--all
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).

--[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. If false or
--no-signed, no signing will be attempted. If true or --signed, the push will fail if the server does not support signed pushes. If set to

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.

--[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.

--[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 her commit, and blindly pushing with --force will lose her work.

-f, --force
Usually, the command refuses to update a remote ref that is not an ancestor of the local ref used to overwrite it. Also, when
--force-with-lease option is used, the command refuses to update a remote ref whose current value does not match what is expected.

This flag disables these checks, and 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.*.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.

--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).

--[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.

--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).

--[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. If check is used
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. If on-demand is used 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. If only is used all submodules will be recursively pushed while the superproject is left unpushed. A value of no or using
--no-recurse-submodules can be used to override the push.recurseSubmodules configuration variable when no submodule recursion is required.

--[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.

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 (remote.origin.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).

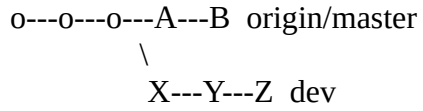
After running this git push on the satellite machine, you would ssh into the mothership and run git merge there to complete the emulation of
git pull that were run on mothership to pull changes made on satellite.

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.

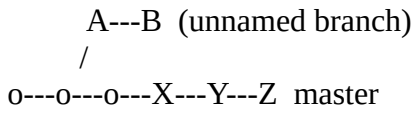
git push origin master:refs/heads/experimental
Create the branch experimental in the origin repository by copying the current master branch. This form is only needed to create a new branch
or tag in the remote repository when the local name and the remote name are different; otherwise, the ref name on 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:




The above command would change the origin repository to




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.

GIT-PULL(1)

OPTIONS

-q, --quiet

This is passed to both underlying git-fetch to squelch reporting of during transfer, and underlying git-merge to squelch output during merging.

-v, --verbose

Pass --verbose to git-fetch and git-merge.

--[no-]recurse-submodules[=yes|on-demand|no]

This option controls if new commits of all populated submodules should be fetched and updated, too (see git-config(1) and gitmodules(5)).

If the checkout is done via rebase, local submodule commits are rebased as well.

If the update is done via merge, the submodule conflicts are resolved and checked out.

Options related to merging

--commit, --no-commit

Perform the merge and commit the result. This option can be used to override --no-commit.

With --no-commit perform the merge and stop just before creating a merge commit, to give the user a chance to inspect and further tweak the
merge result before committing.

Note that fast-forward updates do not create a merge commit and therefore there is no way to stop those merges with --no-commit. Thus, if you
want to ensure your branch is not changed or updated by the merge command, use --no-ff with --no-commit.

--edit, -e, --no-edit

Invoke an editor before committing successful mechanical merge to further edit the auto-generated merge message, so that the user can explain
and justify the merge. The --no-edit option can be used to accept the auto-generated message (this is generally discouraged).

Older scripts may depend on the historical behaviour of not allowing the user to edit the merge log message. They will see an editor opened
when they run git merge. To make it easier to adjust such scripts to the updated behaviour, the environment variable GIT_MERGE_AUTOEDIT can be
set to no at the beginning of them.

--cleanup=<mode>
This option determines how the merge message will be cleaned up before committing. See git-commit(1) for more details. In addition, if the
<mode> is given a value of scissors, scissors will be appended to MERGE_MSG before being passed on to the commit machinery in the case of a
merge conflict.

--ff, --no-ff, --ff-only
Specifies how a merge is handled when the merged-in history is already a descendant of the current history. --ff is the default unless merging
an annotated (and possibly signed) tag that is not stored in its natural place in the refs/tags/ hierarchy, in which case --no-ff is assumed.

With --ff, when possible resolve the merge as a fast-forward (only update the branch pointer to match the merged branch; do not create a merge
commit). When not possible (when the merged-in history is not a descendant of the current history), create a merge commit.

With --no-ff, create a merge commit in all cases, even when the merge could instead be resolved as a fast-forward.

With --ff-only, resolve the merge as a fast-forward when possible. When not possible, refuse to merge and exit with a non-zero status.

-S[<keyid>], --gpg-sign[=<keyid>]
GPG-sign the resulting merge commit. The keyid argument is optional and defaults to the committer identity; if specified, it must be stuck to
the option without a space.

--log[=<n>], --no-log
In addition to branch names, populate the log message with one-line descriptions from at most <n> actual commits that are being merged. See
also git-fmt-merge-msg(1).

With --no-log do not list one-line descriptions from the actual commits being merged.

--signoff, --no-signoff

Add Signed-off-by line by the committer at the end of the commit log message. The meaning of a signoff depends on the project, but it typically
certifies that committer has the rights to submit this work under the same license and agrees to a Developer Certificate of Origin (see
http://developercertificate.org/ for more information).

With --no-signoff do not add a Signed-off-by line.

--stat, -n, --no-stat
Show a diffstat at the end of the merge. The diffstat is also controlled by the configuration option merge.stat.

With -n or --no-stat do not show a diffstat at the end of the merge.

--squash, --no-squash
Produce the working tree and index state as if a real merge happened (except for the merge information), but do not actually make a commit,
move the HEAD, or record $GIT_DIR/MERGE_HEAD (to cause the next git commit command to create a merge commit). This allows you to create a
single commit on top of the current branch whose effect is the same as merging another branch (or more in case of an octopus).

With --no-squash perform the merge and commit the result. This option can be used to override --squash.

With --squash, --commit is not allowed, and will fail.

--no-verify
This option bypasses the pre-merge and commit-msg hooks. See also githooks(5).

-s <strategy>, --strategy=<strategy>
Use the given merge strategy; can be supplied more than once to specify them in the order they should be tried. If there is no -s option, a
built-in list of strategies is used instead (git merge-recursive when merging a single head, git merge-octopus otherwise).

-X <option>, --strategy-option=<option>
Pass merge strategy specific option through to the merge strategy.

--verify-signatures, --no-verify-signatures
Verify that the tip commit of the side branch being merged is signed with a valid key, i.e. a key that has a valid uid: in the default trust
model, this means the signing key has been signed by a trusted key. If the tip commit of the side branch is not signed with a valid key, the
merge is aborted.

--summary, --no-summary
Synonyms to --stat and --no-stat; these are deprecated and will be removed in the future.

--allow-unrelated-histories
By default, git merge command refuses to merge histories that do not share a common ancestor. This option can be used to override this safety

when merging histories of two projects that started their lives independently. As that is a very rare occasion, no configuration variable to
enable this by default exists and will not be added.

-r, --rebase[=false|true|merges|preserve|interactive]
When true, rebase the current branch on top of the upstream branch after fetching. If there is a remote-tracking branch corresponding to the
upstream branch and the upstream branch was rebased since last fetched, the rebase uses that information to avoid rebasing non-local changes.

When set to merges, rebase using git rebase --rebase-merges so that the local merge commits are included in the rebase (see git-rebase(1) for
details).

When set to preserve (deprecated in favor of merges), rebase with the --preserve-merges option passed to git rebase so that locally created
merge commits will not be flattened.

When false, merge the current branch into the upstream branch.

When interactive, enable the interactive mode of rebase.

See pull.rebase, branch.<name>.rebase and branch.autoSetupRebase in git-config(1) if you want to make git pull always use --rebase instead of
merging.

Note
This is a potentially dangerous mode of operation. It rewrites history, which does not bode well when you published that history already.
Do not use this option unless you have read git-rebase(1) carefully.

--no-rebase
Override earlier --rebase.

--autostash, --no-autostash
Before starting rebase, stash local modifications away (see git-stash(1)) if needed, and apply the stash entry when done. --no-autostash is
useful to override the rebase.autoStash configuration variable (see git-config(1)).

This option is only valid when "--rebase" is used.

Options related to fetching
--all
Fetch all remotes.

-a, --append
Append ref names and object names of fetched refs to the existing contents of .git/FETCH_HEAD. Without this option old data in .git/FETCH_HEAD
will be overwritten.

--depth=<depth>

Limit fetching to the specified number of commits from the tip of each remote branch history. If fetching to a shallow repository created by
git clone with --depth=<depth> option (see git-clone(1)), deepen or shorten the history to the specified number of commits. Tags for the
deepened commits are not fetched.

--deepen=<depth>
Similar to --depth, except it specifies the number of commits from the current shallow boundary instead of from the tip of each remote branch
history.

--shallow-since=<date>
Deepen or shorten the history of a shallow repository to include all reachable commits after <date>.

--shallow-exclude=<revision>
Deepen or shorten the history of a shallow repository to exclude commits reachable from a specified remote branch or tag. This option can be
specified multiple times.

--unshallow
If the source repository is complete, convert a shallow repository to a complete one, removing all the limitations imposed by shallow
repositories.

If the source repository is shallow, fetch as much as possible so that the current repository has the same history as the source repository.

--update-shallow
By default when fetching from a shallow repository, git fetch refuses refs that require updating .git/shallow. This option updates .git/shallow
and accept such refs.

--negotiation-tip=<commit|glob>
By default, Git will report, to the server, commits reachable from all local refs to find common commits in an attempt to reduce the size of
the to-be-received packfile. If specified, Git will only report commits reachable from the given tips. This is useful to speed up fetches when
the user knows which local ref is likely to have commits in common with the upstream ref being fetched.

This option may be specified more than once; if so, Git will report commits reachable from any of the given commits.

The argument to this option may be a glob on ref names, a ref, or the (possibly abbreviated) SHA-1 of a commit. Specifying a glob is equivalent
to specifying this option multiple times, one for each matching ref name.

See also the fetch.negotiationAlgorithm configuration variable documented in git-config(1).

-f, --force

When git fetch is used with <src>:<dst> refspec it may refuse to update the local branch as discussed in the <refspec> part of the git-fetch(1)
documentation. This option overrides that check.

-k, --keep
Keep downloaded pack.

--no-tags
By default, tags that point at objects that are downloaded from the remote repository are fetched and stored locally. This option disables this
automatic tag following. The default behavior for a remote may be specified with the remote.<name>.tagOpt setting. See git-config(1).

-u, --update-head-ok
By default git fetch refuses to update the head which corresponds to the current branch. This flag disables the check. This is purely for the
internal use for git pull to communicate with git fetch, and unless you are implementing your own Porcelain you are not supposed to use it.

--upload-pack <upload-pack>
When given, and the repository to fetch from is handled by git fetch-pack, --exec=<upload-pack> is passed to the command to specify non-default
path for the command run on the other end.

--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.

-o <option>, --server-option=<option>
Transmit the given string to the server when communicating using protocol version 2. The given string must not contain a NUL or LF character.
The server’s handling of server options, including unknown ones, is server-specific. When multiple --server-option=<option> are given, they are
all sent to the other side in the order listed on the command line.

--show-forced-updates
By default, git checks if a branch is force-updated during fetch. This can be disabled through fetch.showForcedUpdates, but the
--show-forced-updates option guarantees this check occurs. See git-config(1).

--no-show-forced-updates
By default, git checks if a branch is force-updated during fetch. Pass --no-show-forced-updates or set fetch.showForcedUpdates to false to skip
this check for performance reasons. If used during git-pull the --ff-only option will still check for forced updates before attempting a
fast-forward update. See git-config(1).

-4, --ipv4
Use IPv4 addresses only, ignoring IPv6 addresses.

-6, --ipv6
Use IPv6 addresses only, ignoring IPv4 addresses.

<repository>
The "remote" repository that is the source of a fetch or pull 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).

GIT-COMMIT(1)

OPTIONS
-a, --all
Tell the command to automatically stage files that have been modified and deleted, but new files you have not told Git about are not affected.

-p, --patch
Use the interactive patch selection interface to chose which changes to commit. See git-add(1) for details.

-C <commit>, --reuse-message=<commit>
Take an existing commit object, and reuse the log message and the authorship information (including the timestamp) when creating the commit.

-c <commit>, --reedit-message=<commit>
Like -C, but with -c the editor is invoked, so that the user can further edit the commit message.

--fixup=<commit>
Construct a commit message for use with rebase --autosquash. The commit message will be the subject line from the specified commit with a
prefix of "fixup! ". See git-rebase(1) for details.

--squash=<commit>
Construct a commit message for use with rebase --autosquash. The commit message subject line is taken from the specified commit with a prefix
of "squash! ". Can be used with additional commit message options (-m/-c/-C/-F). See git-rebase(1) for details.

--reset-author
When used with -C/-c/--amend options, or when committing after a conflicting cherry-pick, declare that the authorship of the resulting commit
now belongs to the committer. This also renews the author timestamp.

--short
When doing a dry-run, give the output in the short-format. See git-status(1) for details. Implies --dry-run.

--branch
Show the branch and tracking info even in short-format.

--porcelain
When doing a dry-run, give the output in a porcelain-ready format. See git-status(1) for details. Implies --dry-run.

--long
When doing a dry-run, give the output in the long-format. Implies --dry-run.

-z, --null
When showing short or porcelain status output, print the filename verbatim and terminate the entries with NUL, instead of LF. If no format is
given, implies the --porcelain output format. Without the -z option, filenames with "unusual" characters are quoted as explained for the
configuration variable core.quotePath (see git-config(1)).

-F <file>, --file=<file>
Take the commit message from the given file. Use - to read the message from the standard input.

--author=<author>
Override the commit author. Specify an explicit author using the standard A U Thor <author@example.com> format. Otherwise <author> is assumed
to be a pattern and is used to search for an existing commit by that author (i.e. rev-list --all -i --author=<author>); the commit author is
then copied from the first such commit found.

--date=<date>
Override the author date used in the commit.

-m <msg>, --message=<msg>
Use the given <msg> as the commit message. If multiple -m options are given, their values are concatenated as separate paragraphs.

The -m option is mutually exclusive with -c, -C, and -F.

-t <file>, --template=<file>
When editing the commit message, start the editor with the contents in the given file. The commit.template configuration variable is often used
to give this option implicitly to the command. This mechanism can be used by projects that want to guide participants with some hints on what
to write in the message in what order. If the user exits the editor without editing the message, the commit is aborted. This has no effect when
a message is given by other means, e.g. with the -m or -F options.

-s, --signoff
Add Signed-off-by line by the committer at the end of the commit log message. The meaning of a signoff depends on the project, but it typically
certifies that committer has the rights to submit this work under the same license and agrees to a Developer Certificate of Origin (see
http://developercertificate.org/ for more information).

-n, --no-verify

This option bypasses the pre-commit and commit-msg hooks. See also githooks(5).

--allow-empty
Usually recording a commit that has the exact same tree as its sole parent commit is a mistake, and the command prevents you from making such a
commit. This option bypasses the safety, and is primarily for use by foreign SCM interface scripts.

--allow-empty-message
Like --allow-empty this command is primarily for use by foreign SCM interface scripts. It allows you to create a commit with an empty commit
message without using plumbing commands like git-commit-tree(1).

--cleanup=<mode>
This option determines how the supplied commit message should be cleaned up before committing. The <mode> can be strip, whitespace, verbatim,
scissors or default.

strip
Strip leading and trailing empty lines, trailing whitespace, commentary and collapse consecutive empty lines.

whitespace
Same as strip except #commentary is not removed.

verbatim
Do not change the message at all.

scissors
Same as whitespace except that everything from (and including) the line found below is truncated, if the message is to be edited. "#" can
be customized with core.commentChar.

# ------------------------ >8 ------------------------

default
Same as strip if the message is to be edited. Otherwise whitespace.

The default can be changed by the commit.cleanup configuration variable (see git-config(1)).

-e, --edit
The message taken from file with -F, command line with -m, and from commit object with -C are usually used as the commit log message
unmodified. This option lets you further edit the message taken from these sources.

--no-edit
Use the selected commit message without launching an editor. For example, git commit --amend --no-edit amends a commit without changing its
commit message.

--amend
Replace the tip of the current branch by creating a new commit. The recorded tree is prepared as usual (including the effect of the -i and -o
options and explicit pathspec), and the message from the original commit is used as the starting point, instead of an empty message, when no
other message is specified from the command line via options such as -m, -F, -c, etc. The new commit has the same parents and author as the
current one (the --reset-author option can countermand this).

It is a rough equivalent for:

```
$ git reset --soft HEAD^
$ ... do something else to come up with the right tree ...
$ git commit -c ORIG_HEAD
```

but can be used to amend a merge commit.

You should understand the implications of rewriting history if you amend a commit that has already been published. (See the "RECOVERING FROM
UPSTREAM REBASE" section in git-rebase(1).)

--no-post-rewrite
Bypass the post-rewrite hook.

-i, --include
Before making a commit out of staged contents so far, stage the contents of paths given on the command line as well. This is usually not what
you want unless you are concluding a conflicted merge.

-o, --only
Make a commit by taking the updated working tree contents of the paths specified on the command line, disregarding any contents that have been
staged for other paths. This is the default mode of operation of git commit if any paths are given on the command line, in which case this
option can be omitted. If this option is specified together with --amend, then no paths need to be specified, which can be used to amend the
last commit without committing changes that have already been staged. If used together with --allow-empty paths are also not required, and an
empty commit will be created.

--pathspec-from-file=<file>
Pathspec is passed in <file> instead of commandline args. If <file> is exactly - then standard input is used. Pathspec elements are separated
by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable core.quotePath (see git-config(1)). See also
--pathspec-file-nul and global --literal-pathspecs.

--pathspec-file-nul
Only meaningful with --pathspec-from-file. Pathspec elements are separated with NUL character and all other characters are taken literally
(including newlines and quotes).

-u[<mode>], --untracked-files[=<mode>]
Show untracked files.

The mode parameter is optional (defaults to all), and is used to specify the handling of untracked files; when -u is not used, the default is
normal, i.e. show untracked files and directories.

The possible options are:

- no - Show no untracked files
- normal - Shows untracked files and directories
- all - Also shows individual files in untracked directories.

The default can be changed using the status.showUntrackedFiles configuration variable documented in git-config(1).

-v, --verbose
Show unified diff between the HEAD commit and what would be committed at the bottom of the commit message template to help the user describe
the commit by reminding what changes the commit has. Note that this diff output doesn't have its lines prefixed with #. This diff will not be a
part of the commit message. See the commit.verbose configuration variable in git-config(1).

If specified twice, show in addition the unified diff between what would be committed and the worktree files, i.e. the unstaged changes to
tracked files.

-q, --quiet
Suppress commit summary message.

--dry-run
Do not create a commit, but show a list of paths that are to be committed, paths with local changes that will be left uncommitted and paths
that are untracked.

--status
Include the output of git-status(1) in the commit message template when using an editor to prepare the commit message. Defaults to on, but can
be used to override configuration variable commit.status.

--no-status
Do not include the output of git-status(1) in the commit message template when using an editor to prepare the default commit message.

-S[<keyid>], --gpg-sign[=<keyid>]
GPG-sign commits. The keyid argument is optional and defaults to the committer identity; if specified, it must be stuck to the option without a
space.

--no-gpg-sign
Countermand commit.gpgSign configuration variable that is set to force each and every commit to be signed.

--
Do not interpret any more arguments as options.

<pathspec>...
When pathspec is given on the command line, commit the contents of the files that match the pathspec without recording the changes already
added to the index. The contents of these files are also staged for the next commit on top of what have been staged before.

For more details, see the pathspec entry in gitglossary(7).

GIT-BRANCH(1)

OPTIONS
-d, --delete
Delete a branch. The branch must be fully merged in its upstream branch, or in HEAD if no upstream was set with --track or --set-upstream-to.

-D
Shortcut for --delete --force.

--create-reflog
Create the branch’s reflog. This activates recording of all changes made to the branch ref, enabling use of date based sha1 expressions such as
"<branchname>@{yesterday}". Note that in non-bare repositories, reflogs are usually enabled by default by the core.logAllRefUpdates config
option. The negated form --no-create-reflog only overrides an earlier --create-reflog, but currently does not negate the setting of
core.logAllRefUpdates.

-f, --force
Reset <branchname> to <startpoint>, even if <branchname> exists already. Without -f, git branch refuses to change an existing branch. In
combination with -d (or --delete), allow deleting the branch irrespective of its merged status. In combination with -m (or --move), allow
renaming the branch even if the new branch name already exists, the same applies for -c (or --copy).

-m, --move
Move/rename a branch and the corresponding reflog.

-M
Shortcut for --move --force.

-c, --copy
Copy a branch and the corresponding reflog.

-C
Shortcut for --copy --force.

--color[=<when>]
Color branches to highlight current, local, and remote-tracking branches. The value must be always (the default), never, or auto.

--no-color
Turn off branch colors, even when the configuration file gives the default to color output. Same as --color=never.

-i, --ignore-case
Sorting and filtering branches are case insensitive.

--column[=<options>], --no-column
Display branch listing in columns. See configuration variable column.branch for option syntax.--column and --no-column without options are
equivalent to always and never respectively.

This option is only applicable in non-verbose mode.

-r, --remotes
List or delete (if used with -d) the remote-tracking branches. Combine with --list to match the optional pattern(s).

-a, --all
List both remote-tracking branches and local branches. Combine with --list to match optional pattern(s).

-l, --list
List branches. With optional <pattern>..., e.g. git branch --list 'maint-*', list only the branches that match the pattern(s).

--show-current
Print the name of the current branch. In detached HEAD state, nothing is printed.

-v, -vv, --verbose
When in list mode, show sha1 and commit subject line for each head, along with relationship to upstream branch (if any). If given twice, print
the path of the linked worktree (if any) and the name of the upstream branch, as well (see also git remote show <remote>). Note that the
current worktree’s HEAD will not have its path printed (it will always be your current directory).

-q, --quiet
Be more quiet when creating or deleting a branch, suppressing non-error messages.

--abbrev=<length>

Alter the sha1’s minimum display length in the output listing. The default value is 7 and can be overridden by the core.abbrev config option.

--no-abbrev
Display the full sha1s in the output listing rather than abbreviating them.

-t, --track
When creating a new branch, set up branch.<name>.remote and branch.<name>.merge configuration entries to mark the start-point branch as
"upstream" from the new branch. This configuration will tell git to show the relationship between the two branches in git status and git branch
-v. Furthermore, it directs git pull without arguments to pull from the upstream when the new branch is checked out.

This behavior is the default when the start point is a remote-tracking branch. Set the branch.autoSetupMerge configuration variable to false if
you want git switch, git checkout and git branch to always behave as if --no-track were given. Set it to always if you want this behavior when
the start-point is either a local or remote-tracking branch.

--no-track
Do not set up "upstream" configuration, even if the branch.autoSetupMerge configuration variable is true.

--set-upstream
As this option had confusing syntax, it is no longer supported. Please use --track or --set-upstream-to instead.

-u <upstream>, --set-upstream-to=<upstream>
Set up <branchname>'s tracking information so <upstream> is considered <branchname>'s upstream branch. If no <branchname> is specified, then it
defaults to the current branch.

--unset-upstream
Remove the upstream information for <branchname>. If no branch is specified it defaults to the current branch.

--edit-description
Open an editor and edit the text to explain what the branch is for, to be used by various other commands (e.g.  format-patch, request-pull, and
merge (if enabled)). Multi-line explanations may be used.

--contains [<commit>]
Only list branches which contain the specified commit (HEAD if not specified). Implies --list.

--no-contains [<commit>]
Only list branches which don’t contain the specified commit (HEAD if not specified). Implies --list.

--merged [<commit>]

Only list branches whose tips are reachable from the specified commit (HEAD if not specified). Implies --list, incompatible with --no-merged.

--no-merged [<commit>]
Only list branches whose tips are not reachable from the specified commit (HEAD if not specified). Implies --list, incompatible with --merged.

<branchname>
The name of the branch to create or delete. The new branch name must pass all checks defined by git-check-ref-format(1). Some of these checks
may restrict the characters allowed in a branch name.

<start-point>
The new branch head will point to this commit. It may be given as a branch name, a commit-id, or a tag. If this option is omitted, the current
HEAD will be used instead.

<oldbranch>
The name of an existing branch to rename.

<newbranch>
The new name for an existing branch. The same restrictions as for <branchname> apply.

--sort=<key>
Sort based on the key given. Prefix - to sort in descending order of the value. You may use the --sort=<key> option multiple times, in which
case the last key becomes the primary key. The keys supported are the same as those in git for-each-ref. Sort order defaults to the value
configured for the branch.sort variable if exists, or to sorting based on the full refname (including refs/... prefix). This lists detached
HEAD (if present) first, then local branches and finally remote-tracking branches. See git-config(1).

--points-at <object>
Only list branches of the given object.

--format <format>
A string that interpolates %(fieldname) from a branch ref being shown and the object it points at. The format is the same as that of git-for-
each-ref(1).

CONFIGURATION
pager.branch is only respected when listing branches, i.e., when --list is used or implied. The default is to use a pager. See git-config(1).

EXAMPLES
Start development from a known tag

```
$ git clone git://git.kernel.org/pub/scm/.../linux-2.6 my2.6
$ cd my2.6
$ git branch my2.6.14 v2.6.14   (1)
```

```
$ git switch my2.6.14
```

1. This step and the next one could be combined into a single step with "checkout -b my2.6.14 v2.6.14".

Delete an unneeded branch

```
$ git clone git://git.kernel.org/.../git.git my.git
$ cd my.git
$ git branch -d -r origin/todo origin/html origin/man   (1)
$ git branch -D test                                    (2)
```

1. Delete the remote-tracking branches "todo", "html" and "man". The next fetch or pull will create them again unless you configure them not
to. See git-fetch(1).
2. Delete the "test" branch even if the "master" branch (or whichever branch is currently checked out) does not have all commits from the test
branch.

Listing branches from a specific remote

```
$ git branch -r -l '<remote>/<pattern>'                 (1)
$ git for-each-ref 'refs/remotes/<remote>/<pattern>'    (2)
```

1. Using -a would conflate <remote> with any local branches you happen to have been prefixed with the same <remote> pattern.
2. for-each-ref can take a wide range of options. See git-for-each-ref(1)

Patterns will normally need quoting.

GIT-ADD(1)

OPTIONS

<pathspec>...

Files to add content from. Fileglobs (e.g. *.c) can be given to add all matching files. Also a leading directory name (e.g. dir to add
dir/file1 and dir/file2) can be given to update the index to match the current state of the directory as a whole (e.g. specifying dir will
record not just a file dir/file1 modified in the working tree, a file dir/file2 added to the working tree, but also a file dir/file3 removed
from the working tree). Note that older versions of Git used to ignore removed files; use --no-all option if you want to add modified or new
files but ignore removed ones.

For more details about the <pathspec> syntax, see the pathspec entry in gitglossary(7).

-n, --dry-run

Don’t actually add the file(s), just show if they exist and/or will be ignored.

-v, --verbose

Be verbose.

-f, --force
Allow adding otherwise ignored files.

-i, --interactive
Add modified contents in the working tree interactively to the index. Optional path arguments may be supplied to limit operation to a subset of the working tree. See “Interactive mode” for details.

-p, --patch
Interactively choose hunks of patch between the index and the work tree and add them to the index. This gives the user a chance to review the difference before adding modified contents to the index.

This effectively runs add --interactive, but bypasses the initial command menu and directly jumps to the patch subcommand. See “Interactive mode” for details.

-e, --edit
Open the diff vs. the index in an editor and let the user edit it. After the editor was closed, adjust the hunk headers and apply the patch to the index.

The intent of this option is to pick and choose lines of the patch to apply, or even to modify the contents of lines to be staged. This can be quicker and more flexible than using the interactive hunk selector. However, it is easy to confuse oneself and create a patch that does not apply to the index. See EDITING PATCHES below.

-u, --update
Update the index just where it already has an entry matching <pathspec>. This removes as well as modifies index entries to match the working tree, but adds no new files.

If no <pathspec> is given when -u option is used, all tracked files in the entire working tree are updated (old versions of Git used to limit the update to the current directory and its subdirectories).

-A, --all, --no-ignore-removal
Update the index not only where the working tree has a file matching <pathspec> but also where the index already has an entry. This adds, modifies, and removes index entries to match the working tree.

If no <pathspec> is given when -A option is used, all files in the entire working tree are updated (old versions of Git used to limit the update to the current directory and its subdirectories).

--no-all, --ignore-removal
Update the index by adding new files that are unknown to the index and files modified in the working tree, but ignore files that have been removed from the working tree. This option is a no-op when no <pathspec> is used.

This option is primarily to help users who are used to older versions of Git, whose "git add <pathspec>..." was a synonym for "git add --no-all
<pathspec>...", i.e. ignored removed files.

-N, --intent-to-add
Record only the fact that the path will be added later. An entry for the path is placed in the index with no content. This is useful for, among
other things, showing the unstaged content of such files with git diff and committing them with git commit -a.

--refresh
Don’t add the file(s), but only refresh their stat() information in the index.

--ignore-errors
If some files could not be added because of errors indexing them, do not abort the operation, but continue adding the others. The command shall
still exit with non-zero status. The configuration variable add.ignoreErrors can be set to true to make this the default behaviour.

--ignore-missing
This option can only be used together with --dry-run. By using this option the user can check if any of the given files would be ignored, no
matter if they are already present in the work tree or not.

--no-warn-embedded-repo
By default, git add will warn when adding an embedded repository to the index without using git submodule add to create an entry in
.gitmodules. This option will suppress the warning (e.g., if you are manually performing operations on submodules).

--renormalize
Apply the "clean" process freshly to all tracked files to forcibly add them again to the index. This is useful after changing core.autocrlf
configuration or the text attribute in order to correct files added with wrong CRLF/LF line endings. This option implies -u.

--chmod=(+|-)x
Override the executable bit of the added files. The executable bit is only changed in the index, the files on disk are left unchanged.

--pathspec-from-file=<file>
Pathspec is passed in <file> instead of commandline args. If <file> is exactly - then standard input is used. Pathspec elements are separated
by LF or CR/LF. Pathspec elements can be quoted as explained for the configuration variable core.quotePath (see git-config(1)). See also
--pathspec-file-nul and global –literal-pathspecs.

--pathspec-file-nul
Only meaningful with --pathspec-from-file. Pathspec elements are separated with NUL character and all other characters are taken literally
(including newlines and quotes).

--

This option can be used to separate command-line options from the list of files, (useful when filenames might be mistaken for command-line
options).

EXAMPLES

- Adds content from all *.txt files under Documentation directory and its subdirectories:

  $ git add Documentation/*.txt

  Note that the asterisk * is quoted from the shell in this example; this lets the command include the files from subdirectories of
  Documentation/ directory.

- Considers adding content from all git-*.sh scripts:

  $ git add git-*.sh

  Because this example lets the shell expand the asterisk (i.e. you are listing the files explicitly), it does not consider subdir/git-foo.sh.q

## Branching and Merging

- `git branch`: List, create, or delete branches.
  - `git branch <name>`: Create a new branch.
  - `git branch -d <name>`: Delete a branch.
  - `git branch -M <new-name>`: Rename a branch (force).
- `git checkout <branch>`: Switch to a branch.
- `git merge <branch>`: Merge a branch into the current branch.
- `git rebase <branch>`: Reapply commits on top of another base branch.

---

## Remote Repository Management

- `git remote -v`: List remote repositories.
- `git remote add <name> <url>`: Add a remote repository.
- `git pull <remote> <branch>`: Fetch and merge changes from the remote repository.
- `git push <remote> <branch>`: Push changes to a remote repository.
- `git fetch <remote>`: Fetch changes from the remote repository without merging.

---

## Stashing and Cleaning

- `git stash`: Temporarily save changes that are not ready to commit.
  - `git stash list`: View saved stashes.

    - `git stash pop`: Apply and remove the most recent stash.
- `git clean`: Remove untracked files from the working directory.

---

### Tagging

- `git tag <name>`: Create a new tag.
    - `git tag -a <name> -m "<message>"`: Create an annotated tag.
    - `git tag -d <name>`: Delete a tag.

---

### Undoing Changes

- `git reset`: Reset changes to a previous state.
    - `git reset --soft <commit>`: Reset to a commit but keep changes staged.
    - `git reset --hard <commit>`: Reset to a commit and discard all changes.
- `git revert <commit>`: Create a new commit that undoes the specified commit.

---

### Configuration

- `git config`: Configure Git settings.
    - `git config --global user.name "<name>"`: Set the global username.
    - `git config --global user.email "<email>"`: Set the global email address.
    - `git config --list`: List all configuration settings.

---

### Viewing Differences

- `git diff`: Show differences between commits, branches, or the working directory.
    - `git diff --staged`: Show changes that have been staged.

---

### Miscellaneous

- `git blame <file>`: Show what revision and author last modified each line of a file.
- `git show <commit>`: Show details about a specific commit.
- `git archive`: Create a tar or zip archive of the repository.
- `git cherry-pick <commit>`: Apply changes from a specific commit.

The `git add` command is used to stage changes in your working directory, preparing them for a commit. Here's how it works:

---

## How `git add` Works

1. **Working Directory**: This is where your files and changes are made.
2. **Staging Area (Index)**: The `git add` command moves changes from the working directory to the staging area. This area is like a preparation zone where you specify what changes you want to include in the next commit.

---

## Key Points

- **Files Not Tracked Yet**: If a file is new and not tracked by Git, `git add` tells Git to start tracking it.
- **Modified Files**: For already tracked files, `git add` stages the latest changes to be included in the next commit.
- **Deleted Files**: If a tracked file is deleted, `git add` stages this deletion.

---

## Examples

- **Add a Single File**:

  ```
  git add file.txt
  ```

  Stages `file.txt` for the next commit.

- **Add Multiple Files**:

  ```
  git add file1.txt file2.txt
  ```

  Stages `file1.txt` and `file2.txt`.

- **Add All Changes**:

  ```
  git add .
  ```

  Stages all changes in the current directory (including subdirectories).

- **Add All Files in a Folder**:

  ```
  git add folder_name/
  ```

  Stages all files in `folder_name`.

---

## Staging vs Committing

- `git add` only prepares changes for the commit.

- To save those changes to the repository, you must run:

  ```
  git commit -m "Commit message"
  ```

---

## Behind the Scenes

When you run `git add`, Git:

1. Checks the specified files or patterns.
2. Copies the file snapshots from the working directory into the staging area (Index).
3. Marks the files for inclusion in the next commit.

---

Would you like an illustration or further details about its behavior?

The `git commit` command is used to save staged changes to the Git repository. It creates a new snapshot of your project at a specific point in time. Here's how it works:

---

## How `git commit` Works

1. **Staging Area**: Changes must first be added to the staging area using `git add`.
2. **Create a Commit**: When you run `git commit`, Git takes the snapshot of the files in the staging area and saves it in the repository as a commit.
3. **Log Entry**: Each commit has a unique ID (SHA-1 hash) and contains:
    - A **commit message** describing the changes.
    - The **author** and **timestamp**.
    - A **reference to the previous commit** (parent commit).

---

## Basic Usage

- **Commit Staged Changes**:

  ```
  git commit -m "Descriptive commit message"
  ```

  Creates a new commit with the specified message.

---

## Options

- **`-m`**: Adds a commit message inline.

  ```
  git commit -m "Fix bug in login functionality"
  ```

  This avoids opening the default text editor.

- **`-a`**: Automatically stages modified files (but not new files).

  ```
  git commit -a -m "Commit with all changes"
  ```

  Stages all modified files and commits them.

- **`--amend`**: Modifies the last commit.

  ```
  git commit --amend -m "Updated commit message"
  ```

  Used to edit the commit message or add changes to the previous commit.

- **`--verbose`**: Shows changes staged for commit in the terminal before committing.
- **`--allow-empty`**: Creates a commit even if there are no staged changes.

---

## How It Saves Changes

- **Before Committing**: Changes are in the staging area.

- **After Committing**: Changes are saved to the repository, and the staging area is cleared.

---

## Behind the Scenes

1. Git records the state of the staged files in a commit object.
2. This commit object contains:
    - File content (snapshot).
    - Metadata: author, timestamp, and commit message.
    - A reference to the parent commit(s).

---

## Best Practices for Commit Messages

- Use the imperative mood (e.g., "Add feature" instead of "Added feature").
- Keep the summary line short (50-72 characters).
- Provide additional context in a separate paragraph if needed.

---

Would you like to see how `git commit` works with specific examples?

The `git push` command uploads your local commits to a remote repository, making them accessible to others or storing them on a remote server. Here's how it works:

---

## How `git push` Works

1. **Local Repository**: You make changes, stage them with `git add`, and commit them with `git commit`.
2. **Remote Repository**: A repository hosted on platforms like GitHub, GitLab, or Bitbucket.
3. **Push Changes**: The `git push` command sends your commits from your local branch to the corresponding branch on the remote repository.

---

## Basic Usage

- **Push the Current Branch**:

  ```
  git push origin main
  ```

  Sends commits from the local `main` branch to the remote `main` branch on `origin`.

- **Push All Branches**:

  ```
  git push --all origin
  ```

  Pushes all branches to the `origin` remote.

---

## Options

- **`-u`**: Sets the upstream branch for the current branch.

  ```
  git push -u origin main
  ```

  Links your local `main` branch to `origin/main`, so future pushes can use `git push` without specifying the branch.

- **`--force` (`-f`)**: Forces a push, even if it overwrites changes on the remote branch.

  ```
  git push --force
  ```

  Use cautiously as it can overwrite others' work.

- **`--tags`**: Pushes all local tags to the remote repository.

  ```
  git push --tags
  ```

- **`--dry-run`**: Simulates a push without actually uploading anything.

  ```
  git push --dry-run
  ```

---

## What Happens Behind the Scenes

1. Git checks the commits on the local branch.
2. It compares the local branch with the remote branch.
3. It uploads any new or updated commits from the local branch to the remote branch.

---

## Common Scenarios

- **If Remote Branch Doesn't Exist**: You may need to specify the branch explicitly:

  ```
  git push origin main
  ```

- **If Someone Else Has Pushed**: Your push may be rejected if the remote branch is ahead of your local branch. You must first pull their changes and merge them before pushing.

---

## Best Practices

1. Always pull (`git pull`) before pushing to avoid conflicts.
2. Avoid using `--force` unless absolutely necessary.
3. Use `git push -u origin <branch>` to simplify future pushes.

---

Would you like to explore a detailed example or troubleshoot common `git push` issues?

In the backend, when you execute `git push`, Git interacts with the remote repository (like GitHub, GitLab, or a private server). Here's a detailed explanation of how it works and how Git stores your data:

---

## How `git push` Works in the Backend

1. **Local Repository Snapshot**:
    - When you run `git push`, Git looks at the local repository and the commits you've made. These commits are stored in a hidden directory called `.git` on your local machine.
    - Each commit contains:
        - **File changes**: A snapshot of the project at the time of the commit.
        - **Metadata**: Information like the author, timestamp, and commit message.
        - **Reference to Parent Commit(s)**: A link to the previous commit(s) (creating a chain of commits).
2. **Communication with the Remote Repository**:
    - Git uses the **remote repository's URL** (like `https://github.com/user/repo.git`) and communicates via the Git protocol (over SSH or HTTPS).
    - When you run `git push origin main`, Git performs the following:
        - It **connects** to the remote repository (via SSH or HTTPS).
        - It checks the remote branch (`origin/main` in this case) to compare its state with your local branch.
3. **Push Process**:
    - **Compare Local and Remote**: Git compares your local branch (e.g., `main`) with the remote branch (e.g., `origin/main`). If there are new commits in your local branch that aren't in the remote branch, Git pushes those commits.
    - **Transmit Commits**: Git sends the changes in the form of commit objects. Each commit is a hash (SHA-1) that uniquely identifies the state of the repository at the time of the commit.
    - **Updating the Remote Repository**: The remote server (e.g., GitHub) receives the new commits and **appends** them to the corresponding branch (e.g., `main`).
4. **Backend Storage**:
    - **Commit Object**: On the backend, the remote repository stores commits as objects in its own internal storage, typically in a Git database.
        - **Object Database**: The commits, trees (file directories), and blobs (file content) are stored in an object database. Each object is uniquely identified by a SHA-1 hash, which ensures data integrity.
        - **Objects**: These are stored as compressed files that contain metadata and actual file contents (for blobs) or directory structures (for trees).
        - **Packfiles**: Over time, Git combines smaller objects into larger files called "packfiles" for more efficient storage.

5. **Ref Updates**:
    - **Update References**: Once the remote server receives the commits, it updates its **references** (also known as refs). A reference is a pointer to a specific commit in a branch, like `refs/heads/main`.
    - The server will update the reference for the branch to point to the latest commit (the one you just pushed).
    - **Branch History**: This creates a new "head" for the branch, a pointer to the most recent commit.

---

## Git's Storage Mechanism (Backend)

- **Git Objects**:
    - **Commit Object**: A commit that contains metadata (message, timestamp, author) and pointers to previous commits.
    - **Tree Object**: Represents directories and file names, holding references to the actual files or subdirectories.
    - **Blob Object**: Represents the actual content of files, stored as binary data.
    - **Tag Object**: Points to a specific commit and provides a name (like a release version).
- **Storage Format**:
    - These objects are stored in a compressed form in the `.git/objects` directory (local repository) or in the remote repository's object store.
    - **Hashing (SHA-1)**: Git uses SHA-1 to hash commit data, ensuring the integrity and uniqueness of each object. This way, every commit, file, and directory snapshot is securely identifiable.
- **Packfiles**:
    - Git optimizes storage by packing many small objects into a single file called a "packfile," reducing space and improving performance for large repositories.
    - Packfiles are efficient for transferring data between local and remote repositories.

---

## Git Workflow on Remote (Server-side)

When you push to a remote repository:

1. **Receive the Push**: The remote server (e.g., GitHub) receives the push request over HTTPS or SSH.
2. **Authenticate**: The server verifies the push request (authentication via tokens, SSH keys, etc.).
3. **Update Ref**: The server updates the reference (branch) to the new commit.
4. **Store Objects**: The server stores the new commits in its object database (as commit, tree, and blob objects).
5. **Respond**: After the push is successfully completed, the server responds to your local machine with a success message, confirming the changes.

---

### Security and Integrity:

- **SHA-1 Hashing**: Every Git object (commit, blob, tree, etc.) is identified by a unique SHA-1 hash. This guarantees that data integrity is maintained because any change in the content will result in a different hash.
- **Cryptographic Integrity**: If someone tries to tamper with the commit history, the SHA-1 hashes of objects would change, which would break the repository's integrity, making tampering detectable.

---

### Visualizing the Process:

1. **Local Git**: You make changes → `git add` → `git commit`.
2. **Push**: `git push origin main`.
3. **Remote Git**: The remote repository receives your commits, appends them to the branch, and stores the new commit objects.

---

### Summary

In simple terms:

- `git push` is about transferring new commits from your local machine to a remote server.
- The remote server stores those commits in a special Git object database.
- Each commit is hashed for integrity, and branches are updated with new commit references.

If you have any questions or need further clarification, feel free to ask!