Cheat sheet

Git

Git is a version control system (also called a source control system) that allows programmers and other people working with text files to coordinate changes while working independently. Git also supports binary assets such as pictures, but those formats don't support the line-by-line version management that makes version control really powerful.

Git concepts and workflow

Each user of Git maintains a separate repository (or multiple repositories) for working with the source code. The repository where the project was launched is considered the source of truth. Other users sync up with this remote repository, which can be hosted on a private network or by a public service provider such as GitHub, GitLab, or BitBucket.

There are formal processes for moving content to and from the remote repository. Each change goes through four phases before finally being stored in the remote repository. Each of these four phases has an associated location in which code is stored. These locations are:

- Working directory
- Staging environment
- Local repository
- Remote

The sections that follow describe the purpose and use of each location.

Working directory

The working directory is the location of the code in the local computer's file system. Developers add and make changes to code in the working directory.

Staging environment

The staging environment is an area that is special to Git, and does not appear in many other version control systems. Developers use the `git add` and `git reset` commands to add files to and remove files from the staging environment. You can think of staging as a temporary area where files are stored before being committed to the local repository.

Local repository

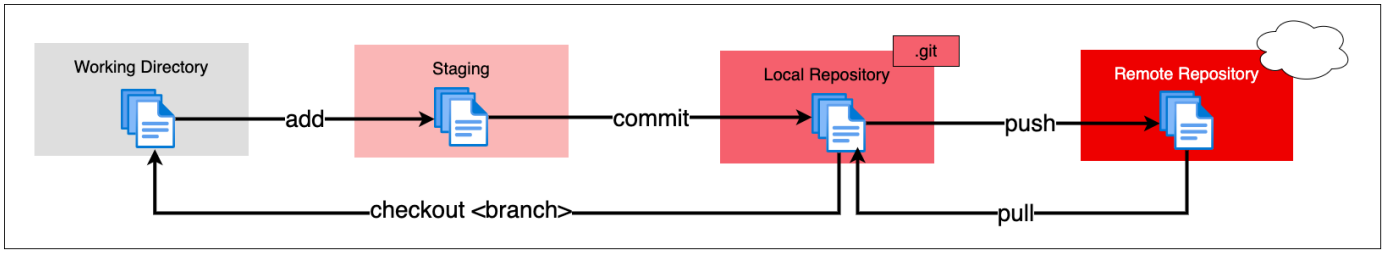
The local repository is the instance of the Git repository stored on the local computer, physically represented by a subdirectory named `.git` (the initial period makes it hidden) under the top-level directory. The `.git` directory contains metadata and other information needed by Git to manage the project. Once content has been added to the staging environment, developers use the `git commit` command to store new and updated content to the local repository.

The `git pull` command updates the local repository from the remote repository, bringing the local directory in sync with what other developers have done.

Remote repository

The remote repository is the Git repository on the network that is the single source of truth for all content being managed under a particular project. Developers use the `git push` command to upload content from the local repository to a remote repository.

The following figure shows the working directory, staging environment, local repository, and remote repository locations along with the `git` commands that move data to and from each location. After content has been pulled from a remote repository to a local repository, that content is available in the local file system.



Branches

Content in the remote repository can be broken into different paths of development through the concept of branches. The root branch is typically named `main`.

Working with repositories

The following sections describe `git` commands that create a local repository, download (clone) a remote repository to a local machine, and update a local repository with changes from a remote repository.

git init

```
git [options] init <repo_directory>
```

Creates a local repository that will be represented by the directory named `.git`. This command is not needed when working in a remote repository. Use `git clone` to start work with other repositories.

If the optional `<repo_directory>` parameter is not provided, the `.git` file is created in the current directory. If the parameter `<repo_directory>` is provided, that directory is created and the `.git` directory is created in that directory.

Example:

The following example uses the `git init` command to create a local repository in the `/home` directory of the user `lennonjohn`. The example creates the directory `coolcode` as the repository directory in which the repository `.git` file is stored:

```
$ git init ./coolcode
Initialized empty Git repository in /home/lennonjohn/coolcode/.git/
```

Note: The `$` in the examples is the command prompt.

git clone

```
git clone [options] <remote_repo_url> <target_directory>
```

Downloads content from a remote repository for local operations. If the parameter `<target_directory>` is provided, the repository contents are downloaded into that directory. (That directory must be empty.) Otherwise, `git` creates a directory based on the remote repository name.

Example:

The following example uses the command `git clone` to download the content of the remote repository found at https://github.com/redhat-developer/developers.redhat.com into the local directory `./rh`.

```
git clone https://github.com/redhat-developer/developers.redhat.com.git ./
rh
```

git pull

When called within a local repository, downloads the latest, current assets from the associated remote repository.

```
git pull [options]
```

Example:

The following example uses `git pull` to download files from the associated remote repository. Because contents in the remote repository and local repository are the same, the command responds with a message `Already up to date`.

```
$ git pull
Already up to date.
```

git fetch

```
git fetch [options] <repository>
```

Downloads code and assets from the remote repository to the local machine without overwriting the existing local code and assets in the current branch. If the optional `<repository>` is not provided, `git fetch` is executed against the Git repository associated with the present working directory.

Example:

The following example uses the `git fetch` command to downloaded updated assets from the corresponding remote repository, but will not merge the deltas in the branches on the local repository.

```
$ git fetch
```

git log

```
git log [options]
```

Displays the Git log file that contains a history of all transactions in the repository.

Example:

The following example uses `git log` with the `--oneline` option to show all activities in the repository in an abbreviated format:

```
$ git log --oneline
80f6259 (HEAD -> main) adding newfile.txt to main
665ecf1 (origin/your-feature, origin/main, origin/dev, origin/HEAD)
reorganizing repo structure
c9b791c reorganizing repo structure
af0f400 Update eapxp-quickstarts.yaml
28d8577 Update README.md
f8be8a1 Update README.md
456b537 Update README.md
415ce57 Update eapxp-quickstarts.yaml
70233e6 Update README.md
9263b26 Update README.md
886f7c1 Update README.md
3a0f42d Update README.md
1768b69 Example YAML: Develop MicroProfile app on JBoss EAP 7.3
10b9670 Added directions on how to create an asset inventory in the README
41e85e1 Initial commit
```

Working with branches

The following sections describe the various `git branch` command expressions you can use to work with branches in a repository.

Getting the current branch name

```
git branch
```

Shows all branches in the local repository, flagging the current branch that is checked out from the local repository.

Example:

The following example reports the current branch that is being worked within in the local repository. In this case the current branch is `my_feature` and is indicated by the asterisk before the branch name:

```
$ git branch
  dev
  main
* my_feature
```

Viewing remote branches

```
git branch -r
```

Displays all the branches in the remote repository.

Example:

The following example uses the `git branch` command along with the -r option to display the names of all branches on the remote repository:

```
$ git branch -r
  origin/HEAD -> origin/main
  origin/main
  origin/my_feature
  origin/your-feature
```

Viewing all branches

```
git branch -a
```

Displays all branches both on the local and remote repositories.

Example:

The following example displays all branches, local and remote, for the repository associated with the current working directory. The `*` symbol indicates the current working branch, in this case `my_feature`:

```
$ git branch -a
  dev
  main
* my_feature
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
  remotes/origin/my_feature
  remotes/origin/your-feature
```

Creating a branch in the local repository

```
git branch <new_branch_name> <existing_branch_name>
```

Creates a new branch. If the optional parameter `<existing_branch_name>` is not provided, the new branch is derived from the current working branch.

Example:

The following example creates the a branch named `dev` that has the directories and files from the existing branch named `main`:

```
$ git branch dev main
```

Changing branches

```
git checkout <branch_name>
```

Retrieves the files in the branch named `<branch_name>` in the local repository. Once `git checkout` is called, developers can work on the files in that branch.

Example:

The following example changes the current working branch to the branch named `dev`. The `checkout` command is followed by a `git branch` command to verify the branch change. The `*` symbol indicates the current working branch, in this case `dev`.

```
$ git checkout dev
Switched to branch 'dev'

$ git branch
* dev
  main
  my_feature
```

Working with content

The following sections describe the various `git` commands you can use to inspect and manage files in a local repository.

Determining the status of the local filesystem

```
git status [options] <directory_or_filename>
```

Reports the status of the current filesystem associated with the local repository. The `<directory_or_filename>` parameter is optional. If no directory or filename is provided, the status of the present working directory is reported.

Example:

The following example uses `git status` to report the status of the file and directories in the present working directory, in comparison to the state of the local repository. The final line of output shows that the local repository is currently in sync with the working directory:

```
$ git status
On branch dev
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   git_cheat_sheet/readme.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Adding new or updated content to staging

```
git add [options] <files or directories>
```

Adds content to the staging environment from the current branch in the local computer's working directory.

Example:

The following example creates a directory named `git_cheat_sheet` in the current branch. Then a file named `readme.md` is added to the directory. Finally, the git add command adds the contents of the directory to the local staging environment:

```
$ mkdir git_cheat_sheet
$ touch ./git_cheat_sheet/readme.md
$ git add ./git_cheat_sheet/
```

Committing new or updated content to the local repository

```
git commit [options] <files or directories>
```

Commits content from the staging environment to the local repository.

Example:

The following example uses the `git commit` command to commit the file `./git_cheat_sheet/readme.md` to the local repository along with a descriptive message: "adding new file for git-cheat-sheet":

```
$ git commit -m "adding new file for git-cheat-sheet" ./git_cheat_sheet/
readme.md
[dev 0c0fb31] adding content for git-cheat-sheet
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git_cheat_sheet/readme.md
```

Pushing new or updated content to the remote repository

```
git push [options] <remote_repository>
```

Uploads content from the local repository to the remote repository. The `<remote_repository>` parameter is optional. If no remote repository is defined, content is pushed to the repository associated with the current working directory. If the remote repository has updates that are not reflected in the local repository, the `push` command fails with an error message.

Example:

The following example uploads all content committed to the local repository to the default remote repository associated with the current working directory:

```
git push
```

Rolling a file back from the staging environment

```
git restore [options] <filename>
```

Rolls back a file to its previous state under version control.

Example:

The following example uses `git add` to add a file named `config.json` to the staging environment, and then uses `git status` to inspect the state of the file, which is now awaiting a commit.

Then the command `git restore` is used with the `--staged` option to remove the `config.json` file from the staging environment. The `git status` command is called again to reveal that the file `config.json` is no longer part of the staging environment:

```
$ git add config.json

$ git status
On branch dev
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

$ git restore --staged config.json

$ git status
On branch dev
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
      modified:   config.json

no changes added to commit (use "git add" and/or "git commit -a")
```

Removing files that were added but not staged

```
git clean [options] <filename>
```

Rolls one or more files back to a particular state according to particular context with the repository–local or remote. For example, rolling back to the last commit.

Example:

The following example displays the files in the working directory associated with a local repository. Then a new file named `config.json` is added to the directory. Finally the command `git clean` is called with the `-f` option to reset the directory to the local repository's original state, removing the added file. The `ls -1` command is called again to show that the file `config.json` has been removed from the working directory:

```
$ ls -1
readme.md

$ echo "{"isCool": 1}" > config.json

$ ls -1
config.json
readme.md

$ git clean -f
Removing config.json

$ ls -1
readme.md
```

Rolling back to the most recent commit

```
git revert [options] <commit_uuid>
```

Reverts the filesystem associated with a local `.git` repository to a previous state. Also updates changes to the local `.git` log.

Example:

The following example displays the files in the directory associated with a local repository. Then a new file named `newfile.txt` is added to the directory and committed to the local repository. The contents of the directory are listed again. The `git log` command shows the latest Git activity.

Then `git revert 98d7128 --no-edit` reverts the state of the directory to the point before the commit `98d7128` was executed. The contents of the reverted directory are displayed. The reversion activity has been captured and is displayed by calling `git log`:

```
$ ls -1
config.json
readme.md

$ touch newfile.txt
$ git add .
$ git commit -m "adding a file named newfile.txt"

$ ls -1
config.json
newfile.txt
readme.md

$ git log --oneline
98d7128 (HEAD -> main) adding a file named newfile.txt
e5cf841 adding configuration file
665ecf1 (origin/your-feature, origin/main, origin/dev, origin/HEAD)
reorganizing repo structure

$ git revert 98d7128 --no-edit
Removing newfile.txt
[main 3f10573] Revert "adding a file named newfile.txt"
 Date: Tue Feb 15 09:13:06 2022 -0800
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 newfile.txt

$ ls
config.json
readme.md

$ git log --oneline
3f10573 (HEAD -> main) Revert "adding a file named newfile.txt"
98d7128 adding a file named newfile.txt
e5cf841 adding configuration file
665ecf1 (origin/your-feature, origin/main, origin/dev, origin/HEAD)
reorganizing repo structure
```

Rolling back to the most recent commit

The following sections describe how to merge files between branches, rebase files between branches, and invoke the a `diff` tool when merge conflicts occur.

git merge

```
git merge [options] <target_branch> <branch_to_merge_from>
```

Merges the files and directories from `<branch_to_merge_from>` into the `<target_branch>`. If the `<target_branch>` parameter is not provided, the files and directories in the `<branch_to_merge_from>` are merged into the current branch.

Example:

The following example shows the current branch as well as the files in that branch. The dev branch has two files, `newfile.txt` and `readme.md`.

Then the branch is changed to `main`. The main branch has one file, `readme.md`. The command `git merge dev --no-edit` merges the files from the dev branch into the the current `main` branch. The option `--no-edit` is used to avoid having to write a message describing the merge. Finally, the `ls -1` command shows that the merge successfully added `newfile.txt` from the dev branch to main:

```
$ git branch
* dev
  main

$ ls -1
newfile.txt
readme.md

$ git checkout main

$ ls -1
readme.md

$ git merge dev --no-edit
Merge made by the 'recursive' strategy.
 newfile.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newfile.txt

$ ls -1
newfile.txt
readme.md
```

git rebase

```
git rebase [options] <other_branch>
```

Merges one repository onto another while also transferring the commits from the merge-from branch onto the merge-to branch. Operationally, Git can delete commits from one branch while adding them to another.

Example:

The following example checks out the branch `dev` and then rebases the updates made in the branch `new_feature` onto the branch `dev`. The commits that were part of `new_feature` are now part of `dev`:

```
$ git checkout dev
Switched to branch 'dev'

$ git rebase new_feature
Successfully rebased and updated refs/heads/dev.
```

git mergetool

```
git mergetool <tool>
```

Invokes an editing tool to resolve merge conflicts between files. If no `<tool>` parameter is provided, `mergetool` uses the globally configured merge editor. You can register a merge editor using the following command:

```
git config --global merge.tool vimdiff
```

In this case, the command indicates that `vimdiff` should be used by default to show diffs between branches.

You also use an alterative merge editor by using the `--tool` option.

Example:

The following example creates a merge conflict and then invokes mergetool using the `--tool` option to run merge editor `vimdiff`.

Note:: The `vimdiff` tool has to be installed on the computer prior to using it with `mergetool`. The output that follows is an emulation of the command-line interface for `vimdiff`.

```
$ git merge dev
Auto-merging newfile.txt
CONFLICT (content): Merge conflict in newfile.txt
Automatic merge failed; fix conflicts and then commit the result

$ git mergetool --tool=vimdiff

Hit return to start merge resolution tool (vimdiff):
+------------------------------------------------------+
| MAIN            | BASE          | DEV                |
+-----------------|---------------|--------------------+
| I am cool       | <<<<<<< HEAD | He was cool        |
|                 |  I am cool    |                    |
|                 |=======        |                    |
|                 |I was cool     |                    |
|                 |>>>>>>> dev    |                    |
+------------------------------------------------------+
```

Rolling back to the most recent commit

The following sections show some ways to keep track of changes in Git.

git blame

```
git blame [options] <file_of_interest>
```

Displays a list of recent commits on a file by committer along with changes in the file. By default each list item displays the commit UUID, the committer, the date of commit, the locale, and the actual content added.

Example:

The following example uses `git blame` to list recent commits on the file `readme.md`. Note that commit `2a86f76f` (the third line in the output) was the most recent change, because its timestamp `2022-02-16 08:41:07` is the most recent:

```
$ git blame readme.md
c9b791ce (John Lennon 2022-02-08 11:00:30 -0800 1) # RHEL 8 Cheat Sheet:
Additional Resources
c9b791ce (John Lennon 2022-02-08 11:00:30 -0800 2)
2a86f76f (Mick Jagger 2022-02-16 08:41:07 -0800 3) Contains a list of
additional resources.
4dfb6c37 (Mick Jagger 2022-02-16 08:32:12 -0800 4)
4dfb6c37 (Mick Jagger 2022-02-16 08:32:12 -0800 5) It is still a work in
progress.
4dfb6c37 (Mick Jagger 2022-02-16 08:32:12 -0800 6)
```

git tag

```
git tag [options] <tag_name>
```

Tags a repository. This command is usually used to mark a release. If the `<tag_name>` parameter is not provided, the command displays a list of existing tags.

Example:

The following example uses git tag to declare a tag with the value `v1.0` The option `-m` is used to apply a message to the tag:

```
$ git tag v1.0 -m "first release of project"
```

The following example uses `git tag` to display a list of existing tags on the repository. The `-n` option is used to show the user-defined message associated with each tag:

```
$ git tag -n
v1.0            first release of project
```