# Git-Upstream Documentation

*Release 0.12.2*

Git-Upstream Maintainers

Apr 14, 2017

# Contents

Git-upstream is an open source Python application that can be used to keep in sync with upstream open source projects. Its goal is to help manage automatically dropping carried patches when syncing with the project upstream, in a manner transparent to local developers.

It was initially developed as a tool for people who are doing active contributions to local mirrors of projects hosted using Gerrit for code review, with the intention that the local changes would be submitted to the upstream Gerrit instance (review.openstack.org for OpenStack) in the future, and would subsequent appear in the upstream mainline.

As it uses git plumbing commands, it can identify identical patches exactly the same as how `git-rebase` works, and is not limited to working with Gerrit hosted projects. It can be used with projects hosted in GitHub or any other git repo hosting software.

Online documentation:

- http://git-upstream.readthedocs.io/en/latest/

# CHAPTER 1

# Developers

Bug reports:

- https://bugs.launchpad.net/git-upstream

Repository:

- https://git.openstack.org/cgit/openstack/git-upstream

Cloning:

```
git clone https://git.openstack.org/cgit/openstack/git-upstream
```

or

```
git clone https://github.com/openstack/git-upstream
```

A virtual environment is recommended for development. For example, git-upstream may be installed from the top level directory:

```
virtualenv .venv
source .venv/bin/activate
pip install -r test-requirements.txt -e .
```

Patches are submitted via Gerrit at:

- https://review.openstack.org/

Please do not submit GitHub pull requests, they will be automatically closed.

More details on how you can contribute is available on our wiki at:

- http://docs.openstack.org/infra/manual/developers.html

# CHAPTER 2

# Writing a patch

All code submissions must be pep8 and pyflakes clean. CI will automatically reject them if they are not. The easiest way to do that is to run tox before submitting code for review in Gerrit. It will run `pep8` and `pyflakes` in the same manner as the automated test suite that will run on proposed patchsets.

# CHAPTER 3

# Unit Tests

Unit tests have been included and are in the `git_upstream/tests` folder. Many unit tests samples are included as example scenarios in our documentation to help explain how git-upstream handles various use cases. To run the unit tests, execute the command:

```
tox -e py34,py27
```

- Note: View `tox.ini` to run tests on other versions of Python, generating the documentation and additionally for any special notes on building one of the scenarios to allow direct inspection and manual execution of `git-upstream` with various scenarios.

The unit tests can in many cases be better understood as being closer to functional tests.

CHAPTER 4

# Support

The git-upstream community is found on the #git-upstream channel on chat.freenode.net

You can also join via this IRC URL or use the Freenode IRC webchat.

# CHAPTER 5

# Contents:

## Introduction

### What does git-upstream do?

git-upstream provides new git subcommands to support rebasing of local-carried patches on top of upstream repositories. It provides commands to ease the use of git for who needs to integrate big upstream projects in their environment. The operations are performed using Git commands.

**Note:** Currently git-upstream works best for projects that are maintained with Gerrit because the presence of Change-Ids allows for fully automated dropping of changes that appear upstream. Nevertheless, the code is quite modular and can be extended to use any part of commit message (e.g., other headers).

Git-upstream currently supports the following features

- **Single upstream branch import**

Your repository is tracking an upstream project and has local changes applied and no other branch is merged in. This can also be applied to tracking upstream packaging branches: *e.g.*, ubuntu/master => ubuntu/saucy-proposed/nova + local packaging changes.

- **Multi branch import (upstream branch + additional branches)**

In this case, your project tracks an upstream repository, merges in an arbitrary number of branches and applies local carried changes.

- **Re-reviewing**

Reviewing (w/ Gerrit) of all locally applied changes if desired. git-upstream creates an import branch in a manner that allows it to be fully re-reviewed or merged into master and pushed.

- **Detailed logging**

git-upstream can output to both console and log file simultaneously. Multiple log levels are supported, and these are managed separately for log file and console output. This means jobs run by Jenkins can save a detailed log file separately as an artefact while printing status information to the console if those running the jobs don't wish to have the console spammed with the details.

- **Dropping of changes that appear upstream**

Compares Change-Id's of changes applied since previous import with those that have appeared on the upstream branch since the last import point.

- **Interactive mode**

Once the list of changes to be re-applied has been determined (and those to be dropped have been pruned), the tool can open an editor (controlled by a user's git editor settings) for users to review those changes to be made and allow them to perform further operations such as re-ordering, dropping of obsolete changes, and squashing.

- **Dropping local changes**

It's always possible for local changes to be superseded by upstream changes, so when these are identified and marked as such, we should drop them.

This can also occur where a change was applied locally, modified when being upstreamed based on review feedback and the resulting differences were ported to the internal as well. While the original change will be automatically dropped, also useful to drop the additional ported changes automatically if possible, rather than have it cause conflicts.

## Installation

To install git-upstream from pypi, run:

```
pip install --user git-upstream
```

Alternatively, the current release can be installed system-wide from pypi:

```
sudo pip install git-upstream
```

Installing directly from source is possible, first clone and then install using pip:

```
git clone https://git.openstack.org/openstack/git-upstream.git
cd git-upstream
pip install .
```

Or setup.py:

```
git clone https://git.openstack.org/openstack/git-upstream.git
cd git-upstream
python setup.py install
```

Or alternatively:

```
git clone https://git.openstack.org/openstack/git-upstream.git
cd git-upstream
easy_install .
```

If you want command line completion (using tab), install the provided "bash completion" file

```
mkdir ~/bin && cp ./bash_completion/git-upstream ~/bin
echo ". ~/bin/git-upstream" >> ~/.bash_profile
```

Verify your installation.

```
pip show git-upstream
---
Name: git-upstream
Version: 0.12.1
Summary: git tool to help manage upstream repositories
Home-page: https://pypi.python.org/pypi/git-upstream
Author: Darragh Bailey
Author-email: dbailey@hpe.com
License: Apache License (2.0)
Location: /home/<username>/.local/lib/python2.7/site-packages
Requires: argcomplete, pbr, six, GitPython

git-upstream --help
usage: git-upstream [--version] [-h] [-q | -v] <command> ...

[...]
```

### Installing for Development

A virtual environment is recommended for development. For example, git-upstream may be installed from the top level directory:

```
virtualenv .venv
source .venv/bin/activate
pip install -r test-requirements.txt -e .
```

### Generating Documentation

Documentation is included in the `doc` folder. To generate docs locally execute the command:

```
tox -e docs
```

The generated documentation is then available under `doc/build/html/index.html`.

- Note: When behind a proxy it is necessary to use `TOX_TESTENV_PASSENV` to pass any proxy settings for this test to be able to check links are valid.

### Unit Tests

Unit tests have been included and are in the `git_upstream/tests` folder. Many unit tests samples are included as example scenarios in our documentation to help explain how git-upstream handles various use cases. To run the unit tests, execute the command:

```
tox -e py34,py27
```

- Note: View `tox.ini` to run tests on other versions of Python, generating the documentation and additionally for any special notes on building one of the scenarios to allow direct inspection and manual execution of `git-upstream` with various scenarios.

The unit tests can in many cases be better understood as being closer to functional tests.

### Test Coverage

To measure test coverage, execute the command:

```
tox -e cover
```

## Subcommands

### import

#### Description

Import code from specified upstream branch. Creates an import branch from the specified upstream branch, and optionally merges additional branches given as arguments. Current branch, unless overridden by the `--into` option, is used as the target branch from which a list of changes to apply onto the new import is constructed based on the specified strategy. Once complete it will merge and replace the contents of the target branch with those from the import branch, unless `--no-merge` is specified.

By default, the import branch is named according to the following format, unless overridden using `--import-branch`:

```
import/<tag-or-git-describe-commit>[-<additional-branch-git-describe-commit>]
```

For example, `import/4.0.0.0rc1-8-geaec95b` refers to an upstream branch who's latest tag is `4.0.0.0rc1`. `8` commits have been made upstream ahead of this tag, and `geaec95b` is SHA1 of the tip before import.

#### Usage

```
git upstream import [-h] [-d] [-i] [-f] [--merge] [--no-merge]
                           [-s <strategy>] [--into <branch>]
                           [--import-branch <import-branch>]
                           [<upstream-branch>] [<branches> [<branches> ...]]
```

#### Arguments

```
positional arguments:
  <upstream-branch>     Upstream branch to import. Must be specified if you
                        wish to provide additional branches.
  <branches>            Branches to additionally merge into the import branch
                        using default git merging behaviour

optional arguments:
  -h, --help            show this help message and exit
  -d, --dry-run         Only print out the list of commits that would be
                        applied.
  -i, --interactive     Let the user edit the list of commits before applying.
  -f, --force           Force overwrite of existing import branch if it
                        exists.
  --merge               Merge the resulting import branch into the target
                        branch once complete
  --no-merge            Disable merge of the resulting import branch
  -s <strategy>, --strategy <strategy>
                        Use the given strategy to re-apply locally carried
                        changes to the import branch. (default: drop)
  --into <branch>       Branch to take changes from, and replace with imported
                        branch.
  --import-branch <import-branch>
                        Name of import branch to use
```

## drop

### Description

Mark a commit as dropped. Marked commits will be skipped during the upstream rebasing process.

See also the "git upstream import" command.

### Usage

```
git upstream drop [-h] [-a <author>] <commit>
```

### Arguments

```
positional arguments:
  <commit>              Commit to be marked as dropped

optional arguments:
  -h, --help            show this help message and exit
  -a <author>, --author <author>
                        Git author for the mark
```

### Note

Commits will be marked with git notes in the namespace `refs/notes/upstream-merge`.

To list of commit id marked with a note, run `git notes --ref refs/notes/upstream-merge`.

To show a specific note run `git notes --ref refs/notes/upstream-merge show <marked commit sha1>`

As `drop` uses git notes to mark commits that have to be skipped during import, notes should be present on the cloned copy of your repository. Thus, if you are going to create notes on a system and perform the actual import on a different system, **notes must be present on the latter**.

You can push notes directly to git repository on the target system or push them in a different repository and then pull notes from your target system.

## supersede

### Description

Mark a commit as superseded by a set of change-ids. Marked commits will be skipped during the upstream rebasing process **only if all the specified change-ids are present in ‘‘<upstream-branch>‘‘ during import**. If you want to unconditionally drop a commit, use the `drop` command instead.

See also the “git upstream import” command.

### Usage

```
git upstream supersede [-h] [-f] [-u <upstream-branch>]
                       <commit> <change id> [<change id> ...]
```

### Arguments

```
positional arguments:
  <commit>                Commit to be marked as superseded
  <change id>             Change id which makes <commit> obsolete. The change id
                          must be present in <upstream-branch> to drop <commit>.
                          If more than one change id is specified, all must be
                          present in <upstream-branch> to drop <commit>

optional arguments:
  -h, --help              show this help message and exit
  -f, --force             Apply the commit mark even if one or more change ids
                          could not be found. Use this flag carefully as commits
                          will not be dropped during import command execution as
                          long as all associated change ids are present in the
                          local copy of the upstream branch
  -u <upstream-branch>, --upstream-branch <upstream-branch>
                          Search change ids values in <upstream-branch> branch
                          (default: upstream/master)
```

### Note

*This command doesn’t perform the actual drop*. Commits to be dropped during the next import, will be marked with git notes in the namespace `refs/notes/upstream-merge`. There is no need to retain notes after an import dropped the correspondent commits, of course it doesn’t harm keeping them either.

To list of commit id marked with a note, run `git notes --ref refs/notes/upstream-merge`.

To show a specific note run `git notes --ref refs/notes/upstream-merge show <marked commit sha1>`.

As `supersede` uses git notes to mark commits that have to be skipped during import, notes should be present on the cloned copy of your repository. Thus, if you are going to create notes on a system and perform the actual import on a different system, **notes must be present on the latter**. You can push notes directly to git repository on the target system or push them in a different repository and then pull notes from your target system.

## Workflows

**Note:** This guide assumes that you are using a branch named *master* to maintain your new features or bug fixes that sit on top of the upstream code of some project (probably somewhat related to OpenStack).

## Importing from upstream: using git-upstream

See installation instructions for details on installing.

### Initial import of an upstream project

To explain the usage of the git-upstream tool we are going to use a real-world (but trivial) example, by performing some sample operations on a project called `jenkins-job-builder`.

In this example, we will create a local file based Git repository to host our mirror of jenkins-job-builder. You could also use an existing internal mirror, a Github fork, etc.

Start by setting the following environment variables:

```
export REPO_NAME="jenkins-job-builder"
export INTERNAL_REMOTE="file:///tmp/jenkins-job-builder.git"
export UPSTREAM_REMOTE="https://github.com/openstack-infra/jenkins-job-builder.git"
export FIRST_IMPORT_REF="0.5.0"
```

**1) Create two empty repositories, one to serve as your working copy, and** one to serve as the remote:

```
git init --bare /tmp/${REPO_NAME}.git
git init $REPO_NAME
cd $REPO_NAME
```

2) Add your remotes

We will name it *origin* and *upstream* (for the sake of originality).

```
git remote add origin $INTERNAL_REMOTE
git remote add upstream $UPSTREAM_REMOTE
```

3) Fetch objects and refs from upstream remote

```
git fetch --all
```

4) Push refs

Push refs defined upstream to the `origin` remote (*i.e.*, the internal copy of the repository with local patches) using the string `upstream` as prefix, also pushing tags.

```
git for-each-ref refs/remotes/upstream --format "%(refname:short)" | \
  sed -e 's:\(upstream/\(.*\)\)$:\1\:refs/heads/upstream/\2:' | \
  xargs git push --tags origin
```

You may want to repeat the last two commands before starting any new feature development or a bug fix.

5) Check-out the first import commit (*e.g.*, tag or SHA1)

This will be the starting point for the internal development.

```
git checkout -b import/$FIRST_IMPORT_REF $FIRST_IMPORT_REF
```

6) Create and switch to the master branch

```
git checkout -b master
```

Now the tips of master, `$FIRST_IMPORT_REF` and `import/$FIRST_IMPORT_REF` should be pointing to the same commit.

Push local master branch to the remote origin, and make `origin master` the default when pushing commits.

```
git push -u origin master
```

### Writing your patches/features

Now start to develop new feature or fix bugs on master, as usual. For this example, we are going to change the .gitreview file in order to use a local Gerrit server.

```
sed -i 's/review\.openstack\.org/gerrit\.my\.org/' .gitreview
```

Don't forget to commit and push (after this step, you may want to use git review as usual)

```
git commit -a -m "Set .gitreview content to use internal gating infra"
git push
```

Our master (local and remote) tip should be now pointing to the last commit.

### Importing single patches from upstream

Before implementing any feature or fixing any bug (in short, before reinventing the wheel), check if someone has already implemented the required code upstream.

If not, try not to develop code only for your specific needs, be ambitious and try to develop something that could be useful for the whole community. This way you can propose your patch upstream and save yourself a lot of trouble which arise when there are many local changes to carry on the tip of upstream releases.

In this example, we tried to use our code and we found out that the job filtering isn't working! Fortunately, Antoine Musso has already fixed this bug, as we can see in the upstream repo.

```
git show --summary 2eca0d11669b55d4ab02ba609a15aa242fd80d14
commit 2eca0d11669b55d4ab02ba609a15aa242fd80d14
Author: Antoine Musso <hashar@free.fr>
Date:   Mon Jun 24 14:36:52 2013 +0200

    job filtering was not working properly

    When passing job names as arguments to 'update', the command is supposed
    to only retain this jobs.  Due to the job being a dict, the filter would
    never match and the none of the job would be updated.

    This has apparently always been broken since the feature got introduced
    in 85cf7a41.  Using job.['name'] fix it up.

    Change-Id: Icf4d5b0bb68777f7faff91ade04451d4c8501c6a
    Reviewed-on: https://review.openstack.org/34197
    Reviewed-by: Clark Boylan <clark.boylan@gmail.com>
    Approved: James E. Blair <corvus@inaugust.com>
    Reviewed-by: James E. Blair <corvus@inaugust.com>
    Tested-by: Jenkins
```

```
```

We are also interested in the following commit, which adds the Environment File Plugin (finally!).

```
git show --summary bf4524fae25c11640ef839aa422ac81bd926ca20
commit bf4524fae25c11640ef839aa422ac81bd926ca20
Author: zaro0508 <zaro0508@gmail.com>
Date:   Mon Jul 1 11:21:24 2013 -0700

    add Environment File Plugin

    This commit adds the Environment File Plugin to JJB.
    https://wiki.jenkins-ci.org/display/JENKINS/Envfile+Plugin

    Change-Id: Id35a4d6ab25b0440303da02bb91007b459979243
    Reviewed-on: https://review.openstack.org/35170
    Reviewed-by: Arnaud Fabre <fabre.arnaud@gmail.com>
    Reviewed-by: James E. Blair <corvus@inaugust.com>
    Approved: Clark Boylan <clark.boylan@gmail.com>
    Reviewed-by: Clark Boylan <clark.boylan@gmail.com>
    Tested-by: Jenkins
```

Import those changes by simply cherry-picking the two commits. Don't forget to push (review!) your changes.

```
git cherry-pick 2eca0d11669b55d4ab02ba609a15aa242fd80d14
git cherry-pick bf4524fae25c11640ef839aa422ac81bd926ca20
git push
```

### Importing new versions from upstream

Time passes and finally a new releases comes out.

```
git fetch --all
git for-each-ref refs/remotes/upstream --format "%(refname:short)" | \
  sed -e 's:\(upstream/\(.*\)\)$:\1\:refs/heads/upstream/\2:' | \
  xargs git push --tags origin
```

A lot of work has been done upstream and we need to rebase our master onto the upstream master branch. In this process we want to skip all the commits cherry-picked some days ago, where they have merged upstream.

### Running git-upstream

Identify the commit/tag/branch to import from, in this example we'll use `0.6.0` as a tag for a recent release we want to import.

Now, it is time to run git-upstream! Before doing so make sure the current branch is master

```
git checkout master
```

```
git-upstream import 0.6.0
Searching for previous import
Starting import of upstream
Successfully created import branch
Attempting to linearise previous changes
Successfully applied all locally carried changes
Merging import to requested branch 'HEAD'
```

```
Successfully finished import:
target branch: 'HEAD'
upstream branch: 'import/0.6.0'
import branch: 'import/0.6.0'
```

***No errors***, we have been lucky!

What has just happened?

git-upstream has created a new branch named `import/0.6.0-base` which tip is branched from the release tag `0.6.0`, and has rebased all changes present in our local master which were not already present in the upstream new release (`import/0.6.0-base`) onto `import/0.6.0-base`.

You can see that running the following command

```
git log --graph --oneline --all --decorate
```

For this trivial example, the only commit not present in the upstream release was about the customisation of the .gitreview file.

The default strategy git-upstream uses to find duplicate entries is exactly the same as `git-rebase`, which works for both cherry-picked and rebased commits. Additionally it also looks at Change-Id entries in commit messages where found, as these help identify patches that were changed before being accepted upstream when using Gerrit for reviews.

---

**Note:** A git commit SHA1 is generated from the following information:

- commit message
- author signature (identity + timestamp)
- committer signature (identity + timestamp)
- tree SHA1 (hierarchy of directories and files within the commit)
- list of the SHA1's of the parent commits

This prevents usage of the commit SHA1 as a method of finding duplicates. Git-upstream makes uses of git's internal patch-id to find identical changes. Git-patch-id generates an id based on the the changes made to the tree, which can be used to identify different commits with the exact same code changes as a duplicate commit.

Git-upstream's makes use of Change-Id's from Gerrit to identify additional commits that have the same intention, but are different due to changes made at the request of the upstream. The final patch being slight different cannot be matched using git-patch-id as it will return a different output to the current carried patch.

---

The local branch `import/0.6.0` now contains our local changes rebased onto the new upstream release. git-upstream has also merged this branch with the local master branch (with a custom merge strategy equivalent to the inverse of 'ours', which is not to be confused with the 'ours' option to the recursive merge strategy) to allow the normal workflow (committing/merging to master for review).

---

**Note:** The "final" merging step is not mandatory. Of course you can keep a separate branch for each new import. On one hand this strategy allows a "cleaner" history as you will always have your local changes rebased on top of the exact copy of the upstream repository. On the other hand you will be creating a new branch every time you want to import upstream code. You can customise the name of the import branch using the `--import-branch <branch name>` option.

---

In principle, you could also replace your master branch (history) with the new import branch created by git-upstream... Unfortunately there is no way to do this without requiring ad-hoc intervention on cloned copies of the repository (aka do-not-do-that(TM))

To disable automatic merging, just use the `--no-merge` flag

```
git-upstream import --no-merge import/0.6.0
```

## Handling conflicts

Of course in the real world things are much more complicated. From time to time, during import, you will get rebasing conflict (for instance due to changes from both local and upstream repository to the same piece of code).

In case of rebasing conflict, git-upstream will stop allowing the user to fix the conflict.

```
git-upstream import import/0.5.0 --into master
Searching for previous import
Starting import of upstream
Successfully created import branch
Attempting to linearise previous changes
ERROR   : Rebase failed, will need user intervention to resolve.
error: could not apply f9b4fca... Fixup for openstack review
When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --abort".
Could not apply f9b4fca... Fixup for openstack review
Import cancelled
```

Let’s find out why git-upstream failed and let’s try to continue the rebasing manually.

```
git status
# HEAD detached from 8e6b9e9
# You are currently rebasing branch 'import/0.5.0' on '8e6b9e9'.
#   (fix conflicts and then run "git rebase --continue")
#   (use "git rebase --skip" to skip this patch)
#   (use "git rebase --abort" to check out the original branch)
#
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
# both modified:      jenkins_jobs/cmd.py
# both modified:      jenkins_jobs/modules/hipchat_notif.py
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Depending on the type of conflict, you will could:

- drop the local change

  Issuing `git rebase --skip`

- edit conflicting code

  Change conflicting code in order to accommodate local changes to the new upstream code. You can later resume rebasing process issuing `git rebase --continue`

By default git-upstream should automatically be re-called as the final step of the rebasing process. Unless however you have used the option `--no-merge` as an argument to the import command.

In such cases, where you wish to subsequently finish, the `import` subcommand provides a `--finish` option to assist:

```
git checkout master
git upstream import --finish --import-branch import/0.5.0 0.5.0
```

### Integration with Gerrit

You may want to use review with Gerrit the output of git-upstream, in order to perform tests, gating, etc.

You have 2 options for doing that:

#### Re-review every new commit

In this case we want to review every new commit (since the last import). In order to do so, use the `--no-merge` flag of git-upstream import command, and:

```
git checkout import-xxxxx
git push gerrit import-xxxxx-base:import-xxxxx
git review import-xxxxx
```

If there is more than one new commit, git-review will ask to confirm the submission of multiple changes.

#### Re-review only the final merge commit

This would be possible by using the `--import-branch` option of import command and **pushing directly** (*i.e.*: bypassing Gerrit) the new branch to the local repo. For instance:

```
TIMESTAMP=$(date +"%Y%m%d%H%M%s")
git upstream import --import-branch "import/import-$TIMESTAMP" upstream/master
git push gerrit import/import-$TIMESTAMP:import/import-$TIMESTAMP
```

Then, create a valid `Change-Id` for the merge commit

```
git commit --amend -C HEAD --no-edit
```

Locally, git-review will still complain about the presence of N+M commits which would be committed BUT on the remote side all those commits will be recognised as already present in one of the two branch involved in the merge.

```
git review -R -y master
```

## Examples

# CHAPTER 6

# Indices and tables

- genindex
- search