Getting Started with Git

TABLE OF CONTENTS

PREFACE

This cheatsheet serves as your trusty companion on your Git journey. Whether you're a seasoned developer looking for a quick reference or a newcomer navigating the Git landscape, you'll find valuable insights, commands, and tips within these pages.

INTRODUCTION

Git is a distributed version control system (DVCS) that has gained immense popularity for its speed, flexibility, and efficiency in managing source code and collaborating on software projects. Unlike traditional version control systems like Subversion (SVN) and CVS, Git employs a fundamentally different approach to tracking changes in code.

KEY DIFFERENTIATORS

- **Distributed Version Control**: Git is a distributed version control system, which means that every developer working with Git has a complete copy of the entire repository, not just the latest code snapshot. This decentralized nature makes it more robust and allows developers to work offline, fostering collaboration without depending on a central server.
- **Branching and Merging**: Git excels at branching and merging. It enables developers to create lightweight branches effortlessly, making it easy to experiment and work on new features without affecting the main codebase. Merging branches is also straightforward and minimizes conflicts efficiently.

ADVANTAGES OF GIT

- **Speed**: Git is exceptionally fast when it comes to committing changes, branching, merging, and searching through the commit history. This speed ensures a smooth and efficient development process.
- **Branching**: Git's branching model promotes parallel development, making it easier to manage multiple features or bug fixes simultaneously.
- **Data Integrity**: Git uses a robust content-addressable storage system to ensure data integrity. Once data is committed, it is virtually impossible to change it without leaving a trace.
- **Collaboration**: Git fosters seamless collaboration among developers, allowing them to work independently and then merge their changes effortlessly.
- **Wide Adoption**: Git is widely adopted across the software development industry and is supported by numerous hosting services like GitHub and GitLab.

DISADVANTAGES OF GIT

- **Learning Curve**: Git has a steeper learning curve compared to simpler version control systems, especially for beginners. The multitude of commands and options can be overwhelming.
- **Complex History**: The commit history in Git can become complex, particularly in large projects, making it challenging to track the history effectively.
- **Git Terminology**: Some Git terminology, such as "rebasing" and "detached HEAD," can be confusing for newcomers.

This cheat sheet will guide you through the essential Git commands and concepts, equipping you with the knowledge to leverage Git's power in your projects.

DOWNLOAD AND INSTALL

Below are instructions for downloading and installing Git on Windows, Linux, and macOS.

WINDOWS

- Download Git Windows installer from the official website for Windows at https://gitforwindows.org/.
- Run the downloaded installer. Follow the installation wizard's prompts:
  - Choose your preferred installation directory.
  - Select components to install (the default selections are usually fine).
  - Choose an editor (the default is usually "Use the Nano editor by default," but you

can select a different text editor if you prefer).

- Choose the terminal emulator used by Git (the default is "Use Windows' default console window," but you can choose "Use MinTTY" for a more Unix-like experience).
- Configure line ending conversions. The default setting, "Checkout as-is, commit Unix-style line endings," is typically suitable.
- Choose how Git should handle line endings in your working directory (usually, "Use Windows-style line endings" is the recommended option for Windows).

- Click "Install" to start the installation process.
- After installation, you can open Git Bash or use Git via the Windows Command Prompt or PowerShell.

LINUX

Ubuntu/Debian

- Open a terminal.
- Update your package list: `sudo apt update`
- Install Git: `sudo apt install git`

CentOS/Fedora

- Open a terminal.
- Install Git using the package manager (`yum` on CentOS or `dnf` on Fedora): `sudo yum install git` or `sudo dnf install git`

MACOS

- On macOS, you can use the Terminal, or you can install Git using the Homebrew package manager if you have it installed. If you don't have Homebrew, you can install it from https://brew.sh/.
- Open the Terminal.
- Install Git using Homebrew: `brew install git`
- Verify the installation by running: `git --version`

If you want to know all the potential nuances of a Git setup, here are several in-depth tutorials on installing and setting up Git on your machine:

- Getting Started and Installing Git
- Setting Up Git
- Installing Git on Mac, Windows, and Linux

Working with remote repositories is one of the primary features of Git. Git repositories are most typically shared via SSH. So to enable repository sharing over SSH and Gitosis follow the instructions documented at Git Community Book.

DEFINING USER CREDENTIALS

Git does not directly support repository authentication or authorization. It delegates this to the communication protocol (commonly SSH) or the hosting operating system via file system permissions. Thus, the user information provided during your first Git setup (per machine) is used for properly crediting your code contributions.

Issue the following commands to setup your Git user.

- `git config --global user.name "Your Name"`: Set your Git username globally.
- `git config --global user.email "youremail@example.com"`: Set your Git email globally.

GETTING STARTED

Using a command prompt, navigate to either a blank folder or the top folder of an existing project that you want to put under version control. Initialize the directory as a Git repository by typing: `git init`. Git builds in place just a single directory, namely `.git`, and uniquely stores all the metadata and repository history in it.

If you want to build on someone else's work, you can clone a remote repository by using: `git clone <repository URI>`. Below are two examples of cloning a remote repository using SSH and HTTP protocols.

```
# ssh protocol (requires SSH
credentials to be established):
$ git clone git@github.com:kioub78
/syntaxhighlighter-amplified.git
```

```
# https protocol:
$ git clone https://github.
com/kioub78/syntaxhighlighter-
amplified.git
```

When you run `git clone`, several actions are performed:

- **Connection to the Remote Repository**: Git establishes a connection to the remote repository specified in the URI provided as an argument to `git clone`. This is typically hosted on a Git hosting service like GitHub, GitLab, Bitbucket, or a custom Git server.
- **Downloading the Repository**: Git downloads the entire content of the remote repository to your local machine. This includes all branches, commits, files, directories, and commit history. It essentially creates a complete snapshot of the remote repository, preserving the entire history.
- **Creating a New Directory**: Git creates a new directory/folder on your local machine with the same name as the remote repository. If you specify a directory name after the URI, Git will use that name instead.
- **Initializing a Local Git Repository**: Within the newly created directory, Git initializes a local Git repository. This local repository is a complete repository in itself, with its own `.git` folder that contains all the necessary configuration and metadata for version control.
- **Setting Up Remote Tracking**: Git sets up a remote tracking branch called `origin/master` (assuming "master" is the default branch) that tracks the remote repository's default branch. This allows you to fetch and pull updates from the remote repository easily.
- **Checking Out the Default Branch**: Git switches to the default branch (usually "master" or "main") as the current working copy.

After these actions, the local copy of the remote Git repository is initialized, and ready for you to start using Git commands to interact with it. You can create new branches, make changes, commit them, and push your changes back to the remote repository, effectively participating in the collaborative development of the project.

GIT TREEISH

In Git, a "treeish" refers to a reference that identifies a tree object or a commit object within the Git repository. Tree objects represent directory structures and file states, while commit objects represent a snapshot of the entire repository at a specific point in time. Treeish references can be used in various Git commands to navigate and operate on the Git repository. Understanding how to use them effectively is crucial for managing and exploring your project's history.

Treeish References:

- **Commit Hash**: A unique 40-character SHA-1 hexadecimal string that identifies a specific commit in the Git history. This is the most precise way to specify a commit.
- **Branch Name**: The name of a branch in your Git repository. A branch name is a symbolic reference that points to the latest commit in that branch.
- **Tag Name**: A tag is a reference to a specific commit. You can use tag names to refer to specific points in your Git history, such as releases or milestones.
- **Ancestry References**: These are references that specify a range of commits. For example, `A..B` refers to all commits reachable from B but not from A.

Commonly Used Treeish References:

Treeish Reference	Explanation
`HEAD`	Refers to the latest commit on the current branch.
`HEAD^` or `HEAD~1`	Refers to the parent of the latest commit.
`HEAD~2`	Refers to the grandparent of the latest commit.
`origin/master`	Refers to the latest commit on the "master" branch in the remote repository called "origin."

Examples:

- `git log <commit-hash>`: View the commit history starting from the specified commit.
- `git diff <branch-name>`: Compare the current branch with the specified branch.
- `git checkout <tag-name>`: Switch to a specific commit or tag.
- `git diff HEAD~2..HEAD~1`: Compare changes between the grandparent and parent of the latest commit.

BASIC WORKFLOW

- `git mv <file1> <file2>`: Move and/or rename a file and track its new destination.
- `git rm <file>`: Remove a file from the current state of the branch.
- `git status`: Check the status of your working directory (modified, new, deleted, or untracked files and folders).
- `git add <filename, directory name, or wildcard>`: Stage (start tracking) changes for commit. You can use a directory name or wildcard for multiple file/folder selections. Use the `-i` option for interactive mode (prompts for each file to be added). Remember, only staged changes can be committed.
- `git commit -m "Your commit message"`: Transactionally saves staged changes to the local repository. A commit message is required. Leaving the message blank will abort the commit and leave the staged blobs in place. In case you desire to make changes to the last commit's message you can use `git commit --amend`.
- `git show`: View statistics and facts about the last commit.
- `git log`: View the commit history. You may use the `--since` flag to apply time restrictions e.g `--since=yesterday` or `--since=3weeks`.

BRANCHING

- `git branch -a`: List all branches (local and remote) in the repository.
- `git branch <branch-name>`: Create a new branch.
- `git branch <branch-name> <from>`: Create a new branch on a specific branch/tag/commit. To take a remote branch as the basis for your new local branch, you can use the `--track` option e.g. `git branch --track <branch-name> origin/<base-branch>`. Remote branches are read-only until "tracked" and copied to a local branch.
- `git checkout <branch-name>`: Switch to an existing branch.
- `git merge <branch-name>`: Merge changes from one branch into the current working branch. Multiple branches can be defined separated by space. In case of conflicts, the files involved are internally marked with >>>>>>>>> and <<<<<<<<< around the conflicting portions of their contents. Once manually updated, you can use `git add` to stage the resolved files and then `git commit` to commit the changes in the current working branch as usual.

REMOTE REPOSITORIES

- `git remote`: List remote repositories. Use the `-v` flag to get full repository addresses rather than names only.
- `git remote add <name> <URI>`: Add a remote repository residing at the specified URI, under the specified name. In Git terms, this name/URI assignment is called a **remote**. **Remote** references can be used in Git commands to identify specific remote repositories. Git automatically creates a remote called **origin** when you clone a remote repository.
- `git fetch <remote> <branch>`: Retrieve changes from a remote repository and branch without merging them into your current local working branch. Just stores the blobs in `.git` directory waiting for further instructions. "<remote>" denotes the assigned name of the remote repository as defined with a `git remote add` command.
- `git pull <remote> <branch>`: Retrieve changes from a remote repository and branch and then merges them into the current local working branch. "<remote>" denotes the assigned name of the remote repository as defined with a `git remote add` command. Pulling is the combination of a `git fetch` and a `git merge` all in one seamless action.
- `git push <remote> <branch>`: Send committed changes from your current local working branch to a remote repository and branch.

"<remote>" denotes the assigned name of the remote repository as defined with a `git remote add` command. You need to have sufficient open permissions as to allow you to write to the remote branch. Use `--tags` flag to push tags also.

RESOLVING CONFLICTS

- `git diff`: Show the differences between the working directory and the last commit - Includes everything unstaged (not git add'ed). Use the `-cached` flag to get everything staged (git add'ed) instead.
- `git diff <commit1> <commit2>`: Show differences between two commits - Includes everything unstaged (not git add'ed) and staged (git add'ed).
- `git merge --abort`: Abort a merge in progress.
- Manually edit conflicted files and use `git add` to mark conflicts as resolved.
- `git blame <file>`: Annotate each line of a source file with the name and date it was last modified.

STASHING CHANGES

- `git stash`: Temporarily save/push changes that are not ready for commit onto a stack. This is particularly useful when your changes are in an incomplete state so you are not ready to commit them, but you need to temporarily return to the last committed state e.g. a fresh checkout.
- `git stash pop`: Apply the most recent stashed changes back into the working copies of the files and remove them from the stash list.
- `git stash list`: List all stashes.

REVERTING CHANGES

- `git reset <file>`: Unstage changes in a file, restoring the working copy to the last committed state of the file.
- `git reset --hard <commit>`: Reset to a specific commit, discarding all changes after that commit.
- `git revert <commit>`: Create a new commit that undoes the changes made in a previous commit.

TAGS

- `git tag`: List all tags.
- `git tag <tag-name> <commit>`: Create a new tag referencing the specified commit hash. Omit the commit attribute to reference the last commit.
- `git tag -a <tag-name> -m "Tag message"`: Create an annotated tag with a message. Annotated tags are stored as full objects in the Git database. They're checksummed; contain the tagger name, email, and date and have a tagging message required.

GITIGNORE

Create a `.gitignore` file to specify which files and directories should be ignored by Git. An example is provided below. More information is available here.

```
# Ignore specific files
filename.txt
secret.key

# Ignore all files in certain
directories
/logs/
/tmp/
/node_modules/
/build/

# Ignore all files with specific
extensions
*.log
*.tmp

# Ignore hidden files
.*

# Ignore a specific directory but
not its subdirectories
/logs/

# Ignore files with a specific
prefix
debug-*.log
```

```
# Negate patterns (exclude specific files)
# Ignore all .log files except important.log
*.log
!important.log

# Ignore macOS and Windows system files
.DS_Store
Thumbs.db
```

HELPFUL COMMANDS

- `git help <command>`: Get help for a specific Git command.
- `git --version`: Check your Git version.
- `git clean -n`: Dry run of removing untracked files.
- `git clean -f`: Remove untracked files.
- `git gc`: prune any orphaned blobs from the tree.
- `git fsck`: check the integrity of the repository.
- `git grep`: search through history.

GUIS AND IDES

Bundled with the standard Git distribution are two user interfaces Gitk and Git-Gui. Nevertheless there is a plethora of 3rd party Git GUIs available. Furthermore the most popular IDEs and code editors including IntelliJ, Eclipse, NetBeans, Sublime Text, Atom, VS Code, Vim, and Emacs all offer native or simple plugin support for Git. Below is a list containing some of the most popular Git GUIs.

Git Gui	Description
GitHub Desktop	GitHub's official desktop application for managing Git repositories. It offers an intuitive interface for performing common Git tasks and easy integration with GitHub.
Sourcetree	A free Git GUI for Windows and macOS that provides a visual way to manage Git repositories, offering features like Git-flow support and repository visualization.
GitKraken	A cross-platform Git client with a sleek and visually appealing interface. It includes collaboration features, Git-flow support, and integrations with various services.
TortoiseGit	A Windows-specific Git GUI that integrates with Windows File Explorer. It provides context menu integration, icon overlays, and many Git features within Explorer.
SmartGit	A cross-platform Git client with a user-friendly interface and a focus on simplicity. It supports Git and Mercurial, and it offers various features for Git workflow.
GitExtensions	A Windows Git client that integrates with Windows File Explorer. It includes Git Bash, Git command-line, and Visual Studio integration, among other features.
Magit	A Git interface for Emacs, making it convenient for Emacs users to interact with Git repositories without leaving their preferred text editor.

Git Gui	Description
Git Cola	A cross-platform Git GUI for those who prefer a lightweight and straightforward interface. It provides basic Git features without being overly complex.
Tower	A Git client for macOS and Windows that focuses on providing a powerful and user-friendly interface. It offers features like Git-flow, submodule support, and integrations with popular Git hosting services.