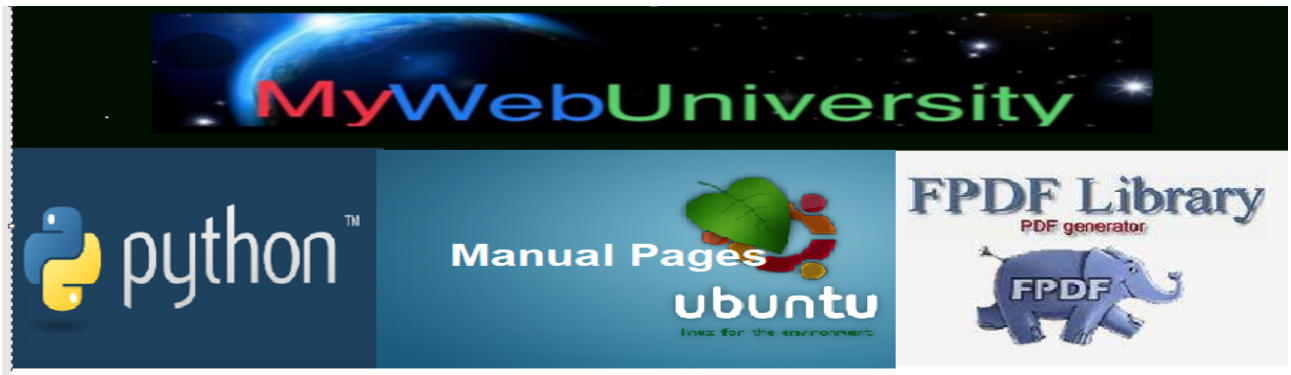

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'git-push.1'

$ man git-push.1

GIT-PUSH(1) Git Manual GIT-PUSH(1)

NAME

git-push - Update remote refs along with associated objects

SYNOPSIS

git push [--all | --branches | --mirror | --tags] [--follow-tags] [--atomic] [-n | --dry-run] [--receive-pack=<git-receive-pack>]
[--repo=<repository>] [-f | --force] [-d | --delete] [--prune] [-q | --quiet] [-v | --verbose]
[-u | --set-upstream] [-o <string> | --push-option=<string>]
[--[no-]signed | --signed=(true|false|if-asked)]
[--force-with-lease[=<refname>[:<expect>]] [--force-if-includes]]
[--no-verify] [<repository> [<refspec>...]]

DESCRIPTION

Updates one or more branches, tags, or other references in a remote repository from your local repository, and sends all necessary data that isn?t already on the remote.

The simplest way to push is git push <remote> <branch>. git push origin main will push the local main branch to the main branch on the remote named origin.

The <repository> argument defaults to the upstream for the current branch, or origin if there?s no configured upstream.

To decide which branches, tags, or other refs to push, Git uses (in order of precedence):

1. The <refspec> argument(s) (for example main in git push origin main) or the --all, --mirror, or --tags options
2. The remote.<name>.push configuration for the repository being pushed to
3. The push.default configuration. The default is push.default=simple, which will push to a branch with the same name as the current branch. See the CONFIGURATION section below for

more on push.default.

git push may fail if you haven?t set an upstream for the current branch, depending on what push.default is set to. See the UPSTREAM BRANCHES section below for more on how to set and use upstreams.

You can make interesting things happen to a repository every time you push into it, by setting up hooks there. See documentation for git-receive-pack(1).

OPTIONS

<repository>

The "remote" repository that is the destination of a push operation. This parameter can be either a URL (see the section GIT URLS below) or the name of a remote (see the section REMOTES below).

<refspec>...

Specify what destination ref to update with what source object.

The format for a refspec is [+]<src>[:<dst>], for example main, main:other, or HEAD^:refs/heads/main.

The <src> is often the name of the local branch to push, but it can be any arbitrary "SHA-1 expression" (see gitrevisions(7)).

The <dst> determines what ref to update on the remote side. It must be the name of a branch, tag, or other ref, not an arbitrary expression.

The + is optional and does the same thing as --force.

You can write a refspec using the fully expanded form (for example refs/heads/main:refs/heads/main) which specifies the exact source and destination, or with a shorter form (for example main or main:other). Here are the rules for how refspecs are expanded, as well as various other special refspec forms:

? <src> without a :<dst> means to update the same ref as the <src>, unless the remote.<repository>.push configuration specifies a different <dst>. For example, if main is a branch, then the refspec main expands to main:refs/heads/main.

? If <dst> unambiguously refers to a ref on the <repository> remote, then expand it to that ref. For example, if v1.0 is a tag on the remote, then HEAD:v1.0 expands to HEAD:refs/tags/v1.0.

? If <src> resolves to a ref starting with refs/heads/ or refs/tags/, then prepend that to <dst>. For example, if main is a branch, then main:other expands to main:refs/heads/other

? The special refspec : (or +: to allow non-fast-forward updates) directs Git to push "matching" branches: for every branch that exists on the local side, the remote side is updated if a branch of the same name already exists on the remote side.

? <src> may contain a * to indicate a simple pattern match. This works like a glob that matches any ref matching the pattern. There must be only one * in both the <src> and <dst>. It will map refs to the destination by replacing the * with the contents matched from the source. For example, refs/heads/*:refs/heads/* will push all branches.

? A refspec starting with ^ is a negative refspec. This specifies refs to exclude. A ref will be considered to match if it matches at least one positive refspec, and does not match any negative refspec. Negative refspecs can be pattern refspecs. They must only contain a <src>. Fully spelled out hex object names are also not supported. For example, git push origin 'refs/heads/*' '^refs/heads/dev-*' will push all branches except for those starting with dev-

? If <src> is empty, it deletes the <dst> ref from the remote repository. For example, git push origin :dev will delete the dev branch.

? tag <tag> expands to refs/tags/<tag>:refs/tags/<tag>. This is technically a special syntax for git push and not a refspec, since in git push origin tag v1.0 the arguments tag and v1.0 are separate.

? If the refspec can?t be expanded unambiguously, error out with an error indicating what was tried, and depending on the advice.pushUnqualifiedRefname configuration (see git-config(1)) suggest what refs/ namespace you may have wanted to push to.

Not all updates are allowed: see PUSH RULES below for the details.

--all, --branches

Push all branches (i.e. refs under refs/heads/); cannot be used with other <refspec>.

--prune

Remove remote branches that don?t have a local counterpart. For example a remote branch tmp will be removed if a local branch with the same name doesn?t exist any more. This also respects refspecs, e.g. git push --prune remote refs/heads/*:refs/tmp/* would make sure that remote refs/tmp/foo will be removed if refs/heads/foo doesn?t exist.

--mirror

Instead of naming each ref to push, specifies that all refs under refs/ (which includes but is not limited to refs/heads/, refs/remotes/, and refs/tags/) be mirrored to the

remote repository. Newly created local refs will be pushed to the remote end, locally updated refs will be force updated on the remote end, and deleted refs will be removed from the remote end. This is the default if the configuration option remote.<remote>.mirror is set.

-n, --dry-run

Do everything except actually send the updates.

--porcelain

Produce machine-readable output. The output status line for each ref will be tab-separated and sent to stdout instead of stderr. The full symbolic names of the refs will be given.

-d, --delete

All listed refs are deleted from the remote repository. This is the same as prefixing all refs with a colon.

--tags

All refs under refs/tags are pushed, in addition to refspecs explicitly listed on the command line.

--follow-tags

Push all the refs that would be pushed without this option, and also push annotated tags in refs/tags that are missing from the remote but are pointing at commit-ish that are reachable from the refs being pushed. This can also be specified with configuration variable push.followTags. For more information, see push.followTags in git-config(1).

--signed, --no-signed, --signed=(true|false|if-asked)

GPG-sign the push request to update refs on the receiving side, to allow it to be checked by the hooks and/or be logged. Possible values are:

false, --no-signed

no signing will be attempted.

true, --signed

the push will fail if the server does not support signed pushes.

if-asked

sign if and only if the server supports signed pushes. The push will also fail if the actual call to gpg --sign fails. See git-receive-pack(1) for the details on the receiving end.

--atomic, --no-atomic

Use an atomic transaction on the remote side if available. Either all refs are updated, or on error, no refs are updated. If the server does not support atomic pushes the push will fail.

-o <option>, --push-option=<option>

Transmit the given string to the server, which passes them to the pre-receive as well as the post-receive hook. The given string must not contain a NUL or LF character. When multiple --push-option=<option> are given, they are all sent to the other side in the order listed on the command line. When no --push-option=<option> is given from the command line, the values of configuration variable push.pushOption are used instead.

--receive-pack=<git-receive-pack>, --exec=<git-receive-pack>

Path to the git-receive-pack program on the remote end. Sometimes useful when pushing to a remote repository over ssh, and you do not have the program in a directory on the default $PATH.

--force-with-lease, --no-force-with-lease, --force-with-lease=<refname>, --force-with-lease=<refname>:<expect>

Usually, git push refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it.

This option overrides this restriction if the current value of the remote ref is the expected value. git push fails otherwise.

Imagine that you have to rebase what you have already published. You will have to bypass the "must fast-forward" rule in order to replace the history you originally published with the rebased history. If somebody else built on top of your original history while you are rebasing, the tip of the branch at the remote may advance with their commit, and blindly pushing with --force will lose their work.

This option allows you to say that you expect the history you are updating is what you rebased and want to replace. If the remote ref still points at the commit you specified, you can be sure that no other people did anything to the ref. It is like taking a "lease" on the ref without explicitly locking it, and the remote ref is updated only if the "lease" is still valid.

--force-with-lease alone, without specifying the details, will protect all remote refs that are going to be updated by requiring their current value to be the same as the remote-tracking branch we have for them.

--force-with-lease=<refname>, without specifying the expected value, will protect

<refname> (alone), if it is going to be updated, by requiring its current value to be the same as the remote-tracking branch we have for it.

--force-with-lease=<refname>:<expect> will protect <refname> (alone), if it is going to be updated, by requiring its current value to be the same as the specified value <expect> (which is allowed to be different from the remote-tracking branch we have for the refname, or we do not even have to have such a remote-tracking branch when this form is used). If <expect> is the empty string, then the named ref must not already exist.

Note that all forms other than --force-with-lease=<refname>:<expect> that specifies the expected current value of the ref explicitly are still experimental and their semantics may change as we gain experience with this feature.

--no-force-with-lease will cancel all the previous --force-with-lease on the command line.

A general note on safety: supplying this option without an expected value, i.e. as --force-with-lease or --force-with-lease=<refname> interacts very badly with anything that implicitly runs git fetch on the remote to be pushed to in the background, e.g. git fetch origin on your repository in a cronjob.

The protection it offers over --force is ensuring that subsequent changes your work wasn?t based on aren?t clobbered, but this is trivially defeated if some background process is updating refs in the background. We don?t have anything except the remote tracking info to go by as a heuristic for refs you?re expected to have seen & are willing to clobber.

If your editor or some other system is running git fetch in the background for you a way to mitigate this is to simply set up another remote:

```
git remote add origin-push $(git config remote.origin.url)
git fetch origin-push
```

Now when the background process runs git fetch origin the references on origin-push won?t be updated, and thus commands like:

```
git push --force-with-lease origin-push
```

Will fail unless you manually run git fetch origin-push. This method is of course entirely defeated by something that runs git fetch --all, in that case you?d need to either disable it or do something more tedious like:

```
git fetch             # update 'master' from remote
git tag base master   # mark our base point
```

git rebase -i master # rewrite some commits

git push --force-with-lease=master:base master:master

I.e. create a base tag for versions of the upstream code that you?ve seen and are willing to overwrite, then rewrite history, and finally force push changes to master if the remote version is still at base, regardless of what your local remotes/origin/master has been updated to in the background.

Alternatively, specifying --force-if-includes as an ancillary option along with --force-with-lease[=<refname>] (i.e., without saying what exact commit the ref on the remote side must be pointing at, or which refs on the remote side are being protected) at the time of "push" will verify if updates from the remote-tracking refs that may have been implicitly updated in the background are integrated locally before allowing a forced update.

-f, --force

Usually, git push will refuse to update a branch that is not an ancestor of the commit being pushed.

This flag disables that check, the other safety checks in PUSH RULES below, and the checks in --force-with-lease. It can cause the remote repository to lose commits; use it with care.

Note that --force applies to all the refs that are pushed, hence using it with push.default set to matching or with multiple push destinations configured with remote.<name>.push may overwrite refs other than the current branch (including local refs that are strictly behind their remote counterpart). To force a push to only one branch, use a + in front of the refspec to push (e.g git push origin +master to force a push to the master branch). See the <refspec>... section above for details.

--force-if-includes, --no-force-if-includes

Force an update only if the tip of the remote-tracking ref has been integrated locally.

This option enables a check that verifies if the tip of the remote-tracking ref is reachable from one of the "reflog" entries of the local branch based in it for a rewrite. The check ensures that any updates from the remote have been incorporated locally by rejecting the forced update if that is not the case.

If the option is passed without specifying --force-with-lease, or specified along with --force-with-lease=<refname>:<expect>, it is a "no-op".

Specifying --no-force-if-includes disables this behavior.

--repo=<repository>

This option is equivalent to the <repository> argument. If both are specified, the command-line argument takes precedence.

-u, --set-upstream

For every branch that is up to date or successfully pushed, add upstream (tracking) reference, used by argument-less git-pull(1) and other commands. For more information, see branch.<name>.merge in git-config(1).

--thin, --no-thin

These options are passed to git-send-pack(1). A thin transfer significantly reduces the amount of sent data when the sender and receiver share many of the same objects in common. The default is --thin.

-q, --quiet

Suppress all output, including the listing of updated refs, unless an error occurs. Progress is not reported to the standard error stream.

-v, --verbose

Run verbosely.

--progress

Progress status is reported on the standard error stream by default when it is attached to a terminal, unless -q is specified. This flag forces progress status even if the standard error stream is not directed to a terminal.

--no-recurse-submodules, --recurse-submodules=(check|on-demand|only|no)

May be used to make sure all submodule commits used by the revisions to be pushed are available on a remote-tracking branch. Possible values are:

check

Git will verify that all submodule commits that changed in the revisions to be pushed are available on at least one remote of the submodule. If any commits are missing the push will be aborted and exit with non-zero status.

on-demand

all submodules that changed in the revisions to be pushed will be pushed. If on-demand was not able to push all necessary revisions it will also be aborted and exit with non-zero status.

only

all submodules will be pushed while the superproject is left unpushed.

no

override the push.recurseSubmodules configuration variable when no submodule recursion is required. Similar to using --no-recurse-submodules.

When using on-demand or only, if a submodule has a push.recurseSubmodules=(on-demand|only) or submodule.recurse configuration, further recursion will occur. In this case, only is treated as on-demand.

--verify, --no-verify

Toggle the pre-push hook (see githooks(5)). The default is --verify, giving the hook a chance to prevent the push. With --no-verify, the hook is bypassed completely.

-4, --ipv4

Use IPv4 addresses only, ignoring IPv6 addresses.

-6, --ipv6

Use IPv6 addresses only, ignoring IPv4 addresses.

GIT URLS

In general, URLs contain information about the transport protocol, the address of the remote server, and the path to the repository. Depending on the transport protocol, some of this information may be absent.

Git supports ssh, git, http, and https protocols (in addition, ftp and ftps can be used for fetching, but this is inefficient and deprecated; do not use them).

The native transport (i.e. git:// URL) does no authentication and should be used with caution on unsecured networks.

The following syntaxes may be used with them:

? ssh://[<user>@]<host>[:<port>]/<path-to-git-repo>

? git://<host>[:<port>]/<path-to-git-repo>

? http[s]://<host>[:<port>]/<path-to-git-repo>

? ftp[s]://<host>[:<port>]/<path-to-git-repo>

An alternative scp-like syntax may also be used with the ssh protocol:

? [<user>@]<host>:/<path-to-git-repo>

This syntax is only recognized if there are no slashes before the first colon. This helps differentiate a local path that contains a colon. For example the local path foo:bar could be specified as an absolute path or ./foo:bar to avoid being misinterpreted as an ssh url.

The ssh and git protocols additionally support ~<username> expansion:

? ssh://[<user>@]<host>[:<port>]/~<user>/<path-to-git-repo>

? git://<host>[:<port>]/~<user>/<path-to-git-repo>

? [<user>@]<host>:~<user>/<path-to-git-repo>

For local repositories, also supported by Git natively, the following syntaxes may be used:

? /path/to/repo.git/

? file:///path/to/repo.git/

These two syntaxes are mostly equivalent, except when cloning, when the former implies --local option. See git-clone(1) for details.

git clone, git fetch and git pull, but not git push, will also accept a suitable bundle file. See git-bundle(1).

When Git doesn?t know how to handle a certain transport protocol, it attempts to use the remote-<transport> remote helper, if one exists. To explicitly request a remote helper, the following syntax may be used:

? <transport>::<address>

where <address> may be a path, a server and path, or an arbitrary URL-like string recognized by the specific remote helper being invoked. See gitremote-helpers(7) for details.

If there are a large number of similarly-named remote repositories and you want to use a different format for them (such that the URLs you use will be rewritten into URLs that work), you can create a configuration section of the form:

```
[url "<actual-url-base>"]
    insteadOf = <other-url-base>
```

For example, with this:

```
[url "git://git.host.xz/"]
    insteadOf = host.xz:/path/to/
    insteadOf = work:
```

a URL like "work:repo.git" or like "host.xz:/path/to/repo.git" will be rewritten in any context that takes a URL to be "git://git.host.xz/repo.git".

If you want to rewrite URLs for push only, you can create a configuration section of the form:

```
[url "<actual-url-base>"]
    pushInsteadOf = <other-url-base>
```

For example, with this:

```
[url "ssh://example.org/"]
    pushInsteadOf = git://example.org/
```

a URL like "git://example.org/path/to/repo.git" will be rewritten to "ssh://example.org/path/to/repo.git" for pushes, but pulls will still use the original URL.

REMOTES

The name of one of the following can be used instead of a URL as <repository> argument:

? a remote in the Git configuration file: $GIT_DIR/config,

? a file in the $GIT_DIR/remotes directory, or

? a file in the $GIT_DIR/branches directory.

All of these also allow you to omit the refspec from the command line because they each contain a refspec which git will use by default.

Named remote in configuration file

You can choose to provide the name of a remote which you had previously configured using git-remote(1), git-config(1) or even by a manual edit to the $GIT_DIR/config file. The URL of this remote will be used to access the repository. The refspec of this remote will be used by default when you do not provide a refspec on the command line. The entry in the config file would appear like this:

```
[remote "<name>"]
    url = <URL>
    pushurl = <pushurl>
    push = <refspec>
    fetch = <refspec>
```

The <pushurl> is used for pushes only. It is optional and defaults to <URL>. Pushing to a remote affects all defined pushurls or all defined urls if no pushurls are defined. Fetch, however, will only fetch from the first defined url if multiple urls are defined.

Named file in $GIT_DIR/remotes

You can choose to provide the name of a file in $GIT_DIR/remotes. The URL in this file will be used to access the repository. The refspec in this file will be used as default when you do not provide a refspec on the command line. This file should have the following format:

```
URL: one of the above URL formats
Push: <refspec>
Pull: <refspec>
```

Push: lines are used by git push and Pull: lines are used by git pull and git fetch. Multiple Push: and Pull: lines may be specified for additional branch mappings.

Named file in $GIT_DIR/branches

You can choose to provide the name of a file in $GIT_DIR/branches. The URL in this file will be used to access the repository. This file should have the following format:

<URL>#<head>

<URL> is required; #<head> is optional.

Depending on the operation, git will use one of the following refspecs, if you don?t provide one on the command line. <branch> is the name of this file in $GIT_DIR/branches and <head> defaults to master.

git fetch uses:

refs/heads/<head>:refs/heads/<branch>

git push uses:

HEAD:refs/heads/<head>

UPSTREAM BRANCHES

Branches in Git can optionally have an upstream remote branch. Git defaults to using the upstream branch for remote operations, for example:

? It?s the default for git pull or git fetch with no arguments.

? It?s the default for git push with no arguments, with some exceptions. For example, you can use the branch.<name>.pushRemote option to push to a different remote than you pull from, and by default with push.default=simple the upstream branch you configure must have the same name.

? Various commands, including git checkout and git status, will show you how many commits have been added to your current branch and the upstream since you forked from it, for example "Your branch and origin/main have diverged, and have 2 and 3 different commits each respectively".

The upstream is stored in .git/config, in the "remote" and "merge" fields. For example, if main's upstream is origin/main:

```
[branch "main"]
  remote = origin
  merge = refs/heads/main
```

You can set an upstream branch explicitly with git push --set-upstream <remote> <branch> but Git will often automatically set the upstream for you, for example:

? When you clone a repository, Git will automatically set the upstream for the default branch.

? If you have the push.autoSetupRemote configuration option set, git push will

automatically set the upstream the first time you push a branch.

? Checking out a remote-tracking branch with git checkout <branch> will automatically create a local branch with that name and set the upstream to the remote branch.

Note

Upstream branches are sometimes referred to as "tracking information", as in "set the branch?s tracking information".

OUTPUT

The output of "git push" depends on the transport method used; this section describes the output when pushing over the Git protocol (either locally or via ssh).

The status of the push is output in tabular form, with each line representing the status of a single ref. Each line is of the form:

<flag> <summary> <from> -> <to> (<reason>)

If --porcelain is used, then each line of the output is of the form:

<flag> \t <from>:<to> \t <summary> (<reason>)

The status of up-to-date refs is shown only if --porcelain or --verbose option is used.

<flag>

A single character indicating the status of the ref:

(space)

for a successfully pushed fast-forward;

+

for a successful forced update;

-

for a successfully deleted ref;

*

for a successfully pushed new ref;

!

for a ref that was rejected or failed to push; and

=

for a ref that was up to date and did not need pushing.

<summary>

For a successfully pushed ref, the summary shows the old and new values of the ref in a form suitable for using as an argument to git log (this is <old>..<new> in most cases, and <old>...<new> for forced non-fast-forward updates).

For a failed update, more details are given:

rejected

Git did not try to send the ref at all, typically because it is not a fast-forward and you did not force the update.

remote rejected

The remote end refused the update. Usually caused by a hook on the remote side, or because the remote repository has one of the following safety options in effect: receive.denyCurrentBranch (for pushes to the checked out branch), receive.denyNonFastForwards (for forced non-fast-forward updates), receive.denyDeletes or receive.denyDeleteCurrent. See git-config(1).

remote failure

The remote end did not report the successful update of the ref, perhaps because of a temporary error on the remote side, a break in the network connection, or other transient error.

from

The name of the local ref being pushed, minus its refs/<type>/ prefix. In the case of deletion, the name of the local ref is omitted.

to

The name of the remote ref being updated, minus its refs/<type>/ prefix.

reason

A human-readable explanation. In the case of successfully pushed refs, no explanation is needed. For a failed ref, the reason for failure is described.

PUSH RULES

As a safety feature, the git push command only allows certain kinds of updates to prevent you from accidentally losing data on the remote.

Because branches and tags are intended to be used differently, the safety rules for pushing to a branch are different from the rules for pushing to a tag. In the following rules "update" means any modifications except deletions and creations. Deletions and creations are always allowed, except when forbidden by configuration or hooks.

1. If the push destination is a branch (refs/heads/*): only fast-forward updates are allowed, which means the destination must be an ancestor of the source commit. The source must be a commit.
2. If the push destination is a tag (refs/tags/*): all updates will be rejected. The source

can be any object.

3. If the push destination is not a branch or tag:

? If the source is a tree or blob object, any updates will be rejected

? If the source is a tag or commit object, any fast-forward update is allowed, even in cases where what?s being fast-forwarded is not a commit, but a tag object which happens to point to a new commit which is a fast-forward of the commit the last tag (or commit) it?s replacing. Replacing a tag with an entirely different tag is also allowed, if it points to the same commit, as well as pushing a peeled tag, i.e. pushing the commit that existing tag object points to, or a new tag object which an existing commit points to.

You can override these rules by passing --force or by adding the optional leading + to a refspec. The only exceptions are that no amount of forcing will make a branch accept a non-commit object, and forcing won?t make the remote repository accept a push that it?s configured to deny.

Hooks and configuration can also override or amend these rules, see e.g. receive.denyNonFastForwards and receive.denyDeletes in git-config(1) and pre-receive and update in githooks(5).

NOTE ABOUT FAST-FORWARDS

When an update changes a branch (or more in general, a ref) that used to point at commit A to point at another commit B, it is called a fast-forward update if and only if B is a descendant of A.

In a fast-forward update from A to B, the set of commits that the original commit A built on top of is a subset of the commits the new commit B builds on top of. Hence, it does not lose any history.

In contrast, a non-fast-forward update will lose history. For example, suppose you and somebody else started at the same commit X, and you built a history leading to commit B while the other person built a history leading to commit A. The history looks like this:

```
      B
     /
---X---A
```

Further suppose that the other person already pushed changes leading to A back to the original repository from which you two obtained the original commit X.

The push done by the other person updated the branch that used to point at commit X to point

at commit A. It is a fast-forward.

But if you try to push, you will attempt to update the branch (that now points at A) with commit B. This does not fast-forward. If you did so, the changes introduced by commit A will be lost, because everybody will now start building on top of B.

The command by default does not allow an update that is not a fast-forward to prevent such loss of history.

If you do not want to lose your work (history from X to B) or the work by the other person (history from X to A), you would need to first fetch the history from the repository, create a history that contains changes done by both parties, and push the result back.

You can perform "git pull", resolve potential conflicts, and "git push" the result. A "git pull" will create a merge commit C between commits A and B.

```
    B---C
   /   /
---X---A
```

Updating A with the resulting merge commit will fast-forward and your push will be accepted.

Alternatively, you can rebase your change between X and B on top of A, with git pull --rebase, and push the result back. The rebase will create a new commit D that builds the change between X and B on top of A.

```
    B   D
   /   /
---X---A
```

Again, updating A with this commit will fast-forward and your push will be accepted.

There is another common situation where you may encounter non-fast-forward rejection when you try to push, and it is possible even when you are pushing into a repository nobody else pushes into. After you push commit A yourself (in the first picture in this section), replace it with git commit --amend to produce commit B, and you try to push it out, because forgot that you have pushed A out already. In such a case, and only if you are certain that nobody in the meantime fetched your earlier commit A (and started building on top of it), you can run git push --force to overwrite it. In other words, git push --force is a method reserved for a case where you do mean to lose history.

EXAMPLES

git push

Works like git push <remote>, where <remote> is the current branch?s remote (or origin,

if no remote is configured for the current branch).

git push origin

Without additional configuration, pushes the current branch to the configured upstream (branch.<name>.merge configuration variable) if it has the same name as the current branch, and errors out without pushing otherwise.

The default behavior of this command when no <refspec> is given can be configured by setting the push option of the remote, or the push.default configuration variable.

For example, to default to pushing only the current branch to origin use git config remote.origin.push HEAD. Any valid <refspec> (like the ones in the examples below) can be configured as the default for git push origin.

git push origin :

Push "matching" branches to origin. See <refspec> in the OPTIONS section above for a description of "matching" branches.

git push origin master

Find a ref that matches master in the source repository (most likely, it would find refs/heads/master), and update the same ref (e.g. refs/heads/master) in origin repository with it. If master did not exist remotely, it would be created.

git push origin HEAD

A handy way to push the current branch to the same name on the remote.

git push mothership master:satellite/master dev:satellite/dev

Use the source ref that matches master (e.g. refs/heads/master) to update the ref that matches satellite/master (most probably refs/remotes/satellite/master) in the mothership repository; do the same for dev and satellite/dev.

See the section describing <refspec>... above for a discussion of the matching semantics.

This is to emulate git fetch run on the mothership using git push that is run in the opposite direction in order to integrate the work done on satellite, and is often necessary when you can only make connection in one way (i.e. satellite can ssh into mothership but mothership cannot initiate connection to satellite because the latter is behind a firewall or does not run sshd).

After running this git push on the satellite machine, you would ssh into the mothership and run git merge there to complete the emulation of git pull that were run on mothership to pull changes made on satellite.

git push origin HEAD:master

Push the current branch to the remote ref matching master in the origin repository. This form is convenient to push the current branch without thinking about its local name.

git push origin master:refs/heads/experimental

Create the branch experimental in the origin repository by copying the current master branch. This form is only needed to create a new branch or tag in the remote repository when the local name and the remote name are different; otherwise, the ref name on its own will work.

git push origin :experimental

Find a ref that matches experimental in the origin repository (e.g. refs/heads/experimental), and delete it.

git push origin +dev:master

Update the origin repository?s master branch with the dev branch, allowing non-fast-forward updates. This can leave unreferenced commits dangling in the origin repository. Consider the following situation, where a fast-forward is not possible:

```
o---o---o---A---B  origin/master
         \
          X---Y---Z  dev
```

The above command would change the origin repository to

```
          A---B  (unnamed branch)
         /
o---o---o---X---Y---Z  master
```

Commits A and B would no longer belong to a branch with a symbolic name, and so would be unreachable. As such, these commits would be removed by a git gc command on the origin repository.

SECURITY

The fetch and push protocols are not designed to prevent one side from stealing data from the other repository that was not intended to be shared. If you have private data that you need to protect from a malicious peer, your best option is to store it in another repository. This applies to both clients and servers. In particular, namespaces on a server are not effective for read access control; you should only grant read access to a namespace to clients that you would trust with read access to the entire repository.

The known attack vectors are as follows:

1. The victim sends "have" lines advertising the IDs of objects it has that are not explicitly intended to be shared but can be used to optimize the transfer if the peer also has them. The attacker chooses an object ID X to steal and sends a ref to X, but isn?t required to send the content of X because the victim already has it. Now the victim believes that the attacker has X, and it sends the content of X back to the attacker later. (This attack is most straightforward for a client to perform on a server, by creating a ref to X in the namespace the client has access to and then fetching it. The most likely way for a server to perform it on a client is to "merge" X into a public branch and hope that the user does additional work on this branch and pushes it back to the server without noticing the merge.)
2. As in #1, the attacker chooses an object ID X to steal. The victim sends an object Y that the attacker already has, and the attacker falsely claims to have X and not Y, so the victim sends Y as a delta against X. The delta reveals regions of X that are similar to Y to the attacker.

CONFIGURATION

Everything below this line in this section is selectively included from the git-config(1) documentation. The content is the same as what?s found there:

push.autoSetupRemote

If set to true assume --set-upstream on default push when no upstream tracking exists for the current branch; this option takes effect with push.default options simple, upstream, and current. It is useful if by default you want new branches to be pushed to the default remote (like the behavior of push.default=current) and you also want the upstream tracking to be set. Workflows most likely to benefit from this option are simple central workflows where all branches are expected to have the same name on the remote.

push.default

Defines the action git push should take if no refspec is given (whether from the command-line, config, or elsewhere). Different values are well-suited for specific workflows; for instance, in a purely central workflow (i.e. the fetch source is equal to the push destination), upstream is probably what you want. Possible values are:

nothing

do not push anything (error out) unless a refspec is given. This is primarily meant for people who want to avoid mistakes by always being explicit.

current

push the current branch to update a branch with the same name on the receiving end. Works in both central and non-central workflows.

upstream

push the current branch back to the branch whose changes are usually integrated into the current branch (which is called @{upstream}). This mode only makes sense if you are pushing to the same repository you would normally pull from (i.e. central workflow).

tracking

this is a deprecated synonym for upstream.

simple

push the current branch with the same name on the remote.

If you are working on a centralized workflow (pushing to the same repository you pull from, which is typically origin), then you need to configure an upstream branch with the same name.

This mode is the default since Git 2.0, and is the safest option suited for beginners.

matching

push all branches having the same name on both ends. This makes the repository you are pushing to remember the set of branches that will be pushed out (e.g. if you always push maint and master there and no other branches, the repository you push to will have these two branches, and your local maint and master will be pushed there).

To use this mode effectively, you have to make sure all the branches you would push out are ready to be pushed out before running git push, as the whole point of this mode is to allow you to push all of the branches in one go. If you usually finish work on only one branch and push out the result, while other branches are unfinished, this mode is not for you. Also this mode is not suitable for pushing into a shared central repository, as other people may add new branches there, or update the tip of existing branches outside your control.

This used to be the default, but not since Git 2.0 (simple is the new default).

push.followTags

If set to true, enable --follow-tags option by default. You may override this configuration at time of push by specifying --no-follow-tags.

push.gpgSign

May be set to a boolean value, or the string if-asked. A true value causes all pushes to be GPG signed, as if --signed is passed to git-push(1). The string if-asked causes pushes to be signed if the server supports it, as if --signed=if-asked is passed to git push. A false value may override a value from a lower-priority config file. An explicit command-line flag always overrides this config option.

push.pushOption

When no --push-option=<option> argument is given from the command line, git push behaves as if each <option> of this variable is given as --push-option=<option>.

This is a multi-valued variable, and an empty value can be used in a higher priority configuration file (e.g. .git/config in a repository) to clear the values inherited from a lower priority configuration files (e.g. $HOME/.gitconfig).

Example:

/etc/gitconfig

push.pushoption = a

push.pushoption = b

~/.gitconfig

push.pushoption = c

repo/.git/config

push.pushoption =

push.pushoption = b

This will result in only b (a and c are cleared).

push.recurseSubmodules

May be check, on-demand, only, or no, with the same behavior as that of push --recurse-submodules. If not set, no is used by default, unless submodule.recurse is set (in which case a true value means on-demand).

push.useForceIfIncludes

If set to true, it is equivalent to specifying --force-if-includes as an option to git-push(1) in the command line. Adding --no-force-if-includes at the time of push overrides this configuration setting.

push.negotiate

If set to true, attempt to reduce the size of the packfile sent by rounds of negotiation in which the client and the server attempt to find commits in common. If false, Git will

rely solely on the server?s ref advertisement to find commits in common.

push.useBitmaps

If set to false, disable use of bitmaps for git push even if pack.useBitmaps is true, without preventing other git operations from using bitmaps. Default is true.

GIT

Part of the git(1) suite