I'm not a bot

The open source Git project just released Git 2.50 with features and bug fixes from 98 contributors, 35 of them new. We last caught up with you on the latest in Git back when 2.49 was released. Before we get into the details of this latest release, we wanted to remind you that Git Merge, the conference for Git users and developers is back this year on September 29-30, in San Francisco. Git Merge will feature talks from developers working on Git, and in the Git ecosystem. Tickets are on sale now; check out the website to learn more. With that out of the way, let’s take a look at some of the most interesting features and changes from Git 2.50. When we covered Git 2.43, we talked about newly added support for multiple cruft packs. Git 2.50 improves on that with better command-line ergonomics, and some important bugfixes. In case you’re new to the series, need a refresher, or aren’t familiar with cruft packs, here’s a brief overview:Git objects may be either reachable or unreachable. The set of reachable objects is everything you can walk to starting from one of your repository’s references: traversing from commits to their parent(s), trees to their sub-tree(s), and so on. Any object that you didn’t visit by repeating that process over all of your references is unreachable. In Git 2.37, Git introduced cruft packs, a new way to store your repository’s unreachable objects. A cruft pack looks like an ordinary packfile with the addition of an .mtimes file, which is used to keep track of when each object was most recently written in order to determine when it is safe1 to discard it.However, updating the cruft pack could be cumbersome–particularly in repositories with many unreachable objects–since a repository’s cruft pack must be rewritten in order to add new objects. Git 2.43 began to address this through a new command-line option: git repack --max-cruft-size. This option was designed to split unreachable objects across multiple packs, each no larger than the value specified by --max-cruft-size. But there were a couple of problems: If you’re familiar with git repack’s --max-pack-size option, --max-cruft-size’s behavior is quite confusing. The former option specifies the maximum size an individual pack can be, while the latter involves how and when to move objects between multiple packs. The feature was broken to begin with! Since --max-cruft-size also imposes on cruft packs the same pack-size constraints as --max-pack-size does on non-cruft packs, it is often impossible to get the behavior you want. For example, suppose you had two 100 MiB cruft packs and ran git repack --max-cruft-size=200M. You might expect Git to merge them into a single 200 MiB pack. But since --max-cruft-size also dictates the maximum size of the output pack, Git will refuse to combine them, or worse: rewrite the same pack repeatedly. Git 2.50 addresses both of these issues with a new option: --combine-cruft-below-size. Instead of specifying the maximum size of the output pack, it determines which existing cruft pack(s) are eligible to be combined. This is particularly helpful for repositories that have accumulated many unreachable objects spread across multiple cruft packs. With this new option, you can gradually reduce the number of cruft packs in your repository over time by combining existing ones together. With the introduction of --combine-cruft-below-size, Git 2.50 repurposed --max-cruft-size to behave as a cruft pack-specific override for --max-pack-size. Now --max-cruft-size only determines the size of the outgoing pack, not which packs get combined into it. Along the way, a bug was uncovered that prevented objects stored in multiple cruft packs from being “freshened” in certain circumstances. In other words, some unreachable objects don’t have their modification times updated when they are rewritten, leading to them being removed from the repository earlier than they otherwise would have been. Git 2.50 squashes this bug, meaning that you can now efficiently manage multiple cruft packs and freshen their objects to your heart’s content. [source, source] Back in our coverage of Git 2.47, we talked about preliminary support for incremental multi-pack indexes. Multi-pack indexes (MIDXs) act like a single pack *.idx file for objects spread across multiple packs. Multi-pack indexes are extremely useful to accelerate object lookup performance in large repositories by binary searching through a single index containing most of your repository’s contents, rather than repeatedly searching through each individual packfile. But multi-pack indexes aren’t just useful for accelerating object lookups. They’re also the basis for multi-pack reachability bitmaps, the MIDX-specific analogue of classic single-pack reachability bitmaps. If neither of those are familiar to you, don’t worry; here’s a brief refresher. Single-pack reachability bitmaps store a collection of bitmaps corresponding to a selection of commits. Each bit position in a pack bitmap refers to one object in that pack. In each individual commit’s bitmap, the set bits correspond to objects that are reachable from that commit, and the unset bits represent those that are not. Multi-pack bitmaps were introduced to take advantage of the substantial performance increase afforded to us by reachability bitmaps. Instead of having bitmaps whose bit positions correspond to the set of objects in a single pack, a multi-pack bitmap’s bit positions correspond to the set of objects in a multi-pack index, which may include objects from arbitrarily many individual packs. If you’re curious to learn more about how multi-pack bitmaps work, you can read our earlier post Scaling monorepo maintenance. However, like cruft packs above, multi-pack indexes can be cumbersome to update as your repository grows larger, since each update requires rewriting the entire multi-pack index and its corresponding bitmap, regardless of how many objects or packs are being added. In Git 2.47, the file format for multi-pack indexes became incremental, allowing multiple multi-pack index layers to be layered on top of one another forming a chain of MIDXs. This made it much easier to add objects to your repository’s MIDX, but the incremental MIDX format at the time did not yet have support for multi-pack bitmaps. Git 2.50 brings support for the multi-pack reachability format to incremental MIDX chains, with each MIDX layer having its own *.bitmap file. These bitmap layers can be used in conjunction with one another to provide reachability information about selected commits at any layer of the MIDX chain. In effect, this allows extremely large repositories to quickly and efficiently add new reachability bitmaps as new commits are pushed to the repository, regardless of how large the repository is. This feature is still considered highly experimental, and support for repacking objects into incremental multi-pack indexes and bitmaps is still fairly bare-bones. This is an active area of development, so we’ll make sure to cover any notable developments to incremental multi-pack reachability bitmaps in this series in the future.[source] This release also saw some exciting updates related to merging. Way back when Git 2.33 was released, we talked about a new merge engine called “ORT” (standing for “Ostensibly Recursive’s Twin”).ORT is a from-scratch rewrite of Git’s old merging engine, called “recursive.” ORT is significantly faster, more maintainable, and has many new features that were difficult to implement on top of its predecessor.One of those features is the ability for Git to determine whether or not two things are mergeable without actually persisting any new objects necessary to construct the merge in the repository. Previously, the only way to tell whether two things are mergeable was to run git merge-tree --write-tree on them. That works, but in this example merge-tree wrote any new objects generated by the merge into the repository. Over time, these can accumulate and cause performance issues. In Git 2.50, you can make the same determination without writing any new objects by using merge-tree’s new --quiet mode and relying on its exit code.Most excitingly in this release is that ORT has entirely superseded recursive, and recursive is no longer part of Git’s source code. When ORT was first introduced, it was only accessible through git merge’s -s option to select a strategy. In Git 2.34, ORT became the default choice over recursive, though the latter was still available in case there were bugs or behavior differences between the two. Now, 16 versions and two and a half years later, recursive has been completely removed from Git, with its author, Elijah Newren, writing: As a wise man once told me, “Deleted code is debugged code!” As of Git 2.50, recursive has been completely debugged deleted. For more about ORT’s internals and its development, check out this five part series from Elijah here, here, here, here, and here.[source, source, source] If you’ve ever scripted around your repository’s objects, you are likely familiar with git cat-file, Git’s purpose-built tool to list objects and print their contents. git cat-file has many modes, like --batch (for printing out the contents of objects), or --batch-check (for printing out certain information about objects without printing their contents).Oftentimes it is useful to dump the set of all objects of a certain type in your repository. For commits, git rev-list can easily enumerate a set of commits. But what about, say, trees? In the past, to filter down to just the tree objects from a list of objects, you might have written something like:$ git cat-file --batch-check='%(objecttype) %(objectname)' \   --buffer git push-stack Enumerating objects: 20, done. Counting objects: 100% (20/20), done. Delta compression using up to 16 threads Compressing objects: 100% (12/12), done. Writing objects: 100% (18/18), 1.61 KiB | 1.61 MiB/s, done. Total 18 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0) To D:\repos\temp\temp69 + 540f580...e12b308 feature/part-3 -> feature/part-3 (forced update) Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0) To D:\repos\temp\temp69 + 44a77b9...98143f9 feature/part-2 -> feature/part-2 (forced update) Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0) To D:\repos\temp\temp69 + 834f474...4480c40 feature/part-1 -> feature/part-1 (forced update) The above output shows that I had feature/part-3 checked out, and when I ran the git push-stack command, it pushed all of the branches to the default remote, origin. I showed in a previous post how I built this command up, so for now I'll just provide the Git alias configuration commands to run, so that you can create your own push-stack alias: git config --global alias.default-branch "!git symbolic-ref refs/remotes/origin/HEAD | sed 's@^refs/remotes/origin/@@'" git config --global alias.merge-base-origin '!f() { git merge-base ${1-HEAD} origin/$(git default-branch); };f ' git config --global alias.stack '!f() { BRANCH=${1-HEAD}; MERGE_BASE=$(git merge-base-origin $BRANCH); git log --decorate-refs=refs/heads --simplify-by-decoration --pretty=format:\"%(decorate:prefix=,suffix=,tag=,separator=%n)\" $MERGE_BASE..$BRANCH; };f ' git config --global alias.push-stack '!f() { BRANCH=${1-HEAD}; git stack $BRANCH | xargs -I {} git push --force-with-lease origin {}; };f ' This command makes it easy to push all the branches in your currently checked out stack to a remote. You can read more about the exact behaviour, assumptions, and options for push-stack in my previous post. git push-stack is particularly useful when you've made changes somewhere in the middle of your stack, or you've just rebased it. No matter how many branches are in your stack, simply run git push-stack and they're all pushed to the remote. If all is going to plan with your Git stack, each branch in your stack will be merged to main one at a time. Depending on how your repository is set up to merge (with or without a merge commit, with or without squashing), this could make it easier or harder to rebase your commits afterwards. Most people have the biggest problems rebasing a stack when a repository uses squashing to merge a PR. This is because Git no longer "understands" that a given branch is now part of the main branch. Let's say we have the following stack, which has already been pushed to the remote: The first PR, for branch feature/part-1, has been approved, and you merge it. As per the repository's settings, the branch is squashed to a single commit when it's merged to main, and the remote branch feature/part-1 is deleted. If you do a git fetch, the commit graph looks something like this: With feature/part-1 merged, you now need to rebase the remainder of the stack on top of origin/main. Unfortunately, if you run simply git rebase origin/main then you'll be hit with a bunch of merge conflicts, as Git can't tell that the commits "First sub-feature" and "Update for first-feature" have been merged into a single commit, "feature/part-1". To work around this you need to be more specific with your rebase command. You need to rebase all the commits between the top of the stack (feature/part-3) and feature/part-1onto origin/main. The key to this rebase operation is the --onto option. Assuming you have checked out feature/part-3, you can run: git rebase feature/part-1 --onto origin/main Assuming you have --update-refs enabled as I have previously encouraged, this moves the whole remainder of the stack for you: $ git rebase feature/part-1 --onto origin/main Successfully rebased and updated refs/heads/feature/part-3. Updated the following refs with --update-refs: refs/heads/feature/part-2 Leaving your local repository looking like this: All that remains is to clean up by pushing the stack and cleaning up the now defunct feature/part-1 branch: git push-stack git branch -D feature/part-1 The final result is that our local repository looks like this: and we're ready to continue with our stack! Just to reiterate, the magic here is the use of --onto in git rebase. You just need to know two things: : The "bottom" of the stack that you want to rebase. Can be a commit SHA or a branch. The feature/part-1 branch in our example : Where you want to rebase your stack onto. Typically origin/main or equivalent. and then use this construct: git rebase --onto With that, I think we've covered the majority of complexities that occur when working with stacked branches. If there are any challenging Git gymnastics you think I've glossed over or missed, do let me know in the comments! In this follow up post to my previous post about using stacked branches, I described how to handle some common scenarios. In particular, I described how to rebase a stack of branches after commits to the main branch, how to push a stack of rebased branches, and how to rebase a stack after one of your branches has been merged. These, coupled with making local changes to branches, are the most common scenarios I encounter working with stacked branches. Hopefully the techniques I describe in this post help you to use stacked branches in your own work Everything you need to know about Git, from getting started to advanced commands and workflows. Quick links: Git is a distributed version control software. Version control is a way to save changes over time without overwriting previous versions. Being distributed means that every developer working with a Git repository has a copy of that entire repository – every commit, every branch, every file. If you're used to working with centralized version control systems, this is a big difference! Whether or not you've worked with version control before, there are a few things you should know before getting started with Git: Branches are lightweight and cheap, so it's OK to have many of them Git stores changes in SHA hashes, which work by compressing text files. That makes Git a very good version control system (VCS) for software programming, but not so good for binary files like images or videos. Git repositories can be connected, so you can work on one locally on your own machine, and connect it to a shared repository. This way, you can push and pull changes to a repository and easily collaborate with others. The tools that make up the core Git distribution are written in C, Shell, Perl, and Tcl. You can find Git's source code on GitHub under git/git. Version control is very important – without it, you risk losing your work. With Git, you can make a "commit", or a save point, as often as you'd like. You can also go back to previous commits. This takes the pressure off of you while you're working. Commit often and commit early, and you'll never have that gut-sinking feeling of overwriting or losing changes. There are many version control systems out there – but Git has some major advantages. Like we mentioned above, Git uses SHA compression, which makes it very fast. Git can handle merge conflicts, which means that it's OK for multiple people to work on the same file at the same time. This opens up the world of development in a way that isn't possible with centralized version control. You have access to the entire project, and if you're working on a branch, you can do whatever you need to and know that your changes are safe. Speaking of branches, Git offers a lot of flexibility and opportunity for collaboration with branches. By using branches, developers can make changes in a safe sandbox. Instead of only committing code that is 100% sure to succeed, developers can commit code that might still need help. Then, they can push that code to the remote and get fast feedback from integrated tests or peer review. Without sharing the code through branches, this would never be possible. If you make a mistake, it's OK! Commits are immutable, meaning they can't be changed. (Note: You can change history, but it will create new replacement commits instead of editing the existing commits. More on that later!) This means that if you do make a mistake, even on an important branch, like main, it's OK. You can easily revert that change, or roll back the branch pointer to the commit where everything was fine. The benefits of this can't be overstated. Not only does it create a safer environment for the project and code, but it fosters a development environment where developers can be braver, trusting that Git has their back. If you're getting started with Git, a great place to learn the basic commands is the Git Cheat sheet. It's translated into many languages, open source as a part of the github/training-kit repository, and a great starting place for the fundamentals on the command line. Some of the most important and most used commands that you'll find there are: git clone [url]: Clone (download) a repository that already exists on GitHub, including all of the files, branches, and commits. git status: Always a good idea, this command shows you what branch you're on, what files are in the working or staging directory, and any other important information. git branch: This shows the existing branches in your local repository. You can also use git branch [branch-name] to create a branch from your current location, or git branch --all to see all branches, both the local ones on your machine, and the remote tracking branches stored from the last git pull or git fetch from the remote. git checkout [branch-name]: Switches to the specified branch and updates the working directory. git add [file]: Snapshots the file in preparation for versioning, adding it to the staging area. git commit -m "descriptive message": Records file snapshots permanently in the version history. git pull: Updates your current local working branch with all new commits from the corresponding remote branch on GitHub. git pull is a combination of git fetch and git merge. git push: Uploads all local branch commits to the remote. git log: Browse and inspect the evolution of project files. git remote -v: Show the associated remote repositories and their stored name, like origin. If you're looking for more GitHub-specific technical guidance, check out GitHub's help documentation or our GitHub for Developers series on YouTube. Depending on your operating system, you may already have Git installed. But, getting started means more than having the software! To get started, it's important to know the basics of how Git works. You may choose to do the actual work within a terminal, an app like GitHub Desktop, or through GitHub.com. (Note: while you can interact with Git through GitHub.com, your experience may be limited. Many local tools can give you access to the most widely used Git functionalities, though only the terminal will give you access to them all.) There are many ways to use Git, which doesn't necessarily make it easier! But, the fundamental Git workflow has a few main steps. You can practice all of these in the Introduction to GitHub Learning Lab course. The main branch is usually called main. We want to work on another branch, so we can make a pull request and make changes safely. To get started, create a branch off of main. Name it however you'd like – but we recommend naming branches based on the function or feature that will be the focus of this branch. One person may have several branches, and one branch may have several people collaborate on it – branches are for a purpose, not a person. Wherever you currently "are" (wherever HEAD is pointing, or whatever branch you're currently "checked out" to) will be the parent of the branch you create. That means you can create branches from other branches, tags, or any commit! But, the most typical workflow is to create a branch from main – which represents the most current production code. Once you've created a branch, and moved the HEAD pointer to it by "checking out" to that branch, you're ready to get to work. Make the changes in your repository using your favorite text editor or IDE. Next, save your changes. You're ready to start the commit! To start your commit, you need to let Git know what changes you'd like to include with git add [file]. Once you've saved and staged the changes, you're ready to make the commit with git commit -m "descriptive commit message". So far, if you've made a commit locally, you're the only one that can see it. To let others see your work and begin collaboration, you should "push" your changes using git push. If you're pushing from a branch for the first time that you've created locally, you may need to give Git some more information. git push -u origin [branch-name] tells Git to push the current branch, and create a branch on the remote that matches it with the same name – and also, create a relationship with that branch so that git push will be enough information in the future. By default, git push only pushes the branch that you've currently checked out to. Sometimes, if there has been a new commit on the branch on the remote, you may be blocked from pushing. Don't worry! Start with a simple git pull to incorporate the changes on the remote into your own local branch, resolve any conflicts or finish the merge from the remote into the local branch, and then try the push again. Pushing a branch, or new commits, to a remote repository is enough if a pull request already exists, but if it's the first time you're pushing that branch, you should open a new pull request. A pull request is a comparison of two branches – typically main, or the branch that the feature branch was created from, and the feature branch. This way, like branches, pull requests are scoped around a specific function or addition of work, rather than the person making the changes or the amount of time the changes will take. Pull requests are the powerhouse of GitHub. Integrated tests can automatically run on pull requests, giving you immediate feedback on your code. Peers can give detailed code reviews, letting you know if there are changes to make, or if it's ready to go. Make sure you start your pull requests off with the right information. Put yourself in the shoes of your teammates, or even of your future self. Include information about what this change relates to, what prompted it, what is already done, what is left to do, and any specific asks for help or reviews. Include links to relevant work or conversations. Pull request templates can help make this process easy by automating the starting content of the body of pull requests. Once the pull request is open, then the real fun starts. It's important to recognize that pull requests aren't meant to be open when work is finished. Pull requests should be open when work is beginning! The earlier you open a pull request, the more visibility the entire team has to the work that you're doing. When you're ready for feedback, you can get it by integrating tests or requesting reviews from teammates. It's very likely that you will want to make more changes to your work. That's great! To do that, make more commits on the same branch. Once the new commits are present on the remote, the pull request will update and show the most recent version of your work. Once you and your team decide that the pull request looks good, you can merge it. By merging, you integrate the feature branch into the other branch (most typically the main branch). Then, main will be updated with your changes, and your pull request will be closed. Don't forget to delete your branch! You won't need it anymore. Remember, branches are lightweight and cheap, and you should create a new one when you need it based on the most recent commit on the main branch. If you choose not to merge the pull request, you can also close pull requests with unmerged changes. If you're wondering where Git ends and GitHub begins, you're not alone. They are tied closely together to make working with them both a seamless experience. While Git takes care of the underlying version control, GitHub is the collaboration platform built on top of it. GitHub is the place for pull requests, comments, reviews, integrated tests, and so much more. Most developers work locally to develop and use GitHub for collaboration. That ranges from using GitHub to host the shared remote repository to working with colleagues and capitalizing on features like protected branches, code review, GitHub Actions, and more. The best place to practice using Git and GitHub is the Introduction to GitHub Learning Lab course. If you already know Git and need to sign up for a GitHub account, head over to github.com. Contribute to this article on GitHub. To use Git on the command line, you will need to download, install, and configure Git on your computer. You can also install GitHub CLI to use GitHub from the command line. For more information, see About GitHub CLI. If you want to work with Git locally, but do not want to use the command line, you can download and install the GitHub Desktop client. For more information, see About GitHub Desktop. If you do not need to work with files locally, GitHub lets you complete many Git-related actions directly in the browser, including: Quickstart for repositories Fork a repository Managing files Setting up Git Download and install the latest version of Git. Note Most Chrome OS devices from 2020 onwards now have a built-in Linux environment, which includes Git. To enable it, go to the Launcher, search for Linux, and click Turn on. If you are using an older Chrome OS device, another method is required: Install a terminal emulator such as Termux from the Google Play Store on your Chrome OS device. From the terminal emulator that you installed, install Git. For example, in Termux, enter apt install git and then type y when prompted. Set your username in Git. Set your commit email address in Git. Authenticating with GitHub from Git When you connect to a GitHub repository from Git, you will need to authenticate with GitHub using either HTTPS or SSH. Note You can authenticate to GitHub using GitHub CLI, for either HTTP or SSH. For more information, see gh auth login. Connecting over HTTPS (recommended) If you clone with HTTPS, you can cache your GitHub credentials in Git using a credential helper. For more information, see About remote repositories and Caching your GitHub credentials in Git. Connecting over SSH If you clone with SSH, you must generate SSH keys on each computer you use to push or pull from GitHub. For more information, see About remote repositories and Generating a new SSH key and adding it to the ssh-agent. Next steps You now have Git and GitHub all set up. You may now choose to create a repository where you can put your projects. Saving your code in a repository allows you to back up your code and share it around the world. Creating a repository for your project allows you to store code in GitHub. This provides a backup of your work that you can choose to share with other developers. For more information, see Quickstart for repositories. Forking a repository will allow you to make changes to another repository without affecting the original. For more information, see Fork a repository. Each repository on GitHub is owned by a person or an organization. You can interact with the people, repositories, and organizations by connecting and following them on GitHub. For more information, see Finding inspiration on GitHub. GitHub has a great support community where you can ask for help and talk to people from around the world. Join the conversation on GitHub Community. Git comes with built-in GUI tools (git-gui, gitk), but there are several third-party tools for users looking for a platform-specific experience. View GUI Clients → Various Git logos in PNG (bitmap) and EPS (vector) formats are available for use in online and print projects. View Logos → If you already have Git installed, you can get the latest development version via Git itself: git clone You can also always browse the current contents of the git repository using the web interface. Now that Git is installed, and it knows who you are, you can start using Git. Lets create our first repository Key Steps to Get Started Create a project folder Navigate to the folder Initialize a Git repository Creating Git Folder Start by creating a new folder for our project: mkdir myproject cd myproject mkdir creates a new directory. cd changes our working directory. Now we are in the correct directory and can initialize Git! Now that we are in the correct folder, we can initialize Git on that folder: git init Initialized empty Git repository in /Users/user/myproject/.git/ You just created your first Git Repository! What is a Repository? A Git repository is a folder that Git tracks for changes. The repository stores all your project's history and versions. What Happens When You Run git init? Git creates a hidden folder called .git inside your project. This is where Git stores all the information it needs to track your files and history. ls -a . .. .git On Windows, you may need to enable “Show hidden files” in File Explorer to see the .git folder. Troubleshooting git: command not found Solution: Make sure Git is installed and added to your PATH. Restart your terminal if needed. Permission denied Solution: Try running your terminal as administrator (Windows) or use sudo (macOS/Linux) if needed. Git Cheat Sheet is a comprehensive quick guide for learning Git concepts, from very basic to advanced levels. By this Git Cheat Sheet, our aim is to provide a handy reference tool for both beginners and experienced developers/DevOps engineers. This Git Cheat Sheet not only makes it easier for newcomers to get started but also serves as a refresher for experienced professionals.Git Cheat SheetIn this Git Cheat Sheet, we’ve covered all the basics to advanced Git commands that the developers required during the development and deployment process. Moreover, it is well-structured and categorized according to different use cases. It includes Git and GitHub, Git download, Git Configuration & Setup, Git commands, Git bash, Creating and Getting Git Projects, Git Snapshots, Branching and Merging in Git, Sharing and Updating in Git, Git Comparison, Managing History in Git, and more.What is Git?Git is the free and open-source distributed version control systems that's responsible for everything GitHub related that happens locally on your computer.Understanding Version Control Version control, also known as source control, is the technique of tracking and managing changes to codes and these are the systems that are software tools that enable software teams to manage modifications to source code as time passes. What is GitHub?GitHub is a widely-used Free-to-use cloud Storage platform with version control and many other essential features that specifically helps developers to manage and deploy their projects on GitHub.Benefits of Using GitHistory Tracking: Git allows you to track every change made in your project, including: who made the change and when it was made.Collaboration: Multiple developers can be able work on the same project at the same time, and Git efficiently manages the merging of changes in code. Branching and Merging: Git enables developers to create branches to work on new features or bug fixes and later merge them back into the main codebase.Offline Work: Git works offline, which means you can commit changes and work on your project even without an internet connection.Learn Git Cheat Sheet (Basic to Advanced Concepts) Learn Git Cheat Sheet (Basic to Advanced Concepts)Git Installation CommandsHere are the Git installation commands for different operating systems:CommandsDescriptionGit for Windows stand-alone installer. For more Details Read Here$ brew install gitInstall Git with Homebrew on Mac OS$ sudo port selfupdateInstall Git with MacPorts on Mac OS$ sudo apt-get install gitInstall Command for Linux$ git --versionShows the current version of your GitGit Configuration & SetupHere are Git configuration and setup commands:Commands Descriptiongit config --global user.name "Your Name"Set your username globally.git config --global user.email "youremail@example.com"Set your email globally.git config --global color.ui auto -Set to display colored output in the terminalgit helpDisplay the main help documentation, showing a list of commonly used Git commands.Initializing a Repository Here are the Git initializing a repository commands:Commands Descriptiongit initInitializes a new Git repository in the current directory.git init Creates a new Git repository in the specified directory.git clone this Clones a repository from a remote server to your local machine.git clone --branch Clones a specific branch from a repository.Basic Git CommandsHere are some basic Git commands:Commands Descriptiongit add Adds a specific file to the staging area.git add . or git add --allAdds all modified and new files to the staging area.git statusShows the current state of your repository, including tracked and untracked files, modified files, and branch information.git status --ignoredDisplays ignored files in addition to the regular status output.git diffShows the changes between the working directory and the staging area (index).git diff Displays the differences between two commits.git diff --staged or git diff --cachedDisplays the changes between the staging area (index) and the last commit.git diff HEADDisplay the difference between the current directory and the last commitgit commitCreates a new commit with the changes in the staging area and opens the default text editor for adding a commit message.git commit -m "" or git commit --message ""Creates a new commit with the changes in the staging area and specifies the commit message inline.git commit -a or git commit --allCommits all modified and deleted files in the repository without explicitly using git add to stage the changes.git notes addCreates a new note and associates it with an object (commit, tag, etc.).git restore Restores the file in the working directory to its state in the last commit.git reset Moves the branch pointer to a specified commit, resetting the staging area and the working directory to match the specified commit.git reset --soft Moves the branch pointer to a specified commit, preserving the changes in the staging area and the working directory.git reset --hard Moves the branch pointer to a specified commit, discarding all changes in the staging area and the working directory, effectively resetting the repository to the specified commit.git rm Removes a file from both the working directory and the repository, staging the deletion.git mvMoves or renames a file or directory in your Git repository.Also, check: Basic Git Commands with ExamplesGit Commit (Updated Commands)Here are some of the updated commands for Git commit:Commands Descriptiongit commit -m "feat: message"Create a new commit in a Git repository with a specific message to indicate a new feature commit in the repository.git commit -m "fix: message"Create a new commit in a Git repository with a specific message to fix the bugs in codebasesgit commit -m "chore: message"Create a new commit in a Git repository with a specific message to show routine tasks or maintenance.git commit -m "refactor: message"Create a new commit in a Git repository with a specific message to change the code base and improve the structure.git commit -m "docs: message"Create a new commit in a Git repository with a specific message to change the documentation.git commit -m "style: message"Create a new commit in a Git repository with a specific message to change the styling and formatting of the codebase.git commit -m "test: message"Create a new commit in a Git repository with a specific message to indicate test-related changes.git commit -m "perf: message"Create a new commit in a Git repository with a specific message to indicate performance-related changes.git commit -m "ci: message"Create a new commit in a Git repository with a specific message to indicate the continuous integration (CI) system-related changes.git commit -m "build: message"Create a new commit in a Git repository with a specific message to indicate the changes related to the build process.git commit -m "revert: message"Create a new commit in a Git repository with a specific message to indicate the changes related to revert a previous commit.Branching and MergingHere are some Git branching and merging commands:Commands Descriptiongit branchLists all branches in the repository.git branch Creates a new branch with the specified name.git branch -d Deletes the specified branch.git branch -aLists all local and remote branches.git branch -rLists all remote branches.git checkout Switches to the specified branch.git checkout -b Creates a new branch and switches to it.git checkout -- Discards changes made to the specified file and revert it to the version in the last commit.git merge Merges the specified branch into the current branch.git logDisplays the commit history of the current branch.git log