Git is the go-to version control tool for most software developers because it allows them to efficiently manage their source code and track file changes while working with a large team. In fact, Git has so many uses that memorizing its various commands can be a daunting task, which is why we've created this git cheat sheet.

This guide includes an introduction to Git, a glossary of terms and lists of commonly used Git commands. Whether you're having trouble getting started with Git, or if you're an experienced programmer who just needs a refresher, you can refer to this cheat sheet for help.

What is Git?

If you work in web or software development, then you've probably used Git at some point. It remains the most widely used open source distributed version control system over a decade after its initial release. Unlike other version control systems that store a project's full version history in one place, Git gives each developer their own repository containing the entire history of changes. While extremely powerful, Git has some complicated command line syntax that may be confusing at first. Nonetheless, once broken down they're all fairly straightforward and easy to understand.

Git Glossary

Before you get started with Git, you need to understand some important terms:

Branch

Branches represent specific versions of a repository that "branch out" from your main project. Branches allow you to keep track of experimental changes you make to repositories and revert to older versions.

Commit

A commit represents a specific point in your project's history. Use the commit command in conjunction with the git add command to let git know which changes you wish to save to the local repository. Note that commits are not automatically sent to the remote server.

Checkout

Use the git checkout command to switch between branches. Just enter git checkout followed by the name of the branch you wish to move to, or enter git checkout master to return to the master branch. Mind your commits as you switch between branches.

Fetch

The git fetch command copies and downloads all of a branch's files to your device. Use it to save the latest changes to your repositories. It's possible to fetch multiple branches simultaneously.

Fork

A fork is a copy of a repository. Take advantage of “forking” to experiment with changes without affecting your main project.

Head

The commit at the tip of a branch is called the head. It represents the most current commit of the repository you’re currently working in.

Index

Whenever you add, delete or alter a file, it remains in the index until you are ready to commit the changes. Think of it as the staging area for Git. Use the git status command to see the contents of your index. Changes highlighted in green are ready to be committed while those in red still need to be added to staging.

Master

The master is the primary branch of all your repositories. It should include the most recent changes and commits.

Merge

Use the git merge command in conjunction with pull requests to add changes from one branch to another.

Origin

The origin refers to the default version of a repository. Origin also serves as a system alias for communicating with the master branch. Use the command git push origin master to push local changes to the master branch.

Pull

Pull requests represent suggestions for changes to the master branch. If you’re working with a team, you can create pull requests to tell the repository maintainer to review the changes and merge them upstream. The git pull command is used to add changes to the master branch.

Push

The git push command is used to update remote branches with the latest changes you’ve committed.

Rebase

The git rebase command lets you split, move or get rid of commits. It can also be used to combine two divergent branches.

Remote

A remote is a clone of a branch. Remotes communicate upstream with their origin branch and other remotes within the repository.

Repository

Git repositories hold all of your project's files including branches, tags and commits.

Stash

The git stash command removes changes from your index and "stashes" them away for later. It's useful if you wish to pause what you're doing and work on something else for a while. You can't stash more than one set of changes at a time.

Tags

Tags provide a way to keep track of important commits. Lightweight tags simply serve as pointers while annotated tags get stored as full objects.

Upstream

In the context of Git, upstream refers to where you push your changes, which is typically the master branch.

See the Git docs reference guide for more in depth explanations of Git related terminology.

Commands for Configuring Git

Set the username:

```
git config –global user.name
```

Set the user email:

```
git config –global user.email
```

Create a Git command shortcut:

```
git config –global alias.
```

Set the preferred text editor:

```
git config –system core.editor
```

Open and edit the global configuration file in the text editor:

```
git config –global –edit
```

Enable command line highlighting:

```
git config –global color.ui auto
```

Commands for Setting Up Git Repositories

Create an empty repository in the project folder:

```
git init
```

Clone a repository from GitHub and add it to the project folder:

```
git clone (repo URL)
```

Clone a repository to a specific folder:

```
git clone (repo URL) (folder)
```

Display a list of remote repositories with URLs:

```
git remote -v
```

Remove a remote repository:

```
git remote rm (remote repo name)
```

Retrieve the most recent changes from origin but don’t merge:

```
git fetch
```

Retrieve the most recent changes from origin and merge:

```
git pull
```

Commands for Managing File Changes

Add file changes to staging:

```
git add (file name)
```

Add all directory changes to staging:

```
git add .
```

Add new and modified files to staging:

```
git add -A
```

Remove a file and stop tracking it:

```
git rm (file_name)
```

Untrack the current file:

```
git rm –cached (file_name)
```

Recover a deleted file and prepare it for commit:

```
git checkout <deleted file name>
```

Display the status of modified files:

```
git status
```

Display a list of ignored files:

```
git ls-files –other –ignored –exclude-standard
```

Display all unstaged changes in the index and the current directory:

```
git diff
```

Display differences between files in staging and the most recent versions:

```
git diff –staged
```

Display changes in a file compared to the most recent commit:

```
git diff (file_name)
```

Commands for Declaring Git Commits

Commit changes along with a custom message:

```
git commit -m "(message)"
```

Commit and add all changes to staging:

```
git commit -am "(message)"
```

Switch to a commit in the current branch:

```
git checkout <commit>
```

Show metadata and content changes of a commit:

```
git show <commit>
```

Discard all changes to a commit:

```
git reset –hard <commit>
```

Discard all local changes in the directory:

```
git reset –hard Head
```

Show the history of changes:

```
git log
```

Stash all modified files:

```
git stash
```

Retrieve stashed files:

```
git stash pop
```

Empty stash:

```
git stash drop
```

Define a tag:

```
git tag (tag_name)
```

Push changes to origin:

```
git push
```

Commands for Git Branching

Display a list of all branches:

```
git branch
```

Make a new branch and switch to it:

```
git checkout -b <branchname>
```

Switch to a branch:

```
git checkout <branchname>
```

Delete a branch:

```
git branch -d <branchname>
```

Merge a different branch with your active branch:

```
git merge <branchname>
```

Fetch a branch from the repository:

```
git fetch remote <branchname>
```

View merge conflicts between branches:

```
git diff <sourcebranch> <targetbranch>
```

Preview changes before merging branches:

```
git diff <sourcebranch> <targetbranch>
```

Push all local branches to a designated remote repository:

```
git push –all
```

Git Tips

Knowing all the Git commands won't get you far if you don't know how to make the most of them. Here are some version control best practices to follow:

1. Commit Often

Keep your commits small by committing changes as often as possible. This makes it easier for team members to integrate their work without encountering merge conflicts.

2. Test, Then Commit

Never commit incomplete work. Always test your changes before sharing your code with others.

3. Use Commit Messages

Write commit messages to let other team members know what kind of changes you made. Be as descriptive as possible so your teammates know exactly what to look for.

4. Branch Out

Take full advantage of branches to help you keep track of different lines of development. Don't be afraid to go out on a limb and create a new branch to experiment with new features and ideas.

5. Settle on a Common Workflow

There are several different ways to set up your Git workflow. Whichever one you choose, make sure you and your teammates are on the same page from the very beginning.

Summary

Unless you have an amazing photographic memory, memorizing every single Git command would be a challenge, so don't bother trying to learn them all. You'll always have this guide to reference when you need a specific command. You may even want to create your own Git cheat sheet with the commands you use most frequently.