```

As you can see, all the configuration values that are defined on any of the configuration files for that hook are now global variables.

1.3.3 Doing reviews (bash only)

If you want to add some custom values for the code review and verified flags, your hook must adopt a specific format on it's output, but don't worry, there's a function to help you with that, here's an example:

```
source tools.sh
code_review_value="-1"
verified_value="1"
message="Some helpful message here"
tools.review "$code_review_value" "$verified_value" "$message"
```

That will add a review with a CR value of -1, and a V value of +1.

1.3.3.1 Skipping review flags

Something tricky here, as doing a review with 0 is not the same as not doing it, if you want to skip the vote for the hook, you should use an empty string for the CR/V flags value, like this:

```
tools.review "" "" "Info message"
```

If you use ‘0’ instead, the review value will be added to the list, and for the reviews, the lowest value among all the hooks that ran is the one that prevails, so if you have a +1 and a 0, 0 is the final value you will have.

1.3.4 Interacting with bugzilla (bash only)

Similar to the conf.sh, we have the bz.sh library, that will allow us to interact with a bugzilla server, a simple example getting some info:

```
source bz.sh
source conf.sh

conf.load

bz.login "$BZ_USER" "$BZ_PASS" \
|| {
    message+="${message:+\n}* Bug-Url: ERROR, There was an error logging into␣
↪bugzilla, "
    message+="please contact the administrator $CONTACT"
    tools.review "" "" "$message"
    bz.clean
    exit 2
}
bug_ids=($(bz.get_bug_id $commit))
for bug_id in ${bug_ids[@]}; do
    prod="$(bz.get_product "$bug_id")"
    echo "Bug $bug_id has prod $prod"
done
bz.clean
```

NOTE: Make sure to run bz.clean at the end, that will get rid of any cache and temporary configurations (to increase speed, bz.get uses a local cache for the bugs)

Another interesting thing to point out, is that *bz.get_bug_id* call, that will extract from the current commit (the one that triggered the hook) all the *Bug-Url:* headers and return an array with the numerical bug ids.

CHAPTER 2

Hooks

2.1 Bash hooks

2.2 Python Hooks

CHAPTER 3

Libs

3.1 Bash libraries

file **`bz.sh`**

Helpful functions to handle bugzilla service interactions

Functions

bz **`get_bug`**(bug_id)

Get's the bug_id bug json information from cache, or if not cached, from the server.

Parameters

- `bug_id`: id of the bug to retrieve

Return Value

- `0`:

bz **`update_bug`**(bug_id, data...)

Updates the given bug.

Parameters

- `bug_id`: id of the bug to update
- `data...`: Each of the post parameters to send (usually as name=value).

Return Value

- `0`: if the bug was updated
- `1`: if it was not updated

bz **is_revert** (commit)
Checks if the given commit is a revert.

Parameters

- `commit`: refspec of the commit to check (default=HEAD)

Return Value

- `0`: if the given commit is a revert
- `1`: otherwise

bz **get_bug_id** (commit)
Extracts the bug ids from the Bug-Url in the given commit.

Note If the patch is a 'revert', it extracts the bug from the reverted commit

Parameters

- `commit`: refspec to get the bug from

bz **login** (bz_user, bz_password)
Logs into bugzilla if not logged in already.

```
Options:
    -b bug_id
      If you pass a bug_id, the token for that bug will already be set and
      cached for further reference

    -s server_url
      Use that url instead of the one in the config file
      (https://bugzilla.redhat.com by default)

Configuration parameters:
   bugzilla_server
     full url to the bugzilla server
```

Parameters

- `bz_user`: User to log into bugzilla
- `bz_password`: Password

bz **get_bug_flags** (bugid)
Retrieves all the '+' flags of the given bug.

Parameters

- `bugid`: Id of the bug to get the flags from

bz **get_bug_status** (bugid)
Retrieves the current status of the bug.

Parameters

- `bugid`: Id of the bug to retrieve

bz **check_flags** (bugid, flagspecs...)
Checks that all the flags exist with '+' in the given bug.

Parameters

- `bugid`: Id of the bug to check the flags for
- `flagspecs...`: can be a single flag or a sequence ot flags separated by '|' to express that those flags are interchangeable, for example flag2|flag2_bis

bz **`add_tracker`** (bug_id, tracker_id, external_id, description)
Add a new external bug to the external bugs list.

Parameters

- `bug_id`: Id of the bug to update
- `tracker_id`: This is the internal tracker id that bugzilla assigns to each external tracker (RHEV gerrit -> 82, oVirt gerrit -> 81)
- `external_id`: Id for the bug in the external tracker
- `description`: Description to add to the external tracker

bz **`update_fixed_in_version`** (bug_id, fixed_in_version)
Update fixed in version field.

Parameters

- `bug_id`: Id of the bug to update
- `fixed_in_version`: New value for the fixed in version field

bz **`update_status_and_version`** (bug_id, bug_status, fixed_in_version, resolution)
Update fixed in version field plus the status field.

Parameters

- `bug_id`: Id of the bug to update
- `bug_status`: New value for the status field
- `fixed_in_version`: New value for the fixed in version field
- `resolution`: New value for the resolution

bz **`update_status`** (bug_id, new_status, commit_id, resolution)

```
Legal status transitions:
  NEW|ASSIGNED|MODIFIED -> POST
  POST -> MODIFIED

If it's a revert any source status is allowed
```

Parameters

- `bug_id`: Id of the bug to update
- `new_status`: New status to set the bug to, only the allowed transitions will end in a positive result (return code 0)
- `commit_id`: Id of the commit that should change the status of this bug
- `resolution`: In case that the status is CLOSED, the resolution is needed

bz **get_external_bugs**(bug_id, external_name)

Parameters

- `bug_id`: Id of the parent bug
- `external_name`: External string to get the bugs from. If none given it will get all the external bugs. Usually one of:
  - oVirt gerrit
  - RHEV gerrit

bz **clean**()

Cleans up all the cached config and data. Make sure that your last scripts calls it before exitting.

bz **get_product**(bug_id)

Prints the product name of the given bug.

Parameters

- `bug_id`: Id of the bug to get info about

bz **get_classification**(bug_id)

Print the classification of the given bug.

Parameters

- `bug_id`: Id of the bug to get info about

bz **is_private**(bug_id)

Parameters

- `bug_id`: Id of the bug to check

Return Value

- `0`: if it's private
- `1`: otherwise

bz **get_target_milestone**(bug_id)

Print the target milestone of the bug.

Parameters

- `bug_id`: Id of the bug to get info about

bz **get_target_release**(bug_id)

Print the target release of the bug.

Parameters

- `bug_id`: Id of the bug to get info about

bz **check_target_release**(bug_id, branch, tr_match, branch_name)

```
Example:

 bz.check_target_release 1234 master 'master|3\.3.*' 'master|!3\.[21].*'
```

(continues on next page)

(continued from previous page)

```
That will check that the bug 1234 target release matches:
    3\.3.*
 And does not match:
    3\.3\.0\..*

 So 3.3.0 or 3.3 will pass but 3.2 and 3.3.0.1 will not
```

Parameters

- `bug_id`: Id of the bug to check the target_release of
- `branch`: Name of the current branch
- `tr_match`: Tuple in the form 'branch_name|[!]regexp'
- `branch_name`: name of the branch that should check the regexp

```
[!]regexp
    regular expresion to match the target release against, if preceded
     with '!' the expression will be negated
```

Return Value

- `1`: if the target release and branch defined in tr_match configuration variable do not match the given bug's target release

bz **`check_target_milestone`** (bug_id, branch, tm_match, branch_name)

```
Example:

  bz.check_target_milestone 1234 master 'master|3\.3.*' 'master|!3\.[21].*'

  That will check that the bug 1234 target milestone matches:
      3\.3.*
  And does not match:
      3\.3\.0\..*

  So 3.3.0 or 3.3 will pass but 3.2 and 3.3.0.1 will not
```

Return Value

- `1`: if the target milestone and branch defined in tm_match configuration variable do not match the given bug's target milestone

Parameters

- `bug_id`: Id of the bug to get the target_milestone from
- `branch`: Name of the current branch
- `tm_match`: Tuple in the form `branch_name|[!]regexp`
- `branch_name`: name of the branch that should check the regexp

```
[!]regexp
    regular expresion to match the target milestone against, if␣
↪preceded
     with '!' the expression will be negated
```

file **conf.sh**

Helper configuration functions

```
Configuration types
---------------------

There are two types of configurations taken into account, static and
temporary


Static configuration
++++++++++++++++++++++

Static configurations store those values that will persist after each
execution of the hooks, for example users and passwords.


Temporary configurations
++++++++++++++++++++++++++

Some hooks use temporary configurations to store values for other hooks to
recover, for example, when storing cookies.

Configuration hierarchy
++++++++++++++++++++++++

It will source the configuration files in order, skipping any non-existing
ones. The paths where it will look for them are, in source order (the most
prioritary first)::

* $hook_path/$event_type.$chain.conf
* $hook_path/$event_type.conf
* $hook_path/$chain.conf
* $hook_path/conf
* $GERRIT_SITE/hooks/conf
* $HOME/hook/conf
* $hook_path/../../../hooks/conf

When running in gerrit, the $hook_path is usually the git repository of the
project the event was triggered for, for example::

   /home/gerrit2/review_site/git/ovirt-engine.git
```

Functions

conf **_get_conf_files**()

Print all the available configuration files from less relevant to more relevant.

conf **_get_conf_file**()

Print current's hook config file.

conf get(name, default)

Prints the given key from the config.

```
Options:
```

(continues on next page)

(continued from previous page)

```
-c conf_file
  Use that config file
```

Note the return code is 1 when the value is not found in the config files, and if specified outputs the default value too

Parameters

- `name`: Name of the key to get the value for
- `default`: Default value if not found

conf **`put`**`(`name, value`)`
Writes the given name/value to the configuration.

```
Options:

-c conf_file
  Use that config file
```

Parameters

- `name`: Key to store the conf value under
- `value`: Value to store

conf **`load`**`()`
Loads the config files from less specific to most so the latest prevails, all the conf entries are loaded as vars.

```
Options:

-c conf_file
  Use that config file
```

conf **`t_put`**`(`key, value`)`
Temporary config file functions, for the current executione.

Parameters

- `key`: Key to store
- `value`: Value to store Store the given key/value to temporary storage

`conf t_get(key, default)`
Print the given key from temporary storage.

Parameters

- `key`: Key to store
- `default`: Default value to print if not found

conf **`t_load`**`()`
load the temporary config

conf **`t_clean`**`()`
Cleanup the temporary config related temporary files.

file **gerrit.sh**

Helpful functions to manage gerrit related actions

Functions

gerrit **get_patch** (patch_id)

Print the bug info.

Note it is cached for faster access

Parameters

- `patch_id`: Patch id to retrieve info, as in 1234 (don't confuse with the change id)

gerrit **is_related** (commit)

Check if the patch has the Related-To tag.

Parameters

- `commit`: Refspec to check

Return Value

- `0`: if it did not have it
- `1`: otherwise

gerrit **parse_params** ()

Parse all the parameters as env variables, leave the rest as positional args.

```
gerrit.parse_params --param1 val1 param2 --param-3 val3
    => [[ param1 == "val1" ]] \
       && [[ param_3 == "val3" ]] \
       && [[ "$1" == "param2" ]]
```

gerrit **review** (result, message, project, commit)

Write a review, it will use the env commit and project vars, as set by parse_params.

Parameters

- `result`: Value for the verified flag (1, 0, -1)
- `message`: Message for the comment
- `project`: Gerrit project that owns the commit
- `commit`: Refspec for the commit of the patch to write a review for

gerrit **status** (id)

Print the status of the given patch.

Parameters

- `id`: Patch id to get the status for

gerrit **is_open** (id)

Check if a patch is open.

Parameters

- `id`: Patch id to check

Return Value

- `0`: if the patch is open
- `1`: otherwise

gerrit **`clean`**(nothing)
Clean up all the temporary files for the current run.

Parameters

- `nothing`: No parameters needed

gerrit **`get_branches`**(pattern)
Print the list of branches for the current project.

Parameters

- `pattern`: If passed, will filter out branches by that pattern (shell-like)

file **`tools.sh`**
Helpful miscellaneous functions

Functions

tools **`is_in`**(value, elem1...)

Parameters

- `value`: Value to look for
- `elem1...`: Elements to look among if a value is in the given list of elements

tools **`trim`**()
Remove the leading and trailing white spaces.

tools **`sanitize`**(word...)
Replace all the bad characters from the given word to fit a bash variable name specification.

Parameters

- `word...`: list of words to sanitize

tools **`hash`**(what, length)
Get a simple md5 hash of the given string.

Parameters

- `what`: Base string for the hash
- `length`: Max length for the hash, (default=10)

tools **`log`**(message...)
logs the given strings to stderr

Parameters

- `message...`: list of messages to log

tools **review** (cr, ver, msg)
print a review message with the format expeted by the hook dispatcher

Parameters

- `cr`: Value for the Code Review flag
- `ver`: Value for the Verified flag
- `msg`: Message for the review comment

tools **match** (base_string, match_string)

Parameters

- `base_string`: String to check for a match
- `match_string`: Tuple in the form '[!]regexp'

```
    [!]regexp
        regular expresion to match the base string against, if␣
↪preceded
         with '!' the expression will be negated

Example:

  tools.match 3.2.1 'master|3\.3.*' 'master|!3\.[21].*'

  That will check that the string 3.2.1 matches:
      3\.3.*
  And does not match:
      3\.3\.0\..*
```

Return Value

- `TOOLS_MATCHES|0`: if the base_string matches all of the match_string passed
- `TOOLS_DOES_NOT_MATCH`: if it does not match because of a positive match
- `TOOLS_SHOULD_NOT_MATCH`: if it was because of a negative match (started with !)

tools **ver_cmp** (version1, version2)
Example:

tools.ver_cmp 3.2.1 3.3

Return Value

- `TOOLS_VERCMP_EQUAL`: if the versions are the same
- `TOOLS_VERCMP_GT`: if version1 > version2
- `TOOLS_VERCMP_LT`: if version1 < version2

Parameters

- `version1`: String with the first version to compare (must be in the form X.Y(.Z)? where X, Y and Z are numbers)
- `version2`: String with the second version to compare (must be in the form X.Y(.Z)? where X, Y and Z are numbers)

dir **/home/docs/checkouts/readthedocs.org/user_builds/ovirt-gerrit-hooks/checkouts/latest/hoo**

dir **/home/docs/checkouts/readthedocs.org/user_builds/ovirt-gerrit-hooks/checkouts/latest/hoo**

3.2 Python libraries

3.2.1 lib.tools

exception `lib.tools.`**`NotComparableVersions`**
    Bases: `exceptions.Exception`

`lib.tools.`**`branch_has_change`**(*branch*, *change*, *repo_path*)

`lib.tools.`**`get_branches`**(*repo_path*)

`lib.tools.`**`get_newer_branches`**(*my_branch*, *repo_path*)

`lib.tools.`**`get_parser_comment_added`**(*description=None*)
    Build the parser for comment-added hook type

`lib.tools.`**`get_parser_pc`**(*description=None*)
    Build the parser for patchset-created hook type

`lib.tools.`**`ver_is_newer`**(*branch1*, *branch2*)
    Returns true if branch1 isnewer than branch2, according to:

    (X+1).Y > X.(Y+1) > X.Y > X.Y.Z

3.2.2 lib.bz

class `lib.bz.`**`Bug`**(*product*, *status*, *title*, *flags*, *target_rel*, *milestone*, *component*, *classification*, *external_bugs*)
    Bases: `object`

    Bug structure

class `lib.bz.`**`Bugzilla`**(*user=None*, *passwd=None*, *url='https://bugzilla.redhat.com/xmlrpc.cgi'*)
    Bases: `object`

    Bugzilla Object

    `extract_bug_info`(*bug_id*)
        Extract parameters from bz ticket

        Parameters **`bug_id`** – bug number

        Return bug object bug object that will hold all the bug data info

    static extract_flags(*list_of_dicts*)
        Get bugzilla bug flags

        Parameters **`list_of_dicts`** – list of dict flags

        :return dict of flag names (key) and flag statuses (value)

    static get_bug_ids(*bug_urls*)
        Get bug id from bug url

        Parameters **`bug_urls`** – list of bug urls

        Returns set of unique bug ids

    static get_bug_urls(*commit*, *bz_server='https://bugzilla.redhat.com'*)
        Get bug urls from the passed commit

        Parameters

- **commit** – commit message string
- **bz_server** – bugzilla server

Returns list of bug urls or empty list

get_external(*bug_id*, *external_bug_id*, *ensure_product=None*)

update_bug(*bug_id*, ***fields*)

update_external(*bug_id*, *external_bug_id*, *ext_type_id*, *status=None*, *description=None*, *branch=None*, *ensure_product=None*)

wrap(*dictionary*)

exception `lib.bz.`**WrongProduct**

Bases: `exceptions.Exception`

3.2.3 lib.gerrit

class `lib.gerrit.`**Change**

static get_ci_reviewers_name()

static get_ci_value(*by_users=None*)

Get the global patchset CI flag value, taking into account that +1 is more prioritary than -1

Parameters

- **change** – dict with the change info as returned by Gerrit.query
- **by_users** – list of user names whose reviews will be taken into account to calculate the global value, if empty or None will not filter the reviewers

static get_flag_values(*flag_name*, *by_users=None*)

Parameters

- **change** – dict with the change info as returned by Gerrit.query
- **flag_name** – Name of the flag to get the review values for
- **by_users** – List of users to filter the reviews by, if empty or not set will get them all

static get_reviewers_name(*flag_name*)

static has_code_change()

class `lib.gerrit.`**Gerrit**(*server*)

Bases: `object`

generate_cmd(*action*, **options*)

query(*query*, *all_approvals=False*, *all_reviewers=False*, *comment=False*, *commit_message=False*, *current_patch_set=False*, *dependencies=False*, *files=False*, *patch_sets=False*, *start=0*, *submit_records=False*, *out_format='json'*)

query_patchsets(**args*, ***kwargs*)

review(*commit*, *project*, *message*, *review=None*, *verify=None*, *ci=None*)

3.2.4 lib.config

3.2.4.1 Configuration

The cofiguration for the hooks is extracted from a hireachical structure, it will try to get the configuration, most prioritary first, from:

- ../config: that is, the parent directory of this file, in a file name config
- $GIT_DIR/hooks/config: That is (when running from gerrit) the hooks dir

inside the git repository, a file named config.

Multilines are not yet supported and the file should be bash compatible, as it might be used from bash scripts.

It's a slightly different behavior than the bash conf.sh lib as this module is not yet finished.

3.2.4.2 API

class `lib.config.`**`Config`**(*files=None*)
    Bases: `dict`

    `read`(*filename*)

`lib.config.`**`load_config`**()

`lib.config.`**`unquote`**(*string*)

3.3 Hook dispatcher

This dispatcher handles the hook execution

3.3.1 Allowed tags in gerrit commit message:

3.3.1.1 Ignore-Hooks

Do not run the given hooks:

```
Ignore-Hooks: hook1, hook2, ...
```

3.3.1.2 Run-Only-Hooks

Run only the given hooks if they exist:

```
Run-Only-Hooks: hook1, hook2 ...
```

3.3.2 Allowed tags in gerrit comment:

3.3.2.1 Rerun-Hooks

Run again the given hooks (patchset-created event hooks only):

```
Rerun-Hooks: hook1, hook2 ...
Rerun-Hooks: all
```

3.3.3 API

class `hook_dispatcher.`**`LoggerWriter`**(*logger*, *level*)
Redirect writes to the given logger.

`write`(*message*)

`hook_dispatcher.`**`csv_to_set`**(*csv_string*)

`hook_dispatcher.`**`flatten`**(*array*, *elems*)
Function that appends to the array the options given by elems, to build a new command line array.

Parameters

- **`array`** – Command line array as ['ls', '-l', '/tmp']
- **`elems`** – Command line options as parsed by argparse, like [('author', 'dcaro'), ('email', 'dcaroest@redhat.com')]

`hook_dispatcher.`**`get_chains`**(*hooks*)

`hook_dispatcher.`**`get_comment_tags`**(*comment*)
Get the tags that were specified in gerrit message

Parameters **`comment`** – Gerrit comment

`hook_dispatcher.`**`get_commit_tags`**(*commit*)
Get the tags that were specified in the comit message

Parameters **`commit`** – Commit to get the message from

`hook_dispatcher.`**`get_hook_type`**(*opts*)
Guess the right hook type, gerrit sometimes resolves the real path

Parameters **`opts`** – options given to the script, so it can guess the hook type

`hook_dispatcher.`**`get_hooks`**(*path*, *hook_type*)
Get all the hooks that match the given type in the given path

Parameters

- **`path`** – Path to look the hooks in
- **`hook_type`** – Hook type to look for

`hook_dispatcher.`**`get_parser`**()
Build the parser for any hook type

`hook_dispatcher.`**`get_tags`**(*tags_def*, *text*)
Retrieves all the tags defined in the tag_def from text

Parameters

- **`tags_def`** – Definition of the tags as specified above
- **`text`** – text to look for tags (commit message, gerrit message…)

`hook_dispatcher.`**`ignore`**(*ignored_hooks*, *current_hooks*)
Return the hooks list without the given hooks

:param ignored_hooks; csv lis tof the hooks to ignore :param current_hooks: array with the currently available hooks

`hook_dispatcher.`**`main`**`()`

`hook_dispatcher.`**`parse_stdout`**`(`*stdout*`)`

The expected output format is:

code_review_value verified_value multi_line_msg

Where code_review_value and verified_value can be an empty line if it's an abstention (no modifying the value any of the previous or next hooks in the chain set) If any of the lines is missing then it will assumed an abstention.

- 2 lines -> No message
- 1 line -> No message and no verified value
- 0 lines -> Not changing anything

And if any of the two first lines fails when converting to integer, it will assume that's the start of the message and count the non-specified votes as abstentions

Examples: Skipping hooks comment - return codes: If rc == 3 the execution of the hook will not be added to the comments, on any other case, it will add a comment with the message provided

message starts here -1 message continues

^^^ this will be parsed as only message and two abstentions

-1 message starts here 0 message continues here

^^^ this will count as CR:-1, and the rest message

0 message

^^^ This will count as CR:0 (will level any positive review) and the rest message

`hook_dispatcher.`**`print_title`**`(`*title*, *line_size=40*, *line_symbol='='*`)`

Printing the title with upper and lower lines with specified line size and symbol

Parameters

- **`title`** – title string
- **`line_size`** – line size (default: 40)
- **`line_symbol`** – line symbol (default: '=')

`hook_dispatcher.`**`rerun`**`(`*tag_val*, *to_rerun_hooks*, *args*`)`

Handles the Rerun-Hooks tag, reruns the given hooks, if all is given it will rerun all the hooks

Parameters

- **`tag_val`** – string passed to "Rerun-Hooks:" tag in comment
- **`to_rerun_hooks`** – array with the csv string passed to the tag
- **`args`** – object with all the passed arguments

:return list of hooks that will be rerun

`hook_dispatcher.`**`run_chains`**`(`*path*, *hooks*`)`

Builds the chains from the given hooks and runs them.

If a hook on a chain returns with 1 or 2, it will break that chain and no futher hooks for that chain will be executed, continuing with the next chain

Parameters

- **`path`** – Path where the hooks are located
- **`hooks`** – hook names to run

`hook_dispatcher.`**`run_hooks`**(*path*, *hooks*)

Run the given hooks form the given path

If any of the hooks returns with 1 or 2, the execution is stopped, if returns with 3, only a warning message is logged, and the execution will continue.

Parameters

- **`path`** – Path where the hooks are located
- **`hooks`** – hook names to run

Returns str Log string with the combined result of all the executed hooks

`hook_dispatcher.`**`run_only`**(*to_run_hooks*, *current_hooks*)

Handles the Run-Only tag, removes all the hooks not matching the given hooks

Parameters

- **`to_run_hooks`** – array with the csv string passed to the tag
- **`current_hooks`** – array with the current available hooks

`hook_dispatcher.`**`send_summary`**(*change_id*, *project*, *results*)

Parse all the results and generate the summary review

Parameters

- **`change_id`** – string to identify the change with, usually the commit hash or the change-id,patchset pair
- **`project`** – project name (i.e 'ovirt-engine')
- **`results`** – Dictionary with the compiled results for all the chains

It will use the lowest values for the votes, skipping the ones that are *None*, and merge all the messages into one.

For example, if it has two chains, one with CR:0 and one with CR:1, the result will be CR:0.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index